UNIVERSITY OF CALIFORNIA,
IRVINE

Architectural Styles and the Design of Network-based Software Architectures

DISSERTATION

submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in Information and Computer Science

by

Roy Thomas Fielding

Dissertation Committee:
Professor Richard N. Taylor, Chair
Professor Mark S. Ackerman
Professor David S. Rosenblum

2000

CHAPTER 2
Network-based Application Architectures

This chapter continues our discussion of background material by focusing on network-based application architectures and describing how styles can be used to guide their architectural design.

2.1 Scope

Architecture is found at multiple levels within software systems. This dissertation examines the highest level of abstraction in software architecture, where the interactions among components are capable of being realized in network communication. We limit our discussion to styles for network-based application architectures in order to reduce the dimensions of variance among the styles studied.

2.1.1 Network-based vs. Distributed

The primary distinction between network-based architectures and software architectures in general is that communication between components is restricted to message passing [6], or the equivalent of message passing if a more efficient mechanism can be selected at run-time based on the location of components [128].

Tanenbaum and van Renesse [127] make a distinction between distributed systems and network-based systems: a distributed system is one that looks to its users like an ordinary centralized system, but runs on multiple, independent CPUs. In contrast, network-based systems are those capable of operation across a network, but not

necessarily in a fashion that is transparent to the user. In some cases it is desirable for the user to be aware of the difference between an action that requires a network request and one that is satisfiable on their local system, particularly when network usage implies an extra transaction cost [133]. This dissertation covers network-based systems by not limiting the candidate styles to those that preserve transparency for the user.

2.1.2 Application Software vs. Networking Software

Another restriction on the scope of this dissertation is that we limit our discussion to application architectures, excluding the operating system, networking software, and some architectural styles that would only use a network for system support (e.g., process control styles [53]). Applications represent the "business-aware" functionality of a system [131].

Application software architecture is an abstraction level of an overall system, in which the goals of a user action are representable as functional architectural properties. For example, a hypermedia application must be concerned with the location of information pages, performing requests, and rendering data streams. This is in contrast to a networking abstraction, where the goal is to move bits from one location to another without regard to why those bits are being moved. It is only at the application level that we can evaluate design trade-offs based on the number of interactions per user action, the location of application state, the effective throughput of all data streams (as opposed to the potential throughput of a single data stream), the extent of communication being performed per user action, etc.

2.2 Evaluating the Design of Application Architectures

One of the goals of this dissertation is to provide design guidance for the task of selecting or creating the most appropriate architecture for a given application domain, keeping in mind that an architecture is the realization of an architectural design and not the design itself. An architecture can be evaluated by its run-time characteristics, but we would obviously prefer an evaluation mechanism that could be applied to the candidate architectural designs before having to implement all of them. Unfortunately, architectural designs are notoriously hard to evaluate and compare in an objective manner. Like most artifacts of creative design, architectures are normally presented as a completed work, as if the design simply sprung fully-formed from the architect's mind. In order to evaluate an architectural design, we need to examine the design rationale behind the constraints it places on a system, and compare the properties derived from those constraints to the target application's objectives.

The first level of evaluation is set by the application's functional requirements. For example, it makes no sense to evaluate the design of a process control architecture against the requirements of a distributed hypermedia system, since the comparison is moot if the architecture would not function. Although this will eliminate some candidates, in most cases there will remain many other architectural designs that are capable of meeting the application's functional needs. The remainder differ by their relative emphasis on the non-functional requirements—the degree to which each architecture would support the various non-functional architectural properties that have been identified as necessary for the system. Since properties are created by the application of architectural constraints, it is possible to evaluate and compare different architectural designs by identifying the

constraints within each architecture, evaluating the set of properties induced by each constraint, and comparing the cumulative properties of the design to those properties required of the application.

As described in the previous chapter, an architectural style is a coordinated set of architectural constraints that has been given a name for ease of reference. Each architectural design decision can be seen as an application of a style. Since the addition of a constraint may derive a new style, we can think of the space of all possible architectural styles as a derivation tree, with its root being the null style (empty set of constraints). When their constraints do not conflict, styles can be combined to form hybrid styles, eventually culminating in a hybrid style that represents a complete abstraction of the architectural design. An architectural design can therefore be analyzed by breaking-down its set of constraints into a derivation tree and evaluating the cumulative effect of the constraints represented by that tree. If we understand the properties induced by each basic style, then traversing the derivation tree gives us an understanding of the overall design's architectural properties. The specific needs of an application can then be matched against the properties of the design. Comparison becomes a relatively simple matter of identifying which architectural design satisfies the most desired properties for that application.

Care must be taken to recognize when the effects of one constraint may counteract the benefits of some other constraint. Nevertheless, it is possible for an experienced software architect to build such a derivation tree of architectural constraints for a given application domain, and then use the tree to evaluate many different architectural designs for applications within that domain. Thus, building a derivation tree provides a mechanism for architectural design guidance.

