Developing Flexible and High-performance Web Servers with Frameworks and Patterns

Douglas C. Schmidt
schmidt@cs.wustl.edu

James C. Hu
jxh@cs.wustl.edu

Department of Computer Science
Washington University
St. Louis, MO 63130
(314) 935-7538 (TEL)
(314) 935-7302 (FAX)

A subset of this paper will appear in ACM *Computing Surveys*, volume 30, 1998.

Abstract

The goal of this paper is to illustrate how frameworks and patterns address complexities that arise in the design and implementation of high-performance distributed software systems. These complexities are both inherent *(e.g., latency reduction and throughput preservation), and* accidental *(e.g., the continuous reinvention of key concepts and components). This paper explains how complexities occurring in the development of high-performance Web servers can be alleviated with the use of design patterns and object-oriented application frameworks. These techniques were applied to the development our high-performance adaptive Web server framework, JAWS. JAWS exemplifies how a framework can remain flexible without sacrificing performance.*

1 Applying Patterns and Frameworks to Web Servers

Developers of Web servers strive to build fast, scalable, and configurable systems. This paper describes some common pitfalls encountered by these developers and how to avoid these pitfalls. Common pitfalls include (1) coping with tedious and error-prone low-level programming details, (2) lack of portability, and (3) the complexity of navigating the wide range of server design alternatives. By carefully utilizing patterns and frameworks, these hazards can be avoided, by allowing developers to leverage reuse of design and code.

1.1 Common Pitfalls of Developing Web Server Software

Web servers perform the following tasks: connection establishment, service initialization, event demultiplexing, event handler dispatching, interprocess communication, memory management and file caching, static and dynamic component configuration, concurrency, synchronization, and persistence. In most Web servers, these tasks are implemented in an *ad hoc* manner using low-level native OS application programming interfaces (APIs), such as Win32 or UNIX/POSIX, which are written in C.

Unfortunately, native OS APIs are not an effective way to develop Web servers or other types of communication middleware and applications [1]. The following are common pitfalls associated with the use of native OS APIs:

Excessive low-level details: Building Web servers with native OS APIs requires developers to have intimate knowledge of low-level OS details. Developers must carefully track which error codes are returned by each system call and handle these OS-specific problems in their servers. Such details divert attention from the broader, more strategic issues, such as protocol semantics and server structure. For example, UNIX developers who use the `wait` system call must distinguish between return errors due to no child processes being present and errors from signal interrupts. In the latter case, the `wait` must be reissued.

Reinvention of incompatible programming abstractions: A common remedy for the excessive level of detail with OS APIs is to define higher-level programming abstractions. For instance, many Web servers create a file cache to avoid accessing the filesystem for each client request. However, these types of abstractions are often rediscovered and reinvented independently by each developer or project. This *ad hoc* development process hampers productivity and creates incompatible components that are not readily reusable within and across projects.

High potential for errors: Programming to low-level OS APIs is tedious and error-prone due to their lack of type-safety. For example, most Web servers are programmed with the Socket API [2]. However, endpoints of communication in the Socket API are represented as untyped handles. This increases the potential for subtle programming mistakes and run-time errors.

Lack of portability: Low-level OS APIs are notoriously non-portable, even across releases of the same OS. For instance, implementations of the Socket API on Win32 platforms (WinSock) are subtly different than on UNIX platforms. Moreover, even WinSock implementations on different versions of Windows NT possess incompatible timing-related bugs that cause sporadic failures when performing non-blocking connections.

Steep learning curve: Due to the excessive level of detail, the effort required to master OS-level APIs can be very high. For instance, it is hard to learn how to program POSIX asynchronous I/O [3] correctly. It is even harder to learn how to write a *portable* application using asynchronous I/O mechanisms since they differ widely across OS platforms.

Inability to handle increasing complexity: OS APIs define basic interfaces to mechanisms like process and thread management, interprocess communication, file systems, and memory management. However, these basic interfaces do not scale up gracefully as applications grow in size and complexity. For instance, a typical UNIX process allows a backlog of only $\sim$7 pending connections [4]. This number is inadequate for heavily accessed Web servers that process hundreds of simultaneous clients.

