Patterns for Layered Object-Oriented Applications

Mauricio J. Vianna e Silva*[1] Sergio Carvalho*[1] John Kapson+

* Laboratório de Metodos Formais, Dept. de Informática,
Pontificia Universidade Católica do Rio de Janeiro, R. Marques de São Vicente 225,
Rio de Janeiro, RJ, 22453-900, Brazil
email: mauricio@lmf-di.puc-rio.br, sergio@inf.puc-rio.br

+Spectrum Consulting Services, Inc., Illinois, USA
email: kapson@netcom.com

Abstract

Three-layered architectures (presentation, business and persistence levels) are sometimes recommended when developing object-oriented applications. Using the increasingly popular fourth generation languages, however, layering is difficult: business rules are usually embedded in user interfaces, which may also directly access databases. In this paper, we present three patterns useful in the construction of object oriented applications accessing relational databases via fourth generation languages.

Motivation

With the increasing interest in object-oriented technology, many companies have started to worry about how to store application business objects in databases. A natural approach to store objects would be to use an object-oriented database system. However, in practice, many companies have instead opted to use a relational database to store objects [Duhl96], as witnessed by the increasing popularity of Fourth Generation Languages (4GL's), such as Powerbuilder and Delphi.

By combining object-oriented technology and relational databases we have the advantages of both worlds [Keat95]. However, this is not easy, and pattern languages have been proposed to bridge the existing gap between the two technologies [Brow96a, Kell96]. The use of Fourth Generation Languages (4GL's), such as Powerbuilder and Delphi, further complicates matters, as these languages do not encourage the object oriented modeling of business rules in a separate level. This in the long run may lead into systems that are immensely expensive to maintain and enhance [Swam97].

A commonly suggested approach for the construction of complex applications is the separation of application concerns in different layers [Busc96, Aars96, Hirs96]. This in turn brings about layer communication and visibility issues: in the case of object oriented applications, how do different layer objects communicate? Which objects are visible?

In this paper, we propose a set of patterns for object oriented applications using 4GL's to access relational databases. They address communication, visibility, reuse and layering issues. Our approach extends the work proposed by [Kell96, Brow96a, Busc96], adding three patterns: Strong Layering, Weak Layering, and Entity Broker. These patterns deal essentially with communication issues in applications that have been partitioned into three layers: presentation, business, and persistence. They

[1] This work was partly supported by CNPq, and was developed for the ARTS (Approach to Real-Time Software) project. ARTS is an ongoing three year project contracted by EQUITEL (a SIEMENS company in Brazil) with the Formal Methods Laboratory of the Catholic University in Rio de Janeiro.

have similar contexts and address the same problem, but offer different solutions with respect to the forces at work.

Common Context:

The development of layered object oriented applications involving presentation, business and persistence levels, using company data stored in an existing, proven, company-wide relational database.
Application classes may be partitioned, according to their functionality, in separate layers. The first layer deals with presentation aspects, such as capturing, displaying and formatting data; the second layer deals with business aspects, such as enforcing the business rules and calculating values; and the third layer deals with database aspects, being responsible for storing and retrieving data [Brow96b].

In all our patterns, the storage and retrieval of objects is done through a separate hierarchy of persistence or data access classes, parallel to the hierarchy of business classes [Your95] (see Figure 1). Classes in this hierarchy should encapsulate any database access required by its associated business class, providing a uniform object-oriented interface to the relational database.

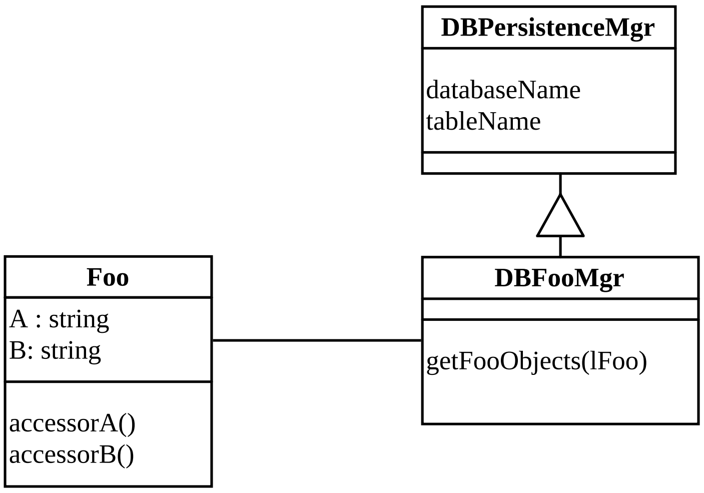

Figure 1: The association between business and persistence classes

Thus, visual object-oriented applications using a business class Foo with a parallel persistence class DBFooMgr, are organized in a layer structure with class FooUserInteface in the presentation layer, class Foo in the business layer and class DBFooMgr in the persistence layer (see Figure 2).

