A Collection of Patterns for Object-Oriented Databases

Author: Manfred Lange, IT Consultant
Hewlett-Packard, NSL (Network Support Lab)
Herrenberger Strasse 130, 71034 Boeblingen, Germany
E-Mail: Manfred_Lange@acm.org

Abstract

Object-oriented database systems (ODBMS) have been around for a while. Still, it does not appear that they have found widespread use. Many applications are still developed base on relational database systems. For these, pattern languages have been developed over a couple of years ([Keller 1995] and [Keller 1998]).

For object-oriented database systems however dedicated papers describing related patterns are rare. This paper tries to fill the gap. At least to some extend, as it does not provide a complete set of patterns, but only a collection.

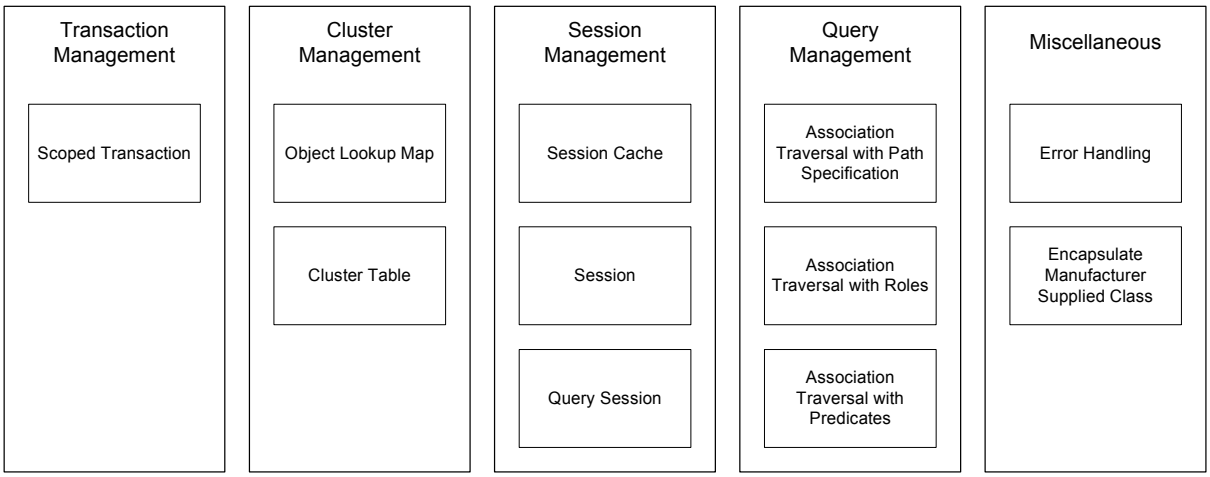
All patterns presented here have been successfully applied to different projects, having built a solid backbone for high availability solutions.

Introduction

In this paper, I use the following categories for the patterns for object-oriented database applications:

- Transaction Management
- Cluster Management
- Session Management
- Query Management
- Miscellaneous

The following picture gives an overview of the patterns described in this paper:



The patterns described in this paper are derived from our work in the area of database access layers. We typically do not allow any client, who typically is another software component, to directly access the database system. There are two reasons for this: first, we do not want to distribute database system or product related code throughout the complete system. Second, we want to be able to arbitrarily change the database schema without being forced to also change all clients. Although I do not want to discuss advantages or disadvantages of one programming language versus others, we have decided that for a back end database access layer, e.g. a component running on a (web-)server, C++ is the best choice regarding resource consumption during runtime. This is the reason why some of the patterns presented here will not necessarily work with other programming languages.

This paper assumes that you have decided to use an object-oriented database system (see appendix for a lightweight introduction to object-oriented databases). However, you can find material for making this decision in [Coldewey].

In closing this introduction, I would like to issue one warning: If you implement an application using an object-oriented database system, do not simply apply all of the patterns described in this paper. The better approach is to use always the database features first. Only if they do no fit your requirements, you should select a pattern from this paper.

Scoped Transaction

Motivation

Providing a consistent view to the objects in the database is the main reason for transactions. However, in conjunction with clustering[1] a transaction can also help to implement efficient caching strategies.

Transactions should be kept as short as possible as they determine the lifetime of locks. In order to keep the throughput high, locks should be released as soon as possible.

In addition, once a transaction has begun, it should not be left uncommitted or aborted for a longer period of time.

Furthermore, for a failsafe operation every call for starting a transaction should be paired with a commit or an abort (rollback) of the very transaction.

In short: How do you guarantee that you close an opened transaction again in all circumstances?

Forces

- All started transactions must be either committed or aborted (rolled back).
- No client must hold locks longer than necessary, because
    - Throughput of the database must be maximized. (Number of transactions per second)
    - The number of concurrent users must be maximized.
- Transactions should be easy to handle, e.g. automatically commit, or roll back a transaction.
- Transactions must be finished within a predefined timeframe, e.g. within 30 seconds.

---

[1] Clustering is a technique of physically locating related objects together, so that they can be efficiently retrieved from the database especially when they are located on the same page on the disk.

Solution

Use a construct of your programming language that allows you to tie the lifetime of the transaction to the lifetime of a scope or a code block.

If a class were the construct you have chosen, then you would create a new instance at the beginning of the block. When the block is left, the instance is destroyed. The destructor of the class tries to commit the transaction, and if this did not work, the destructor would abort the transaction.

Consequences

1. All transactions are handled with pairs of either Start()-Commit() or Start()-Abort(). This ensures that no transaction is left open.

