

Rolling your own Object Persistence Framework

BorCon Asia Pacific 2001

Peter Hinrichsen
© TechInsite Pty. Ltd.



www.techinsite.com.au

The material in this document is copyright TechInsite Pty. Ltd.

23 Victoria Pde.	PO Box 429
Collingwood.	Abbotsford
VIC 3066	VIC 3067
Australia	Australia

The source code of the tiOPF is made available under the Mozilla Public License 1.1

Email: peter_hinrichsen@techinsite.com.au

Web: www.techinsite.com.au

Latest copy of this document: www.techinsite.com.au/documentation/

Download the tiOPF: www.techinsite.com.au/Download.htm

Join the tiOPF mailing list: www.techinsite.com.au/MailingList.htm

Introduction

The tools that come out of the box with Delphi can be used to build a database application very quickly. However, with every data access component you drop into your application, you have tied yourself more closely to that component's API. With every data aware control, you have coupled yourself to a particular database schema.

The alternative is to roll your own persistence framework. This is hard work and will cost hundreds of person hours of work. However, if the business case justifies the effort, the results will be stable, optimised, versatile and very cheap to maintain.

In this document, we will walk through the TechInsite object persistence framework and discuss how Patterns have been used to solve many of the design issues that arose. The topics include:

What is an object persistence framework OPF and how can it help me?

The layers of an object persistence framework and how Patterns can help

- the business object model (Composite)
- object to database mapping (Visitor and Template Method)
- dynamically swapping the persistence layer at runtime (Adaptor Pattern and runtime packages)

Many of the chapters in this document will be dedicated to discussing the core concepts of object to database mapping, and how these concepts are implemented in the tiOPF.

We will finish by designing and building a real life contact management application using the techniques developed through this document. We will start by specifying the system, then constructing the business object model. We will build a GUI and database connection layer using the components that come with the tiOPF and will finish up by modifying the application so it can seamlessly connect to a variety of database.

1. What is an object persistence framework?

In this chapter we look at what an object persistence framework (OPF) is and how it can help you build better business applications. We examine some of the problems inherent in applications that are built using the RAD (rapid application design) approach that Delphi encourages, and examine how an OPF can help reduce these problems.

We take a look at the design requirements of an OPF as specified by the Jedi-Obiwan project (an open source project to build the ultimate OPF for Delphi), and Scott Ambler (a respected writer on the subject of OPFs). We will contrast these requirements with the design of the TechInsite OPF and see how the addresses (or fails to address) these requirements.

2. The Visitor pattern framework

So, the aim of this chapter is to come up with a generic way of performing a family of related tasks on some of the elements in a list. The task we perform may be different depending on the internal state of each element. We may not perform any tasks at all, or we may perform multiple tasks on multiple list elements. We shall use GoF's Visitor pattern to solve this problem and will see how the Visitor can be used to save objects to a variety of text file formats.

3. The Visitor and SQL Databases

Most business applications use a relational database to save their data, so in this chapter we will extend the Visitor framework developed in chapter 2 to store our objects in an Interbase database. We shall extend the Visitor framework with the Template Method pattern to achieve our goal of persisting objects to a relational database.

4. Building an abstract business object model with the Composite pattern

In chapters two and three, we started to develop an abstract collection class, and abstract business object class and we will extend these classes in this chapter by adding more of the functionality that will be required in a complex business system.

5. Installing the tiOPF

In this chapter we will see how to download and install the tiOPF. We will install the GUI components into the component pallet, setup the necessary search paths, compile the support applications and get the demo application running. We will also look at the directory structure and files that come with the framework.

6. A worked example of using the tiOPF

In this chapter we use the framework that can be downloaded as part of the tiOPF to build a real life, contact manager application. We will use UML to specify the business object model, then design a GUI using the TechInsite TPersistent aware controls. We will finish up by looking at three strategies for mapping objects to a database using the persistence manager that is part of the framework.

7. Using the Adaptor Pattern for database independence

The tiOPF allows you to connect to different databases, using different database APIs by simply changing a command line switch. This is achieved by using a wrapper around the database connection components like TDatabase and TQuery, then using a factory to construct the appropriate concrete type of component. This technique is based on the Adaptor pattern and is discussed in this chapter.

8. Using Report Builder with tiOPF

Here, we will show how to utilise Digital Metaphor's Report Builder package with tiOPF.

9. References and further reading

This chapter contains the URLs for the latest copy of this document, the source code for the tiOPF, as well as the articles and papers that have been used as a basis for the tiOPF. The user group, mailing lists and tools that helped in the development of the tiOPF are also listed.

Table of Contents

Introduction	3
Topics	4
1. What is an object persistence framework?	4
2. The Visitor pattern framework	4
3. The Visitor and SQL Databases	4
4. Building an abstract business object model with the Composite pattern	4
5. Installing the tiOPF	4
6. A worked example of using the tiOPF	4
7. Using the Adaptor Pattern for database independence	4
8. Using Report Builder with tiOPF	4
9. References and further reading	5
Table of Contents	5
What is an Object Persistence Framework?	8
Introduction	8
RAD vs OPF	8
Three ways to build an application	8
Four problems with RAD business applications	9
Summary: RAD vs. OPF	10
Some technical requirements of an OPF	10
Architecture of the tiOPF	16
Implementing the Visitor Framework	20
The aims of this chapter	20
Prerequisites	20
The business problem we will work with as an example	20
Setting up the demo	21
Summary	36
The next session	36
The Visitor Framework and SQL Databases	37
The aims of this chapter	37
Prerequisites	37
The business problem we will work with as an example	37
Implementation	37
Summary	65
The next chapter	65
Building an abstract business object model (BOM) with the Composite Pattern	66
The aims of this chapter	66
Prerequisites	66
The business problem we will work with as an example	66
The Composite pattern – what GoF say	67
Implementing the abstract business object model	67
Adding some helper methods	73
Summary	73
The next chapter	73
Installing the tiOPF	73
The aims of this chapter	73
Prerequisites	73
Contributors	73
Installing the tiOPF	73
tiOPF directories and files	73
A worked example of using the tiOPF	73

The aims of this chapter.....	73
Prerequisites	73
Application design.....	73
Application construction.....	73
Summary	73
The next chapter	73
Using the Adaptor Pattern for database independence	73
Authors note	73
Introduction	73
How Binding to a Vendor's API Can Bite You.....	73
Software House & Corporate IT Departments Share the Problem	73
How this Implementation of the Adaptor Evolved.....	73
What the GoF Book Says About The Adaptor	73
Class Adaptor	73
Object Adaptors.....	73
Adapting ZLib	73
Creating a Concrete Instance of the Adaptor.....	73
Creating From a Class Reference	73
Creating from a Factory.....	73
Adapting Data Access Components	73
Summary	73
Using Report Builder with the tiOPF	73
Const	73
Var	73
Begin.....	73
Begin.....	73
Var.....	73
lListColumn : TtiListColumn;.....	73
lColID : Integer;.....	73
Begin.....	73
References and further reading.....	73
Download the source	73
Scott Ambler.....	73
The Gang of Four	73
The Jedi-Obiwan project	73
Australian Delphi User Group (ADUG).....	73
Melbourne Pattern Group	73

Chapter #1

What is an Object Persistence Framework?

Introduction

In this chapter we look at what an object persistence framework (OPF) is and how it can help you build better business applications. We examine some of the problems inherent in applications that are built using the RAD (rapid application design) approach that Delphi encourages, and examine how an OPF can help reduce these problems.

We take a look at the design requirements of an OPF as specified by the Jedi-Obiwan project (an open source project to build the ultimate OPF for Delphi), and Scott Ambler (a respected writer on the subject of OPFs). We will contrast these requirements with the design of the TechInsite OPF and see how the framework addresses (or fails to address) these requirements.

The next section of this chapter discusses some of the problems with using Delphi's RAD approach to build complex business applications. We then have a high level look at some of the requirements of an OPF and see who the current version of the tiOPF meets (or fails to meet) these requirements.

RAD vs OPF

(Rapid Application Design vs Object Persistence Framework) There is no question that the tools that come with Delphi can be used to build a database application very, very quickly. The power of the combination of the BDE Alias, TDatabase, TQuery, TDataSource and TDBEdit is incredible. The problem 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 that particular field name in the database.

The alternative is to roll your own persistence layer. This is hard work and will cost you hundreds of hours of work 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, versatile and extremely satisfying to build.

Three ways to build an application

Consider this statement: There are three possible ways to build an application:

RAD – Rapid Application Design. The process of dropping data access components onto a TDataModule, and hooking them up to data aware controls like the TDBGrid and TDBEdit at design time. Good for simple applications, but if used for complex programs, can lead to an unmaintainable mess of spaghetti code.

OPF – Object Persistence Framework. Design a business object model using UML modeling techniques, then implement the model by descending business classes from your own abstract business object and abstract business object list. Write a family of controls to make building GUIs easier, then design some mechanism for saving these objects to a database, and reading them back again.

Hybrid – Using RAD techniques, but adding a middle layer. Continue using the best of what RAD can offer (the huge selection of GUI controls that are available), but implement some sort of middle layer using Delphi's components as they come out of the box. The TClientDataSet that comes with Delphi 5 Enterprise is a prime candidate as the starting point for this middle layer.

The rest of this chapter discusses the pros and cons of RAD and OPF, then the rest of this document discusses the implementation of an OPF. A chapter on using the hybrid approach would be worth while and is on my personal to do list.

Four problems with RAD business applications

A couple of month ago there was a discussion on the Australian Delphi User Group's (www.adug.org.au) mailing list on the topic of developing your own persistence framework, versus using RAD and data aware controls. Many posts to the mailing list were made and everyone seemed to have a strong opinion one way or the other. Delphi developers either love RAD and hate OPF, or hate RAD or love OPF. Several argued that the hybrid approach was worth a look but the argument was generally polarised. As the discussion on the list continued, most agreed that there was a place for RAD and data aware controls in single tier file based applications, or client server prototypes. Most of the experienced client/server developers who contributed to the discussion agreed that there was no room for RAD and data aware controls in sophisticated client server applications.

Here are four problems with RAD and data aware controls.

1. **Tight coupling to the database design.** One of the biggest problems with using RAD and data aware controls is that the database layer is very tightly coupled to the presentation layer. Any change to the database potentially means that changes must be made in many places throughout the application. It is often hard to find where these changes must be made as the link 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 can not be found with Delphi's search facility. When data aware controls are used, the amount of checking by the compiler of your code is reduced. This means that some bugs may not be detected until run time, 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 a data object's values by property name rather than by using a DataSet's FieldByName method. This gives greater compile time checking of the application and leads to simplified debugging.
2. **RAD and 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 (C:\Program Files\Borland\Delphi4\Bin\sqlmon.exe) and to watch the excess network traffic they create.
3. **Tight coupling to vendor specific database features.** At the simplest level, all 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 some 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 database seamlessly.

4. **Tight coupling to a data access API.** The BDE allows us to swap aliases when you want to change databases, but what if you want to switch from BDE data access, to ADO, IBOjects, Direct Oracle Access (DOA), ClientDataSet or our own 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.

Summary: RAD vs. OPF

	Advantages	Disadvantages	When do I use?
Data aware controls	<ul style="list-style-type: none"> • 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. 	<ul style="list-style-type: none"> • Higher maintenance and debugging costs. • Higher network traffic. • Limited number of data aware controls in Delphi (but plenty if you use InfoPower or other libraries) • Can not be used to edit custom file formats, registry entries or data that is not contained in a TDataSet. • Harder to develop your own data aware controls than regular controls. • Difficult to make work when the database does not map directly into the GUI ie, a well normalised database. • Difficult to make extensive reuse of code. 	<ul style="list-style-type: none"> • Low end customers (small businesses with few user). • 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	<ul style="list-style-type: none"> • Good for complex applications. • Lower network traffic. • Lower total cost of ownership. • When the database is storing non text data like multi-media, or perhaps scientific data which must be manipulated with complex algorithms. • Decouple GUI from database. 	<ul style="list-style-type: none"> • 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. 	<ul style="list-style-type: none"> • High end (corporate) customers with many users where performance is important. • Systems that have 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.

Some technical requirements of an OPF

Source of information

These design notes have been taken from two sources:

- The Jedi-Obiwan project, to build an open source object persistence framework in Delphi; and
- Scott Ambler's writings on object persistence frameworks.

These sources combine to give a much more comprehensive overview of the requirements of an OPF than I could write. In each case, the design requirement is listed (in normal font), then some notes on how the tiOPF addresses (or fails to address) this requirement is made in *Italics*.

How the tiOPF compares with Jedi-Obiwan OPF requirements

The specification, along with other design and discussion documents, and a mailing list for the Jedi-Obiwan project can be joined at <mailto:jedi-obiwan-subscribe@yahoogroups.com>

1. **Object Persistence.** The framework must allow for the storage and retrieval of data as objects. The framework must support the storage and retrieval of complex objects, including the relationships - e.g. inheritance, aggregation and association - between different objects.

The storage and retrieval of data as objects, with their relationships such as inheritance, aggregations and association is well met.

2. **Object Querying.** The framework must provide a mechanism for querying the object storage and retrieving collections of objects based on defined criteria. The framework must support a standard object querying language – e.g. Object Constraint Language (OCL) [5] or Object Query Language (OQL) [6].

Querying is supported at two levels. A group of objects can be retrieved from a database using what ever query language the database used (eg SQL). Once a list of objects has been read, there are methods on the abstract list class which allow filtering and querying. This filtering and querying can be achieved in two ways. A query class can be created to scan for matches; or a query can be run using RTTI to extract a property value by name and compare it to a passed parameter.

There is no OCL beyond what is described above.

3. **Object Identity.** All objects persisted within the framework must be uniquely identified using an Object Identifier (OID). OIDs must be supported within legacy environments and as such the OID mechanism must be sufficiently flexible enough to used to identify data objects even when the underlying storage mechanism doesn't explicitly support such identification.

The abstract base persistent object has a property, OID of type TOID. In the framework, this is an Int64, which means the second part of this requirement is not met. An OID factory would have to be introduced. This suggests all objects must support the IOID interface, or descend from a parent class with OID as a property.

4. **Transactions.** The framework must support transactions, satisfying the ACID Test set out by the Object Management Group [<http://www.omg.org>]. A transaction should be Atomic (i.e. it either happens completely or doesn't happen at all), its result should be Consistent, Isolated (independent of other transactions) and Durable (its effect should be permanent).

To this end the framework must support the same sort of object transaction operations as the

operations found in typical SQL databases implementations, e.g. Commit, Rollback, etc.

The framework supports transactions as provided by a SQL database. For example, you can make some changes to a group of objects and attempt to save them to the database. If the database commit is successful, the objects' state will be set from dirty, back to clean. If the commit is not successful, the objects' state will not be changed so further editing can take place. This is a kind of two phase commit.

There is no support for transactions beyond what is provided by the SQL database being used for the persistence.

5. **XML Data Sources.** The framework must be able to persist data object in repositories other than SQL databases. In particular the framework must support the storage and retrieval of data objects from XML files.

This feature is one of the next two deliverables. There will be a XML parser factory so the developer can choose between MSDOM, or what ever.

6. **Legacy Data.** The framework must provide mechanisms for converting data stored in legacy databases into data objects useable by Delphi applications, as well as be able to store data objects within legacy systems.

The framework may also provide mechanisms for persisting legacy business objects – that is, business objects that predate the existence of the framework. In this case the framework may require some modification of existing business objects but this must not be onerous.

Support for legacy databases is currently not good. Assumptions have been made about OID and the need to interact with a database using SQL. These issues can however, be addressed within the existing framework. ie, there are no dead ends that will prevent this happening.

7. **Heterogeneous Data Sources.** The framework must be able to work with data objects from a variety of data sources at run time. The representation of business objects in repositories must be independent from the business objects themselves, in that an object in one data repository stored in one repository may be stored in an alternative repository without any changes apparent to the object from the perspective of the application using it.

The framework must be able to work with data objects of the same class existing in separate repositories. The framework should also enable the association of data from heterogeneous sources, i.e. data from a variety of legacy or new database systems.

Currently this is not possible. A persistence layer flavor is loaded at startup time by dynamically loading a Delphi package. The need to change this so multiple persistent flavors can be loaded at the same time is one of the next two deliverables.

8. **Reporting Tools.** The framework must support conventional reporting technologies, either by providing supporting classes for common reporting tools (e.g. ReportBuilder) or by storing data using database schemas that are readily accessible by reporting tools.

Export of a collection of objects to HTML, XML and CSV has been provided, but there is currently no link to common reporting tools like ReportBuilder. (this is based on the parent of the business objects being a TPersistent, and used RTTI to write out all the published properties. Sorry, I know there are those who regard the use of RTTI as an outrageous hack – but it works.)

9. **Versioning and Version Migration.** The framework may support the versioning of objects, that is to say: the framework may allow for different versions of the same class of business object to be maintained within the same repository.

The framework must support the migration of data when class interfaces change (e.g. when a new version of an application is released). The framework must support the process of modifying existing data to fit updated business object metadata.

Not sure what is meant here. Perhaps:
`PerMgr.Read(ObjOldVersion) ;`
`ObjNewVersion.Assign(ObjOldVersion) ;`
`PerMgr.Save(ObjNewVersion) ;`

10. **Performance Optimizations.** The framework must be able to be optimized. That is, in performance critical environments where the knowledge of a skilled DBA is required for an application to perform successfully, repository-based optimizations must be available to applications using the framework.

This requirement is well met. The developer can either use the auto generated SQL framework, or can code the SQL himself. This can become quite complex as managing the persistence of a single object takes four SQL statements (Create, Read, Update, Delete), and four persistence classes to handle the mapping between the object to be saved. This is simplified by using a tool called the `tiSQLManager`, which is used to store the SQL in the database. A family of classes is written to interact with this SQL that has been abstracted out of the application.

11. **Separation of Business Logic and User Interface.** The framework must separate user-interface and business logic code, clearly defining the points of integration between the two. Its design must take on a “layered” approach, where high-level complex functionality builds upon simpler, lower level services.

Business logic, presentation logic and the persistence layer are strictly separated. For example, if we have a `TCustomer` class, there will probably be a unit called `Customer_BOM.pas` (for business logic), `Customer_Cli.pas` (for client side, presentation logic) and `Customer_Srv.pas` (for persistence logic).

12. **Separation of Object Data and Object Metadata.** Object metadata must be stored within the framework (i.e. within a supported repository) and not encapsulated within the business objects themselves.

There has been little attempt to separate meta data. Extensive use is made of RTTI if meta data is required. There has been one situation where I required automatically generated SQL, so there are a family of classes which map classes to tables and properties to fields.

13. **Object Data Standard Compliance.** Developers of the framework may elect to design the framework to meet the compliance requirements of the ODMG’s Object Data Standard 3.0 [6]. It is possible that ODS compliance may be implemented as an extension to the framework, but neither this, nor compliancy in general has been decided.

No

14. **Operating Environment.** The framework must support persistence under a variety of deployment models, specifically Standalone, Client-Server and n-Tier.

Yes

15. **Database Schemas / Multi-vendor Databases.** The nature of the framework must allow for any number of database schemas to be used and so must be extensible enough to allow end-users (Delphi developers) to extend the system to match their own persistence requirements. Default schemas may be defined for illustrative, testing or typical persistence requirements but should not be considered the only schemas usable within the framework.

While arguments for a “pure” (i.e. database-vendor neutral) persistence solution carry weight, the performance cost and legacy-data issues outweigh any gains enjoyed from having a single, simple implementation. Given that the framework must be extensible by third parties there is nothing preventing the implementation of a vendor-neutral extension to the framework, along with the vendor-specific implementations that will also be developed.

Yes. Currently we have Oracle via DOA, Paradox via BDE, Interbase via IBOjects, and multi tier

via HTTP and a custom ISAPI DLL. To write another persistence flavor, clone the code from one of the above, modify and recompile into a new package which gets loaded at startup.

How the tiOPF compares with Scott Ambler's requirements

The following 14 requirements were lifted from his paper on object persistence frameworks: 'The design of a robust persistence layer for relational databases' which can be found in full at <http://www.ambysoft.com/persistenceLayer.pdf>

1. **Several types of persistence mechanism.** A persistence mechanism is any technology that can be used to permanently store objects for later update, retrieval and/or deletion. Possible persistence mechanisms include flat files, relational databases, object-relational databases, etc. Scott's paper concentrates on the relational aspect of persistence layers.

This requirement is partially met. The tiOPF is optimised for object RDBMS mapping, and is currently being extended to support flat file and XML persistence layers. This, however is not a trivial task because in the process of optimising the framework for OO-RDBMS mapping, we have boxed our selves into a design corner which is making it difficult to create a persistence layer that maps to a not SQL database.

2. **Full encapsulation of the persistence mechanism(s).** Ideally you should only have to send the messages save, delete and retrieve to an object to save it, delete it or retrieve it respectively. That's it, the persistence layer takes care of the rest. Furthermore, except for well-justified exceptions, you shouldn't have to write any special persistence code other than of the persistence layer itself.

This requirement is well met, although you do not send a message like save or retrieve to the object, you pass the object to the persistence manager and ask it to handle the saving.

*For example, you do not call: `FMyObject.Save` ;
but rather `gPerMgr.Save (FMyObject)` ;*

3. **Multi-object actions.** Because it is common to retrieve several objects at once, perhaps for a report or as the result of a customised search, a robust persistence layer must be able to support the retrieval of many objects simultaneously. The same can be said of deleting objects from the persistence mechanism that meet specific criteria.

This requirement is well met. You can make a call like `gPerMgr.Read (FPeople)` where `FPeople` is an empty list which will hold 0..n `TPerson (s)`

You can also make calls like `FPeople.Delete` which will mark all the `TPerson (s)` in the list for deletion. When `gPerMgr.Save (FPeople)` is called, all the `TPerson (s)` marked for deletion will be removed from the persistence store.

4. **Transactions.** Related to requirement #3 is the support for transactions, a collection of actions on several objects. A transaction could be made up of any combination of saving, retrieving, and/or deleting of objects. Transactions may be flat, an 'all-or-nothing' approach where all the actions must either succeed or be rolled back (cancelled), or they may be nested, an approach where a transaction is made up of other transactions which are committed and not rolled back if the last transaction fails. Transactions also be short-lived, running in thousandths of a second, or long-lived, taking hours, days, or weeks, or even months to complete.

This requirement is partially met. Transactions are supported if the persistence mechanism supports them. (ie a RDBMS). A single transaction will be supported per call to the persistence layer. For example, all objects being saved in a call to `gPerMgr.Save(FPeople)` will be saved in the same transaction. If one fails to save, none will be saved. The abstract business object has a property `ObjectState` which indicates if an object is clean (it does not need to be saved) or dirty, (it must be created, updated or deleted) A single transaction can exist between the objects and the database. There is no support for inter object transactions, or object-GUI transactions.

5. **Extensibility.** You should be able to add new classes to your object application and be able to change persistence mechanisms easily (you can count on at least upgrading your persistence mechanism over time, if not port to one from a different vendor). In other words your persistence layer must be flexible enough to allow your application programmers and persistence mechanism administrators to each do what they need to do.

Not really sure what Scott is getting at here. It is possible to add new objects to the application (I can't see when it would never be possible).

The persistence connection mechanism is wrapped up in a Delphi package which is loaded when the application initialises so changing from one database access API is as easy as loading a different package (BPL)

6. **Object identifiers.** An object identifier (Ambler, 1998c), or OID for short, is an attribute, typically a number that uniquely identifies an object. OIDs are the object-oriented equivalent of keys from relational theory, columns that uniquely identify a row within a table.

Scott's high-low long integer based OIDs are implemented. There is no support for non integer OIDs and this should probably be a requirement as it makes it difficult to map to many legacy databases.

7. **Cursors.** A persistence layer that supports the ability to retrieve many objects with a single command should also support the ability to retrieve more than just objects. The issue is one of efficiency: Do you really want to allow users to retrieve every single person object stored in your persistence mechanism, perhaps millions all at once? Of course not. An interesting concept from the relational world is that of a cursor. A cursor is a logical connection to the persistence mechanism from which you can retrieve objects using a controlled approach, usually several at a time. This is often more efficient than returning hundreds or even thousands of objects all at once because the user may not need all of the objects immediately (perhaps they are scrolling through a list).

There is no support for cursors.

8. **Proxies.** A complementary approach to cursors is that of a 'proxy'. A proxy is an object that represents other objects but does not incur the same overhead as the object that it represents. A proxy contains enough information for both the computer and the user to identify it and no more. For example, a proxy for a person object would contain its OID so that the application can identify it and the first name, last name and middle initial so that the user could recognise whom the proxy object represents. Proxies are commonly used when the results of a query are to be displayed in a list, from which the user will select only one or two. When users select the proxy object from the list the real object is retrieved automatically from the persistence mechanism, an object that is much later than the proxy. For example, the full person object may include an address and a picture of the person. By using proxies you don't need to bring all of this information across the network for every person in the list, only the information that the user actually wants.

The principle discussed here is implemented, but not by using proxies. The abstract business object has a property, ObjectState. One possible ObjectState is posPK, meaning persistent object state 'Primary Key' This means the OID and enough information for a human to navigate a list of the objects has been loaded.

9. **Records.** The vast majority of reporting tools available in the industry today expect to take collections of database records as input, not collections of objects. If your organisation is using such a tool for creating reports within an object-oriented application your persistence layer should support the ability to simply return records as the result of retrieval requests in order to avoid the overhead of converting the database records to objects and then back to records.

There is no support for records, although there is a record like family of classes. The easy alternative to this would be to hook into a TClientDataSet, however this was not done to avoid building a dependency on the Enterprise version of Delphi 5.

10. **Multiple architectures.** As organisations move from centralised mainframe architectures to 2-tier client/server architectures to n-tier architectures to distributed objects your persistence layer should be able to support these various approaches. The point to be made is that you must assume that at some point your persistence layer will need to exist in a range of potentially complex environments.

This requirement has been moderately well met. The framework is currently in use with a Win32 client, talking to an IIS web server, running an ISAPI DLL. The data is transported via XML and HTTP.

11. **Various database versions and/or vendors.** Upgrades happen, as do ports to other persistence mechanisms. A persistence layer should support the ability to easily change persistence mechanisms without affecting the applications that access them, therefore a wide variety of database versions and vendors, should be supported by the persistence layer.

This requirement is well met. To connect to a different database or using a different connection API, just load an alternative Delphi package at runtime.

12. **Multiple connections.** Most organisations have more than one persistence mechanism, often from different vendors, that need to be accessed by a single object application. The implication is that a persistence layer should be able to support multiple, simultaneous connections to each applications persistence mechanism. Even something as simple as copying an object from one persistence mechanism to another, perhaps from a centralised relational database to a local relational database, requires at least two simultaneous connections, one to each database.

Multiple connections to a single database (via database connection pooling), or multiple connections to multiple databases of the same access are possible. Work has been commenced to allow multiple connections to different database types and it will not be too difficult to meet this requirement.

13. **Native and non-native drivers.** There are several different strategies for accessing a relational database, and a good persistence layer will support the most common ones. Connection strategies include using Open Database Connectivity (ODBC), Active Data Objects (ADO), and native drivers supplied by the database vendor and/or third party vendor.

This requirement is well met by swapping runtime packages.

14. **Structured query language (SQL) queries.** Writing SQL queries in your object-oriented code is a fragrant violation of encapsulation – you've coupled your application directly to the database schema. However, for performance reasons you sometimes need to do so. Hard-coded SQL in your application should be the exceptions, not the norm, an exception that should be well-justified before being allowed to occur. The persistence layer will need to support the ability do directly submit SQL code to a relational database.

Several ways of submitting SQL to the database are possible.

The default is for the SQL to be maintained with a tool called the tiSQLManager that stores the SQL in the database. This has many advantages but it does mean the three tiSQLManager tables must exist in the database.

SQL can be generated on the fly (this requires work before it could be regarded as production quality)

SQL can be hard-coded into the application.

Architecture of the tiOPF

The three layers of the tiOPF

The TechInsite Object Persistence Framework is based on a three layer architecture:

The business object model (BOM). The BOM is where the objects that model the business system, and their internal data structures are defined. The relationships these objects have with each other are also modeled, along with some basic business rules like ‘A person can have 0 to many addresses.’ The design of the abstract business object model is discussed in detail in chapter 4.

The presentation layer. The presentation layer is where the user interacts with the BOM. It can be a Windows GUI, a HTML page or an interface into another business system. These possible configurations are shown in the diagrams below. The GUI controls that come with the tiOPF are documented by way of demonstrations that are included with the source. These controls are also used in the worked example in chapter 6.

The persistence layer. The persistence layer is where it is made easy to save objects to a database. One of the principle aims of the persistence layer is to make it possible to write code like `gTIPerMgr.Save(FMyComplexData) ;` and to know that all objects in FMyComplexData will be saved in the correct way, depending on whether the objects must be created in the database, updated or deleted. A good persistence layer will also make it easy to swap databases at application startup, or on the fly so data can be read from one persistence store, manipulated then saved to another persistent store. The persistence layer that is implemented in the tiOPF is based on GoF’s Visitor, Template Method and Adaptor patterns. This is discussed in chapters 2, 3 and 7.

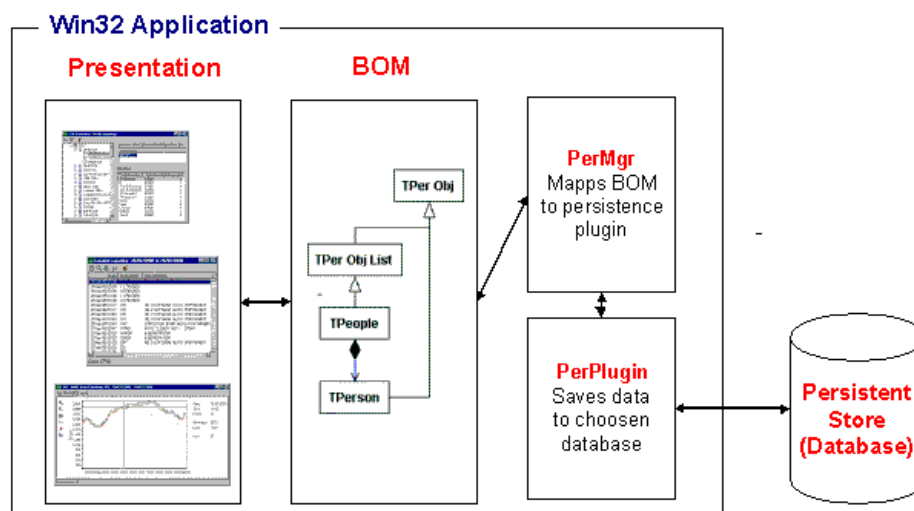
Possible configurations of the tiOPF

There are several possible configurations of the tiOPF, and four of them are discussed below. These include configurations for:

- Two-tier, client server application built using Win32 GUI.
- Multi tier application build using Win32 GUI and HTTP / XML as the data carrier
- Browser based HTML client with server side business objects.
- System to System application with no client for human interaction

Two-tier, client server application built using Win32 GUI.

The diagram below shows the tiOPF configured for a ‘conventional’ two-tier or client-server application. There are three layers within the application that are compiled into a single application. The presentation layer and business object model are in the same physical executable, and the persistence manager and persistence plugin can be in the same executable, or could be deployed as separate ‘Packages’ or DLLs. This is the most common way of building a tiOPF application, and is how the demonstration that comes with the source is architected. This architecture will be used in chapter 6 when we look at a worked example of using the tiOPF.

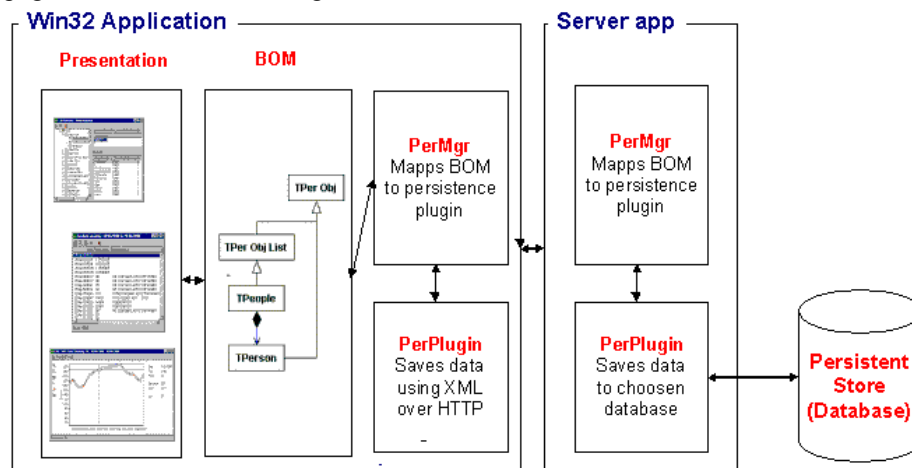


Multi tier application build using Win32 GUI and HTTP / XML as the data carrier

The diagram below shows the tiOPF configured as a multi tier application with two instances of the persistence manager and persistence plugin running. On the client, the presentation layer and business object model are running, with an instance of the persistence manager. There is a persistence plugin which knows how to talk to a web server (or web service) using HTTP and XML. The web service is running another instance of the persistence manager and a persistence plugin which allows it to talk to the database.

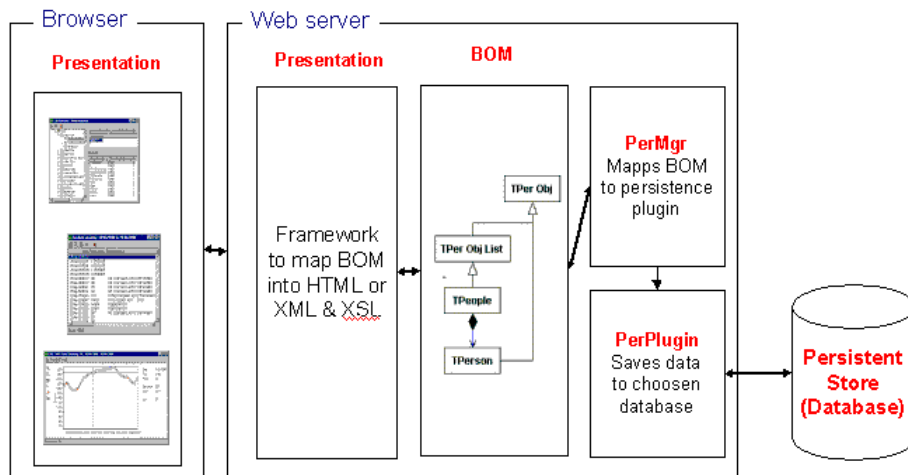
This approach uses a 'fat client' where significant intelligence is build into the client, and a thin server, when the server knows how to read and save objects, but does not implement any business rules. There are many disadvantages to this architecture such as having to manage the deployment of the relatively bulky client (this can be automated using the deployment manager application that comes with the source). There 'killer' advantage to this architecture however, is that the same windows client can be used in system that connects from outside a firewall via a web server, as is used to connect directly to the database behind the firewall. This is achieved by simply swapping the persistence plugin.

There are two applications included with the source called the tiDeployMgr and tiAppLauncher. The tiDeployMgr makes it possible to save an application, and associated binaries to a database along with file size, date and time and version information. When the tiAppLauncher connects to the database, it version checks the files deployed on your local hard drive and updates them as necessary. It then launches the application that will be using XML and HTTP as the communication back to the database via a web server. There is an ActiveX version of the tiAppLauncher that can be imbedded in a HTML page for web based launching.



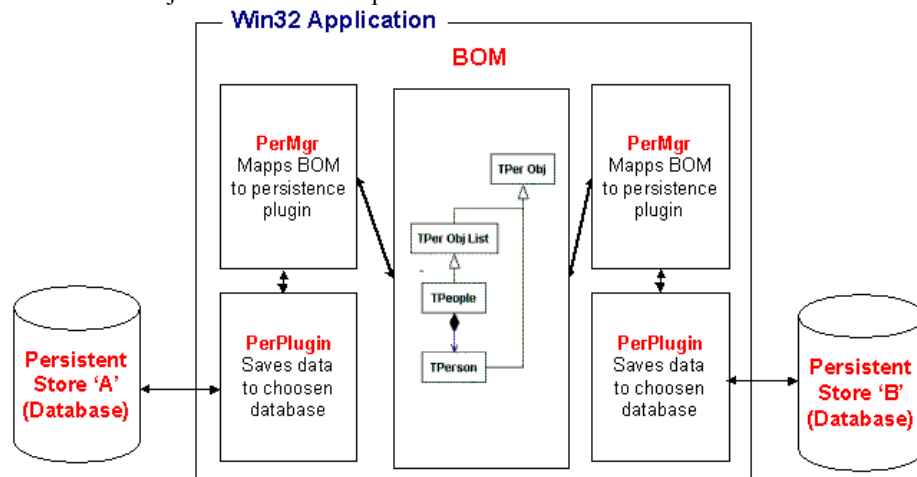
Browser based HTML client with server side business objects.

The diagram below shows the framework reconfigured to run with the business object model, persistence manager and persistence plugin running under a web server. The application produces HTML pages which are viewed inside a web browser. I have not used the framework in this way, but it is, in theory possible.



System to System application with no client for human interaction

The last diagram shows the tiOPF configured as an interface between two disparate business systems. This is a technique I have used regularly for moving data from a legacy system via, say a CSV or XML file. A business object model is designed to represent the data, then two persistence plugins are deployed, one on the legacy database side and one on the new database side. This architecture becomes especially useful when a Windows GUI is also required for manipulating the data and can reuse both the business object model and the persistence code.



Chapter #2

Implementing the Visitor Framework

The aims of this chapter

The aim of this chapter is to introduce the Visitor Pattern as one of the core concepts of the TechInsite Object Persistence Framework (tiOPF). We shall take a detailed look at the problem we are aiming to solve, then investigate some alternative solutions it before introducing the Visitor. As we are developing our Visitor framework, we shall come across another problem that will need our attentions: How do we iterate over a collection of objects in a generic way? This shall be studied too.

So, the aim of this chapter is to come up with a generic way of performing a family of related tasks on some of the elements in a list. The task we perform may be different depending on the internal state of each element. We may not perform any tasks at all, or we may perform multiple tasks on multiple list elements.

Prerequisites

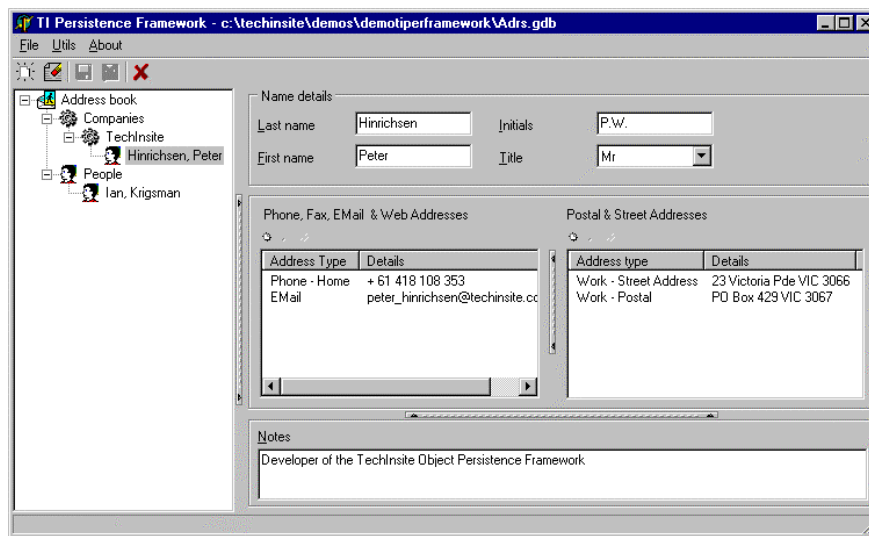
The reader should have a good understanding of Delphi and be clear on the concepts of object oriented programming with Object Pascal. The previous Chapter 'Chapter #1 What is an OPF?' will help understand some of the concepts of an object persistence framework.

The business problem we will work with as an example

As an example, we will construct an address book application that enables us to store peoples' names and contact details. With the increase in the number of different ways we can contact a person, our application must be flexible enough to cater for new addresses types without the need for any re-engineering. (I have memories of having to add a Phone_Mobile column to an application, only to complete that batch of work to be asked to add an EMail column.). We need to allow for two types of addresses: regular addresses like home, work or postal and e-addresses like phone, fax, mobile, EMail, Web.

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

The main form of our application is shown below.



A right mouse click on the tree will let you add or delete a person or company. 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.

The data shall be stored in a variety of formats, and we will look at how to do this with the help of the Adaptor Pattern as described in chapter 7.

Setting up the demo

We shall start work with a simple collection of objects as an example. We will create a list of people, the people will have two properties: Name and EMailAdrs. To start off, the people will be added to the list in the list's constructor (then we shall progressively read them from a text file, then Interbase database). This is an over simplified example, but it is sufficient to use when discussing the problems that the Visitor pattern solve.

Create a new application, and add two classes in the main form's interface section: TPeople, which descends from the TObjectList and TPerson which descends from TObject. The interface section will look like this:

```
TPeople = class( TObjectList )
public
  constructor Create ;
end ;

TPerson = class( TObject )
private
  FEMailAdrs: string;
  FName: string;
public
  property Name : string read FName write FName ;
  property EMailAdrs : string read FEMailAdrs write FEMailAdrs ;
end ;
```

In the constructor of the list, we shall create three instances of TPerson like this:

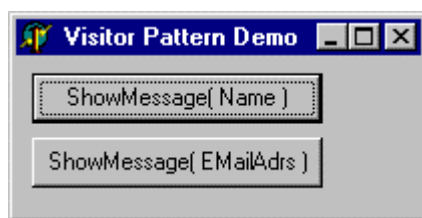
```
constructor TPeople.Create;
var
  lData : TPerson ;
begin
  inherited ;
  lData := TPerson.Create ;
  lData.Name := 'Malcolm Groves' ;
```

```
lData.EmailAdrs := 'malcolm@madrigal.com.au' ;           // (ADUG Vice President)
Add( lData ) ;

lData           := TPerson.Create ;
lData.Name      := 'Don MacRae' ;                       // (ADUG President)
lData.EmailAdrs := 'don@xpro.com.au' ;
Add( lData ) ;

lData           := TPerson.Create ;
lData.Name      := 'Peter Hinrichsen' ;                 // (Yours truly)
lData.EmailAdrs := 'peter hinrichsen@techinsite.com.au' ;
Add( lData ) ;
end;
```

To start off, we are going to iterate over the list and perform two operations on each list element. The operations will be similar, but not the same. In this dumb, over simplified example we will call ShowMessage() on each TPerson's Name and EmailAdrs properties. To start this off, add two buttons to the application's main form: One to show each person's name, the other to show each person's EmailAdrs. The form will look like this:



Step #1. Hard code the iteration

To show each person's name, add the following code in the first buttons OnClick event.

```
procedure TFormMain.btnShowNamesClick(Sender: TObject);
var
  i : integer ;
begin
  for i := 0 to FPeople.Count - 1 do
    ShowMessage( TPerson( FPeople.Items[i] ).Name ) ;
  end;
```

and to show each persons Email address, add the following code to the second button's OnClick event.

```
procedure TFormMain.btnShowEMailsClick(Sender: TObject);
var
  i : integer ;
begin
  for i := 0 to FPeople.Count - 1 do
    ShowMessage( TPerson( FPeople.Items[i] ).EmailAdrs ) ;
  end;
```

(I did say it was over simplified. We will be writing data out to a text file, and saving it to a database soon.)

Now, there are several things I don't like about this code:

- We have two routines that do pretty much the same thing. The only difference is the property we are hitting in ShowMessage()
- Iterating over a list becomes very tedious when you have to write this code in hundreds of places in your application.

- Having to type cast each element of the list as a TPerson is error prone. What if a TAnimal found its way into the list? There is no mechanism to stop this happening, and no mechanism to protect us from errors if it does.

Next, we shall look at ways of improving this code by abstracting away the iteration logic into a parent class.

Step #2. Abstracting the iteration logic

We want to abstract the iteration logic into a parent class. The iteration logic is simple and looks like this:

```
for i := 0 to Flist.Count - 1 do
  // Do something here...
```

Sounds like we are building an instance of the Iterator Pattern. GoF tell about two kinds of Iterators: External Iterators and internal Iterators. The external Iterator was shown to us at last year's BorCon in Malcolm Groves (famous) presentation on writing solid code. We shall use an internal Iterator here because it makes it easier to iterate over the nodes of a tree, which is what we will be ultimately wanting to do. We will be adding a method called Iterate to our collection class. This method will be passed a procedural type parameter that defines a task to be performed on each node of the collection. We shall call this procedural type TDoSomethingToAPerson.

To do this, we firstly extend the interface of the demonstration with a forward declaration of TPerson (because TDoSomethingToAPerson references TPerson, and TPerson references TDoSomethingToAPerson). Next, create a procedure type called TDoSomethingToAPerson, which takes a single parameter of type TPerson. Procedural types allow you to treat procedures and functions as values that can be assigned to variables or passed to other procedures and functions. This way we can define a procedure to call ShowMessage (Person.Name), and one to call ShowMessage (Person.EmailAdrs), then pass these procedures to the generic iteration routine. We add the method DoSomething(pMethod : TDoSomethingToAPerson) to our list of TPeople. The finished interface section will look like this:

```
TPerson = class ; // Forward declaration of TPerson, required by
TDoSomethingToAPerson
  TDoSomethingToAPerson = procedure( const pData : TPerson ) of object ;

TPeople = class( TObjectList )
  public
    constructor Create ;
    procedure DoSomething( pMethod : TDoSomethingToAPerson ) ;
  end ;
```

The implementation of DoSomething() will look like this:

```
procedure TPeople.DoSomething(pMethod: TShowMethod);
var
  i : integer ;
begin
  for i := 0 to Count - 1 do
    pMethod( TPerson( Items[i] ) ) ;
  end;
```

Now, to perform an operation on each element in the list, we must do two things. Firstly define a method of type TDoSomethingToAPerson, and secondly call DoSomething(), passing the a pointer to our TDoSomethingToAPerson as a parameter. The code to do this is shown below:

```
// Somewhere in the application, we must define this method
procedure TFormMain.DoShowName( const pData : TPerson ) ;
```

```
begin
  ShowMessage( pData.Name ) ;
end;

// Then in the form, we implement the call like this
procedure TFormMain.btnMethodPointerShowNameClick(Sender: TObject);
begin
  FPeople.DoSomething( DoShowName ) ;
end;
```

This is progress. We have introduced three layers of abstraction. The generic iteration logic is contained in the list class. The business logic (implemented as ShowMessage) is contained in another part of the application, and the GUI has a single line call to kick off a process.

It is easy to imagine how we could replace the call to ShowMessage with a call to a TQuery instance that runs some SQL to save the data contained in the TPerson. The call to a TQuery might look like this:

```
procedure TFormMain.SavePerson( const pData : TPerson ) ;
var
  lQuery : TQuery ;
begin
  lQuery := TQuery.Create( nil ) ;
  try
    lQuery.SQL.Text := 'insert into people values ( :Name, :EMailAdrs )' ;
    lQuery.ParamByName( 'Name' ).AsString := pData.Name ;
    lQuery.ParamByName( 'EMailAdrs' ).AsString := pData.EMailAdrs ;
    lQuery.Database := gAppDatabase ;
    lQuery.ExecSQL ;
  finally
    lQuery.Free ;
  end ;
end;
```

This introduces the problem of maintaining state. We have connected the TQuery up to an application wide database called gAppDatabase. Where is this going to be maintained? Also, we will very quickly get tired of creating the TQuery, setting up the parameters, executing the query, then remembering to call free. This block of code would be better wrapped up in a class which inherits from an abstract that takes care of creating and freeing the TQuery, and other chores like wiring it up to a TDatabase.

Step #3. Instead of passing a method pointer, we will pass an object

Passing an object to our generic iterate method solves the problem of maintain state. We will call the object we pass a Visitor, and create an abstract Visitor class called TPersonVisitor with a single method Execute. The interface of our abstract visitor looks like this:

```
TPersonVisitor = class( TObject )
public
  procedure Execute( pPerson : TPerson ) ; virtual ; abstract ;
end ;
```

Next, we will add a method called Iterate to the TPeople list. The interface of TPeople now looks like this:

```
TPeople = class( TObjectList )
public
  procedure Iterate( pVisitor : TPersonVisitor ) ;
end ;
```

The implementation of TPeople.Iterate looks like this:


```
procedure TPeople.Iterate(pVisitor: TPersonVisitor);
var
  i : integer ;
begin
  for i := 0 to Count - 1 do
    pVisitor.Execute( TPerson( Items[i] ));
  end;
```

The Iterate method of TPeople is passed a concrete instance of TPersonVisitor. For each TPerson in the list, the execute method of the TPersonVisitor is called with the TPerson as a parameter.

