A small pattern language for Distributed Component Design

Kyle Brown
IBM Corporation
brownkyl@us.ibm.com
Philip Eskelin
CS First Boston
philip@eskelin.com
Nat Pryce
Imperial College, London

Introduction

This language is an exploration of the problem of building distributed systems using component technology. Unfortunately, there are many definitions of what the term "component" means, lending many possible interpretations to our language. As such we will seek to make things as simple and clear as possible, and to define our terms as we go, so that the confusion of the reader will be reduced.

For our purposes, a "component" is a software entity that fulfills a basic role in a system. A component has a well-defined set of points of interaction with a surrounding component framework. A component software framework, like JavaBeans, Enterprise JavaBeans, COM or COM+ supports the component and provides it with services (like distribution, transaction support, or persistence support) that the component may use. A component may be a single class in an object-oriented language, but it usually consists of several cooperating classes that work together with the framework to fulfill the component's role.

For example, an "Account" may be a component in a banking or stock trading system. It would provide a fixed set of access points (an API) to classes and components outside itself, and behave in a particular, predictable way. For example, the "Account" may provide methods to retrieve a balance, or make credits and debits. It would interact in a predefined and particular way with the relational database the account information is stored in, and would behave in a well-understood and reliable way when it is queried from multiple clients running in different threads on the same machine, or in different processes from different machines.

Our pattern format

We have chosen to represent our observations on these technologies, and our proposed solutions to the problems that we have found in the form of a Pattern language. In particular, we have chosen to use the Alexandrian pattern form that was first elucidated in [Alexander]. Alexander's pattern language has several key identifying features that we have chosen to use in our pattern language:

- Patterns are presented linearly, starting with the most general, and moving to the more specific.
- Patterns are written in plain, concise English, with a minimum of stylistic or typographic embellishment
- Patterns follow a particular, fixed format.

In particular, our pattern format will be:

- A *pattern name* in noun or noun phrase form in **large bold type**
- A short paragraph orienting the *context* of this pattern among the previous patterns
- A short *problem statement* presented in **boldface**
- A *discussion* of the problem, particularly focusing on why this problem is difficult and not trivially easy to solve. The discussion explores the forces in the problem, leading to a resolution of these forces where it ends with the word therefore:, leading up to
- A concise *solution* to the problem in **boldface**
- Any diagrams and/or further arguments that are necessary to make the solution understood
- A concluding paragraph showing how this pattern can lead to other patterns in the language.

We have chosen to omit a number of parts of the pattern format presented in [Alexander] such as the number preceding each pattern name (since we have a small pattern language, we do not feel that this is necessary), and the introductory diagram or photograph at the beginning of each pattern. We do not feel that these elements are essential to the pattern format, and we believe we can sufficiently convey the information in our language without them.

How this language came about

This pattern language has evolved over the last year as part of the Component Design Patterns effort began by Philip Eskelin on the wiki web (http://www.c2.com/ppr). Many of the patterns presented here were first proposed on the wiki web, where they received excellent comments and revisions from a number of contributors.

Overview of the Language

This language is a subset of a larger pattern language encompassing the design and use of component frameworks. In particular, These patterns investigate how to address issues raised by the use of a Component framework and an Object-Oriented language in designing distributed systems. The patterns in this mini-language are:

- Replicated Object
- Distributed Facade
- Object Factory
- Distributed Command

Together these four patterns form a micro-architecture for building distributed components that we have seen employed in many different problem domains and component technologies. In particular, they examine what results from using a distribution framework (like CORBA, or EJB's) that utilize proxies.

The reader of these patterns should have a basic understanding of the terminology and architecture of these distributed component frameworks. For a good overview of the CORBA Architecture, see [OMG1], or for a more complete introduction to CORBA with Java, read [Orfali]. For an introduction to the Enterprise Java Beans Architecture, the reader should refer to [Sun].

Component Design Patterns

Replicated Object

When building components in a layered architecture, efficiency and code management concerns often dictate an alternative to always using *Proxies*[1] for all objects.

The overhead of the number of network calls required to handle complex data manipulation in a system that only supports pass-by-reference (Proxy) is restrictive.

