Object Persistence in Object Pascal

Michaël Van Canneyt

October 7, 2006

Abstract

"Traditional" Object Pascal focuses on a dataset-oriented way of programming databases, mainly using relational databases and DB-Aware controls, using the TDataset paradigm. In a series of articles, an alternative approach to database programming will be examined: Object Persistence.

1 Introduction

Delphi and it's VCL have always offered powerful tools for database programming: the TDataset and it's many descendents, combined with powerful DB-aware controls made database programming a breeze. Programming moderately complex relational databases can be done without a single line of code, making Delphi a real RAD tool.

In this way of working, the focus lies on the database: the datasets provide various views of the database, and allow to manage the various records in the various tables. The fact that Delphi uses an Object-Oriented language (Object Pascal) is visible only in the VCL: the components to create the GUI and the Data Access components are created using an Object Oriented design.

However, the data managed in the program is not object oriented by itself: it's just records in tables, related and ordered by the database model.

Even the programming itself is almost procedural: one doesn't have to write a single class manually. Simply filling in some event handlers. That most handlers and procedures are methods of a real class which descends from `TForm`, is a fact that many Delphi programmers need not worry about, indeed, some don't even know about it. Only when more advanced programming is done, the Object Oriented nature of Delphi pops up.

In true Object-Oriented programming or Model-Driven programming, a completely different approach is used. There an Object Oriented model is created. Often this is done UML (Unified Modelling Language), which is a powerful language to describe object-oriented software designs. It describes the application in functional terms (use-case diagrams), the classes involved in the application, together with the relations between the classes. There are also diagrams to describe the interactions between classes (possibly using a timeline), component diagrams describing the various components (i.e. parts) of an application, and even deployment diagrams. All these provide different views of the application. UML does not care how the model is stored in the database. Indeed, the presence of a database is completely irrelevant for the design.

Storing the model data in a database is done using Object Persistence. There are various sets of components available for Delphi and Lazarus which perform this task: object persistence frameworks. All of them provide ways to program a model, and to store it in a database. What is more, they provide also GUI components to create an application which manages the classes. Indeed, in an earlier article by the author ('RTTI components in Lazarus' **Joerg,**

can you find the issue for this ?), a minimal and simplistic object persistence framework for Lazarus was presented.

Examining and comparing these frameworks will be the subject of this and subsequent articles.

2 Comparing traditional and model-driven programming

To illustrate the creation of an application using object persistence more clearly, let's take a look at how a contact management application would be constructed in a traditional application, and how it is done using a model-driven approach using object persistence. Obviously, the application is a database application.

The functional analysis is the same for both approaches: What is needed is an application which manages a list of contact persons (just a first/last name, mobile number and email), and to each contact person a series of addresses can be associated. Each address is identified by its kind (home, work, etc.). What is more, the cities and countries should be selectable through a list, to avoid double entry and differences in spelling.

In a traditional application development cycle, one would e.g. start with a relational database design, and create 5 tables:

Countries Containing e.g. the ISO code and name of the country.

Cities Containing cities as entered by the users, possibly adding a ZIP code. A foreign key establishes the relation with the country.

AddresTypes The table with address types. Strictly speaking this is not necessary.

Addresses Containing all addresses. One foreign key establishes the relation with a city, and one establishes the relation with a contact.

Contacts A list of first and last names, emails, mobiles.

How to create the primary keys and the foreign keys between the tables is a choice of the programmer:

1. The country can be identified by its ISO code, the city by its zip code and country, and the contact by the first/last name.
2. Another method would be to assign a unique ID (using some autoincremental value) to each record in each table, and use that as the primary key. A unique index can be used to ensure the uniqueness of the ISO code, ISO/Zip cobination, and first/lastname combination.

After the database is created, the application will be programmed. The details of the application depend on the choices made in the database model.

In a model-driven design, the development cycle is different. First, a model is created. It will consist of several classes:

TContact A contact person class. It will have 5 'attributes' (they would be implemented as propertiesin Object Pascal): Firstname, lastname and birthday, mobile and email.

TCountry A country class, with 2 attributes: Name and ISO code.

TCity A city class, with 2 attributes: zip code and name.

TAddress An address class, with several attributes: Street, telephone, fax.

Then, the relations (associations) between the various classes are described:

1. Each address instance is associated to 1 and exactly 1 city.
2. Each city instance is associated to 1 and exactly 1 country.
3. Each address instance is associated to 1 and exactly 1 person. The association also specifies the kind of address.

In the last item, there is some choice: it could be that 1 address is related to different contacts (as in several people living in the same house, or working in the same company), thus avoiding duplication. However, it is a deliberate choice not to do so.

Note that the kind of address could also be specified in the address itself, but instead this attribute is specified in the relation. There is no pressing need to do this, but it allows to illustrate a point which is a marked difference with the relational-database design. To be able to set the 'kind' attribute in the association between city and contact, a separate class must be created to represent the association.