1.2 Overcoming Web Server Pitfalls with Patterns and Frameworks

Software reuse is a a widely touted method of reducing development effort. Reuse leverages the application domain knowledge and prior effort of experienced developers. When applied effectively, reuse can avoid recreating and revalidating common solutions to recurring application requirements and software design challenges.

Java's `java.lang.net` and RogueWave `Net.h++` are two common examples of applying reusable OO class libraries to communication software. Although class libraries effectively support component reuse-in-the-small, their scope is overly constrained. In particular, class libraries do not capture the canonical control flow and collaboration among families of related software components. Thus, developers who apply class library-based reuse often reinvent and reimplement the overall software architecture and much of the control logic for each new application.

A more powerful way to overcome the pitfalls described above is to identify the *patterns* that underlie proven Web servers and to reify these patterns in *object-oriented application frameworks* [5]. Patterns and frameworks help alleviate the continual rediscovery and reinvention of key Web server concepts and components by capturing solutions to common software development problems [6].

The benefits of patterns for Web servers: Patterns document the structure and participants in common Web server micro-architectures. For instance, the Reactor [7] and Active Object [8] patterns are widely used as Web server dispatching and concurrency strategies, respectively.

Traditionally, these types of patterns have either been locked in the heads of the expert developers or buried deep within the source code. Allowing this valuable information to reside only in these locations is risky and expensive. For instance, the insights of experienced Web server designers will be lost over time if they are not documented. Therefore, capturing and documenting Web server patterns explicitly is essential to preserve design information for developers who enhance and maintain existing software.

The benefits of frameworks for Web servers: Knowledge of patterns helps to reduce development effort and maintenance costs. However, reuse of patterns alone is not sufficient to create flexible and efficient Web server software. While patterns enable reuse of abstract design and architecture knowledge, abstractions documented as patterns do not directly yield reusable code [9]. Frameworks help developers avoid costly reinvention of standard Web server components by implementing common design patterns and factoring out common implementation roles.

1.3 Relationship Between Frameworks, Patterns, and Other Reuse Techniques

Frameworks provide reusable software components for applications by integrating sets of abstract classes and defining standard ways that instances of these classes collaborate [10]. In general, the components are not self-contained, since they usually depend upon functionality provided by other components within the framework. However, the collection of these components forms a partial implementation, *i.e.*, an application skeleton.

The scope of reuse in a Web server framework can be significantly larger than using traditional function libraries or OO class libraries of components. In particular, the JAWS framework described in Section 2 is tailored for a wide range of Web server tasks. These tasks include service initialization, error handling, flow control, event processing, file caching, concurrency control, and prototype pipelining. It is important to recognize that these tasks are also reusable for many other types of communication software.

In general, frameworks and patterns enhance reuse techniques based on class libraries of components in the following ways.

Frameworks define "semi-complete" applications that embody domain-specific object structures and functionality: Class libraries provide a relatively small granularity of reuse. For instance, the classes in Figure 1 are typically low-level, relatively independent, and general-purpose components like Strings, complex numbers, arrays, and bit sets.

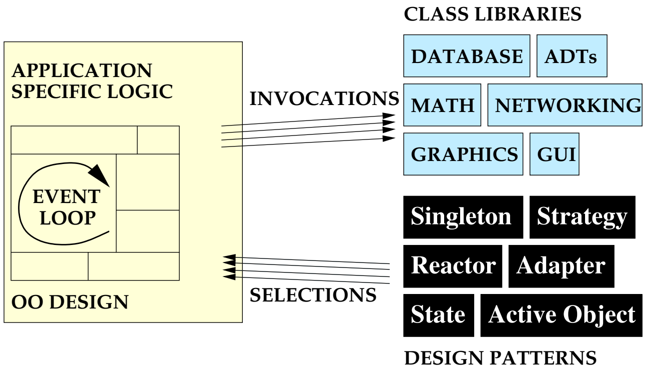

Figure 1: Class Library Component Architecture

In contrast, components in a framework collaborate to provide a customizable architectural skeleton for a family of related applications. Complete applications can be composed by inheriting from and/or instantiating framework components. As shown in Figure 2, frameworks reduce the amount of application-specific code since much of the domain-specific processing is factored into generic framework components.