* * *

Proxy is such a powerful pattern that many programmers begin thinking that it is the complete solution to their distribution problems. However, proxy has the unfortunate side effect that *every* call to a proxy crosses the network. In many situations, this is not only too costly, it is unnecessary. For instance, imagine a simple stock-trading application. (See Figure 1: Original Design)

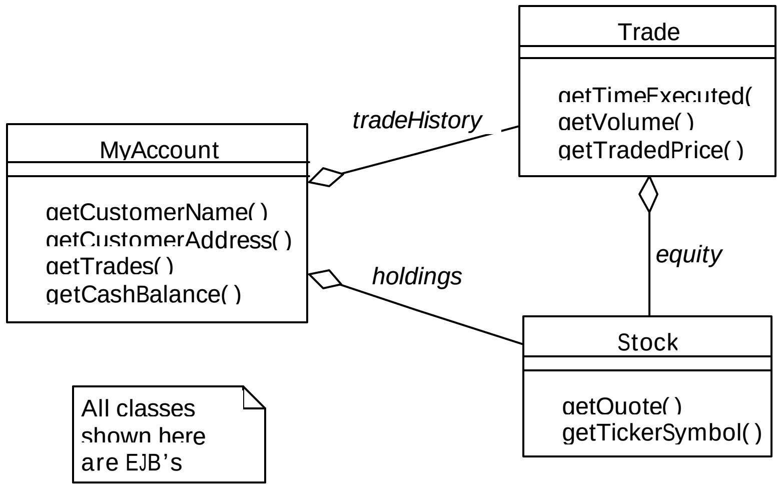

Figure 1: Original Design

Let's imagine that we use an Entity EJB to represent a customer Account. We will also create other EJB's which contain information about his holdings and the trades that he has made. Then, we want to display that information on the screen. Unfortunately, every single piece of information we need, the customer name, his address, his account number, the stock ticker symbols and the amounts of his trades, must be obtained through separate network calls. (See Figure 2: Remote interactions)

[1] Here we refer to the *Proxy* pattern from [Gamma]. In particular, we are referring to a *Remote Proxy* as described in [Gamma].

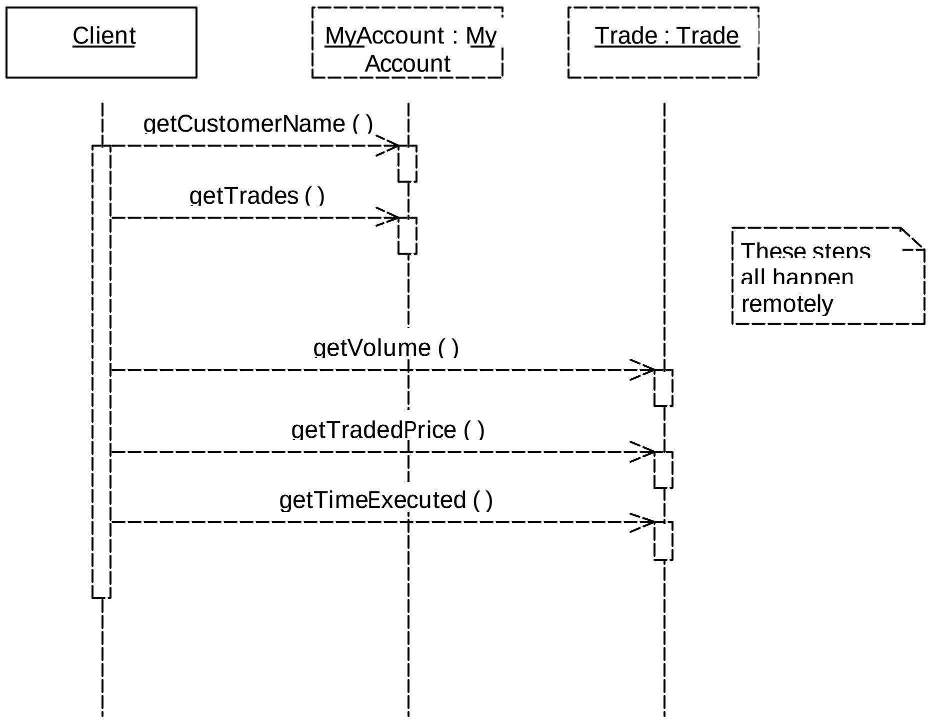

Figure 2: Remote interactions

This makes no sense, especially in the case where there are specific objects (like a Trade, or an Stock) that have been completely fetched from a back-end persistent store and do not vary from one method invocation to our Account EJB to another. Therefore,

Use a pass-by-value approaches for most of the business objects in your system. An object that is passed by value is "serialized" (or *marshalled*) on the server and "reconstituted" (or *unmarshalled*) on the client end. This object is then called a *replicate*. Programmers can choose which objects in their system will need to be manipulated on both ends of a client-server conversation, and replicate them.

