Reflective Persistence

(Reflective CRUD: Reflective Create, Read, Update & Delete)

Macario Polo, Mario Piattini and Francisco Ruiz

Escuela Superior de Informática

University of Castilla-La Mancha

Paseo de la Universidad, 4

13071-Ciudad Real (Spain)

mpolo@inf-cr.uclm.es

Tel.: +34-926-295300 ext. 3730

Abstract

This paper presents a pattern for generating persistence methods in runtime, using the Java API Reflection, with high levels of maintainability and reusability.

1 Introduction

Literature has been very prolific proposing patterns for mapping an object-oriented system to a relational database [BRO96], [KEL97], [AMP00], [IBM00]. Two of the problems that one finds are:

- The transformation of the class diagram of the system (usually the Domain layer class diagram) to a relational database schema.
- The assignment of persistence responsibilities to classes, which depends on the strategy used for transforming the class diagram to the relational schema.

There are several strategies for both tasks:

- Different relational database schemas can be produced from a same class diagram.
- The implementation of persistent operations has a string dependence on the pattern used for performing the transformation. Moreover, there are also several strategies (patterns) to select the classes that will receive persistence responsibilities.

Usually, the programmer needs to write the code for all persistent responsibilities, which

depends on the pattern used for performing the transformation and on the pattern used to assign the persistence responsibilities (the persistent class itself, an associated class, etc.).

We propose a pattern that uses Reflection to avoid the need of writing the code for persistence methods.

2 Pattern: Reflective CRUD (RCRUD)

Context

During the development of an Information System, a relational database has been built from the classes in the Domain layer. For each persistent class in the class diagram a table has been constructed, using foreign keys to represent all permanent relationships among classes. The system must be developed in Java.

Problem

How are persistence methods assigned to persistent classes? How are persistence methods implemented?

Forces

- Maintenance effort: very low maintenance costs are desirable. The impact of adding, modifying or deleting fields from classes or columns from the database would desirably be minimum, as well as the time devoted to it.
- Reusability: the solution proposed should be directly reusable in any other system, with a simple "copy and paste".
- Flexibility: the solution proposed should be easily adaptable to work with different types of transformations from the domain diagram to the relational schema.
- Performance versus cost: the proposed solution must not produce delays that are appreciable by users.

Solution

A superclass which uses Reflection to generate persistence methods in runtime must be written. Each persistent class should be a specialization of it.

Structure

Figure 1 shows a generic class diagram which uses the solution proposed. RCRUD is in charge of generating all persistence methods in runtime. Note that the superclass (RCRUD) has no abstract operations.

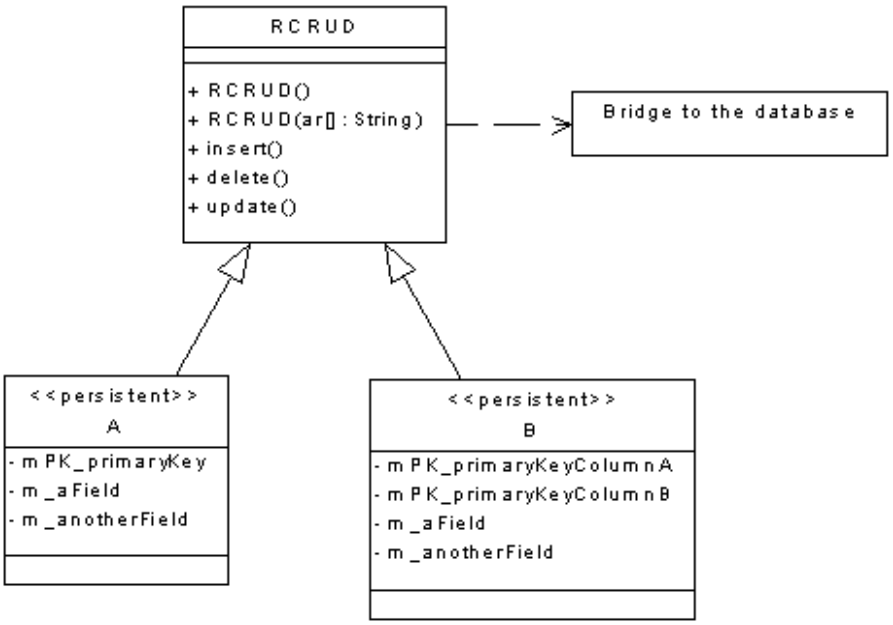

Figure 1. Generic of use of Reflective CRUD

Participants

In Figure 1:

- *A* and *B* are two persistent classes, which need the methods in RCRUD.
- *RCRUD* is the class that, thanks to Reflection, generates CRUD methods in runtime for all persistent classes.
- The *Bridge to the database* is the selected mechanism to access the database. It could be a Broker, a direct connection (through an association, for example), etc.

