Half-Object Assembly - a Pattern System for Distributed Domain Objects in Business Applications

Fridtjof Toenniessen

sd&m GmbH&Co.KG
Thomas-Dehler-Str. 27,
D-81737 Munich, Germany

Phone: +49-89-63812-330; email: fridtjof@sdm.de

1. INTRODUCTION

Distributed systems are a widely discussed topic in today's software engineering practice. They complement or sometimes even replace traditional approaches that employ central mainframe architectures. Very often the distributed architectures go hand in hand with object oriented modeling and these models have been augmented by design patterns in recent years.

Half-Object Assembly is a system of special purpose patterns for distributed business information systems. It helps to form the communication layer between the clients and servers assuming the client/server cut is made through the application kernel to maximize flexibility in application design:

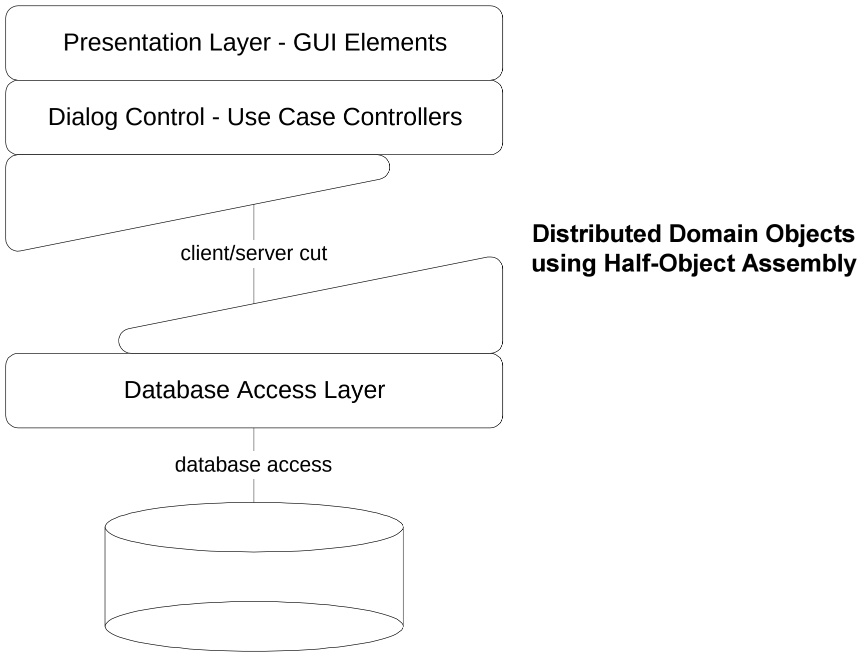
Presentation Layer - GUI Elements
Dialog Control - Use Case Controllers
client/server cut
Distributed Domain Objects using Half-Object Assembly
Database Access Layer
database access

Fig. 1: *Layers in a distributed business information system*

The patterns consider high transaction rates combined with low bandwidth networks by transmitting only those parts of business objects that are of interest and by offering flexible transaction and caching mechanisms.

2. ROADMAP

Half-Object Assembly combines *Half-Object+Protocol* [Mesz], *Cache Proxy* [BMRSS] and a variant of the *Composite-Part* [BMRSS]. To allow flexible transmission and transaction mechanisms one can also use the *Strategy* pattern [GHJV]. The patterns in bold rectangles are discussed below.

