Persistent Objects In A Relational Database

Holger Vogelsang, Uwe Brinkschulte,
Institute for Microcomputers and Automation
University of Karlsruhe
Haid-und-Neu-Str. 7
76131 Karlsruhe, Germany
Tel.: +49+721 6083898
Fax: +49+721 661732
email: {vogelsang | brinks}@ira.uka.de

Abstract

This paper describes the design and implementation of persistent objects for a given relational database system. This is the solution to a problem, raised in a larger application at our institute. To build a reusable man machine service for distributed systems within the object oriented language C++ there is a need to save and restore to contents of objects together with their relations. This is necessary to keep a prebuilt user interface in a database and to allow permanent runtime modifications. On the other hand old style applications using the man machine system must be able to use the same database or at least the same database server. The result is an easy to use object-oriented implementation of object persistence and persistent relations between objects for a relational database. Furthermore, this solution is usable in realtime requirements.

Keywords

Persistent objects, persistent object relations, realtime access, platform independence, relational database, object distribution, C++

1 Introduction

The engineering of computer based systems (ECBS) is in the research area of the "Institute for Microcomputers and Automation" (IMA), [Vog96]. To aid this engineering process, some basic services have been developed at the IMA. There are, for example, a service for general man machine interaction, a process database service and a service for measurement and control. Furthermore, object-oriented design principles are used to create the services within the programming language C++. This decision was made, because a good object-oriented design of data structures can decrease the development time of an application and increase the re-usability. E.g. in the above mentioned man machine service every component of the user interface is represented by an object. Their dependencies and links are expressed by relations between objects. The major problem that arises from such a design is, that there is no standard way to keep the contents of objects together with their relations persistent without giving up a clear object oriented design. Persistent objects are objects in a programming language, which are able to survive a program execution cycle. This means, that their content is not lost after a program crash or normal termination because it is saved in application defined periods of time in an external memory. To assure this functionality in traditional systems, often an application dependent store-and-load mechanism is used to write and restore the object's contents. This can be done by placing persistent objects into a special container object, which itself is responsible for the object management. This has one major disadvantage: persistent objects cannot be treated like other non-persistent objects. A much better approach is to request an object to save its own state or to restore it, because of the following main advantages:

- **Embedding**
  The access to persistent objects is identical to non-persistent objects. There are only additional functions to control the load/save mechanism.
- **Ease of use**
  The programmer can reuse this mechanism in every application without any change. The object interface is identical.
- **State dependence**
  The states of critical or other important objects must be accessible even after a program crash either to find the problem by examining the last state values or to reinitialize the program during a new-start. An object is able to request itself to save its own state after major or important changes.
- **Platform independence**
  A major problem in the above mentioned man machine service was the reuse of user interfaces on different hardware platforms like Sparc-Workstations and Intel-PC's with a variety of compilers. As a result, there was no guarantee that saved objects on one system are usable on other systems due to alignment and byte-order problems. This requires the availability of full type information at runtime, which could not be derived from RTTI in C++. Every object can provide the persistent object system with its own type description.
- **Realtime access**
  This solution allows realtime access to the database if the underlying operating system has realtime capabilities.

Section 2 introduces the concept of persistent objects and its representation in an object-oriented programming language (C++). This approach is extended in section 3 for persistent relations of objects. To physically store persistent objects, a relational database is useful. This is more secure, comfortable and flexible than storing such objects in files. Therefore the above mentioned process database service is used as a background storage. Section 4 introduces this service and shows the mapping between it and the object oriented design by applying stubobjects. Section 6 contains experiences with persistent objects in a large application and some ideas for future extensions. The last section finally shows some speed measures.