The above steps closely actually quite resemble the design of the relational database approach: Indeed, it would be hard to see the difference. To introduce a difference, a small enhancement is made to the model. Some of the contact persons are business contacts. So a new class is introduced:

BusinessContact A business contact person class. It inherits from Contact, but has 2 additional attributes: Company and Title.

In the relational model, this would amount to 2 optional fields in the Contact table.

The details of creating an application for this model, and the creation of the classes in the model, depend on what object persistence framework is used.

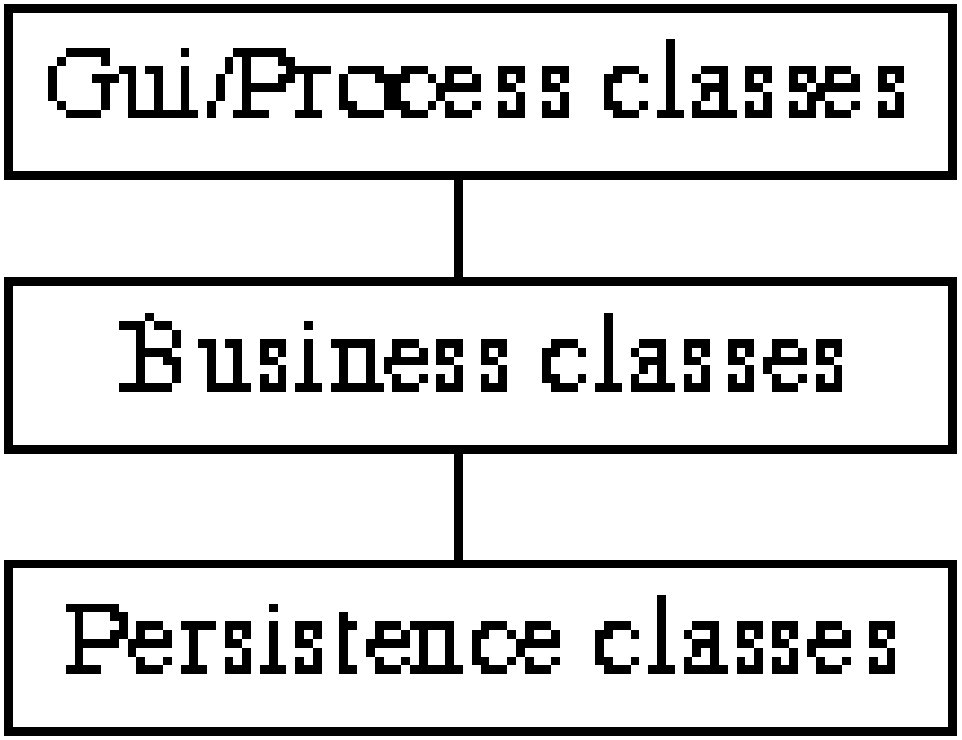
3 Object Persistence: Some theory

Object persistence is obviously not bound to Object Pascal. It is bound to Object Oriented programming (even for languages that do not have language features for Object Oriented programming: a Persistence layer for C also exists). Indeed, for Java, many object persistence frameworks exist, ready made: Hibernate, or JavaBeans. A quick search on Sourceforge reveals many persistence layers for Java, C#, C++ or PHP. And of course, for Object Pascal, persistence layers also exist.

How does it work ? The idea is that an application is built in 3 layers, as depicted in figure 1 on page 4. The top layer is the GUI layer (or process control layer, for daemons and services). It knows nothing about databases, tables, object persistence. It only knows the business objects it has to deal with: `TContact`, `TCountry`, in the case of the contacts application. It only interacts with these objects, which form the middle layer: the business classes. The GUI layer creates instances of them, sets properties and so on. In short, it manipulates the data represented by the business classes. The business classes are built on top of the persistence layer. That is, each business class which needs to be stored somehow, tells the persistence layer what it wants loaded and saved from a persistence storage (usually a database): It tells the persistence layer which properties, links and so forth should be stored.

In a good persistence layer, the business classes have no need to know *how* these properties are stored. It just tells the layer what needs to be stored, not *how* this is done. In the case

Figure 1: Application layers



of a `TContact` class this means that the `TContact` class tells the persistence layer that it wants its 'Firstname' property stored. It does not tell the persistence layer where (which database, which table, which field) this property should be stored: this decision is up to the persistence layer.

Indeed: The persistence layer can be configured to store the 'FirstName' property in a SQL database in a table 'Contacts' in column 'Firstname'. But it could equally well be configured to store this property in another table 'Persons': The `TContacts` class by itself is not aware of this. Indeed, the exact storage could change over time.

This is an important feature: the storage is totally separate from the business logic. This allows flexibility in the storage of the objects: the GUI logic and business logic are unaffected by this; The GUI logic only knows the business classes, and never interacts with the persistence layer at all. The Business logic does interact with the persistence layer, but has no knowledge of the storage details.

