Several Patterns for eBusiness Applications❖

Dragos A. Manolescu[1] and Adrian E. Kunzle[2]

Abstract: The Software development industry has seen a shift from the development of desktop applications to the development of highly scalable, distributed, server-based eBusiness applications. Most developers have come up through the PC world and few know how to deal with the issues of distributed applications, servers, concurrency, scalability, high-availability, and fail-over. Patterns for eBusiness application development will make developers aware of the hard problems that they need to deal with, and will show them ways to solve them. Our set of patterns for building eBusiness applications provides a first step in that direction.

Building software is hard. Building eBusiness applications is even harder. Current eBusiness represents a hybrid of distributed, client-server, concurrent, and networked systems. All these systems require the solving hard problems. Since eBusiness lays on the front line of modern business, it also requires dealing with scalability, high-availability, and fail-over.

To the uninitiated, building eBusiness applications looks similar to building "traditional" applications. Unfortunately the characteristics of eBusiness make the former a much more challenging task than the latter. Astley et al [Astley+2001] observe that "software executing on distributed systems represents a unique synthesis of application code and code for managing requirements such as heterogeneity, scalability, security, and availability." Developers involved in building eBusiness applications should be aware of the obstacles they will encounter. Patterns for eBusiness applications would help developers understand some of these problems beforehand, as well as show how to solve them.

Based on our experience with building eBusiness applications we have harvested several patterns for eBusiness applications. They are not new. In fact, many of them represent techniques widely used in other types of systems (e.g., distributed systems). However, eBusiness gives them an interesting twist.

The majority of patterns in this paper fall in the class of architectural patterns. They answer questions about the high-level organization of eBusiness applications. We begin with APPLICATION SERVER, which answers the question "Where do eBusiness applications reside?" SERVER-SIDE SESSION marries the statefulness of eBusiness

❖ Copyright © 2001, Dragos A. Manolescu and Adrian E. Kunzle. Permission granted to copy for the PLoP 2001 conference. All other rights reserved.
[1] Author's address: Applied Reasoning Systems Corporation, 10955 Lowell Avenue, Suite 300, Overland Park, KS 66210, Phone (913) 319-0900, Email dmanolescu@appliedreasoning.com.
[2] Author's address: Skillgames, 233 Broadway, 19th floor, New York, NY 10279, Phone (212) 471-3514, Email adrian.kunzle@skillgames.com.

applications with the server-centric model of current Web applications. VERTICAL SLICE describes how to organize an application when some of its layers are shared among concurrent sessions. FAIL-OVER THROUGH SERVER CLUSTERING and JOB DRAINING show two ways to improve the availability of eBusiness applications. HEARTBEAT presents a technique for tracking the state of individual machines within a cluster. WEB INSPECTOR shows how to leverage Web technology for viewing and managing eBusiness applications. BUSINESS CONTEXT-AWARE OBJECT RETRIEVAL describes a way of accessing objects, optimized for the interactions typical of using Web browsers. Finally, PREFABRICATED BUSINESS OBJECTS shows how to assemble business objects from reusable components.

1 Application Server

Context

You are building an eBusiness application. The architecture involves Web browsers, Web servers, and the Internet. The Web browsers play the role of thin clients. They display application data to the user, and gather user data from HTML pages. The Web servers feed HTML content to the browsers, and receive user data from them. The Internet connects the clients to the server through HTTP.

Problem

Where do you put your application?

Forces

- Users interact with the application only through Web browsers
- Web browsers are limited to rendering HTML, running usually JavaScript and Java byte codes
- All application data going to the user passes through the Web server
- All user data going to the application passes through the Web server
- Server-side objects share resources
- Deploying software is time-consuming and expensive

Therefore,

Make the Web server a Single Point of Access [Yoder and Barcalow 1997] for user-application interaction. The Web server provides the content delivery technology. Combined with the application—see the UML collaboration diagram from Figure 1—it represents an APPLICATION SERVER.

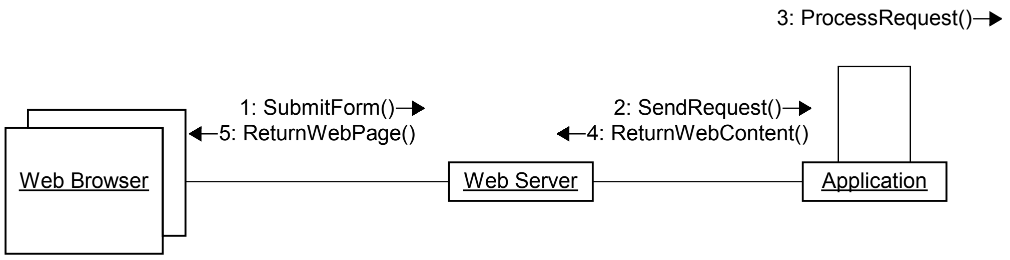


Figure 1: The APPLICATION SERVER consists of a Web server and an application.

The application doesn't handle directly the visual aspect of the presentation. Instead, it uses a presentation technology (e.g., JSP, ASP, PHP, etc.) that the Web server can turn into HTML and ship to browsers. The presentation technology limits the choice of widgets and types of user interaction. Both must be expressed as HTML constructs.

The browser uses HTTP to send user data to the Web server. In turn, this decodes the data and passes it to the application.

The Web server needs to obtain a handle on the application to communicate with it. Make the handle a well-known object. Start simple with a SINGLETON [GoF 1995]. Once you need features like load balancing, consider using a naming service (e.g., JNDI).

Typically the Web server controls the application's lifecycle. Upon start up, the Web server fires off the application. Likewise, when the Web server shuts down, it also closes the application. But the life cycles can be independent. The requirements may specify that the application and the Web server run separately. In that case, they must handle connection and disconnection.

APPLICATION SERVERS let developers release new versions of their software as soon as they complete testing. They have zero-deployment cost.

Running eBusiness applications on an APPLICATION SERVER poses security risks. Malicious users compromise security once they obtain access to the server. Additionally, the server represents a single point of failure, and can become a bottleneck. This will indiscriminately affect all customers using your application.

Known Uses

The registration system used for the PLoP conferences from 1998 through 2000 runs as a servlet within the Apache Web server [Manolescu 1998]. The Web server decodes the requests entered through Web forms and passes them to the servlet. The servlet lets attendees enter, view, and modify their registration information. It also lets an administrator see a summary of registered attendees, remove users, and export the registration information.

eBusiness applications built with the Applied Reasoning Enterprise Object Framework revolve around a coordinator acting as the application-part of an Application Server. When using BEA's WebLogic server, developers specify the coordinator class as a property in the `weblogic.properties` configuration file. At start up WebLogic fires off a SINGLETON [GoF 1995] instance of the coordinator. Likewise, the Web server closes the coordinator when it shuts down.

