Software Architectural Styles for Network-based Applications

Roy T. Fielding
University of California, Irvine

Phase II Survey Paper

Draft 2.0

A software architecture determines how system components are identified and allocated, how the components interact to form a system, the amount and granularity of communication needed for interaction, and the interface protocols used for communication. For a network-based application, system performance is dominated by network communication. Therefore, selection of the appropriate architectural style(s) for use in designing the software architecture can make the difference between success and failure in the deployment of a network-based application.

Software architectural styles have been characterized by their control-flow and data-flow patterns, allocation of functionality across components, and component types. Unfortunately, none of these characterizations are useful for understanding how a style influences the set of architectural properties, or qualities, of a system. These properties include, among others, user-perceived performance, network efficiency, simplicity, modifiability, scalability, and portability.We use these style-induced architectural properties to classify styles for network-based applications with the goal of understanding why certain styles are better than others for some applications, thus providing additional guidance for software engineers faced with the task of architectural design.

Keywords

software architecture, software architectural style, network-based application, software design, software design patterns, pattern languages

1 Introduction

Excuse me ... did you say 'knives'?

— City Gent #1 (Michael Palin), The Architects Sketch [Python, 1970]

As predicted by Perry and Wolf [1992], software architecture has been a focal point for software engineering research in the 1990s. The complexity of modern software systems have necessitated a greater emphasis on componentized systems, where the implementation is partitioned into independent components that communicate to perform a desired task. Software architecture research investigates methods for determining how best to partition a system, how components identify and communicate with each other, how information is communicated, how elements of a system can evolve independently, and how all of the above can be described using formal and informal notations.

This survey explores a junction on the frontiers of two research disciplines in computer science: software and networking. Software research has long been concerned with the categorization of software designs and the development of design methodologies, but has rarely been able to objectively evaluate the impact of various design choices on system behavior. Networking research, in contrast, is focused on the details of generic communication behavior between systems and improving the performance of particular communication techniques, often ignoring the fact that changing the interaction style of an application can have more impact on performance than the communication protocols used for that interaction. My work is motivated by the desire to understand and evaluate the architectural design of network-based application software through principled use of architectural constraints, thereby obtaining the functional, performance, and social properties desired of an architecture. When given a name, a coordinated set of architectural constraints becomes an architectural style.

Some architectural styles are often portrayed as "silver bullet" solutions for all forms of software. However, a good designer should select a style that matches the needs of the particular problem being solved [Shaw, 1995]. Choosing the right architectural style for a network-based application requires an understanding of the problem domain [Jackson, 1994] and thereby the communication needs of the application, an awareness of the variety of architectural styles and the particular concerns they address, and the ability to anticipate the sensitivity of each interaction style to the characteristics of network-based communication [Waldo et al., 1994].

Section 2 examines various definitions of architecture and architectural styles and defines those terms that will be used throughout the remainder of this paper. Section 3 describes what is intended by network-based application architectures, defining the scope of this survey and the style-induced architectural properties used for style comparison. Section 4 describes the survey methodology. A classification of software architectural styles for network-based applications is presented in Section 5, followed in Section 6 by a discussion of related work within the larger research areas of software architecture and distributed systems. Finally, I conclude with some observations on the relevance of this work to software engineering research and practice.

2 Context within Software Architecture Research

In spite of the interest in software architecture as a field of research, there is little agreement among researchers as to what exactly should be included in the definition of architecture. In many cases, this has led to important aspects of architectural design being

overlooked by past research. This section defines a self-consistent terminology for software architecture based on an examination of existing definitions within the literature and my own insight with respect to network-based application architectures. Each definition, highlighted within a box for ease of reference, is followed by a discussion of how it is derived from, or compares to, related research.

2.1 Software Architecture

> A **software architecture** is defined by a configuration of architectural elements—components, connectors, and data—constrained in their relationships in order to achieve a desired set of architectural properties.

2.1.1 Architectural Elements

A comprehensive examination of the scope and intellectual basis for software architecture can be found in Perry and Wolf [1992]. They present a model that defines a software architecture as a set of architectural *elements* that have a particular *form*, explicated by a set of *rationale*. Architectural elements include processing, data, and connecting elements. Form is defined by the properties of the elements and the relationships among the elements — that is, the constraints on the elements. The rationale provides the underlying basis for the architecture by capturing the motivation for the choice of architectural style, the choice of elements, and the form.