So, in our example system, the Trade and Stock objects would be replicates that are passed all at once from the server (the Account EJB) to the client application that needs to display them. In EJB's, this can be done by making the Trade and Stock to be Serializable objects, or JavaBeans[2], rather than making them full EJB's. (See Figure 3: Serializable Objects)

[2] JavaBeans are by definition always *Serializable*

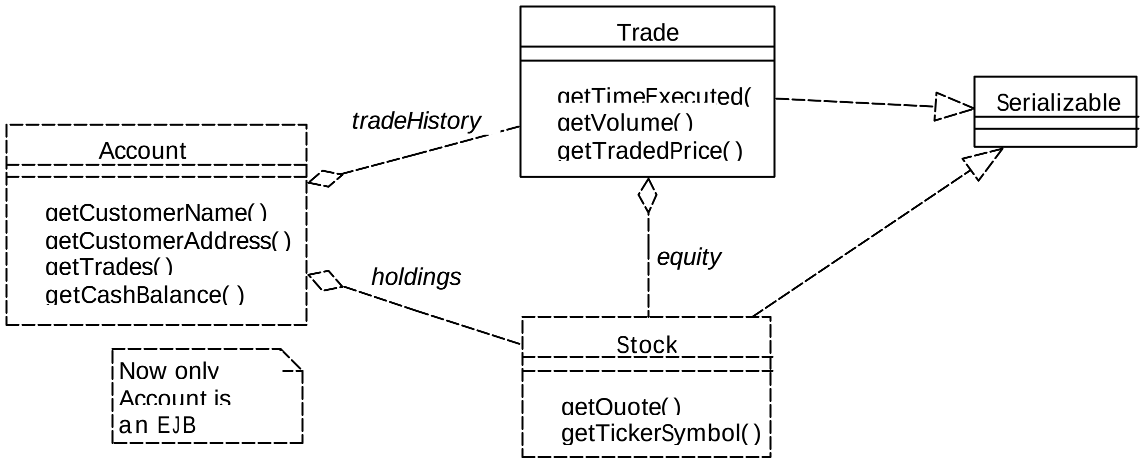

Figure 3: Serializable Objects

This change results in change in the distribution boundary – now only some of the messages from the client to the other objects are remote calls. Now, calls to the Trade are local. (See Figure 4: Local and Remote Mixed)

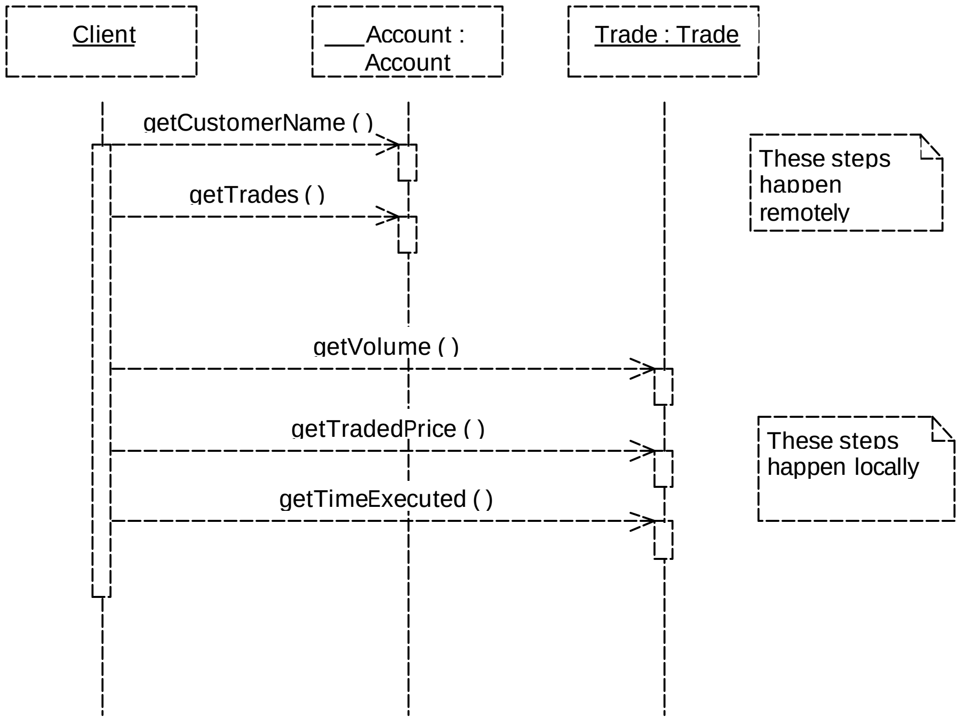

Figure 4: Local and Remote Mixed

Replication is particularly useful when there is a complex object nets that needs to be traversed. If each call to obtain a new "node" in an object net required a network call, the overhead from those calls would be too high to be practical in most applications.