The Application Server represents a central component of Sun's J2EE architecture.

BEA calls an application server any application that offers server-side support for developing and deploying business logic [BEA].

Related Patterns

The APPLICATION SERVER acts as a MANAGER [Sommerland 1997] for the services used by the application. For example, Skillgames.com has services like persistence, credit card processing, security, etc.

From the customer's side the Web server represents the SINGLE POINT OF ACCESS [Yoder & Barcalow 1997] to the APPLICATION SERVER, as well as a special case of the CLIENT-DISPATCHER-SERVER [POSA2 2000] pattern.

This and other patterns in this paper rely on the proper management of threading. The POSA 2 book [POSA2 2000] contains some excellent patterns regarding this area of server systems implementation.

Resulting Context

You now have a multi-user application that uses the Web as the delivery mechanism.

2 Server-side Session

Context

The Web started as a means for sharing documents among scientists. Its designers have built the underlying technology (e.g., HTTP, the transport protocol, and HTML, the markup language) with these goals in mind. Since then, people have realized the Web's potential and have started to exploit it. With the growth of eBusiness applications, the Web is rapidly being transformed into an activity- or transaction-intensive environment.

Unfortunately, the underlying technology is not quite appropriate for these requirements. Specifically, the HTTP protocol is stateless.

Consider a customer that registers on an on-line shopping site. Typically registration begins with a form requesting the name and the contact information. Next, the user provides a credit card number and expiration date. Then they shop. The current technology requires the application to track the state of the registration process, since the delivery vehicle (the Web) doesn't.

Problem

How do you hold on to the state information related to a user's interaction with an eBusiness application?

Forces

- You need to keep state for the time that the customer is on the site
- HTTP is stateless
- Heterogeneous clients are the norm
- You don't want to keep the state information around if the user has abandoned the process
- You don't want to compromise the security of your system by giving your clients access to the state information
- You would like to minimize network traffic
- You want to hide server crashes from the users

Therefore,

Use a session object on the server. The session holds on to all state information required by the application.

But can't the session object reside on the client?

The server-centric architecture of current eBusiness applications makes a server-side session the natural choice. However, a server-side session has several liabilities. First, the server becomes a single point of failure. This makes fail-over hard. Second, the server represents a single point of access. Once in, a cracker can compromise all users' accounts. Finally, the server can become a bottleneck. It must be able to handle concurrent access by a large number of users.

As long as eBusiness applications do little processing on the client (like current technology requires), a client-side session is cumbersome. Having the session on the client brings in different problems. First, the client-server communication becomes a bottleneck. Accessing state information requires going over the network. Second, this solution exposes details about the server application to clients. This is a security problem; for example Schneier discusses a Web attack that changes the price of items in an on-line shopping cart [Schneier 2000]. Finally, clients can be very different. While it's feasible to store several megabytes of session information on a desktop computer, it's definitely not going to work on small-memory systems like Web-enabled PDAs. For example, the PDA shown in Figure 2 provides Web-browsing capabilities with only 512Kbytes of RAM.

Figure 2: Browsing the Web on a wireless PDA with 512 KBytes of SRAM.

If not used for a pre-specified time interval, the session should time out. The time out releases the resources held on by the session. Something has to watch the user-server interaction, and trigger the time out after a period of inactivity. A client-side session should also be able to signal the server when the user closes the window, or moves to a different URL.

A solution that takes into account the above limitations keeps different state information on the server as well as on the client. The bulk of the session state resides on the server to reduce network traffic. The client stores additional information that facilitates fail-over (e.g., an ID for the server-side session) and improves security (e.g., an encryption key). For example, BEA's WebLogic Server achieves fail-over by replicating the session between a primary and a secondary server. A client-side cookie stores a key to the location of the two copies of the Application Server. Should the primary server fail, the client has all the information required to find and switch to the secondary.

Known Uses

Web-based applications use a session object to keep state on the Web server. Many Web servers provide support for keeping a server-side session. Some mechanisms address identifying the session and include cookies (on the client) and URL rewriting. Others address replicating the session on other servers for improved availability.

The Applied Reasoning Enterprise Object Framework provides a server-side `SessionContext` object. The framework lets developers build stateful, server-based eBusiness applications regardless of whether the underlying technology provides a session or not.

Related Patterns

Typically the Server-side session acts like a keyed dictionary. Developers store session-related information in named slots. In effect, the session uses the Property pattern [Foote and Yoder 1998].

Martin Fowler discusses several options for storing session information in his collection of patterns for information systems architecture [Fowler]. Martin's Stateful Session stores the session information on the server.

3 Vertical Slice

Context

You have an APPLICATION SERVER that runs your eBusiness application. Customers use the application through Web browsers, over the Internet. They do this concurrently, independently, in an asynchronous manner. You can't control what they're doing and when they do it.

The last 20 years have brought computers onto everybody's desks. People install software and run separate, identical copies of the same application. This characteristic has shaped the way we design software. But eBusiness applications running on APPLICATION SERVERS change the equation. You no longer have separate applications. Instead, you have separate views. The style of the good ol' 3270 terminals is groovy again (this time around without the green CRTs).

Problem

How do you structure your application to support multiple users concurrently whilst preserving the consistency of your domain objects?

Forces

- Memory limitations make it impractical to give each user their own complete copy of the application on the APPLICATION SERVER, especially when the business info is large
- Sharing common subsystems among concurrent users is a hard problem
- Multiple users interacting with the single logical service layer run into scalability problems without careful and strict access controls
- The application and business logic must know about both the user and the business model
- The business model should be unaware of how customers interact with the system
- Objects shared among multiple threads should be thread safe and idempotent[3]

Therefore,

Build your application such that each user sees a vertical slice, spanning from the user interface (top tier) to the application services (bottom tier).

Each user requires her own objects on session-scoped tiers. For example, users will get unique customer profile objects when they log on to the application. In contrast, the objects on application-scoped tiers are shared. We call these objects service providers. The UML object diagram from Figure 3 shows two sessions sharing three services. For example, all customers using your eBusiness application share the same persistence mechanism provided by an object-relational mapper.

[3] A function $f : D \rightarrow D$ is idempotent if $f(f(x)) = f(x) \forall x \in D$. In the world of objects, an object is idempotent if repeated message sends have the same effect as a single message send.