2. As no transaction is left open, no lock is left on any object in the database.

3. Transactions are easy to handle, as you only define a local variable. When it goes out of scope it automatically either commits or aborts the transaction.

4. Given 1 and 2, Scoped Locking contributes to a higher throughput and availability of the database system

Implementation Issues

For C++ this means introducing a class for handling transactions. The class should have at least the following interface.

```
class Transaction {
public:
  int Start();
  int Commit();
  int Abort(); // aka rollback
  bool IsActive();
private:
  bool m_bIsActive;
  ooTrans[2] m_objyTrans;
};
```

---

[2] This is the class of the ODBMS. However, there is no support for automatically committing or aborting transactions.

When a client wants to start a transaction, it creates an instance of class Transaction on the stack. Then the client has to call Start() on the object. Finally, after all database calls have been done, Commit() has to be called. This is not the tricky part.

The interesting part starts, when an error occurs, e.g. signaled by throwing an exception, the scope is somehow left, but not the normal way. However, the language guarantees that the destructor of local objects will be called. The destructor can therefore check, whether the transaction has been properly committed, and if not, the transaction can be aborted.

The technique with a scoped object is described for C++ in [Stroustrup].

Known Uses

Multiple Hewlett-Packard network support applications use this pattern, e.g. AutoCollect, Network Documentation Tool (NDT).[3]

Related Patterns

Scoped Locking: This pattern uses the same technique for automatically releasing lock on shared resources.

Object Lookup Map

Motivation

Objects have an identity. Among other things, the identity is used to refer to an object.

However, objects are frequently identified by their location, e.g. in C++ the memory location, the address of an object is used for identifying the object. Many object databases such as Objectivity/DB and ObjectStore use their object identifier to determine the physical location of an object.

This alone does not do any harm. However, ODBMS's may support clustering. In this case, it is possible that an object might be moved from one storage place in the database to a different storage place. In consequence, this means that the object identifier changes its value, as it now refers to a different storage place in the database.

---

[3] At present, all these applications are not available standalone. All of them are part of service or support offerings.

In this situation it becomes necessary to be able to look up an object not only by the object identifier of the database, but to have a constant identifier that does not change, even when the object changes its location.

One might think, that the database system supplied naming service might be sufficient for solving this problem. Each object would be stored with an additional name. This feature is meant for attributing a relatively small subset of all objects with a name. This kind of object is typically called a root object. They are normally used to start the navigational access to the database. However, the naming feature does not scale well, if you want to access ALL objects in this way.

In short: How do you locate objects with a unique identifier, if the database systems naming feature is not fast enough for accessing ALL object in this way?

Forces

- You need to identify objects independent their location.
- You need a fast way to locate an object
- Given a complex object network, navigational access is too slow.
- Using navigational access, objects might be moved because of clustering, before the actual access can be done.
- You need to name ALL objects in the database.
- You need a consistent view of the database.

Solution

Implement a scalable map for mapping unique identifiers to database locations.

Consequences

1. Using the map, the unique identifier is sufficient to find an object in the database, even if the database specific object identifier has changed.
2. Locating object using the unique identifier is easy and fast. It is slower than using the database specific object identifier, as the map must be read. However, once the map has been cached, the overhead is very limited, and might be acceptable for your application.
3. It is not necessary to use navigational access to locate an object. For complex object networks, the use of the object map is much faster.

4. Concurrency issues because of moved objects are reduced (but not eliminated), if the object map is used instead of complex navigational access.

5. ALL objects are uniquely identifiable.

Implementation Issues

You must keep the map consistent. This means that under all circumstances, the entries in the map must reflect the actual location of the objects.

You must update the map, when an object is created, when an object is moved to a different location in the database and when an object is deleted.

In addition, the client must make provisions for ensuring a consistent view on that map. It might be OK for one client to "see" the object at location y. This might be completely wrong for a second client requiring the object to be at location z. The reason is that starting a transaction determines the view of a client on the database contents. This view must be consistent[4]. A different client may update the database contents thus potentially also moving or deleting objects, which the first client has already accessed[5]. In this scenario, the two clients may "see" the very same object at two different locations in the database[6].

The implementation must have a very thorough look at all the places that deal with locking and updating the map. In particular, in a transactional system many race conditions can render the system unusable or unstable. Using synchronization objects such as critical sections or mutexes or database locks is required, but must be reviewed.

The best thing to do is to implement such a lookup map in a class of its own. This helps isolating and testing the race conditions.

Depending on the database product, you may achieve consistency of the map with provisions on the client side or the server side. If the product supports triggers on the server side you may want to consider this the best solution.

---

[4] Consistency is not required for all application. However, the programmer's life will not become easier with inconsistent views, as it requires additional effort, if you want to make updates to the database contents.

[5] If a different client inserts new objects, this is not an issue. The first client does not care for these objects, as they will never be part of the view of the current transaction.

[6] If your database system does not support read locks for objects with an update lock for a different client, then this is not an issue, either.

Known Uses

Multiple Hewlett-Packard network support applications use this pattern, e.g. AutoCollect, Network Documentation Tool (NDT).

Related Patterns

Smart pointers: A smart pointer can be used a reference to an object, without requiring the object to stay at the same location. So a smart pointer has from this perspective the same benefits as the Object Lookup Map. The difference is that smart pointers typically are not automatically updated when a different process changes the location of an object.

Smart pointer could be automatically be updated by database triggers. However, the addition of smart pointers would mean, that you could not use database features, which require passing references to their calls. Smart pointers are not real pointers. They simulate real pointers. For a more detailed discussion on smart pointers in C++ see [Meyers].