We create concrete instances of TPersonVisitor called TShowNameVisitor and TShowEmailAdrsVistor that implement the specific behavior we require. This is shown below for TShowNameVisitor:

```
// Interface
TShowNameVisitor = class( TPersonVisitor )
public
  procedure Execute( pPerson : TPerson ) ; override ;
end ;

// Implementation
procedure TFormMain.btnVisitorNameClick(Sender: TObject);
var
  lVis : TPersonVisitor ;
begin
  lVis := TShowNameVisitor.Create ;
  try
    FPeople.Iterate( lVis ) ;
  finally
    lVis.Free ;
  end ;
end;
```

From this we can see that while we have solved one problem, we have created another. The iteration logic is abstracted away to the list class. We are using an object to define the special behavior which will allow us to maintain some state information, but we have blown the number of lines of code to implement this behavior from one as in

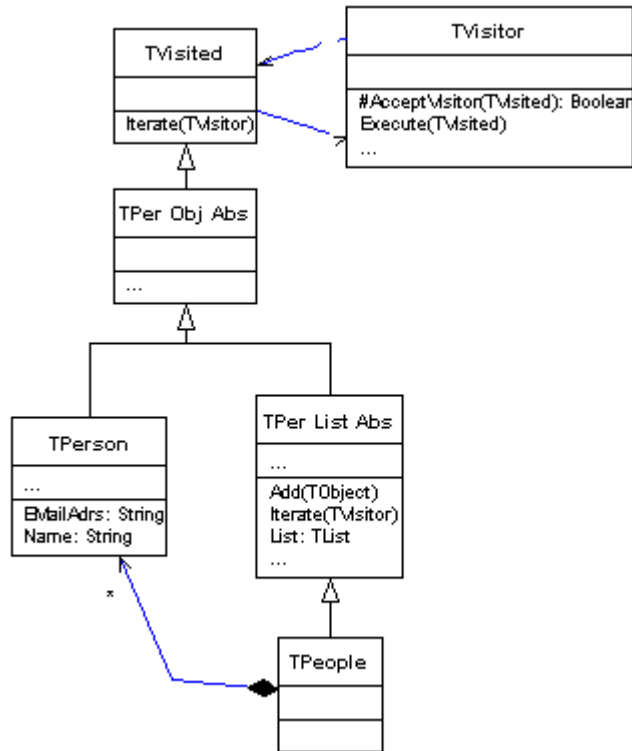
```
procedure TFormMain.btnMethodPointerShowNameClick(Sender: TObject);
begin
  FPeople.DoSomething( DoShowName ) ;
end;
```

to the nine lines in the preceding block of code. What we need now is a visitor manager to take care of the construction and destruction of our visitor classes. We will potentially have several per operation so the visitor manager will handle named lists of visitors and call each visitor in the list in turn against the elements in the list. This will become essential as we move towards using the Visitor to persist data to a relational database because a simple save may require three different SQL statements: CREATE, UPDATE and DELETE.

Step #4. Abstract the Visitor logic

Before we go any further, we must abstract the Visitor functionality away from the business objects so we never have to touch it again. We will do this in three stages. We will create abstract TVisitor and TVisited classes, then we will create an abstract business object, and business object list that will descend from TVisited. We will then re-factor our TPerson and TPeople classes to descend from the newly created abstract classes.

The class diagram of what we are aiming to build looks like this:



The TVisitor has two methods, AcceptVisitor() and Execute(). Both taking a single parameter of type TVisited. The TVisited has a single method called Iterate() which takes a single parameter of type TVisitor. TVisited.Iterate() calls the Execute method on the Visitor that is passed as a parameter against itself, and if it contains other object, against each one of the too. The function TVisitor.AcceptVisitor is necessary because we are building a generic framework. It will be possible to pass a visitor that is designed for handling TPeople a concrete instance of, say a TDog and we must have a mechanism for preventing this from causing an access violation. The TVisited descends from TPersistent because down the track, we will be implementing some functionality that requires RTTI. The interfaces of TVisitor and TVisited are shown below:

```

TVisitor = class( TObj )
protected
  function AcceptVisitor( pVisited : TVisited ) : boolean ; virtual ; abstract ;
public
  procedure Execute( pVisited : TVisited ) ; virtual ; abstract ;
end ; // Both AcceptVisitor and Execute must be implemented in the concrete

TVisited = class( TPersistent )
public
  procedure Iterate( pVisitor : TVisitor ) ; virtual ;
end ;
  
```

Both TVisitor.AcceptVisitor and TVisitor.Execute must be implemented in the concrete class. The implementation of TVisited.Iterate, which contains a call to TVisitor.Execute, is shown below:

```

procedure TVisited.Iterate(pVisitor: TVisitor);
begin
  pVisitor.Execute( self ) ;
end;
  
```

Step #5. Create an abstract business object, and abstract list object

We require two more abstract classes in our framework: An abstract business object, and a list container for the abstract business object. We shall call these TPerObjAbs and TPerVisList and the interface of these classes is shown below:

```
TPerObjAbs = class( TVisited )
private
public
    constructor Create ; virtual ;
end ;
```

We will be adding significantly to TPerObjAbs when we look in more detail at the business object framework, but for the time being, we just add a virtual constructor so we can uniformly override the constructor in descendent classes.

We want our list class, TPerVisList to descend from TVisited so the generic visitor behavior can be implemented (actually, we want it to descend from TPerObjAbs for reasons we will discuss later). Ideally, we would use interfaces to give our list class the iteration behavior, but much of this code base predates the popularity of interfaces, and I have not faced up to the task of re-factoring to take advantage the benefits they can offer.

To create a list class which descends from TVisited and TPerObjAbs, we shall use object containment. The interface of TPerVisList is shown below and the implementation is pretty much what you would expect.

```
TPerVisList = class( TPerObjAbs )
private
    FList : TObjectList ;
    function GetList: TList;
public
    constructor Create ; override ;
    destructor Destroy ; override ;
    property List : TList read GetList ;
    procedure Iterate( pVisitor : TVisitor ) ; override ;
    procedure Add( pData : TObject ) ;
end ;
```

The most important method in this class is the overridden procedure Iterate. In the abstract class TVisitor, iterate is implemented as the one line call `pVisitor.Execute(self)`. In the TPerVisList it is implemented like this:

```
procedure TPerListAbs.Iterate(pVisitor: TVisitor);
var
    i : integer ;
begin
    inherited Iterate( pVisitor ) ;
    for i := 0 to FList.Count - 1 do
        ( FList.Items[i] as TVisited ).Iterate( pVisitor ) ;
    end;
```

This is an important core concept. We now have two abstract business objects TPerObjAbs and TPerVisList. Both have an iterate method that is passed an instance of a TVisitor as a parameter. In each case, the TVisitor's Execute method is called with self as a parameter. This call is made via inherited at the top of the hierarchy. For the TPerVisList class, each object in the owned list also has its Iterate method called with the visitor begin passed as the parameter. This ensures that all objects in the hierarchy are touched by the Visitor.

Step #6. Create a Visitor manager

Now, back to the original problem we created for our selves in step #3. We don't want to be spending all our time creating and destroying visitors. The solution is in the Visitor Manager.

The Visitor Manager performs two main tasks: It maintains a list of registered visitors (visitors are registered in the implementation section of the unit where they are declared.); and calls a group of visitors that are registered with a given command name against the data object it is passed.

To implement the Visitor manager, we will define three more classes: The TVisitorClass, TVisitorMapping and the TVisitorMgr.

The TVisitorClass is a class reference type that will hold an instance TVisitor's class. I find the help text description of class references a little confusing. The best way to understand them is with an example. Lets say we have our abstract Visitor class TVisitor, and a Visitor class reference type TVisitorClass. We also have a concrete Visitor called TSaveVisitor. The TVisitorClass type is declared like this:

```
TVisitorClass = class of TVisitor ;
```

This lets us write code like this:

```
procedure ExecuteVisitor( const pData : TVisited ; const pVisClass : TVisitorClass ) ;
var
  lVisitor : TVisitor ;
begin
  lVisitor := pVisClass.Create ;
  try
    pData.Iterate( lVisitor ) ;
  finally
    lVisitor.Free ;
  end ;
end ;
```

We pass two parameters to this procedure; pData which is an instance of TVisited (like our TPeople), a TVisitorClass, which could be TShowNameVisitor or TShowEMailAdrsVisitor. This procedure takes care of the tedious business of creating the visitor, calling iterate, then freeing the visitor when done.

The second class we create for our visitor manager is called TVisitorMapping. It is a simple data structure to hold two pieces of information: a TVisitorClass and a string called Command. The interface of TVisitorMapping is shown below:

```
TVisitorMapping = class( TObject )
private
  FCommand: string;
  FVisitorClass: TVisitorClass;
public
  property VisitorClass : TVisitorClass read FVisitorClass write FVisitorClass ;
  property Command : string read FCommand write FCommand ;
end ;
```

The final class we create is the TVisitorMgr. When we register a Visitor with the Visitor Manager, an instance of TVisitorMapping is created and added to the list inside the TVisitorMgr. The command and VisitorClass properties are set which allows us to execute a group of visitors identified by a string. The interface of the TVisitorManager is shown below:

```
TVisitorMgr = class( TObject )
private
  FList : TObjectList ;
public
  constructor Create ;
  destructor Destroy ; override ;
  procedure RegisterVisitor( const pCommand : string ; pVisitorClass :
TVisitorClass ) ;
  procedure Execute( const pCommand : string ; pData : TVisited ) ;
end ;
```

The key methods here are RegisterVisitor and Execute. RegisterVisitor is called in the implementation section of the unit where the Visitor is defined and is typically called like this:

initialization

```
gVisitorMgr.RegisterVisitor( 'show', TShowNameVisitor ) ;  
gVisitorMgr.RegisterVisitor( 'show', TShowEMailAdrsVisitor ) ;
```

The implementation of RegisterVisitor is shown below (this code is much the same as the code found in a Delphi implementation of the Factory Pattern)

```
procedure TVisitorMgr.RegisterVisitor( const pCommand: string;  
                                     const pVisitorClass: TVisitorClass);  
var  
  lData : TVisitorMapping ;  
begin  
  lData := TVisitorMapping.Create ;  
  lData.Command := pCommand ;  
  lData.VisitorClass := pVisitorClass ;  
  FList.Add( lData ) ;  
end;
```

The other important method in the TVisitorMgr is Execute. Execute takes two parameters, the command name which identifies the family of visitors to be executed, and the data object which is at the top of the tree to be iterated over. The implementation of Execute is shown below:

```
procedure TVisitorMgr.Execute(const pCommand: string; const pData: TVisited);  
var  
  i : integer ;  
  lVisitor : TVisitor ;  
begin  
  for i := 0 to FList.Count - 1 do  
    if SameText( pCommand, TVisitorMapping( FList.Items[i] ).Command ) then  
      begin  
        lVisitor := TVisitorMapping( FList.Items[i] ).VisitorClass.Create ;  
        try  
          pData.Iterate( lVisitor ) ;  
        finally  
          lVisitor.Free ;  
        end ;  
      end ;  
    end ;  
  end ;  
end;
```

To execute both the ShowName and ShowEMailAdrs visitors (the ones we registered above), one after the other we would make the following call to the Visitor manager.

```
gVisitorMgr.Execute( 'show', FPeople ) ;
```

Next, we will create some persistent Visitors that will let us make calls like

```
// To read from a text file  
gVisitorMgr.Execute( 'read', FPeople ) ;  
  
// To save to a text file  
gVisitorMgr.Execute( 'save', FPeople ) ;
```

but first we will use the tiListView and tiPerAwareControls to create a GUI to use while editing the list of TPeople.

Step #7. Re-factor the BOM to descend from our abstract BO and BO list

We shall start by re-factoring our business objects TPeople and TPerson to descend from the abstract classes TPerVisList and TPerObjAbs. This means that our iteration logic will be wrapped up in the

parent class, never to be touched again. It also makes it possible to use the business objects with the visitor manager.

The new interface of the TPeople looks like this:

```
TPeople = class( TPerVisList ) ;
```

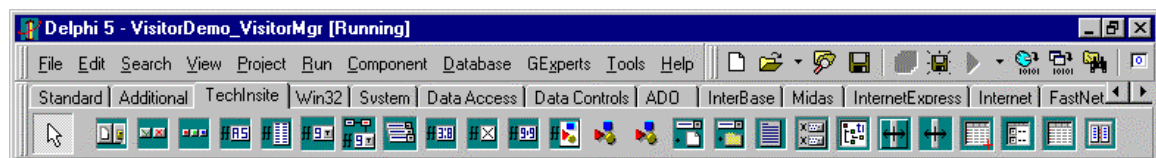
In fact, there is no code to implement in the class TPeople. In theory, we could do without the concrete class TPeople and store our TPerson objects in an instance of TPerVisList. We go to the trouble of creating the concrete class TPeople however because we use the line `result := Visited is TPeople` in the AcceptVisitor method. We need to know the concrete class of the collection so we can decide if we want to process it with a visitor.

The TPerson class now descends from TPerObjAbs We must also moved the persistent properties in the TPerson class from public to published. This allows us to use RTTI to present the data with the TPersistent aware controls, and to use the mapping framework we shall look at later to reduce the amount of code we must write to save the objects to a database. The interface of TPerson now looks like this:

```
TPerson = class( TPerObjAbs )
private
  FEmailAdrs: string;
  FName: string;
published
  property Name : string read FName write FName ;
  property EMailAdrs : string read FEmailAdrs write FEmailAdrs ;
end ;
```

Step #8. Create a GUI

Download and install the TechInsite persistence framework code from <http://www.techinsite.com.au/tiopf/download.htm> and following the instructions at www.techinsite.com.au/tiopf/gettingstarted.htm to install the TPersistent aware controls into your component pallet. When you have finished, your component pallet should look like this:

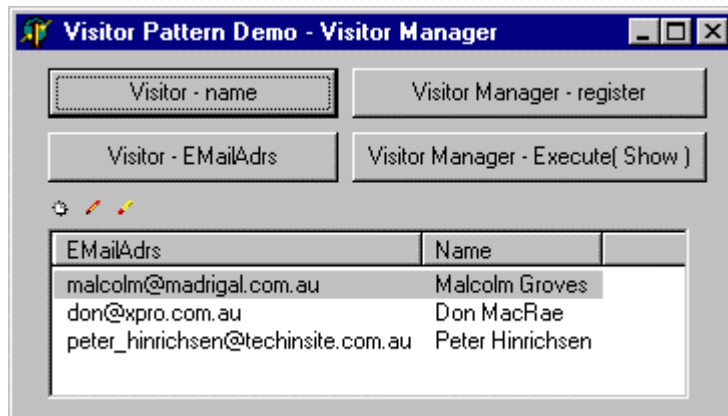


Drop a TtiListView on the application's main form then In the form's constructor create an instance of TPeople and store it to a variable FPeople. Assign the data property of the TtiListView like this:

```
procedure TFormMain_VisitorManager.FormCreate(Sender: TObject);
begin
  FPeople := TPeople.Create ;
  LV.Data := FPeople.List ;
end;
```

Select the TtiListView and in the object inspector change its name to LV. Set the VisibleButtons property to [tiLVBtnVisNew,tiLVBtnVisEdit,tiLVBtnVisDelete] then go to the Events tab of the Object Inspector where we will implement the OnEdit, OnInsert and OnDelete methods.

Run the application and you should have a form that looks like this:



(The TtiListView will automatically detect a TList of TPersistent descendants and display the published properties as columns in the list)

The three small buttons on the TtiListView fire events called OnEdit, OnInsert and OnDelete when clicked. An edit dialog will be implemented under the OnEdit and OnInsert events so we can play around with the data. The code in the OnEdit event would typically look like this:

```
procedure TFormMain_VisitorManager.LVItemEdit( pLV: TtiCustomListView;
                                               pData: TPersistent;
                                               pItem: TListItem);
begin
  TFormEditPerson.Execute( pData ) ;
end ;
```

The things to notice are that the OnEdit event on the TtiListView passes some useful values as parameters including the data object under the TListView, as well as the usual sender and TListItem properties. The edit dialog implements a class procedure called Execute that takes the data object to be edited as a parameter. The dialog we shall build uses the TechInsite TPersistent aware controls, and looks like this:



No rocket science here except that a one liner as calls the dialog described above. The implementation of the class method Execute() looks like this:

```
class procedure TFormEditPerson.Execute( pData: TPersistent);
var
  lForm : TFormEditPerson ;
begin
  lForm := TFormEditPerson.Create( nil ) ;
  try
    lForm.Data := pData ;
    lForm.ShowModal ;
  finally
    lForm.Free ;
  end ;
end;
```

This code can be implement in an abstract form, so we only have to write it once. The only code we do have to write for a dialog is contained in the SetData() method and looks like this:

```
procedure TFormEditPerson.SetData(const Value: TPersistent);
begin
  FData := Value;
  paeName.LinkToData( FData, 'Name' );
  paeEMailAdrs.LinkToData( FData, 'EMailAdrs' );
end;
```

Now that we have a basic form to browse a list of objects and to edit the objects, we can take a look at how to save the data to a text file.

Step #9 Create an abstract text file visitor

We shall create an abstract text file visitor, that knows how to read and write to a file. There are three ways we could do this: Using Delphi's file management routines (like AssignFile() and ReadLn()); using a TStringStream or TFileStream, or using a TStringList.

The file management routines are good, but have been around since the early days of Pascal and are a little dated (IMHO). The Stream approach is better because it is very object oriented and makes it easy to move data from one stream to another. Compression and encryption are also made a piece of cake, however reading from a TFileStream or TStringStream line by line is a chore. The TStringList gives us the LoadFromFile() and SaveToFile() methods which are easy to use, but rather slow on large files. We shall use the TStringList as the interface to a text file here because it is easy to use. I suggest looking at developing a TStream based solution for use in real life though.

The interface of our abstract file visitor looks like this:

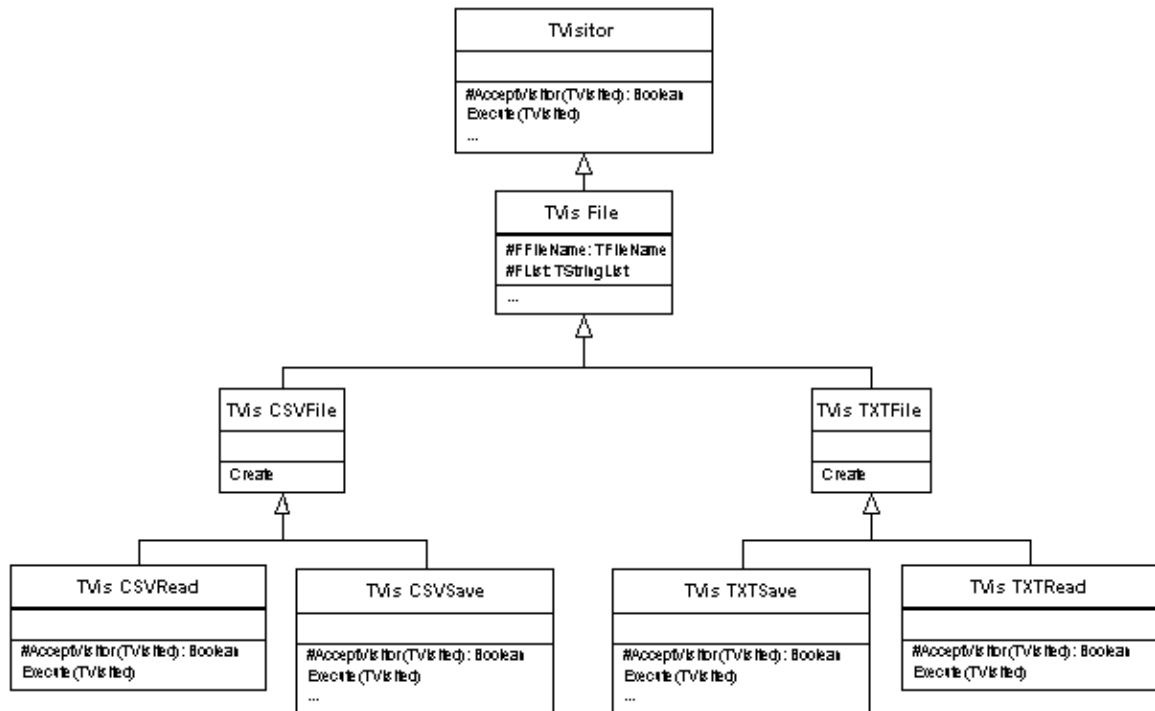
```
TVisFile = class( TVisitor )
protected
  FList : TStringList ;
  FFileName : TFileName ;
public
  constructor Create ; override ;
  destructor Destroy ; override ;
end ;
```

and the implementation looks like this:

```
constructor TVisFile.Create;
begin
  inherited;
  FList := TStringList.Create ;
  if FileExists( FFileName ) then
    FList.LoadFromFile( FFileName ) ;
end;

destructor TVisFile.Destroy;
begin
  FList.SaveToFile( FFileName ) ;
  FList.Free ;
  inherited;
end;
```

The value of FFileName is set in the constructor of the concrete class, which is hardly a good long term solution but it is sufficient to get us working here. The class diagram of the hierarchy we are building is shown below.



The next step is to create two descendants of TVisFile: one for managing fixed length text files (TXT) and the other for managing comma separated value (CSV) files. The implementation of these is shown below:

```

constructor TVisCSVFile.Create;
begin
  FFileName := 'Names.CSV' ;
  inherited;
end;

constructor TVisTXTFile.Create;
begin
  FFileName := 'Names.TXT' ;
  inherited;
end;
  
```

As you can see, the only implementation change is to set the value of FFileName in the constructor.

Step #10. Create the text file read and save Visitors

We implement two concrete visitors: one to read TXT files and one to save to TXT files. The TXT read visitor has the AcceptVisitor and Execute methods overridden. AcceptVisitor checks the object being visited is an instance of TPerson with the one line call `result := pVisited is TPerson`. The implementation of TVisTXTRead.Execute is shown below:

```

procedure TVisTXTRead.Execute(pVisited: TVisited);
var
  i : integer ;
  lData : TPerson ;
begin
  if not AcceptVisitor( pVisited ) then
    Exit ; //==>
  TPeople( pVisited ).List.Clear ;
  for i := 0 to FList.Count - 1 do
    begin
      lData := TPerson.Create ;
      lData.Name := Trim( Copy( FList.Strings[i], 1, 20 ) ) ;
      lData.EMailAdrs := Trim( Copy( FList.Strings[i], 21, 80 ) ) ;
      TPeople( pVisited ).Add( lData ) ;
    end ;
  end ;
  
```

```
end;
```

This visitor clears the internal list in the TPeople class that was passed to the execute method, then scans the internal TStringList and creates an instance of TPerson for each line it finds. The Name and EmailAdrs properties are extracted from the space padded lines in the TStringList.

The TXT file save visitor performs the reverse of this operation. In the constructor the internal TStringList is emptied of data, then AcceptVisitor checks if the object being visited is an instance of TPerson. This is an important difference between the read visitor and the save visitor. The read visitor works on an instance of TPeople and the save visitor works on an instance of TPerson. Strange and confusing? Yes, on the surface it looks like it would be easier to read to and save from a TPeople, scanning the TStringList and writing to the TPeople for a read and scanning the TPeople and writing to the TStringList for a Save. This would definitely be easier when persisting to a text file, however when saving to a relational database, we only want to execute SQL for objects that have changed (newly created, delted or edited) so we want to have the flexiability to test an object's internal state in the AcceptVisitor method before calling Execute. So, TVisTXTSave.AcceptVisitor() has the one line call result := pVisited is TPerson.

TVisTXTSave.Execute simply writes the Name and EMailAdrs properties to the TStringList after padding them with spaces. The implementation of Execute is shown below:

```
procedure TVisTXTSave.Execute(pVisited: TVisited);
begin
  if not AcceptVisitor( pVisited ) then
    Exit ; //==>
  FList.Add( tiPadR( TPerson( pVisited ).Name, 20 ) +
            tiPadR( TPerson( pVisited ).EMailAdrs, 60 ) ) ;
end;
```

The tiPadR() function will pad a string with spaces.

Step #11. Create the CSV file read and save Visitor

The CSV file read and save visitors are similar to the TXT file versions, except that instead of using Copy() and tiPadR() to extract and write the object's properties, we use tiToken() which is a function that can break a delimited string up into it's individual pieces. (tiToken() was cloned from a Clipper function of the same name when I moved to Delphi 1 in the early days of Delphi). The implementation of TVisCSVRead.Execute and TVisCSVSave.Execute are shown below:

```
procedure TVisCSVSave.Execute(pVisited: TVisited);
begin
  if not AcceptVisitor( pVisited ) then
    Exit ; //==>
  // Build up a string comprising the Name and EmailAdrs separated by a comma
  FList.Add( TPerson( pVisited ).Name + ',' +
            TPerson( pVisited ).EMailAdrs ) ;
end;
```

```
procedure TVisCSVRead.Execute(pVisited: TVisited);
var
  i : integer ;
  lData : TPerson ;
begin
  if not AcceptVisitor( pVisited ) then
    Exit ; //==>
  TPeople( pVisited ).List.Clear ;
  for i := 0 to FList.Count - 1 do
    begin
      lData := TPerson.Create ;
      // Parse a string of the form 'Name,EmailAdrs'
      lData.Name := tiToken( FList.Strings[i], ',', 1 ) ;
      lData.EMailAdrs := tiToken( FList.Strings[i], ',', 2 ) ;
      TPeople( pVisited ).Add( lData ) ;
    end;
  end;
```

```
end ;
end;
```

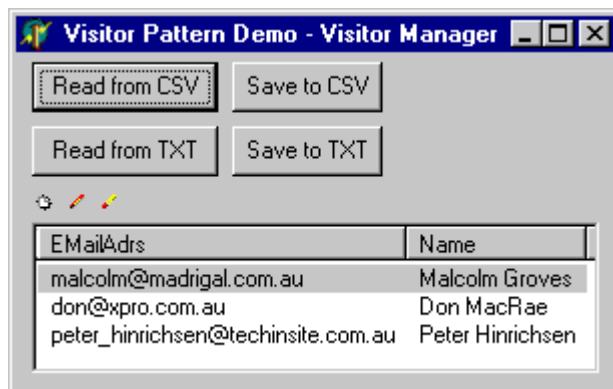
All that remains to be done now is to register the four visitors with the visitor manager, and test the application's ability to read from one file format and to save to another. We register the Visitors in the unit's initialization section like this:

```
initialization

gVisitorMgr.RegisterVisitor( 'CSVRead', TVisCSVRead ) ;
gVisitorMgr.RegisterVisitor( 'CSVSave', TVisCSVSave ) ;
gVisitorMgr.RegisterVisitor( 'TXTRead', TVisTXTRead ) ;
gVisitorMgr.RegisterVisitor( 'TXTSave', TVisTXTSave ) ;

end.
```

This means we can build a GUI like this, with the ability to read and write to and from different file formats.



Extra file formats can be easily added by registering the reader and writer Visitors with the Visitor manager.

Step #12. One final refactor

To prepare the code base for work with a relational database we will refactor it one last time to break the Visitor classes, abstract business classes, concrete business classes, abstract persistence classes and concrete persistence classes into separate units. When this is done, our demo application will have the following units and classes:

Unit name	Purpose	Classes
tiPtnVis.pas	Abstract Visitor and Visited classes	TVisitor TVisited
tiPtnVisMgr.pas	The Visitor Manager	TVisitorMapping TVisitorMgr
tiPtnVisPerObj.pas	Abstract business objects and business object list classes	TPerObjAbs TPerListAbs
People_BOM.pas	Concrete business objects	TPerson TPeople
People_Srv.pas	Concrete persistent classes	TVisFile TVisTXTFile TVisCSVFile TVisCSVSave

		TVisCSVRead TVisTXTSave TVisTXTRead
--	--	---

There are two things to notice about my file naming standard:

- All the units that contain visitor related abstract classes are prefixed tiPtnVis. This was based on an idea I had a while ago where by all units that contain pattern implementations would be prefixed tiPtn and placed in their own directory. This was a great idea at the time, but does not really work in practice as multiple patterns can be found in several units. We are unfortunately stuck with this standard because it would be to much work to change the existing systems that use the framework.

The People classes are implemented in two units: People_BOM.pas and People_Srv.pas. The _BOM unit contains the business object model, the _Srv.pas unit contains the persistence classes (_Srv stands for server side class). Sometimes, there will also be a _Cli.pas unit which will contain client side code like a Singleton instance of a class, or a thread with a progress bar to perform some lengthy database access and to keep the user amused while it is taking place.

Summary

In this chapter we have looked at the problem of iterating over a collection of objects, that may or may not be of the same type. We have used GoF's Visitor pattern to perform an operation on 0 to many of the objects then looked at how the text file read and write visitors can be developed to persist a collection of objects to a text file.

We have created an abstract TVisitor and TVisited class which define the visitor-visited relationship. The TVisited descendants know how to pass an instance of TVisitor over each object that it owns.

We have descended from TVisitedAbs and created an abstract business object class, and abstract business object collection class.

We have create a Visitor manager which allows visitors that perform different tasks to be registered and called by name. This allows us work read and write from different file formats.

The next session

In the next session, we shall develop a family of visitors that will read and write to a relational database. We shall also extend the abstract business objects so they are able to pass a visitor over more complex data structures

Chapter #3

The Visitor Framework and SQL Databases

The aims of this chapter

In the previous chapter we investigated how the Visitor pattern can be used to perform a family of related tasks on some elements in a list in a generic way. We used the framework to create a small application that saves names and email addresses to either a comma separated value (CSV) file or fixed length text (TXT) file.

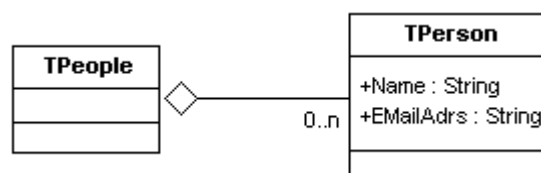
This is useful, but most business applications use a relational database to save their data, so in this chapter we will extend the framework to store our objects in an Interbase database.

Prerequisites

It would be best to have read 'Chapter #2: Implement the Visitor Framework' first because the concepts introduced in this chapter are building on the ideas discussed in Chapter #2

The business problem we will work with as an example

We will continue with our simplified version of the contact management system introduced in the previous chapter. We shall save a simple list of TPeople objects, with two properties Name and EMailAdrs. The class diagram of our business objects looks like this:



Implementation

Creating the database

Before we can begin, we will need a database to work with. The following SQL script will create an Interbase database with a single table called People. The table shall have two columns Name and EMailAdrs. This script is called SQLVisitor_Intabase_DDL.sql and can be found in the directory with the source code. (Details for downloading the source code are at the end of the chapter.) Note the Create Database and Connect commands have hard coded database name, user name and password. If you have unzipped the source into another directory, or have changed Interbase's administrator password, you will have to edit the script.

```

Create Database "C:\TechInsite\OPFPresentation\Source\3_SQLVisitors\Test.gdb"
user "SYSDBA" password "masterkey" ;

connect "C:\TechInsite\OPFPresentation\Source\3_SQLVisitors\Test.gdb"
user "SYSDBA" password "masterkey" ;

drop table People ;

create table People
( OID          Integer not null,
  Name         VarChar( 20 ),
  EMailAdrs   VarChar( 60 ),
  Primary Key ( OID ) ) ;

create unique index People_uk on People ( name ) ;

insert into People values ( 1, "Peter Hinrichsen",
"peter hinrichsen@techinsite.com.au");
insert into People values ( 2, "Don Macrae", "don@xpro.com.au" ) ;
insert into People values ( 3, "Malcolm Groves", "malcolm@madrigal.com.au" ) ;

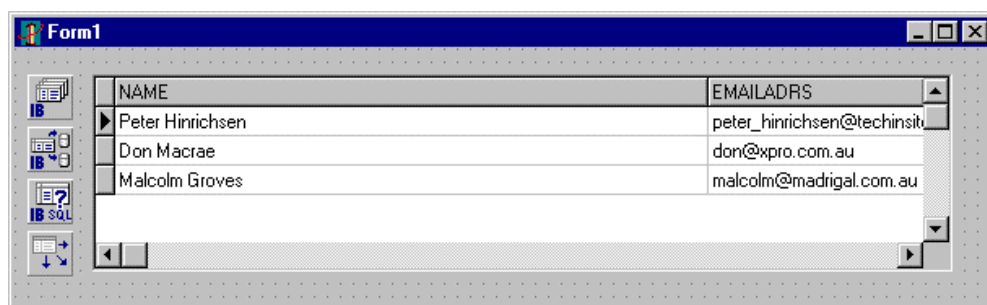
commit ;

```

Creating a database connection

We are going to perform this demonstration using the Interbase Express (IBX) components that come with Delphi. Now, I have been using Interbase for a while now, but have abstracted the database connection code deep into the hierarchy so very seldom have to create a TIBDatabase or TIBQuery manually in code. There are some tricks in wiring these components up with a TIBTransaction that I forget so we will use Delphi's IDE to help walk us through the process. We will create a form and add the components in Delphi's form editor then copy them to the clipboard and paste the code that Delphi generates into a PAS file.

I have dropped a TIBDatabase, TIBQuery, TIBTransaction (that's the one I forget), TDataSource and TDBGrid on a form, wired them up and set the active property to true. We can connect to the database as expected so I will copy the data access controls to the clipboard and paste them into the pas file where we will be creating our SQL visitor. The form to test the data access components is shown below:



If we copy the components at design time onto the clipboard, and paste them into the pas file, we get some text that looks like this:

```
object IBDatabase1: TIBDatabase
  Connected = True
  DatabaseName = 'C:\TechInsite\OPFPresentation\Source\3_SQLVisitors\Test.gdb'
  Params.Strings = (
    'user name=SYSDBA'
    'password=masterkey')
  LoginPrompt = False
  DefaultTransaction = IBTransaction1
  IdleTimer = 0
  SQLDialect = 1
  TraceFlags = []
  Left = 8
  Top = 16
end
object IBTransaction1: TIBTransaction
  Active = True
  DefaultDatabase = IBDatabase1
  Left = 8
  Top = 48
end
object IBQuery1: TIBQuery
  Database = IBDatabase1
  Transaction = IBTransaction1
  Active = True
  CachedUpdates = False
  SQL.Strings = (
    'select * from people')
  Left = 8
  Top = 80
end
```

This is actually the way Delphi stores the components in the form's DFM file. This technique of combining Delphi's IDE with coding components by hand was shown to me by Mark Miller in a presentation he made at BorCon 1999 – a very useful technique.

It only takes a couple of minutes work to edit this code from the DFM format into Pascal source. The finished result looks like this:

```
var
  FDB : TIBDatabase ;
  FTransaction : TIBTransaction ;
  FQuery : TIBQuery ;
  lData : TPerson ;
begin
  FDB := TIBDatabase.Create( nil ) ;
  FDB.DatabaseName := 'C:\TechInsite\OPFPresentation\Source\3_SQLVisitors\Test.gdb' ;
  FDB.Params.Add( 'user_name=SYSDBA' ) ;
  FDB.Params.Add( 'password=masterkey' ) ;
  FDB.LoginPrompt := False ;
  FDB.Connected := True ;

  FTransaction := TIBTransaction.Create( nil ) ;
  FTransaction.DefaultDatabase := FDB ;

  FDB.DefaultTransaction := FTransaction ;

  FQuery := TIBQuery.Create( nil ) ;
  FQuery.Database := FDB ;
  FQuery.SQL.Text := 'select * from people' ;
  FQuery.Active := True ;
end;
```

Our first SQL database visitor

We can wrap this code up as a TVisitor class by implementing the database access in the Visitors Execute method. The Interface of our SQL read Visitor looks like this:

```

TVisSQLRead = class( TVisitor )
protected
function AcceptVisitor( pVisited : TVisited ) : boolean ; override ;
public
procedure Execute( pVisited : TVisited ) ; override ;
end ;

```

We can add some code to scan the query result set, and convert each row to an object and we end up with a big, fat execute method like this:

```

procedure TVisSQLRead.Execute( pVisited: TVisited );
var
  FDB : TIBDatabase ;
  FTransaction : TIBTransaction ;
  FQuery : TIBQuery ;
  lData : TPerson ;
begin
  if not AcceptVisitor( pVisited ) then
    Exit ; //==>

  FDB := TIBDatabase.Create( nil ) ;
  FDB.DatabaseName := 'C:\TechInsite\OPFPresentation\Source\3_SQLVisitors\Test.gdb' ;
  FDB.Params.Add( 'user_name=SYSDBA' ) ;
  FDB.Params.Add( 'password=masterkey' ) ;
  FDB.LoginPrompt := False ;
  FDB.Connected := True ;

  FTransaction := TIBTransaction.Create( nil ) ;
  FTransaction.DefaultDatabase := FDB ;

  FDB.DefaultTransaction := FTransaction ;

  FQuery := TIBQuery.Create( nil ) ;
  FQuery.Database := FDB ;
  FQuery.SQL.Text := 'select * from people' ;
  FQuery.Active := True ;

  while not FQuery.EOF do
  begin
    lData := TPerson.Create ;
    lData.Name := FQuery.FieldName( 'Name' ).AsString ;
    lData.EmailAdrs := FQuery.FieldName( 'EMailAdrs' ).AsString ;
    TPeople( pVisited ).Add( lData ) ;
    FQuery.Next ;
  end ;

  FQuery.Free ;
  FTransaction.Free ;
  FDB.Free ;

end;

```

We can register this Visitor with the Visitor Manager and test it with our experimental application. The result is shown below:



Now that we have got the SQL read Visitor working, we can refactor the code to thin it down, and improve reuse and maintainability.

Move the database connection to an abstract Visitor

The code that creates and frees the TIBDatabase, TIBTransaction and TIBQuery is crying out to be moved to an abstract class. While we are about it, we shall create another pas file to store our abstract SQL visitors and call it tiPtnVisSQL. The interface of our new abstract Visitor called TVisQryAbs looks like this:

```
TVisSQLAbs = class( TVisitor )
protected
  FDB : TIBDatabase ;
  FTransaction : TIBTransaction ;
  FQuery : TIBQuery ;
  // Implement AcceptVisitor in the concrete class
  // function AcceptVisitor( pVisited : TVisited ) : boolean ; override ;
public
  constructor Create ; override ;
  destructor Destroy ; override ;
  // Implement Execute in the concrete class
  // procedure Execute( pVisited : TVisited ) ; override ;
end ;
```

and the constructor and destructors simply contain the code to create and destroy the TIBDatabase, TIBTransaction and TIBQuery as in the previous example.

This means our concrete visitor (the one that maps the SQL result set to objects) has an Execute method that looks like this:

```
procedure TVisSQLRead.Execute( pVisited: TVisited );
var
  lData : TPerson ;
begin
  if not AcceptVisitor( pVisited ) then
    Exit ; //==>

  FQuery.SQL.Text := 'select * from people' ;
  FQuery.Active := True ;

  while not FQuery.EOF do
  begin
    lData := TPerson.Create ;
    lData.Name := FQuery.FieldName( 'Name' ).AsString ;
    lData.EMailAdrs := FQuery.FieldName( 'EMailAdrs' ).AsString ;
    TPeople( pVisited ).Add( lData ) ;
    FQuery.Next ;
  end ;
end;
```

Implementing the Template Method pattern

This is starting to look more elegant, however if we have several of these visitors to read different classes from the database, we will be still duplicating lots of code. In the Execute method above, if we where to be reading, say Addresses instead of People, the lines in **red and bold** would be different, and the lines in *blue and italics* would be duplicated.

```
procedure TVisSQLRead.Execute( pVisited: TVisited );
var
  lData : TPerson ; //
Different
```

```

begin
    if not AcceptVisitor( pVisited ) then //
Duplicated
        Exit ; //
Duplicated

    FQuery.SQL.Text := 'select * from people' ; //
Differnt
    FQuery.Active := True ; //
Duplicated

    while not FQuery.EOF do //
Duplicated
        begin //
Duplicated
            lData := TPerson.Create ; //
Differnt
            lData.Name := FQuery.FieldName( 'Name' ).AsString ; //
Differnt
            lData.EmailAdrs := FQuery.FieldName( 'EmailAdrs' ).AsString ; //
Differnt
            TPeople( pVisited ).Add( lData ) ; //
Differnt
            FQuery.Next ; //
Duplicated
        end ; //
Duplicated
    end;

```

What we want to do is find a way of reusing the *blue italic* code, and only having to retype the **red bold** code. The solution to this is GoF's Template Method pattern.

GoF tell us that the intent of the Template Method is:

Define the skeleton of an algorithm in an operation, deferring some steps to client subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

This is exactly what we need here. If we look at the red bold code, we can see that the two blocks of code that must be written for each object-database mapping are `FQuery.SQL.Text := bla` and the code that maps the rows to the SQL data set to an object list. Lets move these to two procedures and implement them in the concrete class. All the code in *blue italic* will stay in the abstract class.

The interface of the re-factored abstract class, TVisSQLAbs class looks like this:

```

// Interface of the abstract SQL read visitor
TVisSQLAbs = class( TVisitor )
protected
    FDB : TIBDatabase ;
    FTransaction : TIBTransaction ;
    FQuery : TIBQuery ;
    procedure Init ; virtual ; // Implement in the concrete class
    procedure MapRowToObject ; virtual ; // Implement in the concrete class
    procedure SetupParams ; virtual ; // Implement in the concrete class
public
    constructor Create ; override ;
    destructor Destroy ; override ;
    procedure Execute( pVisited : TVisited ) ; override ;
end ;

```

And the implementation of the Execute method of TVisSQLAbs looks like this:

```
// Implementation of the abstract SQL read visitor
procedure TVisSQLAbs.Execute(pVisited: TVisited);
begin
  inherited Execute( pVisited ) ;
  if not AcceptVisitor then
    Exit ; //==>

  Init ; // Implemented in the concrete class

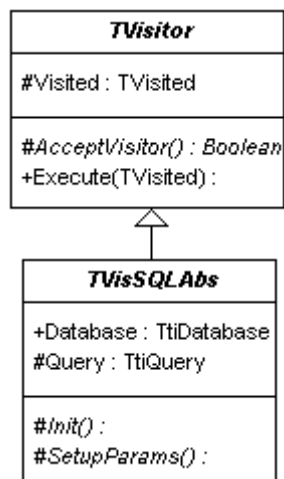
  SetupParams ; // Implemented in the concrete class
  FQuery.Active := True ;
  while not FQuery.EOF do
  begin
    MapRowToObject ; // Implemented in the concrete class
    FQuery.Next ;
  end ;
end;
```

The code below shows the interface and implementation of the concrete SQL Visitor TVisSQLReadPeople.

You can see there are four methods to override and implement:

1. **AcceptVisitor** - is the same as in previous examples – a check to see if the class being visited is of the correct type.
2. **Init** – is where we set the FQuery.SQL.Text value.
3. **SetupParams** – we have snuck this method in here and its name says it all. This is where we set any parameters required by the SQL. We can read properties from the object being visited to get the values.
4. **MapRowToObject** – is where each row of the query result set is turned into an object and added to the list contained in the class being visited.

The UML of the class hierarchy we have implemented so far looks like this:



Implementing the concrete SQL read visitor

Now that we have created an abstract SQL Visitor, we can implement a SQL Visitor to read people. The interface of TVisSQLReadPeople is shown below:

```
TVisSQLReadPeople = class( TVisSQLAbs )
```

```
protected
function AcceptVisitor : boolean ; override ;
procedure Init; override ;
procedure MapRowToObject ; override ;
procedure SetupParams ; override ;
end ;
```

And TVisSQLReadPeople's implementation is shown here. Note that we have overridden the four methods (AcceptVisitor, Init, MapRowToObject and SetupParams) required by the template we setup in the parent class.

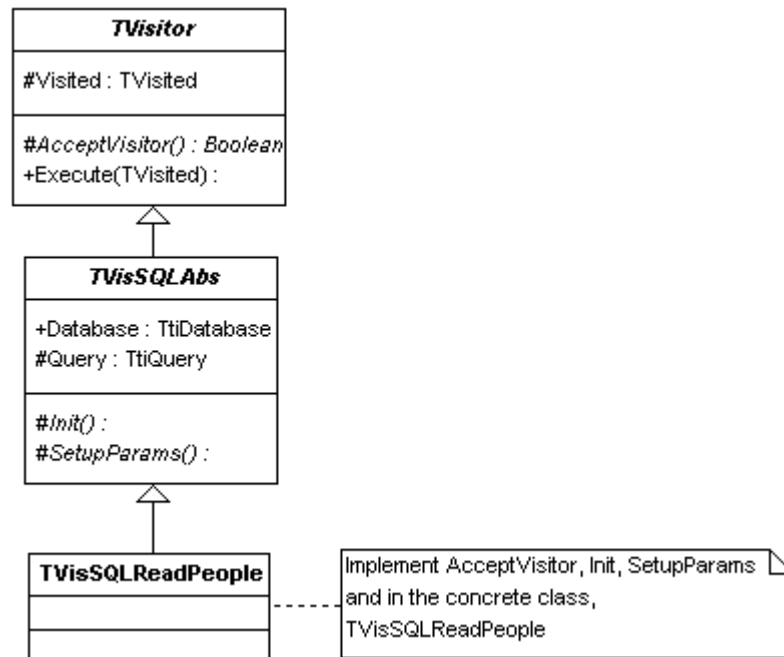
```
// AcceptVisitor is the same as in previous examples
function TVisSQLReadPeople.AcceptVisitor : boolean;
begin
    result := Visited is TPeople ;
end;
```

```
// Init is a new method where we can set the Query.SQL.Text value, or
// perform other setup tasks
procedure TVisSQLReadPeople.Init;
begin
    FQuery.SQL.Text := 'select * from people' ;
    TPeople( Visited ).List.Clear ;
end;
```

```
// SetupParams is where we set and parameters on the SQL. The parameter values can
// be read from the properties of the object being visited
procedure TVisSQLReadPeople.SetupParams;
begin
    // Do nothing yet, we will implement this in a future example
end;
```

```
// MapRowToObject is where the SQL result set rows are turned into objects and is
// called
// once for each row returned by the SQL result set.
procedure TVisSQLReadPeople.MapRowToObject;
var
    lData : TPerson ;
begin
    lData := TPerson.Create ;
    lData.Name := FQuery.FieldName( 'Name' ).AsString ;
    lData.EmailAdrs := FQuery.FieldName( 'EMailAdrs' ).AsString ;
    TPeople( Visited ).Add( lData ) ;
end;
```

The UML or the class hierarchy we have build so far looks like this:



The Visitor is registered with the Visitor Manager as usual, and then executed by passing a call to the visitor manager in the same way as for the text file visitors we build in the previous chapter. This is shown below:

```

// Registering TVisSQLReadPeople is the same as usual
initialization
  gVisitorMgr.RegisterVisitor( 'SQLRead', TVisSQLReadPeople ) ;
end.

// And calling TVisSQLReadPeople is the same as usual
begin
  gVisitorMgr.Execute( 'SQLRead', FPeople ) ;
end;

```

We now have the beginnings of a system that will let us read data from one format, and save it to another. For example, the code below will read TPerson(s) from a SQL database, and save them to a CSV file:

```

// Register the SQLRead Visitor and CSVSave Visitors
initialization
  gVisitorMgr.RegisterVisitor( 'CSVSave', TVisCSVSave ) ;
  gVisitorMgr.RegisterVisitor( 'SQLRead', TVisSQLReadPeople ) ;
end.

// Read from the SQL database, and save to a CSV file is now easy
var
  lPeople : TPeople ;
begin
  lPeople := TPeople.Create ;
  try
    gVisitorMgr.Execute( 'SQLRead', lPeople ) ;
    gVisitorMgr.Execute( 'CSVSave', lPeople ) ;
  finally
    lPeople.Free;
  end ;
end ;

```

As you can imagine, it is possible to register any number of Visitors to read and write from different file formats, or database types. In the next section, we will expand the Visitor framework to save objects (as well as read) to a SQL database.

