Persistence: Implementing Objects over a Relational Database

Version 1.0

Ratio Group Ltd.
17/19 The Broadway
Ealing W5 3NH

Email: info@ratio.co.uk
Web: www.ratio.co.uk

We Know the Object

Table of Contents

1. Introduction

In virtually every business applications the lifetime of the business data is longer than that of the execution time of programs that manipulate it.. This implies that the data need to be stored between successive runs of the program. In Object Orientated Programming (OOP) the process of storing and retrieving objects (or more accurately their attributes) is called persistence. More importantly than this, many of these applications also require concurrent multi-user access to this data. Historically, such systems have been implemented using relational database management systems (RDBMS). The rise in popularity of object oriented programming languages (C++, Java, etc.) has lead to a new generation of systems which must solve the “impedance mismatch” between an object view of the world and the corresponding relational model of the same world and allow objects in an application are stored and made available to other users via an underlying RDBMS.

This paper discusses approaches to storing persistent objects in relational databases in a financial settlement system, to replace an existing legacy system. Designed to be both flexible and high performance, the system was developed over a period of 3 years using a component-based approach and object-oriented techniques. The system comprised over 1,000,000 lines of C++ source code (of which 75% was machine generated), divided into 1,500 classes spread over 30 component subsystems.

2. Overview of the object/relational interfacing problem

2.1. Background to object models

Over the last few years object oriented programming (OOP) using languages such as C++ and Java has become a mainstream activity. As a result of this, there has been an increase in the use of object oriented analysis and design techniques (OOA/D) using notations such as the Unified Modelling Language (UML). UML is a set of notations that enable the design of an object-oriented application to be expressed in graphical form. Central to the modelling notations of UML is the class model (more casually referred to as an object model).

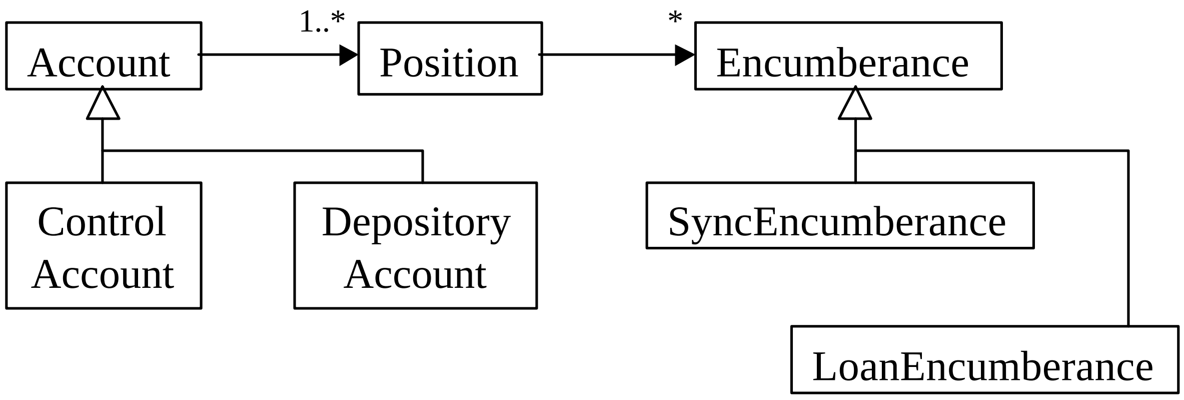

Figure 1 - Example object model

Object models show the structure and relationship of the classes that will make up a system. The example shown above (which is used throughout this paper) shows a number of object modelling constructs:

- inheritance, a key concept of OO programming languages, is shown using a line between two classes with a open triangular arrowhead at the parent class's end. In the example above the Account class has two distinct sub-types (Control Account and Depository Account) which can be used to specialise account behaviour.
- associations represent the fact that there is a conceptual or (in implementation terms) physical link between two classes and are represented by a line between the two classes concerned. In the example above an Account class has at least one associated Position with no specified upper bound on the number of Positions.

2.2. Why use a Relational Database to store Objects?

If the process of persisting objects into a relational database is so hard, why bother at all? Using a "native" object orientated database management system (OODBMS) would certainly remove the problem. However, there are certain obstacles to taking this route:

- In many cases a corporate (relational) data-model is already in place, and is the basis of interoperability between existing applications. Unless these are all to be redeveloped using a common object model and can be ported to a common OODBMS then co-existance with the relational world is going to be needed.
- Many enterprises have made strategic directions on core system software (e.g. Operating System, middleware, DBMS) which an application must comply with. In many cases this mandates an RDBMS (in this case Oracle)
- In applications where performance and scalability are vital, OODBMS have yet to demonstrate the same track records as RDBMS.

2.3. Background to relational databases

Underlying all relational database management systems are a number of common concepts:

- Data is stored in tables within the database, each row of the table constituting a separate record, and each column in the table storing a particular field in the general record structure. Tables are stored in a recoverable manner, usually on magnetic media.
- Relationships between tables are stored using keys. A primary key is a unique identifier constructed such that no two records within a table can have the same key value. A foreign key is a reference to the primary key of another table. Records in one table can therefore be 'linked' to another table using the foreign key mechanism.
- Locking is a database concept used by developers to ensure that multiple users cannot update a related group of data items at the same time. Without locking it would be possible for the data within an application to become hopelessly confused and mismatched, or to put it more formally, to loose its integrity. Locking works by ensuring only single user access to programmer defined regions of the database at any one time.
- A transaction is a database concept that, under programmer control, enables a number of operations (reads or updates) on a database to be grouped together in such a way that either all the operations are completed, or none of them are. Transactions are particularly important in ensuring data integrity, whereby a number of operations on a database are grouped together to ensure they either all succeed or all fail. Without the ability to 'roll-back' all of the changes made within a failing transaction, it would not be possible to maintain the integrity of the data within the database.

2.4. The object relational interfacing problem

At the simplest level, the data to be retrieved or saved has to be determined within the persistence layer using an SQL statement. Objects are loaded using an SQL SELECT statement to retrieve the object’s data attributes. Objects are saved by extracting their persistent attributes and using an SQL INSERT or an UPDATE statement.

The problem is how to store objects using tables, rows and columns, and how to map the multi-user requirements of the object application to the multi-user features of the underlying relational database in a clean way that maintains the OO structure of the application, that minimises the effort required of the application developer, and to maintain an appropriate level of performance.