Consequences

1. Facilitates quick development. To make a class persistent, all that is required is to make it a specialization of RCRUD and to maintain some singular (and common sense) naming conventions (for example: the name of the table corresponding to a class is the

same as the name of the class; the names of the fields in every persistent class that correspond to primary keys start with “mPK”; the names of all the other fields start with “m”).

2. Good maintainability. If a class experiences a change in its structure (fields), all that is needed is the reflection of that change in the corresponding table.
3. Reusability. RCRUD is directly reusable in any other project which requires persistence of objects.
4. Reflective CRUD combines, in a single class, all the advantages of both the Template Method and the Pure Fabrication patterns:

    - With the Template Method, all persistence methods are declared in a superclass: methods, with code common to all classes, are implemented, whereas methods with code that depend on the concrete class are left as abstract operations. In these systems, the Domain layer is low coupled to the persistence layer, since only the superclass has direct knowledge of the database.

    - With the Pure Fabrication, a class is built for each persistent class, which receives the responsibilities of persistence. In this manner, persistent classes have only those responsibilities related to the problem domain, and pure fabrications have only persistent operations. These classes have high cohesion.

    Reflective CRUD implements a Template Method because it contains the definitions of all persistence methods, but with the additional advantage that all of them are concrete. It has also a Pure Fabrication, because all persistence operations are delegated to an associated class (the superclass): therefore, domain classes are not responsible of their persistence, what allows to keep high cohesion of domain classes.

Example

Figure 2 shows a little class diagram which represents the domain layer obtained when modelling a problem. If all these classes need persistence, we can give it to them by doing all of them specializations of RCRUD, as Figure 3 shows.

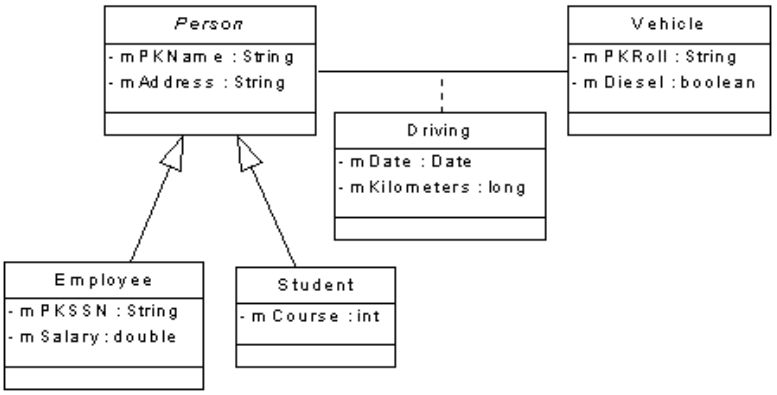

Figure 2. A Domain class diagram.

In Figure 3, every persistent class inherits persistence operations from the RCRUD class, which is in charge of its generation. When an object desires, for example, to be inserted, it executes the `insert` method, which is completely defined in RCRUD.