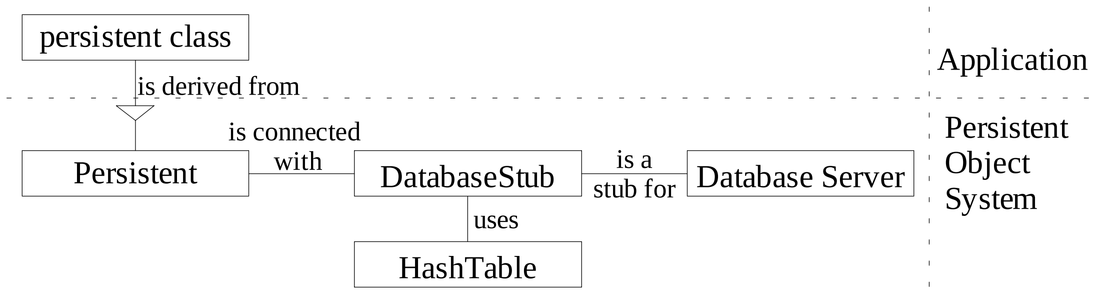
2 Concept of persistent objects

As mentioned above, the main goal in the introduction of persistent objects is a "natural" embedding into a given object-oriented programming language (here C++). The intention is to hide most of the additional functionality from the programmer by applying a clear object-oriented design. Persistent objects should be usable like any other non-persistent object.

2.1 Characteristics and design

Persistent objects are realized by declaring the corresponding class as persistent. This is done by deriving these classes from the internal class "Persistent" on the applications side. On the systems internal side every persistent object is connected to one (of several possible) database objects. This acts as a stub for the real local or remote database server in a heterogeneous net of computers. The connection to a stub is dynamically changeable so that an object's content can be loaded from or stored to different databases. The advantage of this design is that a persistent object can be placed anywhere on a net. No object has to know its own storage place.

Only the reference to the local stub is kept. All database stubs share the same machine-dependent hash table for two purposes: First, to determine whether an object is already in memory and second, to find the memory address for a given object identification. The following picture shows the internal structure together with the application interface. The diagram uses the OMT notation for data structures.



Picture 1: Internal system structure

The following two attributes are characteristics of persistent objects:

- **Unique identification**
  The first problem in persistence that has to be solved is the unambiguous identification of objects in external and internal memory, because internal memory addresses can differ in each program execution cycle. As a solution, every persistent object is provided with a unique identification code. This is either granted by the internal system during the first dynamical creation of the object or given out by the programmer for static objects. This second mechanism for static objects is necessary to allow the storage of an object's reference in the program code.
- **Runtime type information**
  The next problem is the desired platform independence for persistent objects. Since the selected programming language C++ does not provide full runtime type information, a more mighty mechanism has to be created. A persistent class has an internal method called "Metatype" that is first internally called by the object itself to specify its own type information and second used by other objects to determine the metatype of this object. This method uses a local metatype server to store its type information. First, it creates a new metatype with its own class-name. Within the next steps it describes each class component using
  - its name,
  - its position relative to the object together with its size and
  - its type.

  "Metatype" is automatically created for persistent classes by a precompiler to reduce the programming overhead and avoid errors. If the persistence is limited to one hardware platform, the type information can be omitted.

The database stub uses this method to get the full type information for a persistent object when the object is stored or loaded. Therefore the object can be stored in a unique database format.

2.2 Programming interface in C++

The described approach for persistent objects is totally embedded into C++. As written above, every persistent class is derived from the internal class "Persistent". Its constructor is provided with

- the type-id of the corresponding class,
- an optional identification number,
- and a reference to a database stub.

The type-id is necessary to enable the object system to access the metatype information. If no identification is given during the first creation, a unique one will be determined. The connection with a database server is managed through the referenced stub object, which has to be created first. The behavior of persistent objects is controllable through the interface of the class "Persistent":

```
class Persistent {
public:
        Persistent( PerId id, Meta mt, DatabaseStub *stub = NULL );
        Persistent( Meta mt, DatabaseStub *stub = NULL );
        virtual      ~Persistent();
        PerError     Update();
        PerError     Undo();
        PerError     Delete();
        PerId        Id();
        boolean      Created();
};
```

