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
Email: {jensc, wk}@sdm.de

Multilayer Class **Class Structural**

Intent

Separate two layers of functionality in objects that share the same attributes and life cycles. Since statically typed languages make this difficult to accomplish at class level, perform the separation at the method level within a single class.

Motivation

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 belong to the application kernel. Since the Patient is a persistent object, there are other tasks more technical in nature. The DBMS (either an OODBMS or an RDBMS) needs to know when a Patient has changed, for example. If the Patient is stored in a relational database there will be 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 are typical problems usually solved by a layered architecture: An application kernel that contains code specific to the application domain and a database access layer, used to provide persistence related services.

We will discuss three standard ways to separate the layers[2], each of them hiding certain pitfalls:

1. Wrap the database specific code with a Facade [GOF95].

2. Use the Envelope-Letter idiom [COP92].

3. Use an Observer pattern [GOF95].

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. It will be discussed in the Related Patterns section.

Let's start with the Facade. The basic idea is to encapsulate the database specific code in a set of classes, perhaps included in a class library. Though this seems to be appealingly simple, it does not solve the problem. Since the Facades are unaware of the domain level attributes, it is the job of the application classes to prepare their attributes for the Facade or to get them from there. As a result a large part of the database specific code remains in the domain level objects.

However, as simple Facades do not work, we seem to need a specific Facade for every domain level object. This leads to the Envelope-Letter idiom as shown in Figure 1. The Envelope contains all the domain level methods while the Letter owns the attributes and other parts the database needs.

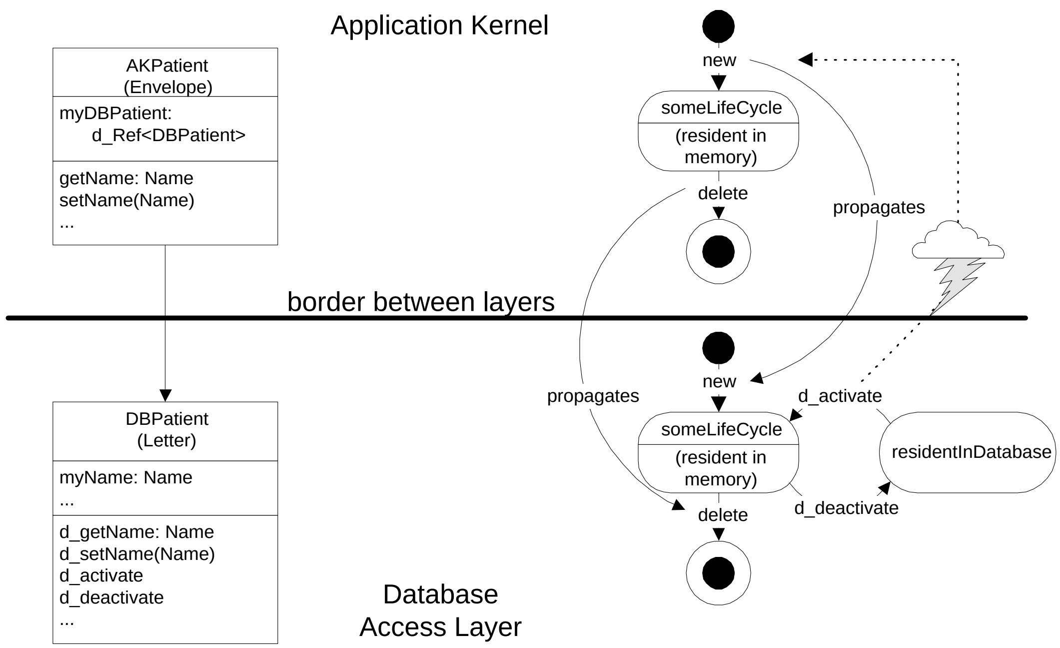


Figure 1: Layer Separation Using Envelope-Letter. The technical methods of the database access layer are usually inherited. The right side illustrates the corresponding life cycles.

Now, consider the life cycles of the two classes, sketched on the right hand side of Figure 1. Both classes have corresponding life cycles, but the Letter augments it with an extra state, called '`residentInDatabase`'. This state indicates that the Letter is currently not loaded into main memory, but resides on the database. For some reason the access layer may decide to activate an object, that is load it into main memory. The trigger may be a query for a certain Patient, for instance. To form a complete Envelope-Letter, the Envelope also has to be instantiated, which forces a call from the access layer into the application kernel - a violation of the layering idea.

