Persistence Frameworks: introducing tiOPF

Michaël Van Canneyt

February 8, 2007

Abstract

tiOPF is a persistence framework for Object Pascal maintained at the TechInsite in Melbourne, Australia. It works with Delphi and Free Pascal, and is full featured, albeit not very RAD. In this article the framework and some of it's core concepts are introduced.

1 Introduction

tiOPF has been around for quite a while. The feature list is impressive, and it sports many tools for easing the coding of an OOP application, including utilities for a variety of other tasks. The web-site of tiOPF is at

`http://www.techinsite.com.au/`

The site contains a lot of information about the framework. Some of the material is a bit dated (the framework has obviously evolved), but it is very instructive, and helps to get a good understanding of the working of the framework. It's mostly written by the original author of the framework, Peter Hinrichsen.

tiOPF offers various database persistence mechanisms: Interbase/Firebird through IBX and FBLib, Oracle (via DOA), Paradox using the BDE, MS-Access and MS-SQL Server through ADO. There is also custom support for storage in CSV and TAB-delimited flat files, as well as XML storage through MSDOM and a custom persistence layer. The XML layers can also be made remotely accessible through the HTTP protocol.

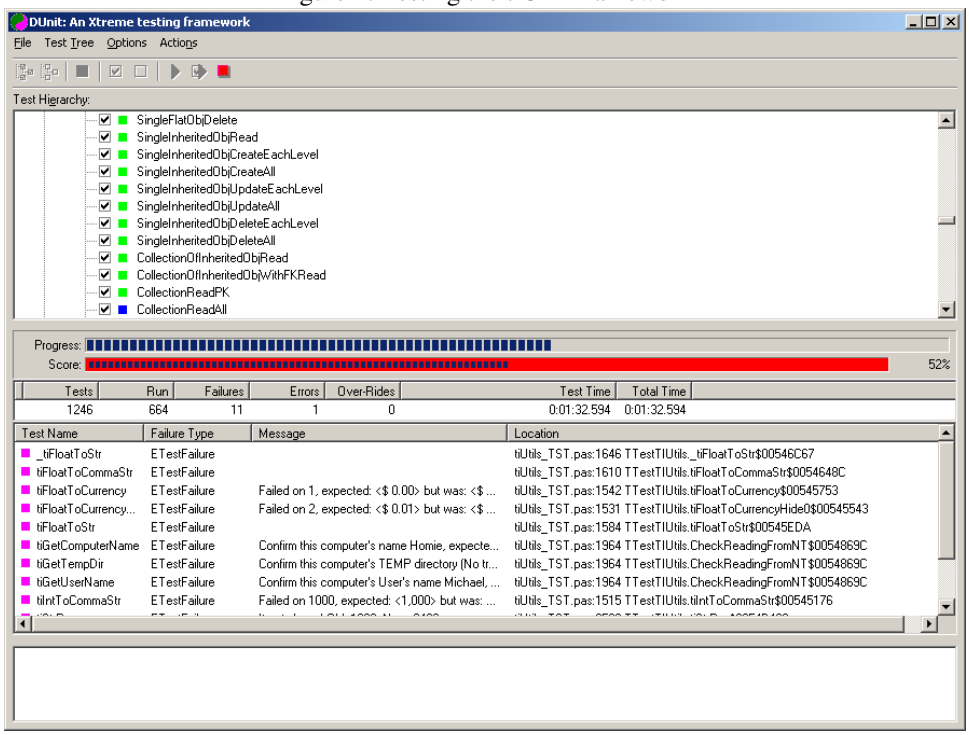
Unlike InstantObjects, tiOPF does not work through a `TDataset` to represent the business objects. This means that the standard DB-Aware controls cannot be used to create a user interface. Instead, tiOPF offers a set of 'Persistence Aware' controls, which can be used to create a user interface. There are around 35 components, basically the ones one would expect.

As an alternative to the 'Persistence Aware' components, tiOPF offers a set of 'Mediating View' components, which are basically a set of components which connect standard VCL controls such as `TEdit` and manage the setting/retrieving of object properties. The implementation makes use of the observer pattern, commonly used in Java. This set of classes makes it possible to connect any control to an object, so the object can be manipulated by the user.