Refactor the SQLVisitor hierarchy for read and update (create, update, delete) SQL

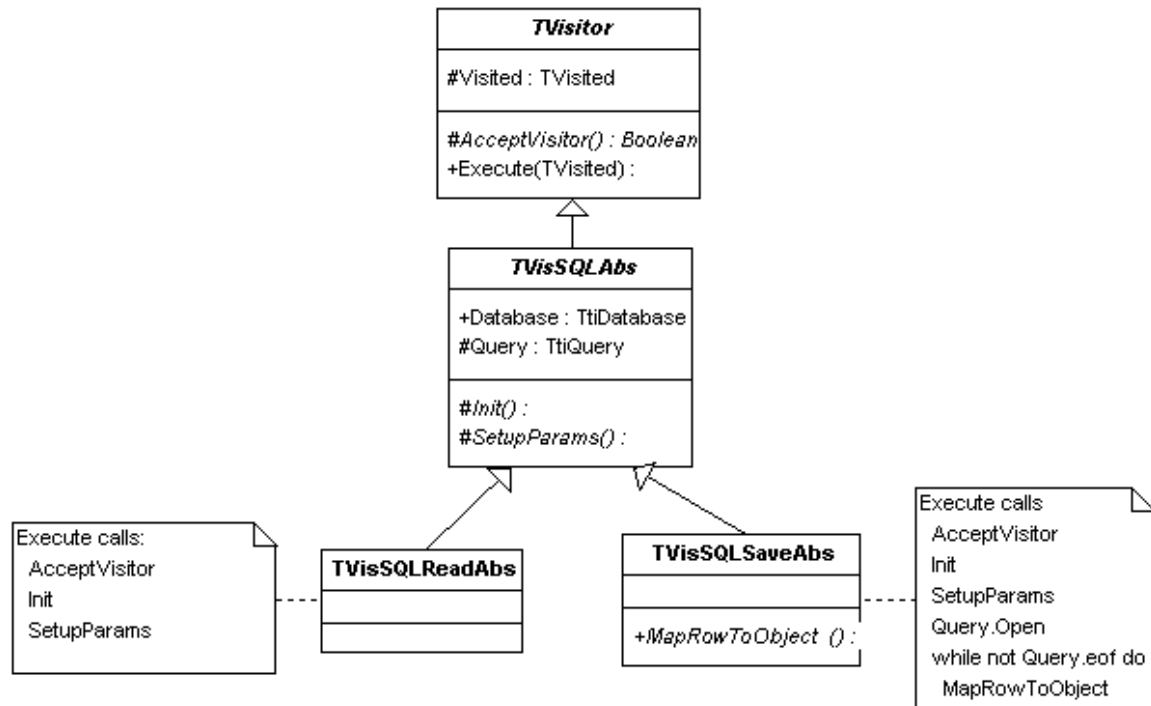
This is where things become a little trickier. When we were creating our CSV file save visitor, it was easy because we just wrote out all the objects in the list. We can't do this with an SQL database because some of the object may already exist in the database, and attempting to save them again would cause duplicate data and possibly database key violations. We could delete all the records before saving them all back like we do when saving to the CSV file, but writing to a text file is very quick, while interacting with an SQL database can be quite slow. We want to cause as little communication with the database as possible and will write an update Visitor to achieve this goal.

We will start by writing a Visitor to save changed objects (calling UPDATE SQL), then introduce the concept of an ObjectState property which will let the framework determine whether it has to run CREATE, UPDATE or DELETE SQL.

The steps we found are necessary when reading objects from a SQL database are:

Task	Method name
Should we be visiting this object?	AcceptVisitor
Setup the Query, assign the SQL	Init
Set any parameters required by the SQL	SetupParams
Turn the SQL result set into objects	MapRowToObject

Calling a SQL UPDATE statement requires pretty much the same steps, except MapRowToObject is not necessary. We really need a Map-Object-To-Row method, however this can be taken care of in SetupParams. We can now refactor our abstract SQL Visitor so the database connection and query remain in the abstract class, and we introduce two more classes at the next level down; one for reading objects by calling SELECT SQL (TVisSQLReadAbs). And one for saving them (TVisSQLUpdateAbs) by running UPDATE, CREATE or DELETE SQL. The execute methods for each will have to be slightly different too. The UML of the modified SQL Visitor hierarchy is shown below:



The interface of the abstract SQL visitor is shown below. We have introduced the Database connection, Query as well as the Init and SetupParams methods.

```

// The abstract SQL visitor contains the database connection, query
// Init and SetupParams methods
TVisSQLAbs = class( TVisitor )
protected
  FDB : TIBDatabase ;
  FTransaction : TIBTransaction ;
  FQuery : TIBQuery ;
  procedure Init ; virtual ;
  procedure SetupParams ; virtual ;
public
  constructor Create ; override ;
  destructor Destroy ; override ;
end ;
  
```

The interface of the abstract SQL read visitor is shown next. We have added the MapRowToObject method and have implemented Execute as shown below:

```

// The abstract SQL read visitor adds the MapRowToObject method, and
// implements the calls to Init, SetupParams and MapRowToObject
// (and FQuery.Open) in the correct order
TVisSQLReadAbs = class( TVisSQLAbs )
protected
  procedure MapRowToObject ; virtual ;
public
  procedure Execute( pVisited : TVisited ) ; override ;
end ;
  
```

```
procedure TVisSQLReadAbs.Execute(pVisited: TVisited);
begin
  inherited Execute( pVisited ) ;
  if not AcceptVisitor then
    Exit ; //==>
  Init ; // Set the SQL. Implemented in the concrete class
  SetupParams ; // Set the Queries parameters. Implemented in the concrete class
  FQuery.Active := True ;
  while not FQuery.EOF do
  begin
    MapRowToObject ; // Map a query row to an object. Implemented in the concrete
class
    FQuery.Next ;
  end ;
end;
```

The interface of the SQL update visitor has no additional methods and only contains the overridden Execute procedure.

```
// The abstract SQL update visitor just has an execute method and implements
// calls to Init and SetupParams (and FQueyr.SQLExec)
TVisSQLUpdateAbs = class( TVisSQLAbs )
public
  procedure Execute( pVisited : TVisited ) ; override ;
end ;
```

And the implementation of the abstract SQL update Visitor's Execute method looks like this:

```
procedure TVisSQLUpdateAbs.Execute(pVisited: TVisited);
begin
  inherited Execute( pVisited ) ;
  if not AcceptVisitor then
    Exit ; //==>
  Init ; // Set the SQL. Implemented in the concrete class
  SetupParams ; // Set the Queries parameters. Implemented in the concrete class
  FQuery.ExecSQL ;
end;
```

Now that we have refactored the abstract SQL read and update visitors, it is a simple matter to write our first concrete Visitor to update objects to a SQL database.

Write an UPDATE Visitor

We now have a stub of an object that will walk us through the steps we have to take to update a changed object in the database. We have to implement the Init method, where we will write some SQL, and the SetupParams method where we will map the object's properties to the query's parameters. We will start by writing some update SQL like this:

```
update People
set
    Name      = :Name
    ,EmailAdrs = :EmailAdrs
where
    Name      = :Old Name
and EmailAdrs = :Old_EmailAdrs
```

This SQL is the same as we get when we drop a TQuery on a form and hook it up to a TUpdateSQL. Now if we use a TQuery TClientDataSet on a form with a TDBGrid, it will internally maintain a list of changed rows, keeping a copy of the both the old and new values. This is where the Old_Name and Old_EmailAdrs values were derived from. Our framework as it stands has no knowledge of how an object looked before it was edited. We have taken over this responsibility from the TQuery and it's about now in our implementation of an OPF that we realise how clever the TQuery and TClientDataSet components really are. This is the point when we can quite rightly reassess whether or not it is worth our while developing a custom persistence framework. (I still maintain that it's well worth the effort.)

The solution to our problem is contained in a paper by Scott Ambler found at <http://www.ambysoft.com/mappingObjects.pdf>. The discussion starts off on page one with the heading 'The importance of OIDs'. In a relational database, a row in a table can be uniquely identified by its primary key fields. In our People table, this field would probably be Name. That's fine, except when we want to join the People table to another table with a foreign key relationship. We can use the name field as the link between the two tables that will work well until we want to change a value in Name. How do we do that? The database's referential integrity will make this very difficult. The solution (and Ambler spends considerable time driving this point home) is to make sure your primary key fields have no business meaning. Integer is the perfect candidate data type for OIDs and is what we will use here. (Ambler describes several strategies to generating OIDs including High/Low integers, which is what is used in the tiOPF, GUIDS and others – his paper is well worth a read.)

We now have several changes to make to our earlier work:

1. The table structure must be extended with the OID column, and the data populated with OID values.
2. Our TPerson class must be extended with an OID property
3. The read visitor must be extended to read an object's OID

We will change the script to create our test table to look like this:

```
create table People
(
    OID          Integer not null,
    Name         VarChar( 20 ),
    EmailAdrs    VarChar( 60 ),
    Primary Key ( OID ) ;

insert into People values ( 1, "Peter Hinrichsen",
"peter_hinrichsen@techinsite.com.au");
insert into People values ( 2, "Don Macrae", "don@xpro.com.au" );
insert into People values ( 3, "Malcolm Groves", "malcolm@madrigal.com.au" );
```

While we are adding the OID column, we will take the opportunity to make a NOT NULL field, and set it as the table's primary key.

Next, our TPerson class must be extended with an OID property. At this stage in the development of the framework I decided that all objects would have a property called OID of type Int64. That's has been a good rule to enforce when building a system from scratch, with complete control over the database schema. However if we where building an OO front end to a legacy system, we would have to come up with a more versatile OID/Primary key strategy. As all objects will have an OID, lets add it to the abstract business object class. We will define a type called TOID that will be an Int64. This will make it easier to change the implementation of TOID down the track. Our re-factored TPerObjAbs class looks like this:

```
TOID = Int64 ;

TPerObjAbs = class( TVisited )
private
    FOID: TOID;
public
    constructor Create ; virtual ;
    property    OID : TOID read FOID write FOID ;
end ;
```

And the changed TVisSQLReadPeople looks like this:

```
procedure TVisSQLReadPeople.MapRowToObject;
var
    lData      : TPerson ;
begin
    lData := TPerson.Create ;
    lData.OID      := FQuery.FieldName( 'OID' ).AsInteger ;
    lData.Name     := FQuery.FieldName( 'Name' ).AsString ;
    lData.EmailAdrs := FQuery.FieldName( 'EmailAdrs' ).AsString ;
    TPeople( Visited ).Add( lData ) ;
end;
```

I have also refactored the CSV and TXT file visitors, but have not shown the code for these here.

Now, back to where we where 20 minutes ago, we can write our UPDATE SQL using the OID value to find the record to update.

```
update People
set
    Name      = :Name
    ,EmailAdrs = :EmailAdrs
where
    OID      = :OID
```

It is now possible to finish the TVisSQLUpdatePeople class with the implementation looking like this:

```
function TVisSQLUpdatePeople.AcceptVisitor: boolean;
begin
    result := Visited is TPerson ;
end;
```

```
procedure TVisSQLUpdatePeople.Init;
begin
    FQuery.SQL.Text :=
        'update People ' +
        'set ' +
        '    Name      = :Name ' +
        '    ,EmailAdrs = :EmailAdrs ' +
        'where ' +
        '    OID      = :OID' ;
end;
```

```
procedure TVisSQLUpdatePeople.SetupParams;
var
  lData : TPerson ;
begin
  lData := TPerson( Visited ) ;
  FQuery.Params.ParamByName( 'OID' ).AsInteger      := lData.OID      ;
  FQuery.Params.ParamByName( 'Name' ).AsString      := lData.Name      ;
  FQuery.Params.ParamByName( 'EMailAdrs' ).AsString := lData.EMailAdrs ;
end;
```

And of course don't forget to register TVisSQLUpdatePeople with the Visitor Manager in the unit's implementation section like this:

```
// Register the SQLSave Visitor
initialization
  gVisitorMgr.RegisterVisitor( 'SQLSave', TVisSQLUpdatePeople ) ;
end.
```

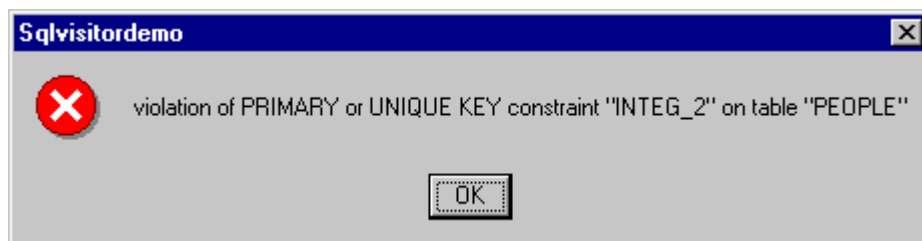
The need for an ObjectState property

It should now be easy to write a CREATE visitor. It can potentially share the same SetupParams code as the UPDATE visitor, with modified SQL. The implementation of the new Init method is shown below:

```
procedure TVisSQLCreatePeople.Init;
begin
  FQuery.SQL.Text :=
    'insert into People ' +
    '( OID, Name, EMailAdrs ) ' +
    'values ' +
    '( :OID, :Name, :EMailAdrs ) ' ;
end;
```

```
// Register the SQLSave Visitor
initialization
  gVisitorMgr.RegisterVisitor( 'SQLSave', TVisSQLUpdatePeople ) ;
  gVisitorMgr.RegisterVisitor( 'SQLSave', TVisSQLCreatePeople ) ;
end.
```

We register the new Visitor with the Visitor Manager, compile and run the code then test by inserting an new value and...



...another problem. Primary key violation on table People. Well, we only have one primary key field and that's OID. This problem is being caused because we are calling our CREATE Visitor for every object in the list, not just those that that must be newly created. We need some way of identifying an object's state: Is it clean, does it need creating, updating or deleting? This can be done by introducing an ObjectState property at the level of TPerObjAbs.

In the tiOPF, ObjectState is an ordinal type and can have the following values:

Value	Description
posEmpty	The object has been created, but not filled with data from the DB
posPK	The object has been created, but only it's primary key has been read
posCreate	The object has been created and populated with data and must be saved to the DB
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 in the database
posClean	The object is 'Clean' no DB update necessary

The TPerObjAbsState type is declared like this:

```
TPerObjectState = (  
    posEmpty, posPK, posCreate, posUpdate, posDelete, posDeleted, posClean ) ;
```

What we are aiming to do is to be able to selectively run visitors depending on whether an object needs to be newly created, updated or deleted from the database. For example, we would like to achieve the same as this case statement, only within the Visitor framework:

```
case Visited.ObjectState of  
    posCreate : // Call CREATE SQL  
    posUpdate : // Call UPDATE SQL  
    posDelete : // Call DELETE SQL  
    else  
        // Do nothing  
end ;
```

This can be achieved by checking Visited.ObjectState inside the Visitors AcceptVisitor method.

Now we have several things to do to finish off the work with ObjectState:

1. Set ObjectState when we create, update or delete an object in the GUI;
2. Filter out objects that have been marked as posDelete or posDeleted from the GUI;
3. Check the ObjectState property in AcceptVisitor; and
4. Set the ObjectState back to posClean after a Visitor has run.

Setting ObjectState when we create, update or delete an object in the GUI is a chore but can be made easier by introducing two new properties on TPerObjAbs: Deleted and Dirty. The interface of TPerObjAbs is extended as shown below:

```
TPerObjAbs = class( TVisited )  
    private  
        FObjectState: TPerObjectState;  
    protected  
        function GetDirty: boolean; virtual ;  
        procedure SetDirty(const Value: boolean);virtual ;  
        function GetDeleted: boolean;virtual ;  
        procedure SetDeleted(const Value: boolean);virtual ;  
    public  
        property ObjectState : TPerObjectState read FObjectState write FObjectState ;  
        property Dirty : boolean read GetDirty write SetDirty ;  
        property Deleted : boolean read GetDeleted write SetDeleted ;  
end ;
```

The implementation of SetDirty and GetDirty is shown below. Set dirty will change the object state to posUpdate from posEmpty, posPK or posClean, and will not change the object state if the object is already in one of the 'Dirty' states.

```
procedure TPerObjAbs.SetDirty(const Value: boolean);
begin
  case ObjectState of
    posEmpty   : FObjectState := posCreate   ;
    posPK      : FObjectState := posUpdate   ;
    posCreate  : FObjectState := FObjectState ; // Do nothing
    posUpdate  : FObjectState := FObjectState ; // Do nothing
    posDelete  : FObjectState := FObjectState ; // Do nothing
    posDeleted : FObjectState := FObjectState ; // Do nothing
    posClean   : FObjectState := posUpdate   ;
  end ;
end;
```

GetDirty checks for an ObjectState of posCreate, posUpdate or posDelete. PosEmpty, posPK, posClean and posDeleted are not regarded as dirty state that mean the object has changed and must be saved back to the database.

```
function TPerObjAbs.GetDirty: boolean;
begin
  result := ObjectState in [posCreate, posUpdate, posDelete] ;
end;
```

And the implementation of the Deleted Get and Set methods looks like this:

```
function TPerObjAbs.GetDeleted: boolean;
begin
  result := ObjectState in [posDelete, posDeleted];
end;
```

As you can see, GetDeleted will return True if ObjectState is either posDelete or posDeleted.

```
procedure TPerObjAbs.SetDeleted(const Value: boolean);
begin
  if ObjectState <> posDeleted then
    ObjectState := posDelete ;
end;
```

We can now use ObjectState, and the two helper properties Deleted and Dirty to filter objects by state when saving to the database.

Insert a new TPerson in the GUI

Our ultimate aim in the next few pages is to write a SQL Create Visitor, but before we can do that, we must insert some new objects into the list for saving to the database. We shall do this by writing extending the edit dialog we developed in the previous chapter.

The TtiListView on the main form has three event handlers: OnItemInsert, OnItemEdit and OnItemDelete. These are fired when the list view is double clicked, or right clicked and Insert, Edit or Delete are selected from the popup context menu. The code under OnItemInsert looks like this:

```
procedure TFormMain_VisitorManager.LVItemInsert (
  pLV: TtiCustomListView;
  pData: TPersistent;
  pItem: TListItem);
var
```

```
lData : TPerson ;  
begin  
  lData := TPerson.Create ;  
  lData.ObjectState := posCreate ; // Set the ObjectState property here  
  lData.OID := GetNewOID ; // A helper function that generates a new OID  
  FPeople.Add( lData ) ;  
  TFormEditPerson.Execute( lData ) ;  
end;
```

The call to TFormEditPerson.Execute(lData) executes the following code, and shows the dialog box as in the screen shot below:

```
class procedure TFormEditPerson.Execute( pData: TPerObjAbs);  
var  
  lForm : TFormEditPerson ;  
begin  
  lForm := TFormEditPerson.Create( nil ) ;  
  try  
    lForm.Data := pData ;  
    lForm.ShowModal ;  
  finally  
    lForm.Free ;  
  end ;  
end;
```



Under the OK button of this dialog we set the TPerson's Dirty property to true like this:

```
procedure TFormEditPerson.btnOKClick( Sender: TObject);  
begin  
  FData.Dirty := true ;  
  ModalResult := mrOK ;  
end;
```

Having to remember to set Dirty := true after an edit is a chore, and is error prone. The need to do this could be removed by setting Dirty := true in each properties Set method, but there would need to be some code to turn this feature off while populating an object from the persistent store.

Another solution is to use visual form inheritance and create an abstract edit dialog that takes care of setting pData.Dirty := True under the OK button. This is how we implement edit dialogs in the tiOPF and shall be discussed later.

Now that we have set the ObjectState property after inserting a new object, we can write the Create Visitor.

Write a CREATE Visitor

The first thing we must do in the Create Visitor is check the ObjectState of the object being processed in the AcceptVisitor method. This will make it possible to run CREATE SQL against only the objects that have been newly created, and UPDATE SQL against objects that already exist in the database, and must be updated.

An example of this is shown below:

```
function TVisSQLCreatePeople.AcceptVisitor: boolean;
begin
    result := ( Visited is TPerson ) and
              ( TPerson( Visited ).ObjectState = posCreate ) ;
end;
```

The full interface of the Create Visitor is shown below:

```
Interface

TVisSQLCreatePeople = class( TVisSQLUpdateAbs )
protected
    function AcceptVisitor : boolean ; override ;
    procedure Init ; override ;
    procedure SetupParams ; override ;
end ;
```

And the implementation of the Create Visitor is shown next:

```
implementation

//-----
---
function TVisSQLCreatePeople.AcceptVisitor: boolean;
begin
    result := ( Visited is TPerson ) and
              ( TPerson( Visited ).ObjectState = posCreate ) ;
end;

//-----
---
procedure TVisSQLCreatePeople.Init;
begin
    FQuery.SQL.Text :=
        'insert into People ' +
        '( OID, Name, EMailAdrs ) ' +
        'values ' +
        '( :OID, :Name, :EMailAdrs ) ' ;
end;

//-----
---
procedure TVisSQLCreatePeople.SetupParams;
var
    lData : TPerson ;
begin
    lData := TPerson( Visited ) ;
    FQuery.Params.ParamByName( 'OID' ).AsInteger := lData.OID ;
    FQuery.Params.ParamByName( 'Name' ).AsString := lData.Name ;
    FQuery.Params.ParamByName( 'EMailAdrs' ).AsString := lData.EMailAdrs ;
end;
```

This Visitor is registered with the Visitor Manager in the usual way:

```
initialization
    gVisitorMgr.RegisterVisitor( 'SQLSave', TVisSQLCreatePeople ) ;
end.
```

Write a DELETE Visitor

Now that we have worked out how to determine which Visitors to call when, we can write our DELETE visitors. Their interface will be the same as the CREATE and UPDATE Visitors because they descend from the same parent (and use the Template Method pattern) and looks like this:

```

Interface

TVisSQLUpdatePeople = class( TVisSQLUpdateAbs )
protected
  function AcceptVisitor : boolean ; override ;
  procedure Init ; override ;
  procedure SetupParams ; override ;
end ;

```

The implementation of TVisSQLDeletePeople looks like this:

```

Interface

//-----
---
function TVisSQLDeletePeople.AcceptVisitor: boolean;
begin
  result := ( Visited is TPerson ) and
             ( TPerson( Visited ).ObjectState = posDelete ) ;
end;

//-----
---
procedure TVisSQLDeletePeople.Init;
begin
  FQuery.SQL.Text :=
    'delete from People ' +
    'where ' +
    'OID = :OID' ;
end;

//-----
---
procedure TVisSQLDeletePeople.SetupParams;
var
  lData : TPerson ;
begin
  lData := TPerson( Visited ) ;
  FQuery.Params.ParamByName( 'OID' ).AsInteger := lData.OID ;
end;

```

And once again, the Visitor is registered with the Visitor Manager in the usual way:

```

initialization
  gVisitorMgr.RegisterVisitor( 'SQLSave', TVisSQLDeletePeople ) ;
end.

```

Registering Visitors in the correct order

It is important to register the Visitors with the Visitor Manager in the correct order. This is especially important if a tree hierarchy has been modeled and is represented in the database by a one to many relationship with database referential integrity. The order that visitors are registered is the order that the SQL is called and I find that the most reliable is to register them in the order of Read, Delete, Update then Create. This is shown below:

```

initialization

  gVisitorMgr.RegisterVisitor( 'SQLRead', TVisSQLReadPeople ) ;
  gVisitorMgr.RegisterVisitor( 'SQLSave', TVisSQLDeletePeople ) ;
  gVisitorMgr.RegisterVisitor( 'SQLSave', TVisSQLUpdatePeople ) ;
  gVisitorMgr.RegisterVisitor( 'SQLSave', TVisSQLCreatePeople ) ;

end.

```


Setting ObjectState back to posClean

Now that we have checked the ObjectState property for posCreate, posDelete or posUpdate in the Visitor's AcceptVisitor method and run the SQL in the Visitor we must set ObjectState back to posClean. To do this we will add an extra class between TVisitor and TVisSQLAbs called TVisPerObjectAwareAbs. This will let us descend our text file visitors from the same parent as the SQL visitor giving them both access to the new method called Final. The interface and implementation of TVisPerObjectAwareAbs looks like this:

```
interface
    TVisPerObjAwareAbs = class( TVisitor )
    protected
        procedure Final ; virtual ;
    end ;
```

```
implementation
    procedure TVisPerObjAwareAbs.Final;
    begin
        if TPerObjAbs( Visited ).ObjectState = posDelete then
            TPerObjAbs( Visited ).ObjectState := posDeleted
        else
            TPerObjAbs( Visited ).ObjectState := posClean ;
        end;
    end;
```

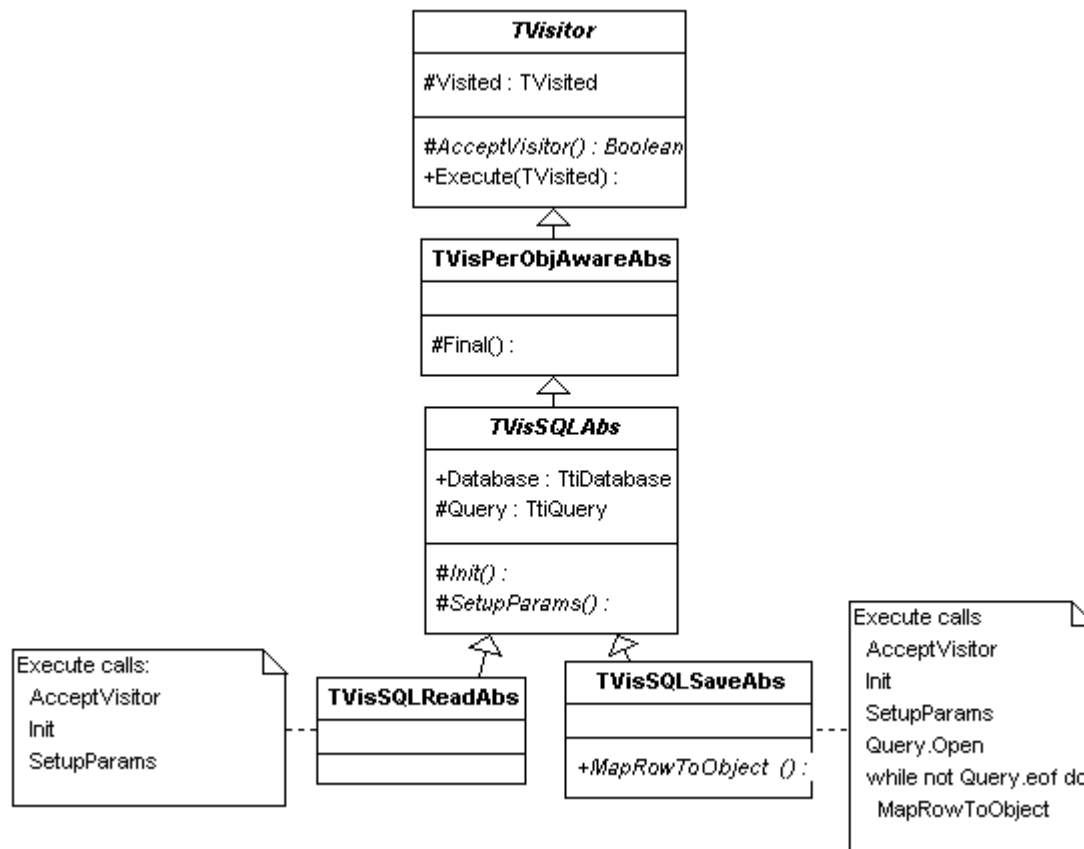
We can now call Final in the Execute method of both the Read and Update Visitors. The Execute method of TVisSQLReadAbs now looks like this:

```
procedure TVisSQLReadAbs.Execute(pVisited: TVisited);
begin
    inherited Execute( pVisited ) ;
    if not AcceptVisitor then
        Exit ; //==>
    Init ; // Set the SQL. Implemented in the concrete class
    SetupParams ; // Set the Queries parameters. Implemented in the concrete class
    FQuery.Active := True ;
    while not FQuery.EOF do
        begin
            MapRowToObject ; // Map a query row to an object. Implemented in the concrete
class
            FQuery.Next ;
        end ;
    Final ;
end;
```

And the execute method of TVisSQLUpdateAbs looks like this:

```
procedure TVisSQLUpdateAbs.Execute(pVisited: TVisited);
begin
    inherited Execute( pVisited ) ;
    if not AcceptVisitor then
        Exit ; //==>
    Init ; // Set the SQL. Implemented in the concrete class
    SetupParams ; // Set the Queries parameters. Implemented in the concrete class
    FQuery.ExecSQL ;
    Final ;
end;
```

The UML of the SQL visitor class hierarchy now looks like this:



Filtering in the GUI

Our objects can now have two states that should prevent them from being displayed in the GUI: posDelete (meaning they have been marked for deletion, but have not yet been deleted from the database) and posDeleted (meaning they have been marked for deletion, and removed from the database). If either of these conditions is true, the Deleted property will return true. If we check the deleted property while painting the TtiListView, we can filter the records we don't want to display. The TtiListView has an OnFilterRecord event that can be programmed like this:

```

procedure TFormMain_VisitorManager.LVFilterData (
  pData: TPersistent;
  var pbInclude: Boolean);
begin
  pbInclude := not TPerObjAbs( pData ).Deleted ;
end;
  
```

When the TtiListView's ApplyFilter property is set to True, the objects that have an ObjectState of posDelete or posDeleted will be filtered out and not displayed.

Add some logging to help debugging

It does not take much time debugging this code before you will realise how difficult it can be to keep track of what is going on. The business object model is decoupled from the persistence layer, but with the continual looping within the TVisited.Iterate method, tracking errors, especially in the SQL can be quite a torturous process. The solution is to add some logging and to help with this and have developed the TtiLog family of classes.

To add logging to the application, add the unit tiLog.pas to the tiPtnVisSQL.pas uses clause, then add the Log() command in the Visitor's execute method like this:

```

procedure TVisSQLReadAbs.Execute(pVisited: TVisited);
  
```

```

begin
  inherited Execute( pVisited ) ;
  if not AcceptVisitor then
    Exit ; //==>
  Log( 'Calling ' + ClassName + '.Execute' ) ;
  Init ;           // Set the SQL. Implemented in the concrete class
  SetupParams ;   // Set the Queries parameters. Implemented in the concrete class
  FQuery.Active := True ;
  while not FQuery.EOF do
  begin
    MapRowToObject ; // Map a query row to an object. Implemented in the concrete
class
    FQuery.Next ;
  end ;
  Final ;
end;

```

and this...

```

procedure TVisSQLUpdateAbs.Execute( pVisited: TVisited);
begin
  inherited Execute( pVisited ) ;
  if not AcceptVisitor then
    Exit ; //==>
  Log( 'Calling ' + ClassName + '.Execute' ) ;
  Init ;           // Set the SQL. Implemented in the concrete class
  SetupParams ;   // Set the Queries parameters. Implemented in the concrete class
  FQuery.ExecSQL ;
  Final ;
end;

```

It is also a good idea to add logging to AcceptVisitor, SetupParams, MapRowToObject and Final with each call that is likely to call an exception being surrounded by a try except block. This makes it easier to locate the source of an error by reading the log trace, which is much quicker than having to step through the code in the IDE.

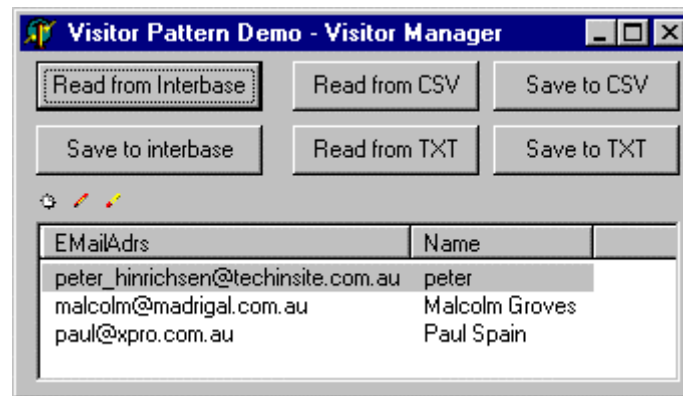
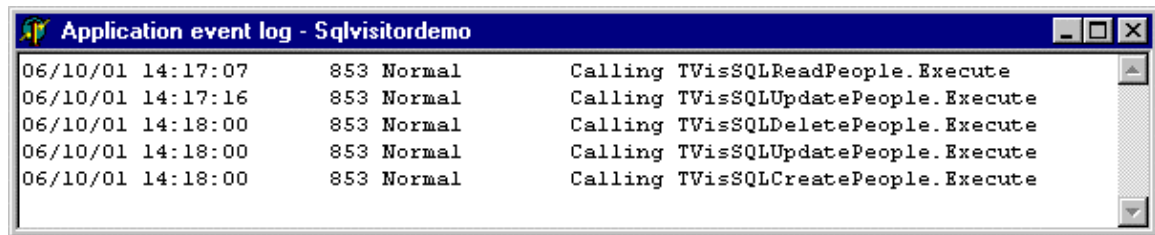
For example, the TVisSQLUpdateAbs.Execute method is refactored like this in its implementation in the tiOPF:

```

procedure TVisQryUpdate.Execute( pData: TVisitedAbs);
  procedure DoExecuteQuery ;
  begin
    try
      Query.ExecSQL ;
    except
      on e:exception do
        tiFmtException( e, ClassName, ' ExecuteQuery',
          DBExceptionMessage( 'Error opening query', e ) ) ;
    end ;
  end ;
begin
  Inherited Execute( pData ) ;
  if not DoAcceptVisitor then
    exit ; //==>
  DoInit ;
  DoGetQuery ;
  DoSetupParams ;
  DoExecuteQuery ;
end ;

```

To get logging working you also have to call SetupLogForClient somewhere in the application and the DPR file is as good a place as any. To turn visual logging on, you must pass the -lv parameter on the command line. An example of how to use the tiLog classes can be found in the DemoTILog directory. An application running with visual logging turned on will typically look like this:



Change the CSV and TXT visitors to ignore deleted objects

When we first wrote the TVisCSVSave and TVisTXTSave classes, we assumed we wanted to save all the objects in the list that was passed to the visitor manager. This was a good strategy for persisting to a text file, but as we discussed above, when saving to a SQL database, we must maintain a list of objects that are marked for deletion so they can be removed from the database as part of the save.

In the main form, under the delete button we had the following code that removes then frees the object from the list:

```
procedure TFormMain_VisitorManager.LVItemDelete(
  pLV: TtiCustomListView; pData: TPersistent; pItem: TListItem);
begin
  FPeople.List.Remove( pData );
end;
```

For saving to a SQL database, this has been changed to marking the object for deletion, rather than removing it from the list.

```
procedure TFormMain_VisitorManager.LVItemDelete(
  pLV: TtiCustomListView; pData: TPersistent; pItem: TListItem);
begin
  TPerObjAbs( pData ).Deleted := true ;
end;
```

The AcceptVisitor method in TVisCSVSave and TVisTXTSave must be extended to skip over records that have been deleted, or marked for deletion:

```
function TVisCSVSave.AcceptVisitor : boolean;
begin
  result := ( Visited is TPerson ) and
    ( not TPerson( Visited ).Deleted ) ;
end;
```

Enable the save button in the GUI only when the object hierarchy is dirty

The way we have designed the application's main form has the save buttons enabled all the time. It would be nice if we could disable the save button when the data in the hierarchy is clean and enable the buttons only when a CREATE, UPDATE or DELETE must be made.

We have added a Dirty property to the TPerObjAbs class, but this only checks the one classes ObjectState. What we need is a way of iterating over all owned objects and to check if any of them are dirty. This is easily achieved by writing an IsDirty Visitor and calling it inside the object's GetDirty method. The interface of TVisPerObjIsDirty looks like this:

```
TVisPerObjIsDirty = class( TVisitorAbs )
private
  FbDirty: boolean;
protected
  function AcceptVisitor : boolean ; override ;
public
  procedure Execute( pVisited : TVisitedAbs ) ; override ;
  property Dirty : boolean read FbDirty write FbDirty ;
end ;
```

And the implementation looks like this:

```
function TVisPerObjIsDirty.AcceptVisitor : boolean;
begin
  result := ( Visited is TPerObjAbs ) and
            ( not Dirty ) ;
end;
```

```
procedure TVisPerObjIsDirty.Execute(pVisited: TVisitedAbs);
begin
  Inherited Execute( pVisited ) ;

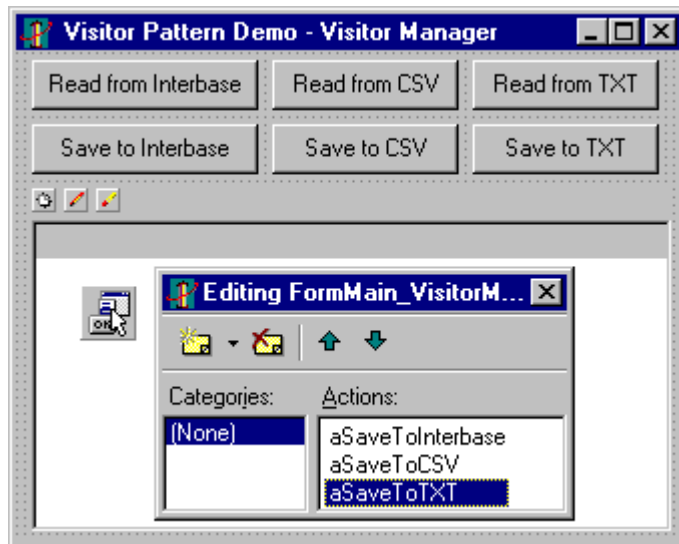
  if not AcceptVisitor then
    exit ; //==>

  Dirty := TPerObjAbs( pVisited ).ObjectState in
    [ posCreate,
      posUpdate,
      posDelete
    ]
end;
```

The call to this Visitor is wrapped up in the TPerObjAbs.GetDirty method like this:

```
function TPerObjAbs.GetDirty: boolean;
var
  lVis : TVisPerObjIsDirty ;
begin
  lVis := TVisPerObjIsDirty.Create ;
  try
    self.Iterate( lVis ) ;
    result := lVis.Dirty ;
  finally
    lVis.Free ;
  end ;
end;
```

This lets us extend the application's main form by adding an ActionList. Double click the ActionList and add three actions as shown below. Move the code from the three save buttons OnClick methods to the actions then hook the save buttons up to the actions.



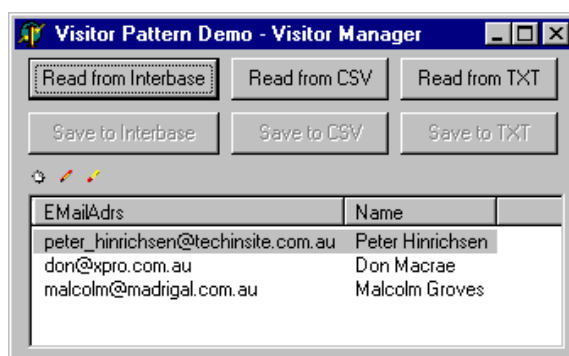
In the ActionList's OnUpdate method add the following code to check the Dirty state of the data hierarchy and enable or disable the save buttons as necessary:

```

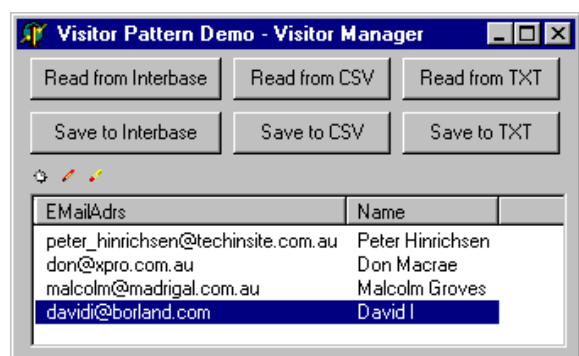
procedure TFormMain_VisitorManager.ALUpdate(Action: TBasicAction;
var Handled: Boolean);
var
    lDirty : boolean ;
begin
    lDirty := FPeople.Dirty ;
    aSaveToInterbase.Enabled := lDirty ;
    aSaveToCSV.Enabled      := lDirty ;
    aSaveToTXT.Enabled      := lDirty ;
    Handled := true ;
end;
    
```

Now the save buttons are disabled when the object hierarchy is clean, and disabled when the object hierarchy is dirty like this:

The object hierarchy is clean



The object hierarchy is dirty



Adding more database constraints

Now let's create a slightly more realistic database schema by adding a unique key on the People table. This can be done by modifying the create SQL as shown below:

```

create table People
(
    OID           Integer not null,
    Name          VarChar( 20 ),
    EMailAdrs    VarChar( 60 ),

```

```

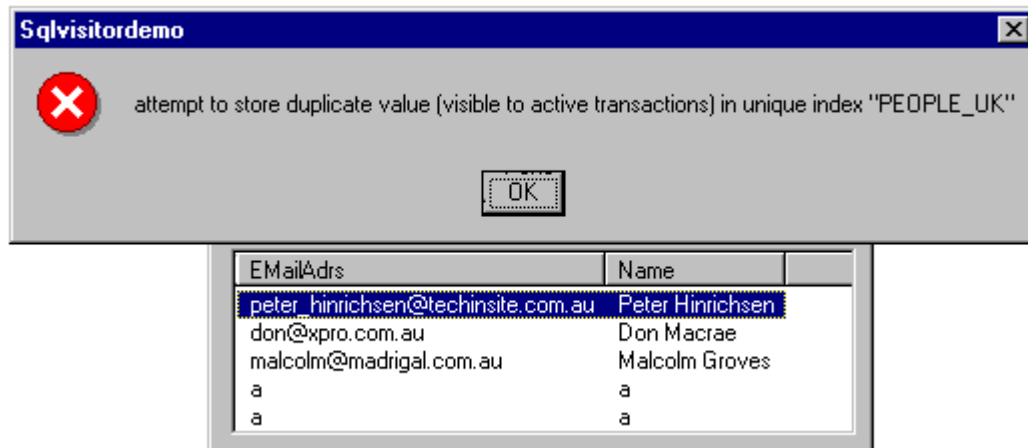
Primary Key ( OID ) ;

create unique index People_uk on People ( name, EMailAdrs ) ;

insert into People values ( 1, "Peter Hinrichsen",
"peter hinrichsen@techinsite.com.au" );
insert into People values ( 2, "Don Macrae", "don@xpro.com.au" ) ;
insert into People values ( 3, "Malcolm Groves", "malcolm@madrigal.com.au" ) ;

```

Run the application and deliberately try to insert duplicate name records to see how the framework handles a database error. Add two duplicate records then click 'Save to Interbase'. You will get unique key error like the one shown below:



Click the 'Read from Interbase' button and you will find that one record was saved, while the other was not. The record that was not saved is still in the client with an ObjectState of posCreate. What we clearly want here is some transaction management so either all objects are saved, or none are saved. Before we can setup transaction management, we must modify the framework so all the visitors share the same database connection.

Share the database connection between Visitors

The First step towards providing transaction support is to change the framework so all calls to the database are funneled through the same database connection and transaction object. You will remember that the TVisSQLAbs class owned an instance of a TIBDatabase and TIBTransaction. This meant that every visitor was processed within its own database connection and transaction. Transactions would be implicitly started and committed by Delphi around each SQL statement. What we want is for all visitors to share the same database connection, and for the Visitor Manager to have control over the database transaction.

To achieve this, we will do four things:

1. Move the database and transaction objects out of the TVisSQLAbs class and wrap them up in an object we will call TtiDatabase. (This will become especially useful when we start work on building a swappable persistence layer. Also, one of the slowest things you can do to a database is connect to it, so sharing a database connection between visitor will improve the application's performance.)
2. Modify the TVisSQLAbs class with a Database property. The SetDatabase method will assign a field variable for later reference, and hook the Query object up to the database connection at the same time.
3. Create a single instance (Singleton pattern) of the database object that has application wide visibility.
4. Modify the Visitor Manager so it sets each Visitors Database property before executing, and then clears it when done.

These four steps are detailed below.

Firstly, we will move the database and transaction objects into their own class. The interface of TtiDatabase is shown here:

```
TtiDatabase = class( TObject )
private
  FDB : TIBDatabase ;
  FTransaction : TIBTransaction ;
public
  constructor Create ;
  destructor Destroy ; override ;
  property DB : TIBDatabase read FDB ;
end ;
```

And the implementation of TtiDatabase is shown here:

```
constructor TtiDatabase.Create;
begin
  inherited ;
  FDB := TIBDatabase.Create( nil ) ;
  FDB.DatabaseName := 'C:\TechInsite\OPFPresentation\Source\3 SQLVisitors\Test.gdb' ;
  FDB.Params.Add( 'user_name=SYSDBA' ) ;
  FDB.Params.Add( 'password=masterkey' ) ;
  FDB.LoginPrompt := False ;
  FDB.Connected := True ;
  FTransaction := TIBTransaction.Create( nil ) ;
  FTransaction.DefaultDatabase := FDB ;
  FDB.DefaultTransaction := FTransaction ;
end;

destructor TtiDatabase.Destroy;
begin
  FTransaction.Free ;
  FDB.Free ;
  inherited;
end;
```

As well as moving the database connection out of the Visitor class, and allowing it to be shared between all visitors in the application, we have wrapped the TIBDatabase component in our own code. This is an important step towards using the Adaptor pattern to make our framework independent of any one-database vendor's API.

Next, we modify the TVisSQLAbs class. The owned database and transaction objects are removed and a field variable is added to hold a pointer to the shared database object. A database property with a SetDatabase method is added and SetDatabase is responsible for hooking the query object up to the database connection. The interface of TVisSQLAbs is shown below:

```
TVisSQLAbs = class( TVisPerObjAwareAbs )
private
  FDatabase: TtiDatabase;
  procedure SetDatabase(const Value: TtiDatabase);
public
  constructor Create ; override ;
  destructor Destroy ; override ;
  property Database : TtiDatabase read FDatabase write SetDatabase ;
end ;
```

The implementation of TVisSQLAbs.SetDatabase is shown below. Notice how there is protection against Value being passed as nil.

```
procedure TVisSQLAbs.SetDatabase(const Value: TtiDatabase);
begin
```



```

FDatabase := Value;
if FDatabase <> nil then
  FQuery.Database := FDatabase.DB
else
  FQuery.Database := nil ;
end;

```

Thirdly we require a globally visible, single instance of the database connection. (This is not how we will ultimately implement a database connection – we will use a thread safe database connection pool, but this is sufficient to get us started.) I use something I call a poor man's Singleton, a unit wide variable hiding behind a globally visible function. This implementation does not come close to providing what a GoF singleton requires, but it does the job and is quick to implement. The code for the single instance database connection is shown below:

```

interface
function gDBConnection : TtiDatabase ;

implementation

var
  uDBConnection : TtiDatabase ;

function gDBConnection : TtiDatabase ;
begin
  if uDBConnection = nil then
    uDBConnection := TtiDatabase.Create ;
  result := uDBConnection ;
end ;

```

The final changes we must make are to extend TVisitorMgr.Execute with some code to set the database property on the visitor before calling TVisitor.Execute. There is a complication here because not all Visitors will have a database property, only those that descend from TVisSQLAbs. As a work around we will check the type of the Visitor, and if it descends from TVisSQLAbs, assume it has a database property and hook it up to the default application database object. We also clear the Visitors database property after the visitor has executed. (This is a bit of a hack and is not how we solve the problem in the framework. We actually have another class called a TVisitorController that is responsible for performing tasks before and after visitors execute. Much more elegant, but rather more complex too.) The modified TVisitorMgr.Execute method is shown below:

```

procedure TVisitorMgr.Execute(const pCommand: string; const pData: TVisited);
var
  i : integer ;
  lVisitor : TVisitor ;
begin
  for i := 0 to FList.Count - 1 do
    if SameText( pCommand, TVisitorMapping( FList.Items[i] ).Command ) then
      begin
        lVisitor := TVisitorMapping( FList.Items[i] ).VisitorClass.Create ;

        try
          if ( lVisitor is TVisSQLAbs ) then
            TVisSQLAbs( lVisitor ).Database := gDBConnection ;

          pData.Iterate( lVisitor ) ;

          if ( lVisitor is TVisSQLAbs ) then
            TVisSQLAbs( lVisitor ).Database := nil ;

        finally
          lVisitor.Free ;
        end ;
      end ;
    end ;
  end;

```

The changes we have made to this section ensure that all database activity is channeled through a single database connection. The next step is to modify TVisitorMgr.Execute so all SQL statements are called within a single database transaction.

Manage transactions

To add transaction support, we must do two things:

1. Expose transaction support through the procedures StartTransaction, Commit and RollBack on the database wrapper class TtiDatabase
2. Extend the Visitor Manager's Execute method with the ability to start a transaction when a group of objects is passed to Execute and commit or roll back the transaction when processing has either finished successfully, or failed.