Clustering Table

Motivation

As opposed to relational database systems, a client application can explicitly tell the ODBMS where it should physically locate a particular object. This is true to a more or less extent depending on the particular database product. The set of rules that determines which objects are located together is called clustering.

Why would you want to do that? There are at two major reasons for this: performance and lock granularity. I will now look a little more into the details for each of them.

Typically, you do not access single objects in the database. Normally you access sets of objects of different classes, which are associated with each other thus building a net of associated objects. If a client accesses any single one of these objects, there is a high probability that the same client will access some of the other objects as well. Therefore, it makes sense to locate these objects closely together, e.g. on one page of the storage subsystem. If the client accesses one of the objects, the page is loaded with it all the other objects stored on that page.

The other reason for clustering is lock granularity. ODBMS's can be distinguished in two major groups depending on the server component type they have: products using an object server, and products employing a page server. In the former case, the server transmits only one object at a time, whereas a page server always transmits a storage page, potentially containing more than one object. [7]

Accessing an object means is equivalent to acquiring a lock. The lock is managed by a server component called the lock server[8]. If the client accesses many objects, then for the object server type of ODBMS, this would mean many calls to the lock server in order to acquire a lock for each of the objects being accessed. For page servers, the system provides one lock per page[9]. Locking one object means, not only will the object be locked, but the page will be locked and therefore all other objects on the same page.

Both techniques result in a performance increase, if you do clustering properly. You will encounter performance problems, if you do not take care of the clustering.

In short: How do you cluster your objects in the database?

Forces

- You want to have a single place in the code with all information about how objects are being clustered.
- You want to be able to experiment with different cluster strategies, e.g. when you do not know yet the access patterns of the clients.
- You want the clients to be unaware of the actual clustering strategy.
- You want to optimize performance by locating related objects close together.

---

[7] Database system theory also describes servers that can adapt and automatically switch between serving single objects and serving pages. In a typical application, the average performance of the page servers is better. However, if you have a high concurrency requirement, you are still free to cluster only one object per page. The result is a behavior that is (almost) identical to an object server.

[8] Depending on the database product, this may be part of the server component, the client runtime environment, or it could also be a separate process. Objectivity has a separate process that runs on the server.

[9] Objectivity uses the aggregate "container". A lock is acquired on a per-container basis. However, I left out this product detail for simplicity reasons.

- You want to optimize concurrency by NOT locating objects close together. If objects are located close together, they might be protected by just one lock instead of several.

Solution

Implement a Clustering Table.

Using the Clustering Table, you do not have to write code at all places where you instantiate new objects in order to provide a clustering hint. Instead, you ask the Clustering Table and he will be responsible for determining the correct location.

You can have many different implementations for your Clustering Table.

The Clustering Table can be table driven. In this case, you have to set up a table, be it hard coded or external. The table contains entries, which in turn maps from some information to a clustering hint. This table is typically very short, approximately between 10 and 20 entries, depending on the solution at hand. The following is an extract of such a table:

Object Type 1	Object Type 2	Move Type	Move Target
Device	Interface	DEEP	Device
Interface	Device	DEEP	Device
Network	Device	DEEP	Network
IPAddress	Device	SHALLOW	Device
…			

Some explanations for this sample: “Object Type 1” is on the from-side of the association; “Object Type 2” is on the to-side of the association. The “Move Type” indicates, whether an object should propagate moves to other associated objects (DEEP), or of not (SHALLOW). Finally the column “Move Target” indicates which object will stay at its current location and hence the other object is being moved.

A second approach would be that you use the Strategy Pattern for implementing the clustering strategy.

Clustering Strategy

Saying, you can use a table or the strategy pattern for implementing a clustering strategy, is not sufficient. The interesting part lies in what information do you use for determining the clustering strategy.

In a first best guess, you should look at your problem domain. Determine that most important abstraction (class) in your problem domain. You can have more than one class that fits this rule. Put the objects of these classes in the middle. Arrange (locate, cluster) all the remaining objects around them.

Next, you run your application. Profile your application and find out how many lock conflicts occur and how much disk-I/O your system needs. See also the consequences section about this item.

You can select the objects, the associations, or a combination of both for your clustering strategy.

For instance, you may give each association in your database schema a weight. Starting with some primary objects (or root objects), you can traverse the associations, and then using the weight of the association to determine, where a newly created object is stored. In this case, your table would contain entries with the class, the association weight, and the clustering hint, e.g. information about whether the newly created object should be clustered close to the object at the other end of the association.

Performance versus Concurrency

Looking at clustering you have to find a compromise between performance and concurrency. If you put objects too close together they might get locked all at the same time. This results in a good performance, but in a poor concurrency behavior, because clients access will be serialized.

On the other hand you can cluster your objects, so that each object has its dedicated lock. In this case the performance will be not that good. However, concurrency will be much higher, as more clients can work in parallel on the objects.

It is your task during the design or implementation phase to discover objects that represent a potential bottleneck.

The best result will usually be a compromise between performance and concurrency. A good mean to find the best compromise is to use the Clustering Table in order to experiment with different clustering strategies.

Consequences

1. All clustering related code is located at one place. You are not required to write clustering code at all places where a clustering hint is required.

2. As all clustering code is located at one single place it is easy to change the clustering strategy. You have to change the implementation of only one class in order to use a different clustering strategy. In the case of using the strategy pattern for implementing the Clustering Table, the following consequence applies: The implementation of only a distinct subset of classes has to be changed in order to use a different clustering strategy.