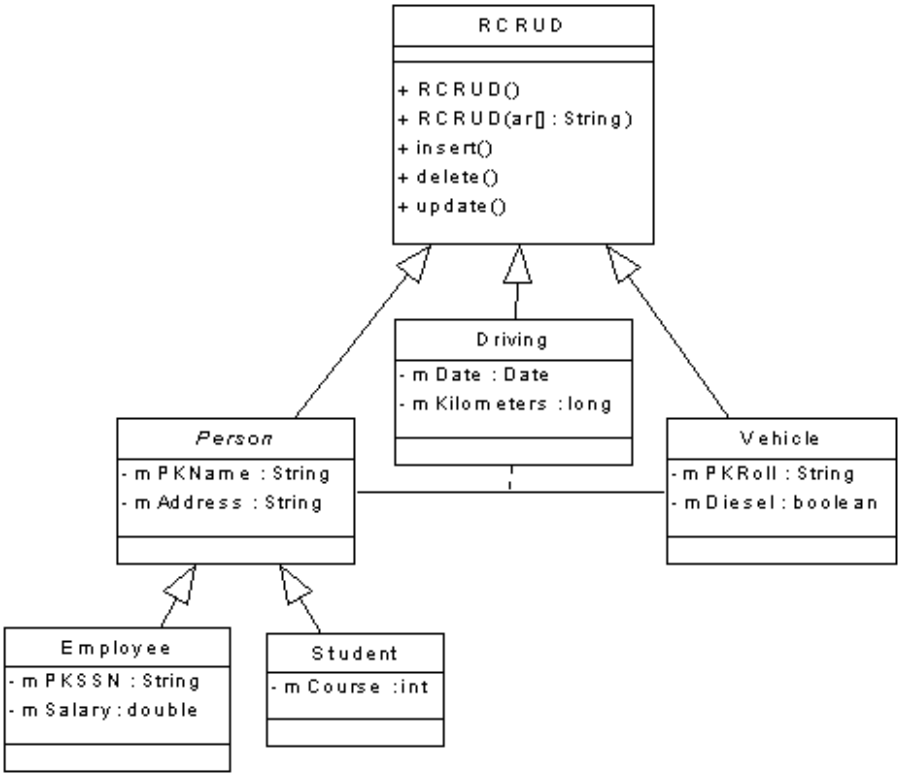

Figure 3. Adding persistence to all classes.

Variants

The Domain diagram is the basis for constructing the relational schema. There are several strategies for transforming a class diagram to a relational schema:

1. To create a table per class, and use foreign keys to represent all permanent relationships among classes (inheritance, associations and aggregations). Using this type of transformation, the relational schema obtained from the class diagram in Figure 2 would be that shown in Figure 4.

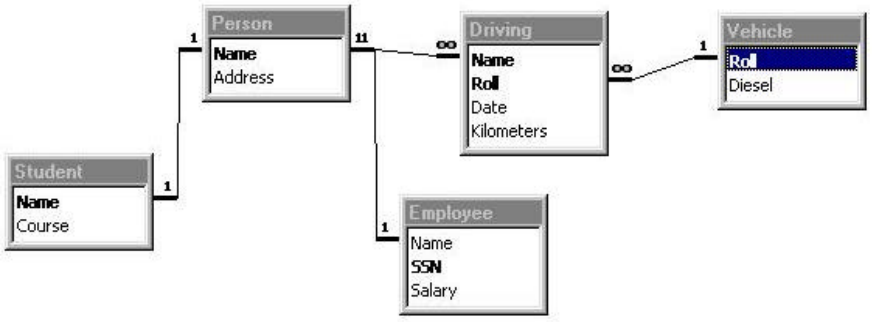

Figure 4. Relational schema obtained constructing a table per class.

2. To use the "One inheritance path, one table" pattern, which reduces each path in an inheritance tree to a single table. Figure 5 is the relational schema obtained applying this transformation to Figure 2.

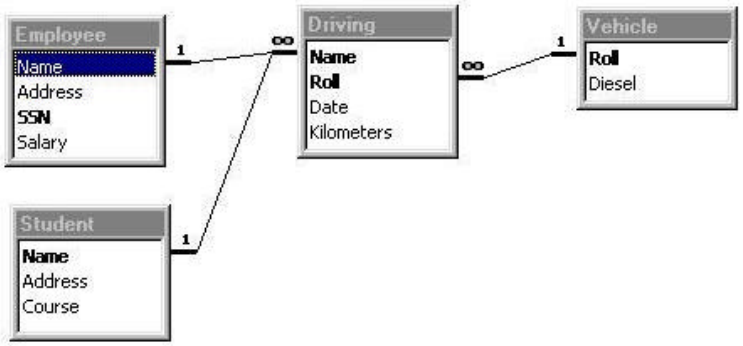

Figure 5. Relational schema obtained from the "one inheritance path, one table" pattern

3. To use the "One inheritance tree, one table" pattern, which reduces a full inheritance tree to a single table, as it is shown in Figure 6, which is the transformation of Figure 2 with this pattern.