The definitions I have chosen for software architecture are an elaborated version of those within the Perry and Wolf model, except that I use the more prevalent terms of *component* and *connector* to refer to processing and connecting elements, respectively, and exclude rationale. Although rationale is an important aspect of software architecture research and of architectural description in particular, including it within the definition of software architecture would imply that design documentation is part of the run-time system. The presence or absence of rationale can influence the evolution of an architecture, but, once constituted, the architecture is independent of its reasons for being. Reflective systems [Maes, 1987] can use the characteristics of past performance to change future behavior, but in doing so they are replacing one lower-level architecture with another lower-level architecture, rather than encompassing rationale within those architectures.

As an illustration, consider what happens to a building if its blueprints and design plans are burned. Does the building immediately collapse? No, since the properties by which the walls sustain the weight of the roof remain intact. An architecture has, by design, a set of properties that allow it to meet or exceed the system requirements. Ignorance of those properties may lead to later changes which violate the architecture, just as the replacement of a load-bearing wall with a large window frame may violate the structural stability of a building. Thus, instead of rationale, our definition of software architecture includes architectural properties. Rationale explicates those properties, and lack of rationale may result in gradual decay or degradation of the architecture over time, but the rationale itself is not part of the architecture.

A key feature of the model in Perry and Wolf [1992] is the distinction of the various element types. *Processing elements* are those that perform transformations on data, *data elements* are those that contain the information that is used and transformed, and *connecting elements* are the glue that holds the different pieces of the architecture together. I use the more prevalent terms of *components* and *connectors* to refer to processing and connecting elements, respectively.

Garlan and Shaw [1993] describe software architecture vaguely as system structure. An architecture of a specific system is a collection of computational *components* together with a description of the interactions between these components—the *connectors*. This definition is expanded upon in Shaw et al. [1995]: The architecture of a software system defines that system in terms of components and of interactions among those components. In addition to specifying the structure and topology of the system, the architecture shows the intended correspondence between the system requirements and elements of the constructed system. Further elaboration of this definition can be found in Shaw and Garlan [1996].

What is surprising about the Shaw et al. model is that, rather than defining the software's architecture as existing within the software, it is defining a description of the software's architecture as if that were the architecture. In the process, software architecture as a whole is reduced to what is commonly found in most informal architecture diagrams: boxes (components) and lines (connectors). Data elements, along with many of the dynamic aspects of real software architectures, are ignored.

Components

> A **component** is an abstract unit of software that provides a transformation of data via its interface.

Components are the most easily recognized aspect of software architecture. Perry and Wolf's [1992] processing elements are defined as those components that supply the transformation on the data elements. Garlan and Shaw [1993] describe components simply as the elements that perform computation. Our definition attempts to be more precise in making the distinction between components and the software within connectors.

A component is an abstract unit of software that provides a transformation of data via its interface. Example transformations include loading into memory from secondary storage, performing some calculation, translating to a different format, encapsulation with other data, etc. The behavior of each component is part of the architecture insofar as that behavior can be observed or discerned from the point of view of another component [Bass et al., 1998]. In other words, a component is defined by its interface and the services it provides to other components, rather than by its implementation behind the interface. Parnas [1971] would define this as the set of assumptions that other architectural elements can make about the component.

Connectors

> A **connector** is an abstract mechanism that mediates communication, coordination, or cooperation among components.

Perry and Wolf [1992] describe connecting elements vaguely as the glue that holds the various pieces of the architecture together. A more precise definition is provided by Shaw and Clements [1997]: A connector is an abstract mechanism that mediates communication, coordination, or cooperation among components. Examples include shared representations, remote procedure calls, message-passing protocols, and data streams.

Perhaps the best way to think about connectors is to contrast them with components. Connectors enable communication between components by transferring data elements from one interface to another without changing the data. Internally, a connector may consist of a subsystem of components that transform the data for transfer, perform the transfer, and then reverse the transformation for delivery. However, the external behavioral abstraction captured by the architecture ignores those details. In contrast, a component may, but not always will, transform data from the external perspective.

Data

> A **datum** is an element of information that is transferred from a component, or received by a component, via a connector.

The presence of data elements is the most significant distinction between the model of software architecture defined by Perry and Wolf [1992] and the model used by much of the research labelled software architecture. Boasson [1995] criticizes current software architecture research for its emphasis on component structures and architecture development tools, suggesting that more focus should be placed on data-centric architectural modeling. Similar comments are made by Jackson [1994].