Firstly, we extended interface of TtiDatabase, with the additional methods StartTransaction, Commit and RollBack as in the code below:

```
TtiDatabase = class( TObject )
private
    FDB : TIBDatabase ;
    FTransaction : TIBTransaction ;
public
    constructor Create ;
    destructor Destroy ; override ;
    property DB : TIBDatabase read FDB ;
    procedure StartTransaction ;
    procedure Commit ;
    procedure Rollback ;
end ;
```

The implementation of TtiDatabase is shown next. You can see that the StartTransaction, Commit and RollBack methods simply delegate the call to the owned TIBTransaction object. The reasons for this shall be looked at in more detail when we study swappable persistence layers and the Adaptor pattern.

```
procedure TtiDatabase.StartTransaction;
begin
    FTransaction.StartTransaction ;
end;

procedure TtiDatabase.Commit;
begin
    FTransaction.Commit ;
end;

procedure TtiDatabase.Rollback;
begin
    FTransaction.Rollback ;
end;
```

Secondly, we extend the TVisitorManager.Execute method transaction support. This means changes in three places. Firstly we start the transaction after assigning the database connection to the visitor's Database property. Next we wrap the call to pData. Iterate(lVisitor) up in a try except block and call RollBack if an exception is raised. We re-raise the exception after calling RollBack so it will bubble to the top of any exception handling code we have added to our application. The final change is to add the code to call commit that is executed when the call to pData. Iterate(lVisitor) is completed successfully. The modified TVisitorMgr.Execute method is shown next:

```
procedure TVisitorMgr.Execute(const pCommand: string; const pData: TVisited);
var
    i : integer ;
    lVisitor : TVisitor ;
begin
```

```

for i := 0 to FList.Count - 1 do
  if SameText( pCommand, TVisitorMapping( FList.Items[i] ).Command ) then
    begin
      lVisitor := TVisitorMapping( FList.Items[i] ).VisitorClass.Create ;

      try
        if ( lVisitor is TVisSQLAbs ) then
          begin
            TVisSQLAbs( lVisitor ).Database := gDBConnection ;
            gDBConnection.StartTransaction ;
            end ;

            try
              pData.Iterate( lVisitor ) ;
            except
              on e:exception do
                begin
                  if ( lVisitor is TVisSQLAbs ) then
                    begin
                      gDBConnection.Rollback ;
                      TVisSQLAbs( lVisitor ).Database := nil ;
                    end ;
                    raise ;
                  end ;
                end ;
              end ;

              if ( lVisitor is TVisSQLAbs ) then
                begin
                  gDBConnection.Commit ;
                  TVisSQLAbs( lVisitor ).Database := nil ;
                end ;
              end ;

            finally
              lVisitor.Free ;
            end ;

          end ;
        end ;
      end ;
    end ;
  end ;
end;

```

This transaction strategy will ensure that either all of the objects are saved to the database, or none are saved to the database and will guarantee that the database is never left in an unstable state. There is one further feature to add which will prevent a Visitor's Final method from being called until the database transaction is successfully committed. At the moment, Final is called after the query has executed. This means that the following sequence of events is possible:

```

Object_1 Run SaveSQL
Object_1 Call Final and set ObjectState to posClean
Object_2 Run SaveSQL Raise an exception and roll back database.

```

This will leave the Object_1.ObjectState property incorrectly set to posClean. This is a little tricky to implement and is managed by the TvisitorController class which is a feature of the framework code you can download. We shall discuss the theory of how this was done in a future chapter.

Summary

We have covered a lot of material in this chapter with the main emphasis on extending the Visitor framework to support persistence to a SQL database. We achieved this by creating an application wide database object (which will be moved to a database connection pool in a future chapter), and adding a TQuery object that is owned by the Visitor. We implemented a structured sequence of tasks in the abstract SQL Visitor's Execute method that was inspired by GoF's Template Method Pattern. We created abstract SQLRead and SQLUpdate Visitors, then implemented concrete classes to manage the READ, CREATE, UPDATE and DELETE SQL calls that are necessary to persist our TPerson class. We added a debug log trace to the Visitor framework to help the programmer keep track of what was going on inside the tight looping caused by the TVisited's Iterate method. We explored the idea of adding a Dirty property to the TPerObjAbs class which can be used to enable or disable a save button on a form, then finally implemented transaction support within the TVisitorMgr.Execute method so a group of objects is either all saved, or not saved at all.

The next chapter

In the next session we shall extend the abstract business object model with additional methods and properties as well as modify the Iterate method so it will allow us to iterate over more complex hierarchies of objects.

Chapter #4

Building an abstract business object model (BOM) with the Composite Pattern

The aims of this chapter

In the previous two chapters we investigated how the Visitor and Template Method patterns can be used together to manage objects that are saved to either a custom text file, or a relational database. In these chapters, we started to develop an abstract collection class, and abstract business object class and we will extend these classes, adding more of the functionality that will be required in a complex business system.

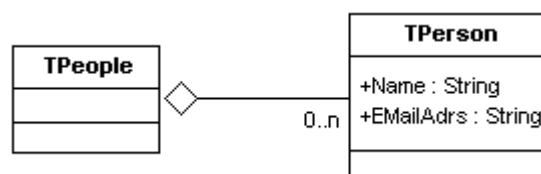
Prerequisites

This chapter builds on the concepts introduced in chapter #2 'The Visitor Framework' so it will be a good idea to read this chapter first.

The business problem we will work with as an example

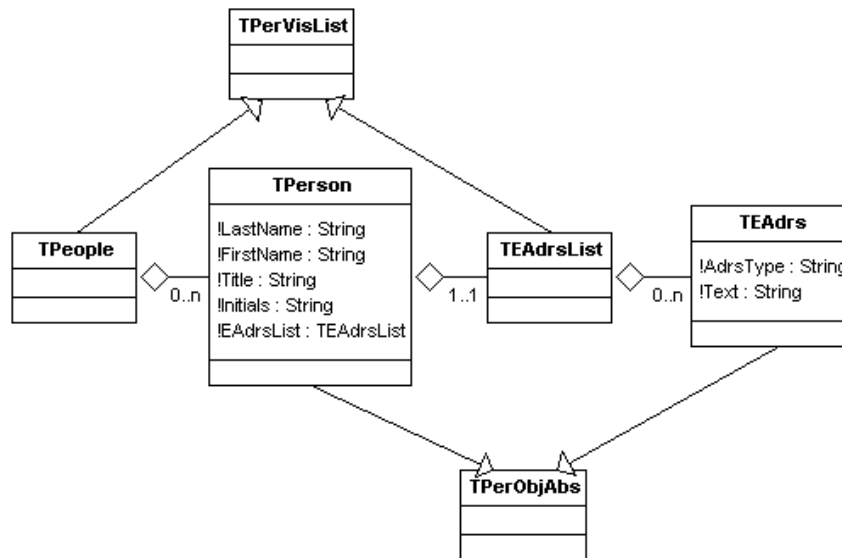
The previous business object model

We will continue to develop the contact management application, and will extend the business object model to represent the data more realistically. In the previous chapter, we created a TPerson class and its corresponding collection class TPeople. This class diagram looked like this:



Rework the address book BOM to better represent the business domain

We shall do several things to improve this business object model. Firstly, the properties that a TPerson can have shall be replaced by the more usual LastName, FirstName, Title and Initials. Secondly, instead of each person only being able to have one EMail address, we shall allow them to have a list of what we will call E-Addresses. An E-Address shall be a one-line address that can contain a phone number, fax number, email address or web URL. (The example application 'DemoTIPerFramework' extends this model further so people can have street or post office box addresses too.) The UML of the extended TPerson family of classes looks like this:



From this diagram we can see four classes:

Class	Description
TPeople	A list of TPerson(s). Descends from TPerVisList and implements list management methods like <code>GetItems(I) : TPerson</code> and <code>Add(pData : TPerson)</code>
TPerson	A person object that descends from TPerObjAbs. Owned by TPeople. Has published properties for LastName, FirstName, Title and Initials. Has a property EAdrsList that is of type TEAdrsList to hold a list of E-Addresses.
TAdrsList	A container for TEAdrs objects, descends from TPerVisList
TEAdrs	An electronic address object that belong to the TPerson. Descends from TPerObjAbs and has properties Address type and address text.

We shall concentrate on implementing a useful abstract business object and business object list first, then implement concrete classes towards the end of this chapter. After we have implemented the class structure, we shall write some helper functions that use the features in the abstract business objects to output the class hierarchy as text for debugging, or to search a hierarchy for an object with a given set of properties..

The Composite pattern – what GoF say

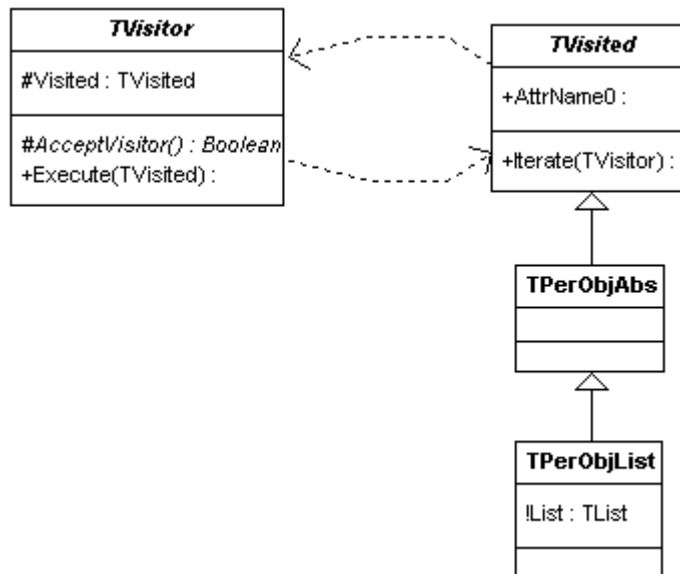
The intent of the Composite, as quoted from 'Design Patterns': Compose objects into tree structures to represent whole-part hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

Implementing the abstract business object model

A type safe relationship between the collection and business objects

The first thing we shall do is to establish the relationship between the abstract collection class and the abstract business object class. The Composite pattern has introduced us to the idea of treating list

objects and regular business objects as the same type of class, so we shall continue with the idea of the collection class descending from the business object class. The UML representing the relationship, which we setup in a pervious chapter looks like this:

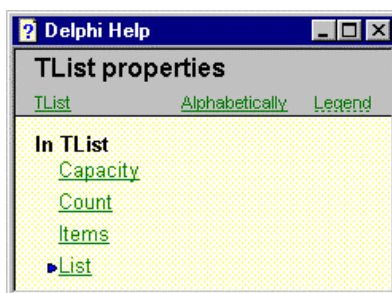


The Visitor – Visited relationship was discussed in chapter #2. Our abstract persistent objects will descend from TVisited because we want to be able to pass visitors over collections of objects with as little work as possible from the programmer. We will be focusing on the relationship between TPerObjAbs and TPerObjList next, but first we will add the list like behavior to TPerObjList.

Adding list behavior to TPerObjAbs

The first question is what behavior should we be adding to TPerObjAbs to turn it into a useful collection class? Delphi’s TList help text provides a good starting point as it details the properties and methods of the TList class. These are shown below:

Properties

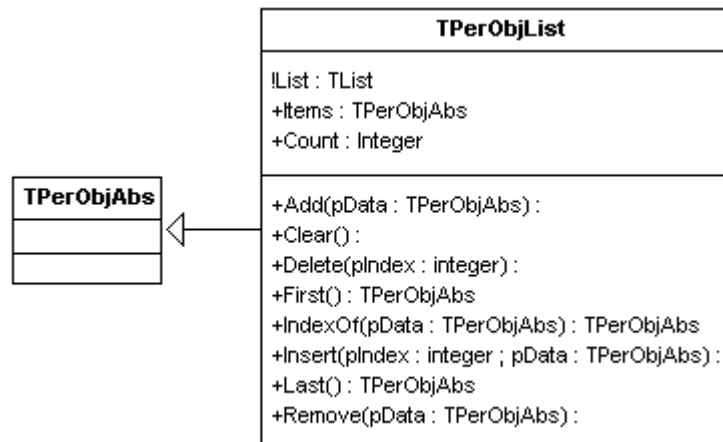


Methods



Of these, we will start by adding the properties Items and Count, and the methods Add, Clear, Delete, First, IndexOf, Insert, Last and Remove. To make it as easy as possible for developers who are new to

the framework, we will give the methods the same signature (or parameters and return type) as in Delphi's TList class, with one change: Where the TList takes a TObject or Pointer as a parameter, we will substitute a TPerObjAbs. Here is the class diagram:



The interface of TPerObjList is shown below:

```

TPerObjList = class( TPerObjAbs )
private
  FList : TObjectList ;
  function   GetList: TList;
protected
  function   GetItems(pIndex: integer): TPerObjAbs; virtual ;
  procedure SetItems(pIndex: integer; const Value: TPerObjAbs); virtual ;
  function   GetCount: integer; virtual;
public
  // Constructor & Destructor
  constructor Create ; override ;
  destructor  Destroy ; override ;

  // Public properties
  property   List : TList read GetList ;
  property   Items[ pIndex : integer ] : TPerObjAbs read GetItems write SetItems ;
  property   Count : integer read GetCount ;

  // Public methods
  procedure  Add( pData : TObject ) ; virtual ;
  procedure  Clear ; virtual ;
  procedure  Delete( pIndex : integer ) ; virtual ;
  function   First : TPerObjAbs ; virtual ;
  function   IndexOf( pData : TPerObjAbs ) : integer ; virtual ;
  procedure  Insert( pIndex : integer ; pData : TPerObjAbs ) ; virtual ;
  function   Last : TPerObjAbs ; virtual ;
  procedure  Remove( pData : TPerObjAbs ) ; virtual ;

  // The Iterate method is still overridden here as we are using the code
  // base we developed in the earlier chapter on the Visitor
  procedure  Iterate( pVisitor : TVisitor ) ; override ;

end ;
  
```

The implementation of TPerObjList is rather dull as each method simply delegates the work to the owned TObjectList (with some type casting as necessary) like this:

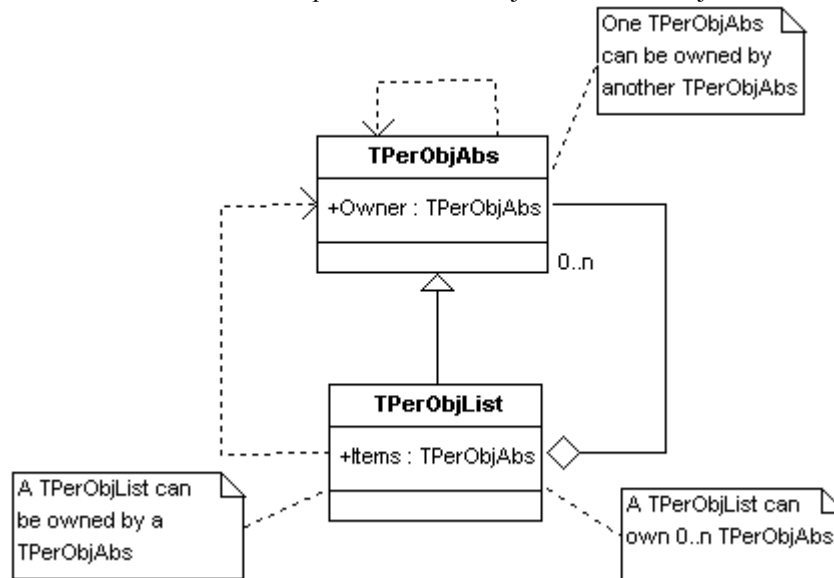
```

function TPerObjList.Last: TPerObjAbs;
begin
  result := TPerObjAbs( FList.Last ) ;
end;
  
```


We shall add one extra property to TPerObjAbs called 'owner'. Owner will behave in much the same way that Delphi's TComponent.Owner property behaves. When we add an object to a TPerObjList, we shall take the opportunity to set its owner property as shown below:

```
procedure TPerObjList.Add(pData: TObject);
begin
  FList.Add( pData );
  pData.Owner := Self ;
end;
```

The details of the relationship between TPerObjAbs and TPerObjList are shown in the UML below.



This diagram tells us 3 things.

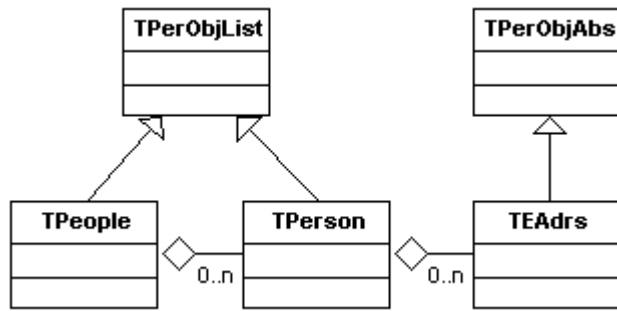
1. TPerObjList inherits from TPerObjAbs. This means we can treat both classes in the same way. This has significant benefits when we start building concrete business classes.
2. TPerObjList maintains an internal list of TPerObjAbs. This list can contain 0 or many instances of TPerObjAbs. The owned instances of TPerObjAbs are indexed by the Items property.
3. TPerObjAbs has a property Owner of type TPerObjAbs. If a TPerObjList owns an instance of TPerObjAbs, the owner property of the TPerObjAbs will have been set to the instance of TPerObjList in its Add method. If a TPerObjAbs owns another TPerObjAbs, the programmer will have to set the Owner property manually in code.

The need to make the iterate method generic

Now that we have coded a firm relationship between TPerObjAbs and TPerObjList, we can look at how to make the iterate method more generic. The implementation of iterate that we developed in chapter #2 looks like this:

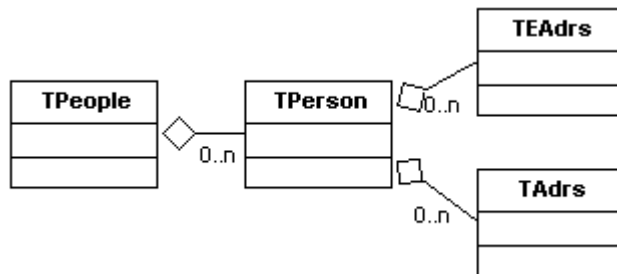
```
procedure TPerObjList.Iterate(pVisitor: TVisitor);
var
  i : integer ;
begin
  inherited Iterate( pVisitor ) ;
  for i := 0 to FList.Count - 1 do
    ( FList.Items[i] as TVisited ).Iterate( pVisitor ) ;
  end;
```

This approach is fine as long as we keep our class hierarchy linear like this:



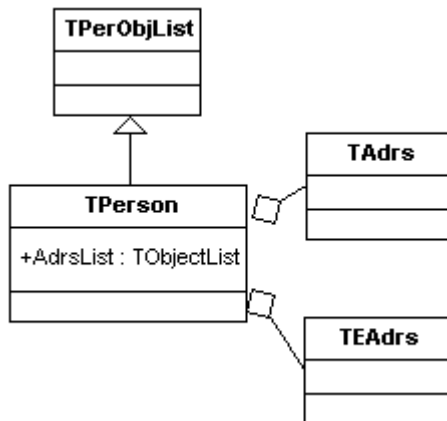
The TPeople own a list of TPerson(s) that own a list of TEAdrs(s). This can be implemented by chaining two TPerObjList classes together, with a TPerObjAbs tacked on the end.

If we call iterate at the top of the tree, passing a visitor like `FPeople.Iterate (TVisDoSomething)` we will be guaranteed of the visitor touching all the elements in the hierarchy. But what if we want to create a class hierarchy like the one shown below, where each TPerson owns two lists: one of TEAdrs(s) and one of TAdrs(s)?

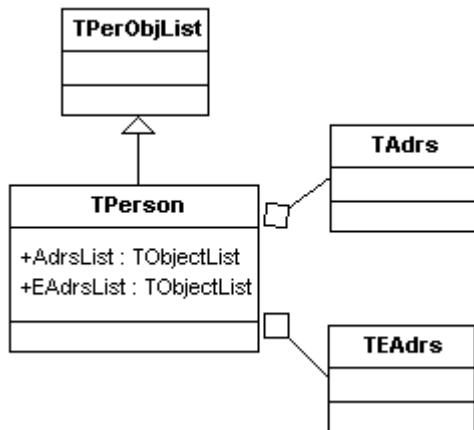


Within our Composite pattern framework, there are three ways of achieving this:

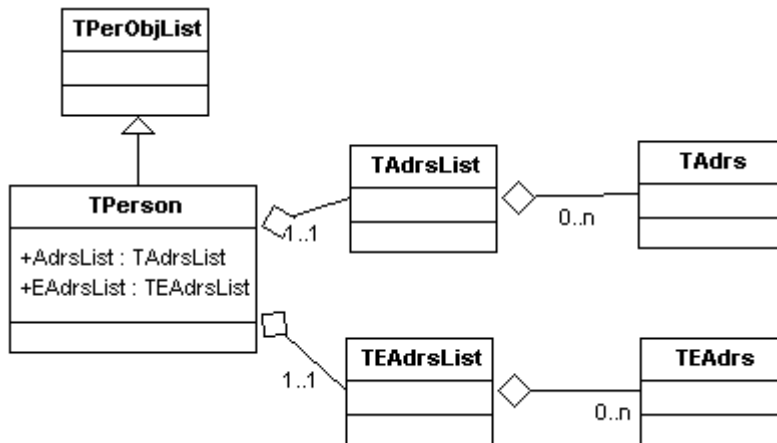
1. TPerson descends from TPerObjList (which gives us a holder for the TEAdrs objects) and we add another TObjList to hold the TAdrs(s), like this:



2. TPerson descends from TPerObjAbs, and we add two TObjList(s); One to hold the TEAdrs(s) and one to hold the TAdrs(s) like this:



3. We create two more classes: TEAdrsList, which holds TEAdrs(s); and TAdrsList, which holds TAdrs(s). We add an instance of TEAdrsList and TEAdrsList to the TPerson class, like this:



I tend to use (3) in real life systems because it allows me to tightly couple each list to the classes it will hold. This makes for a much more robust application from a programmer's point of view.

In the following examples, we shall start by writing a Visitor to convert the objects to text so it is easier to look inside the class hierarchy for debugging. Next, shall implement (2) because it is an easier way to get started then move onto implementing (3) as we fine tune our iterate functionality.

Delphi versions

Before going any further, a word about Delphi versions. Most of the code we are going to look at over the next few sections was developed in Delphi 5 and uses some procedures that were added to TypInfo.pas with the release of version 5. These procedures make it much easier to use RTTI than in previous Delphi versions so the examples will not work with Delphi four or below. At the time of writing, the framework has not been tested with Delphi 6, however other users have reported the need to change some unit names to get the code compiling.

A Visitor to show a class hierarchy as text for debugging

Our aim is to write a generic iterate method, but to test this method and help with debugging, we will write two helper procedures, and a Visitor to output a TPerObjAbs hierarchy as text.

The helper functions we want have a signature like this:

```

// Convert a TPerObjAbs hierarchy to a string
function tiPerObjAbsAsString( pVisited : TPerObjAbs ) : string ;

```

```
// Show a TPerObjAbs hierarchy
procedure tiShowPerObjAbs( pVisited : TPerObjAbs ) ;
```

We shall create a text output Visitor that will have an interface like this:

```
TVisPerObjToText = class( TVisitor )
private
    FStream: TStringStream;
protected
    function AcceptVisitor : boolean ; override ;
public
    constructor Create ; override ;
    destructor Destroy ; override;
    procedure Execute( pVisited : TVisited ) ; override ;
    property Stream : TStringStream read FStream ;
end ;
```

TVisPerObjToText has the usual AcceptVisitor and Execute methods that we have now come to expect to find in a TVisitor descendent, along with an owned TStringStream to write our data out to.

The key method is Execute and its implementation looks like this:

```
procedure TVisPerObjToText.Execute( pVisited: TVisited);
var
    i : integer ;
    lslProps : TStringList ;
    lsValue : string ;
    lsPropName : string ;
begin
    inherited Execute( pVisited ) ;
    if not AcceptVisitor then
        Exit ; //==>

    // Write out the class name of the object we are visiting.
    FStream.WriteString( Visited.ClassName + #13 + #10 ) ;

    // Create a string list to hold a list of property names
    lslProps := TStringList.Create ;
    try
        // Populate the string list, lslProps with a list of published
        // property names read from pVisited. The properties should be
        // simple data types like string or integer only.
        // No class properties at this stage.
        tiGetPropertyNames( TPersistent( pVisited ),
                            lslProps,
                            ctkSimple + [tkVariant, tkEnumeration] ) ;

        // Scan the list of property names and write out the values for each one
        for i := 0 to lslProps.Count - 1 do begin
            // Get the property name from the list
            lsPropName := lslProps.Strings[i] ;
            // Get the property value (the third parameter means we want a string)
            lsValue := GetPropValue( pVisited, lsPropName, true ) ;
            lsValue := ' ' +
                       lslProps.Strings[i] +
                       ' = ' +
                       lsValue ;

            // Write out the property name and value
            FStream.WriteString( lsValue + #13 + #10 ) ;
        end ;
    finally
        lslProps.Free ;
    end ;
end;
```

A note about GetPropValue and SetPropValue:

There are several things going on in this Execute method. Firstly we write out the class name of the object being visited. Next we create a TStringList to hold a list of published property names read from the class being visited. We then call tiGetPropertyNames to populate this list. tiGetPropertyNames is central to everything we shall do with iteration from now on and we will look at its implementation in just a second. Once we have a list of property names, we scan the list and write `name = value` pairs for each property and its corresponding value.

To create and execute this visitor we have the two helper functions tiPerObjAbsAsString and tiShowPerObjAbs. The implementation of tiPerObjAbsAsString looks like this:

```
function tiPerObjAbsAsString( pVisited : TPerObjAbs ) : string ;
var
  lVis : TVisPerObjToText ;
begin
  lVis := TVisPerObjToText.Create ;
  try
    pVisited.Iterate( lVis ) ;
    result := lVis.Stream.DataString ;
  finally
    lVis.Free ;
  end ;
end ;
```

An instance of TVisPerObjToText is created then passed to the iterate method of the object at the top of the class hierarchy we want to display. The visitor writes out its data to the TStringStream then we read this information and pass it back as the function's result.

The second helper function tiShowPerObjAbs is used to call tiPerObjAbsAsString and display its value. The implementation of tiShowPerObjAbs looks like this:

```
procedure tiShowPerObjAbs( pVisited : TPerObjAbs ) ;
begin
  ShowMessage( tiPerObjAbsAsString( pVisited ) ) ;
end ;
```

The data we stored in an Interbase database in the previous section looks like this when it is displayed



The implementation of these two functions in the tiOPF uses the same concepts, but it indents values so it is easier to see which class owns what. The data is shown using a dialog with a scrolling text region (with a fixed with font) because some object hierarchies can be very big and overflow when displayed with ShowMessage.

Getting a list of property names with tiGetPropertyNames

The method tiGetPropertyNames is fundamental to the core of the tiOPF. If you are an interface zealot, and you are offended at the concept of using Delphi's Run Time Type Information (RTTI) at the heart of the framework, the now is a good time to stop reading. From now on, we will be using tiGetPropertyNames everywhere. For example, RTTI is used for these purposes:

- At the lowest level of the hierarchy to control the iteration over owned objects
- In the TtiListView for reading what columns of data to display
- In the TtiTreeView for reading nested objects and the data to display
- In the TtiPerAware edit controls for linking a TPerObjAbs up to the control
- In the TPerObjAbs's Assign and Clone methods which are used for copying objects
- In the class-to-database mapping framework which lets us read complex hierarchies of objects without writing any SQL

There are other ways of achieving this than RTTI, however Delphi makes all this easy with RTTI so why not use it?

tiGetPropertyNames is overloaded and the two possible interfaces are shown below:

```
procedure tiGetPropertyNames( pPersistent : TPersistent ;
                             pSL : TStringList ;
                             pPropFilter : TTypeKinds = ctkSimple ) ; overload ;

procedure tiGetPropertyNames( pPersistent : TPersistentClass ;
                             pSL : TStringList ;
                             pPropFilter : TTypeKinds = ctkSimple ) ; overload ;
```

Three parameters are passed: pPersistent which is either an instance of a TPersistent, or a TPersistent class type, a string list to hold the resulting property names and a property filter which contains a list of the property types to be read into the string list.

```
procedure tiGetPropertyNames( pPersistent : TPersistentClass ;
                             pSL : TStringList ;
                             pPropFilter : TTypeKinds = ctkSimple ) ;
var
  lCount : integer ;
  lSize : integer ;
  lList : PPropList ;
  i : integer ;
begin
  lCount := GetPropList(pPersistent.ClassInfo, pPropFilter, nil);
  lSize := lCount * SizeOf(Pointer);
  GetMem(lList, lSize);
  try
    GetPropList(pPersistent.ClassInfo, pPropFilter, lList);
    for i := 0 to lcount - 1 do
      psl.add( lList[i].Name ) ;
    finally
      FreeMem( lList, lSize ) ;
    end ;
  end ;
end ;
```

Now I am happy to confess that I don't totally understand this code, except to say that GetPropList is declared in the TypInfo.pas unit. If you search the Delphi 5 VCL code you will find it called only once in the entire VCL in DsgnIntf.pas and its this code that I used to work out how to read a list of property names.

```
constructor TPropInfoList.Create(Instance: TPersistent; Filter: TTypeKinds);  
begin  
    FCount := GetPropList(Instance.ClassInfo, Filter, nil);  
    FSize := FCount * SizeOf(Pointer);  
    GetMem(FList, FSize);  
    GetPropList(Instance.ClassInfo, Filter, FList);  
end;
```

I don't understand it, but it does work and works well.

The third parameter passed to `tiGetPropertyNames` is a set of `TTypeKind`. A `TTypeKind` can be any of the following:

```
TTypeKind = (tkUnknown, tkInteger, tkChar, tkEnumeration, tkFloat,  
            tkString, tkSet, tkClass, tkMethod, tkWChar, tkLString, tkWString,  
            tkVariant, tkArray, tkRecord, tkInterface, tkInt64, tkDynArray);
```

In `tiUtils.pas` I have declared the following constants to make using `tiGetPropertyNames` easier:

```
const  
    // Type kinds for use with tiGetPropertyNames  
    // All string type properties  
    ctkString = [ tkChar, tkString, tkWChar, tkLString, tkWString ] ;  
    // Integer type properties  
    ctkInt     = [ tkInteger, tkInt64 ] ;  
    // Float type properties  
    ctkFloat   = [ tkFloat ] ;  
    // Numeric type properties  
    ctkNumeric = [tkInteger, tkInt64, tkFloat];  
    // All simple types (string, int, float)  
    ctkSimple  = ctkString + ctkInt + ctkFloat ;
```

This makes calls to `tiGetPropertyNames` like this possible:

```
// Populate lslProps with a list of string type properties  
tiGetPropertyNames( FMyData, lslProps, ctkString ) ;  
  
// Populate lslProps with a list of numeric type properties  
tiGetPropertyNames( FMyData, lslProps, ctkNumeric ) ;  
  
// Populate lslProps with a list of properties that are not objects or methods  
tiGetPropertyNames( FMyData, lslProps, ctkSimple + [tkVariant, tkEnumeration] ) ;  
  
// Populate lslProps with a list of properties that are objects  
tiGetPropertyNames( FMyData, lslProps, [tkClass] ) ;
```

It is the last call, `tiGetPropertyNames(FMyData, lslProps, [tkClass])` that we will use to make our iteration method generic.

A word about `TPersistent` and published properties

Delphi's help tells us this about `TPersistent`:

`TPersistent` encapsulates the behavior common to all objects that can be assigned to other objects, and that can read and write their properties to and from a stream. For this purpose `TPersistent` introduces methods that can be overridden to:

*Define the procedure for loading and storing unpublished data to a stream.
Provide the means to assign values to properties.
Provide the means to assign the contents of one object to another.*

If we take a look inside Classes.pas, we see that the interface of TPersistent looks like this:

```
{ $M+ }

TPersistent = class(TObject)
private
    procedure AssignError(Source: TPersistent);
protected
    procedure AssignTo(Dest: TPersistent); virtual;
    procedure DefineProperties(Filer: TFiler); virtual;
    function GetOwner: TPersistent; dynamic;
public
    destructor Destroy; override;
    procedure Assign(Source: TPersistent); virtual;
    function GetNamePath: string; dynamic;
end;

{ $M- }
```

The {\$M+} compiler directive that surrounds the TPersistent interface looks interesting and we learn from the Delphi help text that it is \$M that turns on RTTI. The help text tells us this about \$M

The \$M switch directive controls generation of runtime type information (RTTI). When a class is declared in the {\$M+} state, or is derived from a class that was declared in the {\$M+} state, the compiler generates runtime type information for fields, methods, and properties that are declared in a published section. If a class is declared in the {\$M-} state, and is not derived from a class that was declared in the {\$M+} state, published sections are not allowed in the class.

Note: The TPersistent class defined in the Classes unit of the VCL is declared in the {\$M+} state, so any class derived from TPersistent will have RTTI generated for its published sections. The VCL uses the runtime type information generated for published sections to access the values of a component's properties when saving or loading form files. Furthermore, the Delphi IDE uses a component's runtime type information to determine the list of properties to show in the Object Inspector.

Now, we have two choices for the parent class of our business objects: TPersistent or TObject with the \$M switch turned on in the classes interface. We have used TPersistent in the tiOPF because the framework code predates my knowledge of the \$M switch. (ADUG member Mat Vincent only introduced me about the existence of the \$M switch a short time long ago.)

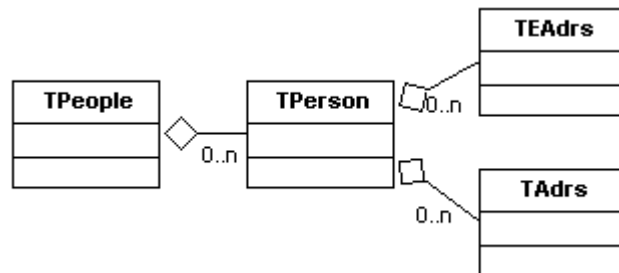
If we look inside TPersistent.Assign, we see that the call is delegated to TPersistent.AssignTo, which in turn delegates the call to TPersistent.AssignError. The implementation of TPersistent.AssignError looks like this:

```
procedure TPersistent.AssignError(Source: TPersistent);
var
    SourceName: string;
begin
    if Source <> nil then
        SourceName := Source.ClassName else
        SourceName := 'nil';
    raise EConvertError.CreateResFmt(@SAssignError, [SourceName, ClassName]);
end;
```


So, TPersistent.Assign dose little other than raise an exception if it is not overridden and implemented in a concrete class. We better remember that when it comes time to write a generic Assign method.

Create some concrete objects to iterate over

Before we can go much further with developing a generic iterate method, we better create a test stub to evaluate our progress. We will create some concrete instances of the TPeople – TPerson – TAdrs and TEAdrs objects discussed in the previous section. As a reminder, the UML of the business object model we will implement is shown below:



This will be implemented by descending TPeople form TPerObjList, and TPerson from TPerObjAbs. TPerson will own two TObjectLists to contain the TEAdrs(s) and TAdrs(s). Both these lists will be published to allow RTTI to work. The interface section of the unit that contains this class hierarchy is shown below:

```

TPeople = class( TPerObjList ) ;

TPerson = class( TPerObjAbs )
private
  FName: string;
  EAdrsList : TObjectList ;
  AdrsList : TObjectList ;
  function GetAdrsList: TList;
  function GetEAdrsList: TList;
public
  constructor Create ; override ;
  destructor Destroy ; override ;
published
  property Name : string read FName write FName ;
  property EAdrsList : TList read GetEAdrsList ;
  property AdrsList : TList read GetAdrsList ;
end ;

TAdrs = class( TPerObjAbs )
private
  FSuburb: string;
  FAdrsType: string;
  FCountry: string;
  FLines: string;
published
  property AdrsType : string read FAdrsType write FAdrsType ;
  property Lines : string read FLines write FLines ;
  property Suburb : string read FSuburb write FSuburb ;
  property Country : string read FCountry write FCountry ;
end ;

TEAdrs = class( TPerObjAbs )
private
  FAdrsType: string;
  FText: string;
published
  property AdrsType : string read FAdrsType write FAdrsType ;
  property Text : string read FText write FText ;
end ;
  
```

We shall hard code some test data like this:

```

procedure PopulatePeopleWithHardCodedData( pPeople : TPeople ) ;
var
  lPerson : TPerson ;
  lAdrs   : TAdrs   ;
  lEAdrs  : TEAdrs  ;
begin
  pPeople.List.Clear ;
  lPerson             := TPerson.Create ;
  lPerson.Name       := 'Peter Hinrichsen' ;
  pPeople.Add( lPerson ) ;

  lAdrs := TAdrs.Create ;
  lAdrs.AdrsType := 'Work-Postal' ;
  lAdrs.Lines   := 'PO Box 429' ;
  lAdrs.Suburb  := 'Abbotsford' ;
  lAdrs.Country := 'Australia' ;
  lPerson.AdrsList.Add( lAdrs ) ;

  lAdrs := TAdrs.Create ;
  lAdrs.AdrsType := 'Work-Street' ;
  lAdrs.Lines   := '23 Victoria Pde' ;
  lAdrs.Suburb  := 'Collingwood' ;
  lAdrs.Country := 'Australia' ;
  lPerson.AdrsList.Add( lAdrs ) ;

  lEAdrs := TEAdrs.Create ;
  lEAdrs.AdrsType := 'EMail' ;
  lEAdrs.Text     := 'peter hinrichsen@techinsite.com.au' ;
  lPerson.EAdrsList.Add( lEAdrs ) ;

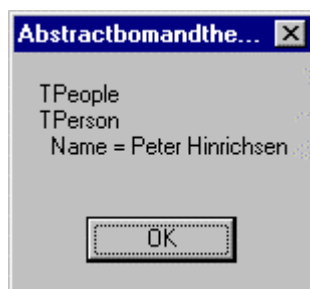
  lEAdrs := TEAdrs.Create ;
  lEAdrs.AdrsType := 'Web' ;
  lEAdrs.Text     := 'www.techinsite.com.au' ;
  lPerson.EAdrsList.Add( lEAdrs ) ;

  lEAdrs := TEAdrs.Create ;
  lEAdrs.AdrsType := 'Phone' ;
  lEAdrs.Text     := '+61 3 9419 6456' ;
  lPerson.EAdrsList.Add( lEAdrs ) ;

end;

```

We can now test the `tiShowPerObjAbs` procedure and as expected will see the `TPeople` and `TPerson`, but the iterate method will stop at this level because it does not know about the existence of either `AdrsList` or `EAdrsList`. The result of the call to `tiShowPerObjAbs` is shown below:



Override Iterate

One way of solving this problem is to override the `TPerson.Iterate` method like this:

```

procedure TPerson.Iterate(pVisitor: TVisitor);
var
  i : integer ;
begin
  inherited Iterate( pVisitor ) ;

  for i := 0 to FAdrsList.Count - 1 do
    ( FAdrsList.Items[i] as TVisited ).Iterate( pVisitor ) ;
  for i := 0 to FEAdrsList.Count - 1 do
    ( FEAdrsList.Items[i] as TVisited ).Iterate( pVisitor ) ;

```

```
end;
```

This has the desired effect as you can see in the dialog below. All the objects in the hierarchy have been touched by the visitor and have had their 'flat' properties written out for display. In the early versions of the framework this is how I iterated over complex hierarchies, but it was very error prone as it was easy to add another contained class to a parent object and to forget to make the necessary changes to the iterate method.



We shall now use RTTI to detect all the owned (and published) instances of list properties and iterate over those in the abstract visitor class. This makes the abstract visitor's Iterate rather more complex, but means we don't have to remember to override a concrete classes' iterate method each time we add an owned list.

Generically detecting and iterating over an owned TList

If we take another look at the method `tiGetPropertyNames` we see that the third parameter is an array (or set) of property types. You will recall that in `TypeInfo.pas`, one of the possible values a property type kind can take is `tkClass`. So, we will use `tiGetPropertyNames` and pass `tkClass` as a parameter so a list of class type property names are returned. We shall then iterate through the list of property names and if the class property is a `TList` descendent, then scan the list elements and call `Iterate` on each one. This way we are one step closer to automating the iteration process. The implementation of `TVisited.Iterate` is shown below:

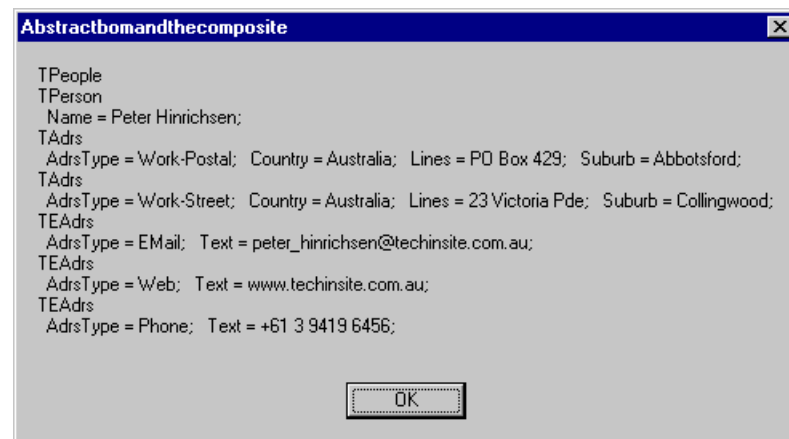
```
procedure TVisited.Iterate(pVisitor: TVisitor);
var
  lClassPropNames : TStringList ;
  i                : integer ;
  j                : integer ;
  lVisited        : TObject ;
  lList           : TList ;
begin
  pVisitor.Execute( self ) ;
  // Create a string list to hold the property names
  lClassPropNames := TStringList.Create ;
  try
    // Get all property names of type tkClass
    tiGetPropertyNames( self, lClassPropNames, [tkClass] ) ;
    // Scan through these properties
    for i := 0 to lClassPropNames.Count - 1 do begin
      // Get a pointer to the property
      lVisited := GetObjectProp( self, lClassPropNames.Strings[i] ) ;
      // If the property is a TList, then visit it's items
      if (lVisited is TList) then
        begin
```

```

lList := TList( lVisited ) ;
for j := 0 to lList.Count - 1 do
    if ( TObject( lList.Items[j] ) is TVisited ) then
        TVisited( lList.Items[j] ).Iterate( pVisitor ) ;
    end ;
end ;
finally
    lClassPropNames.Free ;
end ;
end;

```

This code give's the desired results of iterating over all the elements contained in published TList(s) in the object hierarchy. We can verify this buy running tiShowPerObjAbs and checking that the output matches the screen shot below:



Lets quickly walk through the code and see how it works. First `pVisitor.Execute(self)` is called to ensure that the current object is passed to the Visitor. Next, we create an empty string list and populate it with the names of any class type published properties on the object we are currently visiting (or self). We now scan this list of class type property names and call `GetObjectProp(self, <PropName>)` to get a pointer to the published class property. We check if this object is a TList and if it is, scan it for instances of TVisited. When we find an instance of TVisited in the list, we call `TVisited.Iterate(pVisitor)` an pass the Visitor we are currently working with. We repeat this process for each class type published property, and when done clean up by freeing the resources we used along the way.

Since writing this generic iteration routine the number of iteration related errors in my programming has dropped to almost zero (before developing this approach, I kept making mistakes because I would add another owned list to an object, and forget to override its owner's iterate method. This has had a significant payback in saved developer time.

Now that we have developed an approach for iterating over a class that owns 0..n TLists, we shall extend the logic to iterate over a class that owns 0..n TPerObjAbs(s).

Iterating when objects own objects (rather than lists owning objects)

Why would we want to iterate over a class that owns 0..n TPerObjAbs? Take a look at how we modelled a person as a TPerson:

```

TPerson = class( TPerObjAbs )
published
    property Name : string read FName write FName ;
    property EAdrsList : TList read GetEAdrsList ;
    property AdrsList : TList read GetAdrsList ;
end ;

```

and lets say we want to extend our class hierarchy with a company class represented like this:

```

TCompany = class( TPerObjAbs )
published
  property CompanyName : string read FName write FName ;
  property EAdrsList : TList read GetEAdrsList ;
  property AdrsList : TList read GetAdrsList ;
end ;

```

And we want to add some code to search for, say, a person's or companies E-mail address from EAdrsList. One way would be to add a FindEmailAddress : TEAdrs method to both the TPeople and TCompany classes, but this would mean duplication of code. Another approach would be to descend both TPerson and TCompany from the same parent, but this may not be possible for a variety of reasons. Another solution to this problem is to use object containment. We can create custom TList descendent (or in our case, TPerObjList descendent) and add the special FindEmailAddress code there. That way, wherever we use an instance of TEAdrs, we have access to the FindEMailAdrs method. This is easy to do and I'm sure you have used this technique before, the only complication is how to automatically iterate over this modified object hierarchy. Lets say we refactor the class hierarchy as shown in the code snip below:

```

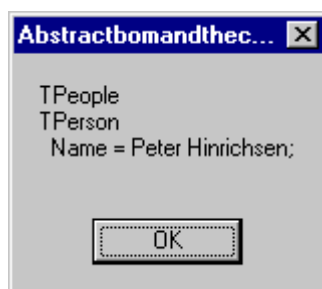
TEAdrsList = class( TPerObjList ) ;
TEAdrs = class( TPerObjAbs )
published
  property AdrsType : string read FAdrsType write FAdrsType ;
  property Text : string read FText write FText ;
end ;

TAdrsList = class( TPerObjList ) ;
TAdrs = class( TPerObjAbs )
published
  property AdrsType : string read FAdrsType write FAdrsType ;
  property Lines : string read FLines write FLines ;
  property Suburb : string read FSuburb write FSuburb ;
  property Country : string read FCountry write FCountry ;
end ;

TPerson = class( TPerObjAbs )
published
  property Name : string read FName write FName ;
  property EAdrsList : TEAdrsList read FEAdrsList ;
  property AdrsList : TAdrsList read FAdrsList ;
end ;

```

We have introduced a list object for both TAdrs and TEAdrs. This will give us fine grain control over the list management which we did not have by using a TObjectList as the container. The TPerson class now has an instance of TAdrsList and TEAdrsList, instead of two instances of TObjectList. As expected, when we call tiShowPerObjAbs passing an instance of TPeople, we get the following result:



Which is what we expect, but not what we want because Iterate is not detecting the instances of TAdrsList and TEAdrsList, owned by TPerson. We must extend TVisited.Iterate to detect published properties of types other than TList. The new implementation of Iterate is shown below:

```

procedure TVisited.Iterate(pVisitor: TVisitor);
// Take a TList and a TVisitor, and call TVisited( Items[i] ).Iterate
// on each element of the TList.
procedure _VisitListElements( lVisited : TObject ; pVisitor : TVisitor ) ;

```

```

var
  i : integer ;
  lList : TList ;
begin
  lList := TList( lVisited ) ;
  for i := 0 to lList.Count - 1 do
    if ( TObject( lList.Items[i] ) is TVisited ) then
      TVisited( lList.Items[i] ).Iterate( pVisitor ) ;
  end ;
var
  lClassPropNames : TStringList ;
  i : integer ;
  lVisited : TObject ;
  lList : TList ;
begin
  pVisitor.Execute( self ) ;
  // Create a string list to hold the property names
  lClassPropNames := TStringList.Create ;
  try
    // Get all property names of type tkClass
    tiGetPropertyNames( self, lClassPropNames, [tkClass] ) ;
    // Scan through these properties
    for i := 0 to lClassPropNames.Count - 1 do begin
      // Get a pointer to the property
      lVisited := GetObjectProp( self, lClassPropNames.Strings[i] ) ;
      // If the property is a TList, then visit it's items
      if (lVisited is TList) then
        VisitListElements( lVisited, pVisitor )
      // If the property is a TVisited, then call iterate( pVisitor )
      else if ( lVisited is TVisited ) then
        TVisited( lVisited ).Iterate( pVisitor ) ;
      end ;
    finally
      lClassPropNames.Free ;
    end ;
  end;
end;

```

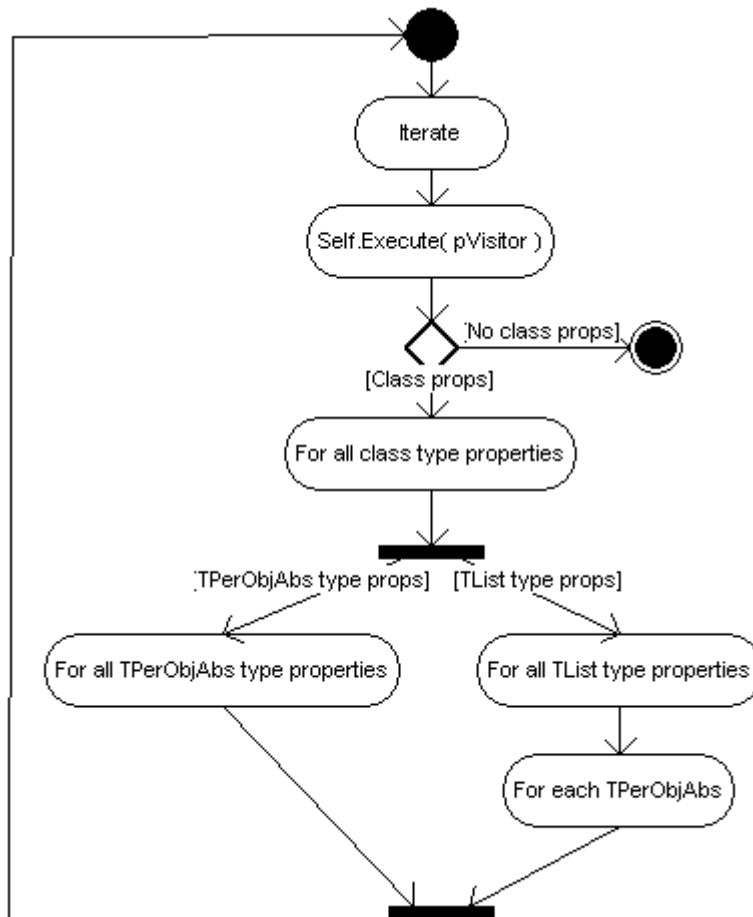
We have made two significant changes to the implementation of Iterate we wrote to automatically detect owned lists: We have moved the list iteration code into a procedure that is private to Iterate, and we have added code to detect owned instances of TVisited. The changes to the object property scanning code look like this:

```

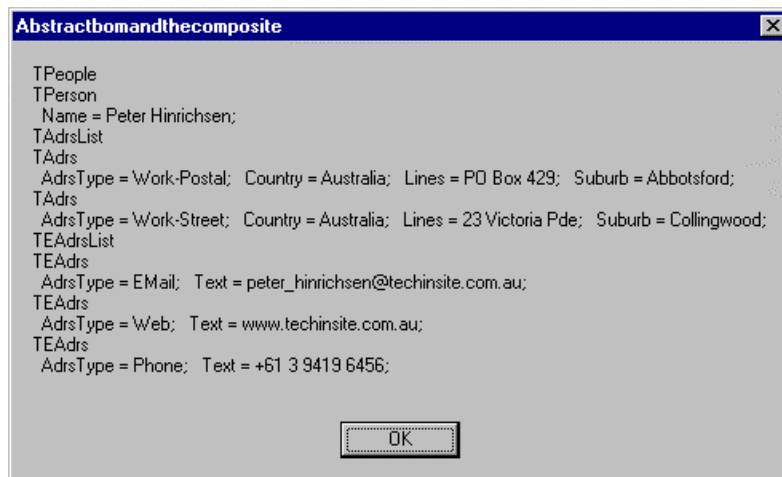
if (lVisited is TList) then
  VisitListElements( lVisited, pVisitor )
// If the property is a TVisited, then call iterate( pVisitor )
else if ( lVisited is TVisited ) then
  TVisited( lVisited ).Iterate( pVisitor ) ;

```

If a class type property is a TList, then `_VisListElements` is called passing the TList and the TVisitor as parameters. This procedure simply iterates over each element of the list and was added to make the code more readable. If the class type property is a TVisited, then its `iterate` method is called. This flow of events is shown in the activity diagram below:



The refactored Iterate method is now producing the expected and desired result:



Notice the slight variation from the version of our application where the TPerson class directly owned two TList(s) rather than two TPerObjList(s): The class name of the list class is being written out which is because the list is having `Execute(Self)` called within its iterate method.