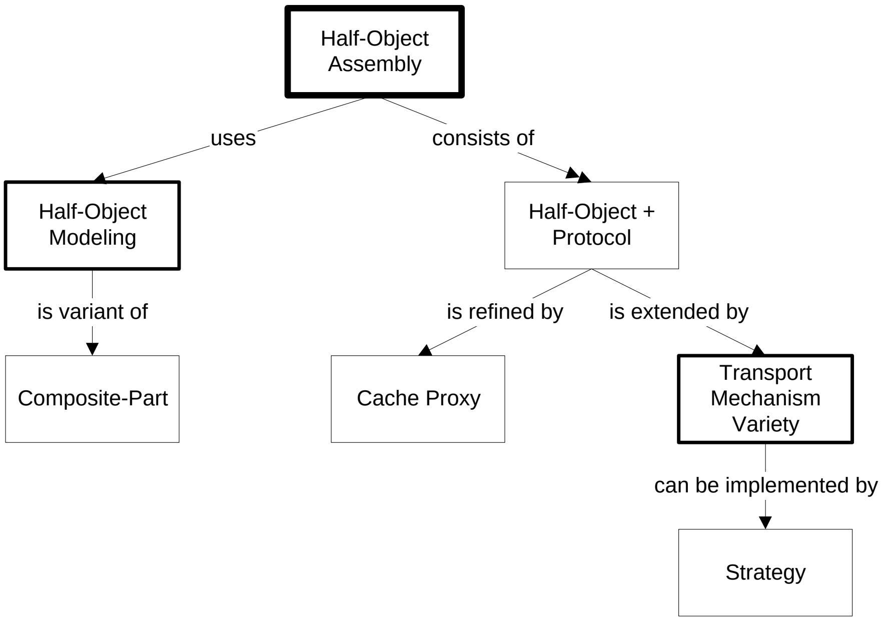

Fig. 2: *Roadmap through the pattern system*

General context

You are designing a large scale distributed business information system, in which many users want to access a huge amount of data in parallel. A low bandwidth network makes it impossible to transmit complete objects for every user transaction. Imagine for example an information system for a railway company, that allows to read and maintain technical data of all locomotives and cars. There will be around 1000 online users spread all over the country and connected by only 9,6 kbit/s lines stemming from legacy applications in mainframe technology.

The pattern system does not address the transport of BLOBs or things like real-time video data. (This is due to the low bandwidth network which often appears at the customer site and allows the transmission of small data packages only.) Instead you are primarily interested in typical business data comprising elementary information like strings or numbers that can easily be transported by high level mechanisms like CORBA or any RPC.

General problem

How can you make the system flexible, responsive, and able to handle large traffic volumes?

General forces

1. *Choosing the right size of the objects*. If the objects of interest are huge, the transmission of complete instances in a low bandwidth network will be unacceptably long. Experience shows that this time increases exponentially if the network has high load. This is intolerable since online users are very often interested only in small parts of the objects. On the other extreme, beware of organizing the transport packages too fine grained! This is because you should avoid too many server accesses when displaying a meaningful amount of information on the screen, e.g. to fill a complete window or at least one page in a notebook. With too many server accesses you may overload the communication mechanism in the server nodes.

2. *Partial views on large objects*. The next step is to assume that the truth lies in the middle: The large objects should offer services for each dialog window to fetch the data needed to fill it. But even this approach has disadvantages: In addition to being already monster objects, their interfaces get even more complex this way. This makes maintenance significantly harder. Furthermore, it seems suspicious to have too much functionality concentrated in only few objects.

3. *Seamless interface*. The mechanisms of object transport should be hidden in a lower layer of the software in order not to overload the application with technical details.

4. *Searching for a pragmatic and light-weight solution*. When designing a distributed object oriented system, of course the Common Object Request Broker Architecture (CORBA) is in the center of interest. So what advice do we get from it? Indeed, CORBA offers a general mechanism for moving and copying objects between network nodes (see the *Life Cycle* services in [COSS]). But this mechanism is too expensive for our purposes: For every object that is copied you have a number of remote calls reducing performance significantly. Furthermore, the standard is rather generic at this point, so it seems that commercial products will at most offer a general framework and you have to code the important parts yourself. So what we need is a pragmatic and light weight solution to replicate objects - without losing the benefits of a CORBA like architecture.

General applicability and known uses

Half-Object Assembly is useful if you have to forge a link between nodes of a distributed information system, where classical business data (strings, numbers etc.) have to be transmitted. It considers disparate programming environments (e.g. Smalltalk and C++) and even heterogenous hardware and operating systems (e.g. the management of a telephone switching system), on which semantically equivalent object models are implemented and the objects' data have to be replicated or exchanged.

Half-Object Assembly is successfully used in the project *Database for Reisezugwagen and Triebfahrzeuge* (DaRT) for the Deutsche Bahn AG (DB AG). The pattern is going to be reused in the project *Production Planning* (PPSF) for the DB AG.