A datum is an element of information that is transferred from a component, or received by a component, via a connector. Examples include byte-sequences, messages, marshalled parameters, and serialized objects, but do not include information that is permanently resident or hidden within a component. From the architectural perspective, a "file" is a transformation that a file system component might make from a "file name" datum received on its interface to a sequence of bytes recorded within an internally hidden storage system. Components can also generate data, as in the case of a software encapsulation of a clock or sensor.

The nature of the data elements within a network-based application architecture will often determine whether or not a given architectural style is appropriate. This is particularly evident in the comparison of mobile code design paradigms [Fuggetta et al., 1998], where the choice must be made between interacting with a component directly or transforming the component into a data element, transferring it across a network, and then transforming it back to a component that can be interacted with locally. It is impossible to evaluate such an architecture without considering data elements at the architectural level.

2.1.2 Configurations

> A **configuration** is the structure of architectural relationships among components, connectors, and data during a period of system run-time.

Abowd et al. [1995] define architectural description as supporting the description of systems in terms of three basic syntactic classes: components, which are the locus of computation; connectors, which define the interactions between components; and configurations, which are collections of interacting components and connectors. Various style-specific concrete notations may be used to represent these visually, facilitate the description of legal computations and interactions, and constrain the set of desirable systems.

Strictly speaking, one might think of a configuration as being equivalent to a set of specific constraints on component interaction. For example, Perry and Wolf [1992] include topology in their definition of architectural form relationships. However, separating the active topology from more general constraints allows an architect to more easily distinguish the active configuration from the potential domain of all legitimate configurations. Additional rationale for distinguishing configurations within architectural description languages is presented in Medvidovic and Taylor [1997].

2.1.3 Architectural Properties

The set of architectural properties of a software architecture includes all properties that derive from the selection and arrangement of components, connectors, and data within the system. Examples include both the functional properties achieved by the system and non-functional properties, such as relative ease of evolution, reusability of components, efficiency, and dynamic extensibility. The goal of architectural design is to create an architecture with a set of architectural properties that form a superset of the system requirements.

The relative importance of the various architectural properties depends on the nature of the intended system. Section 3.3 examines the properties that are of particular interest to network-based application architectures.

2.2 Architectural Styles

> An **architectural style** is a coordinated set of architectural constraints that restricts the roles/features of architectural elements and the allowed relationships among those elements within any architecture that conforms to that style.

Since an architecture embodies both functional and non-functional properties, it can be difficult to directly compare architectures for different types of systems, or for even the same type of system set in different environments. Styles are a mechanism for categorizing architectures and for defining their common characteristics [Di Nitto and Rosenblum, 1999]. An architectural style characterizes a family of systems that are related by shared structural and semantic properties [Monroe et al., 1997]. Each style provides an abstraction for the interactions of components, capturing the essence of a pattern of interaction by ignoring the incidental details of the rest of the architecture [Shaw, 1990].

Perry and Wolf [1992] define architectural style as an abstraction of element types and formal aspects from various specific architectures, perhaps concentrating on only certain aspects of an architecture. An architectural style encapsulates important decisions about the architectural elements and emphasizes important constraints on the elements and their relationships. This definition allows for styles that focus only on the connectors of an architecture, or on specific aspects of the component interfaces.

In contrast, Garlan and Shaw [1993], Garlan et al. [1994], and Shaw and Clements [1997] all define style in terms of a pattern of interactions among typed components. Specifically, an architectural style determines the vocabulary of components and connectors that can be used in instances of that style, together with a set of constraints on how they can be combined [Garlan and Shaw, 1993]. This restricted view of architectural styles is a direct result of their definition of software architecture — thinking of architecture as a formal description, rather than as a running system, leads to abstractions based only in the shared patterns of box and line diagrams. Abowd et al. [1995] go

further and define this explicitly as viewing the collection of conventions that are used to interpret a class of architectural descriptions as defining an architectural style.

New architectures can be defined as instances of specific styles [Di Nitto and Rosenblum, 1999]. Since architectural styles may address different aspects of software architecture, a given architecture may be composed of multiple styles. Likewise, a hybrid style can be formed by combining multiple basic styles into a single coordinated style.

