Crossing Chasms: The architectural patterns

Kyle Brown

Knowledge Systems Corporation

4001 Weston Parkway

Cary, NC 27502

kbrown@ksccary.com

ABSTRACT

This paper is part of a large pattern language (Crossing Chasms) currently under development by Bruce Whitenack of The Object People and myself. The pattern language as a whole addresses the issues faced by teams trying to build large client-server systems using Object Technology and Relational data stores.

This section of Crossing Chasms addresses issues faced by project teams who are just beginning to develop an architecture for client-server systems built using Object Technology. This paper contains four patterns:

- Four Layer Architecture addresses how proper division of labor between the objects in a system can result in higher reuse potential and easier manageability.
- Three tier architecture shows how systems can be organized to best take advantage of the machine resources of an organization.
- Phase-in tiers shows how systems using a four-layer architecture can be built initially for a two-tier architecture, and then moved to a three-tier architecture over time.
- Trim and fit Client demonstrates how the objects in a four layer architecture can be best distributed into the tiers of a two or three tier architecture.

Together these patterns can help a project team make the right decisions to arrive at an extensible, maintainable, manageable architecture for client-server computing.

PATTERN NAME: FOUR LAYER ARCHITECTURE

Problem:

When designing an object system for a client-server environment what is the most appropriate way to structure the overall application architecture?

Forces:

When designing the software architecture in a client-server system, you must come up with a way to divide the labor among team members. Your architecture must also be simple enough to be easily explainable to new team members, so they can understand where their work fits.

In looking for an application architecture, many developers have looked to the pioneering MVC architecture. However, MVC is not the be-all and end-all of object design. While a proper architecture should address the concerns addressed by MVC, and may trace its descent from MVC, modern software systems must also address issues not covered by classic MVC.

MVC promoted reuse by factoring out the UI widgetry away from the domain objects. Modern class libraries derived from MVC have also discovered yet another set of potentially useful and reusable abstractions in separating out the aspect of mediating between views and adapting views to domain models into another set of classes.

However, this still does not address the connection of the domain to the outside world (i.e., object persistency mechanisms, network protocols, etc.). A complete architecture for client-server systems must address these issues as well. Therefore:

Solution:

Factor your application classes into four layers in the following way (see Figure 1: Four Layer Architecture):

- **The View layer**. This is the layer where the physical window and widget objects live. It may also contain Controller classes as in classical MVC. Any new user interface widgets developed for this application are put in this layer. In most cases today this layer is completely generated by a window-builder tool.
- **The Application Model layer.** This layer mediates between the various user interface components on a GUI screen and translates the messages that they understand into messages understood by the objects in the domain model. It is responsible for the flow of the application and controls navigation from window to window. This layer is often partially generated by a window-builder and partially coded by the developer.
- **The Domain Model layer**. This is the layer where most objects found in an OO analysis and design will reside. To a great extent, the objects in this layer can be application-independent. Examples of the types of objects found in this layer may be Orders, Employees, Sensors, or whatever is appropriate to the problem domain.
- **The Infrastructure layer**. This is where the objects that represent connections to entities outside the application (specifically those outside the object world)

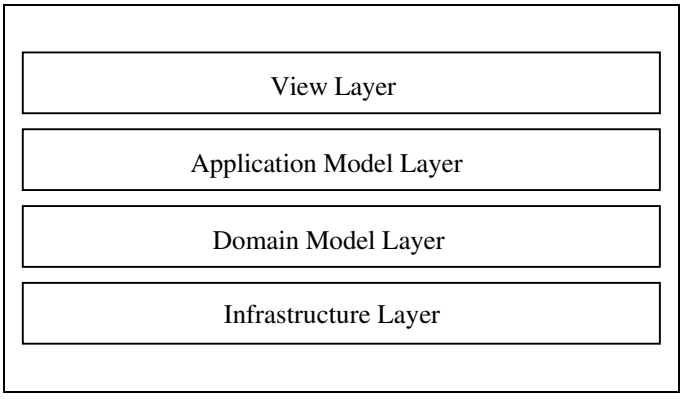

Figure 1: Four Layer Architecture

reside. Examples of objects in this layer would include SQLTables, 3270Terminals, SerialPorts, SQLBrokers and the like.