Layer 1 - Presentation	FooUserInterface
Layer 2 - Business	Foo
Layer 3 - Persistence	DBFooMgr

Figure 2: Three-layered architecture

Common Problem:

What is a good approach to organize object oriented applications accessing relational databases via 4GL’s?

Common Forces:

As mentioned in [Kell96], besides *performance* and *flexibility*, obvious forces when handling databases, the *decoupling* of applications from the physical database is essential. A good decoupling facilitates not only domain modifications, imposed by new business rules, but also changes in the database itself.

Decoupling is also the key when the same database must be accessed by users with different access rights, each seeing a different set of interfaces; it is essential that all database presentations be separated from its implementation. The same force suggests the modeling of business rules in yet another level, since these may change independently of the database, and vice-versa.

Common Participants:

DBPersistenceMgr

- Defines abstract protocols for all persistence classes, such as Insert, Update, Delete; in short, establishes the need, in all descendants, for the usual relational database operations.

DBFooMgr

- Specializes DBPersistenceMgr, and is associated to class Foo. For storing Foo objects, we create methods in DBFooMgr that will receive either a single Foo object or a list of Foo objects. Inside these methods we can use either embedded SQL statements to update the database, or calls to stored procedures passing all attributes of the Foo objects as parameters.
- For recovering Foo objects, we create methods in DBFooMgr with parameters that will be used to query the database and locate the corresponding tuple(s). The query in the database can be done by using embedded SQL directly inside the method, or by calling a stored-procedure, which may be more efficient[Agar95].

Foo

- A business class. With the result of the query, we create the Foo object. Besides returning a single object, DBFooMgr can also return a list of Foo objects created with the information retrieved from a related table. Thus, browsing Foo objects in an interface is done by calling DBFooMgr methods to return a list of objects and then by calling accessor methods defined in class Foo.

FooUserInterface

- A class, or set of classes, constructed to present to the application's user the functionality required.

Pattern 1: Strong Layering

Also Known As:

Restricted Communication

Forces:

The main forces are the *reuse,* and *maintenance* of classes in all layers (particularly business classes), which must be maximized. In contrast, the level of indirection introduced increases the *overall response time*, especially when the layers are in separate physical locations.

Structure:

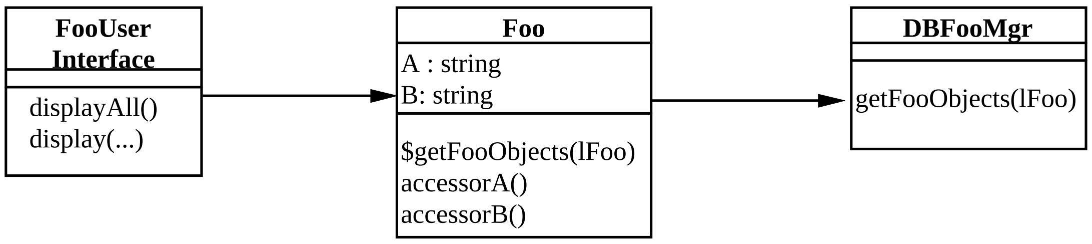

Figure 3: Accessing Foo Data using Foo Class Method

Solution:

Allow the presentation layer class FooUserInterface to communicate only with class Foo in the business layer, isolating the persistence layer from the presentation layer (see Figure 3). If the interface wants to display information about Foo objects, it calls a class method in class Foo to return a list of Foo objects, which in turn calls an instance method in class DBFooMgr (see Figure 4).