Unfortunately, using the term style to refer to a coordinated set of constraints often leads to confusion. This usage differs substantially from the etymology of *style*, which would emphasize personalization of the design process. Loerke [1990] devotes a chapter to denigrating the notion that personal stylistic concerns have any place in the work of a professional architect. Instead, he describes styles as the critics' view of past architecture, where the available choice of materials, the community culture, or the ego of the local ruler were responsible for the architectural style, not the designer. In other words, Loerke views the real source of style in traditional building architecture to be the set of constraints applied to the design, and attaining or copying a specific style should be the least of the designer's goals.

2.3 Architectural Patterns and Pattern Languages

In parallel with the software engineering research in architectural styles, the object-oriented programming community has been exploring the use of design patterns and pattern languages to describe recurring abstractions in object-based software development. A design pattern is defined as an important and recurring system construct. A pattern language is a system of patterns organized in a structure that guides the patterns' application [Kerth and Cunningham, 1997]. Both concepts are based on the writings of Alexander et al. [1977, 1979] with regard to building architecture.

The design space of patterns includes implementation concerns specific to the techniques of object-oriented programming, such as class inheritance and interface composition, as well as the higher-level design issues addressed by architectural styles [Gamma et al., 1995]. In some cases, architectural style descriptions have been recast as architectural patterns [Shaw, 1996]. However, a primary benefit of patterns is that they can describe relatively complex protocols of interactions between objects as a single abstraction [Monroe et al., 1997], thus including both constraints on behavior and specifics of the implementation. In general, a pattern, or pattern language in the case of multiple integrated patterns, can be thought of as a recipe for implementing a desired set of interactions among objects. In other words, a pattern defines a process for solving a problem by following a path of design and implementation choices [Coplien, 1997].

Like software architectural styles, the software patterns research has deviated somewhat from its origin in building architecture. Indeed, Alexander's [1979] notion of patterns centers not on recurring arrangements of architectural elements, but rather on the recurring pattern of events—human activity and emotion—that take place within a space, with the understanding that a pattern of events cannot be separated from the space where it occurs. Alexander's design philosophy is to identify patterns of life that are common to the target culture and determine what architectural constraints are needed to differentiate a given space such that it enables the desired patterns to occur naturally. Such patterns exist at multiple levels of abstraction and at all scales.

> *As an element in the world, each pattern is a relationship between a certain context, a certain system of forces which occurs repeatedly in that context, and a certain spatial configuration which allows these forces to resolve themselves.*
>
> *As an element of language, a pattern is an instruction, which shows how this spatial configuration can be used, over and over again, to resolve the given system of forces, wherever the context makes it relevant.*
>
> *The pattern is, in short, at the same time a thing, which happens in the world, and the rule which tells us how to create that thing, and when we must create it. It is both a process and a thing; both a description of a thing which is alive, and a description of the process which will generate that thing. [Alexander, 1979]*

In many ways, Alexander's patterns have more in common with software architectural styles than the design patterns of OOPL research. An architectural style, as a coordinated set of constraints, is applied to a design space in order to induce the architectural properties that are desired of the system. By applying a style, an architect is differentiating the software design space in the hope that the result will better match the forces inherent in the application, thus leading to system behavior that enhances the natural pattern rather than conflicting with it.

2.4 Architectural Views

> *An architectural viewpoint is often application-specific and varies widely based on the application domain. ... we have seen architectural viewpoints that address a variety of issues, including: temporal issues, state and control approaches, data representation, transaction life cycle, security safeguards, and peak demand and graceful degradation. No doubt there are many more possible viewpoints. [Kerth and Cunningham, 1997]*

In addition to the many architectures within a system, and the many architectural styles from which the architectures are composed, it is also possible to view an architecture from many different perspectives. Perry and Wolf [1992] describe three important views in software architecture: processing, data, and connection views. A process view emphasizes the data flow through the components and some aspects of the connections among the components with respect to the data. A data view emphasizes the processing flow, with less emphasis on the connectors. A connection view emphasizes the relationship between components and the state of communication.

Multiple architectural views are common within case studies of specific architectures [Bass et al., 1998]. One architectural design methodology, the 4+1 View Model [Kruchten, 1995], organizes the description of a software architecture using five concurrent views, each of which addresses a specific set of concerns.

3 Network-based Application Architectures

3.1 Survey Scope

Architecture is found at multiple levels within software systems. This survey 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.