Discussion:

This choice of layers can have many beneficial effects on your application if it is applied in the proper way. First, since the architecture is so simple, it is easy to explain to team members and so demonstrate where each object's role fits into the "big picture".

If a designer is very strict about clearly defining where objects fit within the layers, and the interfaces between the layers, then the potential for reuse of many objects in the system can be greatly increased. A common problem with many object designs is that they are too tightly constrained to the limits of the particular application being built. Many novice designers tend to put too much of the logic of an application in the Application Model layer. In this case, there are few, if any, domain objects that are potentially available for reuse in other applications.

Another benefit of this layering is that it makes it easy to divide work along layer boundaries. Woolf demonstrates how a "layered and sectioned architecture" can be made the basis of a source-code control system. It is easy to assign different teams or individuals to the work of coding the layers in a four-layer architectures, since the interfaces are identified and understood well in advance of coding.

Finally, a four-layer architecture makes it possible to code the bulk of your system (in the domain model and application model layers) to be independent of the choice of persistence mechanism and windowing system.

Sources:

Layering is not a new idea in computer science -- Tannenbaum mentions it in conjunction with the OSI seven-layers communications model. Shaw discusses layering as an architectural choice.

Hendley discusses the benefits in portability gained by additional layering in the View and Application Model layers in Smalltalk. Brown further investigates the reasons for applying four-layer architectures for Smalltalk.

Related Patterns:

Trim and Fit Client shows how a four layer architecture can be used in conjunction with a 3-tier machine architecture in a distributed object environment.

PATTERN NAME: THREE-TIER ARCHITECTURE

Problem:

How do you distribute responsibility among the different machines in an enterprise to best take advantage of each platform's unique capabilities?

Forces:

Many organizations plan large-scale client-server projects by planning for a large capital purchase of desktop machines and network servers, to be purchased along with the development of new software. However, the technology used to develop the software will often change faster than the plan anticipates. Several releases of the target operating system may occur between the time a large project is started and its final delivery.

Because of the above, the client machines purchased are often not up to the final size of the software that is produced. It is not uncommon to see client-server applications in production today where the total amount of client code resident in memory at any time is 12 megabytes or higher.

It must also be kept in mind that the number of clients in a system will be from one to three orders of magnitude greater than the number of servers in a system. This multiplier will heavily weight the cost of a system towards that of the client. If the choice is to buy additional memory and a faster processor for 100 clients, or for one server, the choice is fairly obvious. Therefore:

Solution:

Utilize a machine architecture that splits responsibility into three "tiers" of computation. These tiers are:

- **The Client.** The client should be primarily responsible for the display and interpretation of information. It is the focal point for user interaction with the system as a whole. As such, the client can be optimized for display and fast network access, but may not need to have the memory and computational power available in other tiers.
- **The Departmental server.** The departmental server is usually a dedicated high-end PC-style machine or a specialized UNIX workstation. The server is capable of handling many more computations per second than the clients, and often has a much, much greater amount of physical memory and disk space. This makes it valuable as a localized cache of information shared among many clients. This relieves the burden of storage and computation on the client, and can reduce the network traffic to the Enterprise server.
- **The Enterprise server.** This is traditionally a mainframe. While mainframes have gone out of fashion in the past few years, the fact still remains that for high-volume, high-speed transaction processing, there is no better technology. Organizations have invested a great deal of time and money in these machines and their software -- it is in their interest to preserve as much of that investment as is possible, while still keeping all options open for the future.

Discussion:

A three-tier approach (see Figure 2: Three-tier architecture) gives the best solution for new development, while still supporting existing systems. If it is implemented correctly, the clients are completely de-coupled from the mainframe. Intermediate server code can be developed in such a way as to minimize dependence on the mainframe so that it can be phased out over time if that is desired.

Related Patterns:

Phase-In Tiers shows how to move from a two-tier client-server approach to a three-tier one.