Frameworks are active and exhibit "inversion of control" at run-time: Class library components generally behave *passively*. In particular, class library components often perform their processing by borrowing the thread(s) of control from application objects that are "self-directed."

The typical structure and dynamics of applications built with class libraries and components is illustrated in Figure 1. This figure also illustrates how design patterns can help guide the design, implementation, and use of class library components. Note, that the existence of class libraries, while providing tools to solve particular tasks (*e.g.*, establishing a network connection) do not offer explicit guidance to system design.

In contrast to class libraries, components in a framework are more *active*. In particular, they manage the canonical flow of control within an application via event dispatching patterns

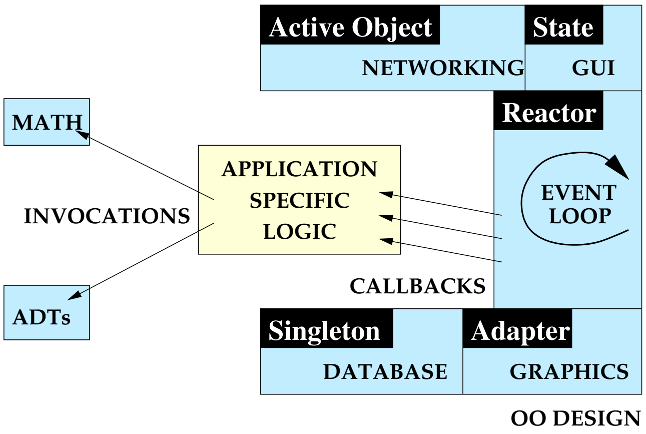

Figure 2: Application Framework Component Architecture

like Reactor [7] and Observer [6]. The callback-driven runtime architecture of a framework is shown in Figure 2.

Figure 2 illustrates a key characteristic of a framework: its "inversion of control" at run-time. Inversion of control allows the framework, rather than each application, to determine which set of application-specific methods to invoke in response to external events (such as HTTP connections and data arriving on sockets). As a result, the framework reifies an integrated set of patterns, which are pre-applied into collaborating components. This design reduces the burden for software developers.

In practice, frameworks, class libraries, and components are complementary technologies [5]. Frameworks often utilize class libraries and components internally to simplify the development of the framework. For instance, portions of the JAWS framework use the string and vector containers provided by the C++ Standard Template Library [11] to manage connection maps and other search structures. In addition, application-specific callbacks invoked by framework event handlers frequently use class library components to perform basic tasks such as string processing, file management, and numerical analysis.

2 The JAWS Web Server Framework

Figure 3 illustrates the major structural components and design patterns that comprise the JAWS Adaptive Web Server (JAWS) framework. JAWS is designed to allow the customization of various Web server strategies in response to environmental factors. These factors include *static* factors (*e.g.*, number of available CPUs, support for kernel-level threads, and availability of asynchronous I/O in the OS), as well as *dynamic* factors

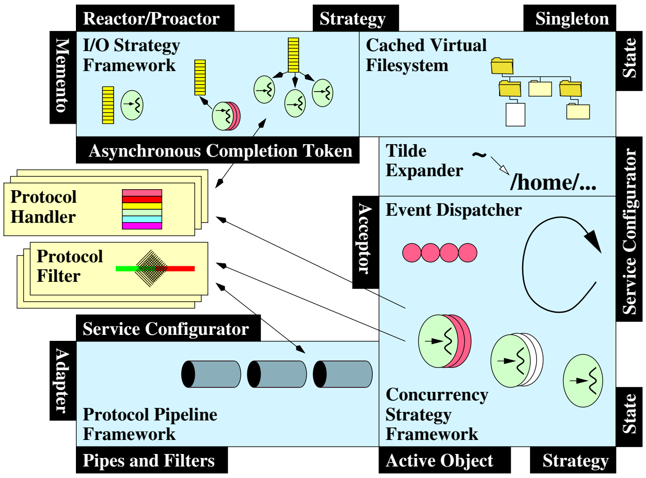

Figure 3: Architectural Overview of the JAWS Framework

(*e.g.*, Web traffic patterns and workload characteristics).