3.1.1 Network-based vs. Distributed

Tanenbaum and van Renesse [1985] 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 [Waldo et al., 1994]. This survey covers network-based systems by not limiting the candidate styles to those that preserve transparency for the user.

The primary distinction between network-based architectures and software architectures in general is that communication between components is restricted to message passing [Andrews, 1991], or the equivalent of message passing if a more efficient mechanism can be selected at run-time based on the location of components [Taylor et al., 1996].

3.1.2 Application Software vs. Networking Software

Another restriction on the scope of this survey 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 [Garlan and Shaw, 1993]). Applications represent the "business-aware" functionality of a system [Umar, 1997].

Application software architecture is an abstraction level of an overall system, in which the goals of a user action are representable as 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.

3.2 Evaluating the Design of Network-based Architectures

An architecture can be evaluated at many levels. Implementation. Protocol. Interactions. But functionality gets in the way. What we are interested in is a method of evaluating the design of a software architecture, such that the design can be replaced or improved over time. [UNFINISHED]

Architectural styles do not generally include application knowledge. Instead, we focus on the architectural properties that a style induces on a system. The specific needs of a network-based application can then be matched against the properties of a given style in order to determine the style's suitability for use within an architecture.

The set of architectural properties of a software architecture includes both functional properties and the equivalent of quality attributes [Bass et al., 1998]. A quality attribute can be induced by the constraints of an architectural style, usually motivated by applying a software engineering principle [Ghezzi et al., 1991] to an aspect of the architectural elements. For example, the *uniform pipe-and-filter* style obtains the qualities of reusability of components and configurability of the application by applying generality to its component interfaces — constraining the components to a single interface type. Hence, the architectural constraint is "uniform component interface," motivated by the

generality principle, in order to obtain two desirable qualities that will become the architectural properties of reusable and configurable components when that style is instantiated within an architecture.

3.3 Architectural Properties of Key Interest

This section describes the architectural properties used to differentiate and classify architectural styles for the survey. 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., [Ghezzi et al., 1991]). Bass et al. [1998] examine qualities in regards to software architecture.

3.3.1 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.

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.

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).

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.

3.3.2 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 will be distributed evenly over time or occur 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?).

3.3.3 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 the variability within an architecture. Generality of connectors leads to middleware (Section 6.3).

3.3.4 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 [Oreizy et al., 1998], 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.

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 [Waldo et al., 1994] can be used to support dynamic evolution.

Extensibility

Extensibility is defined as the ability to add functionality to a system [Pountain et al., 1995]. 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 (Section 5.5.1).

Customizability

Customizability is a specialization of extensibility in that it refers to modifying a component at run-time, specifically so that the component 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 [Fuggetta et al., 1998]. 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 (Section 5.4.2) and code-on-demand (Section 5.4.3) styles.

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 (Section 5.1.1) and code-on-demand (Section 5.4.3) styles are two examples that induce configurability of configurations and components, respectively.

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 (Section 5.1.2) exemplifies these types of constraints.

3.3.5 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 (Section 5.4.5) is an example where the lack of visibility may lead to security concerns.

This usage of visibility differs from that in Ghezzi et al. [1991], where they are referring to visibility into the development process rather than the product.

3.3.6 Portability

Software is portable if it can run in different environments [Ghezzi et al., 1991]. Styles that induce portability include those that move code along with the data to be processed, such as the virtual machine (Section 5.4.1) and mobile agent (Section 5.4.5) styles, and those that constrain the data elements to a set of standardized formats.

3.3.7 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.

4 Classification Methodology

The purpose of building software is not to create a specific topology of interactions or use a particular component type — it is to create a system that meets or exceeds the application needs. The style must conform to those needs, not the other way around. Therefore, in order to provide useful design guidance, a classification of architectural styles should be based on the architectural properties induced by those styles.

4.1 Selection of Architectural Styles for Classification

The set of architectural styles included in the classification is by no means comprehensive of all possible network-based application styles. Indeed, a new style can be formed merely by adding a new architectural constraint to any one of the styles surveyed. The goal is to describe a representative sample of styles, particularly those already identified within the software architecture literature, and provide a framework by which other styles can be added to the classification as they are developed.

