Multilayer Class

Jens Coldewey, Wolfgang Keller
sd&m – software design & management GmbH & Co. KG
Project ARCUS[1], Thomas-Dehler-Str. 27 – D 81737 München - Deutschland
used to be http://www.sdm.de/, Authors can now be found at http://www.objectarchitects.de/ and http://www.coldewey.com/

Multilayer Class **Last Resort**

Abstract

Multilayer Class deals with the connection of adjacent layers in a layered architecture [BMR+96, pp.31] when using statically typed languages. Standard coupling techniques, such as simple calls, Envelope-Letter, or Observer, do well in most situations. However, when the two layers have closely coupled life cycles, the standard solutions may fail. Here, Multilayer Class offers a last resort.

Example

Consider a persistent class *Patient* in a medical laboratory management system. A Patient has several domain methods to assign test results, to change the attributes of the Patient, or to get a list of results, assigned to the Patient. These tasks are domain specific. Since the Patient is a persistent object, there are also technical tasks: The database management system (DBMS) needs to know when a Patient has changed, for example. If you store the Patient in a relational database you need methods that construct a database record out of the attributes and vice versa. Object databases require methods to control clustering of objects in the database, according domain-specific considerations. These problems usually are solved using a layered architecture [BMR+96, pp.31]: A domain kernel, specific to the application domain, and a database access layer, providing persistence services. The domain kernel may call the access layer but not the other way round.

Before presenting the Multilayer Class, we shall discuss three standard ways to separate the layers, each of them hiding pitfalls[2]:

1. Wrap the database-specific code with a Facade [GHJ+94].

---

1 This work is sponsored by the German Ministry of Research and Technology under project name ENTSTAND.

2 Database Broker [BWh95] is another way to achieve persistence. We shall discuss this pattern in the Related Patterns section.

2. Use the Envelope-Letter idiom [Cop92].

3. Use an Observer pattern [GHJ+94].

Let’s start with the Facade. The idea of this approach is to encapsulate the database-specific code in a set of classes. Though this seems to be simple, it does not solve the problem. Because the Facades do not know domain-level attributes, it is the job of the application classes to prepare their attributes for the Facades or to get them from there. Consequently, most of the database-specific code remains in the upper layer.

As simple Facades do not work, we may try an Envelope-Letter idiom as shown in Figure 1. The Envelope contains all the domain-level methods while the Letter owns the attributes and the database access code.

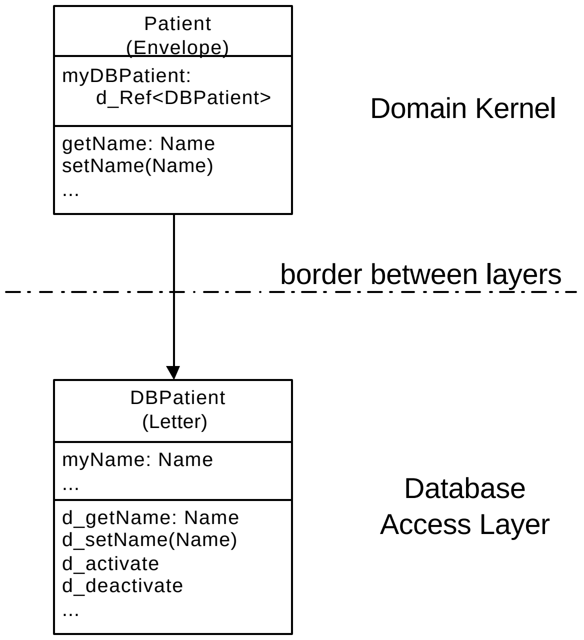

Figure 1: Layer separation using Envelope-Letter. The setName and getName methods of the Patient delegate calls to their database access peer, which owns the attributes.

Now, consider the life cycles of the two classes (Figure 2). Both classes have similar life cycles, but the Letter has an extra state *residentInDatabase*. This state indicates that the Letter currently does not reside in main memory, but on the database. If a client queries for a certain Patient, the access layer *activates* an object: it loads a DBPatient into main memory. To form a complete Envelope-Letter, the access layer also has to instantiate a Patient object. This forces a call into the domain kernel - a violation of the layering idea.