2.1 Components and Patterns in JAWS

JAWS is structured as a *framework of frameworks*. The overall JAWS framework contains the following components and frameworks: an *Event Dispatcher*, *Concurrency Strategy*, *I/O Strategy*, *Protocol Pipeline*, *Protocol Handlers*, and *Cached Virtual Filesystem*. Each framework is structured as a set of collaborating objects implemented using components in ACE [12]. The collaborations among JAWS components and frameworks are guided by a family of patterns, which are listed along the borders in Figure 3. An outline of the key frameworks, components, and patterns in JAWS is presented below.[1]

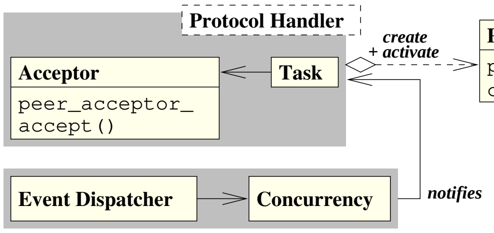

Figure 4: Structure of the Acceptor Pattern in JAWS

Event Dispatcher: This component is responsible for coordinating JAWS' *Concurrency Strategy* with its *I/O Strategy*.

[1]Due to space limitations it is not possible to describe each pattern in detail. The references provide additional information on each pattern mentioned below.

As illustrated in Figure 4, the passive establishment of connection events with Web clients follows the *Acceptor* pattern [13]. New incoming HTTP request events are serviced by a concurrency strategy. As events are processed, they are dispatched to the *Protocol Handler*, which is parameterized by an I/O strategy. JAWS ability to dynamically bind to a particular concurrency strategy and I/O strategy from a range of alternatives follows the *Strategy* pattern [6].

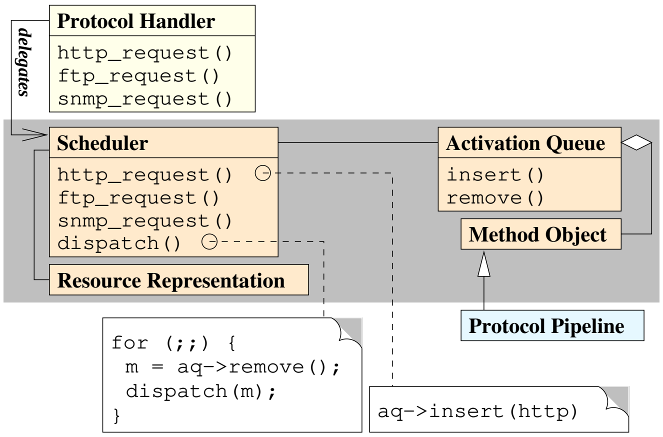

Figure 5: Structure of the Active Object Pattern in JAWS

Concurrency Strategy: This framework implements concurrency mechanisms (such as single-threaded, thread-per-request, or thread pool) that can be selected adaptively at run-time using the *State* pattern [6] or pre-determined at initialization-time. The *Service Configurator* pattern [14] is used to configure a particular concurrency strategy into a Web server at run-time. When concurrency involves multiple threads, the strategy creates protocol handlers that follow the *Active Object* pattern [8]. This is illustrated in Figure 5.

I/O Strategy: This framework implements various I/O mechanisms, such as asynchronous, synchronous and reactive I/O. Multiple I/O mechanisms can be used simultaneously. In JAWS, asynchronous I/O is implemented using the *Asynchronous Completion Token* [15] pattern and *Proactor* [16] pattern, as illustrated in Figure 6. Reactive I/O is accomplished through the *Reactor* pattern [7]. Reactive I/O utilizes the *Memento* pattern [6] to capture and externalize the state of a request so that it can be restored at a later time.