This technique uses a recursive call to the Iterate method and will drill down into a class hierarchy to any depth limited only by the system resources and compiler settings.

Now that we have seen how to use RTTI and recursion to iterate over a composite of objects, we shall extend the abstract base classes with some additional methods like Clone and Assign. These will be used for making duplicate copies of objects so we can edit an object in a buffer. This is necessary if we want to provide an undo option in edit dialogs.

The need for the clone and assign methods

Sometimes we want to copy all the data from one object to another, or make an exact copy of an object so we can implement buffered editing. An example of this is if we want to write a generic pop up dialog box with both <Cancel> and <Save>. There are several ways to achieve this and the one we use in the tiOPF is to clone the object to be edited to a temporary buffer, then edit the buffered object. If the user clicks Save, the data from the buffered object is copied to the real one. If the user clicks Cancel, then the buffered object is simply thrown away. We shall see an example of this in chapter #5, 'A worked example of using the tiOPF'. But before we can look at the worked example, we must see how to create a duplicate copy of an object with the Clone method, and to copy the data from one object to another with the Assign method. We will start by implementing a hard coded assign method, then look at making this method generic so we do not have to implement Assign in every class we write. Next we shall look at writing a Clone method which will return a duplicate copy of an object, then finally look at the issues surrounding calling Assign on an object that owns other objects.

Implementing a hard coded assign method

A hard coded assign method is quite easy to implement, but first, we will build a test stub so we can evaluate our progress. We shall start by assigning the data from one instance of a TAdrs to another, and testing it using the code below:

```
procedure TFormMain_VisitorManager.btnTestAssign(Sender: TObject);
var
  lAdrsFrom : TAdrs ;
  lAdrsTo   : TAdrs ;
  lsAdrsFrom : string ;
  lsAdrsTo   : string ;
begin
  // Create an instance of TAdra to copy from
  lAdrsFrom := TAdrs.Create ;
  lAdrsFrom.AdrsType := 'Work-Postal' ;
  lAdrsFrom.Lines    := 'PO Box 429' ;
  lAdrsFrom.Suburb   := 'Abbotsford' ;
  lAdrsFrom.Country  := 'Australia' ;

  // Create another instance of TAdrs for copying to
  lAdrsTo := TAdrs.Create ;

  // Perform the Assign
  lAdrsTo.Assign( lAdrsFrom ) ;

  // Output the From and To TAdrs(s) to a string
  lsAdrsFrom := tiPerObjAbsAsString( lAdrsFrom ) ;
  lsAdrsTo   := tiPerObjAbsAsString( lAdrsTo ) ;

  // Compare lsAdrsTo with lsAdrsFrom and report on the findings
  if lsAdrsTo = lsAdrsFrom then
    ShowMessage( 'Assign worked' )
  else
    ShowMessage( 'Assign failed' + #13 +
      '|' + lsAdrsFrom + '|' + #13 + #13 +
      '|' + lsAdrsTo + '|' ) ;

end;
```

In this test stub, we create an instance of a TAdrs, then populate it with some dummy values. We create another instance of TAdrs then call `lAdrsTo.Assign(lAdrsFrom)`. We convert the results of both `lAdrsFrom` and `lAdrsTo` into a string then compare the results and report on any differences. The code we have implemented in `TAdrs.Assign` looks like this:

```
procedure TAdrs.Assign(Source: TPersistent);
begin
  AdrsType := TAdrs( Source ).AdrsType;
  Lines    := TAdrs( Source ).Lines;
```



```

Suburb := TAdrs( Source ).Suburb;
Country := TAdrs( Source ).Country;
end;

```

There are several things I don't like about this code. Firstly, we have to remember to copy each property of TAdrs which means our code is that little harder to maintain and slightly more error prone than if the process was automatic. We have seen how to use RTTI to scan through all the published properties, so this would be a good way of implementing a generic assign. The second problem with the code is that the Assign method is introduced in the class TPersistent and assumes we want to assign from a TPersistent to another TPersistent. We really want to assign from a TPerObjAbs to another TPerObjAbs. We must override Assign and change its signature (or parameters) as part of the changes we will be making. This leads us to writing a generic Assign method for the TPerObjAbs class.

Implementing a generic Assign method

We have seen how to use tiGetPropertyNames to populate a string list with a classes' published property names. We can then scan through this list and use the GetPropValue and SetPropValue methods that are found in Delphi 5's TypInfo.pas unit. The code to implement the automated Assign method is shown below:

```

procedure TPerObjAbs.Assign(Source: TPersistent);
var
  lsl : TStringList ;
  i : integer ;
begin
  // Create a string list to hold the published property names
  lsl := TStringList.Create ;
  try
    // Populate the string list with the published property names
    tiGetPropertyNames( self, lsl, ctkSimple ) ;
    // Scan the list of property names
    for i := 0 to lsl.Count - 1 do
      // Call the SetPropValue and GetPropValue methods found in
      // Delphi 5's TypInfo.pas
      SetPropValue( Self,
                    lsl.Strings[i],
                    GetPropValue( Source, lsl.Strings[i] ) ) ;
    finally
      // Clean up
      lsl.Free ;
    end ;
end;

```

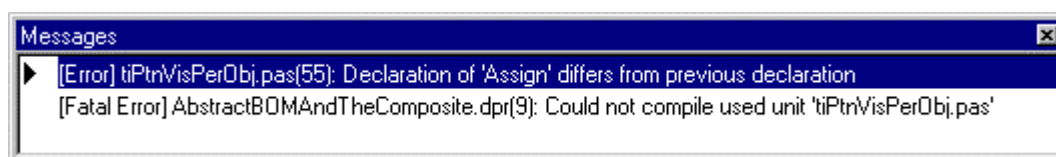
The last change we must make to assign is to change its signature from Assign(Source : TPersistent) to Assign(Source : TPerObjAbs). This appears easy enough to do like this:

```

TPerObjAbs = class( TVisited )
public
  procedure Assign( Source : TPerObjAbs ) ; override ;
end ;

```

but we get the compile time error:



If we remove the override directive like this:

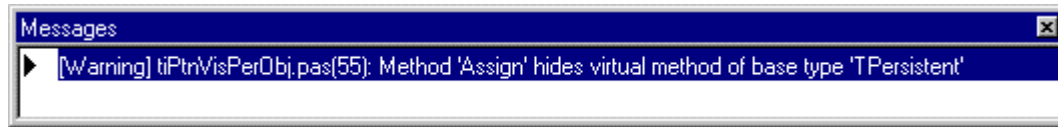
```

TPerObjAbs = class( TVisited )

```

```
public
  procedure Assign( Source : TPerObjAbs ) ;
end ;
```

then we get this compiler warning:



The solution is to use the reintroduce directive like this:

```
TPerObjAbs = class( TVisited )
public
  procedure Assign( Source : TPerObjAbs ) ; reintroduce ;
end ;
```

Re-declaring a method to change its signature by using reintroduce to hide the compiler warning has some side effects. The Delphi help tells this about overriding versus hiding:

If a method declaration specifies the same method identifier and parameter signature as an inherited method, but doesn't include override, the new declaration merely hides the inherited one without overriding it. Both methods exist in the descendant class, where the method name is statically bound.

For example,

```
type
  T1 = class(TObject)
    procedure Act; virtual;
  end;
  T2 = class(T1)
    procedure Act; // Act is redeclared, but not overridden
  end;

var
  SomeObject: T1;
begin
  SomeObject := T2.Create;
  SomeObject.Act; // calls T1.Act
end;
```

So we can safely do this:

```
type
  TPerObjAbs = class( TVisited )
  public
    procedure Assign( Source : TPerObjAbs ) ; reintroduce ;
  end ;

  TMyClass = class( TPerObjAbs ) ;

var
  lObj1 : TPerObjAbs ;
  lObj2 : TPerObjAbs ;
begin
  lObj1 := TMyClass.Create ;
  lObj2 := TMyClass.Create ;
  lObj1.Assign( lObj2 ) ; // This will call Assign on TPerObjAbs as expected
end;
```

```
end ;
```

which is exactly what we do in the framework.

Early on, I was seduced by the idea of redeclaring Clone and Assign in the concrete classes so I did not have to type cast them when I was using these classes in applications. My mistake was to do this:

```
TMyClass = class( TPerObjAbs )
public
    procedure Assign( Source : TMyClass ) ; reintroduce ;
    function Clone : TMyClass ; reintroduce ;
end ;
```

As we will see in the next section, there are times when Assign has to be overridden to implement special behaviour to handle class type properties. In the framework, I have a generic edit dialog that takes an instance of TPerObjAbs as a parameter, makes a clone, lets the user edit the clone, then Assigns the buffer back to the original object. Calling TPerObjAbs.Assign caused the wrong assign to be executed. This was a silly mistake to make but it still took me ages to debug and then understand. So, we can safely change the signature of Assign when subclassing from TPersistent to TVisited to TPerObjAbs, but we cant change the signature of Assign from this level down.

Implementing a generic clone method

Now that we have introduced a generic Assign method, we can have a shot at writing a generic Clone method that will return a copy of the object being cloned. We shall use the same test stub code for evaluating Clone as we did for Assign, except that the call to Assign:

```
// Create another instance of TAdrs for copying to
lAdrsTo := TAdrs.Create ;
// Perform the Assign
lAdrsTo.Assign( lAdrsFrom ) ;
```

will be replaced with a call to clone like this:

```
// Create another instance of TAdrs by calling clone
lAdrsTo := TAdrs( lAdrsFrom.Clone ) ;
```

Notice that we have to type cast the result of Clone to TAdrs in the client code as we can't override Clone in the concrete class. The code that is implemented in Clone looks like this:

```
function TPerObjAbs.Clone: TPerObjAbs;
var
    lClass : TPerObjAbsClass ;
begin
    lClass := TPerObjAbsClass( ClassType ) ;
    result := TPerObjAbs( lClass.Create ) ;
    result.Assign( self ) ;
end;
```

The beauty of this code is that it will generically clone whatever class it is called on. It will also ensure that any code in the concrete classes Create method is called. We can test this code and find that it will work reliably as long as we have correctly implemented Assign on concrete classes that contain object properties.

The problem with assigning object type properties.

Implementing Assign on a class that contains object type properties like TPeople with its properties AdrsList : TAdrsList and EAdrsList : TEAdrsList requires a little more thought. There are times when we will want Assign clone any classes that it owns. This is required for our TPerson class where the TPerson owns the addresses and EAddresses. If the TPerson were associated with another object by a mapping type relationship, rather than an ownership type relationship, we would probably want to copy pointers rather than clone objects. This is summarised below:

- **Clone owned objects:** This is useful when one class owns an instance of another, for example the TPerson class owns instances of TAdrsList and TEAdrsList, therefore it is logical to clone these classes when cloning the TPerson class. The implementation of Clone to achieve this is shown below:

```
procedure TPerson.Clone (pSource: TPerObjAbs);  
begin  
    FEAdrsList := TPerson( pSource ).EAdrsList.Clone;  
    FAdrsList := TPerson( pSource ).AdrsList.Clone;  
end;
```

- **Copy pointers:** This is done when one class has a reference to another class, for example many classes may have references to the shared class. This may happen when you have a TSex object which can either be Male, Female or Unknown. It is possible that an application would have a single instance of a list of TSex objects that are shared among TPerson instances. This could be implemented like this:
-

```
// We have a list of TSex objects that is shared between instances of TPerson(s)
TSex = class( TPerObjAbs )
published
  property TextShort : string read FTextShort write FTextShort ; // 'M', 'F',
etc
  property TextLong  : string read FTextLong  write FTextLong  ; // 'Male',
'Female'
end ;
```

Each TPerson has a pointer to one of the shared TSex objects like this:

```
TPerson = class( TPerObjAbs )
public
  procedure Assign( Source : TPerObjAbs ) ;
published
  property Sex : TSex read FSex write FSex ;
end ;
```

The assign method would be implemented by copying the pointer to the shared TSex object, not by creating a new owned instance of TSex.

```
procedure TPerson.Assign( Source : TPerObjAbs ) ;
begin
  inherited Assign( Source ) ;
  FSex := TPerson( Source ).Sex ;
end ;
```

The challenge is to find an elegant way of implementing these two cases and this is discussed next.

How to assign object property types

The properties that a TPerObjAbs descendant can have will fall into one of three categories:

1. Public properties like OID and ObjectState
2. Published 'Flat' properties with data types like String, Integer, Real, TDateTime or Boolean. These can be copied from one instance of an object to another by using the generic RTTI Assign method we looked at earlier.
3. Class type properties that can be cloned or assigned by either copying a pointer to a shared instanced, or by cloning the class property and creating another instance.

The first step is to refactor the TPerObjAbs class (using the Template Method pattern as a basis) and break Assign up into three steps: AssignPublicProps, AssignPublishedProps and AssignClassProps, with Assign calling the three methods in sequence. This lets us override just the AssignClassProps in any concrete classes that have object type properties. The new interface of TPerObjAbs is shown below:

```
TPerObjAbs = class( TVisited )
protected
  procedure AssignPublicProps(pSource: TPerObjAbs);
  procedure AssignPublishedProps(pSource: TPerObjAbs; pPropFilter: TTypeKinds = []
);
  // You must override this in the concrete if there are class properties
  procedure AssignClassProps(pSource: TPerObjAbs); virtual ;
public
  procedure Assign( pSource : TPerObjAbs ) ; reintroduce ;
end ;
```

And the implementation of Assign, inspired by the Template Method pattern is shown next:

```
procedure TPerObjAbs.Assign(pSource: TPerObjAbs);
begin
  AssignPublicProps( pSource ) ;
  AssignPublishedProps( pSource ) ;
  AssignClassProps( pSource ) ;
  // When you create a concrete class that contains object type properties
  // you will have to override AssignClassProps( ) and implement
  // the necessary behaviour to copy pointers or create new instances
  // of these properties.
end;
```

First of all, Assign calls AssignPublicProps which is simple to implement by hard coding the mapping of properties to the current object (self) from the one being passed as a parameter (pSource) and this is shown below:

```
procedure TPerObjAbs.AssignPublicProps(pSource: TPerObjAbs);
begin
  OID := pSource.OID ;
  ObjectState := pSource.ObjectState ;
  Owner := pSource.Owner ;
end;
```

Next, Assign calls AssignPublishedProps, which is a generic routine that copies all 'flat' or 'simple' property types. We developed this code earlier in the section on implementing a generic Assign method.

Finally, Assign calls AssignClassProps, which contains some code to raise an exception in the abstract class. This will remind the developer of the concrete class that he has forgotten to implement the custom AssignClassProps as required. The implementation of AssignClassProps that can be found in TPerObjAbs looks like this:

```
procedure TPerObjAbs.AssignClassProps(pSource: TPerObjAbs);
begin
  Assert( CountPropsByType( pSource, [tkClass] ) = 0,
    'Trying to call ' + ClassName + '.Assign( ) on a class that contains ' +
    'object type properties. AssignClassProps( ) must be overridden in the ' +
    'concrete class.' );
end;
```

This reminds us that we have to copy pointers, or clone objects in the concrete class like this:

```
procedure TPerson.AssignClassProps(pSource: TPerObjAbs);
begin
  FEAdrsList.Assign( TPerson( pSource ).EAdrsList );
  FAdrsList.Assign( TPerson( pSource ).AdrsList );
end;
```

In TPerson.AssignClassProps, we want to clone the objects, not copy pointers. TPerson creates an owned instance of both TEAdrsList and TAdrsList in its constructor so we do not have to call clone in AssignClassProps. Calling FEAdrsList.Assign() has the same effect as calling FEAdrsList := pSource.Clone here, except that it avoids the possibility of a memory leak.

This ends our discussion on how to assign and clone objects. An example of this in use can be found in the address book application that comes with the tiOPF source code. Next we will look at three helper methods which we use on the TPerObjList and TPerObjAbs to iterate without creating a TVisitor, or to help search for an object with certain properties.

Iterating without creating a TVisitor with ForEach()

We have seen how to use a TVisitor descendent to iterate over all the nodes in a hierarchy of objects that is constructed based on GoF's Composite Pattern. This can be very convenient if you know you want to touch all the objects in a hierarchy, but sometimes the programmer knows he only wants to iterate over the objects in a certain list, and is not interested in touching child objects. This is where a ForEach method becomes useful. For example, say we want to extend our business object model so the TPerson class has an Salary property, and the TPeople class knows how to increase the salary by say, 10% for all the people in the list. The modified interface of TPerson and TPeople might look like this:

```
TPeople = class( TPerObjList )
published
  procedure IncreaseSalary ;
end ;

TPerson = class( TPerObjAbs )
private
  FSalary : real ;
published
  property Salary : real read FSalary write FSalary ;
end ;
```

and the implementation of TPeople.IncreaseSalary looks like this:

```
procedure TPeople.IncreaseSalary;
var
  i : integer ;
begin
  for i := 0 to Count - 1 do
    TPerson( Items[i] ).Salary := TPerson( Items[i] ).Salary * 1.1 ;
  end;
```

There are two things I don' like about this code:

1. We have to manually iterate over the owned objects; and

2. We have to type cast each Items[i] call as a TPerson.

We discussed the problem of having to manually iterate over the elements in a list in chapter #2 'The Visitor Framework' and went to some lengths to understand how to use the Visitor pattern to generically solve this problem. Along the way though, we looked at passing a method pointer to each element in the collection. We shall revisit this approach here as it is simpler to code than the Visitor when we only want to touch the elements in a single list. The Visitor pattern helps us maintain state information as we move from one object to another and as state information is not important here, the method pointer approach shall be ideal.

As discussed in chapter #2, we move the for i := 0 to Count logic into a method on the TPerObjList class and have the specialist business logic in one of the concrete classes. The TPerObjList.ForEach method is shown below:

```
procedure TPerObjList.ForEach(pMethod: TPerObjAbsMethod);
var
  i : integer ;
begin
  for i := 0 to Count - 1 do
    pMethod( Items[i] ) ;
  end;
```

and the modified TPeople class with its two procedures IncreaseSalary, which is public and can be called by a client application, and DoIncreaseSalary which is private and gets called by IncreaseSalary.

```
TPeople = class( TPerObjList )
private
  procedure DoIncreaseSalary( pData : TPerObjAbs ) ;
published
  procedure IncreaseSalary ;
end ;
```

DoIncreaseSalary contains the code to perform the calculation:

```
procedure TPeople.DoIncreaseSalary(pData: TPerObjAbs);
begin
  TPerson( pData ).Salary := TPerson( pData ).Salary * 1.1 ;
end;
```

And IncreaseSalary contains a call to ForEach with DoIncreaseSalary being passed as a parameter.

```
procedure TPeople.IncreaseSalary;
begin
  ForEach( DoIncreaseSalary ) ;
end;
```

This might look like an unnecessarily complex way of achieving something that can be done with a For i := 0 to Count - 1 loop, and for this example it probably is. The ForEach method becomes really useful when you want perform more complex logic on significantly more complex object models of nested objects.

The other problem this example highlighted is the need to typecast each call to Items[I] to a TPerson before we could access any methods that are found in TPerson but not in TPerObjAbs.

Type casting Items[i] in the concrete class

Suppose we want to calculate the average salary of all the TPerson(s) in the TPeople. We could use the ForEach method, although this is not ideal because to calculate an average, we will have to keep a running total and there is no provision for maintaining state in the ForEach method. We write a

TVisitor descendent, but we are only wanting to scan a simple list, so perhaps coding a for i := 0 to Count - 1 loop would be quicker. An example of this is shown below:

```
function TPeople.AverageSalary: real;
var
  i : integer ;
  lSum : real ;
begin
  lSum := 0 ;
  for i := 0 to Count - 1 do
    lSum := lSum + TPerson( Items[i] ).Salary ;
  result := lSum / Count ;
end;
```

We want to remove the need to typecast the Items property every time we reference an instance of TPerson. While we are about it, we will change the TPerObjList.Add method to perform some compile time checking so it will only allow us to add TPerson(s) or their descendents to the list. The interface section of the modified TPeople class is shown below:

```
TPeople = class( TPerObjList )
private
  function GetAverageSalary : real ;
protected
  function GetItems(i: integer): TPerson ; reintroduce ;
  procedure SetItems(i: integer; const Value: TPerson); reintroduce ;
public
  property Items[i:integer] : TPerson read GetItems write SetItems ;
  procedure Add( pObject : TPerson ) ; reintroduce ;
published
  property AverageSalary : real read GetAverageSalary ;
end ;
```

GetItems, SetItems and Add just contain inherited calls to the same methods in the abstract class. Remember the side effects of using method hiding with reintroduce to suppress the compiler warnings. If the GetItems, SetItems and Add method never have any changes to their implementation, this is quite safe. If you add extra code to any of these methods in the concrete classes, you will find the behavior unreliable.

This idea of typecasting the Items property and Add method was suggested by one of the early users of the tiOPF, Ian Krigsman and is a technique I use everywhere I have a tightly coupled collection – class relationship. The technique also allows Delphi's Code Insight feature to work in the IDE which makes life so much easier, especially when you are working with a complex class hierarchy.

Now that we have looked at how to typecast the Items[i] property to return the correct type from the collection, and seen how to use ForEach as a quicker way to code simple iteration, we shall look at how to code a generic find visitor.

Adding some helper methods

Searching a tree with Find

Sometimes there is a need to scan the entire class hierarchy looking for a single instance of an object. I use this technique frequently when I have a family of classes that are mapped to each other (rather than a containment relationship where one class owns another). Reading this type of data typically involves two calls to the database: First to read all the mapped to classes that might be required if you have an application with some lookup list data, and the lookup list items are stored in memory as objects. The second call to the database would typically read a list of OIDs for the lookup list items and we would have to search the lookup list hierarchy for the matching item. This can be simplified with a generic find method that searches for an object by OID.

As usual, we will start by writing a small test stub which is shown below. We have added a TSpinEdit to the main form and use this to enter an OID to search for. We call `FPeople.Find(seOID.Value)` and if an object exists in the `FPeople` hierarchy with a matching OID, then a pointer to this object is returned. If a value is found, we show it with `tiShowPerObjAbs`.

```

procedure TFormMain_VisitorManager.btnFindClick(Sender: TObject);
var
  lData : TPerObjAbs ;
begin
  // seOID is a TSpinEdit where we can enter the OID of the
  // object we are looking for
  lData := FPeople.Find( seOID.Value ) ;
  // if found, then show
  if lData <> nil then
    tiShowPerObjAbs( lData ) ;
end;

```

`FPeople.Find` creates an instance of `TVisPerObjFindByOID` and assigns its `OIDToFind` property. The `Iterate` method is then called and the `Visitor` ripples over each node in the tree. If a matching OID is found then, the visitor breaks out of the loop and the result is returned. The implementation of `TPerObjAbs.Find` is shown below:

```

function TPerObjAbs.Find(pOID: TOID): TPerObjAbs;
var
  lVis : TVisPerObjFindByOID ;
begin
  // Create an instance of TVisPerObjFindByOID
  lVis := TVisPerObjFindByOID.Create ;
  try
    // Set its OIDToFind property
    lVis.OIDToFind := pOID ;
    // Call Iterate
    self.Iterate( lVis ) ;
    // Return the found object if one exists
    result := lVis.Found ;
  finally
    // Clean up
    lVis.Free ;
  end ;
end;

```

The interface of `TVisPerObjFindByOID` is as you would expect and follows the standard `TVisitor` subclass interface with `AcceptVisitor` and `Execute` being overridden. There are two additional properties `OIDToFind` and `Found`, which is where a pointer to the located object can be returned to the calling client. The interface of `TVisPerObjFindByOID` is shown below:

```

TVisPerObjFindByOID = class( TVisitor )
private
  FiOIDToFind: TOID;
  FFound: TPerObjAbs;
protected
  function AcceptVisitor : boolean ; override ;
public
  procedure Execute( pVisited : TVisited ) ; override ;
  property Found : TPerObjAbs read FFound ;
  property OIDToFind : TOID read FiOIDToFind write FiOIDToFind ;
end ;

```

`TVisPerObjFindByOID.AcceptVisitor` perform a type check on `Visited` to ensure it is an instance of `TPerObjAbs` (because if it is not, it probably won't have an `OID` property). `AcceptVisitor` also checks that `Found = nil` so the visitor will break out of the iteration loop once the first object with a matching `OID` is found. The implementation of `TVisPerObjFindByOID.AcceptVisitor` is shown below:

```

function TVisPerObjFindByOID.AcceptVisitor: boolean;

```

```
begin
  result := ( Visited is TPerObjAbs ) and
            ( FFound = nil ) ;
end;
```

TVisPerObjFindByOID.Execute performs the usual call to AcceptVisitor and breaks out if false is returned. The OID of the object being visited is compared to the value we are searching for and if they match, FFound is set to point to the found object. FFound <> nil is used in AcceptVisitor as the means of breaking out of the iteration loop. The implementation of TVisPerObjFindByOID.Execute is shown below.

```
procedure TVisPerObjFindByOID.Execute(pVisited: TVisited);
begin
  inherited Execute( pVisited ) ;
  if not AcceptVisitor then
    Exit ; //==>

  // Check for a matching OID
  if TPerObjAbs( Visited ).OID = FiOIDToFind then
    FFound := TPerObjAbs( Visited ) ;
end;
```

This technique for finding a single object in a hierarchy is useful for searching for objects by OID as long as each OID is unique throughout the entire database. If for example a generator or auto incrementing field has been used to create database primary key fields and there is a chance that OIDs are not unique across the database, then this technique will fail. (OIDs can be unique within a table, but not unique across the database and it's this that will cause unreliable results when TVisPerObjFindByOID is used.

Searching a tree with FindByProperty

The final aim of this section is to create a generic search routine that will scan a Composite hierarchy looking for a single object (or list of objects) by published property value. This can be achieved with the same technique we used earlier – by writing a search Visitor and using RTTI to compare the property values. We start by writing a test stub. In this example, cbProperty is a drop down combo box with a list of available property names and eValue is a TEdit where the search for value can be entered. We call FPeople.FindByProperty passing the property name and value we are searching for. If found, a pointer to the matching object is returned and if not found, nil is returned. This code is shown below:

```
procedure TFormMain_VisitorManager.btnFindByPropClick(Sender: TObject);
var
  lData : TPerObjAbs ;
begin
  // Call FindByProperty, passing a property name and a value
  lData := FPeople.FindByProperty( eProperty.Text, eValue.Text ) ;
  // if found, then show
  if lData <> nil then
    tiShowPerObjAbs( lData ) ;
end;
```

Next, we can write the TPerObjAbs.FindByProperty method where a TVisPerObjFindByProperty visitor is created and the search values are set. The Visitor is passed to the object at the top of the tree where the search will be started and the result of a successful find is passed back as the result of the function. This code is shown below:

```
function TPerObjAbs.FindByProperty(
  pFindProp : string ;
  pFindVal : variant ): TPerObjAbs;
var
  lVis : TVisPerObjFindByProperty ;
begin
  lVis := TVisPerObjFindByProperty.Create ;
```

```

try
  lVis.FindProp := pFindProp ;
  lVis.FindVal := pFindVal ;
  self.Iterate( lVis ) ;
  result := lVis.Found ;
finally
  lVis.Free ;
end ;
end;

```

Finally, we write the TVisPerObjFindByProperty visitor that has the usual overridden methods – AcceptVisitor and Execute. Execute uses the two RTTI methods IsPublishedProp and GetPropValue (found in TypInfo.pas) to check if a property of the required name exists on the object being visited, and if it does, to check its value. A successful find sets the FFound property and breaks the visitor out of its iteration loop. The interface of TVisPerObjFindByProperty is shown below:

```

TVisPerObjFindByProperty = class( TVisitor )
  private
    FFound: TPerObjAbs;
    FFindProp: String;
    FFindVal: Variant;
  protected
    function AcceptVisitor : boolean ; override ;
  public
    procedure Execute( pVisited : TVisited ) ; override ;
    property Found : TPerObjAbs read FFound ;
    property FindProp : String read FFindProp write FFindProp ;
    property FindVal : Variant read FFindVal write FFindVal ;
end;

```

And the implementation of AcceptVisitor and Execute are shown here:

```

function TVisPerObjFindByProperty.AcceptVisitor: boolean;
begin
  result := ( Visited is TPerObjAbs ) and
    ( FFound = nil ) ;
end;

procedure TVisPerObjFindByProperty.Execute(pVisited: TVisited);
begin
  inherited Execute( pVisited ) ;
  if not AcceptVisitor then
    Exit ; //==>

  if ( IsPublishedProp( Visited, FindProp ) ) and
    ( GetPropValue( Visited, FindProp ) = FindVal ) then
    FFound := TPerObjAbs( Visited ) ;
end;

```

FindByProperty is a useful function for finding a single object by matching a single property value. A more useful approach would be to return a list of objects and would cater for checking for matches on multiple properties at the same time. Different match criteria like =, <>, >= or LIKE would be useful as well as the ability to chain operations together with AND, OR or NOT. This is implemented in the tiOPF in the unit tiListView.pas where a generic filter dialog is attached to the tiListViewsPlus class.

Summary

This chapter has covered the core requirements of an abstract business object model including the need for a generic list class and an abstract business object class. We started by basing the framework on GoF's Composite pattern then enforced a type-safe relationship between the collection and the objects it contains. We looked at how to make the TVisited.Iterate method generic so it would detect owned objects, and owned lists of objects and iterate over these as well without the need to special code. We looked at a generic way of writing Clone and Assign methods, then developed a useful Find and FindByProperty Visitors. These are the core principles used in the framework and they are built on in

the code you can download which provides many more useful functions and procedures. There is still plenty of room for improvement of the abstract BOM with a more generic search routine being towards the top of the list.

The next chapter

In the next chapter, we shall use the Visitor framework we have developed, along with the abstract business object model based on the Composite Pattern to build a working contact management application.

Chapter #5

Installing the tiOPF

The aims of this chapter

In this chapter we will see how to download and install the tiOPF. We will install the GUI components into the component pallet, setup the necessary search paths, compile the support applications and get the demo application running. We will also look at the directory structure and files that come with the framework.

Prerequisites

None

Contributors

Peter Hinrichsen	Original developer of the tiOPF and author of the first cut of documentation
James Heyworth & Nico Aragón	Developed the Delphi 6 version of the tiOPF
Jean-Baptiste Fidélia	Modified the Getting Started documentation to reflect the changes for Delphi 6
Christopher Latta	Additional changes to get the Delphi 6 version working.
jfl@baycom.co.nz	Suggestions on using Windows XP

Installing the tiOPF

Download the source

The latest version of the tiOPF source code can be found at <http://www.techinsite.com.au/tiOPF/Download.htm>

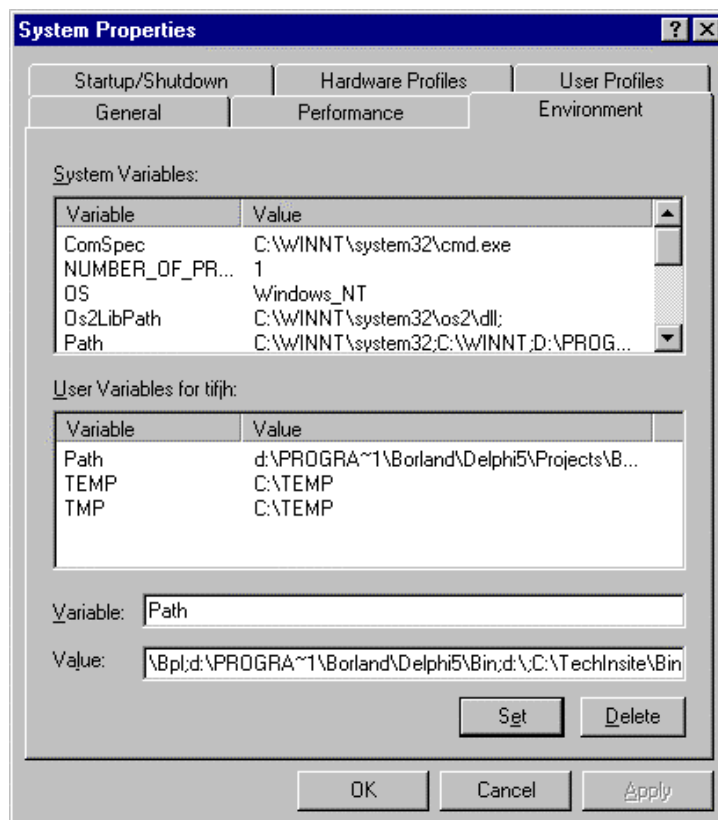
Unzip the file TechInsite.zip into the root of C:. You should have a directory called C:\TechInsite with a bunch of sub directories. It is best to install into C:\TechInsite if you are starting out for the first time as many of the demo applications and the runtime packages have absolute directory paths entered in their Project\Options. You can move things around once you have successfully compiled the first time.

System search path

Delphi requires that the System search path be set so any design time packages can be found when Delphi loads.

Windows NT or 2000

Go to Windows Control Panel and select the System icon. On the Environment tab, add C:\TechInsite\Bin to the search path. You should have a dialog that looks like the one below:



Windows XP

The same as above, but on Windows XP, the “Environments Variables” are accessed from a button on the Advanced tab on the System form.

Windows 95 or 98

You need to add the directory to the path in the [autoexec.bat](#) file. You can either add it to the existing PATH assignment so it looks similar to this (blue path added):

```
PATH=C:\Windows;C:\Windows\Command;C:\TechIn~1\Bin
```

Or to do it a bit clearer, add it to the path at the end of the [autoexec.bat](#) file like this:

```
REM Required for TechInsite tiOPF  
PATH=%PATH%;C:\TechIn~1\Bin
```

You will need to reboot for this setting to take effect.

Delphi's search paths

From Delphi's main menu, load Tools | Environment Options and select the Library tab. Click on the Library Path button and add:

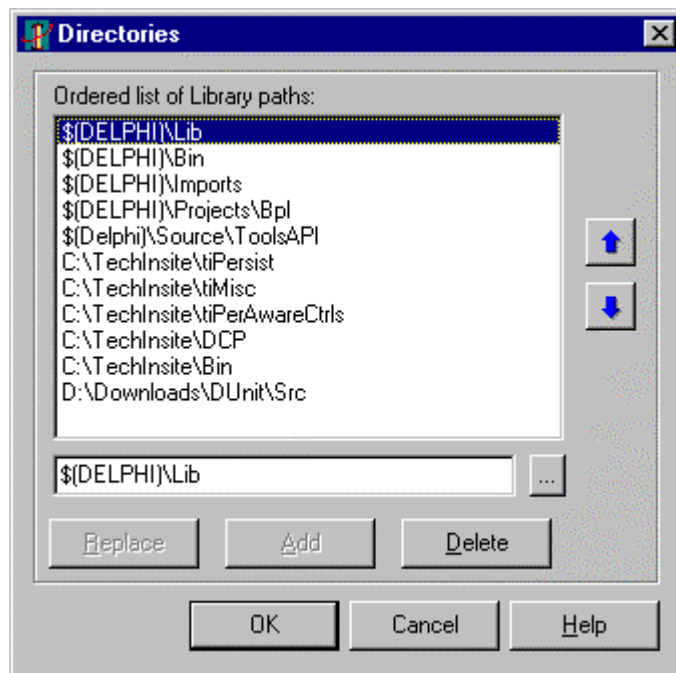
```
C:\TechInsite\TiPersist,  
C:\TechInsite\TiMisc,  
C:\TechInsite\TiPerAwareCtrls  
C:\TechInsite\DCP  
C:\TechInsite\Bin
```

to your search path.

You will probably have to add \$(Delphi)\Source\ToolsAPI as well as this is where the source for the abstract component editor that is used in the GUI controls is located.

If you are going to run the DUnit tests, you will have to add the path to ..\Dunit\Src as well.

The Library Path dialog will look like this:



(If you need to add more search paths because I have missed something, [please let me know](#))

Install the GUI components into Delphi's component pallet

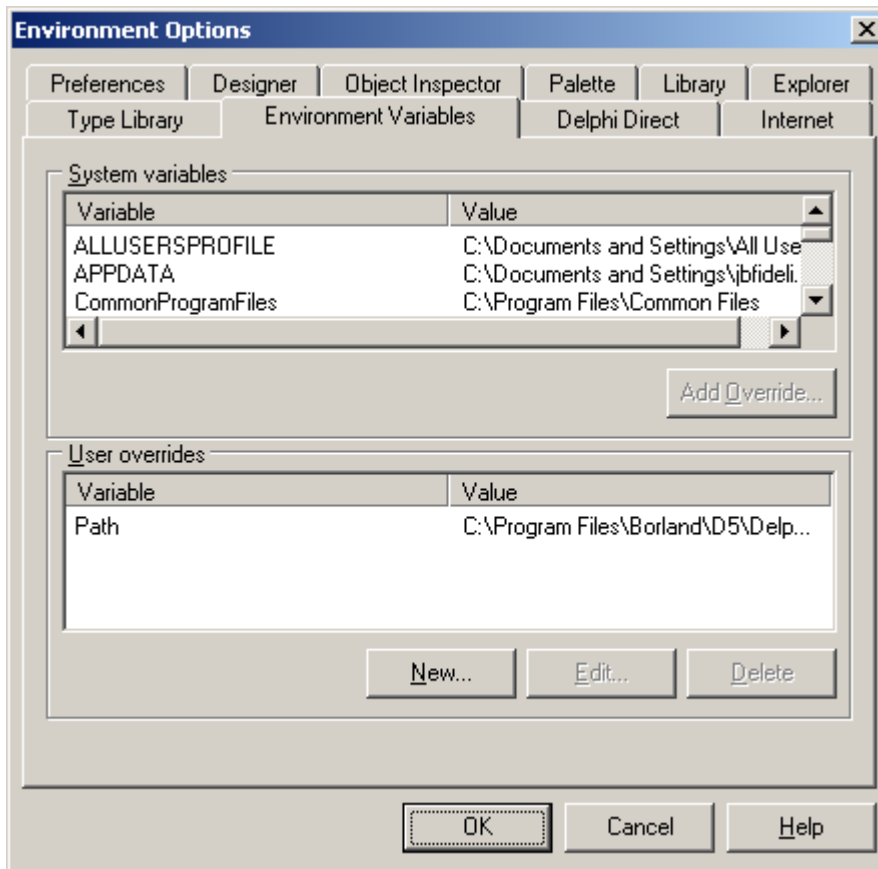
There are two directories for the package files. One for Delphi 5 called dpk5 and one for the Delphi 6 package files called dpk6. The dpk5 packages have been tested and are used in production systems.

The following will describe how to load the tiOPF for Delphi 5. To load for Delphi 6, swap any reference to dpk5 to dpk6.

tiOPF Installation Steps:

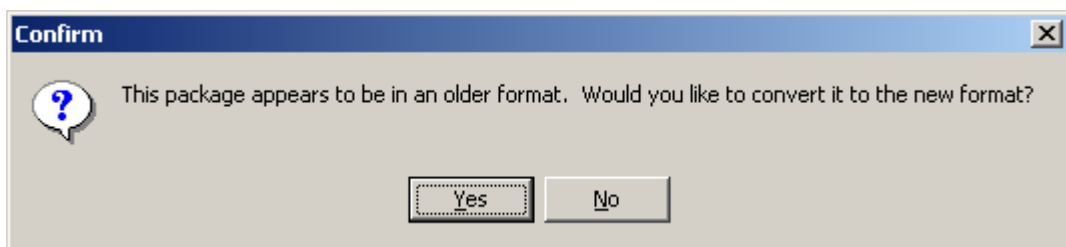
1. If you had previously installed Delphi 5 release of tiOPF, remove all old tiOPF *.dcp and *.bpl that might reside in C:\TechInsite\bin, C:\TechInsite\dcp, \$(Delphi)\Bin, \$(Delphi)\Projects\Bpl, or even in the Windows\System or System32.

- For Delphi 6, make sure the “[Environments Variables](#)” tab from the [Tool\Environment](#) menu doesn’t override the System search path you previously entered in the Control Panel. If it does, have it to point the correct directory:



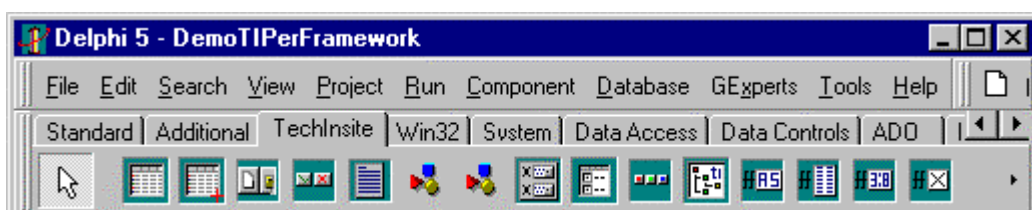
- Open the package `C:\TechInsite\dpk5\tiPerAwareGUI.dpk` and compile. Do not install this into the component pallet as it is a runtime package only.

In Delphi 6 you might receive warnings about old package formats. Just accept the conversion to the new format:



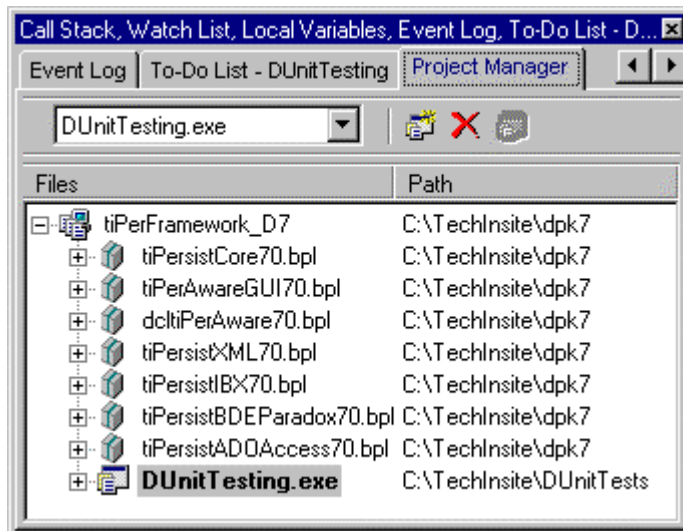
- Open the package `C:\TechInsite\dpk5\delTIPerAware.dpk` and compile, then install into the component pallet.

When done, you should see the TechInsite tab in Delphi’s IDE looking something like this:



Note: Do not install any of the other packages into the component pallet. The rest are non-visual runtime packages that must be distributed with your application. This is important and can be quite a gotcha!

5. Load the program group C:\TechInsite\dpk5\tiPerFramework_D5.bpg (or C:\TechInsite\dpk6\tiPerFramework_D6.bpg or C:\TechInsite\dpk7\tiPerFramework_D7.bpg) into Delphi. This is shown below:



You might have to redo the package group file “tiPerFramework_D6.bpg” from scratch. Just close all unit (menu “File/Close All”). Open the project Manager (menu “View/Project Manager”), add all package and project that was part of the old group file in the same order and save it (menus right-click “Add Existing Project...”, “Save Project Group As...”).

What's in the project group tiPerFramework.dpg?

The project group show above (C:\TechInsite\tiPerFramework_D5.bpg) contains 5 applications and 6 packages.

The packages are:

tiPersistCore - The core persistence layer

tiPerAwareGUI – A runtime package containing the GUI.

dcltiPerAware – A runtime package containing the Register commands to load the components declared in tiPerAwareGUI.

tiPersistXML - The XML persistence layer

tiPersistIBX - The Interbase Express persistence layer

tiPersistBDEParadox - The BDE flavour of the persistence layer

tiPersistADOAccess - The ADO flavour to connect to an Access database.

If you have trouble compiling a package, the first place to look is the Project/Options dialog under the Directories tab. Make sure the output directories for the BPL, DCU and DCP files exist on your hard drive.

The applications are:

DunitTesting.exe – A full suite of DUnit (<http://dunit.sourceforge.net/>) tests to give the tiOPF a thorough work out.

Run Delphi Project | Build all projects and the four packages and three applications should build. (I use Build all projects quite a bit so have added it as a speed button to the Delphi IDE)

Running the support and demo applications

There are several applications in the C:\TechInsite\Demos directory to help you get started. The one to begin with is called DemoTIPerFramework.

You must call DemoTIPerFramework with some parameters to tell the application which persistence layer to use, and which database to connect to.