2 Installation

For Delphi, an installer is provided. It installs the tiOPF sources in a location of your choice. No packages are installed by default, the packages must be compiled and installed

Figure 1: Testing the tiOPF framework



manually. However, a project group with the packages to install is provided for several versions of Delphi, currently D7/D2005/D2006 (it should be compilable with D6 and possibly D5 as well). There is a comprehensive help document provided in the install zip, which clearly explains what to do and how to run some tests, which amounts basically to compile the following packages:

tiOPFCore The heart of the tiOPF framework.

tiOPFGUI The GUI components of the tiOPF framework, runtime version.

tiOPFGUIdsgn Design-time support for the GUI components. This package should be installed in the Delphi IDE.

tiOPFOptions The GUI components of the tiOPF framework,

The packages can be found in the Source/Compilers directory.

At the end of the installation, a test application can be compiled, and a testsuite can be run. This can be seen in figure 1 on page 2. The tests show 15 failures - some are based on the fact that the coders of the tests assumed that the decimal separator is '.', which is not the case in e.g. Belgium or Germany.

For Lazarus, installation instructions can be found on the Lazarus Wiki pages:

```
http://wiki.lazarus.freepascal.org/tiOPF
```

Mainly, this means downloading the latest sources from Subversion:

```
mkdir tiOPF2
cd tiOPF2
```

```
svn co https://svn.sourceforge.net/svnroot/tiopf/tiOPF2/Trunk Source
svn co https://svn.sourceforge.net/svnroot/tiopf/tiOPF2_Demos Demos
svn co https://svn.sourceforge.net/svnroot/tiopf/tiOPF2_Docs Docs
```

Inside the Source/Compilers/FPC directory there are some packages:

tiOPF Core units (a run-time only package)

tiOPFGUI GUI related units and components (a run-time only package).

tiOPFGUIDsgn Registers/Installs the components into the Lazarus component palette (design-time only package). The GUI components used under Lazarus are still experimental and under heavy development.

tiOPFHelpIntegration Integrates the help files into Lazarus's help system.

All units in tiOPF start with the prefix ti, which makes them easily recognizable. The classes in the framework use the same prefix.

3 Architecture

Using tiOPF is not for the faint-of-heart. Whereas InstantObjects takes a very RAD approach to object persistence and should be usable by any beginning programmer (even one who is not very schooled in OOP), because it tries to hide the gory OOP details from the programmer, tiOPF clearly pulls the card of the OOP programmer, using ideas from patterns and many other OOP techniques.

tiOPF currently contains no GUI modeler for the business classes. One could say that it doesn't need one (which would be very true) but it should not be hard to implement one. That it doesn't use one is also an advantage:

- almost any existing class can be made persistent.
- almost no assumptions are made on the database structure, making it possible to work with legacy data. There is a single requirement, namely that each object instance has a unique Object ID.

The downside of this is that there is less point-and-click (the RAD aspect), but there is more coding to be done. This need not be a disadvantage, but the treshold for starting with tiOPF is bigger.

At the heart of the persistence mechanism lies the Visitor pattern. The available documentation spends lots of time explaining this pattern - it will be explained below. (a "pattern" is a term coined by the so-called "Gang of Four" in their seminal book on OOP programming; Basically it's just an algorithm or way of doing things using some well designed classes).

On top of the visitor pattern, `TtiObject` is implemented. This is the base class for all business classes that must be made persistent. The `TtiObject` class is accompagnied by the `TtiObjectList` class which is a class that holds references to a series of objects, and can be used to describe one-to-many relationships in the business model.

The properties of a class that should be persisted by tiOPF need not be declared published. It is possible to have them not published, but then many of the built-in mechanisms are not available, and the persistence should be hard-coded.

Persisting an object takes some work: for each object, the persisting must be defined in code. Depending on the chosen storage mechanism, this is more or less work. For simple

lists, pre-defined mechanisms can be used. For more complex lists, the visitor classes must be coded manually.