Ideally, the persistence layer allows the program developer to concentrate only on the business classes, without having to worry about the storage details. Likewise, and equally important: it should allow the database administrator the freedom to move data in such a way that the data throughput is maximal: the business classes should remain unaffected by this.

Note that in the above description, nothing is said about how the base class tells the persistence layer what properties it wants stored. This can be accomplished in essentially 2 ways:

1. Automatically, through the use of Run-Time Type Information (also called introspection in Java and .NET circles). All published properties will be made persistent.

2. Programmatically: By overriding some methods introduced for this purpose, the per-

sistent classes indicate which properties they want stored.

In both cases it's the responsability of the programmer that programs the business classes to make sure that all relevant properties are made persistent. In the former case by making sure they are published properties, in the latter by enumerating the properties in the appropriate methods. Needless to say that the former method requires less code.

To be able to accomplish all this, the persistence layer itself is comprised of several parts:

Base Class This is the base class for all persistent objects. It is the entry point into the persistence framework for the base classes, and defines some methods to interact with the persistence layer.

Mapper This is a layer which defines how the storage is done: it maps classes and properties to databases, tables and fields. It can be configurable, or not. This mapping could even, in principle, change in time.

Query Engine The persistence layer must have a mechanism to retrieve single objects - this is simple. But it must equally be able to fetch a collection of objects that match a set of criteria. (e.g. all contacts living in a certain city)

Storage This part takes care of the actual storage: it can execute queries on one or more database engines.

In an actual implementation, each of these parts is comprised of several classes. Many implementations will not offer the option of a configurable mapper: the mapping from object to database will be determined by the structure of the business objects, but this need not be so: a run-time configurable mapper could e.g. enable a database administrator to move data between tables.

Note that the application programmer should be unaware of all this; He just gets the business classes, and can use only these.

In practice, the mapping from objects to databases is usually a fixed one, enforced by the persistence layer: it will assign tables and fields to store the data. since RDBMs systems are not ideal implementations, often it will be needed to fine-tune the queries, and it will be necessary to write SQL statements anyway, making it hard if not impossible to move data.

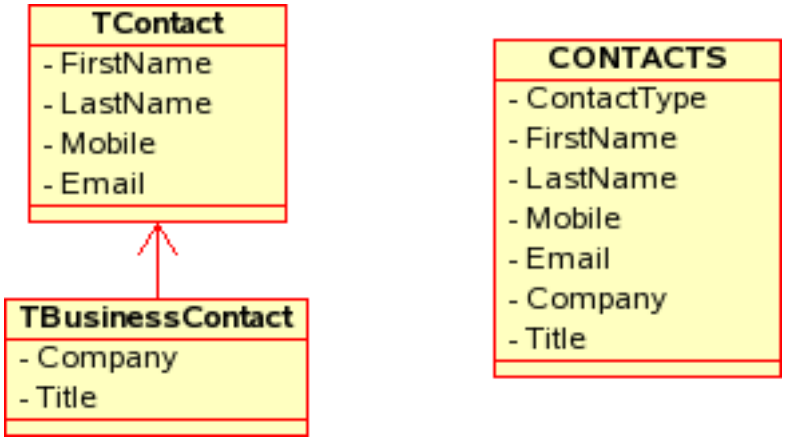
Also, since a program needs to know what database to contact - for instance, a bookkeeping system could store the data of various companies in a database per company - the GUI programmer will need to interact with the persistence layer anyway, telling it where the data is stored.

Another issue is reporting: SQL and relational databases are a standard. Objects are far from standardized, making it hard to create a universal reporting engine for it. Indeed, most reporting engines assume a RDBMs which understands SQL. If the end-user is allowed to run reports on his data (that is what it is for, after all), then being able to chang the location of the data is a bad idea: all of a sudden, the reports will no longer work. Also, while data may be stored in a way that is optimal for loading and saving the objects, for the end user the table structure may not be so transparent.

4 Mapping objects to relational databases

Although it was argued that any form of storage can be used to store the objects, and even multiple storage mechanisms can be used at once, in practice storage is done in a single database, most of the time an SQL-capable database.

Figure 2: One table per hierarchy



Mapping objects to tables and fields in a database is a tricky issue: performance is always the most pressing factor. The requirements of fast loading or fast querying of the database do not always point to the same solution.

Broadly speaking, there are 2 issues when mapping objects to tables in a database:

1. Mapping class hierarchies to tables. It is possible to put each class in it's own table, or a complete class hierarchy in one table, and some stages between these extremes.
2. Mapping class relations to tables and foreign keys. Depending on the case, it makes sense to put an object in a separate table, or store it in the table of the object that owns it. For example, the 'AddressType' can be stored in a field the address table, but can also be stored in a separate table, with a pointer to it in the address table.