There are several ways of doing this:

a) Passing all the parameters on the command line. The available parameters are:

Parameter switch	Meaning	Typical value
-pl	Persistence layer name	ibx
-d	Database name	c:\techinsite\Demos\DemoTIPerFramework\Adrs.gdb
-u	User name	SYSDBA
-p	Password	masterkey
-l	Turn logging on	
-lv	Turn visual logging on	
-config	Configuration file name	-config c:\techinsite\Demos\DemoTIPerFramework\Interbase.ini

For example, these parameters will load the demo application, connected to the Interbase database using Interbase Express, with visual logging turned on:

```
-lv -pl IBX -d c:\techinsite\Demos\DemoTIPerFramework\Adrs.gdb -u SYSDBA -p masterkey
```

These parameters will load the demo application, connected to the Paradox database using the BDE, with visual logging turned on:

```
-lv -pl BDEParadox -d c:\techinsite\Demos\DemoTIPerFramework\ParadoxData -u null -p null
```

Note that -u null and -p null must be passed, because if they are not, the framework will prompt the user for a user name and password at start up.

b) You can also pass the location of a configuration file:

For example, the following will load the demo application, connected to the Interbase database using Interbase Express, with visual logging turned on:

```
-lv -config c:\techinsite\Demos\DemoTIPerFramework\Interbase.ini
```

The following will load the demo application, connected to the Paradox database using the BDE, with visual logging turned on:

```
-lv -config c:\techinsite\Demos\DemoTIPerFramework\Paradox.ini
```

c) You can hard code these settings into your application

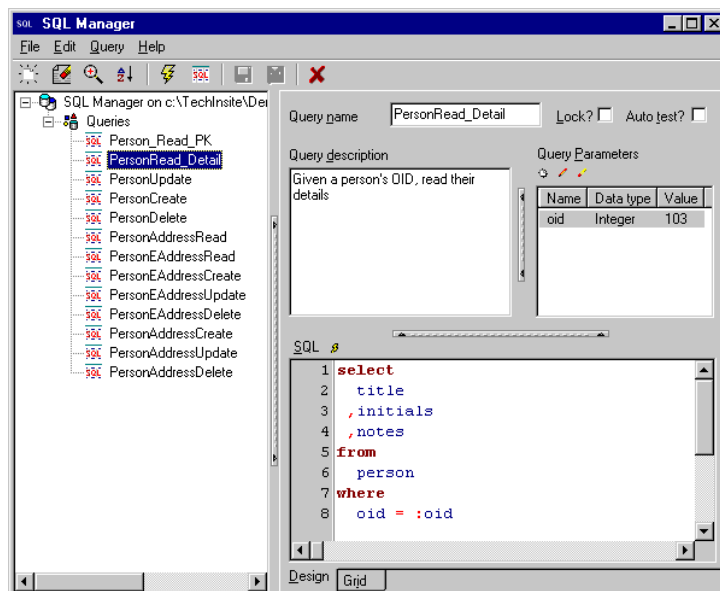
d) You can call deploy the application with a DCD file. (DCD stands for database connection details and is an encrypted text file that contains the database name, user name and password combination for a number of connections.

The various launch methods can be experimented with by calling the batch file RunTest.bat in the DemoTIPerFramework directory.

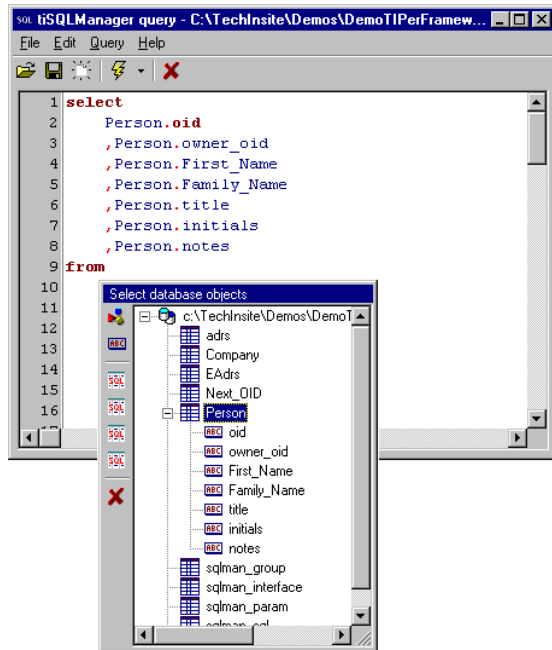
The support applications (temporarily unsupported)

There are also two applications called tiSQLManager and tiSQLEditor in the tiOPFExtras directory. These tools are used for maintaining SQL which can be stored in the database. This way of having an application's SQL stored outside the executable has been used in several large, productions systems, however the tiOPF has since moved on and these applications have not been refactored to keep up with the changes. If you are interested in using the tiSQLManager or tiSQLEditor, please ask the mailing list for help.

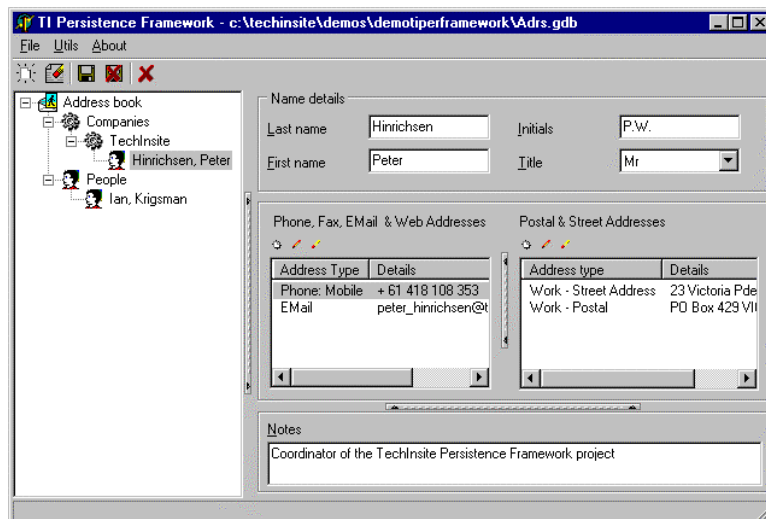
The image below shows the SQLManager, connected to the Interbase database:



The image below shows the tiSQLEditor, connected to the Paradox database. Some SQL is being edited, and the user has hit Ctrl+Space to select from a tree of database objects.



The image below shows the DemoTIPerFramework application, connected to the Interbase database:



There are shortcuts in the \TechInsite\Bin\Shortcuts directory, which will launch the tiSQLManager and tiSQLEditor against the demo databases.

If you find that these getting started notes are not complete, accurate or do not give you enough information to get the packages installed, and the demo application up an running, then please join our [mailing list](#) and raise your questions there. We will update these getting started notes to improve the areas you found lacking.

tiOPF directories and files

Installed directories

When you install the TechInsite object persistence framework, the following directories are created:

Directory	Contents
C:\TechInsite	The root directory of the tiOPF
Bin	Executables and DLLs (packages) are output to this

Directory	Contents
	director.
Shortcuts	Shortcuts to run the tiSQLManager and tiSQLEditor application with the necessary parameters to edit the demo Interbase and Paradox databases that come with the tiOPF
DCP	Output directory for Delphi compiled packages. It is useful to have these all in one place, as Delphi will place them in the Delphi\Bin directory by default (where they're hard to find).
DCU	Delphi compiled units go here. I like to put them all in one place to keep DCU files away from the source code files.
Demos	Demo applications
DemoTIGUI	Demo the dynamic application framework where units of application functionality are deployed in runtime packages that are loaded by the framework core and registered with the package manager.
DemoTIListView	Demo the TtiListView family of controls
DemoTILog	Demo the TtiLog – a thread save application log for debugging or audit trail management.
DemoTIPerAwareMultiSelect	Demo the TtiPerAwareMultiSelect component. Gives two lists of objects side by side. Use the mouse to drag and drop objects from one list to the other.
DemoTIPerFramework	Demo the tiOPF. Uses an address book application to show the controls, business object model, Visitor framework and swappable persistence layer.
DemoTIPersistentAware	Demo the TPersistent aware edit controls.
DemoTIReadOnly	Demo the TtiReadOnly component. Drop one on a form and gain control over the read only state of all the tiPerAware controls on the form.
DemoTISplitter	Demo the TtiSplitter. Gives two panels, side by side with a TSplitter bar between.
DemoTIUtils	Documentation for the tiUtils.pas unit.
DUnitTests	DUnit Test
TestDate	Data used in TestTIPerFramework
dpk5	Runtime (and one design time) packages for Delphi5
dpk6	Runtime (and one design time) packages for Delphi6
SupportApps	Support application used in the tiOPF
tiDBParamMgr	Manage database connection parameters like username and password. Stores them in an encrypted file that can be deployed with an application so a) database connection details are more secure; and b) user names and password can easily be changed.
TiDeployMgr	Contains three applications to manage the deployment of an application: <p>a) tiDeployMgr, which inserts application binaries into the database.</p>

Directory	Contents
	b) tiAppLaunch, which reads the files to be deployed and compares them with the versions currently deployed on the user's machine. A copy any files to be updated, then launches the application.
	c) tiAppLaunchWeb – the same as tiAppLaunch, except it is wrapped up as an ActiveX for deployment in a web page. .Uses HTTP to connect to an ISAPI DLL running on a web server. The ISAPI DLL uses the appropriate persistence layer to connect to the database.
TiLogViewer	A handy viewer for log files.
TiSoapServerCGI	A CGI version of the web server application that must be deployed to get the HTTP persistence layer working. (Because it's CGI, it cant maintain state so will not support database transactions)
TiSoapServerISAPI	An ISAPI version of the web server application that must be deployed to get the HTTP persistence layer working. An ISAPI DLL can maintain state between connections from a client so database transactions are managed.
TiSQLManager	A tool for storing an application's SQL in the actual database. Used with corresponding tiSQLManager enabled Visitors inside a client application.
TiSQLEditor	A tool for editing adhoc SQL that can be stored as a text file.
TiSQLManagerCreateScripts	Scripts for creating tiSQLManager tables.
TiPerAwareCtrls	TPersistent aware edit controls, list view, tree view and other.
TiPersist	The main tiOPF persistence layer code. Includes classes for the core package.

DPK Files

File	Purpose
tiPersistCore.dpk	The core of the swappable persistence layer framework. This package must be in the 'Build with runtime packages' list of the application.
tiPersistBDEParadox.dpk	The BDE – Paradox persistence layer
tiPersistDOA.dpk	The DOA – 'Direct Oracle Access' persistence layer (You must provide the DOA components & license for this to compile) (Stable, tested and rock solid)
tiPersistIBX.dpk	The Interbase Express persistence layer (IBX comes with D5) (Stable, tested and rock solid)
tiPersistADOAccess.dpk	The ADO persistence layer for connecting to an Access database..

Key pas files you will have to add to your uses clauses

The files you will typically have add to a units uses clause are contained in the \TechInsite\TiPersist directory. Remember that you must also have \TechInsite\Bin, \TechInsite\DCP, \TechInsite\TiPerAwareCtrls and \TechInsite\TiPersist in the application or Delphi's search path.

The following table lists the key files in the tiPersist directory. Note that not all the files are listed here as some are work in progress or are not relevant to the core of the framework. Files marked in **Bold and Red** are the ones you will typically have to add to you uses clauses. The other files form part of the framework but you will probably not have to interact with them directly.

PAS Files	
File	Purpose
ctiPersist.pas	Constants defining the names of the available persistence layers.
TiClassToDBMap_BOM.pas, tiClassToDBMap_Srv.pas	Family of classes to manager automatic mapping of a class hierarchy to a database. (You setup the mappings, and these classes will write the SQL)
tiCOM.pas	Manages the calling of CoInitialize, once for each thread.
tiCommandLineParams.pas	Reads and manager parameters passed on the command line to an application.
tiCompressAbs.pas, tiCompressNone.pas, tiCompressZLib.pas	Family of classes to manager compression. Includes a wrapper for Zlib. See application in the demo directory.
tiDataSet_BOM.pas, tiDataSet_Cli.pas, tiDataSet_Srv.pas	Family of classes to give TDataSet like behavior without having to use the TClientDataSet. Would be better to use a third part in memory component, or perhaps the TclientDataSet.
tiDBConnectionPool.pas	Manage a thread safe pool of database connections.
tiHTML.pas	Writes a TList of Tpersistent to a HTML table
tiLog.pas	Thread safe logging to screen, error dialog, console application or text file.
tiNextOID_BOM.pas, tiNextOID_Cli.pas, tiNextOID_Srv.pas	Family of classes for generating application wide unique OIDs. Based on Amblers paper on mapping objects to RDBMSs
tiPersist.pas	Core class to manager all persistence related matters such as loading the correct layer, calls to the Visitor manager and database connection pooling.
tiPoolAbs.pas	An abstract, thread safe pool of anything (used as the base class for the database connection pool)
tiPtnFactory.pas	An abstract Factory. Based on GoF.
tiPtnIterator.pas	An abstract Iterator, based on GoF. Class based, but should be Interface based.
tiPtnStrategy.pas	Abstract Strategy pattern, based on GoF.
tiPtnVis.pas	Abstract Visitor and Visited classes, based on GoF
tiPtnVisMgr.pas	The Visitor manager classes
tiPtnVisPerObj.pas	The abstract persistent object (TPerObjAbs) and persistent object list (TPerObjList)
tiPtnVisPerObj_Cli.pas	'Client side' persistent object helper methods like iPerObjAbsAsString and tiShowPerObjAbs
tiPtnVisSQL.pas	Visitors for managing persistence to relational databases.
tiQuery.pas	Abstract TtiQuery and TtiDatabase classes. The basis for the swappable persistence layer.
tiQueryBDEAbs.pas	Adds BDE functionality to the TtiDatabase and TtiQuery.
tiQueryBDEInterbase.pas	Adds Interbase connectivity via the BDE to TtiDatabase and TtiQuery

PAS Files

File	Purpose
tiQueryBDEParadox.pas	Adds Paradox connectivity via the BDE to TtiDatabase and TtiQuery
tiQueryDOA.pas	Adds Oracle connectivity via the DOA components to TtiDatabase and TtiQuery
tiQueryHTTP.pas	Adds HTTP connectivity via Indy components. Must be used in conjunction with a CGI or ISAPI web server application to provide connectivity to the actual database.
tiQueryIBX.pas	Adds Interbase connectivity via the IBExpress components to TtiDatabase and TtiQuery
tiQueryFactory.pas	Abstract query factory. Defines interface to SQLManager system without giving any concrete implementation.
tiQueryFactorySQL.pas	Implementation of SQLManager system.
tiQueryRemote.pas	Adds remote connectivity behavior to the TtiDatabase and TtiQuery. Used as the parent class for tiQueryHTTP
tiQueryRemoteUtils.pas	Utility methods used in the multi tier flavors of the persistence framework (like tiQueryHTTP)
tiQuerySQLMgr.pas	SQLManager core classes.
tiRegPerLayer.pas	Manages the registration of persistence layers.
tiSingleInstanceApp.pas	Uses a Mutex to force only a single instance of an application.
tiSQLMgr_BOM.pas, tiSQLMgr_Cli.pas, tiSQLMgr_Svr.pas	SQLManager core classes.
tiThreadProgress.pas	A TThread descendant that adds a progress bar to keep users amused while the thread is executing.
tiUtils.pas	Registry, INI file, dialog, string, number, date time, type conversion, win32 API, exception and RTTI management routines
tiWebRequestMgr.pas	Server side code to implement tiQueryHTTP persistence layer. Will manager a request coming from a tiQueryHTTP application and convert the HTTP request into a call to the Visitor framework. Streams the result back as XML.
tiXML.pas	Abstract wrapper for XML parsing engine (Requires work)
tiXMLMSDOM.pas	Wrapper for MSXML DOM
tiXMLDataSetTransport_BOM.pas	Object framework for streaming data sets as XML
tiXMLDataSetTransport_Srv.pas	Visitor for writing a TtiQuery result to XML directly via a TStream (quicker that going via a parser)

Remember, an application using the tiOPF uses runtime packages and the only way to share resources between a runtime package and the EXE is to have the shared resources in a package which is in the application's requires clause. This is discussed in more details in http://www.techinsite.com.au/tiOPF/Doc/6_AWorkedExampleOfUsingTheTIOPF.htm

Chapter #6

A worked example of using the tiOPF

The aims of this chapter

In the previous chapters, we have seen how to use the Visitor pattern, along with the Template Method to map a business object model into a relational database. We have also developed an abstract business object, and collection object to descend our concrete classes from. In this chapter, we will build on what we have covered and create a working application using the TechInsite object persistence framework.

We shall work through the following steps:

1. Write a brief use case for the system we will build
2. Draw a class diagram of the business object model (using the UML design tool minUML)
3. Design the database schema, and document the mapping between objects and tables, properties and columns
4. Code the business object model
5. Write the SQL create script for the database
6. Look at alternative BOM – database mapping strategies:
 - a) Hard code the SQL
 - b) Use the tiSQLManager
 - c) Auto generate the SQL
7. Write the GUI and hook it up to the BOM
8. Modify the application so it will connect to different databases

The order that we work through these steps shall be 1 to 5, then we will implement one of the strategies in 6 so we can write the GUI in 7. We will then return to 6 and implement the other two strategies before moving on to 8 where we will implement the swappable database connection layer.

At the end of this chapter we will have build a contact management application that will seamlessly connect to either Interbase using IBExpress, Paradox using the BDE or Access using ADO.

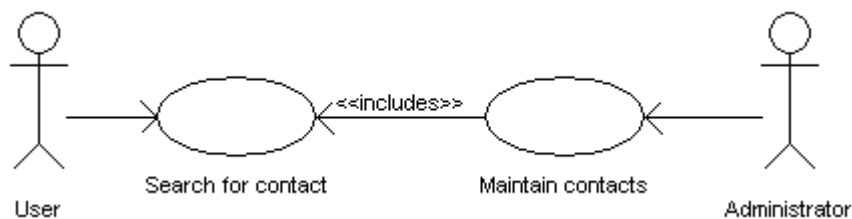
Prerequisites

This chapter builds on the concepts introduced in chapters 2, 3 and 4 so it will be a good idea to read these first. It also assumes that you have installed the tiOPF, details of which are described in chapter 5.

Application design

The use case

There are two actors and two use cases in the system: Administrators who are responsible for maintaining the contact list, and users who will be searching for a contact by name. These actors may or may not be the same person so a read only view will be necessary to prevent users from modifying data that only administrators should have access to. A diagram representing this is shown below:



Look and feel

The application will use a tree view on the left-hand side of the main form to browse and search for the contacts in the database. Initially, all the contacts will be loaded when the application starts, but we will probably want to change as the size of the database grows to provide database level searching.

Use case #1 Search for a contact

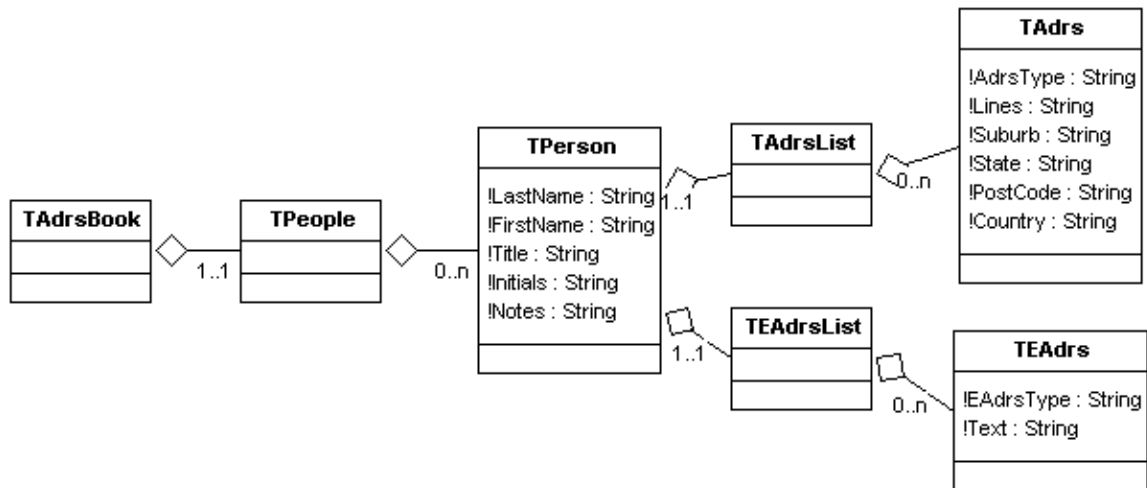
1. Start the application and enough data about each entry in the database will load to allow a human to navigate.
2. Scroll down the tree view and click on the required person – the person’s details will load and be shown on the right hand side of the application.

Use case #2 Maintain contacts

1. Navigate to the contact to be changed as in use case #1, or click insert to add a new contact.
2. Navigate to the part of the contact to be maintained (e.g., name, and phone number, postal address) and make edits.
3. Click save button.

The class diagram

We shall implement the contact management application with 7 classes as shown in the class diagram below:



These classes have the following purposes:

Class	Parent	Description
TContactMgr	TPerObjAbs	The top of the hierarchy. Initially, this class will just be a holder for an instance of TPeople, but later we may extend it to contain lookup list data, and perhaps an instance of TCompanies
TPeople	TPerObjList	A collection of TPeople
TPerson	TPerObjAbs	A person object owned by TPeople. Has published properties for LastName, FirstName, Title and Initials. Has a property of type TAdrsList and TEAdrsList.
TAdrsList	TPerObjList	A container for TAdrs objects
TEAdrsList	TPerObjList	A container for TEAdrs objects
TAdrs	TPerObjAbs	Holds a conventional street or post office box address. Has published properties AdrsType, lines, suburb, state, postcode and country.
TEAdrs	TPerObjAbs	An electronic address object that belong to the TPerson. Has properties Address type and address text.

The database schema & object – database mapping

We shall store the data modelled by these seven classes in three tables in the database as shown in the table below:

Class	Table	Property	Column	Notes
TPerson	Person	<u>OID</u>	OID	Primary key
		FirstName	First_Name	
		LastName	Last_Name	
		Title	title	
		Initials	initials	
		Notes	notes	
TAdrs	Adrs	<u>OID</u>	oid	Primary key
		Owner.OID	owner_oid	Foreign key to Person
		AdrsType	lines adrs_type	
		Suburb	suburb	
		State	state	

Class	Table	Property	Column	Notes
		PCode	pcode	
		Country	country	
TEAdrs	EAdrs	OID	oid	Primary key
		Owner.OID	owner_oid	Foreign key to Person
		EAdrsType	eadsr_type	
		Text	text	

A note about OIDs

The framework supports several methods of Object Identifier (OID) generation. Earlier versions of the framework used integers (which is what we will be using here), but there is additional support for HEX and GUID OIDs. If none of these are suitable, you can easily write your own by descending a new TOID descendent class defined in TiPerObjOIDAbs.Pas. Remember to make your OID table in your database match the storage requirements of OID. You must remember to Use the appropriate OID class generation unit in your BOM otherwise you will get a runtime error message “Attempt to create unregistered OID class”. Likewise, you must also remember to include one and only one OID class generation unit.

Application construction

If you have not installed the tiOPF components, and setup the search path to give access to the tiOPF files, then do this now following the instructions under ‘Installing the tiOPF’

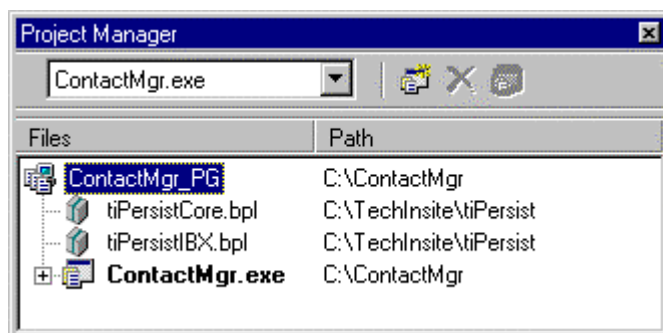
Create a new application

Create a directory to place the application source, and then create a Delphi project group called ContactMgr_PG and save it.

Add the Delphi packages tiPersist.dpk and tiPersistIBX.dpk (because we are starting by building an Interbase application with IBX) found in the \TechInsite\tiPersist directory.

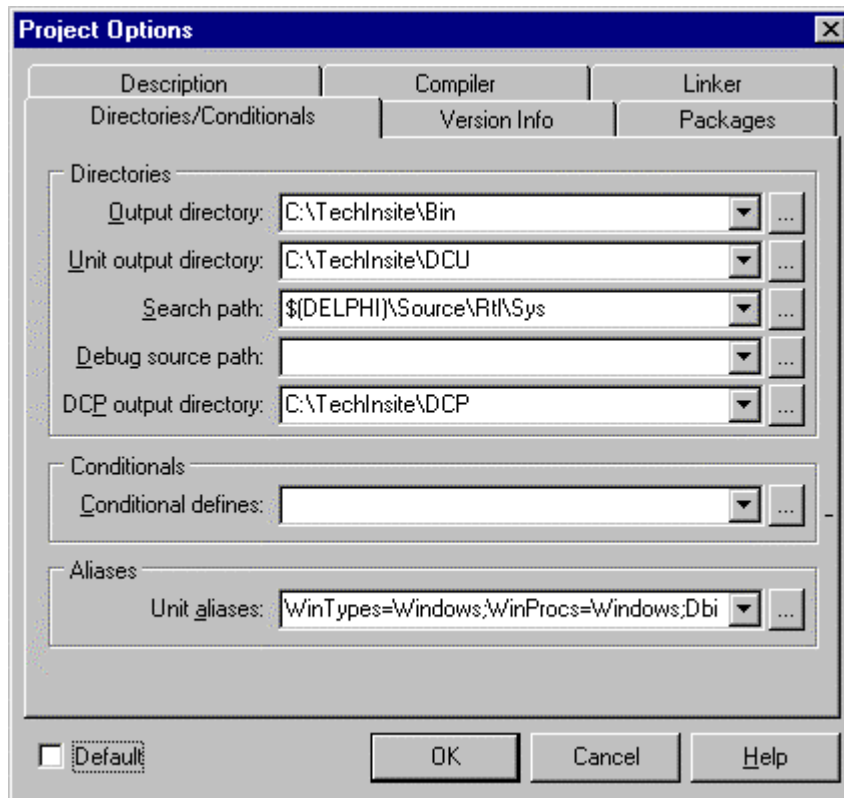
Add a new Delphi project. Call the main form FormMain, and save its pas file as FMain.pas. Call the application ContactMgr.dpr

Arrange the projects in the project tree in the order of tiPersistCore, tiPersistIBX then ContactMgr. This is important because of the dependencies between the three projects. At the end of this, you will probably have a project tree that looks something like this:



I find the project tree provides a very hand way of navigating around the framework files. If I can't remember a unit or class name, I can double click on tiPersistCore and navigate all the units and classes in the package editor that pops up.

I like to have control over the directory that Delphi puts its output files, so go to Project Options and select the Directories/Conditionals tab. Check that the Directories/Conditionals settings of tiPersistCore and tiPersistIBX looks like this:



Note that we want to set the 'Output directory', 'Unit output directory' and 'DCP output directory'. The location of the binaries and DCP files are the most important because we will have to reference these files in the application and will want to know where to look.

Set the 'Output directory' of the ContactMgr application to C:\TechInsite\Bin. It is important that this is the same directory the package BPL files will be placed in because the ContactMgr application will look for the persistence layer it is to load in the same directory as its executable.

Select Project | Build all projects (I set-up a speed button for this because I build all projects quite a bit), then run the ContactMgr and check the files have been output to the directories we expect.

Note, we really want to know where the BPL files are being placed. Delphi will put them in the \Delphi\Bin directory by default. If you have two copies of the same BPL on the search path, and Delphi is not loading the copy you think it is, it can cost you hours of unnecessary debugging (this one still catches me out sometimes).

Code the business object model

Setting up the pas files

Add a new unit to the project and save it under the name ContactMgr_BOM.pas. The root of the name is not important, but the _BOM bit is. We will be creating a family of three units with the names *_BOM.pas, *_Cli.pas and *_Srv.pas to hold the business object model, server side visitors and client side single instance of the BOM

We have seven classes to create (TContactMgr, TPeople, TPerson, TAdrsList, TEAdrsList, TAdrs, and TEAdrs) and will create them one at the time, then populate them with some hard coded data. We will use tiShowPerObjAbs to look inside the objects as we are building them to confirm the structure is working as expected.

Load ContactMgr_BOM.pas in the editor and add tiPtnVisPerObj to the uses clause. Add the class declaration for TContactMgr like this:

```

unit ContactMgr_BOM;

interface
uses
  tiPtnVisPerObj,
  tiPerObjOIDAbs,
  tiPerObjOIDInteger,
  ;
type
  TContactMgr = class( TPerObjAbs )

implementation

end.

```

Setting up some code templates

Ctrl+Left Click TPerObjAbs and you will be taken to its declaration. Directly above the declaration of TPerObjAbs, there are two stubs of code (in comments) which you can paste into ContactMgr_BOM to help get started when writing the interface of TPerObjAbs and TPerObjList descendants.

I usually make two entries in Delphi's Code Insight to speed the process of creating TPerObjAbs and TPerObjList descendants. I add the following blocks of code:

TPerObjList code stub. Code Insight Shortcut name: pol

```

//-----
TMyClasses = class( TPerObjList )
private
protected
  function GetItems(i: integer): TMyClass ; reintroduce ;
  procedure SetItems(i: integer; const Value: TMyClass); reintroduce ;
  function GetOwner: TMyClasses; reintroduce ;
  procedure SetOwner(const Value: TMyClasses); reintroduce ;
public
  property Items[i:integer] : TMyClass read GetItems write SetItems ;
  procedure Add( pObject:TMyClass; pbDefaultDispOrder:boolean = true );
reintroduce;
  property Owner : TMyClass read GetOwner write SetOwner ;
published
end ;

```

TPerObjAbs code stub. Code Insight Shortcut name: poa

```

//-----
TMyClass = class( TPerObjAbs )
private
protected
  function GetOwner: TMyClasses; reintroduce ;
  procedure SetOwner(const Value: TMyClasses ); reintroduce ;
public
  property Owner : TMyClasses read GetOwner write SetOwner ;
end ;

```

Coding TContactMgr, TPeople and TPerson classes

We will code these three classes together, and then populate the object model with some data that is hard coded to test the framework.

Paste in code templates for TContactMgr (based on TPerObjAbs), TPeople (based on TPerObjList) and TPerson (based on TPerObjAbs). Write forward declarations of the three classes, and then add an owned instance of TPeople to TContactMgr so the interface looks like this:

```

TContactMgr = class ;
TPeople     = class ;
TPerson     = class ;

```

```

TContactMgr = class( TPerObjAbs )
private
  FPeople: TPeople;
protected
  function GetCaption : string ; override ;
published
  property People : TPeople read FPeople ;
public
  constructor create ; override ;
  destructor destroy ; override ;
end ;

```

The constructor and destructor manage the creation and destruction of the owned instance of TPeople. Note that the property, People, is published so the automatic iteration code that we developed in the TVisited class will detect the owned class and pass the visitor over each node. We have also added the overridden function GetCaption where simply return the string 'Contact manager' in the implementation. This will become useful when we start building the GUI using the TtiTreeView.

Paste in a code stub for TPeople by copying the TPerObjList template. Change the type of its Owner, and Items properties and modify the signature of their get and set methods. So Owner is of type TContactMgr and Items is of type TPerson. When you have finished, the interface of TPeople should look like this:

```

//-----
TPeople = class( TPerObjList )
private
protected
  function GetItems(i: integer): TPerson ; reintroduce ;
  procedure SetItems(i: integer; const Value: TPerson); reintroduce ;
  function GetOwner: TContactMgr; reintroduce ;
  procedure SetOwner(const Value: TContactMgr); reintroduce ;
public
  property Items[i:integer] : TPerson read GetItems write SetItems ;
  procedure Add( pObject : TPerson ; pDefDispOrdr : boolean = true ) ;
reintroduce ;
  property Owner : TContactMgr read GetOwner write SetOwner ;
published
end ;

```

Hit Ctrl+Shift+C and Delphi will fill out the implementation of TPeople. You now have to hand craft the code for GetItems, SetItem, GetOwner, SetOwner and Add. This takes no time at all because I have Code Insight templates set-up for each of these called igi (inherited GetItems), isi (inherited SetItems), igo, iso and ia.

The finished implementation of TPeople looks like this:

```

// * * * * *
// *
// * TPeople
// *
// * * * * *
procedure TPeople.Add(pObject: TPerson; pDefDispOrdr: boolean);
begin
  inherited Add( pObject, pDefDispOrdr ) ;
end;

function TPeople.GetItems(i: integer): TPerson;
begin
  result := TPerson( inherited GetItems( i ) ) ;
end;

function TPeople.GetOwner: TContactMgr;
begin
  result := TContactMgr( inherited GetOwner ) ;
end;

procedure TPeople.SetItems(i: integer; const Value: TPerson);

```



```

begin
  inherited SetItems( i, Value ) ;
end;

procedure TPeople.SetOwner(const Value: TContactMgr);
begin
  inherited SetOwner( Value ) ;
end;

```

And the finished implementation of TPerson looks like this:

```

// * * * * *
// *
// * TPerson
// *
// * * * * *
function TPerson.GetOwner: TPeople;
begin
  result := TPeople( inherited GetOwner );
end;

procedure TPerson.SetOwner(const Value: TPeople);
begin
  inherited SetOwner( Value ) ;
end;

```

This takes very little time to had code with the help of Code Insight, however it would be still quicker with the help of a wizard, or a UML tool to output the Delphi code. We will add this functionality to the tiOPF one day in the future.)

Next, add the properties LastName, FirstName, Title, Initials and notes to TPerson. Hit Ctrl+Shift+C and Delphi will finish off the interface declaration. The interface of TPerson now looks like this:

```

//-----
TPerson = class( TPerObjAbs )
private
  FTitle: string;
  FFirstName: string;
  FInitials: string;
  FLastName: string;
  FNotes: string;
protected
  function GetOwner: TPeople; reintroduce ;
  procedure SetOwner(const Value: TPeople ); reintroduce ;
public
  property Owner : TPeople read GetOwner write SetOwner ;
published
  property LastName : string read FLastName write FLastName ;
  property FirstName : string read FFirstName write FFirstName ;
  property Title : string read FTitle write FTitle ;
  property Initials : string read FInitials write FInitials ;
  property Notes : string read FNotes write FNotes ;
end ;

```

Testing TContactMgr, TPeople and TPerson classes

We shall now create a single instance of TContactMgr, populate it with some test data by hard coding. The question is, where shall the instance be created? Who shall be responsible for creating it? And who shall be responsible for freeing it?

I usually choose from one of three strategies:

1. **Create is as an application wide, globally visible Singleton.** This is a useful technique when you are working with background data that can be shared between objects, or when you are creating a single form application like the one we are working on here.

2. **Create it as an owned property of some sort of manager class.** This technique is good if there will be 0..n instances of the class, and you want to create them on demand and share them between client classes once they have been created.
3. **Create it and free it in a form** Create and free it in the constructor and destructor of the class that it will interact with (often an application's form, or main form). This is good for MDI applications when you might want several forms all editing the same class of object, but populated with different data.

We shall create the TContactMgr class as a Singleton because we are working with a single form application and it makes it easier to abstract persistence code away from the GUI.

Create a unit called ContactMgr_Cli.pas, and add the following code:

```
unit ContactMgr_Cli;

interface
uses
  ContactMgr_BOM
  ;

// We hide a variable with unit wide scope behind a function.
function gContactMgr : TContactMgr ;

implementation
var
  uContactMgr : TContactMgr ;

// Fakes some of the behaviour expected of a Singleton
function gContactMgr : TContactMgr ;
begin
  // If uContactMgr has not been created, then create one.
  if uContactMgr = nil then
    uContactMgr := TContactMgr.Create ;
  result := uContactMgr ;
end ;

initialization

finalization
  uContactMgr.Free ;

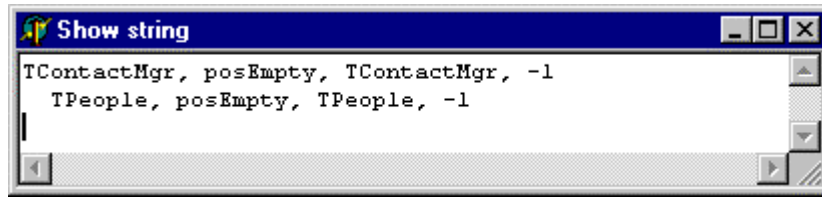
end.
```

This fakes some of the behaviour expected from a Singleton. This technique will allow the singleton to be deleted, which is technically not permitted. It will also allow more than one instance of TContactMgr to be created which also violates the rules of a pure Singleton. The technique is, however, quick to code and easy to understand which is not the case for a pure GoF Singleton when implemented in Delphi.

Now, add the unit tiPtnVisPerObj_Cli to the uses clause of the Implementation section of the main unit then drop a button on the application's main form. Add the following code to the button's OnClick event handler:

```
procedure TFormMain.Button1Click(Sender: TObject);
begin
  tiShowPerObjAbs( gContactMgr ) ;
end;
```

This will output the contents of the empty to TContactMgr to a popup dialog like this:



We will now populate TContactMgr with some hard coded data so create another unit called ContactMgr_TST.pas (TST stands for test) and add the following code:

```
unit ContactMgr_TST;

interface
uses
  ContactMgr BOM
  ;

procedure PopulateContactMgr( pContactMgr : TContactMgr ) ;

implementation

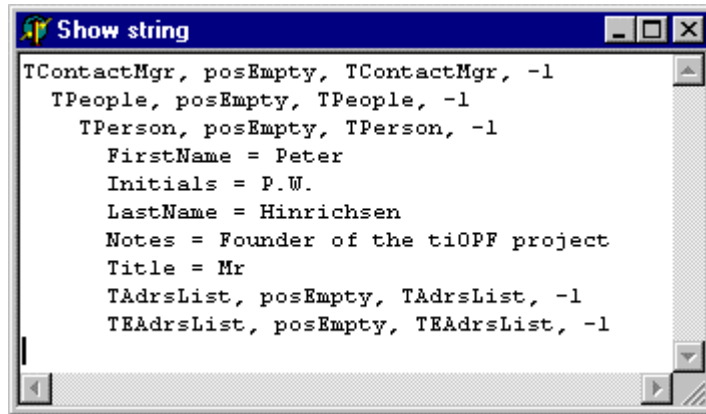
procedure PopulateContactMgr( pContactMgr : TContactMgr ) ;
var
  lPerson : TPerson ;
begin
  lPerson := TPerson.Create ;
  lPerson.LastName := 'Hinrichsen' ;
  lPerson.FirstName := 'Peter' ;
  lPerson.Title := 'Mr' ;
  lPerson.Initials := 'P.W.' ;
  lPerson.Notes := 'Founder of the tiOPF project' ;
  pContactMgr.People.Add( lPerson ) ;

end ;
```

This will populate the contact manager with yours truly.

We have to add a call to PopulateContactMgr somewhere in application and as we are going down the singleton route for the creation of the TContactMgr, we can add the code there:

```
function gContactMgr : TContactMgr ;
begin
  // If uContactMgr has not been created, then create one.
  if uContactMgr = nil then
  begin
    uContactMgr := TContactMgr.Create ;
    PopulateContactMgr( uContactMgr ) ;
  end ;
  result := uContactMgr ;
end ;
```



This is what we are wanting so we can go on and create the remaining four classes.

Coding TAdrsList, TEAdrsList, TAdrs and TEAdrs

We shall not code the four classes TAdrsList, TEAdrsList, TAdrs and TEAdrs. There is a common property shared between TAdrs and TEAdrs called AdrsType, so we will create an abstract address class to provide this common behaviour, and descend both TAdrs and TEAdrs from this abstract class.

First, the interface of TPerson is extended with an owned instance of TAdrsList and TEAdrsList. These list classes are surfaced as published properties so they will be automatically detected by the iteration routine in the TVisited class. The interface extended interface of TPerson is shown below:

```

TPerson = class( TPerObjAbs )
private
  FAdrsList : TAdrsList ;
  FEAdrsList : TEAdrsList ;
protected
  function GetCaption : string ; override ;
  constructor Create ; override ;
  destructor Destroy ; override ;
published
  property AdrsList : TAdrsList read FAdrsList ;
end ;

```

The implementation of TPerson is as you would expect with an instance of TAdrsList and TEAdrsList being created and destroyed in the constructor and destructor. There are four extra lines of code however where the owner relationship between TAdrsList, TEAdrsList, their list elements and the TPerson are set. In the implementation of GetCaption, we will return the string `FirstName + ' ' + LastName`.

Now, from deep in the hierarchy, at say the TAdrs level, we want to be able to chain up the object hierarchy like this to get the owning person like this:

```
lPerson := lAdrs.Owner.Owner ;
```

But with the TAdrs and TEAdrsList classes the way we would build them by default, the owner of a TAdrs is its TAdrsList. We really want the owner property of a TAdrs to reference the TPerson object like this:

```
lPerson := lAdrs.Owner ;
```

This is achieved by setting the ItemOwner property of the TAdrsList and TEAdrsList objects like this:

```
constructor TPerson.Create;
```

```
begin
  inherited;
  FAdrsList := TAdrsList.Create ;
  FAdrsList.Owner := self ;
  FAdrsList.ItemOwner := self ;

  FEAdrsList := TEAdrsList.Create ;
  FEAdrsList.Owner := Self ;
  FEAdrsList.ItemOwner := Self ;

end;
```

This can be a little confusing. The owner of a TAdrs, from the point of view of the class being responsible for freeing the TAdrs is the TAdrsList. The owner of the TAdrs when TAdrs.Owner is called is the TPerson class, which is much more useful when traversing the object hierarchy. This is all taken care of in the TPerObjAbs.Add method.

Next, we create the interface and implementation of TAdrsListAbs, TAdrsList and TEAdrsList. These consist of new implementations of the methods GetItems, SetItems, GetOwner, SetOwner and Add with the compiler warnings being suppressed by using the reintroduce key word. The type of the property Items has also been changed. By changing the type of these properties, along with their get and set methods we firm up the relationship between the collection class and the objects that it holds. This takes a bit of extra work up front, but we will be more than compensated in saved development, debugging and maintenance time. The interface of TAdrsList is shown below:

```
TAdrsListAbs = class( TPerObjList )
protected
  function GetOwner: TPerson; reintroduce ;
  procedure SetOwner(const Value: TPerson); reintroduce ;
public
  property Owner : TPerson read GetOwner write SetOwner ;
end ;

TAdrsList = class( TAdrsListAbs )
private
protected
  function GetItems(i: integer): TAdrs ; reintroduce ;
  procedure SetItems(i: integer; const Value: TAdrs); reintroduce ;
public
  property Items[i:integer] : TAdrs read GetItems write SetItems ;
  procedure Add( pObject : TAdrs ; pDefDispOrdr : boolean = true ) ; reintroduce ;
;
published
end ;
```

The interface of TAdrsList is not shown, but as you would expect, it follows the same pattern as TPeople with each of the Get and Set methods simply calling inherited with some type casting as necessary. The interface and implementation of TEAdrsList follows the same pattern as in TAdrsList with the methods each calling inherited with the appropriate types casting.

TAdrs and TEAdrs both descend from the common parent TAdrsAbs because they have the property AdrsType in common. The interface of TAdrsAbs is shown below:

```
TAdrsAbs = class( TPerObjAbs )
private
  FAdrsType: string;
published
  property AdrsType : string read FAdrsType write FAdrsType ;
end ;
```

In TAdrs, the type of the Owner property is changed to TPerson along with the corresponding Get and Set methods. The properties Lines, Suburb, PCode, State, and country are also added. The interface of TAdrs is shown below:

```
TAdrs = class( TAdrsAbs )
private
  FCountry: string;
  FSuburb: string;
  FLines: string;
  FPCode: string;
  FState: string;
protected
  function GetCaption : string ; override ;
  function GetOwner: TAdrsList; reintroduce ;
  procedure SetOwner(const Value: TAdrsList ); reintroduce ;
  property Owner      : TAdrsList read GetOwner write SetOwner ;
published
  property Lines      : string read FLines  write FLines ;
  property Suburb     : string read FSuburb  write FSuburb ;
  property State      : string read FState   write FState ;
  property PCode      : string read FPCode   write FPCode ;
  property Country    : string read FCountry write FCountry ;
end ;
```

You will now need to add the tiUtils unit to ContactMgr_BOM's uses clause, as this is where the tiStrTran function can be found. The implementation of GetCaption creates a single line view of the address and is shown below:

```
function TAdrs.GetCaption: string;
begin
  result :=
    tiStrTran( tiStrTran( Lines, Cr, ' ' ), Lf, '' ) +
    ' ' + Suburb + ' ' + State + ' ' + PCode + ' ' +
    Country ;
end;
```

The interface of TEAdrs is the same as TAdrs except that the properties Lines, Suburb, etc are replaced with a single property called Text.

We can extend the PopulateContactMgr routine found in ContactMgr_TST.pas by creating instances of TAdrs and TEAdrs. This code looks like this:

```
procedure PopulateContactMgr( pContactMgr : TContactMgr ) ;
var
  lPerson : TPerson ;
  lAdrs : TAdrs ;
  lEAdrs : TEAdrs ;
begin
  lPerson := TPerson.Create ;
  lPerson.LastName := 'Hinrichsen' ;
  lPerson.FirstName := 'Peter' ;
  lPerson.Title := 'Mr' ;
  lPerson.Initials := 'P.W.' ;
  lPerson.Notes := 'Founder of the tiOPF project' ;
  pContactMgr.People.Add( lPerson ) ;

  lAdrs := TAdrs.Create ;
  lAdrs.AdrsType := 'Street' ;
  lAdrs.Lines := '23 Victoria Pde.' ;
  lAdrs.Suburb := 'Collingwood' ;
  lAdrs.PCode := '3066' ;
  lAdrs.State := 'VIC' ;
  lAdrs.Country := 'Australia' ;
  lPerson.AdrsList.Add( lAdrs ) ;

  lAdrs := TAdrs.Create ;
  lAdrs.AdrsType := 'Postal' ;
```

```

lAdrs.Lines      := 'PO Box 429' ;
lAdrs.Suburb    := 'Abbotsford' ;
lAdrs.PCode     := '3067' ;
lAdrs.State     := 'VIC' ;
lAdrs.Country   := 'Australia' ;
lPerson.AdrsList.Add( lAdrs ) ;

lEAdrs := TEAdrs.Create ;
lEAdrs.AdrsType := 'EMail' ;
lEAdrs.Text     := 'peter hinrichsen@techinsite.com.au' ;
lPerson.EAdrsList.Add( lEAdrs ) ;

lEAdrs := TEAdrs.Create ;
lEAdrs.AdrsType := 'Mobile' ;
lEAdrs.Text     := '0418 108 353' ;
lPerson.EAdrsList.Add( lEAdrs ) ;

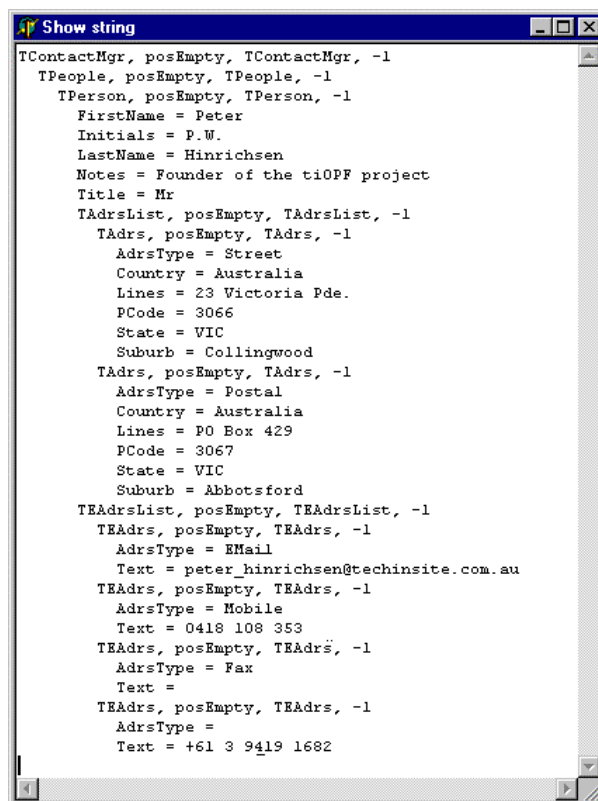
lEAdrs := TEAdrs.Create ;
lEAdrs.AdrsType := 'Fax' ;
lPerson.EAdrsList.Add( lEAdrs ) ;

lEAdrs := TEAdrs.Create ;
lEAdrs.Text     := '+61 3 9419 1682' ;
lPerson.EAdrsList.Add( lEAdrs ) ;

end ;

```

When we run the application and click the button with the `tiShowPerObjAbs()` call, as expected we get the following dialog:



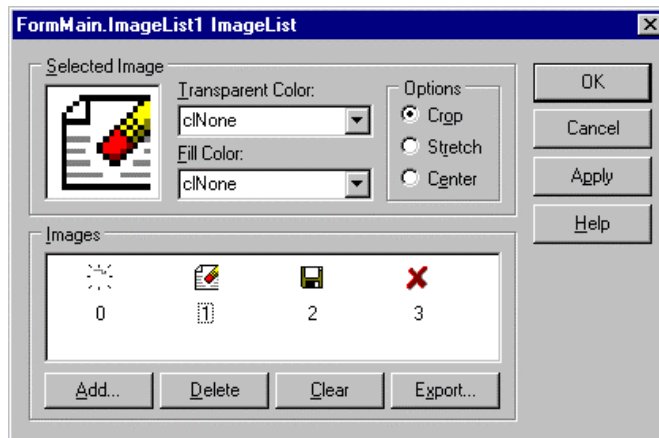
This completes our work with the business object model, so next we can set up the GUI to display the data using a combination of the `TtiTreeView`, `TtiListView` and `TtiPerAware` controls.