Related patterns

Similar problems are addressed in the *Replication* pattern [MoMa] and in *Recoverable Distributor* [IsDe].

Half-Object Assembly (HOA) is one possible application of *Half-Object+Protocol (HOPP)* [Mesz]. The half-objects form the atomic building blocks of HOA. HOA differs from HOPP in that HOPP considers half-objects to be isolated, whereas HOA defines a particular synchronization protocol and emphasizes the interconnection and cooperation of many half-objects forming a complete distributed application domain model. More metaphorically, HOPP describes a violinist, cellist or trumpet player; HOA tries to form an orchestra out of them.

3. DESCRIPTION OF THE PATTERNS

Pattern 1: Half-Object Modeling

Context

You are designing a distributed information system in which object data on remote servers have to be displayed and maintained on the clients. The customer is interested in a huge quantity of information for each "real world" object and tells you of several hundred - partly multivalued - aspects of the objects. By that your application entities resulting from analysis become huge clusters of data. You don’t always want to completely transport them between client and server.

Problem

How can you structure application classes to transmit only those parts of the objects relevant to a specific dialogue?

Forces

1. *Performance*. A first option is to equip big application classes with several different data accessing methods. Dependent of the dialog context, these methods provide the client with suitable parts of the objects. According to ergonomy examinations in [FrBr] for user interfaces the maximal response times for indexed data accesses should be between one and two seconds (dependent on the task). So as a rule of thumb the transported data shouldn't be more than 200 bytes in size for every kilobit per second of network capacity. This consideration bases on two assumptions: First assume that the time needed for indexed database accesses ranges within milli-seconds and therefore can be neglected. Second it is supposed that the overhead of the underlying transport protocol is also neglectable (e.g. the IIOP standardized in [CORBA] produces an overhead of about 40 bytes per message in comparison to classical sockets as long as you don't use the *Any* type in your IDL specification). This limitation vanishes of course when you switch to a broadband fabric (150-250 Mbit/s) where you can model objects of 10-15 megabytes in size and more, in theory. It is more the maintainability question telling you not to model too big objects in this case (see the next point).

2. *Maintainability*. If the objects grow too big, then they should be split up. From the pure design perspective (according to [Meyer]) a class shouldn’t have more than 40 features; otherwise maintainability decreases rapidly. The domain classes considered in our case are entity classes which are more or less intelligent data containers. This means that they contain - besides some internal check routines for data integrity - mainly the standard accessor methods for their instance variables. Since we have 1-2 access methods per instance variable, the average number of instance variables of a class should be between 15 and 20.

Of course your system gets more complex with the number of classes involved. So a reasonable tradeoff must be found between object size, network capacity and system complexity.

Solution

Decompose large objects into smaller ones, whose data are transported in less than a second or are needed to be read from the server within one single user action (e.g. to fill the contents of a subwindow or notebook page). Even if the semantics enforce embedding, let the objects reference each other by pointers only. This way all object parts become autonomous instances and can be instantiated and transported independently. This minimizes network load and memory allocation on client and server. The figure below shows such a decomposition in case of 9,6 kbit/s lines. This enforces objects of about 1 kByte in size to restrict the performance loss due to data transmission to a tolerable amount of time (one second).