Protocol Handler: This framework allows system developers to apply the JAWS framework to a variety of Web system applications. A *Protocol Handler* is parameterized by a concurrency strategy and an I/O strategy. These strategies are decoupled from the protocol handler using the *Adapter* [6] pattern. In JAWS, this component implements the parsing and

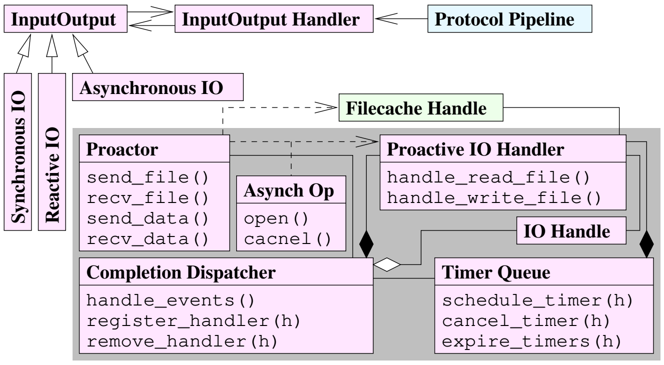

Figure 6: Structure of the Proactor Pattern in JAWS

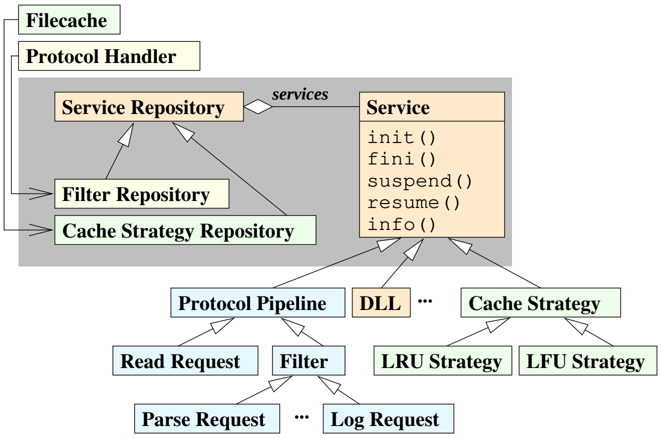

Figure 7: The Service Configurator Pattern in JAWS

handling of HTTP/1.0 request methods. The abstraction allows for other protocols (such as HTTP/1.1, DICOM, and SFP [17]) to be incorporated easily into JAWS. To add a new protocol, developers simply write a new *Protocol Handler* implementation, which is then configured into the JAWS framework.

Protocol Pipeline: This framework allows filter operations to be incorporated easily with the data being processed by the *Protocol Handler*. This integration is achieved by employing the Adapter pattern. Pipelines follow the *Pipes and Filters* pattern [18] for input processing. Pipeline components can be linked dynamically at run-time using the *Service Configurator* pattern, as shown in Figure 7.

Cached Virtual Filesystem: This component improves Web server performance by reducing the overhead of filesystem access. Various caching strategies, such as LRU, LFU, Hinted, and Structured, can be selected following the *Strategy* pattern [6]. This allows different caching strategies to be profiled and selected based on their performance. Moreover, optimal strategies to be configured statically or dynamically using the *Service Configurator* pattern, as shown in Figure 7. The cache for each Web server is instantiated using the *Singleton* pattern [6].

Tilde Expander: This component is another cache component that uses a perfect hash table [19] that maps abbreviated user login names (*e.g.*, `~schmidt`) to user home directories (*e.g.*, `/home/cs/faculty/schmidt`). When personal Web pages are stored in user home directories, and user directories do not reside in one common root, this component substantially reduces the disk I/O overhead required to access a system user information file, such as `/etc/passwd`. By virtue of the *Service Configurator* pattern, the Tilde Expander can be unlinked and relinked dynamically into the server when a new user is added to the system.