When defining the persistence for an SQL database, there are 3 strategies:

1. Hardcode the SQL statements. This means that 4 classes must be implemented:
   - A class to load a (list of) objects.
   - A class to insert a new object.
   - A class to save a modified object.
   - A class to delete an object and one.

   each of these classes is a descendent of a special visitor, as will be seen below.
2. Use an SQL manager class: this defines and stores the SQL queries in the database. For this, some special tables must be present in the database, so this is only an option when the design of the database is under control of the programmer. In this case, this option makes it quite easy to set up persistence.
3. Set up mappings between objects and tables. The mapping between objects and table names must be set up, likewise, the mapping between properties and fieldnames must be set up. In this case, the OPF will auto-generate the queries. This is useful for simple models. It's also the way to use the persistence mechanism for flat-file databases (non-SQL, such as XML, CSV and TAB-delimited)

The first strategy is the most powerful one:

- It allows one to use legacy data, by adapting the SQL to the data at hand.
- The SQL can be completely customized, so that e.g. master-detail relationships can be loaded using 1 optimized query, and customized views are possible.
- Filtering is also possible. The pre-defined queries always returns the whole table or (for objects that are owned by some other objects) the list of owned objects.

It is also the one that requires the most code.

The second and third strategy lend themselves to writing a modeler such as it exists in InstantObjects. They have the additional advantage that the generated SQL is independent of the database, while the first is prone to tying the SQL to a particular database.

4 The visitor pattern

The "visitor pattern" is so ubiquitous in the tiOPF framework, that some extra attention to this technique is needed. It is used to do the persistence, and therefore, correct understanding of this pattern is essential to be able to use the persistence mechanism in tiOPF.

The basic idea of this is quite simple: it's an OOP way of encapsulating a for-loop on a list. Supposing we have a class `TPerson`, with a property `Name`. Furthermore, we have a list of TPerson instances, which we are manipulating in our program. Now we want to uppercase the names of all persons in the list. Traditionally, this would be done like this:

```
Procedure TForm1.Capitalize;

Var
  I : Integer;
```

```
begin
  For I:=0 to FList.Count-1 do
    TPerson(Flist[i]).Name:=UpperCase(TPerson(FList[i]).Name);
end;
```

The typecasts do not exactly contribute to the readability of this code. it becomes more readable when the `With` keyword is used:

```
Procedure TForm1.Capitalize;

Var
  I : Integer;

begin
  For I:=0 to FList.Count-1 do
    With TPerson(Flist[i]) do
      Name:=UpperCase(Name);
end;
```

Suppose that more than one operation is needed: Lowercasing the name, pretty-formatting it (first letter capitalized), adding it to the lines of a memo, and many others. Each operation would be coded much like the Capitalize method in the above code. Each would contain a loop. It's possible to have only 1 procedure that runs the loop, with a callback that is called for each element. This would be coded as follows:

```
TPersonOperation = Procedure (P : TPerson) of Object;

Procedure TForm1.ForeachPerson(Op : TPersonOperation);

Var
  I : Integer;

begin
  For I:=0 to FList.Count-1 do
    Op(TPerson(FList[i]));
end;

Procedure TForm1.CapitalizePerson(P: TPerson);

begin
  P.Name:=UpperCase(P.Name);
end;

Procedure TForm1.Capitalize;

begin
  ForeachPerson(CapizalizePerson);
end;
```

Adding the lowercase operation is then a matter of 2 methods:

```
Procedure TForm1.LowercasePerson(P: TPerson);
```

```
begin
  P.Name:=LowerCase(P.Name);
end;

Procedure TForm1.LowerCase;

begin
  ForeachPerson(LowercasePerson);
end;
```

And similar for each other operation. For the simple example presented here, the gain is obviously small. However it does introduce some flexibility: if for some reaston, the list would be changed by a collection or a linked list, no recoding of the procedure would be necessary: only the Foreach method would have to be recoded. This kind of code can be found in many places, also in the VCL of Delphi.

However, more things can be abstracted. The form still "knows" about the list, it does the looping. The list can be abstracted out:

```
TPersons = Class(TObjectList)
  Procedure Foreach(Op : TPersonOperation);
end;
```

Which would be implemented as:

```
Procedure TPersons.Foreach(Op : TPersonOperation);

begin
  For I:=0 to Count do
    Op(TPerson(Items[i]));
end;
```

And would be called from the form as:

```
Procedure TForm1.Capitalize;

begin
  FList.Foreach(CapitalizePerson);
end;
```