This problem suggests, to use an Observer **[GOF95]** that triggers creation of application kernel objects on activation of their corresponding letters. Figure 2 shows the resulting design with `AKTests` associated to an `AKPatient`.

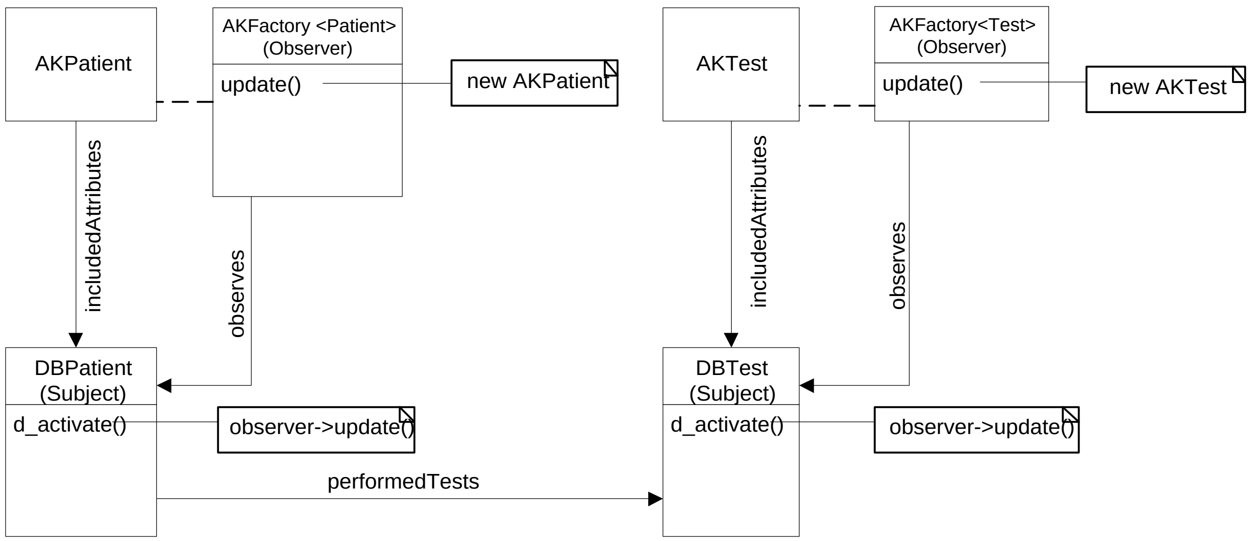


Figure 2: Separation of layers using the Observer pattern.

This design is not only fairly complex; it still has a severe design flaw. Suppose you want to know all the Tests assigned to a certain Patient. If you ask the `AKPatient` to give a list of Tests, you expect it to return `AKTests`. Instead the patient gets the Test reference from `DBPatient`, which only knows `DBTests`, but no `AKTests` - once again a violation of layering (Figure 3).

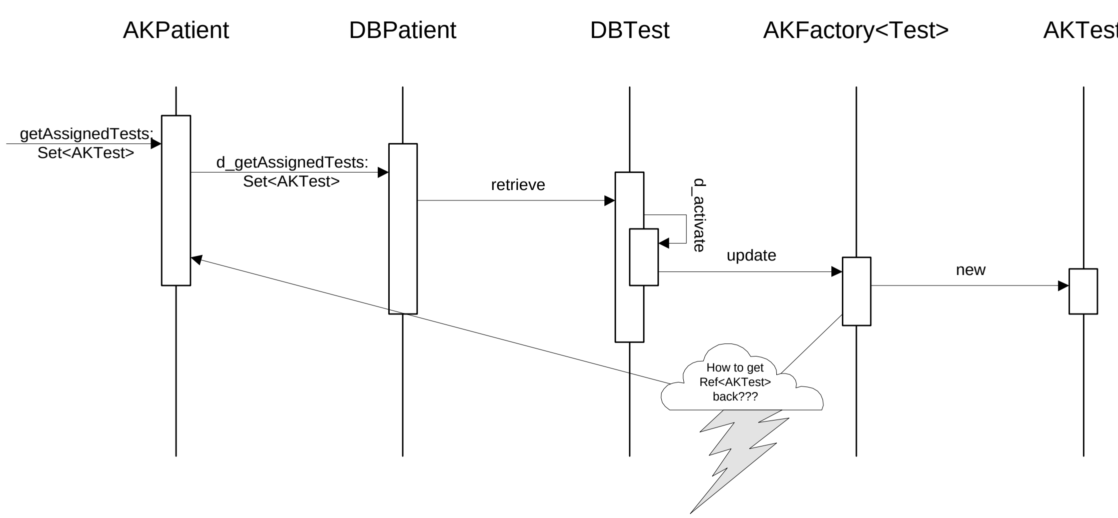


Figure 3: Problems with the Observer: Returning a reference to an object, activated by the database access layer, causes trouble.