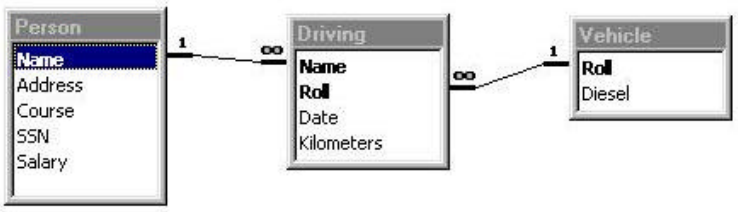

Figure 6. Relational schema obtained with the "One inheritance tree, one pattern"

4. To use different combinations of the previous solutions to transform a same class diagram.

Obviously, methods in RCRUD must have different implementations depending on the type of transformation. For example, when a new instance of Student desires to be inserted in the database:

1) In the first case (Figure 4), it must be firstly inserted in Person, and then in Student, probably through a transaction.
2) In the second case (Figure 5), it may be directly inserted in the Student table.
3) In the third one (Figure 6), it must be inserted in the Person table, leaving nulls the columns of Employee.

Implementation

This section provides some comments and explanations on the first variant of RCRUD: in this one, there exists a table per class, and each class is responsible of its persistence. We begin presenting a constructor for building empty objects, a generic "materializer" (for building objects from records), and we conclude with the methods insert, delete and update.

The full implementation of the three variants of RCRUD illustrated in the previous sections, as well as some examples can be downloaded from the files yet1.zip, yet2.zip and yet3.zip at http://www.inf-cr.uclm.es/www/mpolo/yet/

- **The empty constructor**

The empty constructor must assign a default value to each field in the object (for example, the zero value to `int` and `long` fields, `new String()` to `String` fields, etc.).

```
RCRUD() {
   Field fields[]=this.getClass().getDeclaredFields();
   for (int i=0; i<fieldNames.size(); i++) {
```

```
      try {
         Field f=fields[i];
         if (isValid(f))
            f.set(this, Empty(f));
      }
      catch (Exception e) {}
   }
}
```

This method retrieves through Reflection all the fields in the class. An "empty" or default values is assigned to each "valid" field. A field is "valid" if its type is a Java basic type. With this, we avoid to deal with fields whose type is, for example, a class defined in the domain of the problem (although it is also possible to materialize this kind of fields also with Reflection).

The `Empty(Field)` method is the operation in charge of assigning the empty, default value, depending on the type of the field, which can be retrieved through the `getType()` method provided in the Reflection API. In the implementation of RCRUD that we are currently using, the following is the body of `Empty`:

```
private Object Empty(Field f) {
   if (f.getType().toString().compareTo("class java.lang.Long")==0)
      return new Long(0); else
   if (f.getType().toString().compareTo("class java.lang.Integer")==0)
      return new Integer(0); else
   if (f.getType().toString().compareTo("class java.lang.Double")==0)
      return new Double(0.0); else
   if (f.getType().toString().compareTo("class java.lang.Boolean")==0)
      return new Boolean(false); else
   if (f.getType().toString().compareTo("class java.lang.String")==0)
      return new String(); else
   if (f.getType().toString().compareTo("class java.util.Date")==0)
      return new Date(); else
   if (f.getType().toString().compareTo("class java.util.Vector")==0)
      return new Vector();
   return new Object();
}
```

- **Generic materializer: the constructor with a parameter**

This constructor is used to build instances from the information saved in the database, values of the fields which correspond to columns which are part of the primary key. To build, for example, an Employee whose name is "Maquete" and has 13203881 as SS number, we need to write the following sentence:

```
String args[]={"Maquete", "13203881"};
Employee e=new Employee(args);
```

In the body of this constructor, a `Select` instruction must be generated and executed on the database. Afterwards, the value of each column retrieved must be assigned to each field of the object which is being materialized.