Still, this is not enough. The Capitalize method is stateless. That means that it is the same at each call. Consider the following operation:

```
Procedure TForm1.WritePerson(P : TPerson);

Var
  F : Text;

begin
  AssignFile(F,'persons.txt');
  Append(F);
  Writeln(F,P.Name);
  CloseFile(F);
end;
```

```
Procedure TForm1.WriteList;

begin
  FList.Foreach(WritePerson);
end;
```

Obviously, this is not very efficient. The following operation is better:

```
Var
  F : Text;

Procedure TForm1.WritePerson(P : TPerson);

begin
  Writeln(F,P.Name);
end;

Procedure TForm1.WriteList;

begin
  AssignFile(F,'persons.txt');
  Rewrite(F);
  FList.Foreach(WritePerson);
  CloseFile(F);
end;
```

This is much more efficient. But it introduces some flaws: The variable F is a global variable (it could be a field of TForm1), which is bad. Secondly, the opening and closing of the file needs to be done each time a loop is called. if we wanted to write the file as XML for instance, we'd have to do the same operation:

```
Var
  F : Text;

Procedure TForm1.WriteXMLPerson(P : TPerson);

begin
  Writeln(F,'<Person>',P.Name,'<person>');
end;

Procedure TForm1.WriteXML;

begin
  AssignFile(F,'persons.xml');
  Rewrite(F);
  Writeln(F,'<persons>');
  FList.Foreach(WriteXmlPerson);
  Writeln(F,'</persons>');
  CloseFile(F);
end;
```

The same F is used, making threading impossible. Furthermore, the opening and closing of the file is the same each time.

To improve this, the approach must be changed. The solution is not to pass a method to the `Foreach` call, but to pass an object which will do the actual operation. This object needs one known well-known, virtual method `Execute`:

```
TOperation = Class(TObject)
  Constructor Create; virtual;
  Procedure Execute(P : TPerson); virtual;
end;

TPersons = Class(TObjectList)
  Procedure Foreach(Op : TOperation);
end;
```

The `Foreach` method will now be implemented as:

```
Procedure TPersons.Foreach(Op : TOperation);

begin
  For I:=0 to Count do
    Operation.Execute(TPerson(Items[i]));
end;
```

For the Capitalization method, the form would contain code similar to the following:

```
Type
  TCapitalizer = Class(TOperation)
    Procedure Execute(P : TPerson); override;
  end;

Procedure TCapitalizer.Execute(P : TPerson); override;

begin
  P.Name:=Uppercase(P.Name);
end;

Procedure TForm1.Capitalize;

Var
  Cap : TCapitalizer;

begin
  Cap:=TCapitalizer.Create;
  Try
    Flist.Foreach(Cap);
  Finally
    FreeandNil(Cap);
  end;
end;
```

The writing to XML or plain text file would be implemented with a common ancestor which sets up the file. This ancestor could look as follows:

```
Type
TFileOp = Class(TOperation)
```

```
Private
  F : Text;
Public
  Constructor Create(AFileName : String); virtual;
  Destructor Destroy; override;
end;

Constructor TFileOp.Create(AFileName : String);

begin
  Inherited Create;
  Assign(F,AFileName);
  ReWrite(F);
end;

Destructor TFileOp.Destroy;
begin
  Close(F);
  Inherited;
end;
```

The `Execute` method is overridden in the descendents:

```
Type
TTextOp = Class(TFileOp)
  procedure Execute(P : TPerson); override;
end;

Procedure TTextOp.Execute(P : TPerson);

begin
  Writeln(F,P.Name);
end;
```

For XML, the some extra setup is needed:

```
Type
TXMLOp = Class(TFileOp)
  Constructor Create(AFileName : String); override;
  Destructor Destroy; override;
  procedure Execute(P : TPerson); override;
end;

Constructor TXMLOp.Create(AFileName : String);

begin
  inherited;
  Writeln(F,'<persons>');
end;

Destructor TXMLOp.Destroy;

begin
  Writeln(F,'</persons>');
```

```
  inherited;
end;

Procedure TXMLOp.Execute(P : TPerson); override;

begin
  Writeln(F,'<person>',P.Name,'</person>');
end;
```

The form can now be coded as:

```
Procedure TForm1.SavePersons(AsXML : Boolean);

Var
  F : TFileOp;

begin
  If AsXML then
    F:=TXMLOp.Create('persons.xml')
  else
    F:=TTextOp.Create('persons.txt');
  Try
    Flist.Foreach(F);
  Finally
    FreeAndNil(F);
  end;
end;
```

The above example shows the essence of the visitor pattern: creating an object, which executes an action on a list of objects or on a single object. This object is called the visitor.