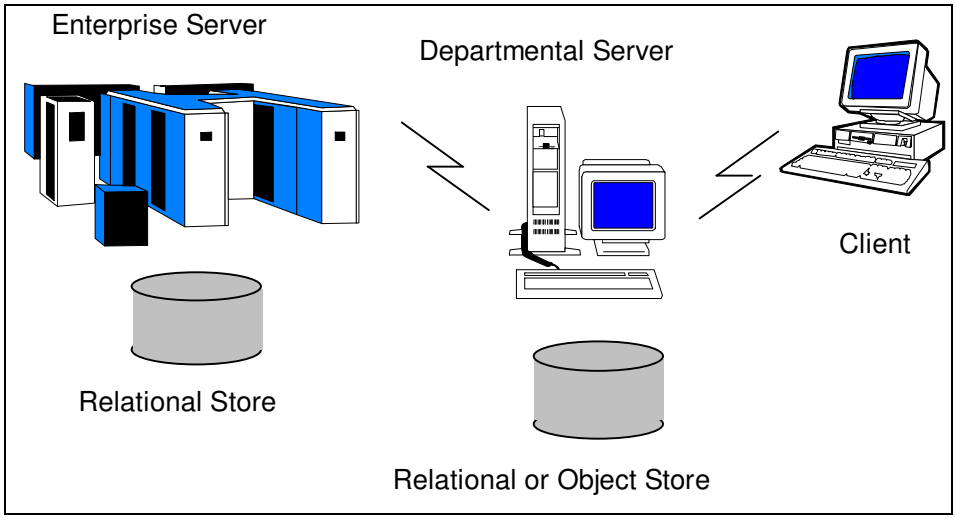

Figure 2: Three-tier architecture

PATTERN NAME: PHASE-IN TIERS

Problem:

You must come up with a solution that supports both your current and planned network architecture, and yet leverages your investment in object technology to produce results quickly.

Forces:

You need to best utilize existing and new computing and network resources. You would like to move to a client-server set of solutions as quickly as possible, but there is no way that everything can be replaced at once. The cost of a total redevelopment effort is prohibitive, and your staff could not complete the effort in a reasonable length of time. Therefore:

Solution:

A good approach is to begin all development on the client (sometimes resulting in a prototype "fat client") and then push the code from the bottom two layers of a four layer architecture onto a server as development progresses. In this way you can add tiers over time, starting with a two-tiered system (i.e., a "fat client") and moving to a three-tiered system later.

Discussion:

Modern distributed object technologies like CORBA, GemStone, IBM VisualAge Distributed Option, and PP-D Distributed Smalltalk make it possible (in fact, relatively easy) to move processing from client machines on to servers. Using these technologies, early releases can be made with fewer tiers than later releases without necessitating big changes in the code of a system.

Using distributed object technology does not preclude using either relational or OODBMS technology as appropriate for object persistence on the server(s) in any of the upper tiers. In fact, one common solution is to utilize an ODBMS for object persistence between tiers one and two, and an RDBMS for persistence between tiers two and three.

Related Patterns:

Trim and Fit Client discusses how to distribute behavior between client and server. Four Layer Architecture discusses how to distribute behavior among the layers of a software architecture.

PATTERN NAME: TRIM AND FIT CLIENT (OR DISTRIBUTE LAYERS TWO BY TWO)

Problem:

Having seen that both “fat” and “thin” clients are not appropriate, what is the proper, “healthy” division of behavior between client and server? How do you design a system such that the system is responsive on a client’s machine, and yet maintainable and architecturally sound?

Forces:

Correctly distributing behavior in a client-server system is a difficult task. When PC’s were first introduced to major companies, they were most often used as terminal emulator front-ends to existing mainframe programs. These so-called “thin” clients did not take advantage of the capabilities or processing power of the new PC clients, and could not allow for complex user interaction to occur on the client side.

In a response to this, the first generation of client-server systems often overloaded the client by placing all of the domain and display logic on the client. These so-called “fat” clients were often characterized by being big, slow, and inefficient at utilizing machine resources (network traffic, CPU horsepower, etc.). In a response to this, the “thin” client model is back in vogue. This time the client consists of a Web browser that accesses dynamically generated HTML pages from a central host. This “Web 3270” approach has the same problems as the original terminal-emulator approach in that all of the processing takes place on the server end, and complex user interaction is not allowed.