The evaluation of architectural properties within a tree of styles is specific to the needs of a particular application domain because the impact of a given constraint is often dependent on the application characteristics. For example, the pipe-and-filter style enables several positive architectural properties when used within a system that requires data transformations between components, whereas it would add nothing but overhead to a system that consists only of control messages. Since it is rarely useful to compare architectural designs across different application domains, the simplest means of ensuring consistency is to make the tree domain-specific.

Design evaluation is frequently a question of choosing between trade-offs. Perry and Wolf [105] describe a method of recognizing trade-offs explicitly by placing a numeric weight against each property to indicate its relative importance to the architecture, thus providing a normalized metric for comparing candidate designs. However, in order to be a meaningful metric, each weight would have to be carefully chosen using an objective scale that is consistent across all properties. In practice, no such scale exists. Rather than having the architect fiddle with weight values until the result matches their intuition, I prefer to present all of the information to the architect in a readily viewable form, and let the architect's intuition be guided by the visual pattern. This will be demonstrated in the next chapter.

2.3 Architectural Properties of Key Interest

This section describes the architectural properties used to differentiate and classify architectural styles in this dissertation. It is not intended to be a comprehensive list. I have included only those properties that are clearly influenced by the restricted set of styles

surveyed. Additional properties, sometimes referred to as software qualities, are covered by most textbooks on software engineering (e.g., [58]). Bass et al. [9] examine qualities in regards to software architecture.

2.3.1 Performance

One of the main reasons to focus on styles for network-based applications is because component interactions can be the dominant factor in determining user-perceived performance and network efficiency. Since the architectural style influences the nature of those interactions, selection of an appropriate architectural style can make the difference between success and failure in the deployment of a network-based application.

The performance of a network-based application is bound first by the application requirements, then by the chosen interaction style, followed by the realized architecture, and finally by the implementation of each component. In other words, software cannot avoid the basic cost of achieving the application needs; e.g., if the application requires that data be located on system A and processed on system B, then the software cannot avoid moving that data from A to B. Likewise, an architecture cannot be any more efficient than its interaction style allows; e.g., the cost of multiple interactions to move the data from A to B cannot be any less than that of a single interaction from A to B. Finally, regardless of the quality of an architecture, no interaction can take place faster than a component implementation can produce data and its recipient can consume data.

2.3.1.1 Network Performance

Network performance measures are used to describe some attributes of communication. *Throughput* is the rate at which information, including both application data and

communication overhead, is transferred between components. *Overhead* can be separated into initial setup overhead and per-interaction overhead, a distinction which is useful for identifying connectors that can share setup overhead across multiple interactions (*amortization*). *Bandwidth* is a measure of the maximum available throughput over a given network link. *Usable bandwidth* refers to that portion of bandwidth which is actually available to the application.

Styles impact network performance by their influence on the number of interactions per user action and the granularity of data elements. A style that encourages small, strongly typed interactions will be efficient in an application involving small data transfers among known components, but will cause excessive overhead within applications that involve large data transfers or negotiated interfaces. Likewise, a style that involves the coordination of multiple components arranged to filter a large data stream will be out of place in an application that primarily requires small control messages.

2.3.1.2 User-perceived Performance

User-perceived performance differs from network performance in that the performance of an action is measured in terms of its impact on the user in front of an application rather than the rate at which the network moves information. The primary measures for user-perceived performance are latency and completion time.

Latency is the time period between initial stimulus and the first indication of a response. Latency occurs at several points in the processing of a network-based application action: 1) the time needed for the application to recognize the event that initiated the action; 2) the time required to setup the interactions between components; 3) the time required to transmit each interaction to the components; 4) the time required to

process each interaction on those components; and, 5) the time required to complete sufficient transfer and processing of the result of the interactions before the application is able to begin rendering a usable result. It is important to note that, although only (3) and (5) represent actual network communication, all five points can be impacted by the architectural style. Furthermore, multiple component interactions per user action are additive to latency unless they take place in parallel.

Completion is the amount of time taken to complete an application action. Completion time is dependent upon all of the aforementioned measures. The difference between an action's completion time and its latency represents the degree to which the application is incrementally processing the data being received. For example, a Web browser that can render a large image while it is being received provides significantly better user-perceived performance than one that waits until the entire image is completely received prior to rendering, even though both experience the same network performance.

It is important to note that design considerations for optimizing latency will often have the side-effect of degrading completion time, and vice versa. For example, compression of a data stream can produce a more efficient encoding if the algorithm samples a significant portion of the data before producing the encoded transformation, resulting in a shorter completion time to transfer the encoded data across the network. However, if this compression is being performed on-the-fly in response to a user action, then buffering a large sample before transfer may produce an unacceptable latency. Balancing these trade-offs can be difficult, particularly when it is unknown whether the recipient cares more about latency (e.g., Web browsers) or completion (e.g., Web spiders).

2.3.1.3 Network Efficiency

An interesting observation about network-based applications is that the best application performance is obtained by not using the network. This essentially means that the most efficient architectural styles for a network-based application are those that can effectively minimize use of the network when it is possible to do so, through reuse of prior interactions (caching), reduction of the frequency of network interactions in relation to user actions (replicated data and disconnected operation), or by removing the need for some interactions by moving the processing of data closer to the source of the data (mobile code).