3. Play around with different clustering strategies. Employ performance tests in order to find out, whether you increased the number of lock conflicts or disk I/O. The former is an indicator for putting too many objects into one lock aggregate. In other words, the object density is too high. The latter indicates, that you should use fewer lock aggregates. You have to find the right balance between lock conflicts and disk I/O.

4. You can implement all clients without them even knowing what a Clustering Table is.

5. During development of your system, you should have different clustering strategies at hand. Once in a while, simply switch from one clustering strategy to a different one. This will result in a different runtime behavior, caused by different race conditions and lock conflicts. The benefit is, that this helps you to find bugs in the client implementation during development of your system.

6. If objects are clustered close together, e.g. in same container or on the same page, performance will be better, as less I/O operations are required.

7. If objects have each their own dedicated lock, concurrency will be optimized, as more database clients will be able to access the objects in parallel.

Implementation Issues

The most difficult part is to determine the set of information, which the clustering manager takes as an input for determining the clustering hint. However, there is no golden rule for this. It heavily depends on the individual problem domain and the application at hand.

Generally, you can always start with a table driven implementation. This is a very simple approach. The table typically takes between 10 and 20 entries with maybe half a dozen columns. Once the table gets larger than that, or if you want to use complex algorithms, e.g. examining and traversing a couple of existing persistent objects, to determine the clustering hint, you should consider using the strategy pattern for your implementation.

However, I do not have any experience with the latter, as up until now, the tables have always been very short.

Known Uses

Multiple Hewlett-Packard network support applications use this pattern, e.g. AutoCollect, Network Documentation Tool (NDT).

Related Patterns

Strategy Pattern, GOF. The strategy pattern can be used to implement the clustering strategy. In that respect, the strategy pattern is a complementary pattern.

Session Cache

Motivation

In order to access the database, each client has to set up a connection. A client in this context is every thread that needs access to the database. In this paper, I call the database connection a session. Creating a session and destroying a session is an expensive operation. The client side runtime has to allocate and initialize caches. Furthermore, it has to set up connections to the object server or the page server as well as to the lock server.

In short: How do you reduce the overhead for establishing and ending database connections (sessions)?

Forces

- Establishing database connections is expensive
- Ending database connections is expensive
- Clients frequently request and abandon database connections

Solution

Introduce a Session Cache. The Session Cache is responsible for maintaining a pool of sessions.

If a client tries to connect to the database, the Session Cache looks in the pool of available sessions first. If a session is available, the Session Cache takes the session from the pool and returns it to the client.

If no such session is available, the Session Cache creates a new session and returns the newly created session.

Once the client is done with using a session, it can return the session to the Session Cache, who in turn puts it back into the pool of available sessions.

An important issue is, how the session will be deleted. Under normal circumstances it is sufficient to define an upper limit of sessions, e.g. 10 sessions. Once the limit has been reached and another session has been added to the pool, the number of sessions that exceeds the limit will be deleted.

If resource consumption should be even lower, then the following strategy might be better. For each session in the pool, a timestamp of its last use is stored. Then, when a predefined time limit has been reached, without the session being reused, the session will get deleted. The advantage is that after the timeout period, the pool might be empty and therefore unused session do not exists. Consequently they don't consume any resources at all.

During shutdown, the session cache is also responsible for closing any open session and also deleting all sessions in the pool. The session cache itself is typically a static variable, or a variable global to the module, if you like.

Consequences

1. If a client request can be satisfied from the session pool, the overhead of establishing the database connection and ending the database connection is significantly reduced. The time for locating a suitable session is neglectable compared to the time required to establish a new database connection.

2. If the client abandons a database connection, it is simply added to the pool of database connections. As it is not actually destroyed the client does not encounter a performance hit.

3. Only if the pool is exhausted, creating a new session induces overhead on the client request.

4. Maintaining a session pool may lead to significantly higher resource consumption, e.g. memory, as all the caches and connections are kept, even if no client is connected to the database. This however can be eliminated, if the Session Cache uses a time-out period for detecting inactivity. If this happens, the Session Cache can then destroy some or all unused sessions. However, this again increases the latency for establishing data connections and the complexity of the Session Cache.

Implementation Issues

When you implement the Session Cache, you must make sure that the current state of the sessions is properly maintained. It is mandatory, that you never assign a session, which is currently in use, to a client. If you do so, it may work quite well for some time (even days), but eventually, the race conditions will lead to unexpected results.

Known Uses

Multiple Hewlett-Packard network support applications use this pattern, e.g. AutoCollect, Network Documentation Tool (NDT).

Related Patterns

Prototype. Exemplar. In both cases, a single instance of a class is created. Then, whenever another instance is required, it is simply a copy of the already existing instance. Session cache takes this a little further by having a predefined number of prototypes or exemplars, namely the sessions.

Session

Motivation

When a client opens a database connection, the client has to specify a number of attributes for the database connection[10]. Among them the client indicates the timeout, the kind of concurrency, the size of the client side cache, and so on. Further more, all these settings are also bound to an actual database connection, represented typically by a class, which the ODBMS's manufacturer provides[11].

Looking at the potentially big number of attributes required, it becomes obvious that it is easy to choose wrong settings.

Additionally, you will not need all the attributes for all database application you develop. For instance, some database systems allow for different kind of concurrency, e.g. dirty reads. Depending on your application, this might be an option or not. In order to simplify the interface presented to the client, you may want to encapsulate it in a separate class.

In short: How do you simplify using a database connection while at the same time providing support for exactly the required connection attributes?