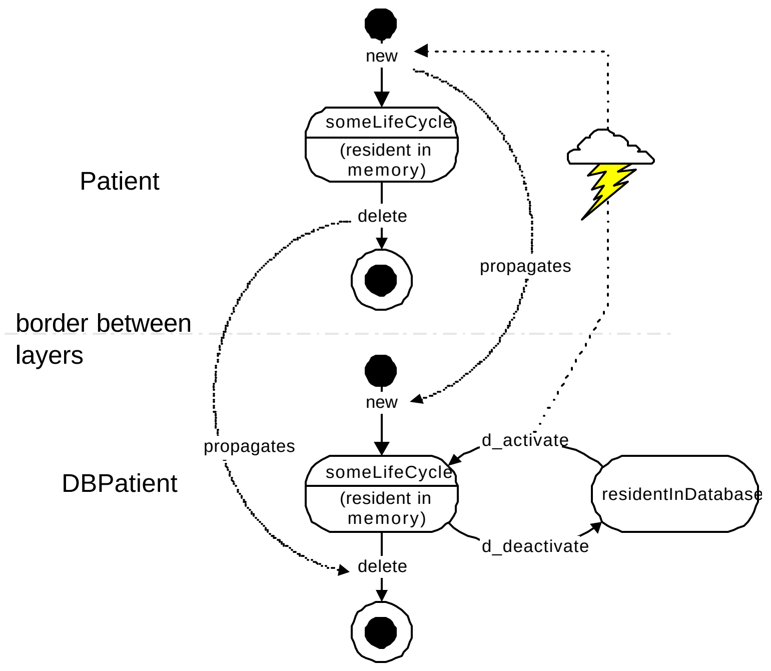

Figure 2: The life cycles of the Envelope-Letter solution depicted in Figure 1. Note, that Patient and DBPatient both have the same life cycle. However, the database access layer has an additional state residentInDatabase denoting that the object is currently not loaded into memory. A transition out of this state means to instantiate a new Patient object and to restore it to the state of DBPatient. This forces an illegal upward call, tagged with a thunder cloud.

Events propagating upwards suggest to use an Observer [GHJ+94]. Figure 3 shows Observers that watch for the activation of letter objects in the access layer. If they observe one, they create a domain kernel object and connect it to its corresponding letter.

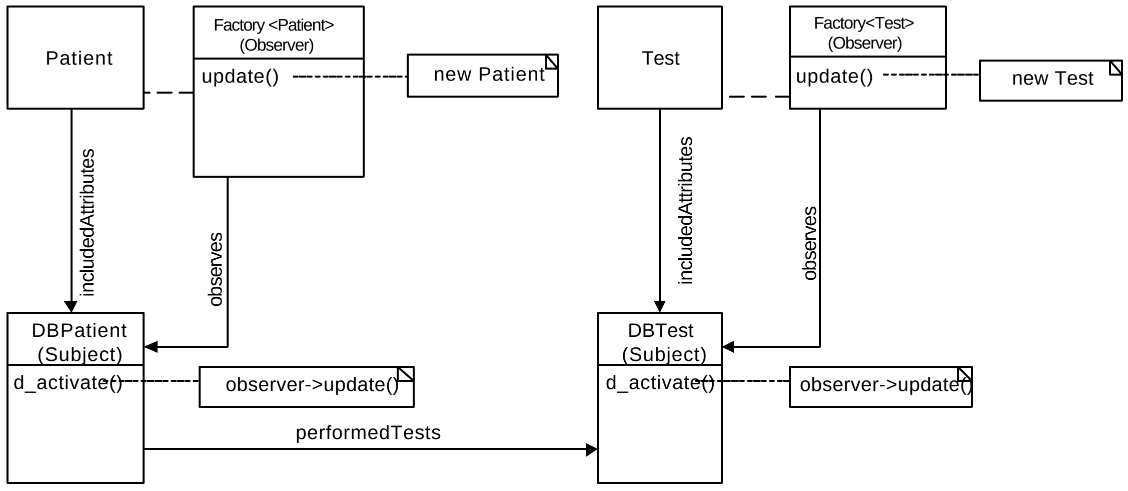

Figure 3: Separating the layers using an Observer, which watches for the activation of database access layer objects. If the access layer activates an object, it informs the Observer which in turn instantiates a new domain kernel object. Using this approach clutters up the design like this figure.

