

A Component Distribution Pattern Language

Kyle Brown

IBM Corporation

brownkyl@us.ibm.com

Philip Eskelin

CS First Boston

philip@eskelin.com

Nat Pryce

University of Manchester

Introduction

A note to our shepherd: This is a work in progress—the deadline for PLoP submission snuck up on us and we have been forced to submit a version of the language that needs a great deal of work. Once you have decided to shepherd our paper, please contact Kyle immediately to receive a more recent version, as work will continue during the paper selection phase of PLoP.

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 unique software entity that fulfills a basic role in a system. In our particular case, “components” are built within the context of a component software framework, like JavaBeans, Enterprise JavaBeans, COM or COM+. A component may be a single class in an object-oriented language, but it usually consists of several cooperating classes that together with the framework help the component fulfill its role.

For example, an “Account” may be a component in a banking system. The Account would have a well-defined set of responsibilities, and a well-defined set of points of interaction with the surrounding component framework. 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, 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 *A Pattern Language* by Alexander, et. al. 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 [Alexander77] 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. As a result of what has been done on wiki, we have decided to revisit some of the assumptions underlying the patterns, and to rewrite the patterns in strict Alexandrian format. Thanks to Brad Appleton for suggesting this course of action, which has turned out to be a very useful and helpful way of rethinking the goals of our language, and assessing how well we have met those goals.

Overview of the Language

So, we will begin our language with a discussion of the three parts of our language, which address three different levels of component software design. They are:

- Component Architecture – which discusses the basics of building components systems, such as how components fit into larger software architectures, and what basic principles components must follow.
- Component Implementation – which investigates the common principles found in the design of most component software frameworks, and examines the implications that these principles have on the design and implementation of software components.
- Component Design – Common principles that naturally arise in systems built using a Component Framework, and how they address particular issues in overall systems design.

Component Architecture

Layered Architecture

The need for a *Layered Architecture* is a primary driving force in component design. It leads naturally to a desire for *Location Transparency*, which drives the requirements for other patterns in the language.

Building modern business software is a complex process that requires the skills and contributions of

many people in a team. Most programs today require interaction with many kinds of existing software systems, such as graphical user interface systems, web browsers, relational databases, and other systems. To be maintainable and reliably debuggable, the concerns of these disparate systems must be kept separate from each other so that systems can be readily extended and maintained.

It is rare to find a single person that understands all of the complex systems used in a single large program. For instance, you can not often find someone who has extensive experience in both relational database programming and Java Swing GUI programming, since these two skill sets are widely separated. For this reason, most organizations have turned to building software with interdisciplinary teams in which each person brings part of the total skills necessary to build the system to the table.

The desire to let members of a team specialize has an impact on the overall design of the program. If a program design does not allow team members to “own” parts of a program (at least for a period of time) in which they can exercise their specialties then the resulting development process will be characterized by strife and ongoing arguments as people try to force their viewpoints onto the design of the parts of a program. Therefore:

Design programs with layers where each layer is characterized by a single, unifying concept. Each layer will usually encapsulate a single technology (such as Java Swing programming, or Relational Database programming) in such a way that people can work on objects within a single layer and exercise their specialties to the greatest extent.

As an example, consider the following software architecture, described in [Brown95]

```
// ** describe a four-layer architecture **//
```

When software is designed in layers, with specializations being unique to a particular layer, then human conflict is minimized in the development process. A side effect of this is that once people become accustomed to the idea of separating responsibilities into different objects residing in different layers of software, then they then see a logical extension of this philosophy. This extension is the idea that layers may reside on different machines, or in different processes, taking advantage of multiprocessing, and taking advantage of the sharing of data and processing. This desire to split layers in this way, coupled with the desire to reduce the amount of effort necessary to implement such a split, leads to the notion of *Location Transparency*.

Location Transparency

Layered Architectures often lead to a desire for location transparency as distribution becomes an option when overall systems complexity is reduced.

When systems are built with distributed components, it is desirable to reduce the coupling between the knowledge the existence of a component and its physical location. High coupling in this area leads to systems that are brittle and difficult to refactor.

The simplest way to build a distributed system is to hard-code the knowledge of the location of the components into the components themselves. For instance, if a Stock Trading Service needs to get the latest prices from a Price Quoting Service, then a simple solution is to hard-code that reference into the Stock Trading Service. However, this has some significant drawbacks.