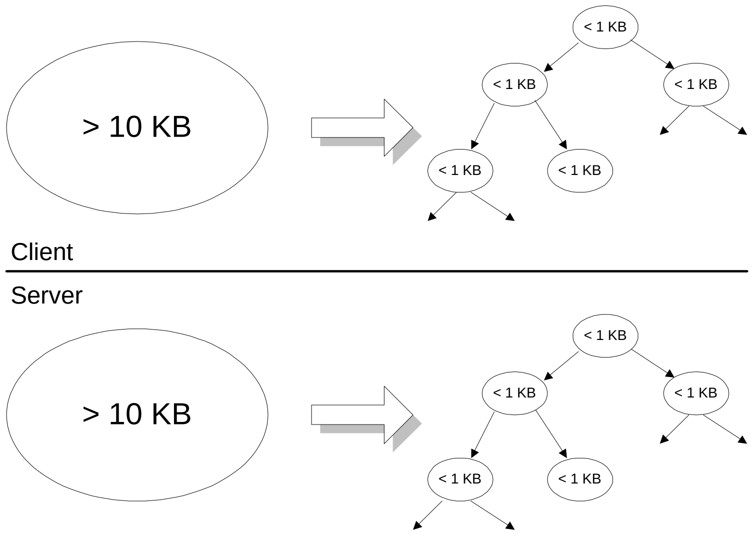

Fig. 3: *Decomposition of large objects*

As the figure suggests, you should use the same semantic domain class model on both client and server. This can be done even when you use different languages on client and server. You only have to restrict yourself to a common denominator of the language features.

Resulting context

1. *A structure for saving memory space and network capacity*. The approach provides an effective structure to instantiate only those parts of an object that are really needed. The client side should provide an appropriate structure for the transmission of small data packages. On the server side it can save a lot of memory space, e.g. when you are using an object oriented database system (OODBMS). This is because the OODBMS usually treat aggregations as one single object and therefore would lock, read or write too many pages at once. (Dependent on database size, the server processes can be increased in size by 10-100 MB of data that are of no interest at the moment.) By using pointers, only the required parts of the whole aggregation is read from the disk.

2. *A structure for decentralizing functionality and responsibility*. Half-Object Modeling does not hide internals of the big objects. All instance variables still must be reachable from the root object. But it becomes possible to distribute the authority of an object's data transmission to the objects

themselves respectively their parts. So there is no single and huge module having the burden to control all server accesses in the system.

3. *Increased complexity*. Half-Object Modeling increases the number of classes in your system, so it becomes more complex. Design your classes in a way that the data are transported by the network in a tolerable amount of time.

4. *Pointers are less secure than embedding*. Make sure that you don't assign pointer attributes from outside an object. This can easily be done by avoiding the corresponding set-methods. The setting of these attributes has to be done by the object itself. This way you avoid undesired sharing of objects which can cause ticklish errors.

Related patterns

Half-Object Modeling (HOM) has some relationship to the *Composite-Part* pattern [BMRSS]. But note the difference: The root object in HOM does not play any role of *Wrapper* ([GHJV]) as it does in the *Composite-Part*. Within HOM the instance variables of every object in the hierarchy are still visible to the client through navigation.

Pattern 2: Half-Object Assembly

Context

Imagine you are designing a distributed information system whose client nodes display and maintain the data of objects spread all over the network. The same semantic domain class models are underlying the implementation of clients and servers. They are possibly implemented in different languages or on different platforms.

Problem

How can you manage the transmission of the object's data without overloading the servers and the network?

Forces

1. *Transparent navigation through the object hierarchy*. It should be possible for the clients to navigate through remote domain objects as if they were local. Only these domain objects should be instantiated on the client which are really needed. All server accesses should be hidden from the application programmer.

2. *Centralization vs distribution*. You have an analogous object hierarchy on clients and servers and the object's data have to be exchanged or replicated between them. You have to decide whether to implement a central object server that forges a link between two worlds or to delegate the responsibility for data transmission to each of the objects themselves. The situation is comparable with that of a database access layer. A central object server has the advantage that the client/server communication is clearly modularized and does not burden the application classes. But it is not scalable at all. This is because with every new class the knowledge and size of such a central

module grows. It has to know about every single server access. Furthermore, when using a compiled language, including this module generally means compile-time dependency from the whole system - a well known design flaw that has to be definitely avoided in large systems.

Solution

Use the general idea of *half-objects* [Mesz] for every application domain object from the decomposition above. Implement the synchronization protocol of the half-objects in two steps. First enhance the client object with a cache of all instance variables of the server object it is connected to. Second every server object has to offer remote services to read respectively write some data structure corresponding to this instance variables.

The transmitted data structure comprises both elementary data to be displayed on the screen *and* remote communication handles of all objects referenced directly by the actual server object. This means that you need to read objects by key only for the roots of object hierarchies. For further navigation all the remote handles are available on the client.

