Patterns for Three-Tier Client/Server Applications

Amund Aarsten, Davide Brugali, Giuseppe Menga
Dept. of Automatica e Informatica
Politecnico di Torino, Italy
email: {amund,brugali}@athena.polito.it, menga@polito.it

Introduction

The three-tier client/server architecture is an evolution of the traditional two-tier model, and is receiving increased interest, particularly for large business applications. The main difference is that in a three-tier architecture, most of the functionality is separated out in a middle layer, called *application servers*, as shown in figure 1. Also, each client can use several application servers, each of which in turn can work against several databases.

Figure 1: Three-tier architecture.

The majority of three-tier applications have until now been based on relational databases and transaction processing (TP) monitors. However, the use of ˝object-oriented technology is growing.

This article presents some patterns for development of object-oriented three-tier applications, with particular focus on the application server layer. The patterns can be seen as a mini pattern language, or a first step towards a complete pattern language for three-tier applications. Pattern 1, *Three-Tier Architecture*, gives the overall application structure. Pattern 2, *Application Server Partitioning*, guides the process of dividing the middle layer into several application servers. Pattern 3, *Abstract Database Interface*, makes the application portable to different database platforms. Pattern 4, *ORB Proxy for Database Object*, deals with the integration of the often incompatible worlds of object brokers and object databases. Pattern 5, *Automatic Object Locks*, liberates developers from the tedious and error-prone work of manually locking and unlocking database objects. Pattern 6, *Shared DB Connections*, applies to the cases where it is necessary to have more simultaneous users than the number of connections the database servers will accept. This leads to pattern 7, *Thread-Based*

Concurrency Management, which shows how concurrency control can be implemented in application servers when database locks can no longer be used to ˝synchronize the clients.

While pattern 5 is fairly C++-specific, the other patterns should apply to most object-oriented three-tier application development projects.

Pattern 1: Three-Tier Architecture

Context

Development of a large business application, where many users share common data and operations on them. In addition, there might be legacy systems which have to be integrated in the new application.

Problem

Which is the best architecture for these applications?

Forces

The traditional client/server model presents a few problems for large applications, especially concerning scalability and portability. Most of the application logic is in the client, and is therefore tied to the user interface code unless the programmers are extremely disciplined; this creates a dependency on the tool and operating platform used, resulting in a system that is difficult to port to other client platforms, and whose user interface cannot be easily customized. As the application continues to grow, chances are that the client hardware platform must be upgraded in order to run the application efficiently. Application upgrades, including simple bug-fixes, will have to be installed on all the clients.

Because of the same dependencies, extending a legacy system with new functionality is also very difficult.

Solution

Implement the application using a three-tier architecture as shown in figure 1. When using object-oriented technology, the roles of the different layers are ˝usually as follows:

- The client layer contains a graphical user interface (GUI) which calls the middle layer in response to user commands, usually going through an object request ˝broker (ORB) such as CORBA [2].
- The middle layer, i.e. the application servers, contains objects representing business entities and the operations on them.
- The database layer handles the storage of the application ˝objects used by the middle layer.

Where possible, wrap legacy systems in application servers by applying the pattern ADAPTER from [10].

The resulting application is easily scalable; the capacity of the system can be increased by adding application servers and partitioning the clients between them. The client, being light-weight, requires moderate computing power, and can be customized and ported with relative ease to other platforms. Many application upgrades can be done by replacing an application server without affecting the clients. The application logic can also be ported to other platforms more easily, since it is clearly separated from the user interface code.

Example

In our experience, the three-tier architecture offers significant advantages even for relatively small applications. For instance, the single-user PC application First Account from the Norwegian company Economica AS [16] encapsulates most of the accounting and invoicing functionality in a dynamic link library (DLL), which in turn works against a local, flat-file database. This separation enabled the developers with knowledge of accounting and object-oriented design to dedicate themselves to the central functionality, and user interface designers with little or no knowledge of programming to fully control their part of the application.

Pattern 2: Application Server Partitioning

Context

A large application will have several application servers.

Problem

How do we decide what goes into each application server?

Forces

Moving some of the application classes to a separate server reduces dependencies, and can allow a part of the application to be upgraded without affecting the other parts. However, it also reduces performance because of the remote procedure calls (RPCs) that now become necessary. Therefore, we must take care to partition the application so that as few ˝RPCs as possible are necessary.