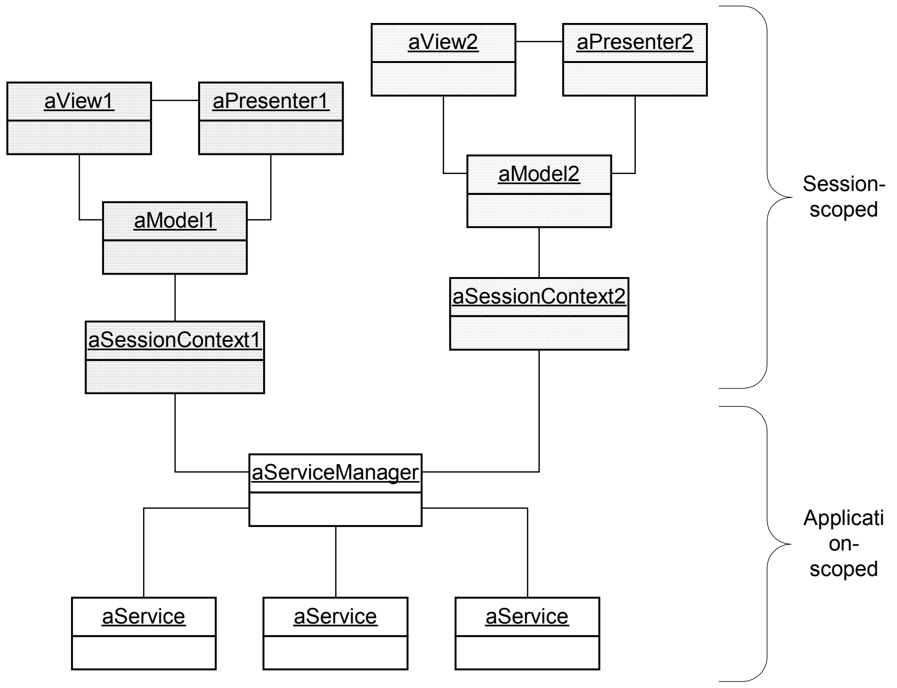


Figure 3: An application slice consists of session- and application-scoped objects. The session-scoped tiers shown here use the Model-View-Presenter pattern [MVP], but that is not a requirement.

Session-scoped objects can be designed in the more traditional style of desktop applications, assuming that there is only one user/thread running through them at a time, even though they end up running on the server. You get application/business logic reuse at the class level, i.e., you create multiple instances of business objects, one for each session. They all come from the same Class template, though.

Application-scoped objects are harder, and must deal with concurrency. All entry points must be thread-safe. These objects can't assume that successive message sends have the same sender. This means that you will have to spend longer designing/building them, but your reward is reuse at the object level (many sessions accessing the same instances), and a greater efficiency in your use of system resources.

The session-scoped part of each Vertical Slice will need a mechanism to access objects in the application-scoped part. Use a Service Manager following the Manager pattern [Sommerland 1997]. The Service Manager controls the lifecycle of the service providers and provides access to them.

Known Uses

eBusiness applications built with the Applied Reasoning Enterprise Object Framework give each user a VERTICAL SLICE of the complete application. From a logical perspective, the presentation, session, and application tiers reside in session-scoped part of the slice. Likewise, the service tier resides in application-scoped part of the slice.

Sun's J2EE architecture uses this pattern. The session-scoped part of the vertical slices consists of HTTP session and session beans. The application-scoped part consists of entity beans.

WebLogic T3Services let developers share services among Web-based applications. In effect, applications can share services within the session-scoped part of the Vertical Slice.

Xterminals talking to an Xserver share libraries and other system services. (Also applies to Citrix in the Windows environment).

Related Patterns

The Vertical Slice slices through layers spanning from user interface to application services. This corresponds to a layered architecture as described in the LAYERS [POSA 1996] pattern.

The service manager represents an instance of the MANAGER pattern [Sommerland 1997]. Objects within the session-scoped part of the vertical slice access service objects by name through the manager—an instance of the PROPERTY [Foote and Yoder 1998].

Doug Lea provides an extensive coverage of design principles and patterns for concurrent programming in Java [Lea 1999].

4 Fail-over through Server Clustering

Context

Your eBusiness application is running on an APPLICATION SERVER. When a user connects to the site, the Web server retrieves from the database the corresponding business objects and initializes a Server-side Session. For example, a CustomerProfile object holds on to the information supplied by the user at registration time, e.g., name, address, credit card number and expiration date, etc. The Web server also runs code that generates dynamic content. Several technologies (JSP, Servlets, ASP, PHP, etc.) support server-side generation of Web pages.

High traffic Web sites improve scalability and availability by distributing the load among a cluster of Web servers—Figure 4. Products like WebLogic Server provide support for clustering.

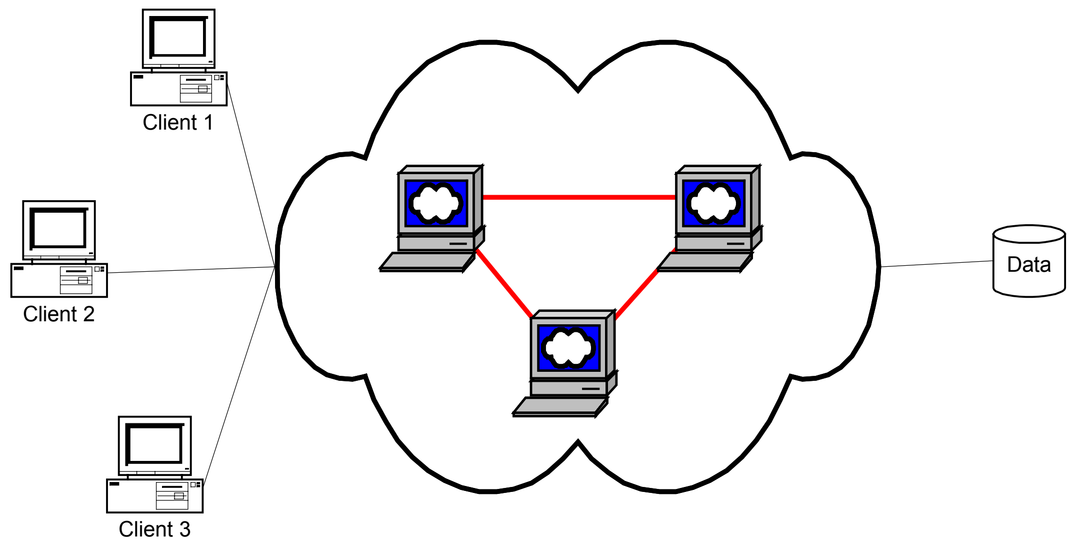


Figure 4: Clustering increases the availability of the Application Server.