This leads us to a number of more specific questions:

- How can classes in an OO programming language be mapped onto tables within a relational database? In particular, how should OO inheritance structures be represented using tables, which have no corresponding concept?
- How should associations in an object model be mapped onto the relational database?
- How and when should objects be loaded into memory, and where should this responsibility lie (with application specific code, with re-usable infrastructure code, etc.)?
- How should transactions be handled from the OO programming language, and where should responsibility for transaction handling lie?
- How should locking be handled from the OO programming language, and where should responsibility for locking lie?
- What steps can be taken to enhance the performance of the application, and where should the responsibility for these steps lie?

These questions are addressed in this paper.

3. Retrieving and manipulating persistent objects

3.1. System Architecture

Before we start our discussion of persistence in detail, it is useful to look at the overall architecture of the system under discussion. The following diagram shows the architectural layering of this system:

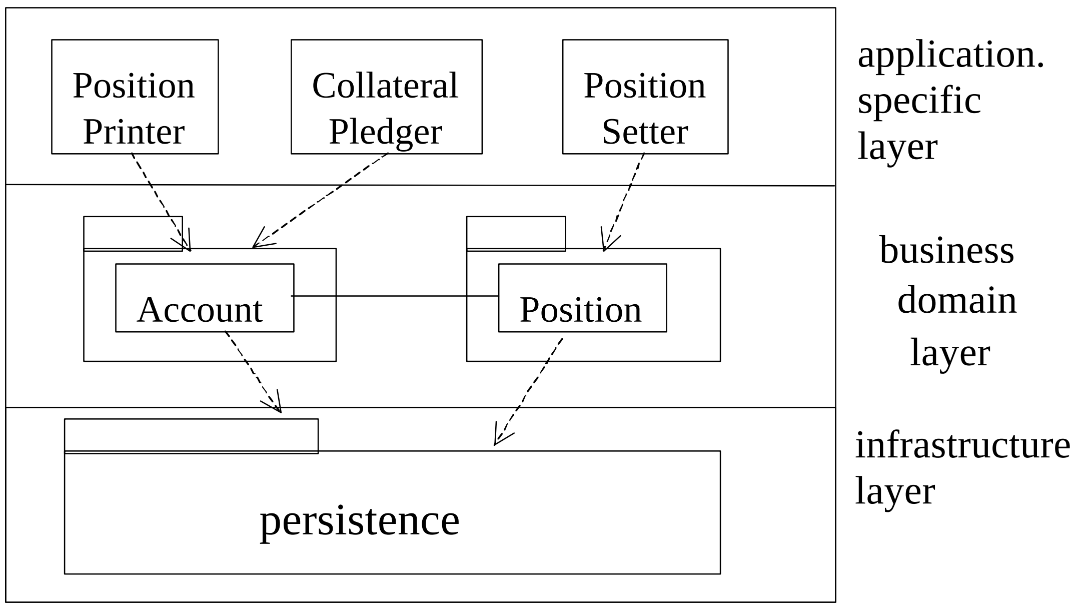

Figure 2 - Overall architecture of system under discussion

The three layers shown on this diagram are:

- the **application** specific layer (otherwise known as the use case layer) contains non-persistent business process objects that drive the application. They implement the steps of use cases (user requirements).
- the business **domain** component layer. The system under discussion in this paper uses a component-based approach, whereby key business objects are captured as components within the system. At this level components collaborate via their interfaces, and implementation details remain hidden from the component user.
- the **infrastructure** layer is a layer of horizontal (i.e. non-application specific) infrastructure which implements generic re-usable mechanisms for, amongst other things, interfacing between the object and relational aspects of the system.

(for further discussion of the layering presented here, see the paper entitled: “An Architectural Reference Model for CBD” at www.ratio.co.uk/techlibrary.html)

Separate packages (component subsystems) are shown within the appropriate layer.

3.2. Illustrative example

To illustrate the points made in this paper, we are going to examine the following fragment of the object model introduced in section 2.1, and examine the requirement that an account print all its associated positions:

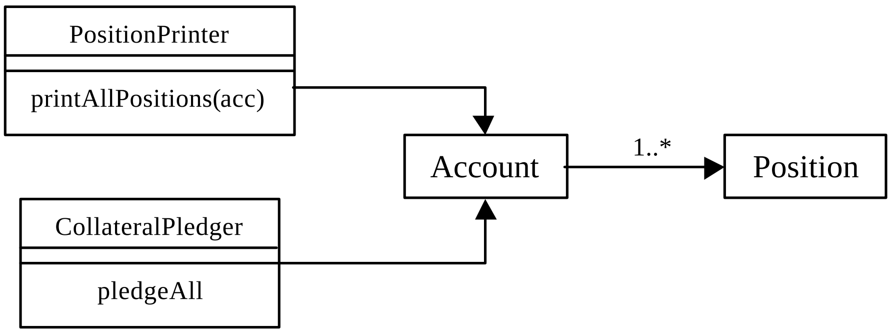

Figure 3 - Fragment of model used as example in this paper

Two 'use case' objects are shown: a PositionPrinter that prints all of the positions for a given Account, and a CollateralPledger object, that pledges all the positions of an account as collateral for a loan. The two 'business' objects shown, Account and Position, form the interface to their corresponding component implementations (see section 3.1).

Following is a sequence diagram showing the collaborations between the high-level objects necessary to implement the 'printAllPositions' use case:

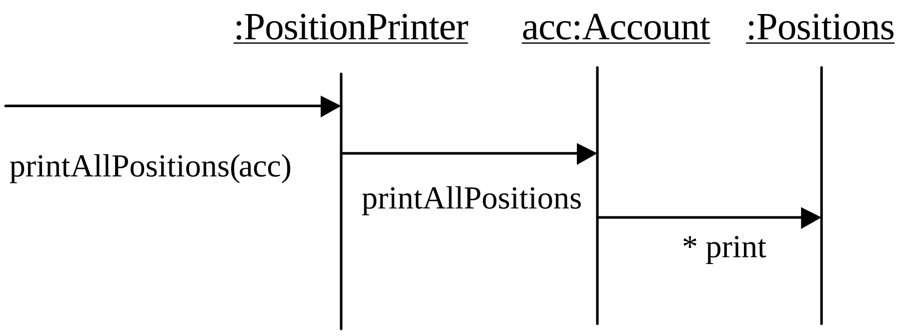

Figure 4 - Sequence diagram to print all Positions associated with a named Account

When it receives the printAllPositions message, the Account loops through all of its associated positions and calls the print method for each of them in turn - a trivial piece of design. However, there is a lot more going on 'under the bonnet' than shown here. As we delve deeper into this, the power of hiding implementation details through good abstraction will become apparent.