The construction of the `Select` instruction is quite easy (see lines 3 and 4 in the following code): as there is a correspondence between the name of the table and the name of the class, the table in the `From` clause is the name of the class (or the name of the class with some kind of transformation). Then, the method called in line 4 (`listOfPKsAndValues(String[])`) is in charge of generating the `Where` clause; it must look for all the fields in the class whose name starts with "mPK", trim this prefix (to produce the string `Name=` instead of `mPKName=`) and concatenate the elements in the array. Possibly, some kind of marks must be also generated depending on the type of the field (for example: `Name='Maquete'` with quotation marks, and `SSN=13203881` with none).

```
1     RCRUD(String PKValues[]) throws Exception {
        this();
3       String SQL="Select * from " + this.getClass().getName() + " where " +
4          listOfPKsAndValues(PKValues);
        ResultSet r=Broker.bd.createStatement().executeQuery(SQL);
        if (r.next()) {
          Vector fieldNames=listOfFields(this.getClass());
          for (int i=0; i<fieldNames.size(); i++) {
9           Field f=
              this.getClass().getDeclaredField((String)
                                  fieldNames.elementAt(i));
            try {
10            Object o=valueOf(f, r, i+1);
11            f.set(this, o);
            }
            catch (Exception j) {}
          }
        }
      }
```

When the instruction has been retrieved in a `ResultSet`, its columns must be assigned to the valid fields (see the notion of "valid field" in the empty constructor section) in the object, which is done in lines 10 and 11.

- **The Insert method.**

This method must generate a SQL instruction as the one shown in the left side of next table, for which the code in the right side can be used:

Insert into the_name_of_the_table (columns) values (values_of_fields)	String SQL=”Insert into “ + this.getClass().getName() + “ (“ + getListOfColumns() + “) “ + “values” + “ (“ + getListOfValues() + “)”;

Table 1. Generation of the Insert method.

As it is seen, the name of the class is used as name of the table to insert the record. The list of columns is extracted with the `getListOfColumns` method, which has the following implementation:

```
String getListOfTableColumns() throws Exception {
   String fieldName, result=new String();
   Field f;
   Field fields[];

   fields=this.getClass().getDeclaredFields();
   for (int i=0; i<fields.length; i++) {
      f=fields[i];
      if (isValid(f.getType().toString())) {
         fieldName=f.getName();
         fieldName=trim(fieldName);
         result+="[" + fieldName + "],";
      }
   }
   return result.substring(0, result.length()-1) ;
 }
```

The list of values is retrieved as a string with the `getListOfValues` method. Some values must be set between quotation marks, which is done with `marks(String, String)` method:

```
String getListOfValues() {
   String fieldClass, fieldValue, result=new String();
   Field f;
   Field fields[];

   try {
      fields=this.getClass().getDeclaredFields();
      for (int i=0; i<fields.length; i++) {
         f=fields[i];
         if (isValid(f.getType().toString())) {
            fieldClass=f.getType().toString();
            fieldValue=f.get(this).toString();
            result+=marks(fieldValue, fieldClass)+",";
         }
      }
   }
   catch (Exception e) {}
   String result2=result.substring(0, result.length()-1);
   return result2;
}
```

Once the `Insert` instruction has been generated, it is executed against the database.

- **The Delete method**

This method must generate and execute a SQL instruction to delete the current object from the database. It will be a `Delete` instruction which removes from the database the record whose primary key coincides with the fields of this object whose name starts with "mPK".

```
public int Delete() throws Exception {
   String SQL="Delete from " + this.getClass().getName() +
      " where " + listOfPKsAndValues();
   return Broker.bd.createStatement().executeUpdate(SQL);
}
```

The `listOfPKsAndValues()` method retrieves a string with the format `[Name]='Maquete' and SSN=13203881`. It operates in the same way that `listOfPKsAndValues(String[])`), which was commented in the Generic materializer section, but in this case it acts on the current object, not on the array passed as parameter.