Forces

- You want to simplify the interface to the database.
- You want to choose a subset of attributes describing a database connection. The client must only able to modify this subset.

---

[10] Usually the database system provides default values in case a client does not want to explicitly specify the values.

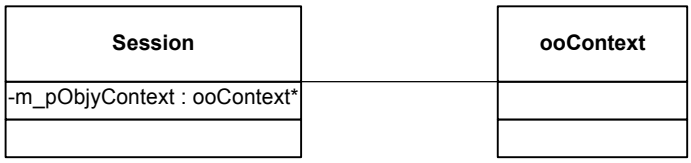
[11] Objectivity's name for that class is ooContext.

- You want to use a different set of default values for the database connections.
- You want to explicitly associate states to the database connection in order to use strict state transitions.

Solution

Encapsulate the database connection in a class Session.

Session also has an embedded member of the manufacturer provided class representing the database connection. Here is an example for Objectivity (ooContext is the manufacturer provided class):



Consequences

1. The clients "see" only the interface you want them to see.
2. The Session class can override the complete set of default values.
3. The Session class can use arbitrary states and state transitions.

Implementation Issues

You have to notice the general implementation issues, as described in "Encapsulate Manufacturer Supplied Class".

Additionally, you have to ensure the proper implementation of the state transitions.

Known Uses

Multiple Hewlett-Packard network support applications use this pattern, e.g. AutoCollect, Network Documentation Tool (NDT).

Related Patterns

Encapsulate Manufacturer Provided Class, Session Cache, Decorator, Adaptor.

Query Session

Motivation

Up to here, I have described scenarios, in which the clients use explicit transactions. The code for accessing the database looks typically like this:

```
IDatabase ptrDatabase;
ptrDatabase->BeginTransaction();
// some calls for reading and updating object go here
ptrDatabase->CommitTransaction();
```

I call this "explicit transactions", as the code contains explicit statements for marking the transaction boundaries. Opposed to this, implementing clients can become very easy, if you can avoid use of explicit transactions.

Working with databases, you often make queries. A set of objects is the result of such a query. Typically, you do not want to change the result set to change while you are working with it. The database system guarantees this within transaction boundaries.

In short: How do you provide non-mutable result sets, if your client does not use explicit transactions?

Forces

- The client does not use explicit transactions.
- The result set must not change while the client uses it (no additions, no removals).
- Result sets of queries are valid only within transaction boundaries.
- The client mixes query/result set related database calls with other (normal) database calls.

Solution

Associate a query session with each result set.

For clients not using explicit transactions, each database call will implicitly cause the database access layer to supply the call with a valid transaction. This does not work for query related calls.

The transaction for a query is started, when the database access layer detects that the client wants to execute a query. A new session, the query session, is then assigned to a result set object. Once the result set object goes out of scope, the transaction is committed (should always work, as this

is always a read-only transaction). In addition the query session is abandoned, or given back to the Session Cache, who in turn adds it to the pool of available query sessions.

When a client mixes normal calls to the database (causing implicit transactions) with calls related to the query or the result set, the database access layer automatically switches between the normal session and the query session.

A small side note: The solution is similar to the scoped transaction in the following respect: in both cases one object is bound to the lifetime of another object. The same way the scoped transaction is bound to a block or scope, the Query Session is bound to the lifetime of the result set.

Consequences

1. The Query Session uses dedicated transaction object. This is distinct from the transaction object, which is implicitly used for clients, which do not indicate their transaction boundaries.

2. The Query Session provides the context in which a result set is valid. The view to the database is consistent. The Query Session is used as long as the client holds at least one reference to the result set. As the transaction is not committed unless the result set goes out of scope the result set does not change until then.

3. The Query Session can survive implicit transactions, as it is completely transparent to the client.

4. The implementation of the database access layer becomes more complex, as you have to add the code for detecting queries and switching between normal sessions and query sessions.

5. You can apply filters and sort criteria to the result set without modifying the raw result set. By removing the filters and the sort criteria, you will revert the result set to its initial contents.

Implementation Issues

I have already discussed the problems associated with the Session and the Session Cache. Introducing the Query Session, the Session Cache may potentially become more complicated, as the Session Cache has now to administer two different types of sessions. The logic for determining which type of session is needed, has to be implemented somewhere, e.g. in the Session Cache

You can reduce this additional complexity by employing two different pools for sessions. One pool contains only “normal” sessions and the other pool contains the Query Sessions.

One aspect simplifies the implementation of Query Session: All Query Sessions are typically read-only transactions.

Known Uses

Multiple Hewlett-Packard network support applications use this pattern, e.g. AutoCollect, Network Documentation Tool (NDT).

Related Patterns

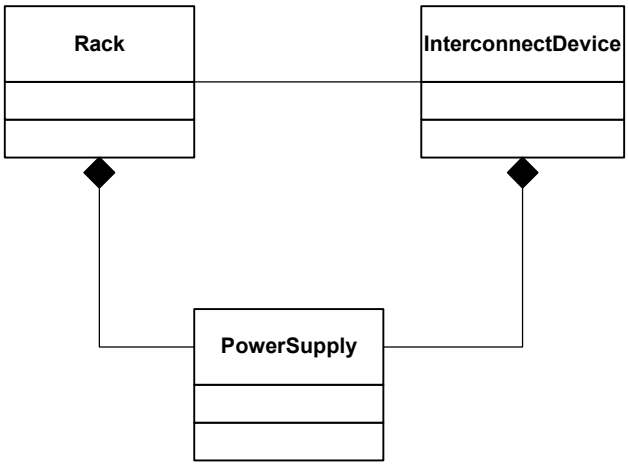
Encapsulate Manufacturer Supplied Class, Session Cache.