- **Persistent**
  The first constructor is used to create a new persistent object, its identification code is determined automatically. The second one creates a local copy of an existing object and reloads its contents or creates a new one with the given id, if no object with this id exists. It is possible to omit a stub, if one of the stubs is marked as default.
- **Update**
  "Update" writes the object's contents into the database, using the local stub to convert the object into a unique database format.
- **Undo**
  "Undo" overwrites the object's contents with the value in the database.
- **Delete**
  "Delete" removes the object only from the database, but not from the memory. The application of "Undo" and "Update" on this object is disabled after a removal.
- **Id**
  "Id" returns the unique identification code.
- **Created**
  "Created" is used to determine whether this object is newly created or loaded from the database. This method is useful in a constructor of a persistent class to do some initializations only for newly created objects.
- **~Persistent**
  It removes the object only from main memory and not from the database.

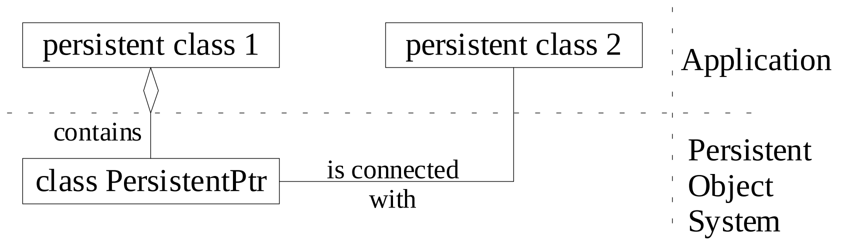
A detailed example is given in chapter 5.

3 Relations between persistent objects

In most object-oriented applications it is necessary not only to keep objects persistent, but to hold the relations between objects. A very simple example is a doubly-linked list of persistent objects, which has to be saved as a totality. Relations between objects sometimes lead to very large data structures, of which not every part is used in any situation. To ensure an efficient main memory utilization a mechanism for a dynamic load and preemption of structure components should be available. An example is the above mentioned man machine service: It holds only the visible user interfaces in main memory.

3.1 Characteristics and design

Relations between non-persistent objects are normally realized by pointers. The main idea behind the construction of persistent relations is the introduction of a special pointer class, called "PersistentPtr". In addition to the pointer, which is only valid in main memory, it contains the destination object's identification code too. Therefore this pointer class can be used only for persistent objects. The next picture shows the class structure in OMT notation for persistent pointers.



Picture 2: Persistent pointer

The use of a real pointer together with the identification number is important for an additional facility: The mechanism for a dynamical reload and preemption of persistent objects needs to know whether the referred object is in main memory, only in database or not present (NULL).
It is important to know this, because the mechanism is completely under the control of the programmer. He has to request a persistent pointer to reload the referred object or to preempt it. This could be a very complex task for large data structures, so that there is a slightly modified pointers class, called "AutoPersistentPtr". An object of this class reloads the referred object during its own (re-)creation. This implies, that data structures, connected by this pointer, are reloading themselves. Only some kinds of base or root objects have to be reloaded on demand.
The following sequence of actions starts after reloading the root object:

1. If the loaded object contains an "AutoPersistentPtr" as a component, this object is created by the language's runtime control and initialized by the persistent object system.
2. The constructor of the created pointer object recreates the referred object if it exists. The sequence continues at step 1 until there is no referred object left.

To stop this mechanism, a large data structure should be divided into self loading sub-structures, which are connected together by "PersistentPtr" as a breakpoint. There are two major strategy rules for large persistent data structures:

1. Objects, which are used together, should be kept in self-loading structures.
2. Breakpoints separate such groups to keep the memory utilization small.

3.2 Multiple references

Introducing relations between persistent objects leads to one problem: How are multiple references to one object handled ? Assume, there are two objects A and B referring an object C. What happens when both of them are reloading or preempting C ? The first creation attempt restores C in main memory. All other creation-accesses only determine the memory address of C and return it. C is not multiply loaded. This requires a reference counter, which is used to free an object only if no references by PersistentPtr exist. This is not a common solution since there can be made a copy of a PersistentPtr without informing the persistent object system.