This pattern first arose in the GemStone object-oriented database for Smalltalk, which provided both OODB and application server functionality. When working with GemStone programmers had the option of choosing to execute methods in either the server process space, or the client process space. This meant that at runtime an object could be declared to either be a replicate, or a proxy.

The set of objects that can be passed by value may be disjoint from the set that can be passed by reference (EJB) or may overlap that set (Java RMI). For example, the semantics of Java RMI are such that methods of Remote Interfaces may return any primitive or legal Java class, so long as that class implements the interface *Serializable*. In this way, objects are serialized on the server end when the method completes, and then deserialized on the client end and returned to the object that initiated the call to the proxy. So any Serializable object may be a replicate in RMI (and by extension, EJB).

However, it is not easy to implement this pattern in all distribution frameworks. For instance, in CORBA 2.2[3], there is no way to define an object that can be passed by value. CORBA 2.2 only provides for structs, analogous to C structures, that are data-only and do not allow for the definition of behavior. So if a programmer wishes to ask a distributed CORBA component for some information, and then manipulate that information on the client (receiving) end, he must first ask for a struct from the local proxy to the CORBA component, and then copy the information from that struct into another object that can manipulate the information. This copying must be hand-coded[4], and is prone to error and also prone to break when the definition of the struct changes in IDL.

CORBA will support passing objects by value in a later version – a Joint submission [98-01-18] on Objects by Value has already been approved. It does so by proposing a new IDL keyword (value) which allows for the creation of objects that have state and methods, but that are not descended from CORBA:Object and thus cannot be represented as an IOR.

When you begin to use Replicated Objects, there are a number of issues that you need to consider. The first is the issue of synchronization of replicates between the client and the server. One solution to this is to use dirty bits to record if the information that is stored in the replicate has been changed by the client. This does not solve the problem of reconciling differences between client and server when the information on the server changes, however. In this case, callbacks may be employed to update the replicate whenever the server changes. However, this requires the server to be aware of what is current on each client – this can lead to excess memory requirements[5] on the server, and network overhead in the number of calls needed to update each client.

Replicated Objects are often what is returned by the methods of a *Distributed Facade*. If *Replicated Objects* are sent from client to server and then later returned again to the server, the overhead of sending multiple copies of unchanging data can result in performance problems. This naturally leads to *Distributed Commands*.

Facade at the Distribution Boundary[6]

In a component distribution system that supports both *Proxies* and *Replicated Objects*, *Facade at the*

[3] See [98-07-01]
[4] In the IDL/Java mapping a CORBA "Struct" is mapped to a Java class, but this class is generated by the IDL compiler. You would not implement business methods on this class because these methods would be lost when the IDL is recompiled. For this reason, designers often have "shadow" classes that contain the same information plus business methods to operate on this information as part of their models. Information is copied from a "shadow" class to the generated class and vice-versa.
[5] This is because the server must keep a record of all the replicates that have been made so that it can callback to them with updated information. An alternative solution to this is to send ALL updates to an object on the client and then let the client decide which objects should be updated from that information.
[6] This pattern has been previously documented (but not in pattern form) in [Alpert] in the section on Facade, and in [Fowler].

Distribution Boundary allows the programmer to strike a balance in number of objects that are made available to the network.

Many solutions using Proxy contain within themselves the seeds of their own destruction at the hands of inexperienced designers. When Proxy is used to excess the sheer number of remote interfaces becomes unwieldy. When using Proxy, a system must preserve part of itself as relatively static and unchanging – otherwise the changes to the clients become problematic.

*** * ***

When we first come into building distributed systems from building monolithic, non-distributed systems, we tend to bring a lot of design strategies that no longer serve us as well in the new context Consider the following manufacturing scenario. Let's say we're building a system for a car factory. Without much thought we identify our first object -- let's call it a Vehicle. With some more thought we come up with a few more; for example, a BodyStyle that defines the similar properties of a set of Vehicles. We might even come up with the idea of BuildInstructions that say "do this particular thing" which are combined together into WorkOrders that are used to create Vehicles having a specific BodyStyle.