This design not only is fairly complex; it still has a flaw. Suppose you want to know all the Tests assigned to a certain Patient. If you request a list of Tests from the Patient, you expect it to return Test classes. Instead the Patient delegates the request to DBPatient, which has to return a Test object consequently. But it should not know Tests, only DBTests - again a violation of layering (Figure 4).

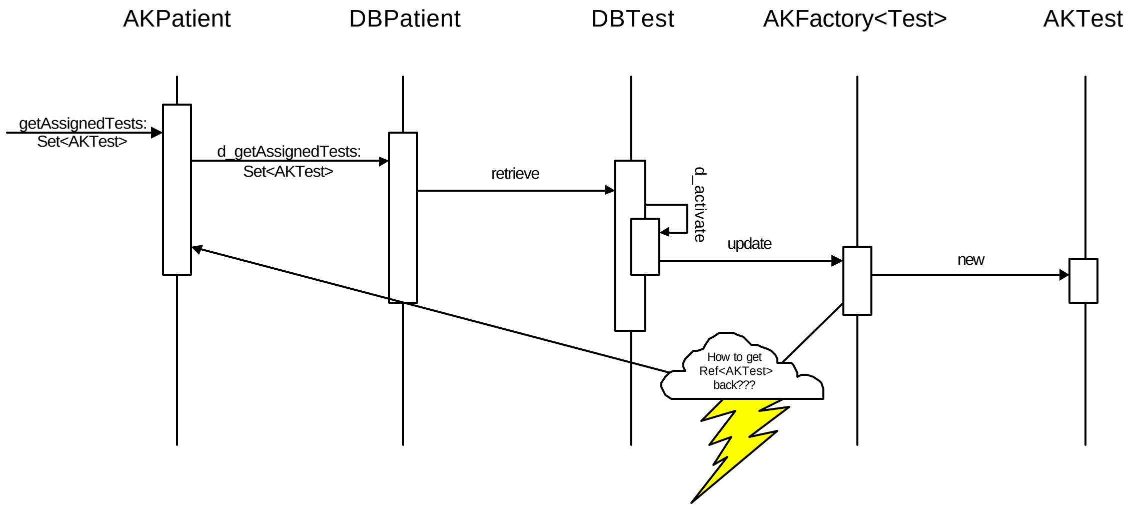
AKPatient
DBPatient
DBTest
AKFactory<Test>
AKTest
getAssignedTests: Set<AKTest>
d_getAssignedTests: Set<AKTest>
retrieve
d_activate
update
new
How to get Ref<AKTest> back???

Figure 4: Problems with the Observer: It causes trouble to return a reference to an object that the database access layer activates.

Context

You are developing a layered system [BMR+96, pp.31]. Designing the interface between the two layers you find that many objects of the upper layer have corresponding classes in the lower layer. These pairs have many attributes in common, their life cycles are closely coupled, and events in both layers may change the state of the upper layer object. Standard separation techniques do not work without upward calls or they result in a design of inadequate complexity.

You have carefully checked, whether the intended layer separation really is the correct place to split. This is usually true if the lower layer peers provide some service while the upper layer peers implement domain knowledge.

Problem

How do you design the communication between the layers without extraordinary effort?

Forces

?? *Preserving the layered structure:* The idea of layering forbids direct calls from lower layers to upper layers. This ensures that different teams can develop each layer and that you can compile and test the lower layer separately. Violating layering may raise the effort for integration and tests significantly.

?? *Upward propagation:* Propagating changes from the lower layer to the upper one needs special decoupling to preserve layering. The adequate decoupling technique depends on the frequency of changes and whether the change should propagate immediately or may wait. If the lower layer instantiates or destroys higher level objects, callback mechanisms may not work.

? Decoupling upward propagation is particularly difficult in statically typed languages, because compilers require the definition of all the classes you call when compiling a class. Therefore, upward calls cause upward compile time dependencies you have to cut.

?? *Complexity:* A complex interface is expensive to develop, hard to use, and hard to maintain. If many upper layer classes access the lower layer, the cost of a complex interface multiplies and may even spoil the gain of layering.

Solution

Extend a single class over both layers, forming a *Multilayer Class*. Unambiguously assign every member of the class to one layer, using a naming convention. Relieve the Multilayer Class from as many lower layer responsibilities as possible and encapsulate them in separate classes. Prefer to use call dependencies over inheritance.