Association Traversal with Path Specification

Motivation

An object usually fulfills its responsibility in collaboration with other objects and hardly stands on its own.

A device management application may use the following problem domain classes for representing a 19" racks, interconnect devices and power supplies[12].



We now want to retrieve all power supplies, be it for a rack or for an interconnect device.

The first approach would be to use a typed iterator on the complete database. This will work in all cases. However, in most cases this will be too slow, as the database system would check every object in the database. The database system has to determine whether each object is an instance of class PowerSupply or of a class derived from PowerSupply.

---

[12] Note: I use these classes for illustration purposes only. To the best of my knowledge, they are not used in any of Hewlett-Packard's products.

A better approach would be to use a navigational access to determine all Rack objects. Then you could traverse the association from the Rack objects to the associated PowerSupply objects. The result would be all PowerSupply objects that provide power to a Rack. The next step would be to traverse the association from each Rack object to the associated PowerSupply objects. The latter would be added to the intermediate result set, leading to the requested result set.

I call a traversal from one class to another class a path. In the given example, I used two different paths: Rack – PowerSupply and Rack – InterconnectDevice – PowerSupply. Both paths describe how to collect the PowerSupply objects.

In short: How do you effectively collect objects for a result set, if multiple association traversals are possible?

Forces

- Not all of the object-oriented database systems provide a query language as powerful as SQL[13].

- Depending on the query, the path for traversing is different. So the solution must provide for a parameter on how to specify the path.

Solution

The solution depends on how persistence is achieved in the database system. The solution presented here assumes that persistence is implemented through inheritance, that means: all persistence-capable classes are derived directly or indirectly from a common base class.

A new class PersistentObject is introduced and derived from the common base class, e.g. ooObj[14], for persistence capable classes. All other classes to be persistence capable are derived from the class PersistentObject.

The class PersistentObject carries a member function called TraverseAssoc(). This function has two parameters, one for the path specifications, and one parameter is passed by ref and is a container that takes all objects that are found during the traversal. The path specification can be a simple table such as the following one:

---

[13] OQL (Object Query Language) is the equivalent to SQL in the object-oriented database world. Unfortunately, the OQL implementations differ from one vendor to the next, and they do not provide constructs for traversal in all cases.

[14] Objectivity uses ooObj as the common base class for all persistence capable classes.

ClassID	RoleID	bCollectObjects
ManagementAgent	Manages	False
Device	N/a	True

Consequences

Again, as for all database related stuff, the most important thing is to take all possible race conditions into account. For instance, what happens when starting traversing, the start object is moved to a different location? Should the traversal restart? Or is it possible to continue?

There is no general answer to this issue. It – again – depends on the used database product. The solution I am using simply restarts the traversal. At present we do not have a performance issue here.

When traversing from one object to another, it is possible to detect, whether the start object or any of the other object has been moved or updated. If this happens, the code bails out and restarts the traversal.

Implementation Issues

Traversing associations can potentially affect many objects, hundreds, thousands or even more. When accessing one of the objects for the first time, it could well be that several versions of that object are available. Several versions can exist, when multiple clients access the database. In this case, each client might have different, yet still consistent view of the database. In this situation the software developer must determine whether to use the version as at the time when the traversal started of the version of as when the object is accessed for the first time.

Known Uses

Multiple Hewlett-Packard network support applications use this pattern, e.g. AutoCollect, Network Documentation Tool (NDT).

Related Patterns

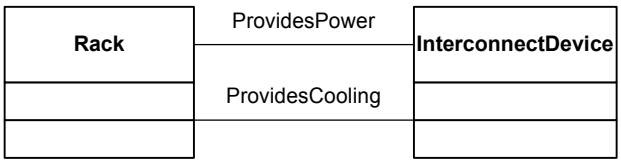
Strategy [GOF], Association Traversal with Roles, Association Traversal with Predicates.

Association Traversal with Roles

Motivation

In this pattern I discuss a different scenario you may encounter, if more than one association exists between two classes.

E.g. a 19” rack may provide power to a set of interconnect devices, such as a router, which is installed in that rack. In addition, the same rack provides cooling to the same or a different set of interconnect-devices.



If you want to retrieve all interconnect devices, the result set depends on the association you traverse. In order to retrieve only interconnect devices for which the rack provides power, you only want to traverse the association named ProvidesPower.

In this case it is obviously not sufficient to just specify the path Rack – InterconnectDevice.

In short: How do you specify roles when traversing associations?

Forces

- Two classes are related to each other with more than one association.
- You want to be able to retrieve both sets on only one association or the join of more than one association.

Solution

Specify roles when traversing associations.

You traverse associations using iterators. Compared to Association Traversal with Path Specification you also supply the iterator with information about the roles for each association.

Given the above sample, the iterator needs information about whether to follow the association “ProvidesPower” or the association “ProvidesCooling”. The Rack has both roles, but one may only be requested.

The following source code shows how an iterator is set up depending on the role:

```
ooItr(InterconnectDevice) iter;
ooHandle(Rack) hRack = ...;
hRack->getRelatedObjects(iter,
                         ooTypeN(InterconnectDevice),
                         1); // role ID
while( iter.next() ) {
   ...
}
```

Consequences

1. Two classes may be related to each other with more than association.

2. Iteration and association traversal requires you to also specify roles, if more than one association exists between two classes.