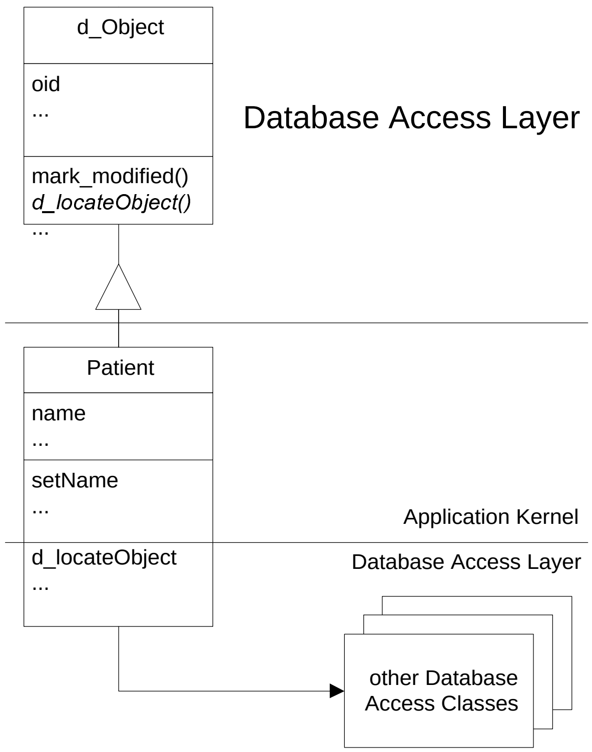


Figure 4: The Patient as a Multilayer Class using ODMG style object names

The Multilayer Class solves these problems by spawning a single class over more than one layer. Yet to preserve the layered structure of the system, every attribute and every method of the class is unambiguously assigned to a certain layer. Hence, the Patient has domain specific methods like `setName` and methods of the database access layer like `d_locateObject`. An attribute or a method is assigned to a specific layer using a naming convention. In addition, domain classes may inherit services of the database layer from a persistence superclass. According to **[ODMG96]**, this class is named `d_Object` in Figure 4.

It is essential, not to put *all* database specific code into the database methods of the Patient. This would result in large classes polluted with SQL code or other database specific stuff. Generic classes should contain all generic tasks; the Patient's database methods should only prepare calls to these generic classes.

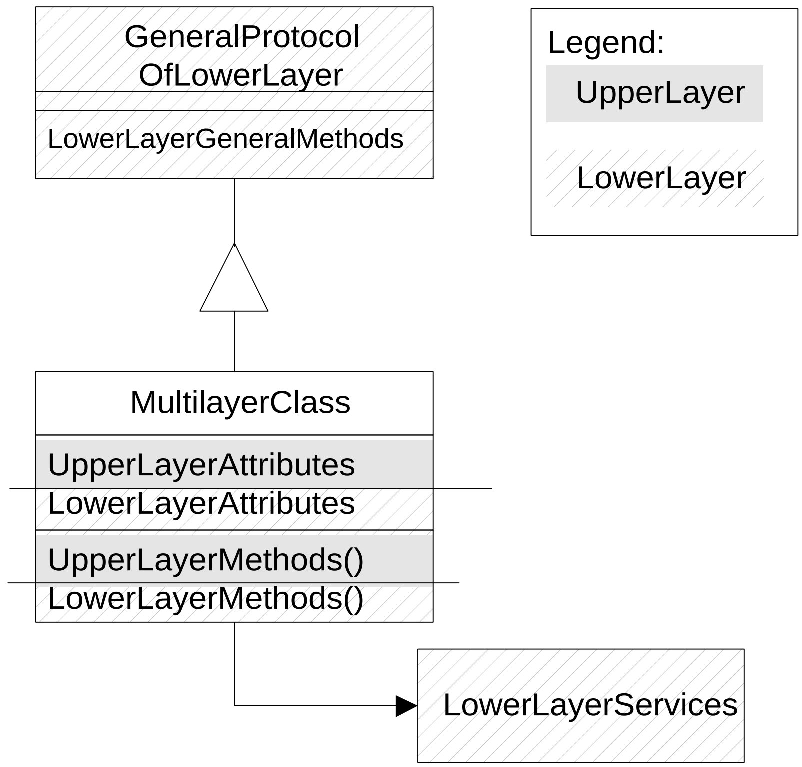
Applicability

Use the Multilayer Class pattern when:

- a class has representations in two layers of a software system
- the attributes and the life cycles in both layers are tightly coupled
- one of the layers is rather generic in nature
- any other way to model layer separation results in complex protocols between the layers.