Obviously, there is some room for more abstraction, as the above code is very specific to the example.

1. The operation should be not only on `TPerson` objects.
2. The operation should not be limited to a list, i.e. it should be possible to have the action executed on a single instance, without any need for a list. This, combined with the previous requirement, leads to the introduction of the `TVisited` class: This has a single method `Iterate`:

   ```
   Procedure Iterate(AVisitor : TVisitor);
   ```

   This is the equivalent of `Foreach`. In the base class, the method simply calls the execute method, passing itself as a parameter:

   ```
   Procedure TVisited.Iterate(AVisitor : TVisitor);
   begin
     Avisitor.Execute(Self);
   end;
   ```

   A list-like descendent of `TVisited` would do the following:

   ```
   Procedure TVisitedList.Iterate(AVisitor : TVisitor);
   ```

```
Var
  I : Integer;

begin
  Inherited Iterate;
  For I:=0 to Count-1 do
    AVisitor.Execute(GetElement(i));
end;
```

Note that it first invokes the inherited `Iterate`. All elements in the list should obviously be of type `TVisited`.

3. The visitor class should be able to decide whether or not it should execute itself on the passed instance. For this an `AcceptVisitor` is introduced:
4. In case of a list, the visitor should be able to signal to the list that the remainder of the list should no longer be visited.

These requirements lead to a definition of `TtiVisited` and `TTiVisitor` as they are found in the tiVisitor unit.

In the context of a Object Persistence framework, it should be obvious where this is leading to: the objects that do the persisting are implemented as visitors, much like the `TFileOp` presented here. The 'state' information is in that case the connection to the database. When a list of objects needs to write itself to the database (or file or whatever), it creates the appropriate persistence visitor, and lets itself be iterated by the visitor.

In case SQL will be written hard-coded, a descendent of `TtiVisitor` must be written.

5 The tiOPF base classes

The visitor pattern is the basis of all classes in tiOPF. figure 2 on page 12 shows the class diagram for the base classes in tiOPF. The `TtiVisitor` and `TtiVisited` classes are defined in unit tiVisitor. The base class for all objects that should be persisted, descends from `TTiVisitor` and is called `TtiObject`. The class that contains a list of other classes is called `TtiObjectList`. It's similar in appearance as TList, but maintains a list of `TTiObject` instances, and is used to represent both simple lists of objects as well as relationships between objects. `TTiVisitorManager` is a factory for `TtiVisitor` classes. Here, visitor classes must be registered under a certain name. The `TtiVisitorCtrl` class is an auxiliary class which can be used to group initialization and finalization of visitors (such as setting up or closing a database connection).

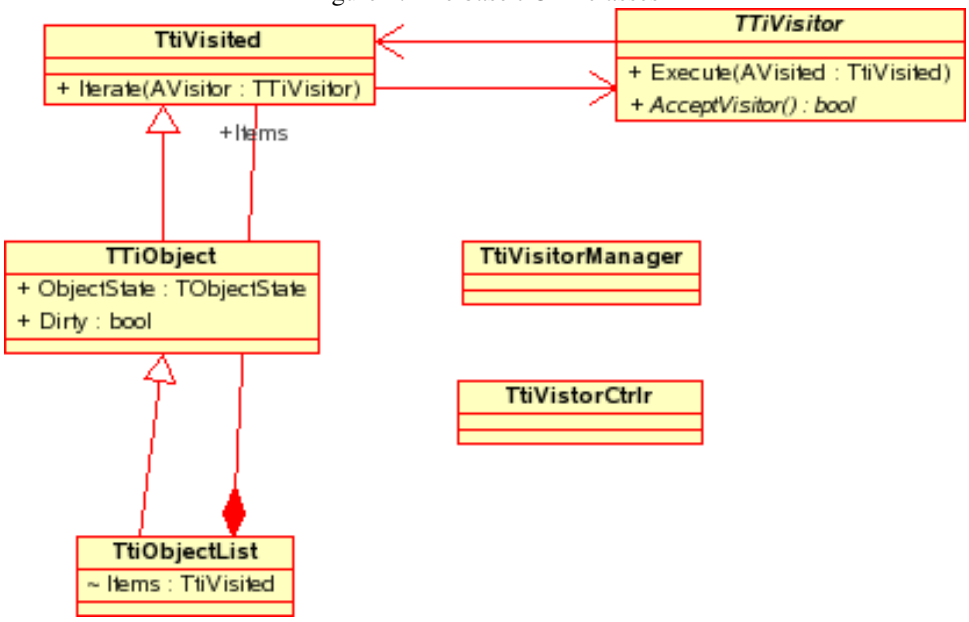
The base class for all persistent objects is `TtiObject`, which means that any object that must be persisted by tiOPF should be a descendent of this class. (similarly, any object that must be streamed in a Delphi form needs to be a descendent of `TPersistent`. It introduces quite a number of properties and methods. The most important one is probably the `ObjectState` property. This property indicates the state of the object, relative to the database: it's an enumerated value of type `TPerObjectState`, which is defined as follows:

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

The meaning of these values is as follows:

posEmpty The object has been created, but has no data filled in.

Figure 2: The base tiOPF classes



posPK The object has been created, and it's primary key data was read from the database. This can be the real primary key (an auto generated number) or some extra data, enough to show the object in a list.

posCreate The object has been created and filled with data. It should be inserted in the database.

posUpdate The object was modified and should be saved to the database.

posDelete The object was deleted, but still needs to be deleted from the database.

posDeleted The object was deleted, and also deleted from the database.

posClean The object is synchronized with the database.

Obviously, if `ObjectState` is one of the values `posCreate`, `posUpdate` or `posDelete` then the object must be saved. Equally, if the object owns some other object and this object is in one of these states, the object as a whole must be saved.

Equally important is the `OID` property. This is the unique key for the object, and is of type TOID or Int64, depending on how tiOPF was compiled. It should be always be loaded from the database. It is used to uniquely determine the object instance.

Note that the object state is not updated always automatically, sometimes it must be updated manually. Particularly: setting a property (e.g. TPerson.Name) in code does not set the ObjectState to `posUpdate`. For this to happen, a Write handler must be used, or it should be set manually.

Other than `ObjectState`, important methods are:

Assign This basically does what `Assign` in `TPersistent` does: it should copy all relevant properties from one instance to another. For all published properties, this is done automatically. To copy public or class-type properties, the `AssignPublicProps` or `AssignClassProps` methods must be overridden, they are called by `Assign`

AssignPublicProps This must be overridden to copy any properties that are not published, and about which tiOPF has no knowledge. `ObjectState` and `OID` are copied automatically.

AssignClassProps This is the tiOPF solution to the 'part of' or 'reference' relationship problem for properties that are a class in themselves (for instance TAdresType in the contact application): if the class is `part` of the 'owning' instance, then the whole class must be copied. If it is a reference to a class which exists by itself (a TCountry, for instance) then only the reference must be copied. It is up to the programmer to decide which of the 2 it is.

Clone If `AssignPublicProps` and `AssignClassProps` are set up correctly, this function creates an exact replica of the object. This can be used when editing an object: create a replica, let the user edit it, and assign it to the original when done. If the user cancels the edit, the replica is simply destroyed, and the original object is untouched.

Find this method can be used to search for a certain object in memory, starting at the current object and descending down any owned objects. It can be passed a OID, or a method which is called for all objects.

The `TObjectList` class is a list of `tiObjects`. It has the following properties

Count the number of items in the list.

Items the items, they are of type `TtiObject`.

CountNotDeleted the number of items in the list that are not marked as deleted. (this is important, as marking an object as deleted does not immediatly remove it from the list or from memory)

OwnsObjects Similar to the property found in the standard `TObjectList` from delphi: Are the objects freed when the list is freed ? If the list contains references to other objects, this property should be `False`

ItemOwner This is the object which owns the objects in the list. By default, this is the list class. If the list represents a series of objects in a property of some class (for instance TCountry.Cities; then the class is the owner of the objects (the Country instance owns the cities, not the cities list itself)