3.3. Implementing associations using collections and smart pointers

It is appropriate that business level sequence diagrams ignore implementation details such as how associations are implemented. Such details are, however, important when considering persistence, so we shall look at them further here.

Superficially at least, implementing associations is not difficult. All that is required is a mechanism by which one object may identify any others to which it is related. When all objects are in memory, this is as simple as storing pointers.

One factor which complicates this picture is that it ignores the fact that objects within an object model are inherently shared - in the example given it is possible for two use case objects to point to the *same* Account object. As we don't want two copies of the same Account to be in memory at once (this would cause problems updating the account) any pointers used must point to the *same* Account object in memory.

This in turn presents us with a memory management problem (in non-garbage collected languages at least) - how do we know when to delete the memory associated with an Account, and who can accept responsibility for this? To put this responsibility on the application programmer would be to introduce a major overhead in using pointers at the application level, requiring every application developer to keep track of how many pointers there were to any particular object at any particular point in time.

Fortunately, a common approach to sharing objects and the associated memory management problems in C++ comes to the rescue. It is to use 'smart pointers' [Meyers]. Smart pointers are objects that appear externally to behave like exactly like ordinary pointers, but which internally use a reference counting mechanism to note that the object being pointed to is shared. Thus, if two smart pointers point to the same object the internal reference count of the pointer will be set to two.

The smart pointer itself takes on responsibility for memory management, deleting memory only when the reference count to the shared object drops to zero.

The object model for a smart pointer to an object of type T looks like this:

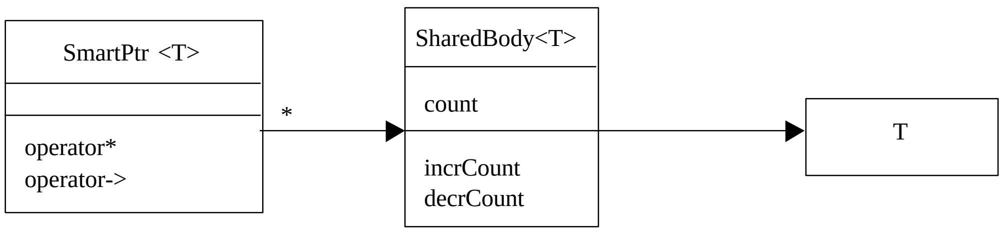

Figure 5 - Implementation of reference counting smart pointer

The smart pointer shown here redefines the standard C++ pointer dereferencing operations (operator* and operator->) and hence provides the illusion to the business object developer that it is no different to any other ordinary pointer, providing an easy to understand abstraction hiding unnecessary detail.

Whenever a smart pointer is created (or destroyed) the count in the middle object is incremented (or decremented) by one using the incrCount() (or decrCount()) methods. When this count falls to zero, and there are no more smart pointers referring to this object, then it is safe to remove the shared object from memory.

One potential problem with this type of smart pointer, however, is cyclic pointing. If a smart pointer points to an object which in turn points back to the first object then the reference counts of these pointers can never fall to zero, and the memory associated with them will never be freed. Although it is possible to break such cycles, it is best to avoid this problem by careful design (following the acyclic dependency principle). This was the approach taken in the system discussed in this paper.

Coming back to the example in figure 3, we can see that any particular Account will have one or more Position objects associated with it. One to many associations require that a variable number of objects be 'pointed to.' To achieve this it is necessary to use some form of collection (vector, list, etc.) that can hold (smart) pointers to any number of objects. Drilling down to show the next level of detail, the model fragment shown in figure 3 becomes:

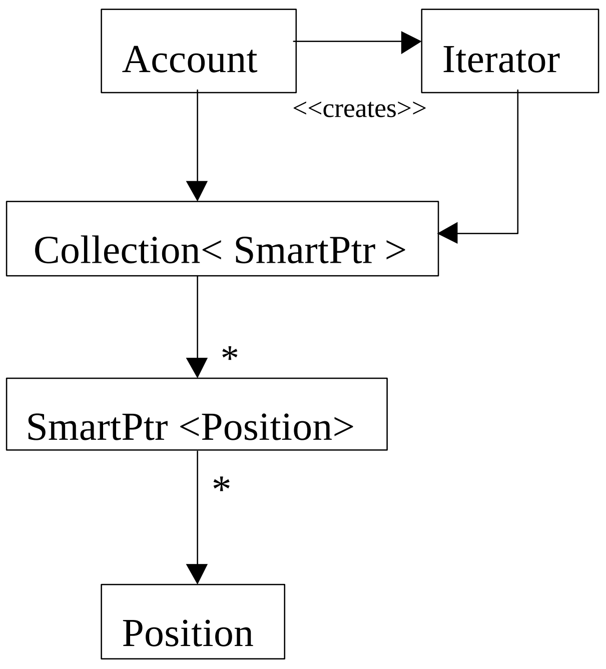

Figure 6 - Implementation of reference counting smart pointer

At this level of detail (abstraction) the sequence of events for printAllPositions is modified to get a collection of positions and process them each in turn using an Iterator. This is discussed further in the next section of this paper. As Positions may be shared between Accounts we are using smart pointers to refer to them (note that for brevity the intermediate SharedBody object of the smart pointer has been elided and the names have been shortened).

3.4. Accessing collections using iterators

An iterator is an object that is used to control sequential access to a collection. At the most basic level, an iterator will contain operations for testing to see if the sequence is finished, accessing the current item, and moving to the next item in the sequence. Iterators enable us to hide the internal implementation details of the collection we are using, and to have a consistent interface for accessing all types of collection.