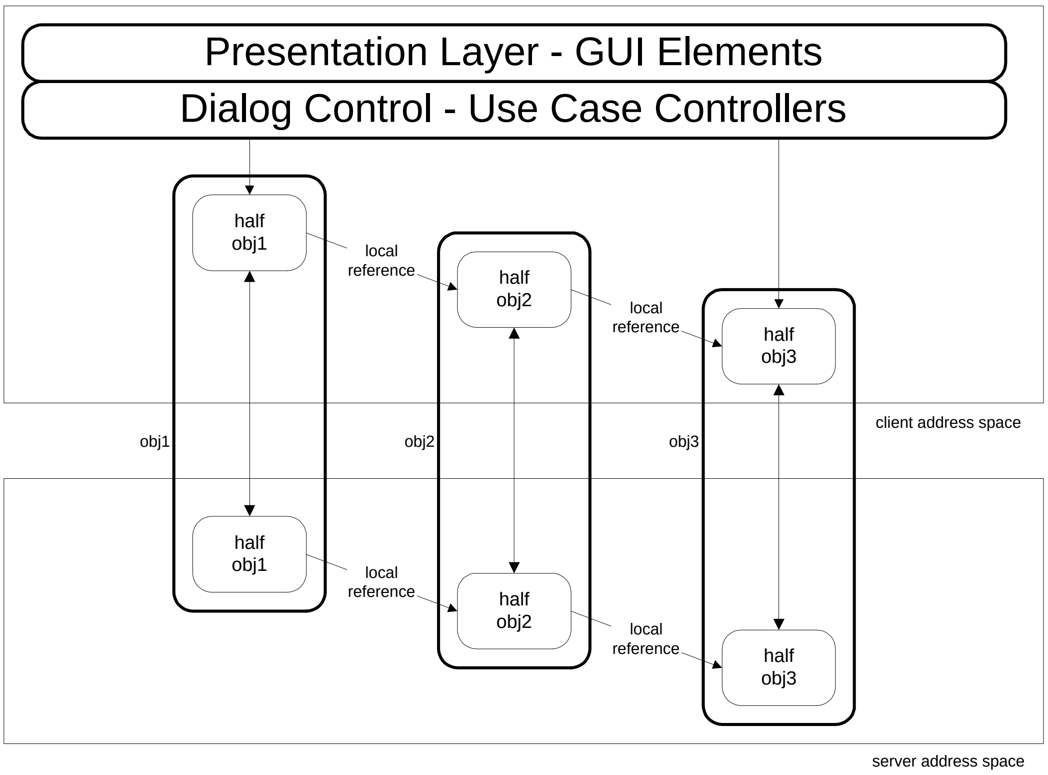

Fig. 4: *A hierarchy of half-objects forms a distributed application object model*

The following scenario shows how you can use this assembly of half-objects to navigate through the object graph and to transport only those objects that are really requested (HOn-C denotes the half-object with number *n* on the client side, HOn-S its counterpart on the server):

1. HO1-C is created, its remote communication handle is read from the server and put into HO1-C. Result: HO1-C is *connected* to the server.

2. When the elementary data from HO1-C are needed, they are fetched from HO1-S via the remote reading message. The returned data structure contains the elementary data for HO1-C and the remote communication handle to HO2-S.

3. HO1-C is equipped with its elemenatry data and with a local reference to a newly created HO2-C. The returned communication handle to HO2-S is put into HO2-C. Result: HO1-C is *cached* and HO2-C is *connected*.

The procedure continues recursively if the data of HO2-C are required. You see that the object hierarchy grows dynamically as the user navigates through it.

A data writing scenario can easily be derived from this: Every client half-object can flush its elementary data to the server half-object which immediately commits the changes to the database. Then the client half-object invokes the flushing on all half-objects directly referenced, so that finally the whole subtree is written to the database. You can optimize network load if you introduce dirty markers for every half object and don't flush any clean objects.

Resulting context

1. *Suitable for code generation*. The standard half-object protocol used here is a simple and generic solution. The work is distributed among the participating objects. It is well suited for automatic code generation, so that application developers don't need to care much about client/server communication.

2. *Minimized network traffic*. Another advantage is that the network traffic is minimized. When reading objects, only those objects are sent over the network that are really needed. When writing a complex object hierarchy to the server, only the objects that have changed are sent to the server.