Once the cluster is up and running various problems can require individual servers to be taken down. For example, hardware failures like a bad disk or an overheating CPU require halting a server for maintenance. Software upgrades or crashes also require a shutdown. You want to be able to remove the server from the cluster, perform the maintenance/upgrade, and then bring it back in the cluster. One of the other servers needs to take over the processing carried out by the one that has to be shut down.

Problem

How should you build your eBusiness application to support fail-over?

Forces

- Typically eBusiness applications deal with large numbers of concurrent customers
- Application failures are big turn-offs in the eBusiness world; you want high-availability
- Saving the state of your application each time it changes is expensive
- Many Web servers provide support for clustering
- The cluster provides support for replication among clustered servers
- Replicating all objects across the cluster doesn't scale
- You can't completely hide server failures since current Web browsers render HTML progressively

Therefore,

Build your eBusiness application to be compatible with clustering support. This support should come from a commercial product; you don't want to build it yourself!

In most scenarios, users interact solely with a primary server, which replicates the objects to one or more secondary servers. Should the primary server become unavailable, the secondary servers have all the state information required to replace the primary server. One of them takes over and the application continues to execute, albeit at a different location within the cluster. But the failure of the primary and the shift to a secondary remains invisible from the outside.

For example, WebLogic supports in-memory replication of HTTP session, EJBs, and RMI objects. The server copies the contents of the HTTP session and EJB/RMI objects from a primary host to a single secondary server.

It is very important to understand how your commercial product supports clustering. Replication of objects can get expensive very quickly, both in terms of network traffic and memory footprint. For example WebLogic uses Java Serialization, which is not optimal when you have large object graphs, and few state changes. Understand what you must replicate, and what you can get away with reconstituting from other sources, such as a persistence layer.

In summary, this pattern provides a means for masking server failures through redundancy. This is a key principle of distributed design [Pradhan 1995].

Known Uses

BEA's WebLogic server supports Web clustering and component/object clustering. Web clustering replicates objects within the session-scoped part of the VERTICAL SLICE. Likewise, component/object clustering replicates EJB and RMI objects residing in the application-scoped part of the VERTICAL SLICE.

Although mainly stateless, server farms such as those used for financial instrument pricing are generally clustered. Because of their lack of state, there is nothing to replicate, which simplifies the problem considerably. However machines can be taken in and out of the cluster at will, with no effect on the user.

Variants

Geographic fail-over allows system administrators to select primary and secondary servers at different locations. Should the site hosting the primary servers become unavailable (e.g., due to a California power outage), the secondary servers are not affected and can take over. ATG Dynamo [Dynamo] provides this feature.

Related Patterns

Many distributed systems use a backup to improve availability. For example, RECOVERABLE DISTRIBUTOR [Islam and Devarakonda 1996] provides several protocols for fault-tolerance. One of these protocols is primary-backup; recovering from a failure amounts to switching to the backup. RECOVERY DISTRIBUTOR can also reconstruct the state of the primary on a backup system, which takes longer but doesn't require the resources needed to maintain an up-to-date copy.

APPLICATION SERVER uses this pattern to increase the number of SERVER-SIDE SESSIONS it can support concurrently, and hide server failures through fail-over.

JOB DRAINING provides a work around that lets system administrators to turn individual machines down when server clustering is not possible (i.e., flipping the switch is not feasible).

HEARTBEAT describes a mechanism to check the availability of individual servers in the cluster.

5 Job Draining

Context

You have deployed your eBusiness application on a cluster of Web servers. Each server runs a different application instance. You need to take down a server for scheduled maintenance. You must do it in a way that is transparent for your users.

If you could use server state replication, such as EJB server clustering, you would be able to unplug the server and your users wouldn't notice. One of the replicas would take over whenever the primary goes down. However, if the server provides replication in a manner that is incompatible with your application, or your server runs long-lived transactions, you're on your own, and can't simply unplug the server.

Problem

How can you gracefully handle scheduled maintenance on an APPLICATION SERVER when it doesn't support clustering?

Forces

- Users who have SERVER-SIDE SESSIONS on the machine scheduled to go down should not be affected
- You can't leverage the clustering capabilities provided by your chosen infrastructure
- Replication of application state is expensive
- Transactions can run for a long time

Therefore,

Prevent the server scheduled to go down from accepting new requests. This involves intercepting any incoming requests and checking whether they correspond to new jobs, or existing jobs. Leave the requests corresponding to existing jobs pass through. Reroute requests corresponding to new jobs to other servers.

Rerouting requests will let the server complete the jobs that are currently running on it. Once they have all finished, you can safely shut down the server.

Who reroutes the requests? Clustered servers typically use a load balancer that distributes incoming requests among cluster members. Since all requests pass through the balancer, the rerouting can take place there. This involves adding a dispatcher on the load balancer, which in turn must be able to execute code—see Figure 5 and Figure 6. The balancer + dispatcher ensemble has to be designed such that more than one of them can exist in the system, so that they don't become a single point of failure. The dispatcher must cope with the traffic passing through the entire site. Additionally, it must be aware of the jobs running on each server. This may not scale when the number of jobs is large.