In our example, the Account object creates an iterator over the collection of smart pointers to Positions, and then accesses each member inside a loop, asking it to print itself. This is illustrated in the following diagram:

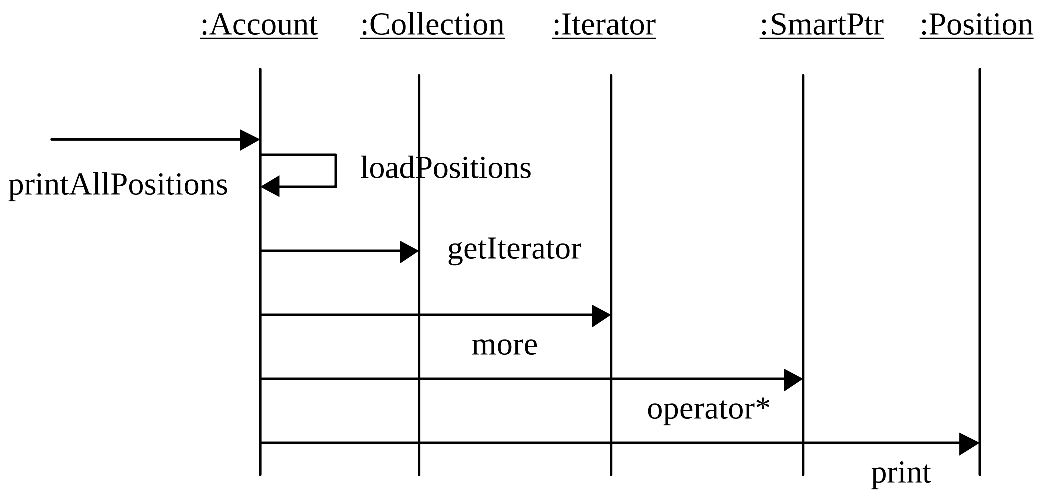

Figure 7 - Using an iterator to access a collection of smart pointers

3.5. Querying to load objects from the database

Our discussion so far has made the assumption that all the objects we are dealing with are in memory already, and has thus neatly avoided the issue of how and when to load objects from an underlying database into memory. When the Account object in figure 7 receives a request to printAllPositions, it must first ensure that the right collection of Positions is in memory. To do this, it may have to request that these be loaded from the underlying database.

Simplistically, one might be tempted to write:

```
SQLQuery query = "SELECT POSN_ID, POSN FROM POSITION WHERE POSN > 200
AND ACCOUNTID = 3343";
ResultSet = database.Execute(query);
```

This approach exposes the column and table names used in the underlying database to the application layer. This is undesirable as it does not present information to the application layer at the appropriate level of abstraction, and is clearly leaking implementation details of a lower layer to a higher one. This means changes to the internals of the lower layer (e.g. table names) would require changes to the higher layer as well.

Another approach would be to write something like this in the application layer:

```
ResultSet = database.FindPositionsGreaterThan(200,3343);
```

Although this maintains the strict layering of the system it pushes application specific code into the general persistence layer. By doing this we make the persistence layer less reusable and more susceptible to modification because of changing user requirements. Neither of these effects is desirable.

The solution to this dilemma is to specify queries using higher level concepts and to have the software translate them into lower level concepts when required. The project under discussion achieved this using a C++ specific mechanism: overloaded operators. Each query is specified at the business object level using attributes of the objects in question, and these are then translated into SQL code at run-time by the persistence layer. For example, the previous query would be written like this:

```
SQLExpr where = PositionQuery::_position > 200 &&
                PositionsQuery::_accountID == 3349;
PsColPtr<Position> resultSet(where);
```

The first line creates an SQL expression at run-time by overloading the ">" and "&&" operators. This query is then executed in the second line to produce the collection resultSet, which could then be used via an iterator to process the results one position at a time.

3.6. Implementing queries using partial lazy loading

To understand how queries are used let us examine the loadPositions method of Account. When loadPositions is called on an Account, the following steps are carried out:

- the Account creates a Query object specifying the Positions it is interested in;

- the Account then creates a Collection (of smart pointers to Positions), passing in the Query object to indicate which Positions will be required;
- the Collection then asks the Query object to translate itself into SQL, to enable the relational database to be interrogated, which then returns the results of the query in an intermediate data structure called a RowSet;
- the RowSet is then attached to the Collection, for later examination;
- the Collection creates as many smart pointers to Positions as it requires to point to the results of the query;
- finally, an in memory cache (of which there will be further discussion later) is checked to see if the objects referred to in the RowSet are already in memory. If they are, the smart pointers are set to point to the cached copy, if not, the smart pointers are given the appropriate information from the RowSet, for later construction of the object (partial lazy loading).

Note that this sequence of events takes place only if the Account does not already have its collection of Positions loaded, and is an example of bulk loading of data into memory (the mechanism for loading a single object is somewhat simpler - see section 3.10).

The printAllPositions method on Account will then iterate over the Collection:

- the iterator gets the next element in the collection (a smart pointer to a Position), and returns it to the printAllPositions method,
- the smart pointer is then dereferenced using operator*, at which point the Position object will be constructed if the smart pointer does not already point to a cached copy. Any newly constructed object will then be put into the cache.

A major point to note here is that the functionality of both the smart pointers and the collections have now becoming integrally entwined with the persistence mechanism.

3.7. Loading 'second level' objects

The loading process described above loads in only the first level of requested objects (i.e. if we loaded a collection of Accounts, we wouldn't load the associated Positions). Often it is desirable to load in other associated objects too since they may be used soon afterwards. This is effectively a form of read-ahead optimisation. The project under discussion implements this optimisation by having a function called LoadEmbeddedObjects that is called whenever an object is loaded. By default this function does nothing, but this behaviour can be overridden on a per class basis to perform whatever loading the developer sees fit. This is an example of the use of the Template Method pattern (see [GoF]).

3.8. More on caching

The process of loading data from disk is optimised by use of an in-memory cache. A cache is a mechanism that is placed between the user of an object and the source of that object

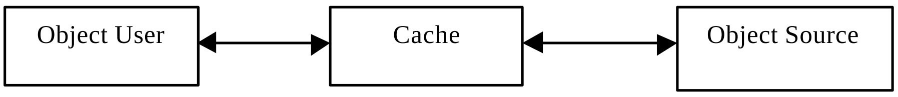

Figure 8 - using a cache to improve performance

such that requests to load an object from the object source (the RDBMS in this case) will pass via the cache. If the cache already contains that object then it will return a reference to its own copy rather than reloading it from disk. If the object is not in the cache (a cache miss) then the object is loaded, placed in the cache, and a reference to the cached copy is returned. This is an example of the use of the Decorator pattern (see [GoF]); the cache has exactly the same interface as the object source and is totally transparent to users.