I have intentionally excluded styles that do not enhance the communication or interaction properties when combined with one of the surveyed styles to form a network-based application. For example, the blackboard architectural style [Nii, 1986] consists of a central repository and a set of components (knowledge sources) that operate opportunistically upon the repository. A blackboard architecture can be extended to a network-based system by distributing the components, but the properties of such an extension are entirely based on the interaction style chosen to support the distribution — notifications via event-based integration, polling *a la* client-server, or replication of the repository. Thus, there would be no added value from including it in the classification even though the hybrid style is network-capable.

4.2 Style-induced Architectural Properties

The classification uses relative changes in the architectural properties induced by each style as a means of illustrating the effect of each architectural style when applied to a system for distributed hypermedia. Note that the evaluation of a style for a given property depends on the type of system interaction being studied, as described in Section 3.2. The architectural properties are relative in the sense that adding an architectural constraint may improve or reduce a given property, or simultaneously improve one aspect of the property and reduce some other aspect of the property. Likewise, improving one property may lead to the reduction of another.

Although our discussion of architectural styles will include those applicable to a wide range of network-based systems, our evaluation of each style will be based on its impact upon an architecture for a single type of software: network-based hypermedia systems. Focusing on a particular type of software allows us to identify the advantages of one style over another in the same way that a designer of a system would evaluate those advantages. Since we do not intend to declare any single style as being universally desir-

able for all types of software, restricting the focus of our evaluation simply reduces the dimensions over which we need to evaluate. Evaluating the same styles for other types of application software is an open area for future research.

4.3 Visualization

I use a table of style versus architectural properties as the primary visualization for this classification. The table values indicate the relative influence that the style for a given row has on a column's property. Minus (–) symbols accumulate for negative influences and plus (+) symbols for positive, with plus-minus (±) indicating that it depends on some aspect of the problem domain. Although this is a gross simplification of the details presented in each section, it does indicate the degree to which a style has addressed (or ignored) an architectural property.

An alternative visualization would be a property-based derivation graph for classifying architectural styles. The styles would be classified according to how they are derived from other styles, with the arcs between styles illustrated by architectural properties gained or lost. The starting point of the graph would be the null style (no constraints). It is possible to derive such a graph directly from the descriptions in Section 5.

5 Architectural Styles for Network-based Applications

5.1 Data-flow Styles

Style	Derivation	Net Perform.	UP Perform.	Efficiency	Scalability	Simplicity	Evolvability	Extensibility	Customiz.	Configur.	Reusability	Visibility	Portability	Reliability
PF			±			+	+	+		+	+			
UPF	PF	–	±			++	+	+		++	++	+		

5.1.1 Pipe and Filter (PF)

In a pipe and filter style, each component (filter) reads streams of data on its inputs and produces streams of data on its outputs, usually while applying a transformation to the input streams and computing incrementally so that output begins before the input is completely consumed [Garlan and Shaw, 1993]. This style is also referred to as a one-way data flow network [Andrews, 1991]. The constraint is that a filter must be completely independent of other filters (zero coupling): it must not share state, control thread, or identity with the other filters on its upstream and downstream interfaces [Garlan and Shaw, 1993].

Abowd et al. [1995] provide an extensive formal description of the pipe and filter style using the Z language. The *Khoros* software development environment for image processing [Rasure and Young, 1992] provides a good example of using the pipe and filter style to build applications.

Garlan and Shaw [1993] describe the advantageous properties of the pipe and filter style as follows. First, PF allows the designer to understand the overall input/output of the system as a simple composition of the behaviors of the individual filters (simplicity). Second, PF supports reuse: any two filters can be hooked together, provided they agree on the data that is being transmitted between them (reusability). Third, PF systems can

be easily maintained and enhanced: new filters can be added to existing systems (extensibility) and old filters can be replaced by improved ones (evolvability). Fourth, they permit certain kinds of specialized analysis (verifiability), such as throughput and deadlock analysis. Finally, they naturally support concurrent execution (user-perceived performance).

Disadvantages of the PF style include: propagation delay is added through long pipelines, batch sequential processing occurs if a filter cannot incrementally process its inputs, and no interactivity is allowed. A filter cannot interact with its environment because it cannot know that any particular output stream shares a controller with any particular input stream. These properties decrease user-perceived performance if the problem being addressed does not fit the pattern of a data flow stream.

One aspect of PF styles that is rarely mentioned is that there is an implied "invisible hand" that arranges the configuration of filters in order to establish the overall application. A network of filters is typically arranged just prior to each activation, allowing the application to specify the configuration of filter components based on the task at hand and the nature of the data streams (configurability). This controller function is considered a separate operational phase of the system, and hence a separate architecture, even though one cannot exist without the other.