- **The Update method**

This method has a special characteristic which distinguishes it from the previous ones: it must generate an `Update` instruction which assigns to the columns of an existing record the values of the fields in this object, but must compare with the old values of the primary key. For example: let us suppose that there is in the database the following person:

Name	SSN	Age
Maquete	13203881	3

We can build an object from this record and then, maybe its SSN is changed:

```
p.mPKName="Maquete"; p.mPKSSN=2801234; Age=3;
```

To save this record in the database, the following instruction must be generated:

```
Update Person set
   Name='Maquete', SSN=2801234, Age=3   ← Current values
where
   Name="Maquete" and SSN=13203881      ← Old values
```

See that the `Set` clause uses the current values of the record, but the `Where` clause looks for the previous values of the primary key in the database (otherwise, the record would not be found). Therefore, a method to keep the initial values of the object/record is needed, in

order to do the following generation of the `Where` clause. We save the initial values in the `oldPKs` class variable of RCRUD with the following method:

```
public void saveOldPks() {
   oldPKs=listOfPKsAndValues();
}
```

It calls to `listOfPKsAndValues()`, which has been commented in the previous section. `saveOldPKs` is executed when the user decides to modify the object which is being edited: for example, when he/she press the `Update` button on the Person screen; then, when the data have been changed and the `Save` button is pressed, the `Update` method, which has the following body, is executed:

```
public int Update() throws Exception {
   String fieldClass;
   String fieldValue;
   String SQL="Update " + this.getClass().getName() + " set ";
   Vector fieldNames=listOfFields(this.getClass());
   try {
      for (int i=0; i<fieldNames.size(); i++) {
         Field f=
            this.getClass().getDeclaredField((String) fieldNames.elementAt(i));
         try { fieldValue=f.get(this).toString(); }
         catch (Exception e) { fieldValue=null; }
         if (fieldValue!=null) {
            fieldClass=f.getType().toString();
            SQL= SQL + "[" + trim(f.getName()) + "]=";
            SQL=SQL+marks(fieldValue, f.getType().toString()) + ",";
         }
      }
   }
   catch (Exception e) { }
   SQL=SQL.substring(0, SQL.length()-1);
   SQL= SQL + " where " + this.oldPKs;
   return Broker.bd.createStatement().executeUpdate(SQL);
}
```

Related patterns

The CRUD (Create, Read, Update & Delete) pattern of Yoder et al. [YJD98] determines that the CRUD methods are the minimal set of operations required to provide persistence to objects.

The Template Method patterns is described by Gamma et al. in [GHJV95]. The Pure Fabrication pattern is detailed in [LAR98].

Patterns “One class, one table”, “One inheritance path, one table” and “One inheritance tree, one table” are used for describing three of the variants of RCRUD. These patterns are described by Keller [KEL97].

Known uses

1) We have used the first variant of RCRUD in the development of an industrial project which involves 139 classes, 42 with persistence in the database. The first versions of the application were developed using the `JDBC:ODBC` bridge with the Microsoft Access database. Afterwards, and due to the necessity of incorporating stored procedures, possibilities of auditing and more security restrictions, we selected the SQL Server 7 database running on Windows NT Server. Furthermore, as the program runs on several platforms (PC and Macintosh), we decided to use a direct connection to the database, with no use of the `JDBC:ODBC` bridge, through the AveConnect driver, which is 100% pure Java (http://www.avenir.net/products/aveconnect.htm).

   In both cases, RCRUD has operated fine. Only a little modifications on its code have been done, but due to the change of database, and not to the change of accessing way: as the `boolean` data type in Access is `bit` in SQL Server, instead of inserting or updating `true` or `false`, a conversion to the numbers 1 or 0 must be done. For example:

```
if (fieldClass.compareTo("class java.lang.Boolean")==0) {
   if (fieldValue.toLowerCase().compareTo("false")==0)
      SQL=SQL + "0,";
   else SQL=SQL + "1,";
} else
   ....
```

   The use of RCRUD to access members in runtime does not produce any appreciable delay: all the persistence methods are executed quickly; also the construction of objects from records (which is the operation with more access to metadata) operates very well.

2) The RCRUD class has been also used in other projects, as EasyTest, a single program to generate randomly and to correct test exams. EasyTest, including its source code, is available at http://www.inf-cr.uclm.es/www/mpolo/easytest

3) Ambler describes a persistence layer with some classes in charge of generating CRUD methods (see [AMB00], specially pages 9, 10 and 19). However, the proposed framework uses a wide set of classes that saves the needed information to

generate the methods in a persistent storage, instead of accessing dynamically to the object members.

4) Other authors are involved in projects or researches that use metadata, although no explicitly in the same context of RCRUD. Joe Yoder reports of some works at http://www.joeyoder.com/Research/metadata/
5) In any case, the use of Reflection and metadata are incipient themes that produce many interest in the scientific and practitioner community: in fact, also Joe Yoder reports on the celebration of several workshops related in some conferences, as ECOOP and OOPSLA.