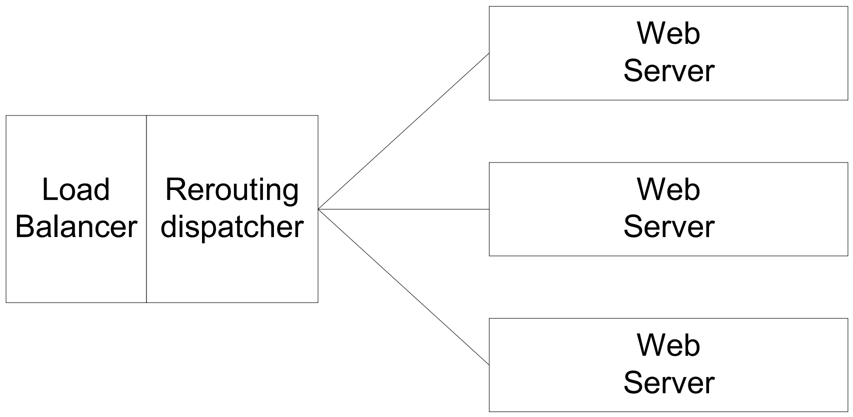


Figure 5: Job draining with a Rerouting Dispatcher

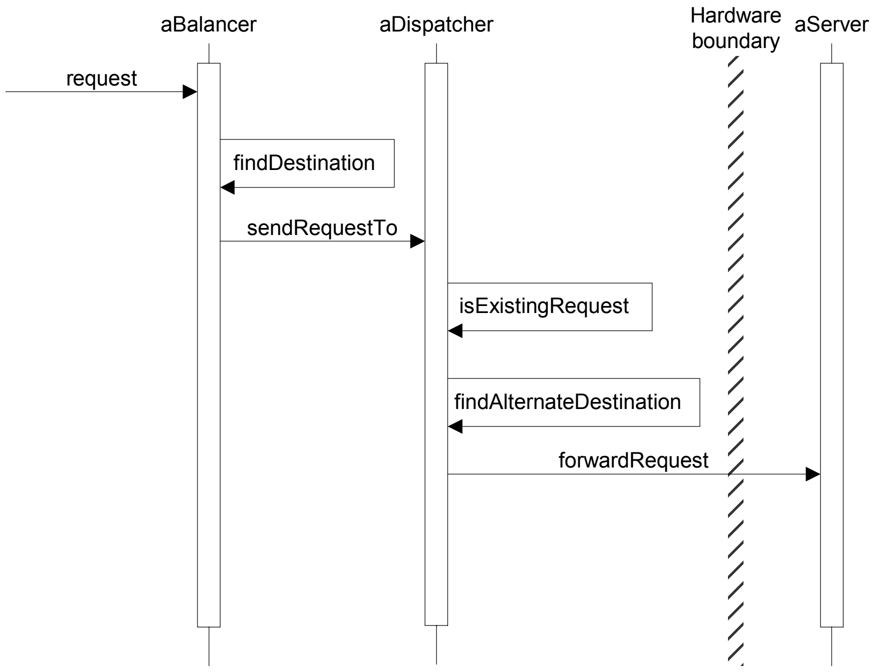


Figure 6: Rerouting Dispatcher, UML sequence diagram. The sequence shows the flow of messages when isExistingRequest() returns false, and findAlternateDestination() returns an alternate destination.

An alternative involves using rerouting proxies on each server—see Figure 7 and Figure 8. When a server is scheduled to go down, its proxy starts rerouting requests to other servers. This solution has the advantage that each proxy tracks only the local jobs. You can also use a "dumb" load balancer that merely dispatches requests in a round-robin manner. However, the rerouting proxies need to communicate whenever the status of a server changes.

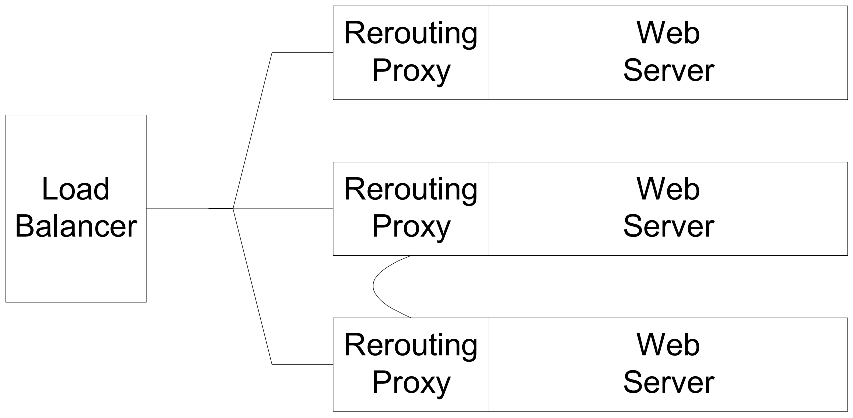


Figure 7: Job draining with Rerouting Proxies

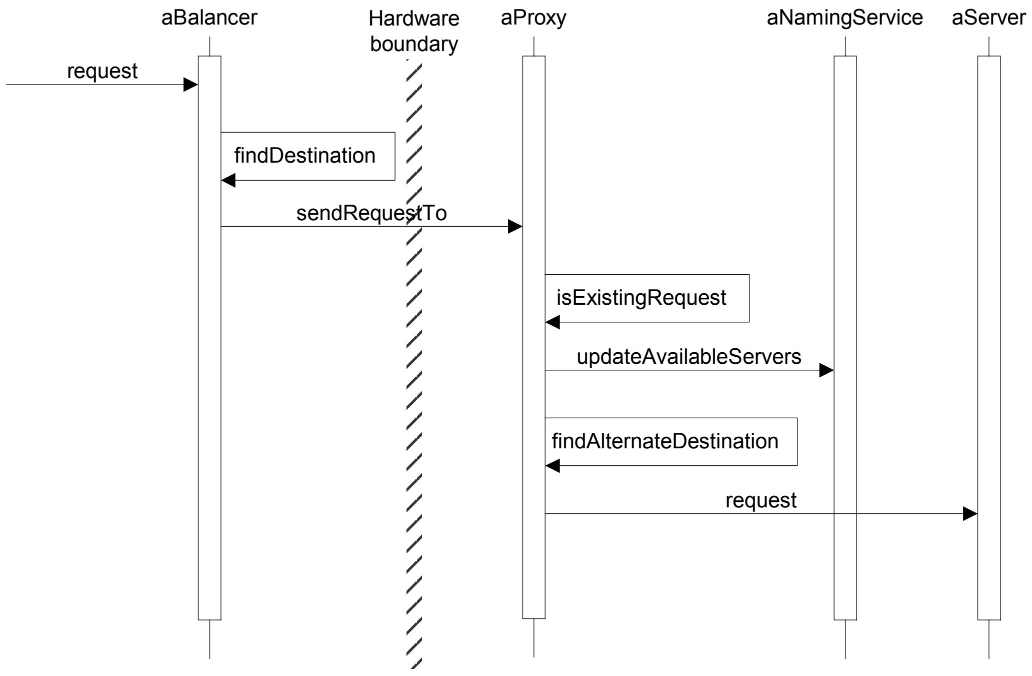


Figure 8: Rerouting Proxies, UML sequence diagram. The sequence shows the flow of messages when isExistingRequest() returns false, and findAlternateDestination() returns an alternate destination.

Known Uses

Skillgames.com uses Rerouting Proxies that route the incoming requests to the Web servers that are available. The proxies use the Java Messaging Service to communicate with each other.

The NCSA Load Share Facility system lets administrators drain jobs from their supercomputers for scheduled maintenance. LSF uses a job scheduler that resembles the rerouting dispatcher.