The project under discussion used a two-level cache lookup mechanism to find objects. The first level uses the name of the database table as its key, and the second level uses the object's unique key (its primary key on the database). An object model of the cache is shown in figure 9.

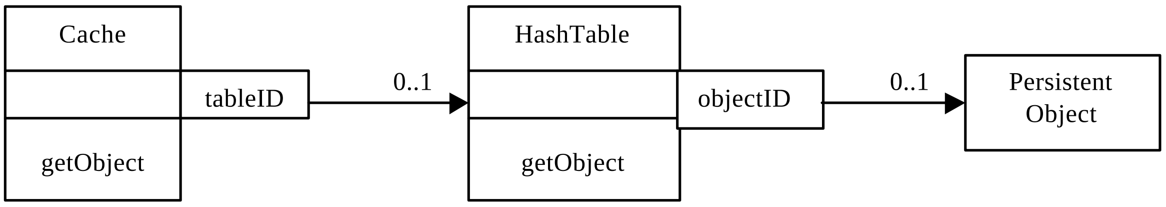

Figure 9 - Two level cache lookup mechanism

3.9. Mapping objects to tables

Relational databases typically support only fundamental data types such as integers, floating point numbers, timestamps and character strings, and in particular have no support for classes or other aggregate data structures. When implementing an OO system over a relational database, it is therefore necessary to define an explicit mechanism by which classes are mapped onto tables.

Ignoring containment and implementation inheritance, a simple mapping between objects and tables is possible: each attribute with an object becomes a column on a table, and each object becomes a row on a table.

Typically relational databases provide no support for implementation inheritance. It is therefore necessary to define a strategy for mapping inheritance hierarchies to tables. There are three approaches to this problem:

- a common parent table plus separate subclass tables
  - this approach wastes no space, but requires joins to load objects into memory
- a separate self-contained table per class
  - this approach wastes no space, but requires a different table to be accessed depending on the type of the object being loaded
- one table per single inheritance hierarchy
  - this approach is wasteful of space - the table will contain lots of nulls - but means that a single table can be accessed to load all objects in the hierarchy.

For example, the following object model:

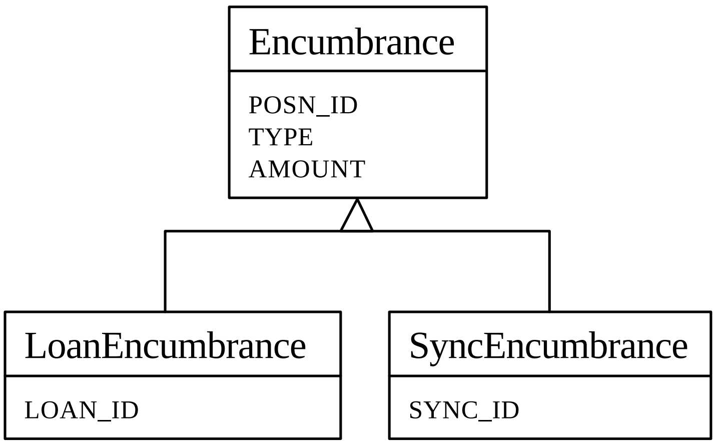

Figure 10 - implementation inheritance example

can be implemented in any of the following three ways:

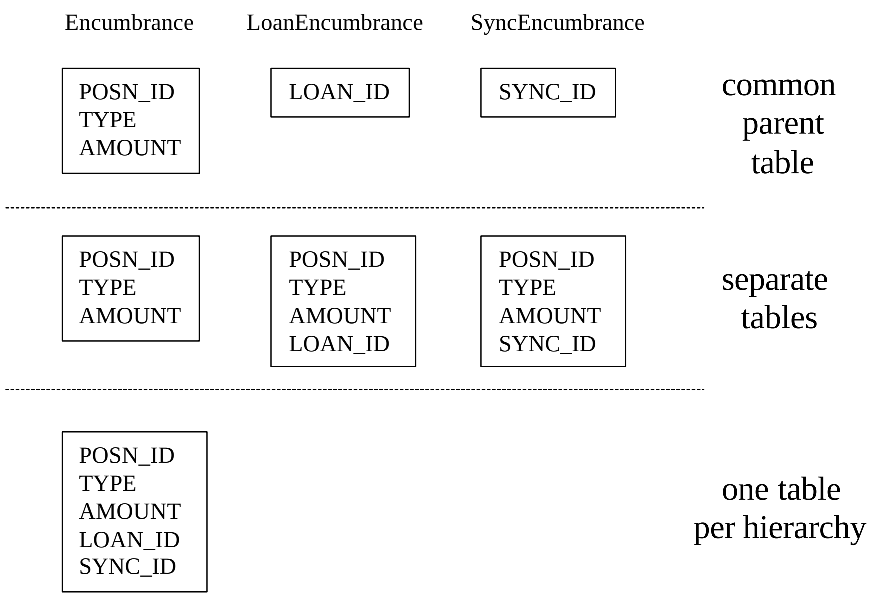

Figure 11 - Alternative table structures for implementation inheritance

In this particular example the amount of space wasted by the third approach is small as there are few differences in the attributes between the parent class and the children.

One significant difference between the three inheritance mapping schemes shown above is how they deal with polymorphic loading. If one wants a collection of smart pointers to Encumbrances (see figure 10), each of which might be a Loan or a Sync encumbrance, then the third approach is much the best in terms of speed as the parent and all the subclasses are in one table. Either of the other schemes would require multiple table lookups to load such a polymorphic collection.

The third of these mapping schemes is the one used in the project under discussion - primarily for speed and for polymorphic loading.

3.10. Polymorphic object creation using factory maps

The issue of polymorphic loading raises other questions: when we take a row from the database, what class of C++ object should we create for it? How should this be achieved? And how can such a mechanism be made extensible such that more application classes can be easily added to the system?

One approach is to use factory objects. A factory object is an object whose prime responsibility is the creation of objects of one particular class. Factory objects may also be allocated other roles, such as finding objects with a given property or retrieving all objects of a given type. This turns them effectively into manager objects that know about all objects of a given class.