Ideally, the partitioning should be the result of an exhaustive analysis of the interactions between all the objects in the application, but this is not practical for ˝serious applications.

Solution

The analysis and design phases will produce a large object model, also called the Business Object Model (BOM). In a large application, the BOM covers several different lines of business and business processes.

Group objects from the BOM which are tightly coupled and have high cohesion into application servers. Typically, these groups will represent lines of business or business processes. If your notation uses class categories (as in [1,3]), these can provide good starting points. Some classes need to be in several application servers for performance reasons; do this as little as possible in order to minimize dependencies.

Pattern 3: Abstract Database Interface

Context

Application objects are usually stored in an object-oriented database. Sometimes, however, the customer dictates a database platform; in that case the ˝database is usually relational.

As also noted in [4], all object databases have significally different programming interfaces. Even if a standard exists [5], this standard is somewhat incomplete, and every implementor fails to support some parts of it. This means in practice that an application written for a specific object database is not portable to other databases.

The situation is even worse if a relational database is used, potentially requiring a major rewrite of large parts of the application.

Problem

How can the application servers interact with the database ˝without depending on its interface?

Forces

A common solution to this problem is to place code specific to the database platform in a limited subset of methods of the classes in which this code is used. Although this approach controls the location of the code that later must be changed, it does not scale very well. In a large application there can be thousands of classes; the task of porting to a different database can soon become overwhelming even if only a few methods of each class must be ˝changed.

Solution

Define a standard interface against the database, and implement this interface for the specific database required by each project; see the pattern *Interface to Control Modules* in [6] and the use of separate interface and implementation objects in [7]. If separate interface and implementation objects are used, the switching between different implementations corresponds to the BRIDGE pattern in [10].

The most important parts of the standard database interface will be for managing connections, transactions, locks, object references, and standard collections such as SortedCollection which are not defined by the standard [5]. See [4] for further discussions of these issues.

If a relational database must be used, this separation also allows the tuning of how objects are mapped to relational tables. See [8, 9] for further discussion of this. In this case the database interface object (or more likely, subsystem) acts as an ADAPTER [10] to the relational object representation.

Separating the database interface from its implementation also enables the use of the pattern *Prototype and Reality* from [6], greatly simplifying prototyping and experimental development by allowing the use of an in-memory simulated database during the time when the database schema would have to change frequently.

Pattern 4: ORB Proxy for Database Object

Context

Application objects are accessed through an ORB; at the same time they are stored in the database. ORBs and databases are typically incompatible, in that a class cannot be both an exported ORB class and a persistent database class.

Problem

How can we integrate the ORB and the database in order to implement both aspects of the application objects?

Forces

ORB interface and database schema compilers will typically generate code where classes are modified so that the leftmost base-class is always a class needed by the runtime mechanisms. On the

object layout level, this means that the ORB runtime will assume that some methods it needs from an object occupy the first entries in the virtual method table; the database runtime assumes the same thing.

Since the ORB and database objects cannot coexist in a common ˝inheritance hierarchy, there must be two separate hierarchies. This effectively doubles the number of classes in the application server, which can create maintainence problems unless we find a good way to coordinate the two hierarchies. It also doubles the number of objects at runtime, which can have a negative impact on performance.

Solution

Make the ORB object a proxy [10, 11] of the database object.

In a large project, maintaining two separate definitions of each application class is not practical. Instead, use a tool for defining classes and generate the ORB and database class definition from the representation used by this tool, as shown in figure 2.

This pattern, while being a straightforward application of PROXY from [10], is also one of the applications of piecemeal programming language extension discussed in [12]. In that case, as described in [13], private data is included in the .IDL ORB interface definition file, and the database class is generated by a translator written in PERL.

Typically, the use of a proxy would imply that the application code resides in the methods of the database object. However, this is not necessarily the case here, as discussed in the patterns *Automatic Object Locks* and *Limit DB Connections* below.

The total number of proxies that a call from the client goes through in order to get to the database object is quite high: Including the ORB proxy on the client side, the ORB skeleton on the application server side, the ORB implementation object, and the object in the application server's database client interface, there are four proxies. The numbers of indirection means that the performance could potentially suffer. In our experience, however, the performance loss is inconsequential compared to the network and database overhead.