The impact of the various performance issues is often related to the scope of distribution for the application. The benefits of a style under local conditions may become drawbacks when faced with global conditions. Thus, the properties of a style must be framed in relation to the interaction distance: within a single process, across processes on a single host, inside a local-area network (LAN), or spread across a wide-area network (WAN). Additional concerns become evident when interactions across a WAN, where a single organization is involved, are compared to interactions across the Internet, involving multiple trust boundaries.

2.3.2 Scalability

Scalability refers to the ability of the architecture to support large numbers of components, or interactions among components, within an active configuration. Scalability can be improved by simplifying components, by distributing services across many components (decentralizing the interactions), and by controlling interactions and configurations as a

result of monitoring. Styles influence these factors by determining the location of application state, the extent of distribution, and the coupling between components.

Scalability is also impacted by the frequency of interactions, whether the load on a component is distributed evenly over time or occurs in peaks, whether an interaction requires guaranteed delivery or a best-effort, whether a request involves synchronous or asynchronous handling, and whether the environment is controlled or anarchic (i.e., can you trust the other components?).

2.3.3 Simplicity

The primary means by which architectural styles induce simplicity is by applying the principle of separation of concerns to the allocation of functionality within components. If functionality can be allocated such that the individual components are substantially less complex, then they will be easier to understand and implement. Likewise, such separation eases the task of reasoning about the overall architecture. I have chosen to lump the qualities of complexity, understandability, and verifiability under the general property of simplicity, since they go hand-in-hand for a network-based system.

Applying the principle of generality to architectural elements also improves simplicity, since it decreases variation within an architecture. Generality of connectors leads to middleware [22].

2.3.4 Modifiability

Modifiability is about the ease with which a change can be made to an application architecture. Modifiability can be further broken down into evolvability, extensibility, customizability, configurability, and reusability, as described below. A particular concern

of network-based systems is dynamic modifiability [98], where the modification is made to a deployed application without stopping and restarting the entire system.

Even if it were possible to build a software system that perfectly matches the requirements of its users, those requirements will change over time just as society changes over time. Because the components participating in a network-based application may be distributed across multiple organizational boundaries, the system must be prepared for gradual and fragmented change, where old and new implementations coexist, without preventing the new implementations from making use of their extended capabilities.

2.3.4.1 Evolvability

Evolvability represents the degree to which a component implementation can be changed without negatively impacting other components. Static evolution of components generally depends on how well the architectural abstraction is enforced by the implementation, and thus is not something unique to any particular architectural style. Dynamic evolution, however, can be influenced by the style if it includes constraints on the maintenance and location of application state. The same techniques used to recover from partial failure conditions in a distributed system [133] can be used to support dynamic evolution.

2.3.4.2 Extensibility

Extensibility is defined as the ability to add functionality to a system [108]. Dynamic extensibility implies that functionality can be added to a deployed system without impacting the rest of the system. Extensibility is induced within an architectural style by reducing the coupling between components, as exemplified by event-based integration.

2.3.4.3 Customizability

Customizability refers to the ability to temporarily specialize the behavior of an architectural element, such that it can then perform an unusual service. A component is customizable if it can be extended by one client of that component's services without adversely impacting other clients of that component [50]. Styles that support customization may also improve simplicity and scalability, since service components can be reduced in size and complexity by directly implementing only the most frequent services and allowing infrequent services to be defined by the client. Customizability is a property induced by the remote evaluation and code-on-demand styles.

2.3.4.4 Configurability

Configurability is related to both extensibility and reusability in that it refers to post-deployment modification of components, or configurations of components, such that they are capable of using a new service or data element type. The pipe-and-filter and code-on-demand styles are two examples that induce configurability of configurations and components, respectively.

2.3.4.5 Reusability

Reusability is a property of an application architecture if its components, connectors, or data elements can be reused, without modification, in other applications. The primary mechanisms for inducing reusability within architectural styles is reduction of coupling (knowledge of identity) between components and constraining the generality of component interfaces. The uniform pipe-and-filter style exemplifies these types of constraints.

2.3.5 Visibility

Styles can also influence the visibility of interactions within a network-based application by restricting interfaces via generality or providing access to monitoring. Visibility in this case refers to the ability of a component to monitor or mediate the interaction between two other components. Visibility can enable improved performance via shared caching of interactions, scalability through layered services, reliability through reflective monitoring, and security by allowing the interactions to be inspected by mediators (e.g., network firewalls). The mobile agent style is an example where the lack of visibility may lead to security concerns.

This usage of the term visibility differs from that in Ghezzi et al. [58], where they are referring to visibility into the development process rather than the product.

2.3.6 Portability

Software is portable if it can run in different environments [58]. Styles that induce portability include those that move code along with the data to be processed, such as the virtual machine and mobile agent styles, and those that constrain the data elements to a set of standardized formats.