Now this works fine in a single process-space system. We can do all sorts of nifty things using these objects. We can create reports on what instructions to execute, and which ones have been executed. We can find out what Vehicles are currently in production, and how many of what BodyStyles are being built, and change, add and update all of the above objects.

Now, consider the following problem -- we need to distribute this system using CORBA. We want client machines to run the GUI's, while bigger server boxes handle most of the processing. We also want to split the processing into the parts of the system that handle the robotics (which must never go down) and the parts of the system that handle reports (which can go down occasionally).

The naive programmer says -- "No problem. CORBA gives us proxies, so we'll just take our existing objects and write IDL interfaces for them." But they soon discover that that they are then writing CORBA interfaces for nearly every object in our system. Not just Vehicles, BodyStyles, etc. but also for the things they contain like PaintColors and Accessories. Suddenly they have a LOT of IDL, and that in itself becomes a problem.

Also, they start to notice that they are crossing the network a LOT. Objects in one process space are sending hundreds of messages to closely linked objects in other process spaces. Every change to the system requires a recompilation of IDL, and a recompilation of other classes. Testing becomes problematic, since every test must be done over the network. There must be a way to limit the number of remote interfaces. Therefore:

Take a different approach. Start to look in a design for the groups of objects that are closely linked together, and bind them together inside a single process space. Define a few remote interfaces between these new groups. In other words, apply the *Facade*[7] pattern, e.g. build new objects that act as gatekeepers that hide the complexity (and sheer numbers) of the objects they wrap. This results in fewer remote interfaces to manage, while the facades help determine which messages really need to cross the network, and which can stay local.

The following benefits and liabilities apply to using Distributed Facades:

- *Less flexibility.* When a component acts as a facade that contains many smaller components, one tradeoff can be that adding new components means you must update the interface and implementation of the facade component and test to ensure that all existing components still work properly.
- *Easier to manage change.* A benefit of the facade is that you are in effect wrapping what would be separate smaller physical components with one larger physical one, then allowing logical access to each component inside it. You present a "view" into these components with the facade. Each of these components can share a common infrastructure and operate off of the same framework. They

[7] This refers to the *Facade* pattern from [Gamma]

can reuse standard libraries, and reduce version discrepancy headaches that sometimes happen in less-controlled development, test, or production environments.

We first applied the Facade pattern in this way when refactoring a large GemStone software project. We had the problem of wanting to reduce the distribution cross-section to minimize network traffic and swapping between the local and distributed object spaces (We were using a pass-by approach, e.g. *Replicated Objects* for most of our objects).

Later we applied this pattern in an options trading system that we developed with a client, where we developed a set of "services" that each did one key thing like "trade options" or "handle quotes". Each service wrapped up many of domain objects within a relatively simple Facade API that it presented to the other services.

A *Distributed Facade* must interact with the rest of the parts of the system. It will often use an *Object Factory* to create the *Replicated Objects* that it returns to the client. It may also be able to use and interpret *Distributed Commands* that are sent to the server to determine what changes need to be made to components on the server.

Object Factory

Replicated Objects must be created from within a *Facade at the Distribution Boundary* from other data sources.

Component factories are an effective solution for the creation of distributed components. However, Replicated Objects are not full-fledged components and do not have the same level of support in the environment as components do. As such, allowance must be made for the creation of these objects.

* * *

Most component frameworks provide a facility like a component factory[8] that allows distributed components to be built without programmer intervention. However, Replicated Objects are not as strongly supported in these environments, since the environments do not provide for factories to create them in the same way that distributed components can be created.

A Replicated Object, in this sense, is not a “full-fledged” distributed component like an EJB or a CORBA component, but is something else. When developing designs that use replicated objects it is a good idea to follow this same factory pattern, though, as the same arguments about centralized object creation and lifecycle management apply as well to Replicated Objects as they do “full-fledged” distributed components.

Consider the following architecture, again derived from our Stock trading example. In this design a Facade (in our case a AccountFacade, which is a Session EJB) acts as a large-grained distributed component that provides services to a client GUI application. One of the services provided by this EJB is to return a list of available Trades for viewing. Let’s further assume that both our Trades and Stocks are Entity EJB’s in this version of the model.