To see how objects are created, loaded and initialised, we shall examine what happens when a simple smart pointer is dereferenced. The sequence diagram for the operations that take place when a cache miss occurs looks like this:

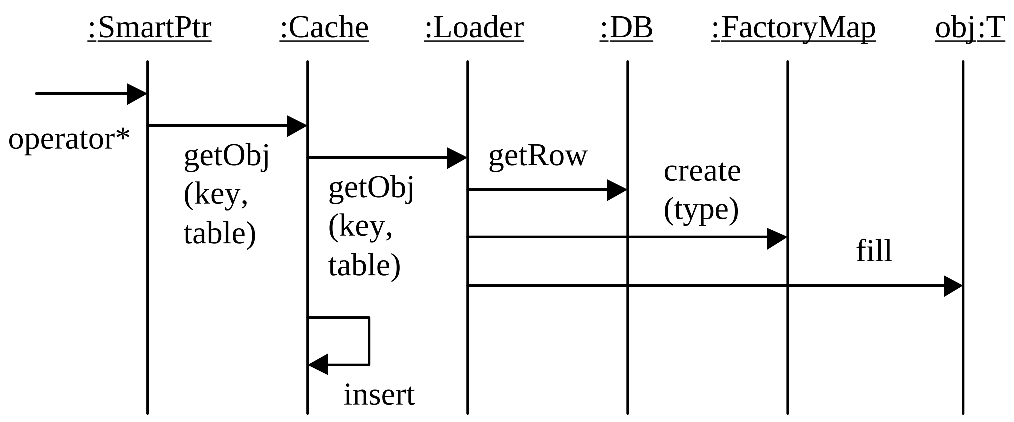

Figure 12 - dereferencing a single smart pointer

When the smart pointer is dereferenced, it queries the cache to see if the underlying object has been loaded already. If not, then the cache must fetch the object and then insert it into itself. The loader retrieves the relevant row from the database as specified by the object's key and the table ID. One of the columns contains a string that is the name of the object's class. In order to create an object of this class there is a map that maps strings to a static creation function of the corresponding class called the factory map.

This map is created and filled when the program starts. When the map has translated the string into a function pointer then that function is called and a new blank object of the correct class is returned. This blank object is then filled using the contents of the row retrieved from the database, the new object is placed in the cache, and the object's address in the cache is returned and dereferenced by the smart pointer.

4. Making changes persistent

Up to now we have considered read-only access to objects. We must now consider what happens when objects are to be modified and written back to the database and the effects this has on multi-user systems.

4.1. Account example

Again we shall use the account example as in the read-only case, but this time we shall modify the associated objects. The method we shall show is the pledgeAllPositions method. This marks all of the positions for this account as having been used as guarantees for loans.

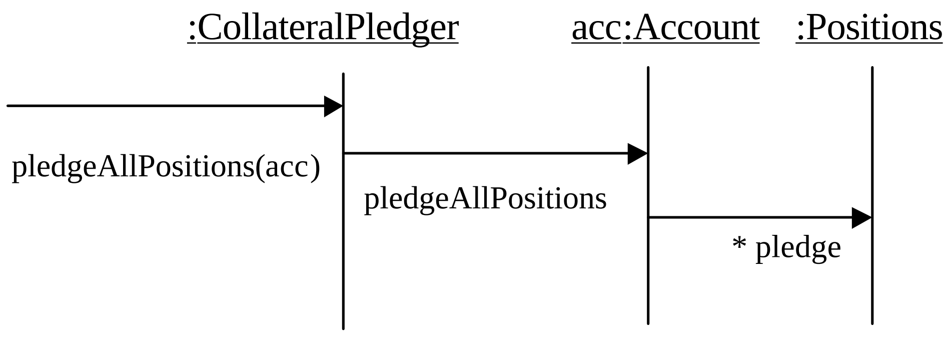

Figure 13 - Example of updating Account Positions

The object model and the high-level sequence diagram are identical to the previous version except that the pledgeAllPosistions method is called, which changes the state of Position objects.

4.2. Writing objects to the database

When should an object that has changed be written back to the database?

In the system under discussion, the approach taken was to write back to the database at the end of an application transaction all modified objects that were attached to a 'save hook' list. Activity (use case) objects such as CollateralPledger hook in any top level objects that require saving (e.g. an Account object). At the end of the CollateralPledger use case, the save hook is traversed and the 'save to database' method called on any hooked in objects. These in turn will recursively save any associated objects which might have been changed. The implicit save hook is thus a list of persistent objects to be saved that is maintained by the application framework.

4.3. Multi-user issues - transactions

Up to now we have ignored an important point: there will typically be more than one user accessing the database at one time. This brings up the possibility of different users' modifications interfering with each other. In RDBMS based systems this issue is traditionally handled by two mechanisms: database transactions and locking.

Database transactions are a way of ensuring that changes to a database are applied consistently. A number of changes are grouped together into a transaction, and either all of the changes are made or none of them are. Locking is a concept by which whole tables, or groups of rows within a table, can be 'locked' restricting others access to them. The concepts are combined to ensure that one user's modifications cannot interfere with another's.

The system under discussion used transactions and locking to deal with multi-user considerations. The begin/commit/rollback aspects of transaction handling were encapsulated in an abstract framework interface shown in figure 14:

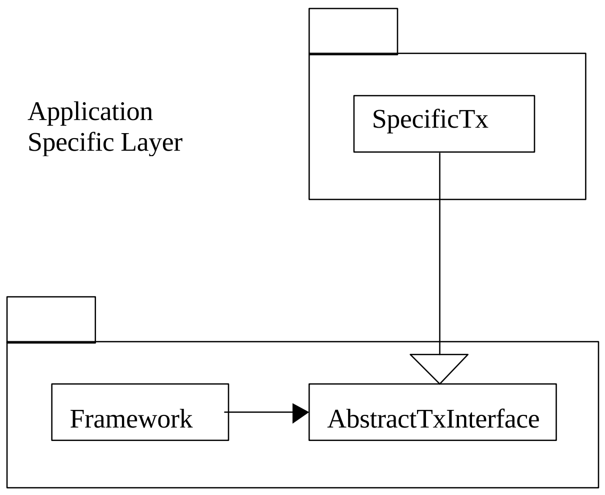

Figure 14 - Abstract transaction framework and specific use by application

This framework wraps transactions around application code as shown in figure 15:

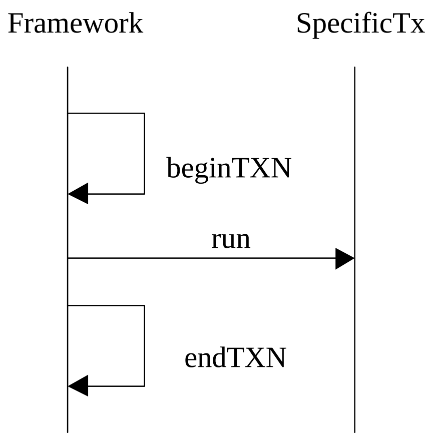

Figure 15 - Abstract transaction framework and specific use by application

The beginTXN method of the framework will 'begin' the underlying database transaction. The run method on the application specific subclass SpecificTx (a activity/use case object) will be called to do the work, and the endTXN call will perform a database COMMIT if the transaction terminates normally and a database ROLLBACK if an error has occurred or an exception has been thrown.