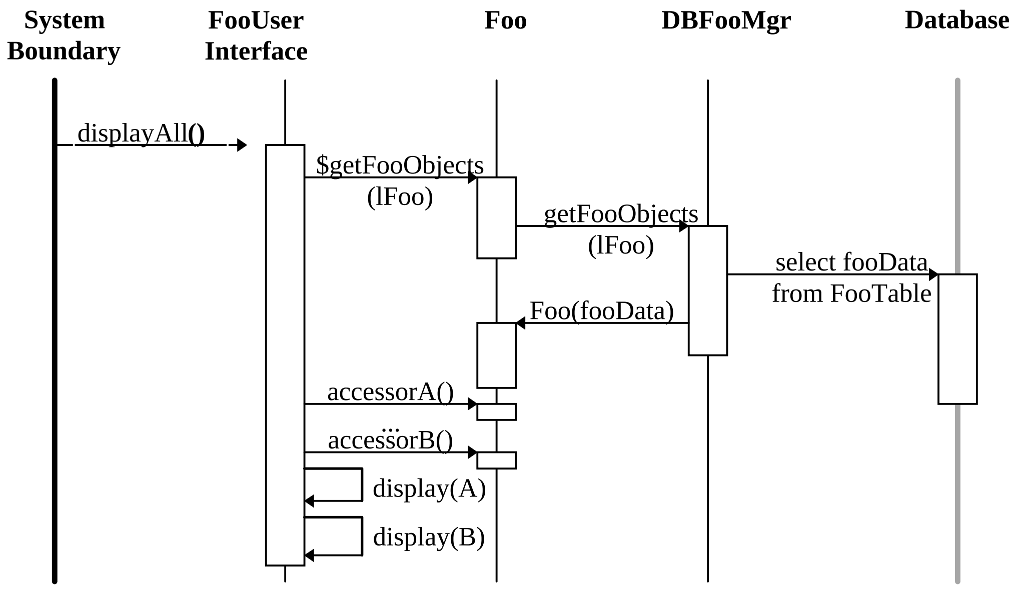

Figure 4: Behavior using Foo class methods

Benefits:

- This approach provides a nice isolation between the presentation and persistence layers, by forcing all requests to be mediated by the business layer, enhancing the maintenance of these classes.
- Developers of new interfaces need only to know about business classes, isolated from implementation specific classes.

Drawbacks:

- With this approach developers end up defining, in class Foo, methods that are responsible for object persistence or data accesses, and which are not related to its business behavior.
- In addition, this approach may not be efficient when the objects in the layers are in separate physical locations. This introduces one level of indirection for accessing Foo objects, because all calls have to go through the business layer (class Foo).

Pattern 2: Weak Layering

Also Known As:

Relaxed Layering, Free Communication

Forces:

The main force motivating the use of this pattern is *overall response time*, which must be minimized. Thus different layer objects may communicate at will, even though they belong to different layers, have been designed by different teams, etc. As long as visibility is granted, free communication can occur. This is common in 4GL's, where user actions frequently result directly in database accesses. In a strictly layered architecture (see next pattern) such visibility rights would be inexistent, increasing overall response time.

A contrasting force is *design complexity,* which increases when developers are required to know not only about classes in their own layers, but also about classes in other layers. Other contrasting forces are *maintenance* and *learning effort,* much more difficult since the layer separation is somewhat relaxed via free communication rights. The same is true if layers are located in separate physical locations.

Structure:

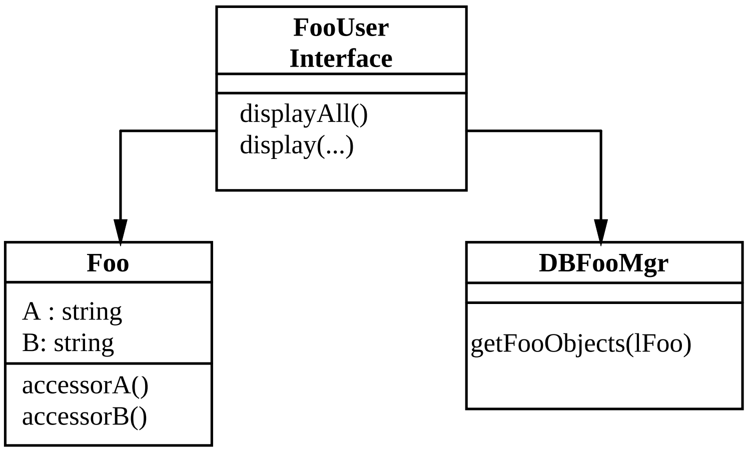

Figure 5: Accessing Foo data using classes Foo and DBFooMgr

Solution:

Allow the class FooUserInterface in the presentation layer to communicate directly with classes Foo and DBFooMgr in the business and persistence layers, respectively (see Figure 5). If the interface wants to display information about Foo objects, it calls a method in the DBFooMgr class to return a list of objects and then it uses Foo's accessor methods to get the information desired (see Figure 6).

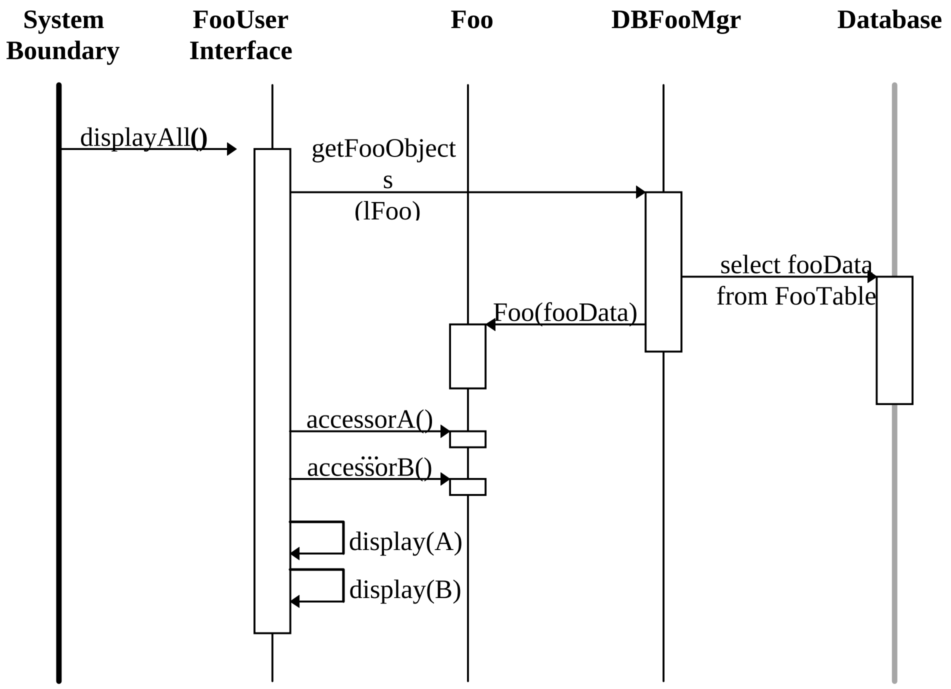

Figure 6: Behavior using DBFooMgr directly

Benefits:

- This approach may be efficient with respect to response time, especially when the objects in the layers are in separate physical locations [Brow96b], since Foo objects can be stored/retrieved by accessing the persistence layer directly.
- In addition, class Foo has only methods related to its business behavior, moving to its associated persistence class DBFooMgr, the responsibility for implementing database accesses of retrieving and storing Foo objects. Thus, this approach allows the class Foo to be *pure* and its subclasses to inherit only business behavior, not polluting its interface with implementation details like persistence methods.

Drawbacks:

- This approach is inflexible to changes in all levels, because application knowledge may not be correctly localized. For example, a change in a business rule may not be captured by a direct transaction between an interface and a persistence manager.
- The developer is required to know not only about business classes, but also about database implementation classes. It does introduce several interactions among the classes in the layers, creating tight couplings among them.
- Maintenance and learning costs increase accordingly, due to the weakening of the layered architecture caused by free communication rights.

Pattern 3: Entity Broker

Forces:

Reuse and *maintenance* issues of presentation, business and persistence classes dominate design decisions when this is the chosen organization. These forces are stronger when the implementation system allows asynchronous messages sent to generic receivers (see Implementation below).

Another force to be considered has to do with the *commitment to layered architectures* we decide to impose on the application: an entity broker actually introduces another dimension, another class condensing the functionalities of a business object, its user interfaces, and persistence manager.