Setup the GUI to display the object hierarchy

Set-up the main form

Add a TToolBar, TActionList, TMainMenu, TStatusBar and TImageList to the form. Name the TToolBar TB, the TMainMenu MM, the TStatusBar SB, and the TImageList to ILButtons (we will eventually have 2 image lists on this form). Set the TToolBar's Flat property to true, and change its height to 25. Add four buttons and two separators to the toolbar, positioning the separators before the 3rd and 4th buttons. Now, add a Close menu item to the File menu, and New, Delete and Save menu items to the Edit option of the main menu you added. We will hook up the appropriate actions in a moment.

Double click the image list and add the New, Delete, Save and Cancel images from the \TechInsite\Images\Buttons directory. Arrange the images in order as shown below:



Double click the Action list and add four actions: aNew, aDelete, aSave and aClose. Setup their captions to read &New, &Delete, &Save and &Close, with their shortcuts being Ins, Del, Ctrl+S and Alt+F4. Note, that we can't actually assign Alt+F4 as a shortcut key from the Object Inspector; we must do this by writing the following piece of code in the main form's FormCreate event handler: -

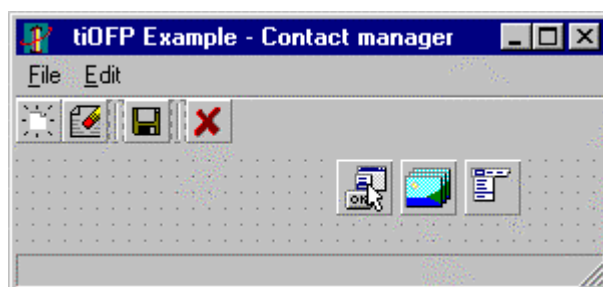
```

Procedure TFormMain.FormCreate(Sender : TObject);
Begin
    aClose.ShortCut := ShortCut(VK_F4, [ssAlt]);
End;
    
```

The Shortcut function is defined in the Menus unit, so don't forget to add this to the uses clause, or your code will not compile.

Assign the image index properties of each action to match its image index in the image list. Set the Hint properties to 'New', 'Delete', 'Save' and 'Close'. Double click each action and create an OnClick event handler. Wire up the actions to the tool buttons and main menu items so the form looks like the one shown below. Run and test.

Add the single command Close to the aCloseExecute event (to close the form and shut down the application.)



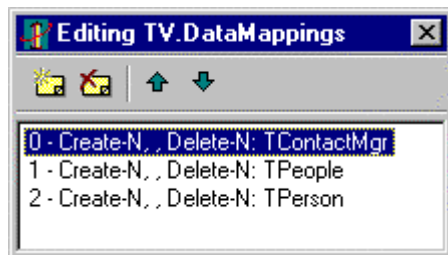
Add a TtiTreeView

Go to the TechInsite tab on the component pallet and add a TtiTreeView, which we shall name TV. Set its HasChildForms property to true, and SplitterVisible property to true. You can set its align property to alClient, but I prefer to do this in the form's constructor because it leaves the form less cluttered at design time.

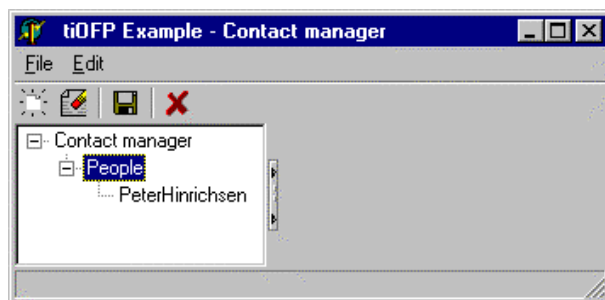
While we are in the form's constructor, set the tree view's Data property to point to the single instance of the TContactMgr. The form's constructor will look like this:

```
procedure TFormMain.FormCreate(Sender: TObject);
begin
  aClose.ShortCut := ShortCut(VK_F4, [ssAlt]);
  TV.Align := alClient;
  TV.SplitterPos := 110;
  TV.Data := gContactMgr;
end;
```

Next, we will configure the tree view to display the necessary nodes of the object hierarchy, and hide the nodes we do not want to display. Click on the tree view's DataMappings property and add three mappings to the collection editor. Set the mappings DataClass properties to TContactMgr, TPeople and TPerson. You will end up with a collection editor that looks like this:



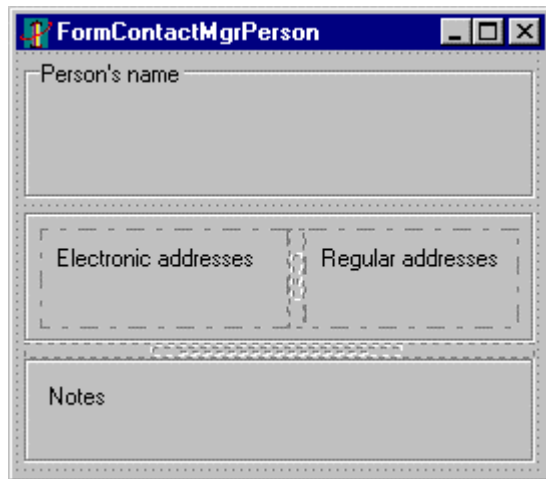
Run the application and you should see a tree view displaying the three levels of the contact manager class hierarchy: TContactMgr, TPeople and TPerson as shown below:



Next, we shall add a form to display the details of the TPerson on the right hand side of the tree view.

Add a form to display the currently selected node on the TtiTreeView

Add a new form to the project and save it with the file name FContactMgr_Person. (Make sure that it is not added to the projects list of auto create forms.) Give the form the name FormContactMgrPerson then add two TtiSplitterPanel(s), one inside the other, and three TLabel(s) to the form. Arrange the as shown below and set their Anchors properties so the panels resize as expected when the form is resized. (Note, you can change the properties of the TtiSplitterPanel by double clicking.)



The TtiTreeView can have a form associated with each data type it will display. When a node is selected, if it is registered against a form the form will be shown to the right hand side of the tree view. The tree view controls the form using RTTI so it must have a standard interface as shown in the code stub below, which can be found in the unit tiTreeView.pas.

```
{
  // This stub of code defines the interface a child form
  // must have if it is to be displayed as in the right
  // hand pane of the TtiTreeView
  TMyChildForm = class(TForm)
  private
    FData: TPersistent; // Can be any TPersistent descendant
    FTreeNode: TTreeNode;
    function GetValid: boolean;
    procedure SetData(const Value: TPersistent);
  published
    property Data : TPersistent read FData write SetData ;
    property Valid : boolean read GetValid ;
    property TreeNode : TTreeNode read FTreeNode write FTreeNode ;
  public
    end;
}
```

Paste this code into the interface section of the newly create form and change the type of FData from TPersistent to TPerson. Add ContactMgr_BOM and ComCtrls to the unit's uses clause then press Ctlr+Shift+C so Delphi will create the implementation of the unit. (ComCtrls.pas is required because we have added a reference to a TTreeNode, which is defined in this unit.)

Add a TtiSplitterPanel to the form so we will be able to identify if when it is displayed by the tree view. Arrange the TtiSplitterPanel so its sides are about 8 pixels in from each edge, and then set its top, left, bottom and right anchors to true. Double click the TtiSplitterPanel and set its SplitterAlignment property to horizontal. Next, add a stub of implementation to the SetData and Valid methods as shown below:

```
function TFormContactMgrPerson.GetValid: boolean;
begin
  result := true ;
end;

procedure TFormContactMgrPerson.SetData(const Value: TPerson);
begin
  FData := Value ;
end;
```

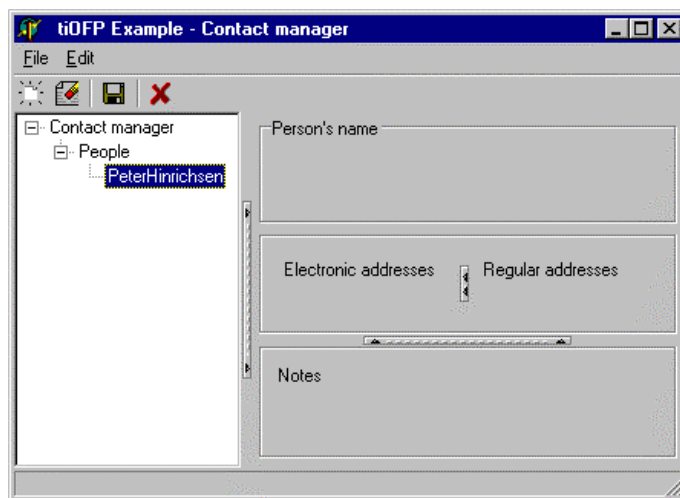
Go to the application's main form and in the OnCreate event, add the line: TV.RegisterChildForm(TPerson, TFormContactMgrPerson). This will associate the form we have just created with the TPerson class. The implementation of TformMain.OnCreate now looks like this:

```

procedure TFormMain.FormCreate(Sender: TObject);
begin
  aClose.ShortCut := ShortCut(VK_F4, [ssAlt]);
  TV.Align := alClient ;
  TV.SplitterPos := 110 ;
  // This line associates the TPerson class with
  // TFormContactMgrPerson for display and editing
  TV.RegisterChildForm( TPerson, TFormContactMgrPerson ) ;
  TV.Data := gContactMgr ;
end;

```

Add FContactMgr_Person and ContactMgr_BOM to the main form's uses clause then compile and test the application; you should see the form we created appear when a TPerson is selected. There should be no form visible on the right hand side of the tree view when a node that is not a TPerson is selected. The application should look this when it is running:



Next we will add some TPersistent aware controls to display the data associated with a TPerson.

Setup a form to display all the details of a TPerson

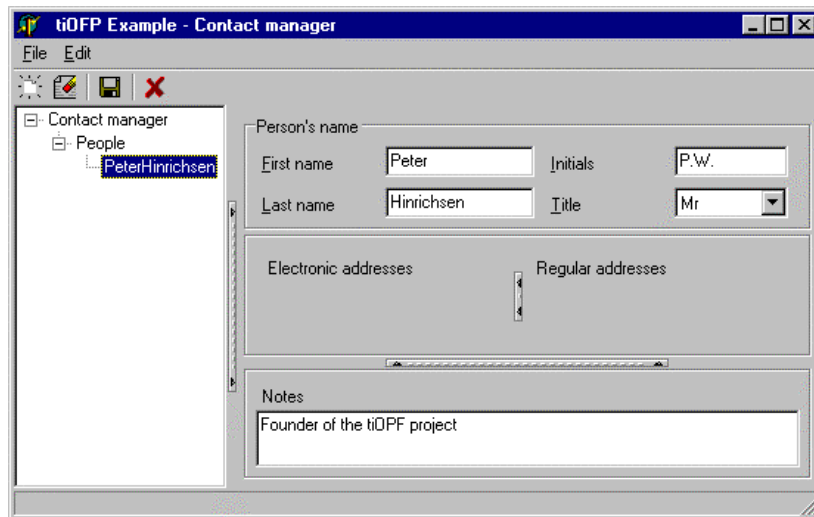
We must display five 'flat' properties including FirstName, LastName, Initials, Title and Notes. We will display the first three with TtiPerAwareEdit controls, the title with a TtiPerAwareComboBox and Notes with a TtiPerAwareMemo. Add these controls to the form, then go to the SetData method and connect up the controls to the data property like this:

```

procedure TFormContactMgrPerson.SetData(const Value: TPerson);
begin
  FData := Value ;
  paeFirstName.LinkToData( FData, 'FirstName' ) ;
  paeLastName.LinkToData( FData, 'LastName' ) ;
  paeInitials.LinkToData( FData, 'Initials' ) ;
  paeTitle.LinkToData( FData, 'Title' ) ;
  paeNotes.LinkToData( FData, 'Notes' ) ;
end;

```

Run the application and when you select a person note in the tree view, you should see this:



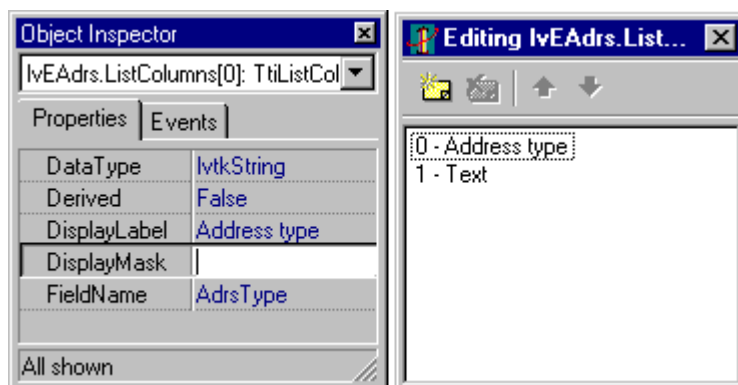
The next step is to add two TtiListView(s) and connect them up to display the TAdrs(s) and TEAdrs(s). Add the TtiListView(s) and call them lvAdrs and lvEAdrs. Set their all their anchors properties to true connect them up to the TPerson in the SetData method like this:

```
procedure TFormContactMgrPerson.SetData(const Value: TPerson);
begin
    // Snip
    if FData = nil then
    begin
        lvAdrs.Data := nil ;
        lvEAdrs.Data := nil ;
        Exit ; //==>
    end ;
    lvAdrs.Data := FData.AdrsList.List ;
    lvEAdrs.Data := FData.EAdrsList.List ;
end;
```

Select lvEAdrs and click the ListColumns property in the property editor. The ListColumn collection editor will open. Add two collection items and set their FieldName and DisplayLabel properties as below:

Collection Item	Field Name	Display Label
0	AdrsType	Address type
1	Caption	Text

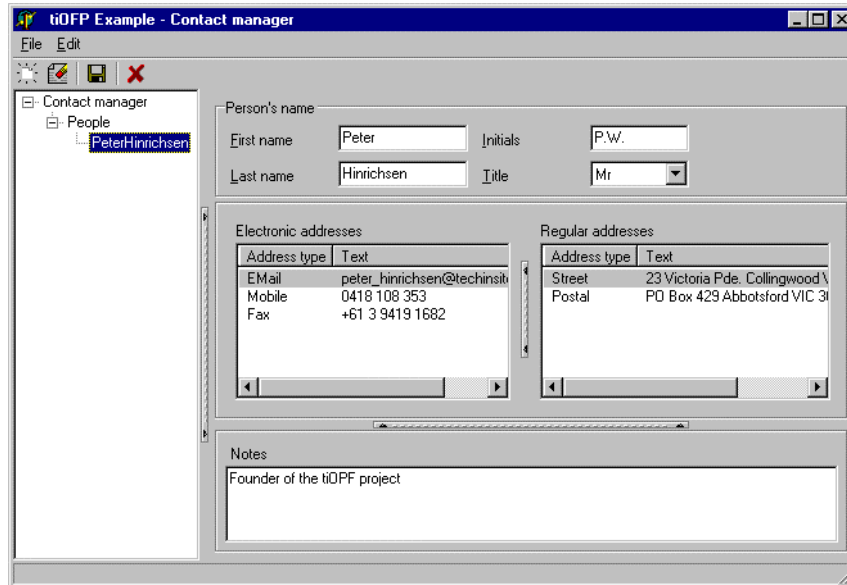
The collection editor for the ListColumns property, along with the property editor for collection item 0 are shown below:



Repeat the process for lvAdrs and enter the values shown below:

Collection Item	FieldName	DisplayLabel
0	AdrsType	Address type
1	Caption	Text

Set the RunTimeGenCols property on both list views to false then compile the application and the main form should now look like this:



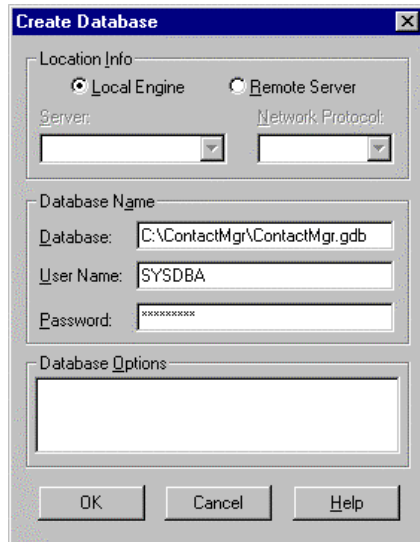
Now that we have created the BOM, and build a GUI to display the data, we can create an Interbase database to save the objects to, and then write the Visitors to manage the mapping of objects to tables and properties to columns. After that, we will add edit and delete features to the GUI, and implement Visitors to save the objects.

Write the SQL create script for the database

In the section on database schema and object – database mapping, we identified three persistent objects that require three tables to store their information. Remind yourself of the way we decided to map objects to tables and properties to columns. The SQL schema to implement this will be written next, but first we must create an Interbase database, and then connect to it using a tool for running adhoc SQL statements.

Create the Interbase database

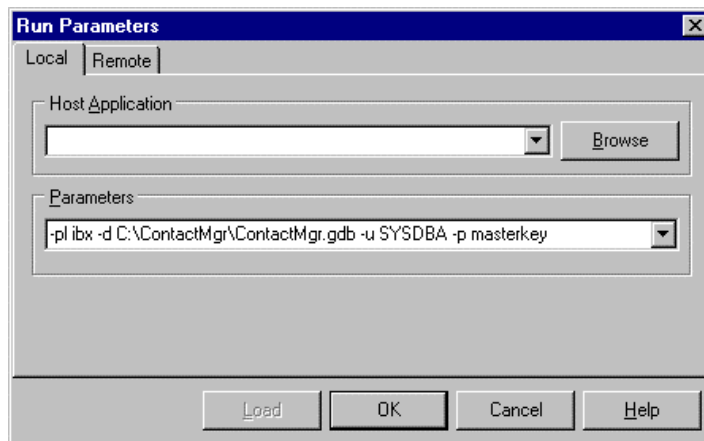
Make sure the Interbase server is running, and then load WinISQL (or what ever tool you prefer for creating an Interbase database) the dialog to create a database in WinISQL is shown below:



This will create an Interbase database called ContactMgr.gdb in the directory C:\ContactMgr\

Compile the tiSQLEditor

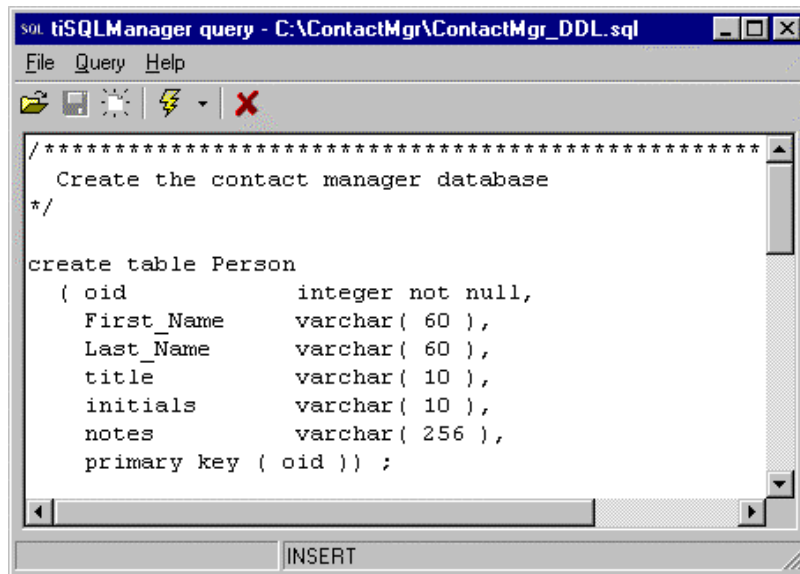
Add the application C:\TechInsite\SupportApps\tiSQLManager\tiSQLEditor.dpr to the project group. Check the command line parameters are entered as shown below:



The command line parameters have the following meanings:

Parameter switch	Value	Meaning
-pl	IBX	Persistence layer name
-d	C:\ContactMgr\ContactMgr.gdb	Database name
-u	SYSDBA	User name
-p	masterkey	Password

The tiSQLEditor window should show like the one shown below:



Enter the SQL create script shown below, then save it to C:\ContactMgr\ContactMgr_DDL.SQL.

```

/*****
  Create the contact manager database
*/

create table Next_OID
( oid          integer not null);

create table Person
( oid          integer not null,
  First Name   varchar( 60 ),
  Last_Name    varchar( 60 ),
  title        varchar( 10 ),
  initials     varchar( 10 ),
  notes        varchar( 256 ),
  primary key ( oid ) );

create table adrs
( oid          integer not null,
  owner_oid    integer not null,
  adrs_type    varchar( 20 ),
  lines        varchar( 180 ),
  suburb       varchar( 30 ),
  state        varchar( 30 ),
  pcode        varchar( 20 ),
  country      varchar( 30 ),
  primary key( oid ) );

alter table adrs
add foreign key ( owner_oid ) references Person( OID ) ;

create table EAdrs
( oid          integer not null,
  owner_oid    integer not null,
  eadrs_type   varchar( 20 ),
  text         varchar( 60 ),
  primary key( oid ) );

alter table EAdrs
add foreign key ( owner_oid ) references Person( OID ) ;

commit ;

```

Note the first table created NEXT_OID. This is required by all databases the framework uses. It provides a central point for handing out Object Identifiers (OIDs). Without this table you will not be able to create any new objects.

You can run this script using either one of the tools that come with Interbase, or by selecting each statement, one at the time in the tiSQLEditor then clicking the run button. (This is my preferred approach for small scripts)

Insert some test data


Create another script in the tiSQLEditor and save it as ContactMgr_TestData.sql. Enter the SQL as shown below, then run the script, or run each statement one at the time by selecting it and pressing F8.

```
/*  
*****  
Create the contact manager test data  
*/  
  
insert into Person values  
  ( 1, 'Peter', 'Hinrichsen', 'Mr.', 'P.W.', 'The founder of the tiOPF project' ) ;  
  
insert into adrs values  
  ( 2, 1, 'Street', '23 Victoria Pde.', 'Collingwood', 'VIC', '3066', 'Australia' ) ;  
  
insert into adrs values  
  ( 3, 1, 'Postal', 'PO Box 429', 'Abbotsford', 'VIC', '3067', 'Australia' ) ;  
  
insert into EAdrs values  
  ( 4, 1, 'EMail', 'peter_hinrichsen@techinsite.com.au' ) ;  
  
insert into EAdrs values  
  ( 5, 1, 'Mobile', '0418 108 353' ) ;  
  
insert into EAdrs values  
  ( 6, 1, 'Fax', '+61 3 9419 1682' ) ;  
  
commit ;
```

Test that the data has been correctly inserted by running the following queries in the tiSQLEditor:

```
select * from person ;  
select * from adrs ;  
select * from eadrs ;
```

The results of the third select statement are shown below:



OID	OWNER_OID	EADRS_TYPE	TEXT
4	1	EEmail	peter_hinrichsen@techinsite.com.au
5	1	EMobile	0418 108 353
6	1	EFax	+61 3 9419 1682

Record count: 3 Time to execute on server: 0ms Time to dowr

Now that we have created a database, and inserted some test data we can look at the various strategies available for mapping the database into the business objects we created.

Alternative BOM – database mapping strategies

There are three strategies available for mapping objects to a relational database:

1. Hard code the SQL into the application, and use the Visitor framework to map the SQL to the objects. This is implemented by linking in the unit Adrs_SrvHardCodeSQL.pas

2. Use the tiSQLManager application to maintain the SQL outside the application. This has the advantage of decoupling the SQL from the application, but the disadvantage of forcing you to add the tiSQLManager tables to the database. (This will be corrected when we have an XML persistence layer available)
3. Set up mappings between the objects and tables, properties and columns and let the persistence framework generate the SQL for you. This can be implemented by linking in Adrs_SrvAutoGenSQL.pas (This strategy is under construction, and this example will work in most classes. There are some problems with BLOBS, and the way the mappings are defined is a little messy)

In summary, you must link in one and only one of the three unit files. The best way of achieving this is to create a dependencies unit that will provide a central point for determining the strategy to be used. Create a new unit called ContactMgr_Dependencies, which looks like this: -

```
Unit ContactMgr_Dependencies;

Interface
Uses
    ContactMgr_BOM,
    //ContactMgr_SvrHardCode
    //ContactMgr_SvrSQLMgr
    ContactMgr_SvrAutoMap
;

Implementation
End.
```

As you can see, the above example is using the auto-map option, but switching options is as simple as un-commenting the relevant unit (assuming you have created these units in the first place!). You now need to add this to the project source. Select Project...View Source and modify the source so it looks like the listing below: -

```
Program ContactMgr;

Uses
    tiLog,
    tiPersist,
    Forms,
    FMain In 'FMain.pas' {frmMain},
    ContactMgr_BOM In 'ContactMgr_BOM.pas',
    ContactMgr_Cli In 'ContactMgr_Cli.pas',
    ContactMgr_Tst In 'ContactMgr_Tst.pas',
    FContactMgr_Person In 'FContactMgr_Person.pas' {TFormContactMgr_Person},
    ContactMgr_Dependencies In 'ContactMgr_Dependencies.pas';

{$R *.RES}

Begin
    SetupLogForClient;
    Application.Initialize;
    gTIPerMgr.LoadPersistenceFramework;
    Application.CreateForm(TfrmMain, frmMain);
    Application.Run;
    gTIPerMgr.UnloadPersistenceFramework;
End.
```

Hard code the SQL

Create a new unit called ContactMgr_SvrHardCode.pas

Add tiPtnVisSQL to the visitor and type the interface as shown below:

Hit Ctrl+Shift+C to fill in the implementation

Add ContactMgr_BOM, tiPersist and tiPtnVisPerObj to the implementation uses clause.

```
unit ContactMgr_SvrHardCode;

interface
uses
  tiPtnVisSQL
  ;

type

  TVisPeopleReadHardCode = class( TVisOwnedQrySelect )
  protected
    function AcceptVisitor : boolean ; override ;
    procedure Init ; override ;
    procedure SetupParams ; override ;
    procedure MapRowToObject ; override ;
  end ;

implementation
uses
  ContactMgr_BOM
  , tiPersist
  , tiPtnVisPerObj
  ;
```

In the implementation of AcceptVisitor, type result := Visited is TPerson

```
{ TVisPeopleReadHardCode }

function TVisPeopleReadHardCode.AcceptVisitor: boolean;
begin
  result := ( Visited is TPerson ) ;
end;
```

In the implementation of Init, enter the SQL to select all the people. This can be simplified by entering SELECT * FROM PEOPLE in the tiSQLEditor and executing the query. There is then an option to copy the SQL to the clipboard with the delimiters and line breaks added. The result is shown below:

```
procedure TVisPeopleReadHardCode.Init;
begin
  Query.SQL.Text :=
    'select ' +
    '   OID ' +
    '   ,FIRST_NAME ' +
    '   ,LAST_NAME ' +
    '   ,TITLE ' +
    '   ,INITIALS ' +
    '   ,NOTES ' +
    'from ' +
    '   person ' ;
end;
```

Enter the MapRowToObject implementation as shown below. This is made easier by executing the SELECT SQL, then using the SQL | MapRowToObject to copy the necessary code to the clipboard and pasting it into Delphi. The code looks like this as it is copied to the clipboard.

```
procedure TVisEAdrsReadHardCode.MapRowToObject;
var
  lData : T<Class type> ;
begin
  lData := T<Class type>.Create ;
```

```
lData.Oid.AssignFromTiQuery(Query);
lData.FirstName := Query.FieldAsString[ 'FIRST_NAME' ] ;
lData.LastName := Query.FieldAsString[ 'LAST_NAME' ] ;
lData.Title := Query.FieldAsString[ 'TITLE' ] ;
lData.Initials := Query.FieldAsString[ 'INITIALS' ] ;
lData.Notes := Query.FieldAsString[ 'NOTES' ] ;
lData.ObjectState := posClean ;
TPerVisList( Visited ).Add( lData ) ;
end;
```

All you have to do is replace the T<Class type> place holders with TPerson and the result looks like this:

```
procedure TVisPeopleReadHardCode.MapRowToObject;
var
  lData : TPerson ;
begin
  lData := TPerson.Create ;
  lData.Oid.AssignFromTiQuery(Query);
  lData.FirstName := Query.FieldAsString[ 'FIRST_NAME' ] ;
  lData.LastName := Query.FieldAsString[ 'LAST_NAME' ] ;
  lData.Title := Query.FieldAsString[ 'TITLE' ] ;
  lData.Initials := Query.FieldAsString[ 'INITIALS' ] ;
  lData.Notes := Query.FieldAsString[ 'NOTES' ] ;
  lData.ObjectState := posClean ;
  TPerVisList( Visited ).Add( lData ) ;
end;
```

SetupParams has no code in the implementation:

```
procedure TVisPeopleReadHardCode.SetupParams;
begin
  // Do nothing
end;
```

Register the visitor with the persistence framework like this:

```
initialization
  gTPerMgr.RegReadVisitor( TVisPeopleReadHardCode ) ;
end.
```

The 12 Visitors that comprise the full implementation of the Contact Manager persistence visitors, implemented by hard coding SQL can be found in ContactMgr_SrvHardCode.pas.

Use the tiSQLManager

Create the tiSQLManager tables

```
create domain domain_oid as integer not null ;
create table Next_OID ( oid domain_oid ) ;
insert into next_oid ( oid ) values ( 100000 ) ;

create domain domain_sql as blob sub_type 1 ;
create domain domain_str20 as varchar( 20 ) not null ;
create domain domain_str50 as varchar( 50 ) not null ;
create domain domain_str100 as varchar( 100 ) ;
create domain domain_integer as integer default 0 not null ;
create domain domain_boolean as char( 1 ) default 'F' check ( value in ( 'T', 'F' ) ) ;

/* SQLMan_Group table */

create table sqlman_group
( oid domain_oid,
```

```

group name    domain str50,
disp_order   domain_integer,
primary key ( oid ) ;

/* SQLMan_SQL table */
create table sqlman_sql
( oid          domain_oid,
  group_oid    domain_oid,
  disp_order   domain_integer,
  query version domain_integer,
  query name   domain_str50,
  query description domain_sql,
  query_locked domain_boolean,
  test_include domain_boolean,
  sql          domain_sql,
primary key ( oid ) ;

/* SQLMan Param Table */
create table sqlman_param
( oid          domain_oid,
  sql_oid      domain_oid,
  disp_order   domain_integer,
  param name   domain_str20,
  param type   domain_str20,
  param_value  domain_str50,
  param_isnull domain_boolean,
primary key ( oid ) ;

/* SQLMan Interface Table */
create table sqlman_interface
( oid          domain_oid,
  sql_oid      domain_oid,
  disp_order   domain_integer,
  field name   domain_str20,
  field type   domain_str20,
primary key ( oid ) ;

/* Add ref integ */
alter table SQLMan_SQL
add foreign key ( Group_OID ) references SQLMan_Group ( OID ) ;

alter table SQLMan_Param
add foreign key ( SQL_OID ) references SQLMan_SQL ( OID ) ;

alter table SQLMan_Interface
add foreign key ( SQL_OID ) references SQLMan_SQL ( OID ) ;

/* Add unique constraint */
create unique index I_SQLMan_Group
on SQLMan_Group
( Group Name ) ;

create unique index I_SQLMan_SQL
on SQLMan_SQL
( Query_Name ) ;

create unique index I_SQLMan_Param
on SQLMan_Param
( SQL_OID, Param_Name ) ;

create unique index I_SQLMan_Interface
on SQLMan_Interface
( SQL OID, Field Name ) ;

insert into sqlman_group
( oid, group_name, disp_order ) values
( 1, 'System', 0 ) ;

insert into sqlman_group values ( 2, 'Queries', 0 ) ;

commit ;

```

Write the tiSQLManager Visitors

```
Unit ContactMgr_SvrSQLMgr;
```

```

interface
uses
    tiPtnVisSQL
    ;

type

    // * * * * *
    // *
    // * Read visitors
    // *
    // * * * * *
    TVisPeopleReadSQLMgr = class( TVisQrySelect )
    protected
        function    AcceptVisitor : boolean ; override ;
        procedure  Init           ; override ;
        procedure  SetupParams    ; override ;
        procedure  MapRowToObject ; override ;
    end ;

    // * * * * *
    // *
    // * Delete visitors
    // *
    // * * * * *
    TVisPeopleDeleteSQLMgr = class( TVisQryUpdate )
    protected
        function    AcceptVisitor : boolean ; override ;
        procedure  Init           ; override ;
        procedure  SetupParams    ; override ;
    end ;

    // * * * * *
    // *
    // * Update visitors
    // *
    // * * * * *
    TVisPeopleUpdateSQLMgr = class( TVisQryUpdate )
    protected
        function    AcceptVisitor : boolean ; override ;
        procedure  Init           ; override ;
        procedure  SetupParams    ; override ;
    end ;

    // * * * * *
    // *
    // * Create visitors
    // *
    // * * * * *
    TVisPeopleCreateSQLMgr = class( TVisQryUpdate )
    protected
        function    AcceptVisitor : boolean ; override ;
        procedure  Init           ; override ;
        procedure  SetupParams    ; override ;
    end ;

implementation
uses
    ContactMgr BOM
    , tiPersist
    , tiPtnVisPerObj
    , cQueryNames
    ;

    { TVisPeopleReadSQLMgr }

    function TVisPeopleReadSQLMgr.AcceptVisitor: boolean;
    begin
        result := ( Visited is TPeople ) ;
    end;

    procedure TVisPeopleReadSQLMgr.Init;
    begin
        QueryName := cgQryPerson_Read ;
    end;

    procedure TVisPeopleReadSQLMgr.MapRowToObject;
    var

```

```

lData : TPerson ;
begin
lData := TPerson.Create ;
lData.Oid.AssignFromTiQuery(Query);
lData.FirstName := Query.FieldAsString[ 'FIRST NAME' ] ;
lData.LastName := Query.FieldAsString[ 'LAST NAME' ] ;
lData.Title := Query.FieldAsString[ 'TITLE' ] ;
lData.Initials := Query.FieldAsString[ 'INITIALS' ] ;
lData.Notes := Query.FieldAsString[ 'NOTES' ] ;
lData.ObjectState := posClean ;
TPerVisList( Visited ).Add( lData ) ;
end;

procedure TVisPeopleReadSQLMgr.SetupParams;
begin
// Do nothing
end;

{ TVisPeopleDeleteSQLMgr }

function TVisPeopleDeleteSQLMgr.AcceptVisitor: boolean;
begin
result := ( Visited is TPerson ) and
( Visited.ObjectState = posDelete ) ;
end;

procedure TVisPeopleDeleteSQLMgr.Init;
begin
QueryName := cgQryPerson Delete ;
end;

procedure TVisPeopleDeleteSQLMgr.SetupParams;
begin
Visited.OID.AssignToTiQuery(Query);
end;

{ TVisPeopleUpdateSQLMgr }

function TVisPeopleUpdateSQLMgr.AcceptVisitor: boolean;
begin
result := ( Visited is TPerson ) and
( Visited.ObjectState = posUpdate ) ;
end;

procedure TVisPeopleUpdateSQLMgr.Init;
begin
QueryName := cgQryPerson Update ;
end;

procedure TVisPeopleUpdateSQLMgr.SetupParams;
var
lData : TPerson ;
begin
lData := TPerson( Visited ) ;
lData.Oid.AssignToTiQuery(Query);
Query.ParamAsString[ 'FIRST_NAME' ] := lData.FirstName ;
Query.ParamAsString[ 'LAST_NAME' ] := lData.LastName ;
Query.ParamAsString[ 'TITLE' ] := lData.Title ;
Query.ParamAsString[ 'INITIALS' ] := lData.Initials ;
Query.ParamAsString[ 'NOTES' ] := lData.Notes ;
end;

{ TVisPeopleCreateSQLMgr }

function TVisPeopleCreateSQLMgr.AcceptVisitor: boolean;
begin
result := ( Visited is TPerson ) and
( Visited.ObjectState = posCreate ) ;
end;

procedure TVisPeopleCreateSQLMgr.Init;
begin
QueryName := cgQryPerson_Create ;
end;

procedure TVisPeopleCreateSQLMgr.SetupParams;
var

```

```

lData : TPerson ;
begin
lData := TPerson( Visited ) ;
lData.Oid.AssignToTiQuery(Query);
Query.ParamAsString[ 'FIRST NAME' ] := lData.FirstName ;
Query.ParamAsString[ 'LAST NAME' ] := lData.LastName ;
Query.ParamAsString[ 'TITLE' ] := lData.Title ;
Query.ParamAsString[ 'INITIALS' ] := lData.Initials ;
Query.ParamAsString[ 'NOTES' ] := lData.Notes ;
end;

initialization

gTIPerMgr.RegReadVisitor( TVisPeopleReadSQLMgr ) ;

gTIPerMgr.RegSaveVisitor( TVisEAdrsDeleteSQLMgr ) ;

gTIPerMgr.RegSaveVisitor( TVisPeopleUpdateSQLMgr ) ;

gTIPerMgr.RegSaveVisitor( TVisPeopleCreateSQLMgr ) ;

end.

```

Auto generate the SQL

```

unit ContactMgr_SvrAutoMap;

interface

implementation
uses
  tiPersist
  , tiClassToDBMap_BOM
  , ContactMgr_BOM
  ;

initialization

gTIPerMgr.ClassDBMappingMgr.RegisterMapping( TPerson, 'Person', 'OID', 'OID', [pktDB] ) ;
gTIPerMgr.ClassDBMappingMgr.RegisterMapping( TPerson, 'Person', 'LastName', 'Last_Name' ) ;
gTIPerMgr.ClassDBMappingMgr.RegisterMapping( TPerson, 'Person', 'FirstName', 'First Name' ) ;
gTIPerMgr.ClassDBMappingMgr.RegisterMapping( TPerson, 'Person', 'Title', 'Title' ) ;
gTIPerMgr.ClassDBMappingMgr.RegisterMapping( TPerson, 'Person', 'Initials', 'Initials' ) ;
gTIPerMgr.ClassDBMappingMgr.RegisterMapping( TPerson, 'Person', 'Notes', 'Notes' ) ;
gTIPerMgr.ClassDBMappingMgr.RegisterCollection( TPeople, TPerson ) ;

gTIPerMgr.ClassDBMappingMgr.RegisterMapping( TAdrsList, 'Adrs', 'OID', 'OID', [pktDB] ) ;
gTIPerMgr.ClassDBMappingMgr.RegisterMapping( TAdrs, 'Adrs', 'OID', 'OID', [pktDB] ) ;
gTIPerMgr.ClassDBMappingMgr.RegisterMapping( TAdrs, 'Adrs', 'AdrsType', 'Adrs Type' ) ;
gTIPerMgr.ClassDBMappingMgr.RegisterMapping( TAdrs, 'Adrs', 'Lines', 'Lines' ) ;
gTIPerMgr.ClassDBMappingMgr.RegisterMapping( TAdrs, 'Adrs', 'Suburb', 'Suburb' ) ;
gTIPerMgr.ClassDBMappingMgr.RegisterMapping( TAdrs, 'Adrs', 'State', 'State' ) ;
gTIPerMgr.ClassDBMappingMgr.RegisterMapping( TAdrs, 'Adrs', 'PCode', 'PCode' ) ;
gTIPerMgr.ClassDBMappingMgr.RegisterMapping( TAdrs, 'Adrs', 'Country', 'Country' ) ;
gTIPerMgr.ClassDBMappingMgr.RegisterCollection( TAdrsList, TAdrs, ['Owner_OID'] ) ;

gTIPerMgr.ClassDBMappingMgr.RegisterMapping( TEAdrsList, 'EAdrs', 'OID', 'OID', [pktDB] ) ;
gTIPerMgr.ClassDBMappingMgr.RegisterMapping( TEAdrs, 'EAdrs', 'OID', 'OID', [pktDB] ) ;
gTIPerMgr.ClassDBMappingMgr.RegisterMapping( TEAdrs, 'EAdrs', 'AdrsType', 'EAdrs Type' ) ;
gTIPerMgr.ClassDBMappingMgr.RegisterMapping( TEAdrs, 'EAdrs', 'Text', 'Text' ) ;
gTIPerMgr.ClassDBMappingMgr.RegisterCollection( TEAdrsList, TEAdrs, ['Owner_OID'] ) ;

end.

```

Write the GUI and hook it up to the BOM

TFormMain

```

implementation
uses

```

```

tiPtnVisPerObj
,tiPtnVisPerObj_Cli
,tiPersist
,ContactMgr_Cli
,ContactMgr_BOM
,FContactMgrChild Person
,tiUtils
,tiRegIni
,Menus
;

{$R *.DFM}

procedure TFormMain.aInsertExecute(Sender: TObject);
begin
    TV.DoInsert ;
end;

procedure TFormMain.aDeleteExecute(Sender: TObject);
begin
    TV.DoDelete ;
end;

procedure TFormMain.aSaveExecute(Sender: TObject);
begin
    gTIPerMgr.Save( gContactMgr ) ;
end;

procedure TFormMain.aCloseExecute(Sender: TObject);
begin
    Close ;
end;

procedure TFormMain.FormCreate(Sender: TObject);
begin
    aClose.ShortCut := ShortCut(VK_F4, [ssAlt]);
    gReg.ReadFormState( Self ) ;
    TV.Align := alClient ;
    TV.RegisterChildForm( TPerson, TFormChildPerson ) ;
    TV.Data := gContactMgr ;
end;

procedure TFormMain.tiTVMappingPersonOnDelete(ptiTreeView: TtiTreeView;
pNode: TTreeNode; pData: TObject; pParentNode: TTreeNode;
pParentData: TObject);
begin
    tiPerObjAbsConfirmAndDelete( pData as TPerObjAbs ) ;
end;

procedure TFormMain.tiTVMappingPersonOnInsert(ptiTreeView: TtiTreeView;
pNode: TTreeNode; pData: TObject; pParentNode: TTreeNode;
pParentData: TObject);
var
    lData : TPerson ;
begin
    lData := TPerson.CreateNew ;
    ( pParentData as TPeople ).Add( lData ) ;
end;

procedure TFormMain.tiTVMappingPeopleOnInsert(ptiTreeView: TtiTreeView;
pNode: TTreeNode; pData: TObject; pParentNode: TTreeNode;
pParentData: TObject);
var
    lData : TPerson ;
begin
    lData := TPerson.CreateNew ;
    ( pData as TPeople ).Add( lData ) ;
end;

procedure TFormMain.TVFilterData(pData: TObject; var pbInclude: Boolean);
begin
    pbInclude := not ( pData as TPerObjAbs ).Deleted ;
end;

procedure TFormMain.aShowPerObjAbsExecute(Sender: TObject);
begin
    tiShowPerObjAbs( TV.SelectedData as TPerObjAbs ) ;

```



```

end;

procedure TFormMain.ALUpdate(Action: TBasicAction; var Handled: Boolean);
begin
  aInsert.Enabled := TV.CanInsertSelected ;
  aDelete.Enabled := TV.CanDeleteSelected ;
  aSave.Enabled := gContactMgr.Dirty ;
end;

procedure TFormMain.FormDestroy(Sender: TObject);
begin
  gReg.WriteFormState( Self ) ;
end;

```

TformContactChildPerson

```

private
  FData: TPerson ;
  FTreeNode: TTreeNode;
  function GetValid: boolean;
  procedure SetData(const Value: TPerson );
published
  property Data : TPerson read FData write SetData ;
  property Valid : boolean read GetValid ;
  property TreeNode : TTreeNode read FTreeNode write FTreeNode ;
public
end;

implementation
uses
  tiPtnVisPerObj
  , FAdrsEdit
  , FEAdrsEdit
  ;

{$R *.DFM}

{ TForm1 }

function TFormChildPerson.GetValid: boolean;
begin
  result := true ;
end;

procedure TFormChildPerson.SetData(const Value: TPerson );
begin
  FData := Value ;

  paeFirstName.LinkToData( FData, 'FirstName' ) ;
  paeLastName.LinkToData( FData, 'LastName' ) ;
  paeInitials.LinkToData( FData, 'Initials' ) ;
  paeTitle.LinkToData( FData, 'Title' ) ;
  paeNotes.LinkToData( FData, 'Notes' ) ;

  if FData = nil then
  begin
    lvAdrs.Data := nil ;
    lvEAdrs.Data := nil ;
    Exit ; //==>
  end ;
  lvAdrs.Data := FData.AdrsList.List ;
  lvEAdrs.Data := FData.EAdrsList.List ;

end;

procedure TFormChildPerson.paeFirstNameChange(Sender: TObject);
begin
  if FData <> nil then
    TreeNode.Text := FData.Caption ;
end;

procedure TFormChildPerson.FormShow(Sender: TObject);
begin
  if ( FData <> nil ) and

```

```

    ( FData.ObjectState = posCreate ) then
    paeFirstName.SetFocus ;
end;

procedure TFormChildPerson.lvEAdrsItemInsert(pLV: TtiCustomListView;
pData: TPersistent; pItem: TListItem);
var
    lData : TEAdrs ;
begin
    lData := TEAdrs.CreateNew ;
    if TFormEAdrsEdit.Execute( lData ) then
        FData.EAdrsList.Add( lData )
    else
        lData.Free ;
end;

procedure TFormChildPerson.lvEAdrsItemEdit(pLV: TtiCustomListView;
pData: TPersistent; pItem: TListItem);
begin
    TFormEAdrsEdit.Execute( pData as TPerObjAbs ) ;
end;

procedure TFormChildPerson.lvEAdrsItemDelete(pLV: TtiCustomListView;
pData: TPersistent; pItem: TListItem);
begin
    tiPerObjAbsConfirmAndDelete( pData as TPerObjAbs ) ;
end;

procedure TFormChildPerson.lvAdrsItemInsert(pLV: TtiCustomListView;
pData: TPersistent; pItem: TListItem);
var
    lData : TAdrs ;
begin
    lData := TAdrs.CreateNew ;
    if TFormAdrsEdit.Execute( lData ) then
        FData.AdrsList.Add( lData )
    else
        lData.Free ;
end;

procedure TFormChildPerson.lvAdrsItemEdit(pLV: TtiCustomListView;
pData: TPersistent; pItem: TListItem);
begin
    TFormAdrsEdit.Execute( pData as TPerObjAbs ) ;
end;

procedure TFormChildPerson.lvAdrsItemDelete(pLV: TtiCustomListView;
pData: TPersistent; pItem: TListItem);
begin
    tiPerObjAbsConfirmAndDelete( pData as TPerObjAbs ) ;
end;

procedure TFormChildPerson.lvEAdrsFilterData(pData: TPersistent;
var pbInclude: Boolean);
begin
    pbInclude := not ( pData as TPerObjAbs ).Deleted ;
end;

```

TFormEditAdrs

```

type
    TFormAdrsEdit = class(TFormTIPerEditDialog)
    paeLines: TtiPerAwareMemo;
    paeSuburb: TtiPerAwareEdit;
    paeState: TtiPerAwareEdit;
    paePCode: TtiPerAwareEdit;
    paeCountry: TtiPerAwareEdit;
    paeAdrsType: TtiPerAwareComboBoxStatic;
    private
    protected
        procedure SetData(const Value: TPerObjAbs); override ;
        function FormIsValid : boolean ; override ;
    public
        { Public declarations }
    end;

```

```
implementation

{$R *.DFM}

{ TFormAdrsEdit }

function TFormAdrsEdit.FormIsValid: boolean;
begin
    result := true ;
end;

procedure TFormAdrsEdit.SetData(const Value: TPerObjAbs);
begin
    inherited SetData( Value ) ;
    paeAdrsType.LinkToData( DataBuffer, 'AdrsType' ) ;
    paeLines.LinkToData( DataBuffer, 'Lines' ) ;
    paeSuburb.LinkToData( DataBuffer, 'Suburb' ) ;
    paeState.LinkToData( DataBuffer, 'State' ) ;
    paePCode.LinkToData( DataBuffer, 'PCode' ) ;
    paeCountry.LinkToData( DataBuffer, 'Country' ) ;
end;
```

TFormEditEAdrs

```
type
    TFormEAdrsEdit = class(TFormTIPerEditDialog)
        paeAdrsType: TtiPerAwareComboBoxStatic;
        paeText: TtiPerAwareEdit;
    private
        { Private declarations }
    protected
        procedure SetData(const Value: TPerObjAbs); override ;
        function FormIsValid : boolean ; override ;
    end;

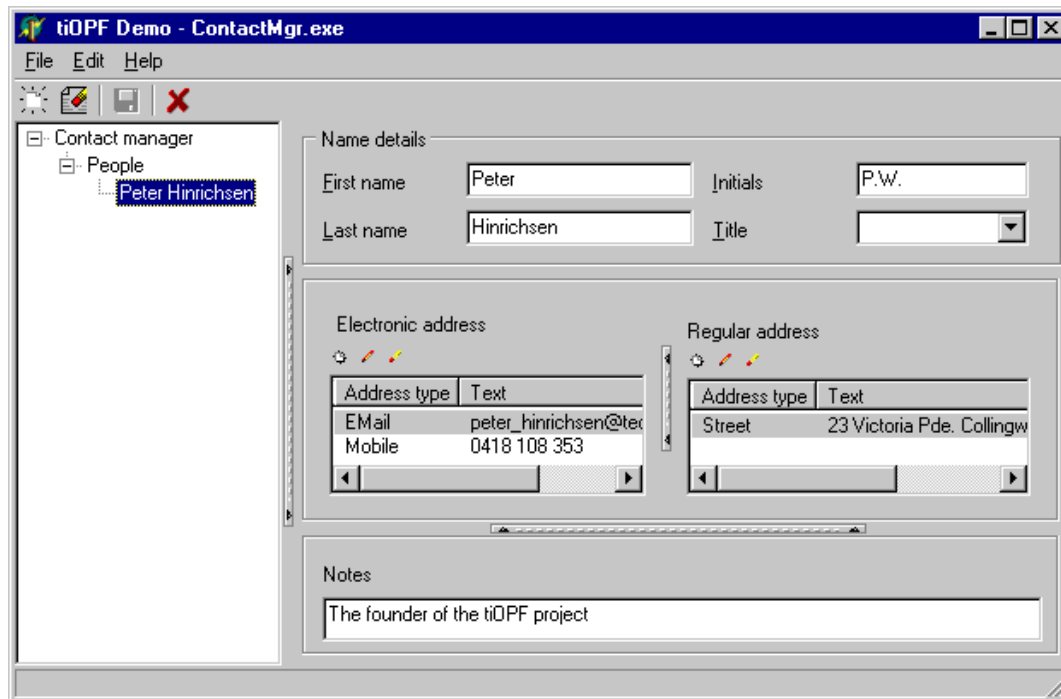
implementation

{$R *.DFM}

{ TFormEAdrsEdit }

function TFormEAdrsEdit.FormIsValid: boolean;
begin
    result := true ;
end;

procedure TFormEAdrsEdit.SetData(const Value: TPerObjAbs);
begin
    inherited SetData( Value ) ;
    paeAdrsType.LinkToData( DataBuffer, 'AdrsType' ) ;
    paeText.LinkToData( DataBuffer, 'Text' ) ;
end;
```



Summary

In this chapter we have looked at what is involved in developing a real-world application using the framework. We learnt how to design a Business Object Model (BOM) and create a suitable database schema for storage. We also learnt how to utilise some of the persistent-aware VCL controls that are provided with the framework, as well as looking at the pros and cons of the different BOM to Database mapping strategies.