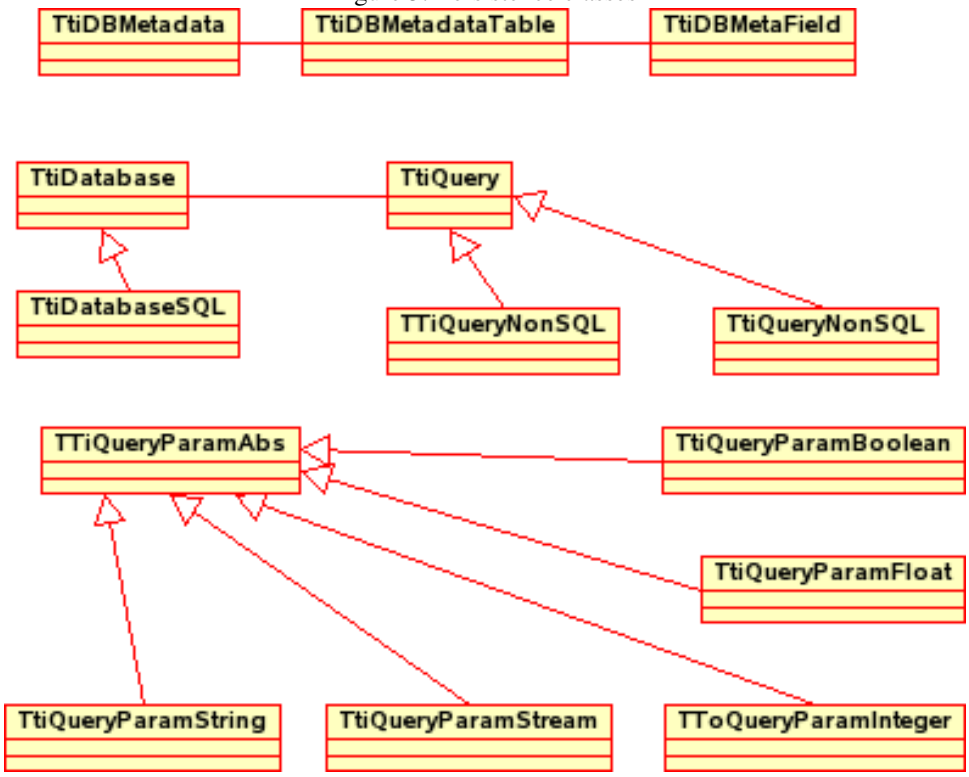
The list has the usual operations as `IndexOf`, `Add`, `Insert` `Delete` and `Remove` which one expects in a list. The `Iterate` method will automatically iterate over all elements in the list.

Each application that uses tiOPF will always use a `TtiObject` descendent and `TtiObjectList` descendent in pairs: One for a single instance of an object, one for a list of instances. For the contacts application, this would be `TCountry` and `TCountries`, or `TContact` and `TContacts`. A 'read' visitor would be used to fill a `TCountries` list with `TCountry` instances. In fact this concept is so important that `tiOPF` contains code templates for a descendent and a list descendent which can be copied and filled with the correct code for the business class.

When loading filtered lists of objects (for instance, a list of contacts that live in a certain country), it's a good idea to create a new lists that describes exactly this: TCountryContacts, and register a specialized read visitor which fills this list.

The `TtiVisitorManager` class is used to keep a list of known visitors. Each visitor has a command name associated with it. When the persistence framework needs to fulfill

Figure 3: Persistence classes



a certain operation on a TtiObject (e.g. saving), it will execute *all* iterators registered with this command name, in the order that they were registered. This is important to note: the 'save' command will usually use 3 iterators: an insert, an update and a delete iterator. All 3 iterators will be tried on the instance. Each iterator must therefore check whether it is appropriate, i.e. a delete iterator should only operate on the object if it is in the `posDelete` state. What is more, an iterator should check whether the object it gets passed in it's `Execute` method is of the correct type: the iterators do not have a class associated with them, so they are tried all, till one fits.

6 Persistence classes

The classes presented in the previous section are the classes that will be used by the programmer to code the business model. In this section, the classes that take care of the persisting are presented.

There are several classes involved, and the classes depend on the way one wishes to perform the persistence. However, there are 2 kinds of classes that form the basis of the persistence engine. The classes are presented in figure 3 on page 14. There are a lot of them, and each has it's function.