3. *Many client half-objects connected to one server half-object*. In contrast to *HOPP* [Mesz] the half-objects may be split up into more than two address spaces: It is possible that several client half-objects are connected to one single server half-object when users are reading the same objects.

4. *Simple but restricted transaction logic*. Be aware of the simple and restricted transaction logic of this approach. Committing each half-object separately implies splitting of one logical writing transaction into several technical transactions which in sum don't have the ACID properties of transactions (Atomicity, Consistency, Isolation, Duration, see [Date]). For example, if the user had changed the root and some other object in the hierarchy and "commits" this changes, it could happen that the root object is correctly written to the database, but then the referenced object refuses the commit because another user changed it in the mean time. This violates the atomicity and isolation property of transactions. Therefore this approach is suitable only when there are no integrity constraints within different objects in the hierarchy. But don't be confused - there are indeed many situations in which such a simple strategy is appropriate. When used, it can dramatically increase the data throughput.

5. *No automatic cache validation*. Note that a client half-object remains cached and clean even when another user has changed the corresponding object on the server in the mean time. No notification takes place. This fits well to transactions using optimistic concurrency control [Date].

6. *Many server accesses if the object model is fine grained.* If there are many small objects in the system, reading can be too slow since every object is fetched by a separate remote call. The same is true for writing.

7. *Leaving some objects transient on the client.* In the discussion so far every application domain object is connected as a half-object to the server. If you have many small objects - e.g. *Flyweights* [GHJV] - it may be desirable not to connect all of them to avoid connection overhead. Then you can instantiate those small objects transiently as simple data containers without connecting them. In this case you cannot use them for further navigation or write them directly. This has to be done by their supervisor object being connected to the server.

Related patterns

The atomic building blocks of Half-Object Assembly are the half-objects from *HOPP* [Mesz] as mentioned above. But note that the half-objects considered here hold an asymmetry because several client half-objects can be connected to one server half-object. Therefore they could be called third-, quarter-, fifth-objects and so on. In order not to overdo it and to refer to existing terminology they are still called half-objects.

The communication handle of the half-objects stems from the *Proxy* idea that is de-facto-standardized in [CORBA]. It is discussed in a more general way in [GHJV]. In our case the half-objects can be considered as a somehow "localized" form of *Cache Proxy* as described in [BMRSS] respectively [Rohnert]. Whereas a cache proxy is considered alone and manages several objects at once, a half-object is responsible for caching the data of one single object only. From the pure caching perspective one half-object alone is rather primitive. The real benefit comes from the co-operation of many of them.

All these methods semantically belong to a separate layer: the communication layer, a special case of access layer. Embedding these methods within the application class corresponds to the architectural pattern *Multilayer Class* (see [CoKe1]).

Pattern 3: Transport Mechanism Variety

Context

You are building a distributed information system and must distribute your application domain objects between clients and servers. You are applying the Half-Object Assembly pattern for doing this. Unfortunately you have a very fine grained object model or some integrity constraints between different half-objects so that the standard data transport mechanism enforces either too many server accesses or splits logical transactions which can eventually cause data inconsistencies.

Problem

How can you avoid too many (fine grained) server accesses or splitting of logical transactions?

Forces

Transaction control on client or server? Splitting of logical transactions can be avoided if you open, commit or rollback server (database) transactions explicitly on each client. These transactions will then usually span several remote reading and writing messages between half-objects. But as experience shows, this makes the system more complex and error prone. You have to consider extreme situations like losing connection or client crashes while a transaction on the server still remains open. You also have to maintain different transaction contexts in the server processes if they are accessed by more than one client.

You should take as simple an approach as possible when building large industrial size information systems. So it turnes out to be best to leave the transaction control completely on the server. This means every remote call transporting data causes one transaction to be opened and closed before the call returns. Therefore all server processes are stateless. Of course this means more complex and comprehensive messages sent between half-objects.

Centralization vs distribution. When leaving the transaction control on the server you have to group several half-objects to units that are read or written in one single step, causing only one single remote message. You have to decide whether to implement a central control object that does the job of collecting and decoding a set of half-objects or to distribute the responsibility among the half objects themselves. Again, as in the forces section of the Half-Object Assembly pattern, a central module has the advantage that the task is clearly modularized and does not burden the application classes. But it sacrifices scaleability and should therefore be avoided in large systems.