The next chapter

We will discuss how to use the Adaptor pattern and runtime packages to create a swappable database connection layer.

Chapter #7

Using the Adaptor Pattern for database independence

Authors note

This chapter is a cut-and-paste from an article printed in The Delphi Magazine. It must be re-written in the context of the tiOPF but I have included it here because I have run out of time to edit it as necessary. The general principal is vital to the framework, but the examples in the article are not totally relevant. Please keep an eye on <http://www.techinsite.com.au/documentation/> for an updated version. You will notice that I step painfully through the development of a Factory again, this is because this article was written as a stand alone piece, and not as a part of a larger work. Sorry for the repetition, but I can't jus chop it out without making other structural changes.

The tiOPF framework code that uses the Adaptor can be found in the tiPersist directory. The abstract classes TtiDatabase and TtiQuery can be found in tiQuery.pas. The concrete implementation can be found in tiQueryDOA.pas, tiQueryIBX.pas and so on.

I hope the material is of use to you, and as I said, keep an eye out for updates.

Introduction

The Delphi Component Pallet is growing with every release of Delphi. Apart from the GUI controls we have had since Delphi 1, we now have several non-visual controls, all performing the same basic task, to choose from.

For example:

- For SQL database access we can choose between the BDE controls (TDatabase and TQuery) and ADO. If we are targeting a specific database, we may want to bypass a generic data connection layer and go directly to the database's API using a family of controls like IBOjects for Interbase, or DOAⁱ for Oracle.
- For Internet connection, you may have started your project with an early version of Delphi using the FastNet components, but now want to move up to WinShoes, or the more recent Indy components.

- For data compression, you may have started your project using the ZLib compression algorithm that is supplied on the Delphi CD, and now want to change to the more widely used Zip format.
- You may have started a project with no data encryption, but changed requirements mean you have to add encryption without breaking any existing systems.
- For XML parsing, you can choose between building a dependency on the MSDom DLL that comes with Internet Explorer 5, or use one of the native Delphi parsers that are available on the web (this chapter was written before the official release of Delphi 6, which now contains a native XML parser).

The next section of this chapter will discuss some of the problems of building a dependency on a single vendor's component. We then take a look at what GoFⁱⁱ say about the Adaptor Pattern and take a look at the various ways it can be implemented with Delphi. We finish by using the Adaptor to wrapper the ZLib compression library. Once we have wrapped ZLib, we examine two ways of creating concrete instances of the adapted class by using a class reference and the Factory Pattern.

How Binding to a Vendor's API Can Bite You

Say you want to use a third party component in your application to perform a task like one of those listed above. There are three ways you can use this component:

- You can drop it on a form or data module. (This technique is simple to code, but it is difficult to make changes to the application once built.)
- You can create it, use it then free it in code. (This technique is more complex to code, but easier to change the application's behaviour at compile-time.)
- You can wrapper the component in your own code giving it a standard interface at the same time, then use either a class reference variable or a Factory to create the concrete component for use. (This requires significantly more work to code, but is very versatile as the application's behaviour can be changed at compile-time or run-time.)

Dropping the component on a form or data module will work fine if there is only going to be one of them in the application. This may be the case for a FTP component where all FTP calls can be channelled through the same routine. If you want to change to another vendor's component, it is not too hard to make the necessary modification at design time and then re-compile. This technique becomes very clumsy if there are many instances of the component created at design time in different places in the application. It is difficult to use search and replace on DFM files. It will also be necessary to make changes to the units included with the application in the uses clause - this is both time consuming and error prone.

A better alternative is to create, use then free the component in code and hopefully, to route all use of the component through the same block of code. The switch from one vendor's component to another could be made by either cut & paste, or by swapping the unit containing the code that creates the component with another.

The most versatile solution is to write a component wrapper, then create the component using a class reference or Factory. This means that making the single line change where the unit that implements the appropriate concrete class is included in the project does the switch from one vendor to another. Alternatively, if a Factory is used to create the component, the change can be made at run time, which has the potential of giving the user control the behaviour of the application.

Software House & Corporate IT Departments Share the Problem

The challenge to make the right decision when selecting a component affects us in different ways depending on what sort of business we are programming for. One of my customers decided from the start that they would use Oracle so do not mind building a dependency on the DOA controls in their Delphi application. This dependency takes the form of TOracleSession and TOracleDataSet

components (the DOA equivalent to TDatabase and TQuery) being littered over the application's forms and data modules. This really does not matter to them as long as they NEVER want to change databases.

The decision to drop DOA components directly onto forms and data modules did, however cause me problems as a contractor. The library of routines and components that I take from client to client could not be used because the interface of the DOA components, while very similar to the BDE equivalents contained some annoying differences.

The same problem is potentially created as soon as you drop any component that has an interface that you do not control into an application.

For example, if you want to download a file using FTP from within a Delphi application, you may start using the FastNet TNMFTP component that has an interface like this:

```
NMFTP1.Connect ;
NMFTP1.ChangeDir( DirName : string ) ;
NMFTP1.Download( RemoteFile : String ; LocalFile : string ) ;
```

Later, you may want to move to the Indy TIdFTP component that has an interface like this:

```
IdFTP1.Connect( AutoLogin : Boolean = true ) ;
IdFTP1.ChangeDir( ADirName : string ) ;
IdFTP1.Get( ASourceFile : String ; ADest : TStream ) ;
IdFTP1.Get( ASourceFile : String ;
           ADestFile : String ;
           ACanOverWrite : Boolean = false ) ;
```

As you can see, there are subtle but annoying differences.

An IT department within a big company may be able to bind tightly to a suite of components and never face any problems. However, they may merge with another organisation, or take over a smaller company and be asked to integrate their information technology systems, which are based on different component vendors APIs.

A contractor or a software house with many clients will probably be forced to use a variety of different components to achieve the same result because of having to work within the different standards required by the varying clients. This is one place where the Adaptor Pattern has really worked for me.

I can use exactly the same persistence framework with my customers who use DOA as for those who use IBOjects or the BDE. I just change a single line in the project's DPR file as show below:

To configure as a BDE based application:

```
program AdaptorDemo;

uses
  // The abstract class which defines the interface
  tiQueryAbs,
  // Pull in the BDE flavour of the framework
  tiQueryBDE
  ;
```

To configure as an IBOjects based application:

```
program AdaptorDemo;

uses
  // The abstract class which defines the interface
  tiQueryAbs,
  // Pull in the IBOjects flavour of the framework
```

```
tiQueryIB  
;
```

To configure as a DOA based application:

```
program AdaptorDemo;  
  
uses  
  // The abstract class which defines the interface  
  tiQueryAbs,  
  // Pull in the Oracle flavour of the framework  
  tiQueryDOA  
  ;
```

How this Implementation of the Adaptor Evolved

The implementation of the Adaptor we shall look at here evolved in a project I worked on early in 1999. We were using Delphi to process information contained in a relational database, then writing out a text files which were used as the input to another process. When the bulk of development was completed and we were working towards deployment we realised that our text files were several Meg in size and were too big to transfer over the slow network link we had been allocated. The obvious solution was to compress the files before sending them and to decompress them when they arrived. We started looking around for a compression component and found several on the web costing around about \$100 US. We went to our project manager for approval to buy the component and where told that the budget for the project had been closed, and we could not purchase any more software. But, it's only \$100 and it will probably cost 20 times that amount to write something ourselves. Sorry! But? No!

After clearing our heads in the coffee shop in the basement of the building, we realised that Delphi comes with the compression library, ZLib on the install CD. We agreed to use ZLib with the aim of replacing it with a Zip routine when the project manager's purse strings were a little looser.

After messing around with ZLib for a while, we grew sick of using its clumsy class based, and buffer based interfaces. We wanted to be able to do something simple like

```
lOutputString := CompressString( pInputString );
```

rather than having to create a TCompressionStream or use pointers and buffers as would otherwise be required. (We look at this more closely in the section on ZLib that follows)

Wrapping looked like the way forward. We would write a method called CompressString, and hide the need to create and free a TCompressionStream under the hood. Once this was done, it would be a simple matter to extend the interface of our new ZLib compression class to compress and decompress files, streams and buffers as well as strings.

The steps we took were to create a compression class that used ZLib behind an interface designed the way we wanted it. Once this was working, we re-factored to give a pure abstract class that declares the interface we want to use, but contains no implementation. Our ZLib compression class then descends from the pure abstract and adds the functionality required. Next, we created our compression objects via a class reference then finally wrote a Factory to take care of the object creation. We also took the opportunity to create a compression class that actually did no compression. This was useful in debugging if we suspected the compression algorithm was introducing bugs.

What the GoF Book Says About The Adaptor

Page 139 of GoF tells us that the intent of the Adaptor is:

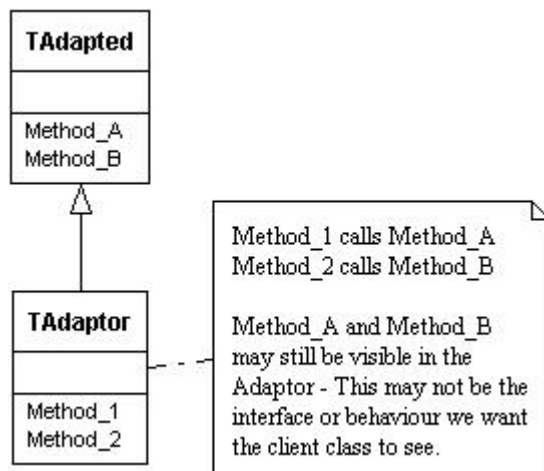
'Convert the interface of a class into another interface clients expect. Adaptor lets classes work together that couldn't otherwise because of incompatible interfaces'.

They go on to give one of their usual bad example that features a graphical toolkit. (After all, not many of us build WYSIWYG editors, so how about an example that a business programmer can relate to guys?) The discussion of the graphical tool kit goes on for three pages. The persistent reader is finally rewarded as they get to page 142 where the consequences of The Adaptor are discussed in great and enlightening detail.

GoF introduce us to two ways of implementing the Adaptor Pattern. They call these Class Adaptors and Object Adaptors:

Class Adaptor

Class Adaptors use inheritance (or multiple inheritance, which we can fake with Delphi's interfaces) as shown below:



To implement GoF's Class Adaptor, we inherited from the class to be adapted and add the new methods and properties as required.

While this is an adequate solution, it is not the one I usually use. If the class to be adapted has a complex interface and we only want to adapt a few of the methods, this will do the trick. If we have several classes to adapt, and want the to inherit from the same parent class (so they can be created with a Factory), we must use an Object Adaptor. Some more advantages and disadvantages of Class Adaptors are listed below.

Advantages:

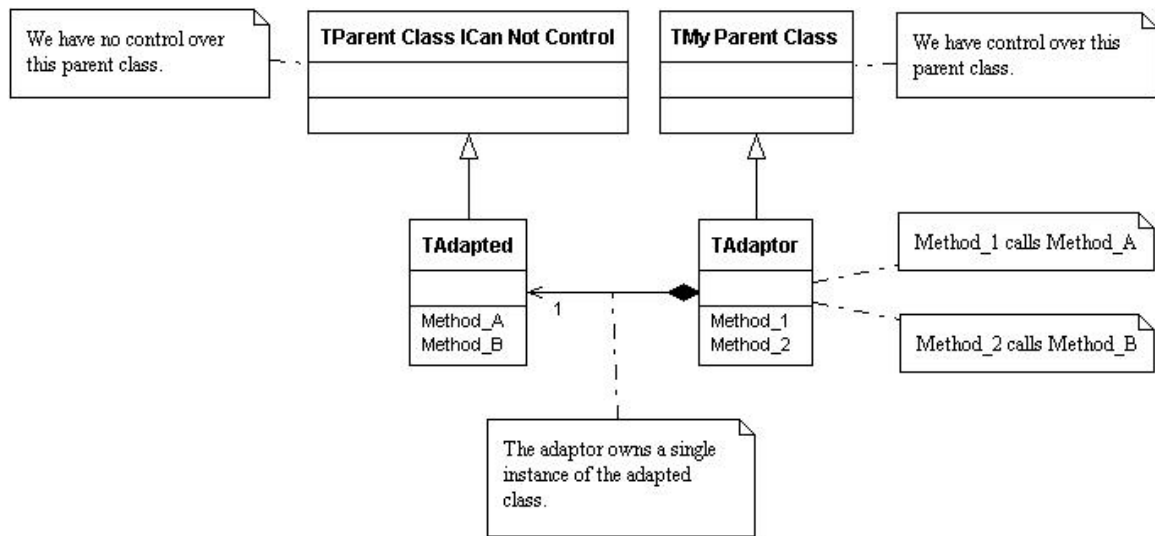
- Adaptation by inheritance means there is less code to write;
- inheritance is good when we only want to adapt part of the interface;
- inheritance means fewer classes created at run-time.

Disadvantages:

- If we have two unrelated classes that provide the same functionality, we will not be able to use a class reference or Factory to create an instance of the type we want at run-time;
- We may confuse the developers using our Adaptor, as the old interface of the adapted class will be visible as well as the newly created adapted interface.

Object Adaptors

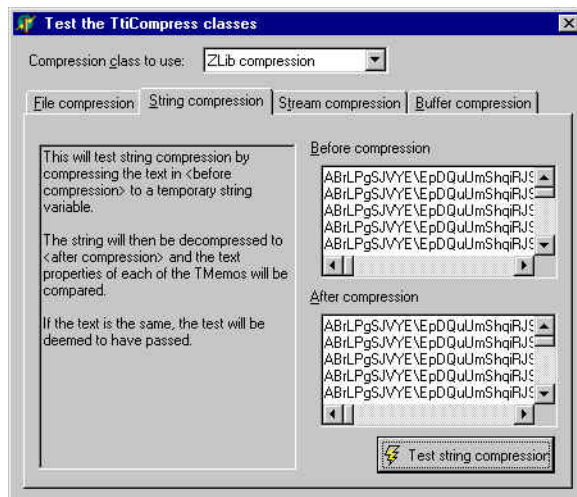
Object Adaptors require more work to implement; however the benefits are such that I think it is worth the effort. The UML of the Object Adaptor is shown below:



The key issue is that we want to have full control over the adapted class. We want to be able to specify its interface as well as its parent class so we can create instances with a Factory. This point is key to the rest of the chapter.

Adapting ZLib

In this example, we shall construct a wrapper for ZLib that gives us control over the interface. We shall implement the Null Object Pattern by creating a compression class that does not actually do anything. Finally, we shall create these compression objects using a Factory so the behaviour of the application can be changed at runtime. The main form of this test application is show below:



The ZLib library can be found on Delphi 5's CD in \Info\Extras\ZLib and comprises two Pas files along with a directory full of C source and header files, as well as some Obj files which can be linked into your Delphi application. ZLib.dcu and ZLibConst.dcu are also installed in the \Borland\Delphi5\Lib directory.

A quick look at ZLib.pas reveals two interfaces into the compression routines: An object-based interface and a single call function.

The object-based interface is shown below:

```

// Compression
TCompressionStream = class(TCustomZlibStream)
private
    function GetCompressionRate: Single;
public
    constructor Create(CompressionLevel: TCompressionLevel; Dest: TStream);
    destructor Destroy; override;
    function Read(var Buffer; Count: Longint): Longint; override;
    function Write(const Buffer; Count: Longint): Longint; override;
    function Seek(Offset: Longint; Origin: Word): Longint; override;
    property CompressionRate: Single read GetCompressionRate;
    property OnProgress;
end;

// Decompression
TDecompressionStream = class(TCustomZlibStream)
public
    constructor Create(Source: TStream);
    destructor Destroy; override;
    function Read(var Buffer; Count: Longint): Longint; override;
    function Write(const Buffer; Count: Longint): Longint; override;
    function Seek(Offset: Longint; Origin: Word): Longint; override;
    property OnProgress;
end;

```

The procedural interface is shown next:

```

procedure CompressBuf( const InBuf: Pointer;
                      InBytes: Integer;
                      out OutBuf: Pointer;
                      out OutBytes: Integer);

procedure DecompressBuf( const InBuf: Pointer;
                        InBytes: Integer;
                        OutEstimate: Integer;
                        out OutBuf: Pointer;
                        out OutBytes: Integer);

```

We decided that the procedural interface would be easier to implement for our purposes, we just had to create a wrapper that would give us the option of choosing between buffer, stream, string and file-based compression and decompression.

An interface along the lines of the one shown below and was designed to give us maximum flexibility.

```

TtiCompressAbs = class( TObject )
protected
    // Stream compression and decompression
    function CompressStream( pFrom : TStream ;
                            pTo : TStream ) : real ;
    virtual ; abstract ;
    procedure DecompressStream( pFrom : TStream ;
                                pTo : TStream ) ;
    virtual ; abstract ;

    // Buffer compression and decompression
    function CompressBuffer( const pFrom: Pointer ;
                             const piFromSize : Integer;
                             out pTo: Pointer ;
                             out piToSize : Integer) : real ;
    virtual ; abstract ;
    procedure DecompressBuffer( const pFrom: Pointer ;
                                const piFromSize : Integer;
                                out pTo: Pointer ;
                                out piToSize : Integer) ;
    virtual ; abstract ;

    // String compression and decompression
    function CompressString( const pFrom : string ;

```

```

        var pTo : string ) : real ;
    virtual ; abstract ;
procedure DecompressString( const From : string ;
    var pTo : string ) ;
    virtual ; abstract ;

// File compression and decompression
function CompressFile(    const pFrom : string ;
    const pTo : string ) : real ;
    virtual ; abstract ;
procedure DecompressFile( const pFrom : string ;
    const pTo : string ) ;
    virtual ; abstract ;

end ;

```

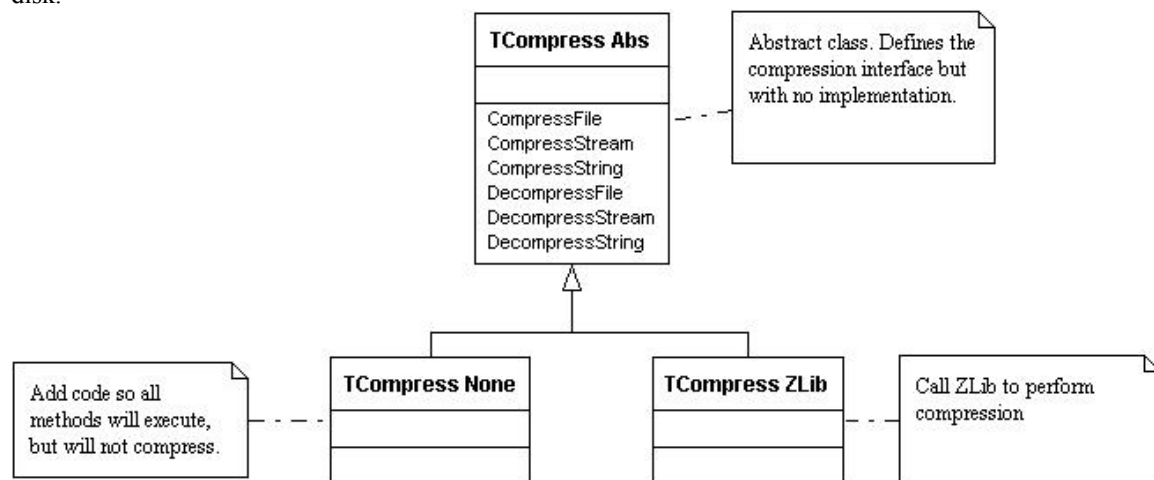
Two concrete classes were coded:

- TtiCompressZLib which implements the ZLib functionality; and
- TtiCompressNone which gives no compression.

The idea of having a compression class, which does not actually compress anything, came from studying the Null Object Pattern at a Melbourne Pattern Group meeting in August 2000. The intent of The Null Object Patter is *‘Provide a surrogate for another object that shares the same interface but does nothing. The Null Object encapsulates the implementation decisions of how to “do nothing” and hides those details from its collaborators.’* The text of the (I think brilliant) Null Object Pattern can be found at <http://citeseer.nj.nec.com/woolf96null.html>

The full source code of each of these classes can be found on the TechInsite web site.

The class hierarchy that has been implemented in the demo application is shown in below and can be found in the units tiCompressAbs.pas, tiCompressNone.pas and tiCompressZLib.pas on the companion disk.



Creating a Concrete Instance of the Adaptor

Now that we have coded our class hierarchy, we want to create an instance of the concrete class for use. The first method we can use is to specify the appropriate concrete class directly in code. This will lock us into using this class from design time, through compile to run-time. We may do something like:

```

Var
  lCompress : TtiCompressAbs ;
Begin
  lCompress := TtiCompressZLib.Create ;
  lCompress.CompressString( pStringIn, pStringOut ) ;

```

Creating From a Class Reference

If we want to make it easier to vary the way our application behaves, we would be better off using a class reference to create our concrete instances. If you have not used a class reference before, you can learn more about the in the Delphi Help under 'Class Reference'.

Taking our abstract compression class, a class reference declaration would look like this:

```
Type
  TtiCompressAbs = Class( TObject )
  // more...

  TtiCompressClass = Class of TtiCompress ;
```

(This line of code can be found in tiCompress.pas on the companion disk)

Using a class reference lets us write code like this:

```
Var
  lCompresClass : TtiCompressClass ;
  lCompress      : TtiCompressAbs ;
Begin
  // Notice this is a reference to a class type,
  // not an instance of the class.
  lCompressClass := TtiCompressZLib ;
  try
    lCompress := lCompressClass.Create ;
  // Use the instance...
```

This code is not very useful, but it does illustrate the concept. If we have our three classes declared in their own units like this:

TtiCompressAbs – in tiCompressAbs.pas;

TtiCompressZLib – in tiCompressZLib.pas; and

TtiCompressNone – in tiCompressNone.pas

then we start to see some useful functionality.

If tiCompressAbs.pas has a globally visible variable of type TtiCompressClass, then we can set this variable in the initialization section of both tiCompressZLib.pas and tiCompressNone.pas. If we always create the concrete instances of the compression object through the class reference, we can control the behaviour of our application by linking in either tiCompressZLib.pas or tiCompressNone.pas.

TiCompressAbs.pas will contain the following in its interface section:

```
// A globally visable variable to hold an instance of the
// compression class we will be using in the application.
var
  gTtiCompressClass : TtiCompressClass ;

function CreateDefaultCompress : TtiCompressAbs ;

implementation

// A helper function to create a compression object, with some
// checking before they are created
function CreateDefaultCompress : TtiCompressAbs ;
begin
  Assert( gTtiCompressClass <> nil, 'gTtiCompressClass not assigned' ) ;
  Result := gTtiCompressClass.Create ;
End ;
```

In `tiCompressZLib.pas` and `tiCompressNone.pas`, we will have the following lines of code in the initialization sections:

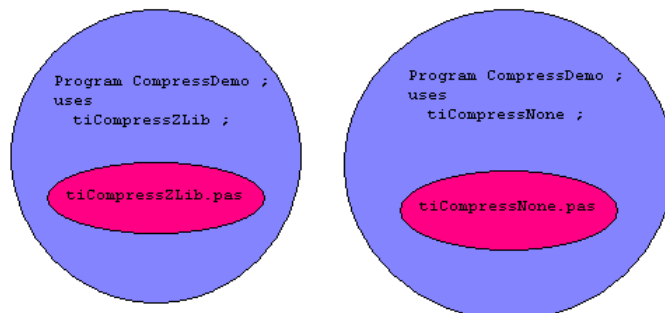
```
// tiCompressZLib.pas  
initialization  
  gtiCompressClass := TtiCompressZLib ;
```

```
// tiCompressNone.pas  
initialization  
  gtiCompressClass := TtiCompressNone ;
```

The behaviour of the application can be changed by changing a single line of code in the project's DPR file like this.

```
Program CompressDemo ;  
Uses  
  TiCompressZLib, // This will give the application ZLib behaviour  
  // TiCompressNone // This will give the application no compression behaviour  
 ;
```

This is illustrated in below:



In summary, this technique is useful for easily changing the behaviour of an application at compile time.

Creating from a Factory

An article on The Factory Pattern was published in September 1999 in edition 49 of The Delphi Magazine. To recap, the Factory can be implemented in Delphi as a TList of objects that map a string that identifies a class to a class reference that can be used to create an instance of the class.

To allow the behaviour of the application to be changed at runtime, we must create a class that maps a string identifier to a class reference. We then build a list of these mappings when the application initialises. To create a concrete instance of a class, we search the list for the appropriate mapping, and then use its class reference to return an instance.

We shall look at the two classes we use to achieve this: the class mapping, and the Factory. The interface and implementation of the class mapping (TtiCompressClassMapping) is shown below:

```
Interface  
  
  // A class to hold the TtiCompress class mappings. The Factory maintains  
  // a list of these and uses the CompressClass property to create the objects.
```

```

// -----
TtiCompressClassMapping = class( TObject )
private
  FsMappingName : string;
  FCompressClass : TtiCompressClass;
public
  Constructor Create( const pMappingName : string ;
                    pCompressClass : TtiCompressClass ) ;
  property MappingName : string read FsMappingName ;
  property CompressClass : TtiCompressClass read FCompressClass ;
end ;

implementation

// Overloaded constructor - used to create an instance
// of TtiCompressClassMapping and to preset it's properties.
// -----
constructor TtiCompressClassMapping.Create( const pMappingName: string;
pCompressClass: TtiCompressClass);
begin
  inherited Create ;
  FsMappingName := pMappingName ;
  FCompressClass := pCompressClass ;
end;

```

TtiCompressClassMapping comprises two properties, MappingName of type string and CompressClass of type TtiCompressClass. There is an overloaded constructor that lets us create an instance of TtiCompressClassMapping and preset its properties in a single call.

The second class we shall use is called TtiCompressFactory and is basically just a wrapper around a TObjectList with a method to register a class mapping. There is also a function that we can call to create a concrete instance of a TtiCompressAbs. This code is shown below:

```

Interface

// Factory Pattern - Create a descendant of the TtiCompress at runtime.
// -----
TtiCompressFactory = class( TObject )
private
  FList : TObjectList ;
public
  constructor Create ;
  destructor Destroy ; override ;
  procedure RegisterClass( const pCompressionType : string ;
                          pCompressClass : TtiCompressClass ) ;
  function CreateInstance( const pCompressionType : string )
                        : TtiCompressAbs ; overload ;
end ;

implementation

// -----
constructor TtiCompressFactory.Create;
begin
  inherited ;
  FList := TObjectList.Create ;
end;

// -----
destructor TtiCompressFactory.Destroy;
begin
  FList.Free ;
  inherited;
end;

// Register a TtiCompress class for creation by the Factory
// -----
procedure TtiCompressFactory.RegisterClass(
const pCompressionType: string; pCompressClass: TtiCompressClass);
var
  i : integer ;
begin
  for i := 0 to FList.Count - 1 do

```

```

// SameText is an undocumented function in SysUtils.pas. There is a note
// accompanying the source code which says:
// SameText compares S1 to S2, without case-sensitivity. Returns true if
// S1 and S2 are the equal, that is, if CompareText would return 0. SameText
// has the same 8-bit limitations as CompareText }
if SameText( TtiCompressClassMapping( FList.Items[i] ).MappingName,
            pCompressionType ) then
    raise exception.CreateFmt( 'Compression class <%s> already registered.',
                              [pCompressionType] ) ;
FList.Add( TtiCompressClassMapping.Create(
                                                pCompressionType, pCompressClass ) ) ;
end;

// Call the Factory to create an instance of TtiCompress
// -----
function TtiCompressFactory.CreateInstance( const pCompressionType: string)
                                          : TtiCompressAbs;
var
    i : integer ;
begin
    result := nil ;
    for i := 0 to FList.Count - 1 do
        if SameText( TtiCompressClassMapping( FList.Items[i] ).MappingName,
                    pCompressionType ) then
            begin
                result := TtiCompressClassMapping( FList.Items[i] ).CompressClass.Create ;
                Break ; //==>
            end ;
    raise exception.CreateFmt(
        '<%s> does not identify a registered compression class.',
        [pCompressionType] ) ) ;
end;

```

The key methods are RegisterClass and CreateInstance. Register Class takes two parameters: a string to identify the compression type, and a class reference that can be used to create the compression object. RegisterClass is called in the initialisation section of both tiCompressZLib.pas and tiCompressNone.Pas and typically look like this:

```

// In tiCompressZLib.pas
initialization
    // Register the TtiCompressZLib class with the Factory
    gCompressFactory.RegisterClass( 'Zlib Compression', TtiCompressZLib ) ;

```

```

// In tiCompressNone.pas
initialization
    // Register the TtiCompressNone with the CompressFactory
    gCompressFactory.RegisterClass( 'No Compression', TtiCompressNone ) ;

```

The code inside RegisterClass first scans the list of TtiCompressClassMapping(s) looking for an already created instance, then raises an exception if one was found (this means a programmer was trying to register a mapping under the same name more than once.) Next, an instance of TtiCompressClassMapping is created with its pre-assigned properties and is added to the list.

The code inside CreateInstance performs the same search for a registered class, then when found, calls Create against the mapped class reference and returns a concrete instance of the appropriate compression class.

We only want one instance of the compression to exist in memory at any time, so we implement it as the Singleton Pattern. I do this using a variable with unit wide visibility, which is hidden behind a globally visible function. This is not a true singleton as it is possible to create more than one instance of TtiCompressFactory, and it is also possible to free the Factory before the application terminates. (Some more 'pure' implementations of the Singleton Pattern are discussed in issues 41 and 44 of The Delphi Magazine). My implementation of the compression Factory as a singleton is shown below:


```

Interface

// The CompressFactory is a singleton which is implemented as a variable with
// unit wide visibility hidden behind a globally visible function.
// -----
function gCompressFactory : TtiCompressFactory ;

implementation
var
  // A var to hold our single instance of the TtiCompressFactory
  uCompressFactory : TtiCompressFactory ;

// -----
function gCompressFactory : TtiCompressFactory ;
begin
  if uCompressFactory = nil then
    uCompressFactory := TtiCompressFactory.Create ;
    result := uCompressFactory ;
  end ;

initialization
  // Do not bother creating an instance of TtiCompressFactory here, it will
  // be created on demand when it is first used.

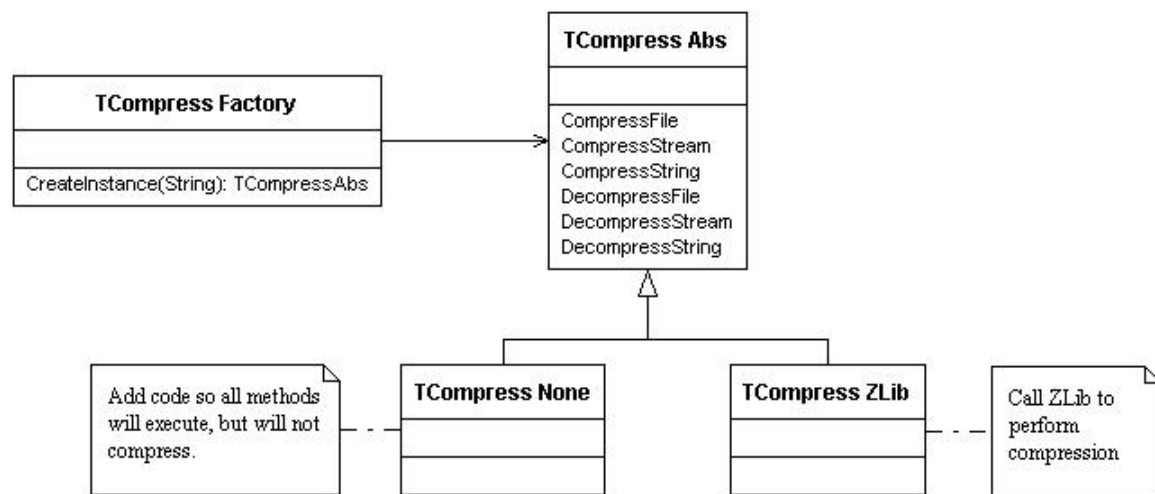
finalization
  // Free the TtiCompressFactory in the finalization section
  uCompressFactory.Free ;
    
```

When we want to create and use a compression object, we call the Factory like this:

```

Var
  lCompress : TtiCompressAbs ;
begin
  // We can dynamically change the type of compression
  // by passing a different string here
  lCompress := gCompressFactory.CreateInstance( 'ZLib Compression ) ;
  try
    // use the encryption object
    finally
      lCompress.Free ;
    end ;
  end ;
    
```

In summary, we have extended our compression class hierarchy with a Factory as shown in the UML below:



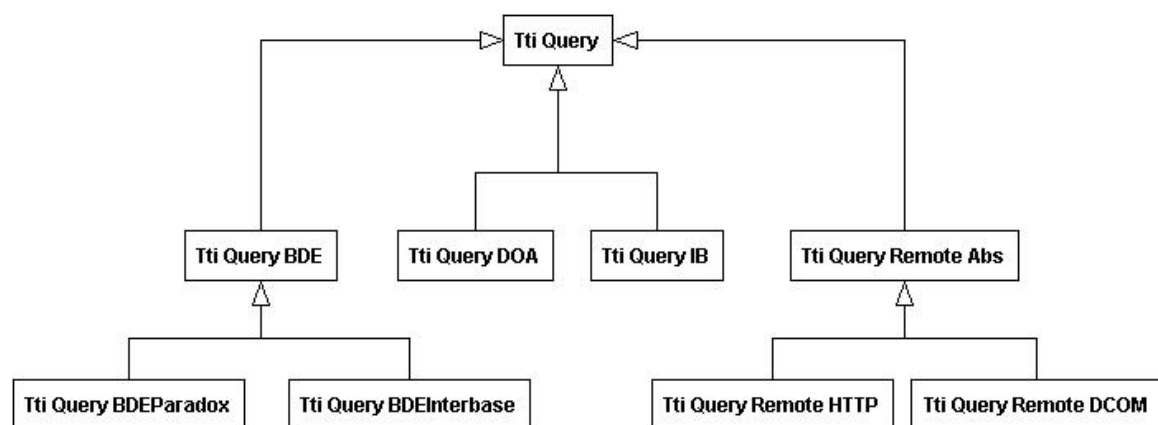
To recap, we have build:

- An abstract compression class that defines the interface, but has no implementation. This is contained in the unit tiCompress.pas.
- A ZLib compression class, which is contained in the unit tiCompressZLib.
- A No compression class, which is an implementation of the Null Object Pattern which has the same interface as ZLib compression but performs no compression. This is contained in the unit tiCompressNone.pas.
- A Factory to control which concrete compression class to create at runtime. This is contained in listing tiCompress.pas.
- We also create a demonstration application with a main form that will let us test the various methods on the different compression classes.

Adapting Data Access Components

Adapting a compression or encryption class is useful, but these algorithms are seldom core to the functionality of an application. Data access components are core to most business applications, but their interfaces are usually complex and have many dependencies. Most data access components are descendants of TDataSet and can be interacted with in code, or wired up to data aware controls.

If you're not bothered about losing the ability to connect to a data aware control via a TDataSource, there are some benefits in adapting the interface of Delphi's data access components. My class hierarchy of data access components is shown in the UML below:



This shows the adaptors used in the TechInsite persistence framework. The starting point is the virtual abstract class TtiQuery which implements navigation and field access methods, much the same as the TDataSet does. The TtiQueryBDE implements BDE style connectivity via a TQuery and has been tailored for both Interbase and Paradox connectivity in the TtiQueryBDEInterbase and TtiQueryBDEParadox. TtiQueryDOA gives Oracle access via the DOA ('Direct Oracle Access') components, while the TtiQueryIB gives Interbase connectivity via IBOjects.

By now, you are probably thinking that this is all a waste of effort as this functionality is all wrapped up in the TDataSet ancestor. Well, a TDataSet brings along considerable fat and for optimised Oracle access, it's best to use a component that is not a TDataSet descendent.

The TtiQueryRemoteHTTP and TtiQueryRemoteDCOM give similar functionality to using the TClientDataSet with MIDAS without the need to deploy MIDAS and pay MIDAS licences. This architecture has been use to build an application that can connect directly to Oracle from behind the companies firewall. With the flick of a command line switch, the same application can communicate with the database via a web server over port 80 using HTTP.

This framework makes it possible to write code like:

```
Var
  lQueryIB : TtiQueryIB ;
  lQueryXML : TtiQueryLocalXML ;
  lCustomer : TCustomer ;
begin

  lCustomer := TCustomer.Create ;
  lCustomer.OID := 100 ;
  lQueryIB := TtiQueryIB.Create ;
  lQueryIB.Read( lCustomer ) ;
  lQueryXML := TtiQueryLocalXML.Create ;
  lQueryXML.Save( lCustomer ) ;
```

This means the persistence layer can be swapped at run-time that is very difficult to do with the component on form style of developing. The full source of the TechInsite persistence framework, along with a demonstration of this technique is available for free from <http://www.techinsite.com.au/>

Summary

In this chapter we have studied the Adaptor Pattern in some detail, and have also revisited the Factory Pattern and glimpsed at the Null Object Pattern. We have seen how the Adaptor, Factory and Null Object Patterns can be used together to delay the implementation of a complex algorithm, or to allow one algorithm to be replaced with another at compile time or run time.

We have seen how to wrapper the ZLib compression routines to give a more convenient interface, and explored the idea of wrapping data access components to reduce our dependency on a specific vendor's data access API.

Good luck implementing the Adaptor Pattern and please let me know of your experiences on the EMail address below.

Chapter #8

Using Report Builder with the tiOPF

This chapter was provided by Andrew Denton adenton@q-range.com after a discussion on the mailing list about reporting techniques and the tiOPF.

The following code is taken from a base form in one of my applications that utilises printing a tiListViewPlus (a normal tiListView should work fine also) using Report Builder. Create a form with the following components: -

A tiListViewPlus (named lvTrans)
A TppReport (named rptBase)
A TppJITPipeline (named plData)

Set the ReportUnits property of the report to be utMillimeters – this is important for the field placement later on. You will also need to attach your tiListView to a TPerObjList descendant. I'm assuming that everyone knows how to do this! Next, create two Private methods in your form declaration as follows:

-

```
Function plCheckEOF : Boolean;  
Function plGetFieldValue(aFieldName : String) : Variant;
```

These are the required event handlers for the pipeline that we need to tell it how to provide the data. Hit CTRL+SHIFT+C for code completion - we will come to these in detail shortly, but we'll leave them blank for now. We also need some kind of UI component to enable the user to request a report, I generally use an action list, but to keep things simple we'll settle for dropping a button on the form. Name the button btnPrint and double-click it to bring up its OnClick event handler. Now comes the coding part! Make your event handler look like this: -

```
Procedure TfrmTransBase.btnPrintClick(Sender : TObject);  
Const  
    FieldSpacing = 10;  
Var  
    I, OldPos : Integer;  
    MyType : TppDataType;  
    CurXPos : Integer;
```

```

Function GetReportFieldSize(FieldType : TppDataType) : Integer;
Begin
  Result := 5;
  Case FieldType Of
    dtDateTime : Result := 100;
    dtInteger : Result := 50;
    dtString : Result := 120;
    dtCurrency : Result := 60;
  End; { Case }
End;

Procedure AddReportField(Const pFieldName : String; FieldNo : Integer; FieldType :
TppDataType);
Var
  lblField1 : TppDBText;
Begin
  lblField1 := TppDBText.Create(Self);
  lblField1.Band := rptBase.DetailBand;
  lblField1.spLeft := CurXPos;
  lblField1.spTop := 2;
  lblField1.DataPipeline := plData;
  lblField1.AutoSize := False;
  lblField1.DataField := pFieldName;
  lblField1.spWidth := GetReportFieldSize(FieldType);
  If FieldType In [dtCurrency, dtInteger] Then
    lblField1.Alignment := taRightJustify;
  CurXPos := CurXPos + lblField1.spWidth + FieldSpacing;
End;

Procedure AddColumnHeading(Const pHeading : String; FieldNo : Integer; FieldType :
TppDataType);
Var
  lblHeading1 : TppLabel;
Begin
  lblHeading1 := TppLabel.Create(Self);
  lblHeading1.Band := rptBase.HeaderBand;
  lblHeading1.spLeft := CurXPos;
  lblHeading1.spTop := 5;
  lblHeading1.AutoSize := False;
  lblHeading1.Caption := pHeading;
  lblHeading1.Font.Style := [fsBold];
  lblHeading1.spWidth := GetReportFieldSize(FieldType);
  If FieldType In [dtCurrency, dtInteger] Then
    lblHeading1.Alignment := taRightJustify;
End;

```

Begin

```

// Let's create a report.
// First we need to interrogate the list view for it's columns so we can create the
// fields and add the fields to the report
CurXPos := 2;
OldPos := lvTrans.SelectedIndex; // Remember where we were in the list.
For I := 0 To lvTrans.ListColumns.Count - 1 Do
Begin
  With lvTrans.ListColumns.Items[I] Do
    Begin
      MyType := dtString;
      Case DataType Of
        lvtkString : MyType := dtString;
        lvtkFloat : MyType := dtCurrency;
        lvtkDateTime : MyType := dtDateTime;
        lvtkInt : MyType := dtInteger;
      End; { Case }
      AddColumnHeading(DisplayLabel, I, MyType);
      plData.DefineField(FieldName, MyType, 25); // 25 is a temporary hack value for
now.
      AddReportField(FieldName, I, MyType);
    End;
  End; { Loop }
  // Now we need to hookup the event handlers to the JIT Pipeline to provide access to
  // the // underlying data.

  plData.OnCheckEOF := plCheckEOF;
  plData.OnGetFieldValue := plGetFieldValue;
  // Set the report's title.
  lblReportTitle.Caption := 'Q-Bar Quick Report - ' + Caption;
  // Prevent annoying listview scrolling when preparing report.

```

```
LockWindowUpdate(Self.Handle);
                                // Now all that's done we print the report.
Try
    rptBase.PrintReport;
Finally
    LockWindowUpdate(0); // Now we're good to go.
End;
If (OldPos <> -1) Then
    lvTrans.PositionCursor(OldPos)
Else
    lvTrans.PositionCursor(0); // Reset the list view's highlight bar.
End;
```

Now we've coded the main printing routine we need to write the code for the event handlers, which looks something like this: -

```
Function TfrmTransBase.plCheckEOF : Boolean;
Begin
    Result := (plData.RecordIndex >= lvTrans.Items.Count);
End;

Function TfrmTransBase.plGetFieldValue(aFieldName : String) : Variant;
Var
    lListColumn : TtiListColumn;
    lColID : Integer;
Begin
    // Need to obtain the value of the property aFieldName from the current list item.
    // Using PositionCursor isn't ideal but AFAIK it's the only way of accessing the
    // underlying Object.
    lvTrans.PositionCursor(plData.RecordIndex);
    If (lvTrans.SelectedData <> Nil) Then
        Begin
            lListColumn := lvTrans.ListColumns.FindByFieldName(aFieldName);
            lColID := lListColumn.ID;
            If lListColumn.Derived Then
                Result := lvTrans.Selected.SubItems.Strings[lColID]
            Else
                Result := GetPropValue(lvTrans.SelectedData, aFieldName) // The magic of RTTI !!!
        End;
    :)
End
Else
    Result := 'Error';
End;
```

Compile and run your test project and you should now be able to print a report based on the view presented by a tiListView or tiListViewPlus. Couple this with Form Inheritance and you have all your basic reporting functionality already written. Of course, the routines described here could be improved on - most noticeable is the lack of any kind of totalling on the final report, but it should give you an insight into what can be achieved with these two packages.

Chapter #9

References and further reading

Download the source

The latest copy of this paper, along with the source code of the demonstration applications is available from www.techinsite.com.au/tiOPF/Documentation/ The full source code of the TechInsite persistence framework is available from www.techinsite.com.au/tiOPF/Download.htm.

There is a mailing list for users of the tiOPF to discuss issues and share ideas at www.techinsite.com.au/tiOPF/maillinglist.htm.

Scott Ambler

Two of Scott Ambler's papers have been used as references in the design of the tiOPF:

'Mapping Objects', which can be found at <http://www.ambysoft.com/mappingObjects.pdf> was used to design the high/low OID generation strategy which is used throughout the framework.

The design of a robust persistence layer for relational databases' which can be found in full at <http://www.ambysoft.com/persistenceLayer.pdf>. This papers was used as the starting point for the automatic object to database mapping framework that was touched on in chapter 7.

The Gang of Four

In any substantial work involving object oriented design, it's impossible to escape the influence of 'The Gang of Four'. Design Patterns: Elements of Reusable Object-Oriented Software. Gamma, Helm, Johnson, Vlissides. Addison Wesley, 1995.

The Jedi-Obiwan project

The Jedi-Obiwan project is an open source project to build an object persistence framework in Delphi. The specification, along with other design and discussion documents and a mailing list can be joined at <mailto:jedi-obiwan-subscribe@yahoogroups.com>

Australian Delphi User Group (ADUG)

As a contract programmer without the support of an employer I depend heavily on our local user group for technical assistance, companionship and mentoring. My local user group is ADUG (www.adug.org.au) and has introduced me to the most generous group of professional programmers you could ever hope to meet.

Melbourne Pattern Group

A friend and fellow student of Patterns, Don Macrae, reckons that you have to ‘Get’ a Pattern before you can utilise it fully. ‘Getting’ a Pattern involves reading about it, talking about then implementing it a couple of times in several different ways. After a while, it becomes a comfortable tool in your programmers tool box and you are able to use it with little or no thought – just the way we all type try-finally-end around a resource we have created.

One of the best ways to ‘Get’ a Pattern is to join a Pattern discussion group and an index of these groups can be found at <http://hillside.net/patterns/Groups.html>. The Pattern group that helped me ‘Get’ the patterns that are central to the framework is in Melbourne, Australia and the web site can be found at <http://www.melbournepatterns.org>

ⁱ DOA – Direct Oracle Access by All Round Automations. <http://www.allroundautomations.nl>

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