2.3.7 Reliability

Reliability, within the perspective of application architectures, can be viewed as the degree to which an architecture is susceptible to failure at the system level in the presence of partial failures within components, connectors, or data. Styles can improve reliability by avoiding single points of failure, enabling redundancy, allowing monitoring, or reducing the scope of failure to a recoverable action.

2.4 Summary

This chapter examined the scope of the dissertation by focusing on network-based application architectures and describing how styles can be used to guide their architectural design. It also defined the set of architectural properties that will be used throughout the rest of the dissertation for the comparison and evaluation of architectural styles.

The next chapter presents a survey of common architectural styles for network-based application software within a classification framework that evaluates each style according to the architectural properties it would induce if applied to an architecture for a prototypical network-based hypermedia system.

REFERENCES

1. G. D. Abowd, R. Allen, and D. Garlan. Formalizing style to understand descriptions of software architecture. *ACM Transactions on Software Engineering and Methodology*, 4(4), Oct. 1995, pp. 319–364. A shorter version also appeared as: Using style to understand descriptions of software architecture. In *Proceedings of the First ACM SIGSOFT Symposium on the Foundations of Software Engineering (SIGSOFT'93)*, Los Angeles, CA, Dec. 1993, pp. 9–20.

2. Adobe Systems Inc. *PostScript Language Reference Manual*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1985.

3. C. Alexander. *The Timeless Way of Building*. Oxford University Press, New York, 1979.

4. C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, and S. Angel. *A Pattern Language*. Oxford University Press, New York, 1977.

5. R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3), July 1997. A shorter version also appeared as: Formalizing architectural connection. In *Proceedings of the 16th International Conference on Software Engineering*, Sorrento, Italy, May 1994, pp. 71–80. Also as: Beyond Definition/Use: Architectural Interconnection. In *Proceedings of the ACM Interface Definition Language Workshop*, Portland, Oregon, *SIGPLAN Notices*, 29(8), Aug. 1994.

6. G. Andrews. Paradigms for process interaction in distributed programs. *ACM Computing Surveys*, 23(1), Mar. 1991, pp. 49–90.

7. F. Anklesaria, et al. The Internet Gopher protocol (a distributed document search and retrieval protocol). *Internet RFC 1436*, Mar. 1993.

8. D. J. Barrett, L. A. Clarke, P. L. Tarr, A. E. Wise. A framework for event-based software integration. *ACM Transactions on Software Engineering and Methodology*, 5(4), Oct. 1996, pp. 378–421.

9. L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison Wesley, Reading, Mass., 1998.

10. D. Batory, L. Coglianese, S. Shafer, and W. Tracz. The ADAGE avionics reference architecture. In *Proceedings of AIAA Computing in Aerospace 10*, San Antonio, 1995.

11. T. Berners-Lee, R. Cailliau, and J.-F. Groff. World Wide Web. Flyer distributed at the *3rd Joint European Networking Conference*, Innsbruck, Austria, May 1992.

12. T. Berners-Lee, R. Cailliau, J.-F. Groff, and B. Pollermann. World-Wide Web: The information universe. *Electronic Networking: Research, Applications and Policy*, 2(1), Meckler Publishing, Westport, CT, Spring 1992, pp. 52–58.

13. T. Berners-Lee and R. Cailliau. World-Wide Web. In *Proceedings of Computing in High Energy Physics 92*, Annecy, France, 23–27 Sep. 1992.

14. T. Berners-Lee, R. Cailliau, C. Barker, and J.-F. Groff. W3 Project: Assorted design notes. Published on the Web, Nov. 1992. Archived at <http://www.w3.org/History/19921103-hypertext/hypertext/WWW/WorkingNotes/Overview.html>, Sep. 2000.

15. T. Berners-Lee. Universal Resource Identifiers in WWW. *Internet RFC 1630*, June 1994.

16. T. Berners-Lee, R. Cailliau, A. Luotonen, H. Frystyk Nielsen, and A. Secret. The World-Wide Web. *Communications of the ACM*, 37(8), Aug. 1994, pp. 76–82.

17. T. Berners-Lee, L. Masinter, and M. McCahill. Uniform Resource Locators (URL). *Internet RFC 1738*, Dec. 1994.

18. T. Berners-Lee and D. Connolly. Hypertext Markup Language — 2.0. *Internet RFC 1866*, Nov. 1995.

19. T. Berners-Lee, R. T. Fielding, and H. F. Nielsen. Hypertext Transfer Protocol — HTTP/1.0. *Internet RFC 1945*, May 1996.

20. T. Berners-Lee. WWW: Past, present, and future. *IEEE Computer*, 29(10), Oct. 1996, pp. 69–77.

21. T. Berners-Lee, R. T. Fielding, and L. Masinter. Uniform Resource Identifiers (URI): Generic syntax. *Internet RFC 2396*, Aug. 1998.

22. P. Bernstein. Middleware: A model for distributed systems services. *Communications of the ACM*, Feb. 1996, pp. 86–98.

23. A. D. Birrell and B. J. Nelson. Implementing remote procedure call. *ACM Transactions on Computer Systems*, 2, Jan. 1984, pp. 39–59.