To map class hierarchies, there are several options:

One table per hierarchy for each class hierarchy, one table is used. The term hierarchy refers to inheritance. In the example of the contacts application, the 'TBusiness-Contact' and 'TContact' classes would be stored in a single table. The attributes of 'TBusinessContact' are then nullable fields in the table. On top of the fields which map to properties, an additional field would exist, which indicates the type of object of a particular row. For more complex hierarchies, a series of boolean fields may be more appropriate: one boolean field per class in the hierarchy. This situation is depicted in figure 2 on page 6.

One table per class In this approach, each declared class gets it's own table, with just the properties of that class declared as fields in the table. A system of foreign keys would then relate the class inheritance: the foreign key indicates the record where the parent class' information is stored. This situation is depicted in figure 3 on page 7.

One table per concrete class In this class, each class that is directly instantiated gets a table which contains a field for all the properties of the class *and all parent classes.* obviously, this can create a lot of duplicate data. This situation is shown in figure 4 on page 7

The generic solution This is the most generic solution. It works with 5 tables, related through foreign keys:

Figure 3: One table per class

Figure 4: One table per concrete class

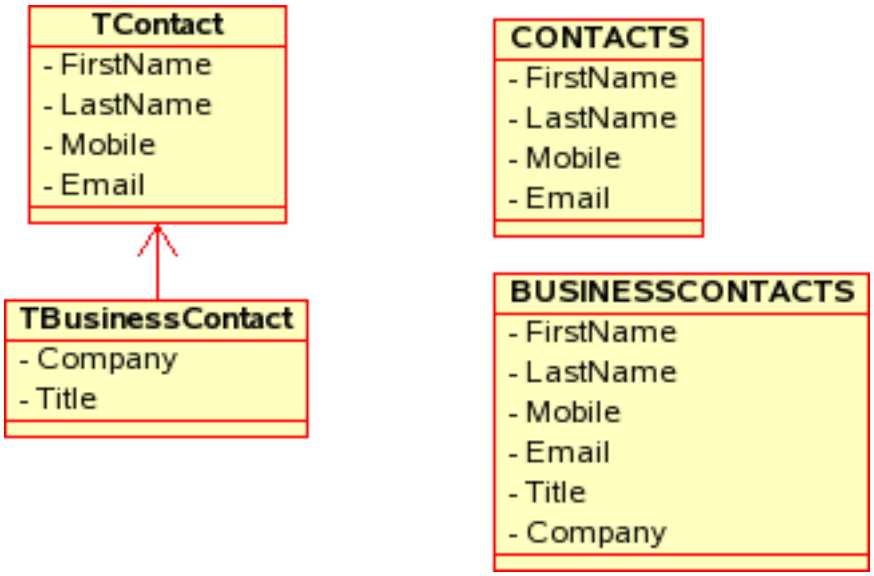
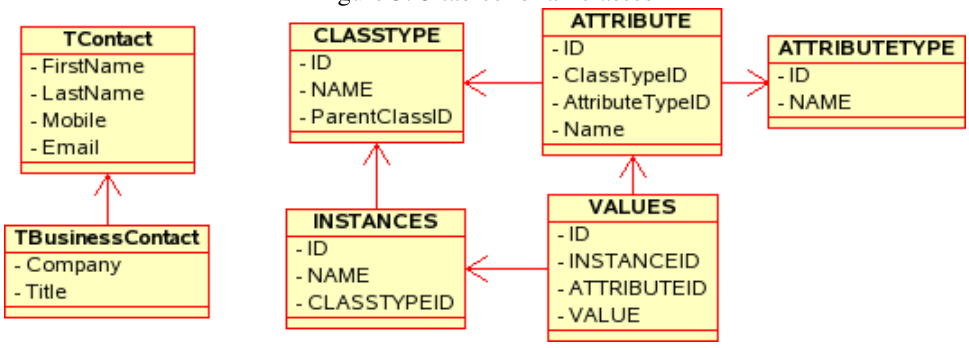


Figure 5: 5 tables for all classes



1. A table for class definitions. Each class gets a record in this table, with an indication of it's parent class (no multiple inheritance).
2. A table for attribute types. One record per attribute type (string, integer, date) is stored here.
3. A table for attributes: a record for each attribute of each class.
4. A table for class instances. A record per instance, irrespective of the actual class: a pointer to the class type indicates the class of the instance.
5. A table for attribute values. For all attributes of each instance, a record exists. This figure is depicted in figure 5 on page 8.