Related Patterns

JOB DRAINING helps you achieve a limited level of fail-over when you have an APPLICATION SERVER and can't rely on the underlying infrastructure to replicate the SERVER-SIDE SESSION.

The Rerouting Proxy represents an instance of the PROXY pattern [GoF 1995, Rohnert 1995].

Resulting Context

You will need a way to detect when an APPLICATION SERVER becomes unavailable.

6 Heartbeat

Context

You have deployed your APPLICATION SERVER on a cluster. Each member of the cluster runs instances of your application.

Problem

How do you know when one Web server becomes unavailable?

Forces

- Knowing the availability of your servers lets you take proactive measures to alert the local system
- You would like to centralize the disconnect and reconnect mechanisms
- Waiting for timeouts degrades performance and consumes system resources
- Constantly checking the availability of the Web servers consumes system and network resources
- You have to decide who has the responsibility of taking action when something goes wrong
- Failures within the cluster should be transparent to the user

Solution

Make each live server broadcast periodic "I'm alive" messages. A cluster monitor listens for these broadcasts and resets time out counters corresponding to each server. The monitor considers any servers whose counter doesn't reset within a given time interval as unavailable. Consequently, the monitor notifies the cluster, which takes proactive action for compensating for the unavailable server.

The frequency of the periodic broadcasts depends on the application. On the one hand, a low frequency translates into observable delays at the front end, and has the risk of annoying your customers. On the other hand, frequent broadcast will flood the network with control messages, as well as consume system resources.

Known Uses

The Linux kernel supports several forms of watchdogs. The system reboots if the watchdog device hasn't been written for a certain amount of time.

BEA's WebLogic server uses IP multicast to broadcast heartbeat messages that advertise server availability within the cluster.

Ethernet uses a heartbeat/pulse signal to alert whenever devices connected to the network become unavailable—e.g., the power is turned off.

Tibco, a commercial messaging middleware product sends heartbeats between all its distribution brokers, indicating that they are alive.

According to Bruce Schneier [Schneier 2000] security systems use this pattern to prevent burglars from disconnecting them from the phone system.

Classic Blend and DCOM use HEARTBEAT in the context of distributed components.

Variants

RECOVERABLE DISTRIBUTOR [Islam and Devarakonda 1996] checks the availability of Local Failure Handlers (LFH) through polling rather than broadcast. Therefore, instead of broadcasting the "I'm alive message," the Global Failure Handler (GFH) iterates through all LFHs and sends them a message. If it doesn't receive an acknowledgement after several attempts the GFH decides that the LFH has failed.

Related Patterns

JOB DRAINING requires a means of checking the state of the servers in the clusters. HEARBEAT provides a means of doing so in a centralized manner.

7 Web Inspector

Context

In mature software departments, developers have access to rich tools that enable them to write correct, efficient code, and to debug problems effectively. These tools include step debuggers (most IDEs have these), profilers (jProbe) and object inspectors (VisualAge's Scrapbook and other features). For 2-tier applications that get deployed as desktop executables, these tools are often sufficient. In a 3 or greater tier application, such as an Application Server, many of these tools are not yet available. It is also hard to completely replicate the n-tiered deployment environment on each desktop. The result is code that often runs fine on the developer's desktop, but breaks in the deployment environment. Distributed debuggers are available, but are often IDE or deployment platform specific, and hard to configure or attach while the system is running. Remote profiling is available, and is good way of getting timing and object allocation information. It would be a great help to developers to also be able to inspect running objects in the deployment environment, to check on values of variables, and possibly even execute messages against them.

Problem

You have enabled your application in a remote APPLICATION SERVER. How can you quickly and easily examine what's going on, reconfigure on the fly, etc.?

Forces

- Building distributed applications is hard; debugging distributed programs is really hard
- Typically deployed applications don't contain debugging information
- Your server is not physically accessible; rather, it is relocated to a hosting service with a fat pipe to the Internet
- Access restrictions can prohibit the use of a remote debugger
- The solution should be vendor-independent
- You'd like zero-deployment effort
- Most monitoring tools are application unaware
- Providing access under the hood poses a massive security hole
- Requires scaffolding on the server that otherwise you wouldn't need

Solution

Build support within the system for examining the state of your application through a web browser. This can take many forms:

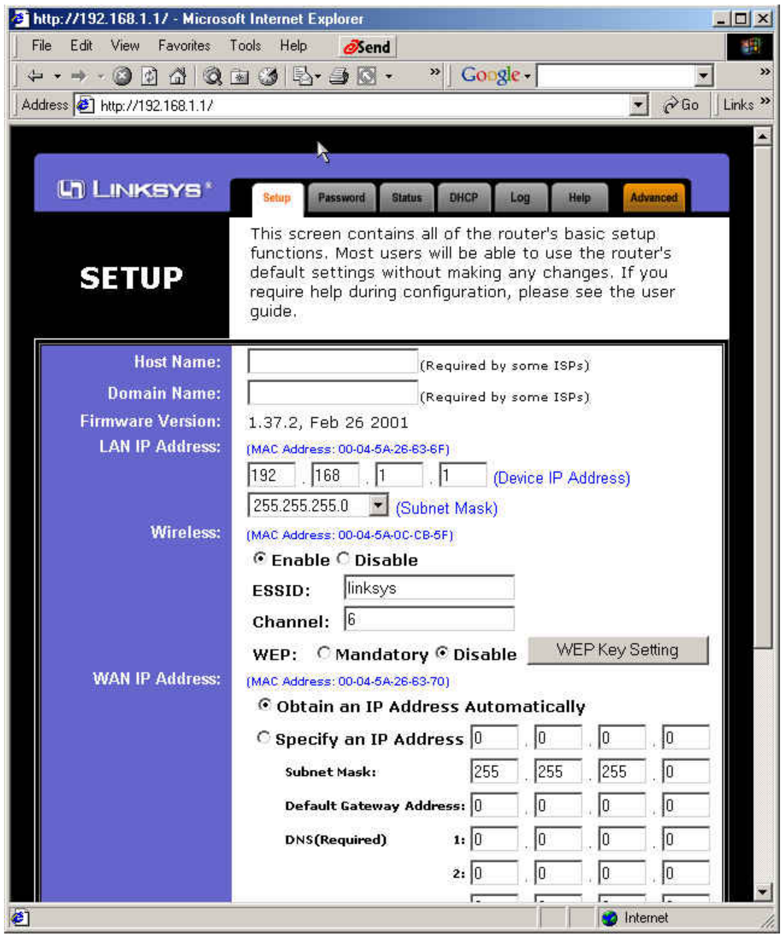
1. Interrogating known objects, with pre-defined messages.
2. Allowing for the inspection of any object, so long as you can navigate to it. (Requires reflection)
3. Allowing for the sending of messages to any object, potentially allowing for state changes on the fly. (Requires reflection and "compilation")

To build this support, you will have to add this functionality on the APPLICATION SERVER. This must be done in a controlled fashion, and involves strong security, authentication and auditing.