4.4. Cache and database coherency

Our discussion of the caching mechanism presented in this paper so far has mentioned only one cache. Since this cache in resident is memory (and not in the database), it is not possible to isolate the changes made by one user from being visible to another.

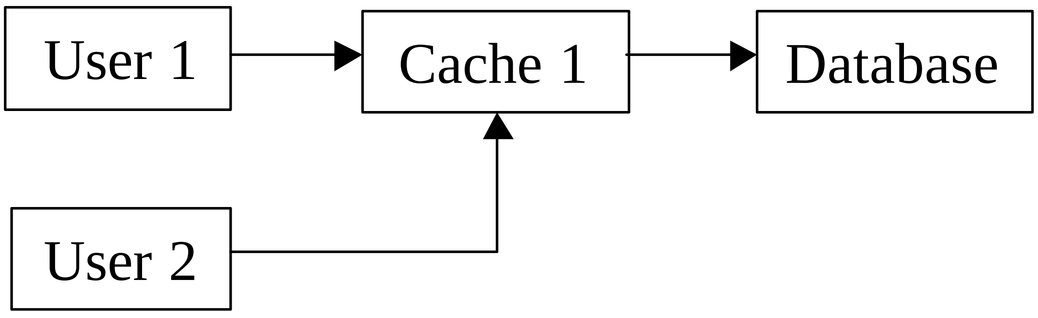

Figure 16 - Single cache model

One solution to this, adopted on the project we’re discussing is to have one cache per thread of control. This means that each thread sees only its own local changes to cached objects and cannot be affected by other users' modifications.

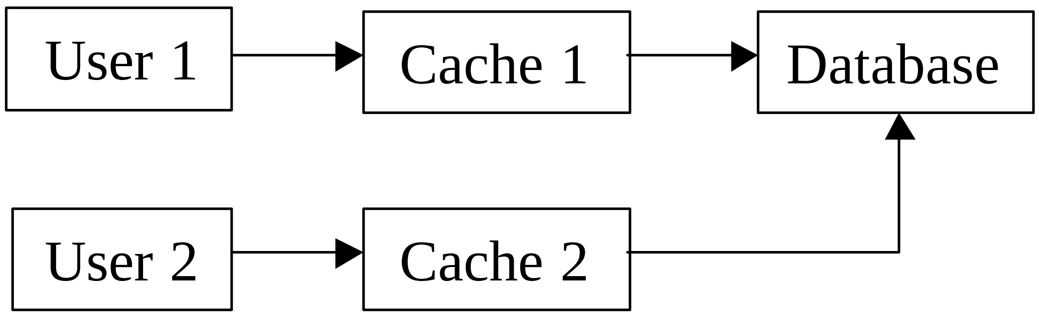

Figure 17 - Cache per thread of control model

5. Summary

Figure 18 gives an overview of the preceding discussion. It shows how the application framework relates to the use case objects and persistent objects, how the smart pointers relate to the cache, and how the persistence mechanism handles database transactions. The four major layer boundaries are shown with dotted lines.

In summary, the sequence of persistence related events based on the 'pledge positions' example discussed earlier is as follows:

- the application framework starts a database transaction
- the framework calls up to the use case object layer through the LogicalUnitOfWork abstract interface implemented by the CollateralPledger object (note: the previous discussion called this a SpecificTx).
- the CollateralPledger registers any objects (which must derive from PersistentObject) that must be saved by attaching them to the implicit save hook list
- it then calls the Account object to pledge all its positions
- the Account creates a query to fetch its positions and fills it with smart pointers to positions
- these smart pointers are dereferenced and the cache is searched for a matching object
- if no such object exists in memory it is created using the AccountFactory object which is registered in the FactoryMap. (The AccountFactory code is generated automatically by the code generation tools used on the project).
- this new object is then filled with the results retrieved from the database
- when the call to the use case object returns, control passes back to the application framework which writes all objects that are hooked to the save hook list and commits the database transaction. If an error occurs, the contents of the hook list are not written and the database transaction is rolled back.