Structure

Figure 5 shows the structure of a Multilayer Class. The Multilayer Class has members of both layers, which offer the absolute minimum of services. GeneralProtocolOfLowerLayer and LowerLayerServices offer the remaining services of the lower layer via inheritance and use dependency, respectively.

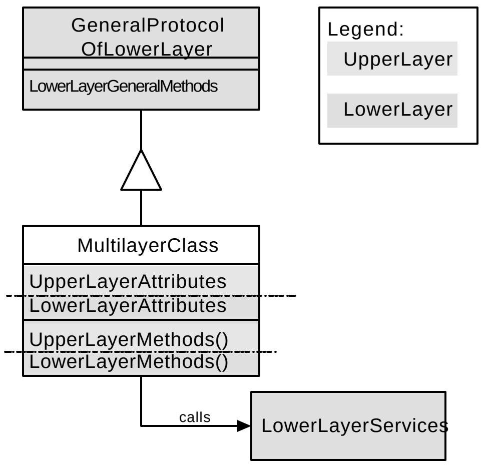

Figure 5: Structure of the Multilayer Class. Shaded parts denote methods and attributes of the upper layer, dashed parts show lower layer members.

?? **GeneralProtocolOfLowerLayer**

?? supplies attributes and methods that all Multilayer Classes need to provide services of the lower layer

?? **MultilayerClass**

?? contains services and attributes of both layers. The attributes of the upper layer are also accessible for methods of the lower layer. Every service is unambiguously assigned to a single layer.

?? inherits common services of the lower layer from GeneralProtocolOfLowerLayer and implements them if necessary.

?? **LowerLayerServices**

?? supply all generic services of that layer. Hence, the Multilayer Class implements only a small part of the lower layer functionality. Either a single class or a complete subsystem may supply LowerLayerServices.

Implementation Issues

?? *Preserve the idea of layers when implementing the methods*. Lower layer methods should not call methods of the higher layer. Violating this principle will make it impossible to change the lower layer without knowing anything about the upper layer. This would spoil the intent of layers. Choosing a solution that uses different classes, the compiler detects violations. With the Multilayer Class you have to check it during code reviews.

?? *The lower layer part of the MultilayerClass should be as small as possible* because the separation of the two layers is weak. Ideally, it should only adapt to LowerLayerServices. As a rule of thumb there are two good candidates for the lower layer methods of a MultilayerClass:

?? Methods that *need* direct access to attributes and that influence the life cycle, such as d_activate or d_deactivate in Figure 3.

?? Methods specific to this class, such as d_locateObject, which determines the database cluster to which an object belongs in an object database.

?? *Keep the interface of the Multilayer Class clean*. Upper layer methods of Multilayer Classes should not call lower layer methods of *other* classes. Violation will result in a relaxed layered architecture. Relaxing layers should not be an act of coincidence and negligence but a result of sound design considerations. Whether lower layer methods of different Multilayer Classes may call each other is worth a discussion. Prefer a solution without inter-class calls, because these calls disregard all domain-specific checks.

?? *Find good naming conventions*. One of the main ideas of the pattern is the unambiguous assignment of methods and attributes to one of the layers. The easiest way to do this is using good naming conventions. A good solution is a layer prefix such as 'd_' for a database access layer. Again code reviews should check consistency.

?? *Think about using templates when the lower layer is generic*. If you have a language that supports templates you may be able to cope with boring code in a better way than generating it.

Consequences

?? *You may develop the layers separately:* You can split the work on the two layers between different sub-teams, if the lower layer is generic. Provided, you can generate the lower layer parts of a Multilayer Class, one team may develop the generators or templates while the other may concentrate on the upper layer. If your compiler allows to use separate files for the implementation of different methods, you may even separate the teams when both layers need manual work. However, most compilers do not allow you, to put different methods of a class into different linkage modules. Therefore, putting the different layers into separate linkage modules causes problems. It is also not feasible, to separately test the layers. Consequently, the pattern provides only weak decoupling of the layers. Still, decoupling is good enough if the lower layer is generic.

?? *Status changes propagate upwards:* Because both layers use the same physical attributes, construction and deletion are performed for both layers at once. Consider the activation of an object stored in an object database as an example: When the database system loads the object into main memory, both layers are ready to work without any further action.

?? *Complexity is low:* The pattern merely is a coding convention. Therefore it is simple to understand and to maintain.