3.3 Programming interface in C++

"PersistentPtr" is implemented as a template class. The template is instantiated with the type of the referred object.

```
template <class type>
class PersistentPtr{
        type            *ptr;
        long            id;
public:
        PersistentPtr( type *p = NULL );
        ~PersistentPtr();

        void            Preempt();
        PerError        Reload();
        PerId           Id();
        type            *Ptr();
        void            Set( type *pp, PerId new_id );
};
```

Of course, most of the functionality is hidden by overloaded operators, which are not shown here. There are no syntactical differences between operations on standard and persistent pointers. Only the control of reloading and preemption has to be done by explicit method calls.

- **Preempt**
  "Preempt" removes the referred object from main memory.
- **Reload**
  "Reload" either creates the referred object in main memory and read its content from the database or updates the internal pointer if the object is already in memory (see section 3.2, Multiple references).
- **Id, Ptr**
  These methods return the identification or the main memory address of the referred object.
- **Set**
  "Set" resets the persistent pointer to an other object.

"AutoPersistentPtr" is derived from "PersistentPtr" with one extension: The constructor calls "Reload" to re-read the referred object after its own creation. A reload is not necessary and therefore ignored when calling the copy-constructor.

4 Realization using a relational database

To store the persistent objects and their relations introduced in chapters 2 and 3, a database system is used. This provides more security, easy and comfortable programming, access control, flexibility for future extension, etc. than storing these objects in simple files. As mentioned in the introduction, one of the basic services developed at IMA to aid systems engineering is a process database service. This service, called "Merlin", is used in the present application.

4.1 Process database "Merlin"

The process database service "Merlin" is a general service for data management. It was designed to meet the requirements of modern information and automation systems. Its main properties are:

- an easy-to-use but powerful interface to the application program
- cooperative and distributed network database management
- real-time system operation
- 24 hour on-line availability
- configurable data security
- a small amount of necessary systems resources
- portability

For data storage, Merlin uses relational data structures. As a special feature, large unformated objects up to 4 Gbyte size can be stored as part of the database relations. Merlin offers an easy set oriented interface containing operations for very fast data access and manipulation, data security and data control. This interface is embedded in the language C/C++. Using multiple client server architectures, database operations can be distributed in a heterogeneous network. This distribution is mostly hidden to the application program. To provide platform independence and portability, Merlin can operate on a variety of hardware and software platforms. These attributes make Merlin suitable for storing the persistent objects.
A more detailed description of "Merlin", its aims and features can be found in [Bri93],[Bri94] and [Bri95].

4.2 Mapping between persistent objects and the database: The stub object

As mentioned above, the internal stub objects are the interface between the persistent objects and the database service. A stub provides a very small set of methods, accessed only by the "Persistent" class. There is no other access during the program execution cycle except for a user-controlled creation and destruction of these objects. This has to be done explicitly to allow a dynamic reconnection with different database services.

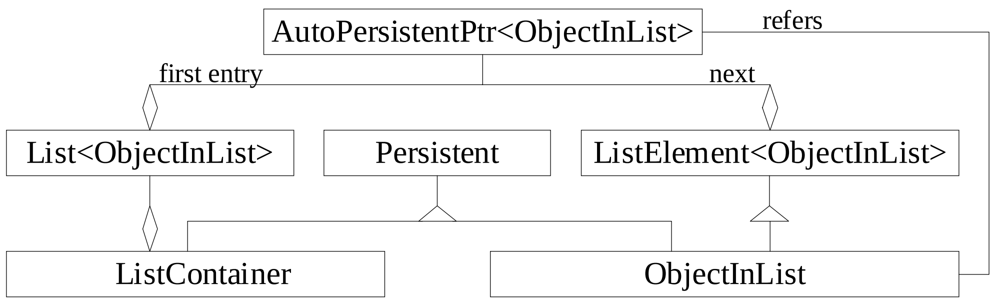
```
class DatabaseStub {
public:
        DatabaseStub( const char *db_name, const char *server, boolean default_stub = FALSE );
        ~DatabaseStub();
        PerError    Update( PerId id, void *object, Meta mt );
        PerError    Undo( PerId id, void *object, Meta mt );
        PerError    Delete( PerId id );
        PerId       Reload( PerId id, void *object, Meta mt );
        void        *Ptr( PerId id );
        PerId       NextFreeId();
};
```