If a requirement of the system was that it should be highly available, then it is very likely that each Service would have a backup service running in parallel to it. If the Price Quoting Service went down, then the Stock Trading Service should be able to switch to the backup. However, this adds complexity to the system. What if you wanted two backups? Each additional measure of safety adds more coding complexity into the Stock Trading Service, and more highly couples it to its existing environment.

To solve this problem we must make the location of each distributed component somehow separate from the knowledge of its existence. Therefore:

Build systems such that clients requests can be satisfied in the same way regardless of whether the

object providing the services resides in the same machine or process space, or in a different machine or process space. Provide a way to translate general requests for something that can handle a request into a specific object reference. This can decouple the parts of a system and provide for better redundancy and reliability.

Doing this results in less coupling between components. If a client only knows an abstract "handle" to a server, and knows that the server implements a particular interface, then the client is protected from implementation changes in the server.

In the CORBA world, the notion of Location Transparency has been implemented in two particular CORBA Services; the CORBA Naming Service, which allows objects to be identified by name independently of the machine the run on, and the CORBA Trader Service, which matches requesters of services with providers of services.

The Enterprise JavaBeans framework builds on the CORBA naming service by providing a single, common interface by which objects are obtained over the network. The specification states that JNDI (the Java Naming and Directory Service) must be used as the only way to obtain a remote reference to an EJB over the network.

(D)COM also provides Location Transparency via the registry database. The registry generally replaces the INI files we used to store and retrieve settings back in the old days. Under HKEY_CLASSES_ROOT, you see a bunch of programmatic IDs (Prog IDs) and one key that contains the class IDs (HKEY_CLASSES_ROOT\CLSIDS). This allows the creator of a COM component to use the ProgID or CLSID without knowing its location.

However, this solution has in it a new potential problem, e.g. it creates a new single point of failure. The central "broker" component that exchanges abstract handles for actual references becomes a single point of failure. The entire system can fail if that component becomes unavailable.

/ ** Add in the links to the rest of the language **/

Component Implementation

Proxied Object

// ** This is simply a retelling of Proxy, as presented in [Gamma] and [Buschmann]. Make it short and sweet. **//

Component Factory

When building distributed objects that participate in a *Layered Architecture* we find the need for a common way of constructing *Proxied Objects*.

One of the biggest problems in distributed programming is how to manage the lifecycles of remote objects. Creating, locating and removing Proxied Objects are tasks that are central to any distributed system. If care is not taken in creating these objects, memory leaks can occur as useless, unreachable objects remain in the system, occupying resources. Nonstandard ways of constructing distributed objects can lead to confusion and misunderstanding as conflicting solutions are used.

One of the first questions asked by someone learning an component distribution technology is how is a component created? This question is central to understanding how to best make use of this technology. The problem is that most books and manuals for technologies like CORBA and RMI do not spend enough effort discussing this problem, leading to conflicting solutions.

Here is the key: Books on CORBA or RMI that focus on introducing the technology can only present so much code in an example and yet have that example be understood by the readers who wish to learn this technology. As a result, only small examples can be shown, usually focusing on demonstrating a particular

feature of the particular distribution technology. In such cases, the most common way to demonstrate a feature is to create a single server object, usually in a main() method that shows the feature that is being illustrated.

The problem is that while this works for a single object in an example, it is not a scalable solution. Unfortunately, many people have followed this approach in their programs, even though it does not provide for the removal, location, or recovery of the objects that are created in this way. So, since this approach cannot scale, we must find another way of creating our objects. Therefore,

Define a standardized way of creating, locating and destroying both transient and persistent components. The way to do this is to define object factories for each component type that is responsible for lifecycle management. Standardized methods can allow these factories to be queried through an interface discovery mechanism, which helps in implementing Location Transparency.

/** More discussion **/

// ** Links to other patterns **/

Container Independence

Container Managed Persistence

/** Add the context links to the previous patterns **/

In many cases, developers naively make persistence part of the nature of their components. A particular persistence strategy is hard-coded into the domain and is not separable from it. This has a substantial impact on the reusability and scalability of an application.