24. M. Boasson. The artistry of software architecture. *IEEE Software*, 12(6), Nov. 1995, pp. 13–16.

25. G. Booch. Object-oriented development. *IEEE Transactions on Software Engineering*, 12(2), Feb. 1986, pp. 211–221.

26. C. Brooks, M. S. Mazer, S. Meeks, and J. Miller. Application-specific proxy servers as HTTP stream transducers. In *Proceedings of the Fourth International World Wide Web Conference*, Boston, Massachusetts, Dec. 1995, pp. 539–548.

27. F. Buschmann and R. Meunier. A system of patterns. Coplien and Schmidt (eds.), *Pattern Languages of Program Design*, Addison-Wesley, 1995, pp. 325–343.

28. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-oriented Software Architecture: A system of patterns*. John Wiley & Sons Ltd., England, 1996.

29. M. R. Cagan. The HP SoftBench Environment: An architecture for a new qeneration of software tools. *Hewlett-Packard Journal*, 41(3), June 1990, pp. 36–47.

30. J. R. Cameron. An overview of JSD. *IEEE Transactions on Software Engineering*, 12(2), Feb. 1986, pp. 222–240.

31. R. S. Chin and S. T. Chanson. Distributed object-based programming systems. *ACM Computing Surveys*, 23(1), Mar. 1991, pp. 91–124.

32. D. D. Clark and D. L. Tennenhouse. Architectural considerations for a new generation of protocols. In *Proceedings of ACM SIGCOMM'90 Symposium*, Philadelphia, PA, Sep. 1990, pp. 200–208.

33. J. O. Coplien and D. C. Schmidt, ed. *Pattern Languages of Program Design*. Addison-Wesley, Reading, Mass., 1995.

34. J. O. Coplien. Idioms and Patterns as Architectural Literature. *IEEE Software*, 14(1), Jan. 1997, pp. 36–42.

35. E. M. Dashofy, N. Medvidovic, R. N. Taylor. Using off-the-shelf middleware to implement connectors in distributed software architectures. In *Proceedings of the 1999 International Conference on Software Engineering*, Los Angeles, May 16–22, 1999, pp. 3–12.

36. F. Davis, et. al. *WAIS Interface Protocol Prototype Functional Specification (v.1.5)*. Thinking Machines Corporation, April 1990.

37. F. DeRemer and H. H. Kron. Programming-in-the-large versus programming-in-the-small. *IEEE Transactions on Software Engineering*, SE-2(2), June 1976, pp. 80–86.

38. E. Di Nitto and D. Rosenblum. Exploiting ADLs to specify architectural styles induced by middleware infrastructures. In *Proceedings of the 1999 International Conference on Software Engineering*, Los Angeles, May 16–22, 1999, pp. 13–22.

39. R. T. Fielding. Maintaining distributed hypertext infostructures: Welcome to MOMspider's web. *Computer Networks and ISDN Systems*, 27(2), Nov. 1994, pp. 193–204.

40. R. T. Fielding. Relative Uniform Resource Locators. *Internet RFC 1808*, June 1995.

41. R. T. Fielding, E. J. Whitehead, Jr., K. M. Anderson, G. Bolcer, P. Oreizy, and R. N. Taylor. Web-based development of complex information products. *Communications of the ACM*, 41(8), Aug. 1998, pp. 84–92.

42. R. T. Fielding, J. Gettys, J. C. Mogul, H. F. Nielsen, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol — HTTP/1.1. *Internet RFC 2616*, June 1999. [Obsoletes RFC 2068, Jan. 1997.]

43. R. T. Fielding and R. N. Taylor. Principled design of the modern Web architecture. In *Proceedings of the 2000 International Conference on Software Engineering (ICSE 2000)*, Limerick, Ireland, June 2000, pp. 407–416.

44. D. Flanagan. *JavaScript: The Definitive Guide, 3rd edition*. O'Reilly & Associates, Sebastopol, CA, 1998.

45. D. Flanagan. *Java™ in a Nutshell, 3rd edition*. O'Reilly & Associates, Sebastopol, CA, 1999.

46. J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, E. Sink, and L. Stewart. HTTP Authentication: Basic and Digest Access Authentication. *Internet RFC 2617*, June 1999.

47. N. Freed and N. Borenstein. Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies. *Internet RFC 2045*, Nov. 1996.

48. N. Freed, J. Klensin, and J. Postel. Multipurpose Internet Mail Extensions (MIME) Part Four: Registration Procedures. *Internet RFC 2048*, Nov. 1996.

49. M. Fridrich and W. Older. Helix: The architecture of the XMS distributed file system. *IEEE Software*, 2, May 1985, pp. 21–29.

50. A. Fuggetta, G. P. Picco, and G. Vigna. Understanding code mobility. *IEEE Transactions on Software Engineering*, 24(5), May 1998, pp. 342–361.

51. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley, Reading, Mass., 1995.

52. D. Garlan and E. Ilias. Low-cost, adaptable tool integration policies for integrated environments. In *Proceedings of the ACM SIGSOFT '90: Fourth Symposium on Software Development Environments*, Dec. 1990, pp. 1–10.