These methods of the stub, which deal with loading and updating of objects, are provided with the metatypes of the persistent objects. This allows the conversion of the objects into a unique database format. The stub will not be discussed in detail because its functionality is completely hidden from the user. This section was only intended to show the interface to the database.

5 An example

After the introduction of persistent objects and persistent relations between them, a simple example in this section shows the application in a program. For this purpose, a singly-linked-list is used as a basic data structure It is neither a complete implementation nor does it have a practical use. This example only demonstrates the most important points.
The list "List" concatenates objects of the same persistent class "ObjectInList", which is derived from the class "ListElement", containing the reference to the next object in the list. The

list deals with normal (non-persistent) pointers at its interface in order to simplify the application. To keep this example small, there are no operators overloaded. First, the resulting class relations are shown, using the OMT notation.



Picture 3: Single-Linked-List

```
template <class type>
class ListElement{
        AutoPersistentPtr<type>  next;
public:
        ListElement(){}
        ~ListElement(){}

        void      Set( AutoPersistentPtr<type> &next_el ){ next = next_el; }
        type      *Next(){ return( next.Ptr()); }
        Meta      Metatype(){ ... }
};
```

The list-class is a template too.

```
template <class type>
class List {
        AutoPersistentPtr<type>  first;
public:
        List() : Persistent( Metatype() ){}
        List( PerId id ): Persistent( Metatype(), id ){}
        ~List(){ PreemptAll(); }

        void      InsertFirst( type *obj ){ obj->Set( first ); first = obj; }
        type      *First(){ return( first.Ptr()); }
        type      *Next( type *obj ){ return( obj->Next()); }
        void      PreemptAll(){ .... }
        void      UpdateAll(){ ... }
        Meta      Metatype(){ ... }
};
```

The application of this singly-linked-list is easy: First, the class of the concatenated objects is defined:

```
class ObjectInList, public Persistent, public ListElement<ObjectInList>{
        ... some variables ...
public:
        ... some methods ...
};
```

The next piece of code shows a class, containing a list-object. Objects of this class are persistent too.

```
    class ListContainer, public Persistent{
            List<ObjectInList>          list;
    public:
            List( PerId id ) : Persistent( id, Metatype() ){}
            ~List(){ list.UpdateAll(); Update(); }
            void      Insert( ObjectInList *oil ){ list.InsertFirst( oil ); }
            Meta      Metatype(){ ... }
            ... some other methods ...
};
```

6 Experiences and further extensions

The described version of a persistent object system was introduced as a general solution for a special problem, the man machine service. But the concept allows further extensions to build a more flexible system for other kinds of applications. These extensions were not necessary until now, so there is no realization yet.

- **Synchronization for distributed applications**
  In distributed applications, sharing a common set of persistent objects, it is important to introduce a synchronization scheme for a coordinated access. Very useful in this context are rights, defined on objects, to permit or deny an access. The realization is simple, because the underlying database service has a built-in facility for this purpose.
- **Object selection**
  In some applications it could be useful to reload only these objects, which fulfill given selection conditions. For example, an object component has a special value.
- **Precompiler**
  The precompiler for an automatic generation of metatype information is under construction.

Although these extensions are not available yet, persistent objects are extremely useful in object-oriented programming. The reuse of existing, persistent data structures for other projects is very easy. Much development and debugging time can be saved. This was recognized in other applications at the institute, using persistent objects.

7 Speed measurements

This last section contains some speed measurements on different hardware platforms. More practical measurements can be found in [Kes96]. This approach is used in the knowledge representation system C3L++ to make some speed comparisons with a commercial system, POET.