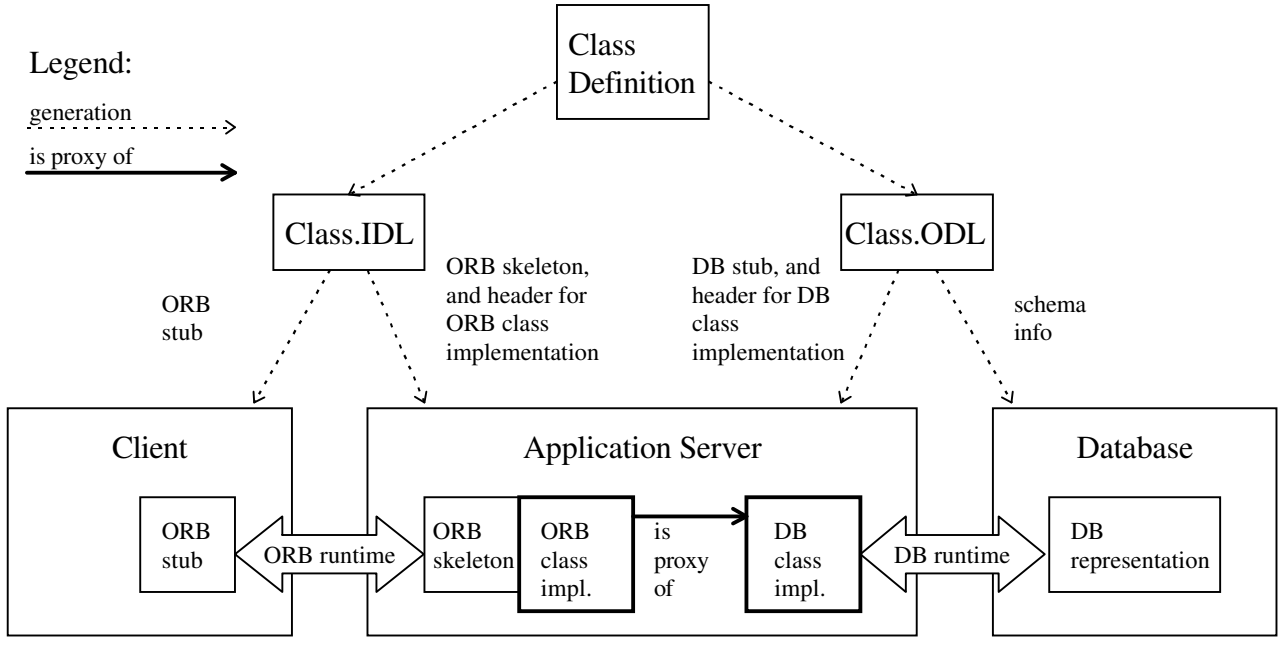

Figure 2: ORB implementation object is a proxy of the database ˝object

Pattern 5: Automatic Object Locks

Context

Application servers handle several requests from clients simultaneously. Even if each client connection runs in a separate memory space, they share the application objects that are stored in the database. Also, an application server shares the same database ˝with other servers.

Concurrency is handled using the locks and transactions of the ˝database.

Problem

Many object databases require the programmer to ask explicitly for a lock before modifying an object, and/or notifying the database after the modification. ˝This is tedious and error-prone.

Forces

We must minimize the reliance on programmer discipline, and instead exploit mechanisms available in the programming language and the tools. At the same time, the constructs we employ cannot require programmers to change significantly their style of ˝programming.

Database locking and change notification can be hidden behind accessor methods, removing this burden from the programmers. However, while accessor methods are the norm for some programming languages, like Smalltalk, it imposes a non-standard style of programming for many other languages, including C++. Although C++ programmers with experience in CORBA use accessor methods for public attributes, it can upset them a great deal if they are dictated to use them for private attributes also.

Furthermore, in most databases private attributes cannot be used in index definitions or queries. This means that they must be defined as public, and hence the programmers' discipline cannot be verified by the compiler in any way.

Solution

Virtualize the data members after the approach used in the pattern *Objectify System State* in [15]. Define a class `DBData<T>` to use in place of data members of type `T`. Each `DBData` contains references to the database object and the member it corresponds to. The operations that modifies that member would then check for a lock in the same manner as the accessor methods discussed above.