So the best thing is to enhance the intelligence of those half-objects that are responsible for the data transport units mentioned above. There are different options to what extent this enhancement is appropriate. To avoid too much complexity, the authority of every half-object to change the standard mechanism should be restricted to the objects "under its control", which generally means the objects that can be accessed through it by navigation. In the frequent case of a tree-like structure this means that the root of any subtree can influence solely the objects contained in the subtree.

Solution

There are two possibilities to solve the problem:

1. *Adding several different remote methods to the half-objects*. A half-object can use more comprehensive reading and writing messages that treat a whole subtree emanating from it. This is advisable if you have only a few different mechanisms in consideration. Be aware not to overload the application classes with too much communication methods.

2. *Applying the Strategy pattern [GHJV]*. If you have to consider a lot of different remote reading and writing policies (or need some runtime flexibility), implement them within separate strategy objects.

(In the following let again HOn-C and HOn-S denote the client and server part of the half-object pair with number *n*.)

Of course, in both cases, HO1-S has to offer more comprehensive synchronization messages than in the standard case above since now the synchronization of each half-object is carried out in one single step by the root half-object HO1.

The second approach needs some explanation. As can be seen in figure 5, the strategy object referenced by HO1-C is doing the job of coding and decoding all half-objects within the subtree under HO1-C. The strategy object accesses these objects via a back-reference to HO1-C.