An TradeBean is a Replicated Object that is manipulated by code inside of the GUI client. The Session EJB must have some way of obtaining the information that makes the Trade from the database. The data in the database is represented in the program by a set of Entity EJB’s that wrap existing database tables. We thus have three parts of our application represented, but there is a hole in our design – there is no way to create the TradeBean from the Entity EJB, and there is no way to update the information in the EJB’s if a TradeBean changes.

[8] A *component factory* is a special factory object for distributed components like an EJB Home, or the factories specified in the CORBA Lifecycle service.

What is needed is something like a Component Factory, but one that works for Replicated Objects. We want to be able to create (and possibly cache or reuse) Replicated Objects, and to manage for updates when needed. Therefore:

Create Object Factories for each type of Replicated Objects that are responsible for creation, update, and instance management of these objects. These factories will participate in Managed Transactions, and provide services to a Distributed Facade.

Our sample application (with the addition of an Object Factory, *TradeFactory*) would then have the following design (See Figure 5: Object Factory Example):

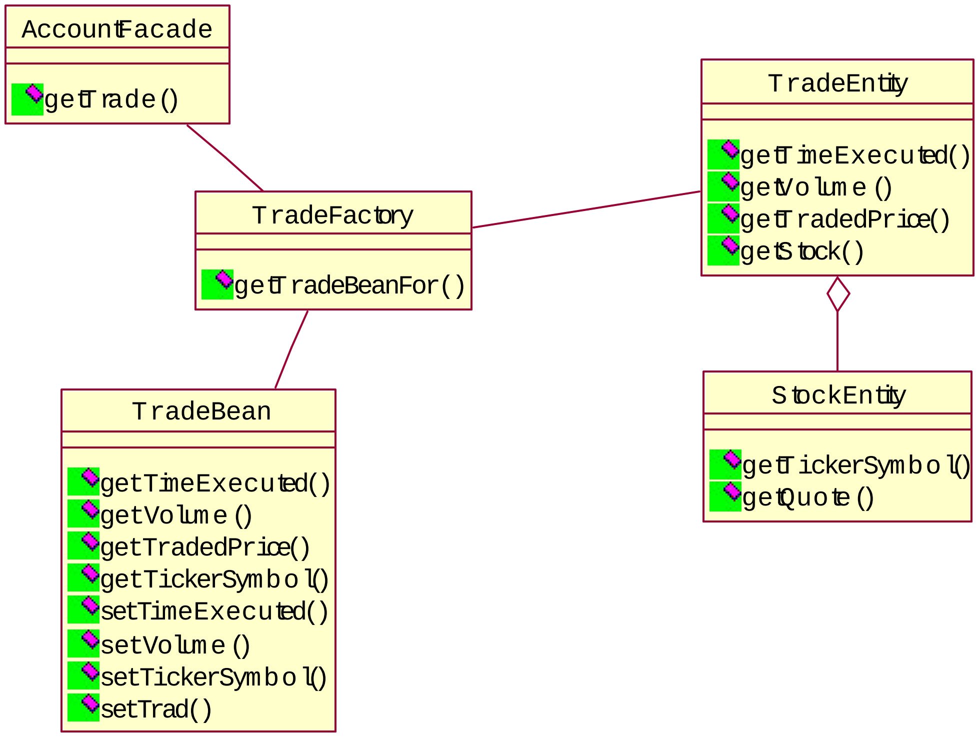

Figure 5: Object Factory Example

We would like to create a *TradeFactory* that responds to request from the Session EJB to create a *TradeBean* for a particular key value. It would accomplish this by using the Homes (Component Factories) of the appropriate Entity EJB's to locate the EJB's that contained the needed information, and then copy that information into a new Trade, which it would return to the Session EJB. Likewise when an update occurred, the Session EJB would instruct the Object Factory to carry out the update – it would locate the appropriate Entity EJB's and then update them within a single transaction regulated by the Session EJB. A simplified version of this interaction (leaving out locating the EJB Homes) is shown below (See Figure 6: Object Factory Interaction):