Known Uses

The JProbe Java profiler lets developers look at the information it has collected through a Web browser, from a remote location.

Several Cable/DSL routers run an internal Web server and let their users configure them through a Web browser. Figure 9 shows an example.


http://192.168.1.1/ - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Send
Google
Address http://192.168.1.1/
Go
Links
LINKSYS
Setup
Password
Status
DHCP
Log
Help
Advanced
SETUP
WEP Key Setting
Internet

This screen contains all of the router's basic setup functions. Most users will be able to use the router's default settings without making any changes. If you require help during configuration, please see the user guide.

Host Name: (Required by some ISPs)

Domain Name: (Required by some ISPs)

Firmware Version: 1.37.2, Feb 26 2001

LAN IP Address: (MAC Address: 00-04-5A-26-63-6F)
192 . 168 . 1 . 1 (Device IP Address)
255.255.255.0 (Subnet Mask)

Wireless: (MAC Address: 00-04-5A-0C-CB-5F)
[✓ Enable] Disable
ESSID: linksys
Channel: 6
WEP: Mandatory [✓ Disable]

WAN IP Address: (MAC Address: 00-04-5A-26-63-70)
[✓ Obtain an IP Address Automatically]
Specify an IP Address 0 . 0 . 0 . 0
Subnet Mask: 255 . 255 . 255 . 0
Default Gateway Address: 0 . 0 . 0 . 0
DNS(Required) 1: 0 . 0 . 0 . 0
2: 0 . 0 . 0 . 0

Figure 9: Web configuration panel for a Linksys cable/DSL router.

Web servers like Apache and WebLogic provide an "administrative console" through which systems administrators configure them.

Related Patterns

WEB INSPECTOR comes in handy when you build an APPLICATION SERVER and want to be able to see what's going on.

REFLECTION [POSA 1996] describes the mechanisms required for reflection in pattern form.

8 Business Context-Aware Object Retrieval

Context

During its lifetime, a SERVER-SIDE SESSION interacts with many objects. Often, many of these will be part of complex object graphs, and will be brought in from different sources (database, flat file, remote system, etc.)

For example, when logging in to an on-line trading service, you want to first see a summary of your portfolio, stock prices of shares you are watching, and significant market news. This requires the system to access a lot of different pieces of information, but not in much detail. Next you want to see the details of the holdings in your portfolio, and their current valuations. This requires access to the details of each holding.

Problem

How do you bring object graphs into the Application Server in a manner that is both efficient and matches the information paths that the user navigates?

Forces

- Users don't surf sites in a depth-first manner
- APPLICATION SERVERS have to support a large number of concurrent users
- Transitive closures of rich business object models are often large
- Object retrieval is expensive
- Web-based eBusiness applications have different usage patterns compared with traditional desktop applications

Therefore,

Use smart proxies to stub out objects that may not be needed. When the user tries to access a stubbed out object, the APPLICATION SERVER asks the proxy to fault in the object that it represents.

For example, a Portfolio object that stores its most recent value doesn't need to access each holding to present that value to the user. Therefore, when building the user's summary page, you only need to access the Portfolio object, and can stub out all its components.

Likewise, a CustomerProfile object might hold onto the customer's address. The application doesn't use the address except when the customer wants to edit her profile. When the customer logs on to the APPLICATION SERVER, retrieve only the CustomerProfile instance, and stub out the Address.

You need a mechanism to specify what instance variables on a parent should get stubbed out rather than retrieved along with the parent. The GemStone/S object persistent store and the OpenTalk distributed application framework provide this sort of mechanism, via a class side/instance side descriptor.

Known uses

When a customer first logs on to Skillgames.com, only the basic part of a `CustomerProfile` object, along with some preferences is loaded from the database. Other child objects such as addresses, phone numbers etc. are stubbed out. This is done to reduce the time it takes to "log on". 70% of the time, the user will end the session and never go to My Account (the area of the site that requires the child objects). For the 30% that do, the penalty is only a slight increase in time to get to the My Account page, created by the need to fault in the child objects.

Many portfolios, especially Mutual Funds, are officially priced daily. This means that yesterday's total value is usually stored on the Portfolio itself. Consequently a summary value of the portfolio can be displayed without ever retrieving the holdings that make up the portfolio. Only when a real-time price is needed, and each holding has to be re-valued and summed, does the system have to fault in the holdings.

Related Patterns

This pattern is in fact a variant of PROXY [GoF 1995, Rohnert 1995]. Faulting in business objects in a manner tuned for the typical way people surf the Web (which is one of the forces) makes it different.

9 Prefabricated Business Objects

Context

If you look at eBusiness applications, they all require a similar set of core objects. What differs between each application is which parts you assemble together. The difference is in the whole, not in the parts.

For example, you are building a system that contains a representation of your customer. You start by creating one object called `CustomerProfile` which contains all the state and behavior of your customer on itself. However, as the system evolves, you discover the need for a representation of an internal user. At first glance, they appear to be similar, containing names, addresses, SSN etc., so you start to refactor.

This results in two container objects, a `CustomerProfile` and an `InternalUser`, and sub-components such as `Address`, `PhoneNumber`, etc. You get commonality at the component level, but significant differences in the way that they are put together at the container level. For example, the `CustomerProfile` holds a home address, but the `InternalUser` doesn't.

As this evolution continues and the objects have additional requirements imposed on them, the container object ends up having less and less direct state, and more and more contained objects. It essentially becomes a façade, delegating a lot of its required behavior to its children. In extreme cases, the `CustomerProfile` can end up with only a single piece of state, a unique identifier such as user ID.

Problem

How do you compose a set of small objects that don't make sense on their own in the business domain, into an object that is meaningful to the business domain?

Forces

- How do you balance encapsulation of composite objects with the sum of their interfaces getting promoted to the Container's interface. In other words, at what point does the interface on the container get too broad, and you should start exposing contained objects directly.
- Making the container object too customized for your specific needs reduces reuse and extensibility
- If one of the components of your container has features that you don't need, what do you do with that functionality? You could mask it by leaving it out of the container's interface, but then you violate the Liskov Substitution Principle.
- Specifically for Customer objects, extensibility is very important since they are the focus for many eBusiness applications, and the business expectation is that base functionality should be available through third-party component libraries by now. However, the business will almost always demand some level of customization.