Retrieving all possible attributes of a given instance would be done as follows:

```
SELECT
  CLASSTYPE.NAME as AttributeClassname,
  ATTRIBUTE.NAME as AttributeName,
  ATTRIBUTETYPE.NAME as AttributeType,
  VALUE as AttributeValue,
FROM
  INSTANCE
  LEFT JOIN VALUES
    ON (INSTANCES.ID=VALUES.INSTANCEID)
  LEFT JOIN ATTRIBUTE
    ON (VALUES.ATTRIBUTEID=ATTRIBUTE.ID)
  LEFT JOIN ATTRIBUTETYPE
    ON (ATTRIBUTE.ATTRIBUTETYPEID=ATTRIBUTETYPE.ID)
  LEFT JOIN CLASSTYPE
    ON (ATTRIBUTE.CLASSTYPEID=CLASSTYPE.ID)
WHERE
  (Instance.ID=:ID)
```

While this solution does not need a lot of tables and does not need any changes to the database as classes are added to the model, querying this model is a near impossible task, as can be seen from the sample query above. Not to mention that the attributes would be returned as one row per attribute, instead of one row per instance with the attributes in various fields.

Which approach should be taken is largely dependend on the project. The one table per class is the most normalized (in the database sense of the word) solution, but can be harder to query. Most likely, the used scheme will change as the project evolves.

There are not so many ways to map relations between objects to a database. Obviously, databases have their own concept of 'relations': foreign keys, and they map naturally to relations between objects. There are 3 types of relationships between objects:

One-to-one In this case, the foreign key may be in either of the tables that represent the 2 objects. Take the case of an inventory of overhead projectors in a school: There could be 2 classes `TProjector` and `TRoom`. The relation between the room and resource could be described by properties `TRoom.Projector` and `TProjector.InRoom`. The foreign key for this relation could be located in the table holding `TRoom` data, but it could also be located in the table holding `TProjector` data.

One-to-many Here, there is no discussion where the relationship is stored. In the contacts application, the relation between `TCity` and `TCountry` is stored in the table that holds `TCity` data: the pointer to the `TCountry` instance is stored there.

Many-to-many Normally, this is implemented with an associative table: an auxiliary table which implements a one-to-many relationship with the original tables. In the case of a school, there is such a relationship between a `TTeacher` and `TStudent` class: The teacher teaches many students, and a student is taught (different subjects) by many teachers. The only way to correctly map this is to introduce a new table which holds a reference to both the table for `TTeacher` and the table for `TStudent`.

One could implement this by having 10 fields (`Student1` to `Student10`) in the table for the `TTeacher` class, and having e.g. 5 fields (`Teacher1` to `Teacher5`) in the table for `TStudent`. Obviously, this would mean that a teacher can only teach 10 students, and vice versa, a student can have only 5 teachers. This can be sometimes a valid approach - if it is known that the number of relations is limited - but mostly this will not be the case.

An important difference between object and database relations is that database relations are always bi-directional: a foreign key can be traversed in 2 directions. For the contacts application, this means that it's always possible to construct a list of cities that are in a certain country. But this relationship does not necessarily exist in both directions in the object model: a `TCity` class may have a `Country` property which refers to a `TCountry` instance, but the `TCountry` class need not have a `Cities` property (a `TObjectList`) referring to all `TCity` instances that point to it.

It may be that extra data needs to be introduced in the objects so the object persistence framework can do it's job: a unique ID for each instance (using a generator, or a GUID). This makes it easier to load the object instance from the database. A timestamp is usually also added, to check for concurrent access to the database, and a boolean property to indicate whether the object was loaded from the database or was newly created. This data is not part of the object model by itself, but is needed for efficient functioning of the persistence framework. Obviously the timestamp and ID will be saved in the database.

One advantage of such an ID is that all primary keys and foreign keys have the same structure. For most database systems, comparing a single integer or GUID is also a much faster operation than comparing keys based on string operations; furthermore the indexes maintained by the database will be uniform and - in general - smaller than indexes based on character fields.

Obviously, all the above is only valid for a datamodel for which a new database will be created. Sometimes a model needs to be made representing data in existing databases. In that case, the mapping from model to database will not always be possible in a clean and efficient manner. Nevertheless, a good Object Persistence framework caters for this situation as well.

5 Available Object Persistence frameworks

After having seen what an Object Persistence framework should do, it is time to show that Object Persistence frameworks for Delphi (or Lazarus) exist. Fortunately, several exist. An overview:

ECO comes with the Architect version of Delphi as of Delphi 2005. It's a full-fledged environment that enables Delphi to be used for model-driven programming, and provides a lot more than just Object Persistence; It is a full-fledged UML tool, which as one of it's features provides object persistence. It has grown from Bold for Delphi, which was acquired by Borland. It can use most database access technologies supported by Delphi as storage mechanisms.