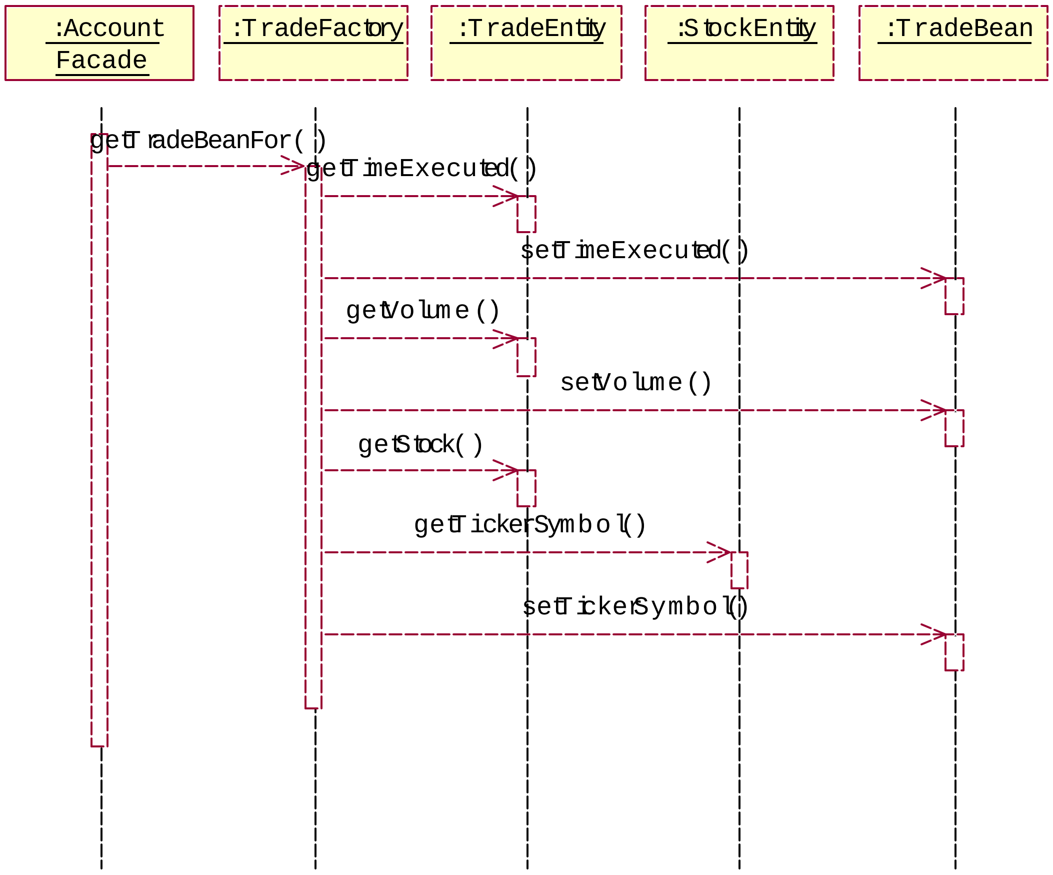

Figure 6: Object Factory Interaction

Note that this pattern differs from several other published patterns that serve similar purposes. For instance "Relational Database Access Layer"[9] talks about the problem of taking information from a relational database and creating objects from that information. In our case, the information may already be in the form of an object (such as an Entity EJB), but not in a form that is suitable for replication. In other cases, your Object Factory may, in fact, act as a facade onto a Relational Database Access Layer. For instance, a common way of using the TopLink Object-to-Relational mapping framework with EJB's is to use it's built-in Object Factories to create objects that can be obtained by clients of a Session EJB acting as a Distributed Facade.

There is also a degenerate case of this pattern that is often seen in EJB designs. In the degenerate case, the Entity bean itself acts as the replicated object factory and creates a JavaBean that contains a copy of the information that it contains. For instance, WebSphere 3.0 and VisualAge for Java 3.0 contain a set of code generators to generate "CopyHelper" JavaBeans that contain copies of the information in Entity beans. These CopyHelper beans are also capable of synchronizing this information back to the database. It accomplishes this through the generation (on the Entity EJB) of two methods called copyToEJB() and copyFromEJB() that update and read the information from the Entity Bean respectively.

[9] Found in [Keller]

The problem with this degenerate case is that it assumes that "one size fits all". It assumes that all clients of an Entity bean will want the same (complete) set of attributes from the Entity Bean copied to a JavaBean. When you begin to consider Entity Beans that have relationships to other Entity Beans (like our example) it becomes clear that this may cause problems as the question arises as to how "deep" the copy should be. Having external factories is a more flexible solution in that it allows for the construction of different JavaBeans containing different subsets of the data in the Entity Bean(s).

Distributed Command

When developing a system that uses *Replicated Objects*, objects must be sent in both directions across the network connection (from client to server and server to client). This can cause efficiency concerns and make programming difficult.