The three classes at the top are auxiliary classes: they represent the metadata of a database, i.e. the table and field definitions. These are used when setting up mappings between classes:

TtiDBMetaData represents the database, which has an array property `Items` of type `TtiDBMetaDataTable`.

TtiDBMetaDataTable represents a table in the database. It has an indexed array property `Items` of type `TtiDBMetadataField`.

TtiDBMetaDataField represents a field in a table.

The `TtiDatabase` class abstracts the database. A specific persistence layer (IBX, FBlib, XML etc) will create a descendent of this class. The class has the usual properties (`DatabaseName`, `UserName`, `Password`, `Connected`). More importantly, it has a function to create an instance of a `TtiQuery` query class, which can be used to execute queries on the database. The `TTiQuery` class has also the usual properties (`SQL`, `FieldCount`, `FieldNames` and various array parameter properties such as `ParamAsString`: they are abstract functions, which should be implemented in descendent classes: Each persistence layer creates a descendent of this class to go with the database descendent. The word `Query` should be regarded as broader than the strict SQL sense of the word: It should be more viewed as an object representing rows in one or more tables.

The full declarations of these objects would take too much to enumerate. Suffice it to say that these objects resemble the normal `TDatabase`, `TQuery` and `TParams` counterparts from Delphi. These objects reside in the tiQuery unit, and are completely independent of the DB unit, which makes it in theory possible to create tiOPF applications with a personal version of Delphi, which does not have DB support.

The `TtiDatabaseSQL` and `TtiQuerySQL` classes are intended for SQL based databases, whereas the `TtiQueryNonSQL` class should be used in case no `SQL` based database is used, it implements support for parameters, based on `TtiQueryParamAbs` and it's descendants `TtiQueryParamInteger` etc. It's used for instance in the XML persistence layer.

All these classes are collected in the `TtiOPFManager` class. This is the central class of the persistence layer: There is one global instance of this class, and it collects the various vistors, manages the persistence layers, and connects to or disconnects from the database as needed:

LoadPersistenceLayer Loads a specific persistence layer. This can be dynamical (loading a package) or statical (if it's compiled-in).

UnLoadPersistenceLayer Unloads a loaded persistence layer.

ConnectDatabase connects to a database, using the loaded persistence layer.

This functionality is implemented using the `TtiPeristenceLayers` and `TtiPersistenceLayer` classes: Each persistence layer is described to the framework using a `TtiPersistenceLayer` class, and is managed by the `TtiPeristenceLayers` class.

the `TtiOPFManager` class also maintains the visitors, using the following classes:

RegisterVisitor this can be used to register any kind of visitor.

RegReadPKVisitor this can be used to register a visitor which loads an object from the database layer, and puts it it posPK mode. It will be used in the `ReadPK` call.

RegReadThisVisitor this is used to completely read a single object from the database layer.

RegReadVisitor this is used to completely read one or more objects in a list from the database. It will be used in the `ReadThis` call.

RegSaveVisitor this is used to completely write one or more objects in a list to the database. It will be used in the `Save` call.

The following methods do the actual reading/writing of objects from the database, using the visitors registered with the calls above:

ReadPK Read the Primary Key information for an object from the database.

ReadThis Completely read a single object from the database.

Read Completely read a one or more objects in a list from the database.

Save Save an object or a list to the database.

When manually saving or loading an instance of an object, the global `gTIOPFManager` instance can be used. For instance, to create and save a single contact object, one would write code like this:

```
Var
  T : TContact;

begin
  T:=TContact.Create;
  T.FirstName:='Michael';
  T.LastName:='Van Canneyt';
  T.Email:='michael@freepascal.org';
  gTIOPFManager.Save(T);
end;
```

The persistence layer takes care of the rest.

7 Conclusion

As it was said in the beginning of this article: tiOPF is not for the faint of heart. After all that was written here, with all the classes and ideas introduced here, it is finally possible to start explaining how to start writing an application. Many things have not yet been explained: no interaction with the GUI, no Queries have been written.

At the end of this article, only the basis for working with the tiOPF framework has been laid. Much has not yet been covered: tiOPF is big. But the basic building blocks of any tiOPF application have been presented, and will be used in a future contribution to code a contacts application as it was done for InstantObjects. At that time, some of the GUI elements in the tiOPF framework will also be presented.