InstantObjects Originally a closed-source solution for Object Persistence, it was open-sourced some years ago, and continues to be developed rather actively by a team of programmers. It provides object persistence mechanisms for many database engines, and offers a modeling tool which is integrated with the IDE. It has been around for some years, and is very stable. One of its key features is that it allows to use ordinary DB-aware controls to be used to represent the objects, by using a clever bridge between collections and datasets. The largest part of it is usable with Lazarus as well. More info can be found on the website:

`http://www.instantobjects.org/`

tiOPF is an open source object persistence framework which has been developed at TechInsite, Melbourne, Australia. It also has been around for many years, it's very stable, and comes with an extensive testsuite, based on DUnit. It introduces a number of visual controls which are persistent-object-aware. The latest versions in Subversion compile with Lazarus. More information on tiOPF can be found on

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

Wich offers a comprehensive documentation download.

DePO is an Object Persistence framework which offers less features than the above three, but is definitely usable as well: it comes with a lot of Database engines. Started by Cosimo de Michele, it is now maintained by Cesar Romero. Downloads can be found on

`http://www.liws.com.br/depo/`

Note that documentation is scarce, and mostly in Portugese.

Jazz is the successor of DePO by Cesar Romero. It is currently in development, and more information can be found on

`http://jazz.liws.com.br/`

SnapObjects is the successor of DePO by Cosimo de Michele. It is demonstrated and downloadable on

`http://digilander.libero.it/snapobject/`

In subsequent articles, most of these persistent frameworks will be examined, and their features will be compared. More importantly, the advantages and disadvantages will be compared to the traditional way of developing a database application.

6 A contact manager application: The objects

To program a contacts application with Objects, the data in the contacts application must be defined as classes (or even interfaces). As described earlier, 5 classes will be needed to describe the model data:

TCountry will represent a country.

TAddressType will represent an address type. Whether this should be a separate persistent class or not, remains an issue to be solved.

TCity will represent a city. It will - somehow - refer to a TCountry instance.

TContact will represent a contact person. It will have some kind of array property which will contain the various addresses for this contact person.

TBusinessContact will represent a business contact person. It inherits from TContact.

TAddress will represent an address of a contact person. It will have a reference to a `TCity` instance, and possibly one to a `TAddressType`.

How these classes are written for the sample application, depends on the object persistence mechanism. More in particular: all the mechanisms require that the base class for all objects that must be stored persistently, depend on a certain base class, which is different for each framework. Some of the frameworks require the properties to be published, for others, the properties may be public; Some require Get/Set methods, others do not. It is therefore hard to give the declarations for the objects above, and this will be postponed to the articles describing the various frameworks.

7 The traditional RAD approach: The database

The contacts application will be programmed in the various Persistence Frameworks, to be able to compare the features and drawbacks. To be able to compare the advantages (or drawbacks) of model-driven programming with Database Driven programming, the Contacts application will first be programmed using a traditional RAD database approach.

This means a database will be needed: The design of the database should be known before the application is programmed: The GUI interacts in a rather direct way with the database using TDataset and DB-aware controls.

As a database engine Firebird will be used, but it should work equally well on other database engines. There will be 5 tables, as described earlier. Each table contains an autonumber field as the primary key (which is actually an Object-Oriented approach to databases).

There are 3 auxiliary tables, and all of these, the `AddressTypes` is the simplest:

```
CREATE TABLE ADDRESTYPES (
  AT_ID INTEGER NOT NULL,
  AT_TYPE VARCHAR(30),
  CONSTRAINT ADDRESSTYPES_PK PRIMARY KEY (AT_ID),
  CONSTRAINT U_ADRESSTYPE UNIQUE (AT_TYPE));
```

The unique key ensures that each type occurs only once. Almost equally simple is the `Countries` table:

```
CREATE TABLE COUNTRIES (
  CN_ID INTEGER NOT NULL,
  CN_ISO VARCHAR(3) NOT NULL,
  CN_NAME VARCHAR(80) NOT NULL,
  CONSTRAINT COUNTRIES PRIMARY KEY (CN_ID),
  CONSTRAINT U_COUNTRY UNIQUE (CN_ISO));
```

Note the primary key and the unique constraint.

The `Cities` table has a foreign key that refers to this table:

```
CREATE TABLE CITIES (
  CI_ID INTEGER NOT NULL,
  CI_ZIP VARCHAR(15) NOT NULL,
  CI_CITY VARCHAR(80) NOT NULL,
  CI_COUNTRY_FK INTEGER NOT NULL,
  CONSTRAINT CITIES_PK PRIMARY KEY (CI_ID),
  CONSTRAINT U_CITY UNIQUE (CI_ZIP,CI_COUNTRY_FK));
```

The `Addresses` table contains the most fields:

```
CREATE TABLE ADDRESSES (
  AD_ID INTEGER NOT NULL,
  AD_CONTACT_FK INTEGER NOT NULL,
  AD_TYPE_FK INTEGER NOT NULL,
  AD_CITY_FK INTEGER NOT NULL,
  AD_STREET VARCHAR(100),
  AD_STREETNR VARCHAR(15),
  AD_TELEPHONE1 VARCHAR(20),
  AD_TELEPHONE2 VARCHAR(20),
  AD_FAX VARCHAR(20),
  CONSTRAINT ADRESSES_PK PRIMARY KEY (AD_ID));
```

Note that the city and type fields are pointers to entries in the `Cities` and `AddressTypes` tables. This has the advantage that all types and cities can be uniquely identified, but has the disadvantage that they must be present in the `Cities` or `AddressTypes` tables before they can be used. In the case of the address types, this is not such a problem. In the case of cities, this can be a bigger problem if an easy-to-use GUI must be constructed. For web applications, this will not be such a problem.