Structure:

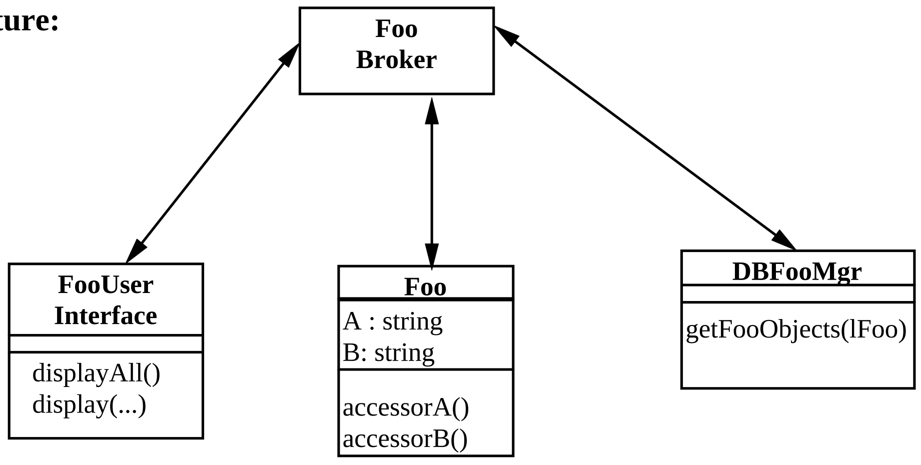

Figure 7: Accessing Foo data using a High Level Broker

Solution:

Introduce a new class that mediates the three layers. For instance, the interactions among FooUserInterface, Foo and DBFooMgr are encapsulated in a new class, called the FooBroker (see Figure 7), functioning similarly to the Mediator pattern [Gamm95].

User interfaces communicate all requests to the business object broker, which in turn acts upon classes Foo, DBFooManager and even the interfaces themselves (see Figure 8).

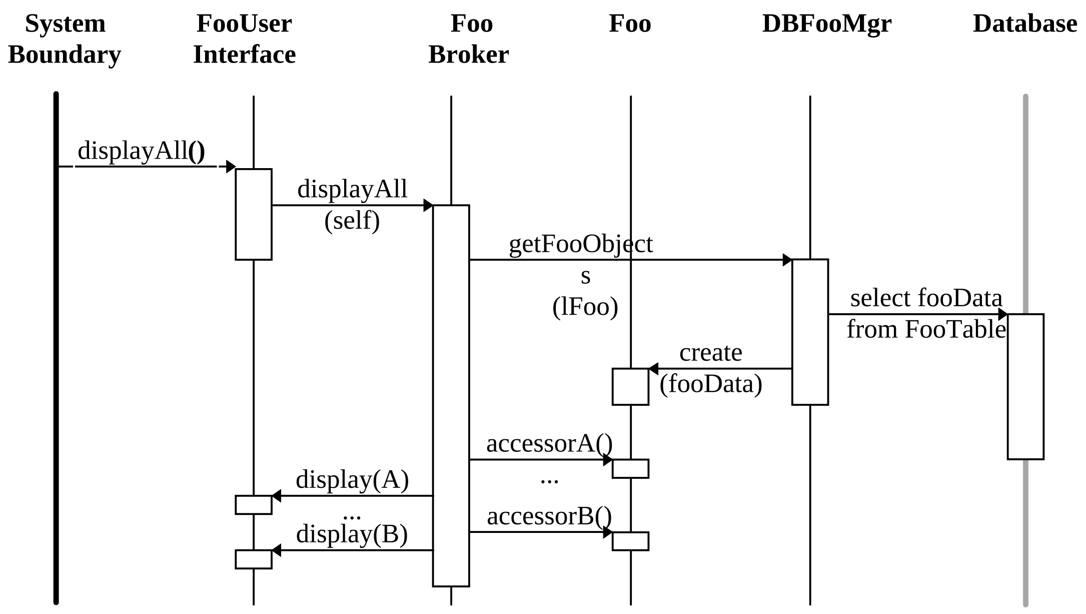

Figure 8: Behavior using a high level broker

Benefits:

- This solution facilitates the maintenance of each class managed by the broker. By decoupling FooUserInterface from Foo and DBFooMgr, we localize changes to individual classes, reducing the impact of any class change.
- In addition, the decoupling of classes allows them to vary independently, which enhances their extensibility and reusability.
- A business object may have several interfaces, different presentations offered to users with different access rights; this pattern facilitates the addition and removal of such interfaces.