53. D. Garlan and M. Shaw. An introduction to software architecture. Ambriola & Tortola (eds.), *Advances in Software Engineering & Knowledge Engineering*, vol. II, World Scientific Pub Co., Singapore, 1993, pp. 1–39.

54. D. Garlan, R. Allen, and J. Ockerbloom. Exploiting style in architectural design environments. In *Proceedings of the Second ACM SIGSOFT Symposium on the Foundations of Software Engineering (SIGSOFT'94)*, New Orleans, Dec. 1994, pp. 175–188.

55. D. Garlan and D. E. Perry. Introduction to the special issue on software architecture. *IEEE Transactions on Software Engineering*, 21(4), Apr. 1995, pp. 269–274.

56. D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch, or, Why it's hard to build systems out of existing parts. In *Proceedings of the 17th International Conference on Software Engineering*, Seattle, WA, 1995. Also appears as: Architectural mismatch: Why reuse is so hard. *IEEE Software*, 12(6), Nov. 1995, pp. 17–26.

57. D. Garlan, R. Monroe, and D. Wile. ACME: An architecture description language. In *Proceedings of CASCON'97*, Nov. 1997.

58. C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering*. Prentice-Hall, 1991.

59. S. Glassman. A caching relay for the World Wide Web. *Computer Networks and ISDN Systems*, 27(2), Nov. 1994, pp. 165–173.

60. Y. Goland, E. J. Whitehead, Jr., A. Faizi, S. Carter, and D. Jensen. HTTP Extensions for Distributed Authoring — WEBDAV. *Internet RFC 2518*, Feb. 1999.

61. K. Grønbaek and R. H. Trigg. Design issues for a Dexter-based hypermedia system. *Communications of the ACM*, 37(2), Feb. 1994, pp. 41–49.

62. B. Hayes-Roth, K. Pfleger, P. Lalanda, P. Morignot, and M. Balabanovic. A domain-specific software architecture for adaptive intelligent systems. *IEEE Transactions on Software Engineering*, 21(4), Apr. 1995, pp. 288–301.

63. J. Heidemann, K. Obraczka, and J. Touch. Modeling the performance of HTTP over several transport protocols. *IEEE/ACM Transactions on Networking*, 5(5), Oct. 1997, pp. 616–630.

64. K. Holtman and A. Mutz. Transparent content negotiation in HTTP. *Internet RFC 2295*, Mar. 1998.

65. P. Inverardi and A. L. Wolf. Formal specification and analysis of software architectures using the chemical abstract machine model. *IEEE Transactions on Software Engineering*, 21(4), Apr. 1995, pp. 373–386.

66. ISO/IEC JTC1/SC21/WG7. *Reference Model of Open Distributed Processing*. ITU-T X.901: ISO/IEC 10746-1, 07 June 1995.

67. M. Jackson. Problems, methods, and specialization. *IEEE Software*, 11(6), [condensed from Software Engineering Journal], Nov. 1994. pp. 57–62.

68. R. Kazman, L. Bass, G. Abowd, and M. Webb. SAAM: A method for analyzing the properties of software architectures. In *Proceedings of the 16th International Conference on Software Engineering*, Sorrento, Italy, May 1994, pp. 81–90.

69. R. Kazman, M. Barbacci, M. Klein, S. J. Carrière, and S. G. Woods. Experience with performing architecture tradeoff analysis. In *Proceedings of the 1999 International Conference on Software Engineering*, Los Angeles, May 16–22, 1999, pp. 54–63.

70. N. L. Kerth and W. Cunningham. Using patterns to improve our architectural vision. *IEEE Software*, 14(1), Jan. 1997, pp. 53–59.

71. R. Khare and S. Lawrence. Upgrading to TLS within HTTP/1.1. *Internet RFC 2817*, May 2000.

72. G. E. Krasner and S. T. Pope. A cookbook for using the Model-View-Controller user interface paradigm in Smalltalk-80. *Journal of Object Oriented Programming*, 1(3), Aug.–Sep. 1988, pp. 26–49.

73. D. Kristol and L. Montulli. HTTP State Management Mechanism. *Internet RFC 2109*, Feb. 1997.

74. P. B. Kruchten. The 4+1 View Model of architecture. *IEEE Software*, 12(6), Nov. 1995, pp. 42–50.

75. D. Le Métayer. Describing software architectural styles using graph grammars. *IEEE Transactions on Software Engineering*, 24(7), July 1998, pp. 521–533.

76. W. C. Loerke. On Style in Architecture. F. Wilson, *Architecture: Fundamental Issues*, Van Nostrand Reinhold, New York, 1990, pp. 203–218.

77. D. C. Luckham, J. J. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering*, 21(4), Apr. 1995, pp. 336–355.

78. D. C. Luckham and J. Vera. An event-based architecture definition language. *IEEE Transactions on Software Engineering*, 21(9), Sep. 1995, pp. 717–734.

79. A. Luotonen and K. Altis. World-Wide Web proxies. *Computer Networks and ISDN Systems*, 27(2), Nov. 1994, pp. 147–154.