3. The implementation of the iterator becomes more complex, as it has to consider roles in addition to the path specifications.

Implementation Issues

You should use an enumeration for specifying roles. I recommend to use one enumeration per class.

Known Uses

Multiple Hewlett-Packard network support applications use this pattern, e.g. AutoCollect, Network Documentation Tool (NDT).

Related Patterns

Association Traversal with Path Specification, Association Traversal with Predicates.

Association Traversal with Predicates

Motivation

In addition to Association Traversal with Path Specification or Association Traversal with Roles you may want to restrict the result set to objects that also pass a filter, e.g. you want to retrieve only interconnect devices of a particular type.

In short: How do you restrict a result set while traversing associations?

Forces

- You want not only to traverse associations but restrict the result set to objects that do suffice certain criteria

Solution

Combine predicates with paths or roles.

Again you implement an iterator for traversing associations. In addition to a path specification, a role or both, you provide the iterator with a set of predicates.

Consequences

1. You can restrict the result set with a set of predicates.

2. The implementation of iterators becomes more complex, as additional information is required. Using default values can minimize this issue, in case you do not want to restrict the result set.

Implementation Issues

You should avoid to pass the predicates as strings, e.g. “Manufacturer = Cisco AND IOSVersion >= 11.0”. Instead use an array such as the following:

AND/OR	**Name**	**Boolean Op**	**Value**
	Manufacturer	=	Cisco
AND	IOSVersion	>=	11.0

Note, that the first row does not contain an entry in the first column.

This table can become as long as required.

If the underlying database system provides a query language, I recommend to use it. However, you should still avoid to pass strings from the client to the database access layer, as otherwise you will have to also implement a parser for that string (or rely on the database system).

You definitely have to implement a parser, if the underlying database system does not support a query language, or if the query language you expose at the interface is different to the one of the database.

ODMG 3.0 defines a query language named "Object Query Language" (OQL). For more information, see [Cattell 2000].

Known Uses

Multiple Hewlett-Packard network support applications use this pattern, e.g. AutoCollect, Network Documentation Tool (NDT).

Related Patterns

Association Traversal with Path Specification, Association Traversal with Roles

Error Handling

Motivation

Some software components, including the database system I am using, use a UNIX-style signaling mechanism for error handling.

Using C++ as programming language, the software developer may want to use C++ exception handling for error handling. In this case the best solution would be, if the error handler could directly throw an exception. Unfortunately, the software component may not be written with exceptions in mind. Then variables and objects created on the heap or on the stack may not get deleted properly. Or even worse: database locks or synchronization objects may not get released, leading to a deadlock situation.

A straightforward solution would be to check the return value of each call into the software component and if the value indicates an error the client code would throw the exception.

In short: How do you map UNIX-style signaling mechanism to C++-like exceptions?

Forces

- The client wants to use C++ exceptions for error handling.
- The software component uses an UNIX-like signaling mechanism for indicating errors.
- The software component does not work correctly, when an exception is thrown in the error handler.

Solution

In the error handler store all necessary error information at a well-known place, which is accessibly to the client of the software component.

Implement a class ErrorCheck that in its constructor cleans the error information of the client.

In the destructor, the object checks whether an error has been signaled, and if so, it throws an exception.

Consequences

1. The benefit of the solution is, that if a new version of the software component becomes available, the implementation of the error handler can then throw exceptions directly. The client code does not need to be modified in any way.
2. Another not so obvious benefit is that for debugging purposes, the destructor of the ErrorCheck class can be modified in a way, that errors are simulated. In this setup, the error handling code of the client can be verified. This is a very powerful technique for testing and debugging multithreaded database applications.

Implementation Issues

Some programming languages do not execute the destructor immediately, when the reference to an object goes out of scope. Here you have to employ different language features.

Known Uses

Multiple Hewlett-Packard network support applications use this pattern, e.g. AutoCollect, Network Documentation Tool (NDT).

Related Patterns

Scoped locking,

Scoped transaction.

Encapsulate Manufacturer Supplied Class

Motivation

As typical approach for specializing manufacturer supplied classes you would normally simply introduce a new class derived from the existing class.

However, manufacturer supplied classes are typically implemented in runtime libraries. Normally you do not get the source code for these classes. The runtime library might allocate a separate heap. Furthermore, you have to note that the destructor is a class method whereas the destructor is an object method. This can lead to the effect that for creating an object of a manufacturer supplied class memory is allocated from one heap, whereas the destructor tries to free the memory on a different heap. This happens in particular, if the destructor is declared virtual.

In short: How do you avoid side effects, if you want to specialize a manufacturer-supplied class?

Forces

- You do not have the source code for the manufacturer-supplied class.
- Both, creation and destruction of instances of the manufacturer-supplied class must be done on the same heap.
- The manufacturer-supplied class may or may not have a v-table.
- The interface of the manufacturer-supplied class is too complex.

Solution

Instead of deriving from a manufacturer supplied, embed a member of that class in the encapsulating class. E.g. assume that the ODBMS you use has a class named d_transaction. If you want to specialize this class, your encapsulating class should look like this:

```
class Transaction {
public:
  ...
private:
  // Attributes
  d_transaction   m_nativeTransaction;
};
```

Consequences

1. Encapsulating the class instead of deriving a new class avoids possible side effects with regards to memory management.

2. You can have a complete new set of public member functions.

3. If you want to replace your current ODBMS by a ODBMS of a different vendor, the change becomes much easier, as only the encapsulating classes have to be changed[15].

Implementation Issues

In most cases, only a subset of the original functionality will be required.