Drawbacks:

- The broker may become very complex when it encapsulates several components, which interact in many different ways.
- The broker may become difficult to maintain, because it is tightly coupled with the classes it mediates.

Implementation:

The broker architecture has its benefits maximized when component objects may:

- Communicate with their brokers by sending them asynchronous messages.
- Refer to the broker using some reserved pronoun, such as PARENT.

In this way, we can develop presentation, business and data access classes as real black boxes, facilitating for example interface changes. Reuse is maximized, since such classes can be placed in other contexts without code changes. Brokers must of course be aware of the message sending behavior of their components, in order to handle them accordingly.

Conclusions

We described three patterns addressing general layer communication techniques, specializing their use in the organization of object oriented applications accessing relational databases. In all, the main concerns were reuse and maintenance, and overall response time.

In selecting a pattern from this set, one has to consider the cost of object communication in the application (the overall response time) versus the need to reuse and maintain application classes. From the point of view of presentation objects, important in fourth generation languages, a decision process might go on like this:

- Should presentation objects only know about the existence of pools of business objects but be isolated from the fact that a database exists? If so, then some form of class method for providing access to this object pool can be provided, and the details of the DBFooMgr are hidden away within the implementation of the Foo class; this can be done with Strong Layering.
- Should the presentation objects know about the concept of persistent storage and the fact the objects need to be (re)created from some kind of database? If so, then use Weak Layering and let the interface know about DBFooMgr.
- Should the presentation objects be independent of the business objects and the concept of a persistent storage? If so, then use the Entity Broker approach to encapsulate their interactions.

Acknowledgments

We would like to thank our "shepherd" Jens Coldewey for his insightful and helpful comments.

References

[Aars96] A. Aarsten, D. Brugali, G. Menga, "Patterns for Three-Tier Client/Server Applications," In *Pattern Languages of Programs (PloP),* Monticello, Illinois, 1996.

[Agar95] S. Agarwal, C. Keene, and A. Keller, "Architecting Object Applications for High Performance with Relational Databases," In *OOPSLA Workshop on Object Database Behavior, Benchmarks, and Performance*, Austin, TX, October 1995.

[Brow96a] K. Brown, and B. Whitenack, "Crossing Chasm: A Pattern Language for Object-RDBMS Integration," In J. Vlissides, J. Coplien, and N. Kerth (eds.*), Pattern Languages of Program Design 2,* Addison-Wesley, 1996, pp. 227-238.

[Brow96b] K. Brown, "Crossing Chasm: The Architectural Patterns," In *Pattern Languages of Programs (PloP),* Monticello, Illinois, 1996.

[Busc96] F. Buschmann, R. Meunier, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture: A System of Patterns,* John Wiley & Sons, 1996.

[Duhl96] J. Duhl, "Integrating Objects with Relational Data, " In *Object-Magazine*, SIGS Publication, March 1996, pp. 89-90.

[Gamm95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[Hirs96] R. Hirschfeld, "Three-Tier Distribution Architecture," In *Pattern Languages of Programs (PloP),* Monticello, Illinois, 1996.

[Keat95] G. Keating, and J. Thomas, "The Winning Combination: Object/Relational Solutions," In *Object-Magazine*, SIGS Publication, September 1995, pp. 64-67.

[Kell93] A. Keller, R. Jensen, and S. Agarwal, "Persistence Software: Bridging Object-Oriented Programming and Relational Databases," In *ACM SIGMOD*, May 1993.

[Kell96] W. Keller, and J. Coldewey, “Relational Database Access Layer,” In *Pattern Languages of Programs (PloP),* Monticello, Illinois, 1996.

[Swam97] V. Swaminathan, and J. Storey, “Domain-Specific Frameworks,” In *Object-Magazine*, SIGS Publication, April 1997, pp. 53-57.

[Your95] E. Yourdon, K. Whitehead, J. Thomann, K. Oppel, P. Nevermann, *Mainstream Objects: An Analysis and Design Approach for Business*, Prentice Hall, 1995.