?? *The pattern works best when you generate one layer automatically*: Frequently the lower layer is generic. For example, a database access layer is mostly boring code, which performs some database actions and returns. Changes may apply during database tuning, but the first cut of this layer derives from the attributes of the class. If you have the attributes in a 'parser friendly' format, you better write a generator that generates one layer automatically.

?? *This is a last resort pattern!* Though the solution balances the forces quite well, it is not taken from the Hall of Fame of object-oriented design. If you use a Multilayer Class, your object model is not compatible with the layer structure of the system. Hence, your system structure is not hierarchical any more. However, using a Multilayer Class is better than no layering at all. Decide for a Multilayer Class much as a surgeon decides to amputate a limb: Use it only when you have double-checked that there really is no better solution.

Example Resolved

Lets return to our introductory example. As Figure 6 shows, we have annexed database methods to the Patient, forming a Multilayer Class. These methods care for object instantiation, loading from the database and retrieval. According to [ODMG96] the names of the database access members have "d_" as prefix.

Furthermore, the Patient inherits an object id from a common database access class, called d_Object, which also defines abstract methods. Other database access classes provide support, such as transaction control and error management.

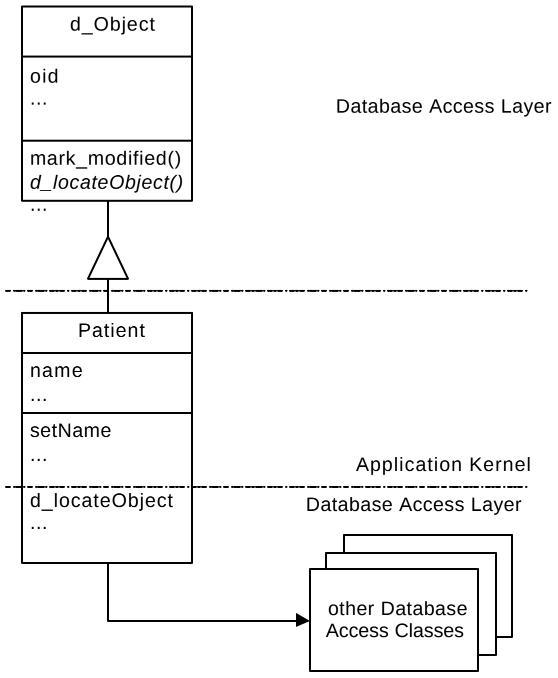

Figure 6: The Patient as a Multilayer Class using ODMG style object names

Variants

?? *Spanning more than two layers*. Theoretically you may span a Multilayer Class over more than two layers. However, the cases where the context fits to this pattern are very rare. We could neither find nor even imagine a situation where you encounter all the listed problems at several layer borders at once.

Related Patterns

Layers [BMR+96] defines a layered architecture and lists forces and consequences of their usage.

The (Database) Broker pattern [BWh95] offers a better separation of layers to implement persistence in dynamically typed languages. It contains a separate Broker class, which knows how to instantiate domain kernel objects and how to fill the attributes. However, to protect the layered structure, the Broker has to retrieve the class layout and corresponding method names of the domain kernel object during runtime. This is easy to implement in dynamically typed languages such as Smalltalk. Statically typed languages usually lack this feature. Simulating it in C++ is possible, but it requires a tricky, complex design. Paying this effort may be reasonable when the layer separation also needs to be a class separation, for instance to form a client/server cut.

Facade [GHJ+94] and Envelope-Letter [Cop92] are popular patterns to separate subsystems. They are a good choice whenever changes in the lower layer do not affect the status of the upper layer. Another alternative to separate layers is the Observer [GHJ+94] pattern or Model View Controller [BMR+96]. Their decoupling is better but it yields in a complex design, if the Observer also has to care for instantiation.

Another way to add functionality to a class is Extension Objects [Gam96]. They extend the interface of an object to support further services using aggregation. As an Example you may add a spell checker interface to a text processor object. However, the pattern introduces additional complexity to the class: "An extended interface is more complicated to use than one which is provided by the subject itself. A client has to query for the interface and check whether it exists" [Gam96]. Additionally they do not support instantiation of a new object from an interface extension.

Generation Gap [Vli96b] uses inheritance to extend the functionality of a class. This pattern describes a generated super class offering hooks for customization. In contrast, typical Multilayer Classes have generated parts that depend on the attributes of the non-generated classes.