80. P. Maes. Concepts and experiments in computational reflection. In *Proceedings of OOPSLA '87*, Orlando, Florida, Oct. 1987, pp. 147–155.

81. J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *Proceedings of the 5th European Software Engineering Conference (ESEC'95)*, Sitges, Spain, Sep. 1995, pp. 137–153.

82. J. Magee and J. Kramer. Dynamic structure in software architectures. In *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering (SIGSOFT'96)*, San Francisco, Oct. 1996, pp. 3–14.

83. F. Manola. Technologies for a Web object model. IEEE Internet Computing, 3(1), Jan.–Feb. 1999, pp. 38–47.

84. H. Maurer. *HyperWave: The Next-Generation Web Solution*. Addison-Wesley, Harlow, England, 1996.

85. M. J. Maybee, D. H. Heimbigner, and L. J. Osterweil. Multilanguage interoperability in distributed systems: Experience Report. In *Proceedings 18th International Conference on Software Engineering*, Berlin, Germany, Mar. 1996.

86. N. Medvidovic and R. N. Taylor. A framework for classifying and comparing architecture description languages. In *Proceedings of the 6th European Software Engineering Conference held jointly with the 5th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Zurich, Switzerland, Sep. 1997, pp. 60–76.

87. N. Medvidovic. *Architecture-based Specification-time Software Evolution*. Ph.D. Dissertation, University of California, Irvine, Dec. 1998.

88. N. Medvidovic, D. S. Rosenblum, and R. N. Taylor. A language and environment for architecture-based software development and evolution. In *Proceedings of the*

1999 International Conference on Software Engineering, Los Angeles, May 16–22, 1999, pp. 44–53.

89. A. Mockus, R. T. Fielding, and J. Herbsleb. A case study of open source software development: The Apache server. In *Proceedings of the 2000 International Conference on Software Engineering (ICSE 2000)*, Limerick, Ireland, June 2000, pp. 263–272.

90. J. Mogul, R. Fielding, J. Gettys, and H. Frystyk. Use and Interpretation of HTTP Version Numbers. *Internet RFC 2145*, May 1997.

91. R. T. Monroe, A. Kompanek, R. Melton, and D. Garlan. Architectural Styles, Design Patterns, and Objects. *IEEE Software*, 14(1), Jan. 1997, pp. 43–52.

92. M. Moriconi, X. Qian, and R. A. Riemenscheider. Correct architecture refinement. *IEEE Transactions on Software Engineering*, 21(4), Apr. 1995, pp. 356–372.

93. H. F. Nielsen, J. Gettys, A. Baird-Smith, E. Prud'hommeaux, H. Lie, and C. Lilley. Network Performance Effects of HTTP/1.1, CSS1, and PNG. *Proceedings of ACM SIGCOMM '97*, Cannes, France, Sep. 1997.

94. H. F. Nielsen, P. Leach, and S. Lawrence. HTTP extension framework, *Internet RFC 2774*, Feb. 2000.

95. H. Penny Nii. Blackboard systems. *AI Magazine*, 7(3):38–53 and 7(4):82–107, 1986.

96. Object Management Group. *Object Management Architecture Guide, Rev. 3.0*. Soley & Stone (eds.), New York: J. Wiley, 3rd ed., 1995.

97. Object Management Group. *The Common Object Request Broker: Architecture and Specification (CORBA 2.1)*. <http://www.omg.org/>, Aug. 1997.

98. P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture-based runtime software evolution. In *Proceedings of the 1998 International Conference on Software Engineering*, Kyoto, Japan, Apr. 1998.

99. P. Oreizy. Decentralized software evolution. Unpublished manuscript (Phase II Survey Paper), Dec. 1998.

100. V. N. Padmanabhan and J. C. Mogul. Improving HTTP latency. *Computer Networks and ISDN Systems*, 28, Dec. 1995, pp. 25–35.

101. D. L. Parnas. Information distribution aspects of design methodology. In Proceedings of IFIP Congress 71, Ljubljana, Aug. 1971, pp. 339–344.

102. D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12), Dec. 1972, pp. 1053–1058.

103. D. L. Parnas. Designing software for ease of extension and contraction. *IEEE Transactions on Software Engineering*, SE-5(3), Mar. 1979.

104. D. L. Parnas, P. C. Clements, and D. M. Weiss. The modular structure of complex systems. *IEEE Transactions on Software Engineering*, SE-11(3), 1985, pp. 259–266.

105. D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4), Oct. 1992, pp. 40–52.

106. J. Postel and J. Reynolds. TELNET Protocol Specification. *Internet STD 8, RFC 854*, May 1983.

107. J. Postel and J. Reynolds. File Transfer Protocol. *Internet STD 9, RFC 959*, Oct. 1985.

108. D. Pountain and C. Szyperski. Extensible software systems. *Byte*, May 1994, pp. 57–62.

109. R. Prieto-Diaz and J. M. Neighbors. Module interconnection languages. *Journal of Systems and Software*, 6(4), Nov. 1986, pp. 307–334.