Each Address is associated with a contact person through the `AD_CONTACT_FK` foreign key:

```
CREATE TABLE CONTACTS (
  CO_ID INTEGER NOT NULL,
  CO_FIRSTNAME VARCHAR(80),
  CO_LASTNAME VARCHAR(80),
  CO_COMMENT VARCHAR(255),
  CO_MOBILE VARCHAR(20),
  CO_EMAIL VARCHAR(80),
  CONSTRAINT CONTACTS_PK PRIMARY KEY (CO_ID));
```

Firebird does not have an auto-number field type. Instead, a unique ID is created using a generator (a unique sequence) per table, which are created using the following statements:

```
CREATE GENERATOR GEN_ADDRESSES;
```

```
CREATE GENERATOR GEN_CITIES;
CREATE GENERATOR GEN_CONTACTS;
CREATE GENERATOR GEN_COUNTRIES;
CREATE GENERATOR GEN_ADDRESTYPES;
```

The generator names are chosen so they match the table names. This requirement will be explained below.

The programmer can retrieve a new ID from the generator, but to ensure that a unique ID is always assigned to a new record, a trigger can be used. For the `Contacts` table, this trigger would look like this:

```
CREATE TRIGGER CONTACTS_ID FOR CONTACTS
ACTIVE BEFORE INSERT POSITION 0
AS
BEGIN
  If (NEW.CO_ID is null) then
    NEW.CO_ID= GEN_ID(GEN_CONTACTS,1);
END
```

Actually, as will be obvious from the below, for a `TDataset` based approach as in Delphi, the key field value needs to be retrieved in advance.

8 The Database-Driven contacts application

Even using the RAD approach to database applications, it is good practice to separate the data code from the GUI code. Delphi provides for this using the `TDatamodule`: designed to contain the data-access components. Even so, it will become apparent that typical GUI elements will find their way into the data layer.

So, to program the contacts application, first a `TDataModule` instance is designed. This datamodule contains the database connection, and datasets for all tables. Various component sets can be used for this, in the example application, DBExpress will be used: a `TSQLConnection` component will represent the connection to the database. `TSimpleDataset` instances will be used for the various queries and tables; The naming scheme is simple: `SDCities` for the Cities table, and so on.

For each table, a query equivalent to `SELECT * FROM TABLENAME` is generated. The query for the addresses dataset (`SDAddresses` is a bit different. It looks as follows:

```
SELECT
  *
FROM
  ADDRESSES
WHERE
  (AD_CONTACT_FK=:CO_ID)
```

Which will retrieve all addresses for a single contact person. The `CO_ID` is a parameter, which makes it possible to use this dataset as a detail dataset in a master-detail relationship - the master dataset being the list of contacts.

For each query, the persistent fields are created, plus some additional lookup fields. For instance, for the `CI_COUNTRY_FK` field in the `SDCities` dataset field, a lookup field is generated which is called `CI_COUNTRY`: it looks up the `CN_NAME` field in a lookup dataset (`SDSelectCountry`, in this case), using the `CI_COUNTRY_FK` as a key field.

The reason for defining the lookup field is that Delphi 'knows' how to display lookup fields in a Data-Aware grid: Instead of showing the `CI_COUNTRY_FK` field in a grid - which would be a meaningless number - the lookup field is shown in the grid: when a lookup field is shown in a field, and it must be edited, then Delphi will present a dropdown list filled with all possible values from the lookup dataset. The chosen value will be stored in the original key field.

The same is done for the `AD_TYPE_FK` and `AD_CITY_FK` fields in the `DSAddresses` dataset.

The `AD_ID` field (or any other key field) will be shown as `Required` in the Object inspector. This means that Delphi will verify that the field has a value, when the `Post` method of `TDataset` is called. Delphi is clever enough to know that an autoincremental field will get it's field value from the database, but for Firebird, the ID field is a normal required integer field: The dataset code will require a value - and Delphi has no way of knowing that the value will be generated in the trigger, and will therefore complain that no ID value was provided when the data is posted.