Known Uses

The Multilayer Class pattern was one of the key design decisions in the ChaMPs project, which designed an object oriented access layer for RDBMS [Hah+95]. Special tools generated the database access methods, using a semiformal description of the classes. Subsequent projects of sd&m adapted this pattern for database access, such as the HYPO Project [Kel+96] or EASY, an order management system for DeTeMobil.

Some C++ bindings of Object Databases use Multilayer Classes. It is an optional part of the C++ language binding, contained in the ODMG standard ([ODMG96, Section 5.3.4]).

With Objectivity/DB every persistent class has to inherit database behavior from the class ooObject [Obj92]. The Object Definition Language contains information about associations. A preprocessor analyzes the definition and generates additional member methods to manipulate the associations.

The Microsoft Foundation Classes use this pattern to store objects in files and to collect debug information: You derive every class from a base class called CObject. Beside others, the base class defines two methods called CObject::Serialize and CObject::Dump. The former stores or retrieve an Object from a special type of stream while the latter dumps the class contents to a debugging subsystem. You can consider both methods members of lower layer services, which deal with archiving and debugging support, respectively.

Similarly, the CORBA Persistence Object Services [Ses96] uses this pattern to interface to the Persistent Data Service (roughly an abstraction of a database). The architecture uses the CosStream::Streamable class to write or read persistent objects to a stream, which interfaces to the Persistent Data Service. If you interpret the externalize_to_stream and internalize_from_stream methods as part of the persistence mechanism, this is a Multilayer Class.

Acknowledgments

We thank *Bobby Woolf*, our PLoP shepherd, who gave us many valuable hints to improve the paper. The participants of the PLoP writer's workshop gave us a lot of useful comments and encouraged us to proceed[3].

Chad Smith and *John Vlissides* helped us to improve the language. *Gudrun Öchsl, Andreas Mieth, and David van Camp* also provided helpful feedback. Thanks to you all.

References

[BMR+96] **Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal**: *Pattern-Oriented Software Architecture - A System of Patterns;* John Wiley & Sons Ltd., Chichester, England; 1996, ISBN 0-471-95869-7

[BWh95] **Kyle Brown, Bruce G. Whitenack**: *Crossing Chasms- A Pattern Language for Object-RDBMS Integration*; Knowledge Systems Corp, Cary, North Carolina, 1995

[Cop92] **James O. Coplien**: *Advanced C++ - Programming Styles and Idioms*; Addison-Wesley Publishing Company, Reading, Massachusetts, 1992; ISBN 0-201-54855-0

[Gam96] **Erich Gamma**: *The Extension Objects Pattern;* In PLoP'96 - Proceedings (section 5.3); University of Illinois at Urbana-Champaign, 1996

[GHJ+94] **Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides**: *Design Patterns - Elements of Reusable Object-Oriented Software;* Addison-Wesley Publishing Company, Reading, Massachusetts, 1994; ISBN 0-201-63361-2

[Hah+95] **Wolfgang Hahn, Fridtjof Toenniessen, Andreas Wittkowski**: *Eine objektorientierte Zugriffsschicht zu relationalen Datenbanken*; In Informatik Spektrum 18(Heft 3/1995); pp. 143-151, Springer Verlag 1995

[Kel+96] **Wolfgang Keller, Christian Mitterbauer, Klaus Wagner**: *Objektorientierte Datenintegration über mehrere Technologiegenerationen*; Proceedings ONLINE, Kongreß VI, Hamburg 1996.

[Obj92] **Objectivity, Inc.**: *Objectivity Programming Tutorial - Version 2.0*, Objectivity Inc., Mountain View, California, 1992

3 *Kyle Brown* contributed the "Last Resort" category, *Ralph Johnson* the amputation metaphor.

[ODMG96] **Rick G. G. Catell (ed.), et. al.:** *Object Database Standard: ODMG-93 - Release 1.2;* Morgan Kaufmann Publishers, San Mateo, California, 1996; ISBN 1-55860-396-4

[Ses96] **Roger Sessions:** *Object Persistence - Beyond Object-Oriented Databases*; Prentice-Hall International, New Jersey, 1996; ISBN 0-13-192436-2

[Vli96b] John **Vlissides:** *Generation Gap*; In C++ Report Nov/Dec 96; SIGS Publications, New York; ISSN 1040-6042 (to be published)