The price of memory, hard-drive space and processor speed come down almost daily. However, it is apparent that the requirements placed on these resources by modern software is expanding at an even faster rate. It is not uncommon to see large-scale client-server programs that take up 15 or more megabytes of memory by themselves. At the same time, these same advances that make client machines more powerful are also making multiprocessor servers more cost-effective. However, most current client-server systems do not take full advantage of this processing power, as the server is most often used only as a data or file server. Therefore:

Solution:

Break the system for distribution between the Application Model and Domain Model layers, or at some appropriate point inside the Domain Model layer (see Figure 3: Distributed layers). The upper two layers will reside on the client. The lower two layers will reside on the server. This will allow the code that receives the most user interaction (the upper two layers) to handle these close to the user. On the other hand, the code that handles the business logic will reside on the server. This makes it easier to design the interaction between the objects in your system if you know ahead of time where in the network these objects will reside.

Discussion:

This solution minimizes the amount of processing that must be done on a Client, and can reduce its need for memory and computational power. Note that once a system is broken up this way it does not require that the top two layer be implemented in the same

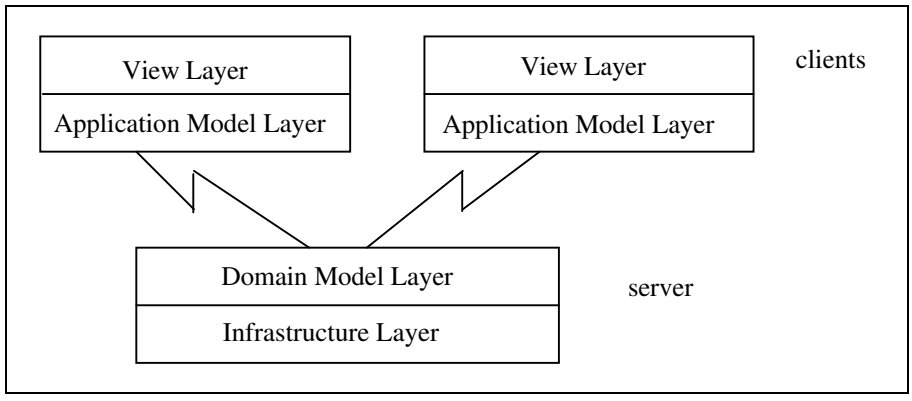

Figure 3: Distributed layers

language as the bottom two layers. A heterogeneous system will work if some type of object-to-object communication is provided. It would be perfectly acceptable to write the top two layers in Java, and the bottom two in C++ or Smalltalk, or have the entire system written from top to bottom in one language. So long as an object communication technology like CORBA or SOM can be used to provide intra-machine message passing, there should not be any restrictions put on the choice of language or platform.

Sources:

Texas Instrument's ControlWORKS project was the first large-scale project I have seen that successfully implemented this method. Since then many companies have broken their applications up in this way -- it is in fact recommended by GemStone as the best use of their product.

Related Patterns:

Four-layer architecture demonstrates why systems should have clear layer boundaries and how that helps make systems more manageable.

BIBLIOGRAPHY

[Brown 95] Kyle Brown, “Remembrance of Things Past: Layered Architectures for Smalltalk Applications”, *The Smalltalk Report*, July/August 1995.

[Brown 96] Kyle Brown and Bruce Whitenack, “Crossing Chasms: A Pattern Language for Object-Relational Integration”, in *Pattern Languages of Program Design 2*, Vlissides, Coplien and Kerth, eds., Addison-Wesley, 1996.

[Shaw] Mary Shaw, “Some Patterns for Software Architectures”, in *Pattern Languages of Program Design 2*, Vlissides, Coplien and Kerth, eds., Addison-Wesley, 1996.

[Tanenbaum] Andrew Tanenbaum*, Computer Networks*, Second Edition, Prentice-Hall, 1988

[Woolf] Bobby Woolf, “Partitioning Smalltalk Code into ENVY/DEVELOPER Components” in *Pattern Languages of Program Design 2*, Vlissides, Coplien and Kerth, eds., Addison-Wesley, 1996.