Consider the following scenario: A group of programmers sets out to build a new order processing system. They find two sets of components they need to build (roughly "models" and "views") and begin working on coding their components. They decide that their system will use Java Serialization to persist its Orders. They make this decision because it is simple, and because only one department of only a few people will ever *use* this system. They go ahead and code the persistence directly into their models, with some supporting code in the application-level "glue" that ties the models and views together.

Then two months after they ship, disaster strikes. Marketing discovers their little application. They want to roll it out onto the web for thousands of customers to use. When the developers review their code they find that they will have to recode over half of the application because of how deeply embedded their persistence code is. How could they have avoided this needless pain? Obviously, the problem is one of factoring. You cannot embed a persistence strategy into the code, yet persistence is needed to make applications work. Therefore:

Use a separate object (a container) to manage persistence. As a container may contain many differing components, it makes sense to centralize their persistence management. Moving the responsibility to the container provides consistency across all components in that container, and eases the burden on the component developers.

The container has a specific contract with the objects it contains. In short:

- The container defines a persistent format in terms of the containment but that leaves the representation of the contained components unspecified. Ie: the format is all "packaging" but no "content".
- Contained components define a persistent format in which to save their internal state and leave the "packaging" details to their container.
- The container makes itself persistent by writing out "packaging" data and delegating to the

contained components to fill in the "content" of that packaging.

```
// ** Add the links to later patterns **//
```

Managed Transactions

```
// ** This pattern addresses the problem of transaction context and propagation of this context through multiple components. **//
```

Component Design

Replicated Object

When building components in a *Layered Architecture*, efficiency and code management concerns often dictate an alternative to always using *Proxied Objects* 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 (Proxied Objects) is restrictive. Likewise, schemes that allow only structured data to be passed among cooperating processes, rather than true objects, results in messy object designs, and can result in high overhead due to copying data from structures into objects.

```
//** Describe the CBOE CORBA experience **//
```

Therefore

Allow for pass-by-value approaches for some objects. An object that is passed by value is “dehydrated” on the server and “reconstituted” on the client end. This object is then called a *replicate*. The set of objects that can be passed by value can be either disjoint from the set that can be passed by reference (EJB) or can overlap that set (Java RMI). Programmers can choose which objects in their system will need to be manipulated on both ends of a client-server conversation, and replicate them.

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 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.

```
// ** Discuss the EJB CORBA pass-by value semantics **//
```

```
//** Wrap up with pointers to later patterns **//
```

Distributed Facade

In a component distribution system that supports both *Proxied Objects* and *Replicated Objects*, *Distributed Facade* allows the programmer to strike a balance in the ways in which objects are made available to the network.

Many solutions using Proxied Object contain within themselves the seeds of their own destruction at the hands of naïve designers. When Proxied Object is used to excess the sheer number of remote

interfaces becomes unwieldy. When using Proxied Object, a system must preserve part of itself as relatively static and unchanging – otherwise the changes to the clients become problematic. Also, when tightly linked objects are placed in different process spaces system performance suffers because of network overhead.

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 we soon discover that we're 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 we have a LOT of IDL, and that in itself becomes a problem.

Also, we start to notice that we're crossing the network a LOT. Objects in one process space are sending hundreds of messages to closely linked objects in other process spaces. We're constantly translating from internal (Java) objects into CORBA structures and vice versa. Our system performance is abysmal, and the reliability takes a nosedive. So, apparently this is not the right path. We need to limit the number of network calls, and also limit the number of remote interfaces. Therefore:

Take a different approach. Start to look in your 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 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 can reuse standard libraries, and reduce version discrepancy headaches that sometimes happen in less-controlled development, test, or production environments.

I 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 Business Objects* for most of our objects).

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

This pattern has been documented previously in *The Design Patterns Smalltalk Companion* in the section on Facade, and in Martin Fowler's book *Analysis Patterns*.

// ** add links to other patterns **//

Object Factory

Replicated Objects are not full-fledged components. As such, allowance must be made for the creation of these objects from components available in the system.

// ** need to fill in the rest of the pattern beyond the context **//

Distributed Command

When developing a system that uses *Replicated Objects*, object transfers must occur in both directions across the network connection. 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, and 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, its 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" and then treat the group of commands as a single

command. Use a strategy like TwoPhaseCommit to merge commands 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*, with the genetic objects being *Replicated Business Objects*. Further, imagine that we applied *Distributed Facade* 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 *Distributed 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 (GUI) layers 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.

// ** Add the links to other patterns **//