2.2 JAWS Web Server Performance

Our research [20, 21] demonstrates that it is possible to improve server performance through superior server design (a similar observation was made in [22]). Thus, while a "hard-coded" server, *i.e.*, one that uses fixed concurrency, I/O, and caching strategies, can provide excellent performance, a flexible server framewor like JAWS need necessarily not perform poorly.

Figure 8 below illustrates how the flexible nature of the JAWS framework enables it to adapt from its baseline performance to perform as well as, and in some cases better than, state of the art commercial Web servers produced by Zeus and Netscape. We achieved this level of performance through systematic benchmarking of different configurations of JAWS under different server load conditions. We then selected the combination of features that yielded the best overall performance [21].

3 Concluding Remarks

Computing power and network bandwidth has increased dramatically over the past decade. However, the development of high-performance Web servers has remained expensive and error-prone. Much of the cost and effort stems from the repeated rediscovery and reinvention of fundamental design patterns and framework components. Moreover, the growing heterogeneity of hardware architectures and diversity of OS and network platforms makes it hard to build correct, portable, and efficient Web servers from scratch.

In general, OO application frameworks and patterns help to reduce the cost and improve the quality of communication software [23]. In the context of Web servers, these benefits

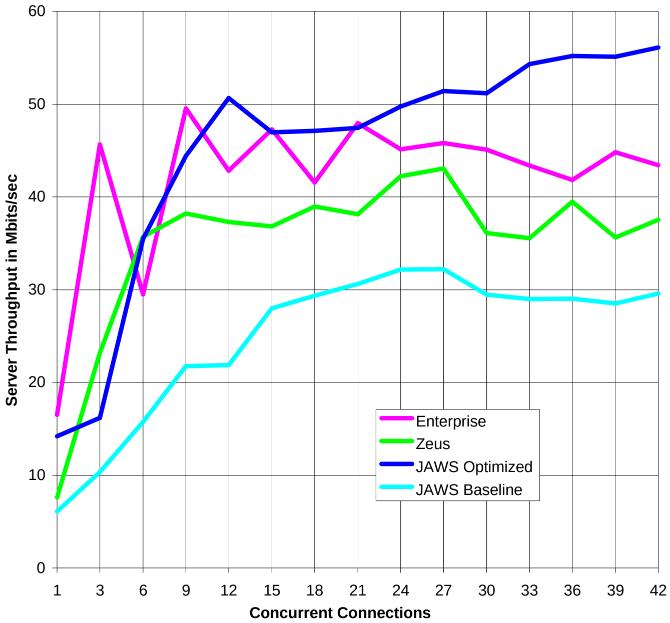

Figure 8: Comparative Performance for JAWS

accrue from leveraging proven software designs and reusable components that can be customized to meet new application requirements.

The JAWS framework described in this article exemplifies how high-performance Web server software development can be simplified and unified. One measure of success of the JAWS framework is illustrated by the fact that it outperforms other commercial and non-commercial Web servers. Commercial Web servers, such as Netscape Enterprise and Zeus, provide excellent performance, but their techniques for doing so remain behind the veils of proprietary software. Free Web servers, such as Apache and PHTTPD, provide good performance but the lack the architectural reuse that the JAWS framework provides.

The JAWS framework is freely available at `www.cs.wustl.edu/~schmidt/ACE.html`. This URL contains complete source code, documentation, and further technical information on JAWS.

References

[1] D. C. Schmidt and C. Cleeland, "Applying Patterns to Develop Extensible and Maintainable ORB Middleware," *Communications of the ACM, to appear*, 1998.

[2] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman, *The Design and Implementation of the 4.4BSD Operating System*. Addison Wesley, 1996.

[3] "Information Technology – Portable Operating System Interface (POSIX) – Part 1: System Application: Program Interface (API) [C Language]," 1995.

[4] W. R. Stevens, *UNIX Network Programming, Second Edition*. Englewood Cliffs, NJ: Prentice Hall, 1997.