3 Additional benefits of RCRUD and of the use of Reflection

In the industrial project described in the first point of the "Known uses", a screen was designed for each persistent class. The structure and behaviour of these screens is very similar (Figure 7). Due to this similarity, a generic, abstract screen can be designed, with quite all the structure functionality on it: common buttons, menus and several methods. We have called `FrameRCRUD` to this screen. It has a reference to a generic `RCRUD` object.

Figure 7. Structure of screens for persistent classes.

The response to the selection of the `New` button (or the `New` option in the menu bar) is to "empty" the screen to create a new object that, later, will be saved in the database. To empty the screen, the idea that appears more quickly is to declare an abstract method in `FrameRCRUD` and to implement it in the specializations; however, as a screen is a class and it is possible to access in runtime to the members of a class, we can write a method in `FrameRCRUD` that checks the type of each widget, writing the empty string if it is a `TextField` or `TextArea`, setting false as state if it is a `CheckBox`, and so on. The `Template` button has the

same behaviour than `New`, but in this case, the screen is not emptied. Both buttons call then to the `enable(boolean)` method, which allows the modification of the data in the screen.

Puerta et al. [PEGM94] and Konglathu [KON98] have studied the relationship between the Domain and Presentation layers more in depth, in order to automate the generation of user interfaces. However, the definition of both RCRUD and its corresponding `FrameRCRUD` allows to generate the user interface with less effort than in these works.

The structure of an application built through RCRUD and `FrameRCRUD` is shown in Figure 8. Bodies of its methods appear in Table 2.

```
private void template() {
    mOperation= kNEW;
    enable(true);
}
```

```
private void modify() {
  mObject.saveOldPks();
  mOperation=
      kMODIFY;
  enable(true);
}
```

```
private void exit() {
   dispose();
}
```

```
private void save() {
   enable(false);
   try {
    // loads mObject with the data in the widgets
     reload();
     if (mOperation==kNEW)
       mObject.Insert();
     else
       mObject.Update();
    }
    catch (Exception e) {
       Dialog d=new DialogError (this,
        "Error saving", true, e.toString());
       d.setVisible(true);
       enable(true);
   }
}
```

```
private void delete() {
 DialogConfirmation d=
    new DialogConfirmation
      (this, "Attention", true);
 d.setVisible(true);
 if (d.mOption==d.YES) {
    try {
       mObject.Delete();
       empty();
    }
    catch (Exception e) {
     Dialog d2=
       new DialogError
         (this, "Error",
          true,
          e.toString());
      d2.setVisible(true);
      d2.dispose();
    }
 }
 d.dispose();
}
```

Table 2. Body of methods in FrameYet.