Towever, the `DBData` objects should not be part of the database classes -- the references involved just add overhead to the database, and makes the job of the database administrator much more complex. If the ORB's interface definition language allows the definition of private members (most do), they would go into the ORB implementation class. This ˝means that the code that implements the application server has to go into the methods of the ORB implementation class, while the database class only holds data, as illustrated in figure 3. If private members cannot be used in an interface definition, another level of indirection must be used (ORB implementation object is a proxy of the `DBData`-containing implementation object, which in turn through its members is a proxy of the database object.) This solution also lets us define all data members of the database classes as public, enabling their use for indexes and queries, while the `DBData` members used by the programmer can be public, private, or protected as needed.

`DBData` members or accessor functions can be generated automatically from the same tool mentioned in *ORB Proxy for Database Object*.

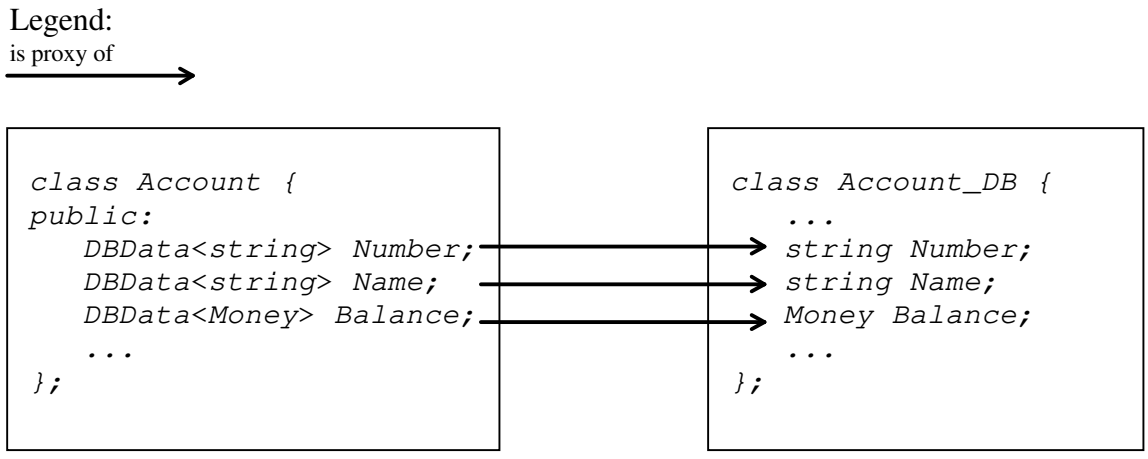

Figure 3: ORB class members as proxies of DB class members.

Pattern 6: Shared DB Connections

Context

You are developing an object-oriented distributed application ˝based on a *Three-Tier Architecture* for a customer who has also been considering traditional relational ˝and TP monitor technology.

The customer knows that one of the characteristics of TP monitors is their ability to reduce the number of simultaneous database connections required by an application. This can significantly reduce the cost of the base software for a large installation, since the cost of most database servers depend on the number of simultaneous connections they will ˝accept.

Problem

The client expects your object-oriented solution to offer the same savings on the initial cost of the base software as a solution based on relational technology.

Forces

The traditional solution to this is to only allow the same number of simultaneous client transactions as the number of database connections the application server has available. However, this severely limits the advantages of an interactive graphical user ˝interface, including an effective mix of browsing and data entry.

Solution

Manage the available database connections by queueing individual requests (instead of transactions) to the database interface, serving each request when a connection becomes available. Note that once a client has used a certain database connection, subsequent database requests from that client will have to use the same connection; therefore, maintain a dictionary which maps each client to its database connection.

Since the database will only allow a single transaction simultaneously for each connection, while there might be several client transactions overlapping in time sharing the same connection, client transactions must be separated from database transactions. This is described in the next pattern, *Thread-Based Concurrency Management*.

Pattern 7: Thread-Based Concurrency Management

Context

You are developing application servers in a three-tier ˝application using *Shared DB Connections*.

Problem

Without one database connection for each client, the concurrency control mechanisms of the database can no longer be used to coordinate the clients.