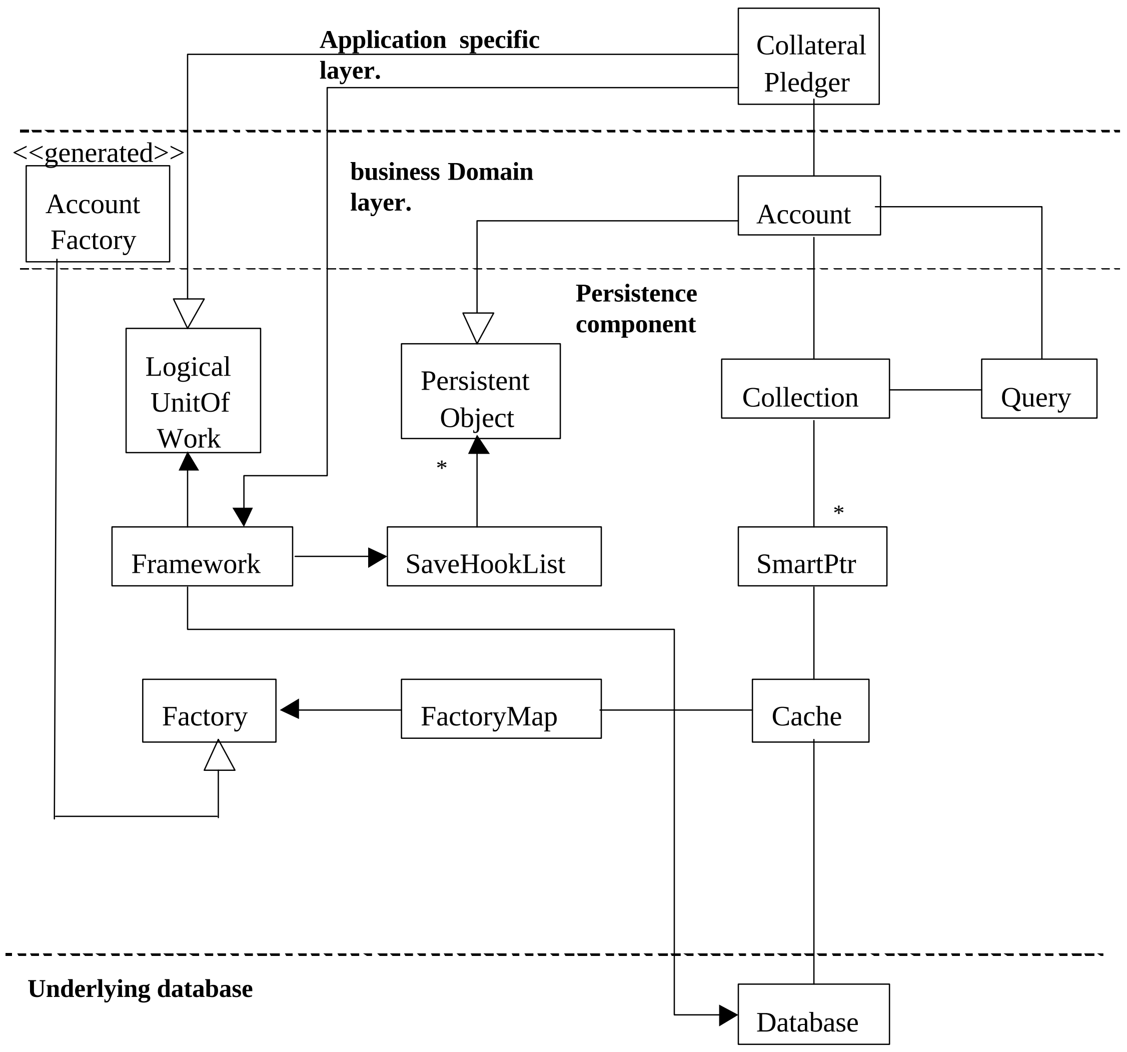

Figure 18 - Overview of persistence object model

In this paper we have shown how an object model can be persisted onto a commercial relational database system. The framework for doing this is layered, preserves encapsulation, has well managed dependencies, and is reusable. It is sufficiently flexible that it can accommodate any of the three standard ways of mapping object hierarchies to database tables without affecting user level code.

Smart pointers and persistent collections are used throughout to hide all of the persistence details from the application developer. This includes the mapping of objects onto tables, the loading and caching of objects, the specification of queries to indicate sets of objects to be loaded, transaction handling, and locking of tables. The system discussed has achieved this with good overall system performance, and without compromising on either encapsulation or layering – a significant accomplishment and one that contributes to the reusability of this persistence infrastructure.

6. References

[Meyers]	Scott Meyers, 1992, Effective C++, Addison-Wesley.
[GoF]	Gamma, Helm, Johnson & Vlissides, 1995, Design Patterns, Addison-Wesley.

"Components in Financial Systems", Available from Ratio Group.
"Objects and Component in a Financial System", Available from Ratio Group.
"Patterns in Financial Systems", Available from Ratio Group.

7. Appendix - More on optimisations

Performance was one of the main drivers in the design of the settlement project. Several implementation specific optimisations were added to increase performance. These were added in without breaking the application domain model and without polluting the technical infrastructure. Some of the optimisations are implementation specific and use features available with only an Oracle database, whereas some of the optimisations are generic and would work with any SQL database.

7.1. Use of physical row identifiers (Oracle ROW_ID)

When a row is retrieved from the Oracle database it is possible to get the physical record number of that row. In Oracle this is known as the ROW_ID. If the ROW_ID is specified in an SQL update then no searching of the database or its indexes is required as the exact record to be written is known. This is the fastest access path available when using an Oracle database.

7.2. Read-only configuration data caches and cacheable indexes

There are two read-only caches in the settlement project: one for configuration data that is needed globally and the other is under program control. Both caches are implemented using the operating system's shared memory facilities.

7.3. Potential use of Observer pattern for cache invalidation

One potential optimisation that is not currently implemented is the use of the Observer pattern to ensure that the copy of an object in the cache is the same as that on the disk by invalidating the cached copy when the database is changed.

7.4. Oracle array updates

Accessing the Oracle database using an SQL query or an SQL update has a fixed overhead. Therefore if a large amount of data is to be transferred to or from the database then for efficiency reasons it makes sense to transfer as much as possible per query to spread the cost of the query rather than transferring one row at a time. The settlement application uses an Oracle specific optimisation that allows arrays of data to be transferred in one SQL statement rather than having to perform individual transfers. This optimisation has been measured as being an order of magnitude faster than single inserts and updates.

8. Appendix - Error handling

Error handling is an essential part of a robust system. It cannot be presumed that all operations will succeed, and therefore we must be prepared to handle the consequences of potential failures. Such failures can be separated into two classes: those to do with the problem domain (such as attempting to overdraw an account) and those to do with the implementation (such as dereferencing a null pointer, or running out of memory). In the settlement project these two classes of errors are handled differently. Application errors are handled by the use of function return codes that indicate the success or failure of an operation. It is then up to the application developer to decide how to handle such errors. Possible courses of action would be to skip this entire transaction and move onto the next, or to abort the entire process and perform a clean shutdown of the program. Only the developer knows in a given context what the appropriate action should be. A warning message might be logged and if so it is up to the programmer to add sufficient context information to the message to allow for a post-mortem analysis to take place.

The other class of errors is technical errors such as an attempt to dereference a null pointer, or a programming error such as attempting to cast an object to an invalid type. This class of errors, on the other hand, is handled through the use of the C++ exception mechanism. When an error such as an attempt to dereference a null pointer is discovered an exception is thrown by the infrastructure. This exception is caught within the infrastructure itself and is always considered to be a fatal error – the error is logged, all pending database transactions are rolled back, and an orderly shutdown of the program is performed. This guarantees the correctness of the system in the presence of implementation failure.