When you employ a replication solution, you now face the problem of how to send updates to replicated objects from the client to the server. If you send the entire changed object back across the network you may be sending much more information than is necessary, since most of the object structure will not have changed. This is not a very efficient solution, since the same data crosses the network twice. It also makes planning for transactions complex, since the replicates (not being full-fledged components) are not transactional objects.

* * *

For instance, let's suppose you are working on a system that allows a user to modify a complex, highly interrelated object model. Consider a genomics system that tracks genetic markers through a family tree in order to pinpoint how genes are inherited. There are at least three axes of information that users would be interested in:

- The family tree itself (who descends from who)
- The information about individuals in the tree (who showed what symptoms and who's assays showed them to have what markers)
- Information about the genetic markers (what markers are used, where they are found on the genome, etc.)

The problem is that the three axes are interlocking. Modifications to one object can have serious ramifications to other parts of the system. For instance, if a marker is changed, it's relation to the individuals must change. Likewise if an individual's genetic assay is found to be incorrect and changes, then statistics and calculations about the family and markers might now be incorrect. There are two "standard" approaches to maintaining the consistency of this information that can be tried:

- You can apply pessimistic concurrency on the entire object structure. In this case, a large chunk of the object structure is locked when the first user requests it. This has the drawback that other users are kept from modifying the structure at the same time -- something that is not reasonable in a multiuser environment.
- You can utilize optimistic concurrency on the entire structure. In this case, users are allowed to modify the structure as they choose, but the first one to "commit" his changes "wins". Anyone who had also modified those same structures would "lose" and find that their changes were now lost.

A third approach, versioning, can also be tried. In this case, a new "Version" of the structure is created for each user. However, this just trades the current problem for a different, but equally difficult, version reconciliation problem. Therefore, since none of the previous solutions have worked, try the following:

Encapsulate the user changes as *Commands*[10] and then treat the group of commands as a single command that executes within a transaction boundary. Use a strategy like Two Phase Commit to merge commands

[10] Here we refer to the *Command* pattern from [Gamma]

issued by different users together.

In a distributed system this solution becomes even more attractive. Imagine that our hypothetical genomics system was built using a layered architecture[11], with the genetic objects being *Replicated Objects*. Further, imagine that we applied *Facade at the Distribution Boundary* to encapsulate the "real" interactions of the domain model so that the GUI front-end only communicated with the business model through the intermediaries of the facade components.

In this case we find that the Command pattern applied in this way not only helps with the concurrency control of the system, but provides a significant benefit in that the changes that are sent from the upper (presentation) layers of an application to the lower layers are sent in the form of "deltas" to objects, rather than full copies of the objects themselves. This reduces the amount of network traffic, and reduces the amount of logic needed on the server side to determine which parts of the model have changed and which have not.

Acknowledgements

I would especially like to thank our Shepherd, Robert Hirschfield for all the hard work he put into commenting on this paper and the great suggestions for improvement that he made.

Bibliography

[98-01-18] OMG TC Document orbos/98-01-18, "Objects By Value" Joint Revised Submission with Errata, Object Management Group

[98-07-01] OMG TC Document 98-07-01, "The Complete CORBA/IIOP 2.2 Specification", Object Management Group

[Alexander] Christopher Alexander et.al., *A Pattern Language: Buildings, Towns, Cities*, Oxford University Press, London, 1977

[Alpert] Sherman R. Alpert, et. al., *The Design Patterns Smalltalk Companion*, Addison-Wesley Longman, Reading, MA, 1998

[Buschmann] Frank Buschmann, et. al., *Pattern-Oriented Software Architecture: A System of Patterns*, John Wiley and Sons, West Sussex, England, 1996

[Fowler] Martin Fowler, *Analysis Patterns*, Addison-Wesley Longman, Reading MA, 1996

[Gamma] Erich Gamma et.al., *Design Patterns: Elements of Reusable Object-Oriented Design*, Addison-Wesley Longman, Reading MA,1994

[Keller] Wolfgang Keller and Jens Coldeway, "Relational Database Access Layer", in *Pattern Languages of Program Design 3*, Addison-Wesley, 1998

[OMG] "CORBA For Beginners", part of the OMG web site; **http://www.omg.org/corba/beginners.html**

[Orfali] Robert Orfali, et. al., *Client/Server Programming with CORBA and Java, Second Edition*, John Wiley and Sons, 1998

[Sun] "EJB Learning Center"; **http://java.sun.com/products/ejb/ training.html**

[11] As in the *Layers* pattern in [Buschmann]