Forces

For instance, an object might be locked and changed by one client transaction, after which an other client using the same connection commits. The object locked by the first client will be written and its lock released, effectively commiting part of the first client's transaction. This violates the atomicity property of transactions, makes it impossible for the first client to do a rollback, and could even make it impossible for it to continue since the object might ˝now be locked by someone else.

Solution

Implement application servers as multi-threaded CORBA servers, where all client connections run in the same memory space, but each client request creates a new thread in which the request is executed. This is similar to the pattern *Client/Server/Service* in [6], with the exception that the latter supports thread-local storage in addition to the shared server ˝resources.

Create a class to support the concept of client transactions, which must be separated from database transactions. Make a mapping mechanism from the ID of a client to the corresponding client transaction and any other required context, and use this information to set up the new thread created for each client request. This can be done using a mechanism such as the "filter" callbacks supported e.g. by Orbix [14].

Since simultaneous client transactions can share the same database connection, optimistic transaction control must be used. Each client transaction must remember which objects it has modified (e.g. through the *Automatic Object Locks* mechanism,) and at commit time write them all to the ˝database.

Optimistic concurrency control can be avoided among client transactions in the same server by having them communicate through a set (suitably protected e.g. by a semaphore) holding all currently locked objects. If the *Application Server Partitioning* works well, there will be far more conflicts within an application servers than between different ˝ones.

To allow rollback of client transactions, modified and deleted objects are simply read back from the database. Alternatively, their state could be saved upon insertion in the lock set; this would also allow for optimistic concurrency control among client transactions if this is preferrable in a given situation.

Conclusions

The collection of patterns presented here focus on the application server layer of three-tier applications, and its integration with the database layer. While we believe the first five patterns should be applicable to almost every object-oriented application ˝built around an ORB and a database, the two last patterns can be a lot of work to implement. Even so, it is a one-time investment, and the ability to reduce the number of database connections is a strong ˝selling point.

There are other important patterns involved in building object-oriented three-tier applications; for for instance, the implementation of associations in the database, application packaging, and user interface issues. We hope to cover them in a future paper.

Acknowledgements

These patterns are the results of work at the Norwegian company Economica AS, in cooperation with the CIM group at Politecnico di Torino. We would like to thank all the people who read and commented early versions of this paper. Special thanks to our shepherd, Gerard Meszaros, whose comments were invaluable for structuring the concepts and making the patterns coherent and readable.

References

[1] G. Booch. *Object Oriented Design with Applications. Benjamin/Cummings*, 1991.
[2] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, September 1992.
[3] G. Booch and J. Rumbaugh. *Unified Method for Object-Oriented Development, Version 0.8*. Rational Software Corporation, 1996.
[4] J. Wikman. *Database Independence*. Nokia Research Center, February 1996.
[5] E. G. G. Cattell (ed.) *ODMG-93: The Object Database Standard*. Morgan Kaufman, 1994.
[6] A. Aarsten, D. Brugali, and G. Menga. Designing Concurrent and Distributed Control Systems: An Approach Based on Design Patterns. To appear in ˝Communications of the ACM, Nov. 1996.
[7] J. Coplien. *Advanced C++ Programming Styles and Idioms*. Addison-Wesley, 1992.
[8] K. Brown and B. G. Whitenack. *Crossing Chasms*. Proc. PLoP'95.
[9] S. Agarwal, C. Keene, and A. M. Keller. *Architecting Object Applications for High Performance with Relational Databases*. Persistence Software, Inc.
[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
[11] H. Rohnert. *The Proxy Design Pattern: Revisited*. Proc. PLoP'95.
[12] C. F. Salviano and V. S. S. Nair. One-for-Many: *A Design Pattern for a Piecemeal Programming Language Extension through One-to-Many Translation*. Proc. PLoP'95.
[13] Object Design, Inc. *Integrating ObjectStore and Orbix 1.3*. 1995.
[14] Iona Technologies, Ltd. *Orbix 2 Reference Guide, Release 2.0*. November 1995.
[15] A. Aarsten, G. Menga, and L. Mosconi. *Object-Oriented Design Patterns in Reactive Systems*. Proc. PLoP'95.
[16] *First Account brukermanual*. Economica AS, 1996. (In Norwegian.)