So, in the `AfterInsert` method of each dataset, a call similar to the following is inserted:

```
procedure TContactsDataModule.SDCountriesAfterInsert(DataSet: TDataSet);
begin
  SDCountriesCN_ID.AsInteger:=GetID('COUNTRIES');
end;
```

Where the `GetID` method looks as follows:

```
function TContactsDataModule.GetID(TableName: String): Integer;

Const
  SGen = 'SELECT GEN_ID(GEN_%s,1) as THEID FROM RDB$DATABASE';

begin
  With QGetID do
    begin
    SQL.Text:=Format(SGen,[TableName]);
    Open;
    try
      If (EOF and BOF) then
        Raise EContacts.CreateFmt(SErrNoID,[TableName]);
      Result:=Fields[0].AsInteger;
    Finally
    end
    end;
end;
```

The `QGetID` class is an instance of `TSQLQuery`, part of DBExpress. Now it becomes obvious why the names of the generators in the database were chosen as `GEN_` plus the table name.

Since DBExpress is used, and `TSimpleDataset` is a client dataset, it's `Post` method saves the changes only locally: only the dataset contains the changes, the database does not yet contain the changes. That is why in the `AfterPost` and `AfterDelete` events, the `ApplyUpdates` call is issued:

```
procedure TContactsDataModule.DatasetAfterPost(
          DataSet: TDataSet);
```

```
begin
  (Dataset as TSimpleDataset).ApplyUpdates(0);
end;
```

This call will actually apply the changes to the database. Obviously, the changes could be cached locally, and applied at a later time - one of the the advantages of using the clientdataset technology.

After all the datasets have been created, the GUI of the application can be programmed. For this, 5 forms are created:

MainForm this is the main form of the application. It shows the available contacts in a (editable) data-aware grid, and shows a menu which allows to edit the 'system' tables: address types, countries, cities.

CountriesForm this form allows to edit the countries table. It's essentially a simple DBGrid and TDBNavigator component, connected to the `SDCountries` dataset on the datamodule. The primary key field (`CN_ID`) is not shown in the grid - this will be so for all grids.

AddressTypesForm this form allows to edit the address types. Again, it is simply a DBGrid and TDBNavigator component, this time connected to the `SDAddressTypes` dataset on the datamodule.

CitiesForm this form allows to edit the cities table. Again it essentially a simple DBGrid and TDBNavigator component, connected to the `SDCities` dataset on the datamodule. Here, care must be taken not to show the `CI_COUNTRY_FK` field is not shown in the grid. Instead, the `CI_COUNTRY` lookup field must be shown.

AddressForm Here, the details of a contact are shown, as well as the addresses for this contact person. The addresses are shown in a grid, and again the lookup fields for type and city are shown instead of the actual foreign key fields.

With the exception of the address form, all secondary forms are shown modally. The address form can be shown nonmodal - as the current contact record in the main form changes, the details are refreshed in the detail form. To achieve this, the following code must be inserted in the `AfterScroll` event of the `SDContacts` dataset:

```
procedure TContactsDataModule.SDContactsAfterScroll(
          DataSet: TDataSet);

Var
  B : Boolean;

begin
  With SDAddresses do
    begin
    B:=Active;
    If B then
      Close;
    Params[0].AsInteger:=SDContactsCO_ID.AsInteger;
    If B then
      Open;
    end;
end;
```

The menu item `OnClick` handlers to show the auxiliary forms are quite simple and straightforward, they look for instance as follows:

```
procedure TMainForm.MICitiesClick(Sender: TObject);
begin
  With TCitiesForm.Create(Self) do
    try
      ShowModal;
    Finally
      Free;
    end;
end;
```

And with this, the whole application is coded.

As is obvious, not much code was needed to create a simple working database application which manages 5 tables which are related through foreign keys: True RAD.

However, there are some points which must be noted:

1. While an effort was made to separate GUI and Data logic (a split between forms and datamodules), the separation is not perfect: the lookup fields are defined in the database part, while strictly speaking, they are only needed for the correct functioning of the GUI (the DBGrid).
2. Care must be taken that a value for the primary key is retrieved when a new record is inserted.
3. After posting the data to the dataset, a second call is needed to apply the changes to the database. This is a consequence of the use of the ClientDataset approach used in the application. In the case of direct access to the database, this would not be needed.
4. Master-detail relationships between datasets are defined in function of the GUI: if a different address screen had been designed, the relation would have been different.

None of these points are difficult or surprising, but they should be identified: they are all consequenses of the chosen RAD approach to creating the database application. They will be contrasted with the approach in object-oriented applications which use object persistence.

The final application looks as in figure 6 on page 17

9 Conclusion

The basis for a series of articles comparing various Object Persistence frameworks has been laid: The purpose of Object Persistence has been described, it's workings have been explained. A sample application has been defined. The various available Object Persistence frameworks have been enumerated, and the Objects that will be needed have been identified. A Database-driven implementation has been made to be able to contrast it with the model-driven implementations. The stage for the subsequent articles has been set: In a series of articles, some of the mentioned object persistence frameworks will be presented and compared: at most one framework per article.

Figure 6: The finished contacts application