	Hardware Platform	Memory	Compiler	Environment
1	Sparc 10	32 MB	GNU-C 2.7.2	Sun-OS 4.1.3 + X11R6
2	PC Pentium, 133 MHz	16 MB	GNU-C 2.6.3 (DJGPP)	DOS 6.22 + Smartdrv (2MB)
3	PC Pentium, 133 MHz	16 MB	GNU-C 2.7.2 (ELF)	LinuX 1.3.60 without X11
4	PC Pentium, 133 MHz	16 MB	Borland C++ 4.5	Win 3.11 with Win32s

The tests are made with 50000 operations on different objects. The measurements are given in msec per object and operations per second, where the following operations are tested:

	Operation	Description
1	Create	create new persistent objects, which are not in the database
2	1st Update	write objects' contents to database (after a newly creation)
3	Other Updates	write objects contents to database (object exists in database)
4	Undo	read contents of already-created objects back from database
5	Remove	delete persistent objects only in main memory, not in the database
6	Reload	create objects in main memory, all objects exist in database
7	Find	find object references with given object handle
8	Delete	delete objects in main memory and database

	GNU-C (DOS)		GNU-C (LinuX)		GNU-C (Sparc/X11)		BC 4.5 (Win32)	
	msec/Op	Op/sec	msec/Op	Op/sec	msec/Op	Op/sec	msec/Op	Op/sec
Create	0.048	20800	0.040	24750	0.091	10980	0.279	3600
1st Update	0.667	1500	0.673	1490	1.147	870	0.952	1050
Other Updates	0.533	1870	0.439	2280	0.500	2000	0.719	1390
Undo	0.393	2550	0.235	4250	0.515	1940	0.455	2200
Remove	0.004	227240	0.006	177110	0.086	11680	0.009	113900
Reload	0.347	2880	0.208	4820	0.513	1950	0.418	2390
Find	0.001	≈1000000	0.001	777380	0.005	197080	0.001	909090
Delete	0.480	2080	0.346	2890	2.747	360	0.556	1800

8 Literature

[Ban94] Oliver Bantleon, Ulrich Eisenecker, Georg Missiakas, *"Streams++: Portable Bibliothek für persistente Objekte in C++"*, iX 3-4/1994

[Bri93] U. Brinkschulte, *"MERLIN - Ein Prozeßdatenhaltungssystem für Echtzeitanwendungen"*, Conference Proceedings, Echtzeit 93, Karlsruhe, 1993

[Bri94] U. Brinkschulte, *"Architektur eines Datenhaltungsdienstes"*, Conference Proceedings, 39. Wissenschaftliches Kolloqium, TU Illmenau, September 1994

[Bri95] U. Brinkschulte, *"Database Services"*, Conference Proceedings, KEOOA 95, Knowledge Engineering and Object Oriented Automation Workshop, Strasbourg, May 1995

[Dea90] Alan Dearle, Rex di Bona, James Farrow, Frans Henskens, Anders Lindström, John Rosenberg, Francis Vaughan, *"Grasshopper: An othogonally persistent operating system"*, Department of Compute Science, University of Adelaide and Sydney

[Kes96] T. Kessel, M. Schlick, H.-M. Speiser, U.Brinkschulte, H. Vogelsang, *"Implementing a Description Logics System on Top of an Object-Oriented Database System"*, Proc. of the International KRDB-Workshop at the ECAI'96 Conference, Budapest, Hungary, August 1996

[Sin92] Vivek Singhal, Sheedal V. Kakkad, Paul R. Wilson, *"Texas: An Efficient, Portable Persistent Store"*, Proc. of the Fifth International Workshop on Persistent Object Systems, San Miniato, Italy, September 1992

[Ste92] Al Stevens, *"Persistent Objects in C++"*, Dr. Dobb's Journal, December 1992

[Vog96] Holger Vogelsang, Uwe Brinkschulte, Marios Siormanolakis, *"Archiving System States by Persistent Objects"*, IEEE conference on ECBS'96, Friedrichshafen, Germany, 1996