Many of the needed functionality can be achieved by simply forwarding a call to the implementation of the manufacturer supplied class.

Known Uses

Multiple Hewlett-Packard network support applications use this pattern, e.g. AutoCollect, Network Documentation Tool (NDT).

Related Patterns

Adapter, Bridge.

---

[15] This is not necessarily true, if some of your code relies on a specific ODBMS specific feature, e.g. MROW, which is a concurrency model available only with Objectivity.

Appendix

A Lightweight Introduction to Object-Oriented Database Systems

Although object-oriented database management systems (OODBMS) have been around for a while, they still lack widespread usage. Some industries, such as the telecom industries, is using OODBMS on a regular basis also for mission critical systems. There is however, resistance in the more traditional branches such as banking or insurance. The latter still have – in part caused by legal requirements – a preference for relational (or even hierarchical) databases.

In essence an OODBMS provides means for making objects persistent. The idea is that the software developer does not need to learn an additional language (such as the DDL[16] part of SQL[17]) for defining the schema of the database, but can use the programming language (e.g. C++ or Java) instead.

Apart from that OODBMS have a lot in common with relational database management systems (RDBMS). The things they have in common are transactions, queries, iterators, etc.

But be aware, although the differences seem to be small, their impact is tremendous. I found examples, which were almost impossible to implement with a RDBMS, as it used more than 16 tables in a join and the RDBMS at hand had a limit at 16 tables. Opposed to that it was pretty simple to implement the same design with the OODBMS.

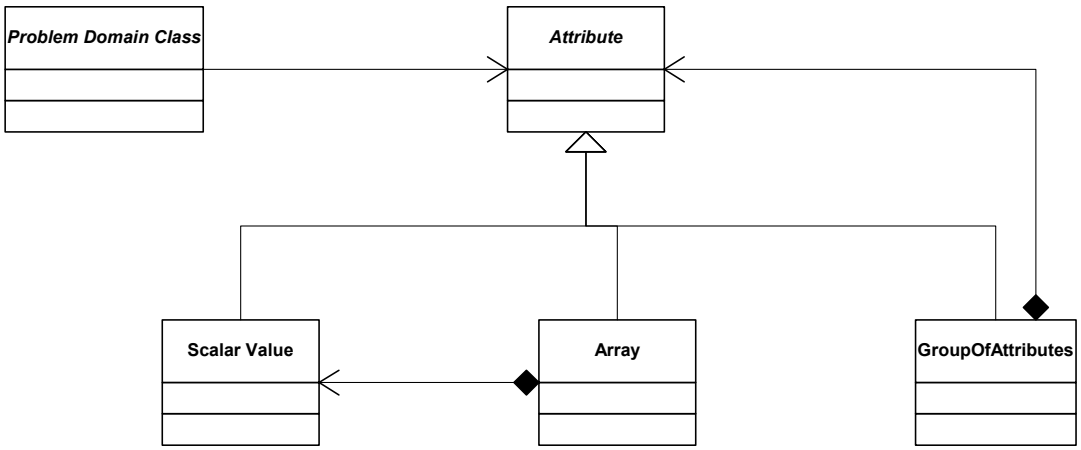
A Short Sample

Given an arbitrary problem domain, you might want to have the flexibility to add attributes, e.g. scalar values, arrays or groups of attributes, to any object you like without changing the schema of the database. Using UML the following picture shows one possible design:

---

[16] DDL = Data Definition Language

[17] SQL = Structured Query Language



Using an ODBMS[18] with C++ binding the schema would be defined as:

```
class ProblemDomainClass {
// other stuff left out for brevity
private:
  ooRef(Attribute) m_attributes[];
};

class Attribute {
  // other stuff left out for brevity
};

class ScalarValue : public Attribute {
  // other stuff left out for brevity
};

class Array : public Attribute {
  // other stuff left out for brevity
private:
  ooRef(ScalarValue) m_members[];
}

class GroupOfAttributes : Attribute {
  // other stuff left out for brevity
private:
  ooRef(Attribute) m_groupedAttributes;
};
```

[18] In this paper, I am using the DDL of Objectivity. For more information on objectivity, see http://www.objectivity.com

References

Catell 2000	Rick Catell, and Douglas Barry, Editors: "The Object Data Standard: ODMG 3.0", Morgan Kaufmann Publishers, ISBN 1-55860-647-5
Coldewey	Jens Coldewey: "Choosing Database Technology", http://www.coldewey.com/publikationen/database.html
GOF	"Design Patterns – Elements of Reusable Object-Oriented Software", Addison-Wesley, ISBN 0-201-633612
Keller 1998	Wolfgang Keller: "Object/Relational Access Layers", Proceeding of EuroPLoP 1998
Keller 1995	Wolfgang Keller: http://www.objectarchitects.de/ObjectArchitects/orpatterns
Meyers	Scott Meyers: "More Effective C++: 35 New Ways to Improve Your Programs and Designs", Addison-Wesley, 1996, ISBN 0-201-63371-X
Stroustrup	Bjarne Stroustrup: The C++ Programming Language, Addison Wesley, 1997, ISBN 0-201-889544

Acknowledgements

I would like to thank my shepherd Jens Coldewey for his very useful comments. It was not only fun, but I also learned a lot during the shepherding.

Thanks go also to the people from Micram AG, Bochum, for their help, especially Raimund Backes and Törk Hansen.

Finally, but not least, I would like to thank the participants of the writers workshop at the EuroPLoP 2000 for their support and the many suggestions for improvement.