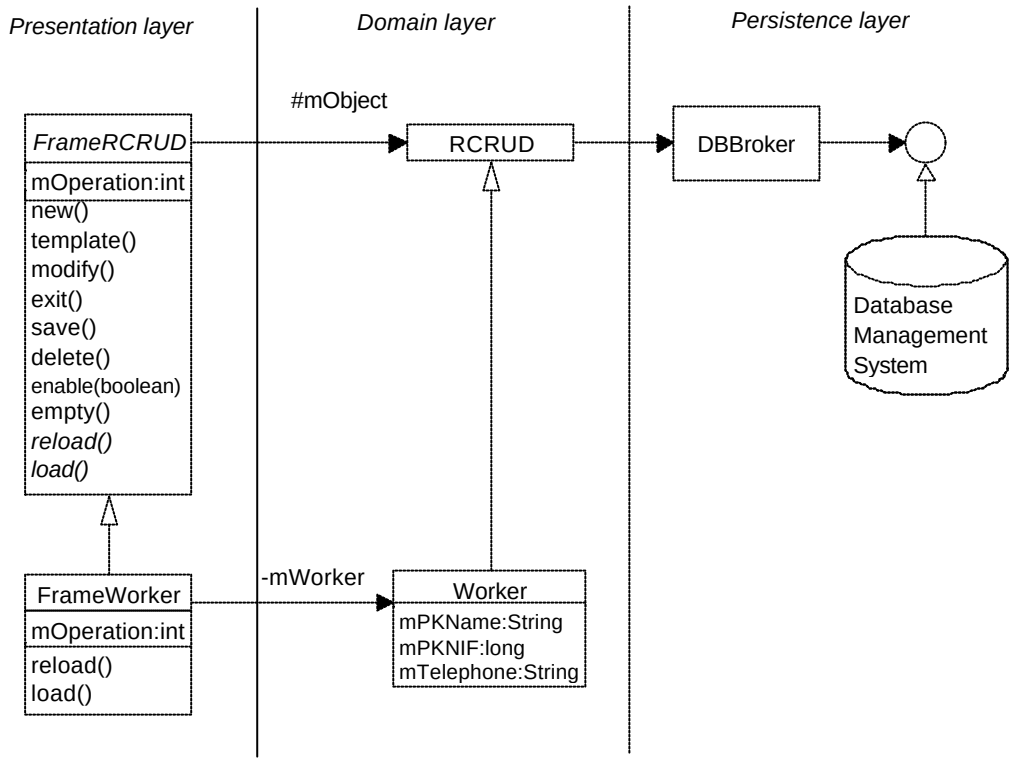

Figure 8. Structure of applications using RCRUD.

4 Conclusions and future trends

Systems developed with RCRUD produce persistent classes in a very easy way: to do persistent a domain class, the programmer only needs to write the fields of the class (taking into account some name conventions) and do it a RCRUD's child. None persistence method must be written in the class, since all of them are inherited from RCRUD. In this way, persistent classes have a high cohesion, since all their methods are related to the domain problem. The coupling added to the system is very low: every class has only the "filial" relationship with RCRUD, plus those relationships (associations and aggregations) related to the problem. Moreover, RCRUD is the only class which has knowledge of the database (in our examples and industrial applications, via a database broker). In this manner, Domain classes do not access to the persistence mechanisms.

As it is seen in Figure 8, the applications we have developed do not use observers to refresh the status of the presentation layer, since they are refreshed every time they get the focus. Currently, we are working on the incorporation of a generic observer to the RCRUD class.

Also, from the migration to the new database management system to use stored procedures, we saw as a possibility the creation and execution in runtime of stored procedures to do the persistent operations.

5 Acknowledgments

The authors want to thank Wolfgang Keller, our shepherd, for all the readings of the previous versions of this paper. Without him, this paper would never have been a pattern paper.

This work is part of the DOLMEN project (Distributed Objects, Languages, Methods and Environments), which is partially supported by FEDER with number TIC2000-1676-C06-06.

6 References

[AMB00] Ambler, S.W. *The Design of a Robust Persistence Layer*. Ronin International. Available at (April 6, 2001): http://www.ambysoft.com/persistenceLayer.pdf

[BRO96] Brown, K. and Whitenack, B. *Pattern Languages of Program Design, vol. 2*. Reading, MA: Addisson-Wesley.

[GHJV95] Gamma, E., Helm, R., Johnson, R. and Vlissides, J. *Design Patterns*. Reading, MA: Addisson-Wesley.

[IBM00] IBM. *Object to relational table mapping techniques with persistence*. Visual Age Developer Domain. Available at (April 6, 2001):

http://www7.software.ibm.com/vad.nsf/Data/Document3124

[KEL97]. Keller, W. *Mapping objects to tables. A pattern language*. Proceedings of the 1997 European Pattern Languages of Programming Conference, Irrsee, Germany.

[KON98] Konglathu, J.A. Automated generation of user interfaces. Available at (December 26, 2000): http://www.cs.unc.edu/~konglath/pvt/cb/

[LAR98] Larman, C. *Applying UML and Patterns*. Upper Saddle River, NJ: Prentice-Hall.

[PEGM94] Puerta, A.R., Eriksson, H., Gennari, J.H. and Musen, M.A. *Model-based automated generation of user interfaces*. Proceedings of the 12th National Conference on Artificial Intelligence, pp. 471-477. Seattle, WA, USA.

[YDJ98] Yoder, J.W., Johnson, R.E. and Wilson, Q.E. *Connecting Business Objects to Relational Databases*. Available at (April 6, 2001):

http://www.joeyoder.com/Research/objectmappings/Persista.pdf