[5] M. E. Fayad and D. C. Schmidt, "Object-Oriented Application Frameworks," *Communications of the ACM*, vol. 40, October 1997.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

[7] D. C. Schmidt, "Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching," in *Pattern Languages of Program Design* (J. O. Coplien and D. C. Schmidt, eds.), pp. 529–545, Reading, MA: Addison-Wesley, 1995.

[8] R. G. Lavender and D. C. Schmidt, "Active Object: an Object Behavioral Pattern for Concurrent Programming," in *Pattern Languages of Program Design* (J. O. Coplien, J. Vlissides, and N. Kerth, eds.), Reading, MA: Addison-Wesley, 1996.

[9] D. C. Schmidt, "Experience Using Design Patterns to Develop Reuseable Object-Oriented Communication Software," *Communications of the ACM (Special Issue on Object-Oriented Experiences)*, vol. 38, October 1995.

[10] R. Johnson, "Frameworks = Patterns + Components," *Communications of the ACM*, vol. 40, Oct. 1997.

[11] A. Stepanov and M. Lee, "The Standard Template Library," Tech. Rep. HPL-94-34, Hewlett-Packard Laboratories, April 1994.

[12] D. C. Schmidt, "ACE: an Object-Oriented Framework for Developing Distributed Applications," in *Proceedings of the* 6^{th} *USENIX C++ Technical Conference*, (Cambridge, Massachusetts), USENIX Association, April 1994.

[13] D. C. Schmidt, "Acceptor and Connector: Design Patterns for Initializing Communication Services," in *Pattern Languages of Program Design* (R. Martin, F. Buschmann, and D. Riehle, eds.), Reading, MA: Addison-Wesley, 1997.

[14] P. Jain and D. C. Schmidt, "Service Configurator: A Pattern for Dynamic Configuration of Services," in *Proceedings of the* 3^{rd} *Conference on Object-Oriented Technologies and Systems*, USENIX, June 1997.

[15] I. Pyarali, T. H. Harrison, and D. C. Schmidt, "Asynchronous Completion Token: an Object Behavioral Pattern for Efficient Asynchronous Event Handling," in *Pattern Languages of Program Design* (R. Martin, F. Buschmann, and D. Riehle, eds.), Reading, MA: Addison-Wesley, 1997.

[16] T. Harrison, I. Pyarali, D. C. Schmidt, and T. Jordan, "Proactor – An Object Behavioral Pattern for Dispatching Asynchronous Event Handlers," in *The* 4^{th} *Pattern Languages of Programming Conference (Washington University technical report #WUCS-97-34)*, September 1997.

[17] Object Management Group, *Control and Management of Audio/Video Streams: OMG RFP Submission*, 1.2 ed., Mar. 1997.

[18] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture - A System of Patterns*. Wiley and Sons, 1996.

[19] D. C. Schmidt, "GPERF: A Perfect Hash Function Generator," in *Proceedings of the* 2^{nd} *C++ Conference*, (San Francisco, California), pp. 87–102, USENIX, April 1990.

[20] J. Hu, I. Pyarali, and D. C. Schmidt, "Measuring the Impact of Event Dispatching and Concurrency Models on Web Server Performance Over High-speed Networks," in *Proceedings of the* 2^{nd} *Global Internet Conference*, IEEE, November 1997.

[21] J. Hu, S. Mungee, and D. C. Schmidt, "Principles for Developing and Measuring High-performance Web Servers over ATM," in *Proceeedings of INFOCOM '98*, March/April 1998.

[22] H. F. Nielsen, J. Gettys, A. Baird-Smith, E. Prud'hommeaux, H. W. Lie, and C. Lilley, "Network Performance Effects of HTTP/1.1, CSS1, and PNG," in *To appear in Proceedings of ACM SIGCOMM '97*, 1997.

[23] D. C. Schmidt and M. E. Fayad, "Lessons Learned: Building Reusable OO Frameworks for Distributed Software," *Communications of the ACM*, vol. 40, October 1997.