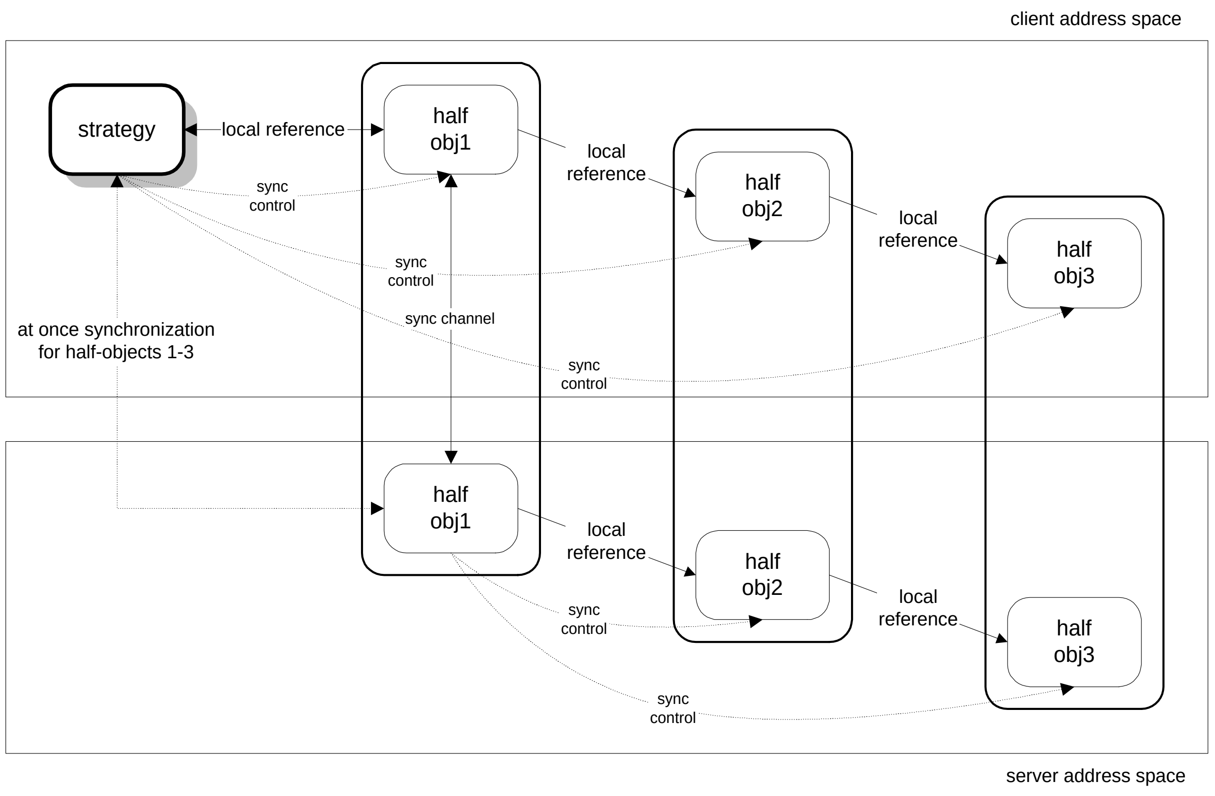

Fig. 5: *Half-object 1 using a strategy object for synchronizing several half-objects at once*

The solution with strategies can even be carried further. You can imagine to also equip HO1-S with a strategy to retain the client/server symmetry. It is left to the designer whether to use strategies on the server or not.

Resulting context

1. *Ability to implement a variety of mechanisms*. With this approach you eliminate all drawbacks of the half-objects' standard protocol described above. In addition, a variety of other policies come into mind, e.g. a locking mechanism for updates with pessimistic concurrency control, where all the read objects are shared or even exclusively locked in the database. This strategy can be combined with the "several objects at once" strategy to get extremely safe and comprehensive transactions.

2. *Runtime flexibility*. If you use the *Strategy* approach you preserve runtime flexibility: At runtime you can plug in different strategies for transaction and transmission of objects. The decision can even be left to the user by offering different strategies within the user interface of the system. The high-level code of the application is not affected since behavior changes dynamically.

3. *Increased complexity of the remote methods*. One effect should not be underestimated. Whether you implement the remote methods within the half-objects themselves or in separate strategy classes, the code of these methods remains complex anyway because you have to (re-)construct an

object hierarchy from a more or less flat data structure and vice versa. If the (de-)coding is very complex it is recommended to store it outdoors in a separate strategy class. By doing so the application domain classes are kept from being overloaded with technical details and remain easy to survey.

Related patterns

The *Strategy* pattern is discussed in [GHJV]. In case of preserving logical transactions the strategy object is a kind of transaction object as described in *Relational Database Access Layer* [CoKe2].

Acknowledgements

First I would like to thank Jim Coplien for shepherding this paper. His very insightful remarks, his experience and energy in working helped me a lot in qualifying this paper for EuroPLoP '97.

Thanks also to Wolfgang Keller for pre-reviewing the paper several times before submission.

References

[CoKe1] **J. Coldewey, W. Keller**: *Multilayer Class*, pre-print for the proceedings to PLoP96.

[CoKe2] **J. Coldewey, W. Keller**: *Relational Database Access Layer*, pre-print for the proceedings to PLoP96.

[BMRSS] **F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal**: *Pattern Oriented Software Architecture*, J. Wiley & Sons 1996.

[CORBA] **OMG**: *The Common Object Request Broker: Architecture and Specification, Revision 2.0*, July 1995.

[COSS] **OMG:** *CORBAservices: Common Object Services Specification, Revised Edition*, March 1995.

[Date] **C. J. Date:** *An Introduction to Database Systems VOL I*, Addison Wesley 1985

[FrBr] **M. Frese, F. Brodbeck:** *Computer in Büro und Verwaltung*, Springer 1989

[GHJV] **E. Gamma, R. Helm, R. Johnson, J. Vlissides:** *Design Patterns*, Addison Wesley 1995

[IsDe] **N. Islam, M. Devarakonda:** *An Essential Design Pattern for Fault-Tolerant Distributed State Sharing*, Communications of the ACM 39/10, October 1996

[Meyer] **B. Meyer:** *Reusable Software, The Base Object Oriented Component Libraries*, Prentice Hall 1994

[Mesz] **G. Meszaros:** *Half-Object + Protocol (HOPP)*, Pattern Languages of Program Design, Addison Wesley 1995, p. 129-132

[MoMa] **T. J. Mowbray, R. C. Malveau:** *CORBA Design Patterns*, J. Wiley & Sons 1997

[Rohnert] **H. Rohnert**: *The Proxy Design Pattern Revisited*, Pattern Languages of Program Design, Addison Wesley 1994, p. 105-118.