110. J. M. Purtilo. The Polylith software bus. *ACM Transactions on Programming Languages and Systems*, 16(1), Jan. 1994, pp. 151–174.

111. M. Python. The Architects Sketch. *Monty Python's Flying Circus TV Show, Episode 17*, Sep. 1970. Transcript at <http://www.stone-dead.asn.au/sketches/architec.htm>.

112. J. Rasure and M. Young. Open environment for image processing and software development. In *Proceedings of the 1992 SPIE/IS&T Symposium on Electronic Imaging*, Vol. 1659, Feb. 1992.

113. S. P. Reiss. Connecting tools using message passing in the Field environment. *IEEE Software*, 7(4), July 1990, pp. 57–67.

114. D. S. Rosenblum and A. L. Wolf. A design framework for Internet-scale event observation and notification. In *Proceedings of the 6th European Software Engineering Conference held jointly with the 5th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Zurich, Switzerland, Sep. 1997, pp. 344–360.

115. R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun network filesystem. In *Proceedings of the Usenix Conference*, June 1985, pp. 119–130.

116. M. Shapiro. Structure and encapsulation in distributed systems: The proxy principle. In *Proceedings of the 6th International Conference on Distributed Computing Systems*, Cambridge, MA, May 1986, pp. 198–204.

117. M. Shaw. Toward higher-level abstractions for software systems. *Data & Knowledge Engineering*, 5, 1990, pp. 119–128.

118. M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnick. Abstractions for software architecture and tools to support them. *IEEE Transactions on Software Engineering*, 21(4), Apr. 1995, pp. 314–335.

119. M. Shaw. Comparing architectural design styles. *IEEE Software*, 12(6), Nov. 1995, pp. 27–41.

120. M. Shaw. Some patterns for software architecture. Vlissides, Coplien & Kerth (eds.), *Pattern Languages of Program Design, Vol. 2*, Addison-Wesley, 1996, pp. 255–269.

121. M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.

122. M. Shaw and P. Clements. A field guide to boxology: Preliminary classification of architectural styles for software systems. In *Proceedings of the Twenty-First Annual International Computer Software and Applications Conference (COMPSAC'97)*, Washington, D.C., Aug. 1997, pp. 6–13.

123. A. Sinha. Client-server computing. *Communications of the ACM*, 35(7), July 1992, pp. 77–98.

124. K. Sollins and L. Masinter. Functional requirements for Uniform Resource Names. *Internet RFC 1737*, Dec. 1994.

125. S. E. Spero. Analysis of HTTP performance problems. Published on the Web, <http://metalab.unc.edu/mdma-release/http-prob.html>, 1994.

126. K. J. Sullivan and D. Notkin. Reconciling environment integration and software evolution. *ACM Transactions on Software Engineering and Methodology*, 1(3), July 1992, pp. 229–268.

127. A. S. Tanenbaum and R. van Renesse. Distributed operating systems. *ACM Computing Surveys*, 17(4), Dec. 1985, pp. 419–470.

128. R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead Jr., J. E. Robbins, K. A. Nies, P. Oreizy, and D. L. Dubrow. A component- and message-based architectural style for GUI software. *IEEE Transactions on Software Engineering*, 22(6), June 1996, pp. 390–406.

129. W. Tephenhart and J. J. Cusick. A unified object topology. *IEEE Software*, 14(1), Jan. 1997, pp. 31–35.

130. W. Tracz. DSSA (domain-specific software architecture) pedagogical example. *Software Engineering Notes*, 20(3), July 1995, pp. 49–62.

131. A. Umar. *Object-Oriented Client/Server Internet Environments*. Prentice Hall PTR, 1997.

132. S. Vestal. MetaH programmer's manual, version 1.09. *Technical Report*, Honeywell Technology Center, Apr. 1996.

133. J. Waldo, G. Wyant, A. Wollrath, and S. Kendall. A note on distributed computing. *Technical Report SMLI TR-94-29*, Sun Microsystems Laboratories, Inc., Nov. 1994.

134. L. Wall, T. Christiansen, and R. L. Schwartz. *Programming Perl, 2nd ed*. O'Reilly & Associates, 1996.

135. E. J. Whitehead, Jr., R. T. Fielding, and K. M. Anderson. Fusing WWW and link server technology: One approach. In *Proceedings of the 2nd Workshop on Open Hypermedia Systems, Hypertext'96*, Washington, DC, Mar. 1996, pp. 81–86.

136. A. Wolman, G. Voelker, N. Sharma, N. Cardwell, M. Brown, T. Landray, D. Pinnel, A. Karlin, and H. Levy. Organization-based analysis of Web-object sharing and caching. In *Proceedings of the 2nd USENIX Conference on Internet Technologies and Systems (USITS)*, Oct. 1999.

137. W. Zimmer. Relationships between design patterns. Coplien and Schmidt (eds.), *Pattern Languages of Program Design*, Addison-Wesley, 1995, pp. 345–364.

138. H. Zimmerman. OSI reference model — The ISO model of architecture for open systems interconnection. *IEEE Transactions on Communications*, 28, Apr. 1980, pp. 425–432.