Do *not* use this pattern, if your language offers enough run-time type information to solve the problem with Brokers (see page 7).

Structure



Participants

- **`GeneralProtocolOfLowerLayer`** (`d_Object`)
  - supplies attributes and methods that all Multilayer Classes need to provide services of the lower layer
- **`MultilayerClass`** (`Patient`)
  - 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.
- **LowerLayerServices** (Other database access classes)
  - Supplies all generic services of that layer. Either a single class or a complete subsystem may supply LowerLayerServices.

Collaborations

- The `MultilayerClass` inherits common services of the lower layer from `GeneralProtocolOfLowerLayer` and implements them if necessary.

- LowerLayerServices offer all generic services of this layer, needed by the Multilayer Class. Hence, the Multilayer Class implements only a very small part of the lower layer functionality.

Consequences

The Multilayer Class implies the following consequences:

- *The life cycles of both layers are coupled*. Because both layers use the same physical object, 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 DBMS loads the object into main memory, both layers are ready to work without any further action.

- *The different layers are technically not separated*. It causes problems to put the different layers into separate linkage modules. Anyway it is easy, to split the work on the two layers between sub-teams, using separate files to implement the different methods, for example.

- *The lower layer part of the `MultilayerClass` should be minimal.* Since the separation of the two layers is weak, the lower layer share of a `MultilayerClass` should be as small, as possible. Ideally, it should only contain code to prepare calls to LowerLayerServices. As a rule of thumb there are two types of methods, that are good candidates for the lower layer methods of a `MultilayerClass`:

    – Methods *needing* direct access to attributes and influencing the life cycle, such as `d_activate` or `d_deactivate`.

    – Methods specific to this class, like `d_locateObject,` which determines the database cluster, an object belongs to in an object database.

- *The pattern works best when one layer is generated automatically*. Frequently one of the layers is rather invariant. A database access layer, for example, is mostly boring code that performs some database layer calls and returns. Changes may apply during database tuning, but the first cut of this layer derives from the attributes and associations of the class. If this information is available in a ‘parser friendly’ format, it is easy to write a generator that generates one layer automatically.

Implementation Issues

Consider the following issues when using this pattern:

- *Observe the idea of layers when implementing the methods.* Methods of the lower layer should not call methods of a higher layer. Violating this principle will make it impossible to change the lower layer without knowing anything about the upper layer. This would contravene the intent of the pattern. Choosing a solution that uses different classes, you can ensure that items of the upper layer are out of the naming scope of the lower layer and thus let the compiler find violations. With the Multilayer Class you have to check it during code reviews.

- *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 layer prefix such as ‘d_’ for a database access layer is a popular solution. Again code reviews should check consistency.

- *Think about using templates when the lower layer is rather 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

Related Patterns

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

Facade **[GOF95]** and Envelope-Letter **[Cop92]** are popular patterns to separate subsystems. They are a good choice, if changes in the lower layer never affect the status of the upper layer..

Another alternative to separate layers is the Observer **[GOF95]** pattern or MVC. Their decoupling is better but it still has the coupled life cycles problem.

Known Uses

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

Some C++ bindings of Object Databases use Multilayer Classes. It is an optional part of the C++ language binding of 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 C++ header files are generated from the Object Definition Language (ODL) that is an extension of C++. The ODL contains information about associations. The pre-processor generates additional member methods to manipulate the associations.

Acknowledgments

The authors like to thank *Bobby Woolf*, our shepherd, who gave us many valuable hints to improve the paper. *Chad Smith* helped us to improve the language. *Andreas Mieth* and *Gudrun Öchsl* also provided helpful feedback. Thanks to you all.

References

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

[Cop92] **Jim Coplien:** *Advanced C++ - Programming Styles and Idioms*, Addison-Wesley 1992.

[GOF95] **Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides**: *Design Patterns, Elements of Reusable Object-oriented Software*, Addison-Wesley 1995.

[Hah+95] **Wolfgang Hahn, Fridtjof Toenniessen, Andreas Wittkowski**: *Eine objektorientierte Zugriffsschicht zu relationalen Datenbanken*, 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

[ODMG96] **Rick G. G. Cattell (Ed.) et. al.:** *Object Database Standard: ODMG-93 - Release 1.2* Morgan Kaufmann Publishers, San Mateo, California, 1996