Solution

Start with a minimal core object, and create a rich facade that provides all the additional behavior and data. Also use the strategy pattern to provide flexibility in operation where necessary.

For example, we want to create a `CustomerProfile` object that is to be part of a eCommerce framework, and therefore comes with a rich set of capabilities "out of the box." We need to allow the customer the choice in what subset of capabilities they want to include in their deployed application, as well as the ability for them to add their own specific functionality. We also need to make sure that the framework is internationalizable.

The core object here is called `CustomerCore`. It contains a unique identifier, a user ID, and a name. This is then wrapped by a FAÇADE, `BasicCustomerProfile`, that adds a homeAddress, workAddress, homePhone, workPhone. We now have a usable `CustomerProfile` object.

The people buying the framework then have two ways to extend what we have already supplied. The first is to create their own façade, replacing `BasicCustomerProfile`, that holds on to the discrete components directly (such as Address and Phone objects). The second is to create their own FAÇADE that holds onto a `BasicCustomerProfile`. Neither approach is limited by inheritance, and therefore provides maximum flexibility.

Known Uses

Trade objects in financial systems. Trade objects are usually composed of many pieces of information such as Counterparty, settlement details, pricing data, basic trade info etc. The container (Trade) is almost always very thin (usually with just a trade ID), with all the behavior living in the reusable components.

Related Patterns

The object made of existing objects acts as a FAÇADE [GoF 1995]; it could also be considered a special case of the WHOLE-PART [POSA 1996] pattern.

You could use FLYWEIGHT [GoF 1995] to save space by sharing components among facades. For example, if the Employee object is a Prefabricated Business Object, several employees can share the same business address.

10 Putting it Together

Your company has decided that the new application that you are building should be "Web enabled." You have also determined that combination of simple CGI scripts, database access, and HTML won't meet your needs. So, what do you do?

First you make your business logic presentation-independent. Then you can select the best presentation technology for your needs (JSPs, Servlets, DHTML, PHP, etc.) and hook it up to your business logic. These two together form your application. Finally you add a Web server to the mix, and voilà, you now have an APPLICATION SERVER.

Next you need to decide what session information should be maintained while the customer interacts with the site. You encapsulate this body of information into a SERVER-SIDE SESSION.

You have already partitioned your application into a business logic and presentation tier. You now need to break up the business logic into parts that can be shared among sessions, and parts that are unique to individual sessions. In effect, you are building your VERTICAL SLICE. BUSINESS CONTEXT-AWARE OBJECT RETRIEVAL brings in objects in a way that is compatible with customers navigating Web pages. Following PREFABRICATED BUSINESS OBJECTS you will compose your domain objects out of existing, already tested business objects.

If you can tap into the support provided by your chosen infrastructure, you will use the FAIL-OVER THROUGH SERVER CLUSTERING. Otherwise you can use JOB DRAINING with either a rerouting dispatcher, or rerouting proxies. HEARTBEAT will tell you which server has failed.

Once deployed, WEB INSPECTOR lets you look under the hood of your application with a Web browser.

11 Acknowledgments

We would like to thank the following colleagues and friends from Applied Reasoning who have reviewed a draft of this paper: Bill Burdick, Mike Diamond, Rich MacDonald, Fritz Passow, and Max Ross. Our shepherd Eduardo Fernandez has also provided insightful comments during the shepherding process.

12 References

[Astley+2001] Mark Astley, Daniel C. Sturman, and Gul A. Agha, *Customizable Middleware for Modular Distributed Software*, CACM May 2001, Volume 44, Number 5, pp. 99—107.

[BEA] *What is a Java Application Server?*, BEA Inc. white paper, available on the Web at http://www.bea.com/products/weblogic/server/paper_Java.shtml.

[Foote & Yoder 1998] Brian Foote and Joseph W. Yoder, *Metadata and Active Object-Model,* Fifth Conference on Patterns Languages of Programs (PLoP '98) Monticello, Illinois, August 1998. Available as Technical Report #WUCS-98-25, Dept. of Computer Science, Washington University, September 1998.

[Fowler] Martin Fowler, Information Systems Architecture (work in progress). Available on the Web at http://www.martinfowler.com/isa/.

[GoF 1995] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns*, AWL 1995.

[Islam and Devarakonda 1996] Nayeem Islam and Murthy Devarakonda, *An Essential Design Pattern for Fault-Tolerant Distributed State Sharing*, CACM October 1996, Volume 39, Number 10, pp. 65—74.

[Lea 1999] Doug Lea, *Concurrent Programming in Java: Design Principles and Patterns*, 2nd edition, AWL 1999.

[Manolescu 1998] Dragos A. Manolescu, *An Object-Oriented Framework for On-Line Registrations.* Available on the Web at http://micro-workflow.com/CAT/.

[MVP] The Model-View-Presenter pattern. Available on the Web from http://www.object-arts.com/EducationCentre/Patterns/MVP.htm.

[POSA 1996] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerland, and Michael Stal, *Pattern-Oriented Software Architecture*, Volume 1, Wiley 1996.

[POSA2 2000] Douglas Schmidt, Michael Stal, Hans Rohnert, Frank Buschmann, *Pattern-Oriented Software Architecture*, Volume 2: Patterns for Concurrent and Networked Objects, Wiley 2000.

[PLoPD1] James O. Coplien and Douglas C. Schmidt, editors, *Pattern Languages of Program Design,* Addison-Wesley 1995.

[PLoPD3] Robert C. Martin, Dirk Riehle, and Frank Buschmann, editors, *Pattern Languages of Program Design, Volume 3*, Addison-Wesley 1997.

[PLoPD4] Neil Harrison, Brian Foote, and Hans Rohnert, editors, *Pattern Languages of Program Design, Volume 4*, Addison-Wesley 1999.

[Pradhan 1995] Dhiraj K. Pradhan, *Fault-Tolerant Computer System Design*, Prentice Hall, 1995.

[Rohnert 1995] Hans Rohnert, *The Proxy Design Pattern Revisited*. In [PLoPD1].

[Schneier 2000] Bruce Schneier, *Secrets & Lies—Digital Security in a Networked World*, John Wiley and Sons, 2000.

[Sommerland 1997] Peter Sommerland, *Manager*. In PLoPD3 [PLoPD3].

[Yoder & Barcalow 1997] Joseph W. Yoder and Jeffrey Barcalow, *Architectural Patterns for Enabling Application Security*. In PLoPD4 [PLoPD4].