5.1.2 Uniform Pipe and Filter (UPF)

The uniform pipe and filter style adds the constraint that all filters must have the same interface. The primary example of this style is found in the Unix operating system, where filter processes have an interface consisting of one input data stream of characters (stdin) and two output data streams of characters (stdout and stderr). Restricting the interface allows independently developed filters to be arranged at will to form new applications. It also simplifies the task of understanding how a given filter works.

A disadvantage of the uniform interface is that it may reduce network performance if the data needs to be converted to or from its natural format.

5.2 Replication Styles

Style	Derivation	Net Perform.	UP Perform.	Efficiency	Scalability	Simplicity	Evolvability	Extensibility	Customiz.	Configur.	Reusability	Visibility	Portability	Reliability
RR			++		+									+
$	RR		+	+	+	+								

5.2.1 Replicated Repository (RR)

Systems based on the replicated repository style [Andrews, 1991] improve the accessibility of data and scalability of services by having more than one process provide the same service. These decentralized servers interact to provide clients the illusion that there is just one, centralized service. Distributed filesystems, such as XMS [Fridrich and Older, 1985], and remote versioning systems, like CVS [www.cyclic.com], are the primary examples.

Improved user-perceived performance is the primary advantage, both by reducing the latency of normal requests and enabling disconnected operation in the face of primary server failure or intentional roaming off the network. Simplicity remains neutral, since

the complexity of replication is offset by the savings of allowing network-unaware components to operate transparently on locally replicated data. Maintaining consistency is the primary concern.

5.2.2 Cache ($)

A variant of replicated repository is found in the cache style: replication of the result of an individual request such that it may be reused by later requests. This form of replication is most often found in cases where the potential data set far exceeds the capacity of any one client, as in the WWW [Berners-Lee, 1996], or where complete access to the repository is unnecessary. Lazy replication occurs when data is replicated upon a not-yet-cached response for a request, relying on locality of reference and commonality of interest to propagate useful items into the cache for later reuse. Active replication can be performed by pre-fetching cachable entries based on anticipated requests.

Caching provides slightly less improvement than the replicated repository style in terms of user-perceived performance, since more requests will miss the cache and only recently accessed data will be available for disconnected operation. On the other hand, caching is much easier to implement, doesn't require as much processing and storage, and is more efficient because data is transmitted only when it is requested. The cache style becomes network-based when it is combined with a client-stateless-server style.

5.3 Hierarchical Styles

Style	Derivation	Net Perform.	UP Perform.	Efficiency	Scalability	Simplicity	Evolvability	Extensibility	Customiz.	Configur.	Reusability	Visibility	Portability	Reliability
CS					+	+	+							
LS			–		+		+				+		+	
LCS	CS+LS		–		++	+	++				+		+	
CSS	CS	–			++	+	+					+		+
C$SS	CSS+$	–	+	+	++	+	+					+		+
LC$SS	LCS+C$SS	–	±	+	+++	++	++				+	+	+	+
RS	CS			+	–	+	+					–		
RDA	CS			+	–	–						+		–

5.3.1 Client-Server (CS)

The client-server style is the most frequently encountered of the architectural styles for network-based applications. A server component, offering a set of services, listens for requests upon those services. A client component, desiring that a service be performed, sends a request to the server via a connector. The server either rejects or performs the request and sends a response back to the client. A variety of client-server systems are surveyed by Sinha [1992] and Umar [1997].

Andrews [1991] describes client-server components as follows: A client is a triggering process; a server is a reactive process. Clients make requests that trigger reactions from servers. Thus, a client initiates activity at times of its choosing; it often then delays until its request has been serviced. On the other hand, a server waits for requests to be made and then reacts to them. A server is usually a non-terminating process and often provides service to more than one client.

Separation of concerns is the principle behind the client-server constraints. A proper separation of functionality should simplify the server component in order to improve scalability. This simplification usually takes the form of moving all of the user interface functionality into the client component. The separation also allows the two types of components to evolve independently, provided that the interface doesn't change.

The basic form of client-server does not constrain how application state is partitioned between client and server components. It is often referred to by the mechanisms used for the connector implementation, such as remote procedure call [Birrell and Nelson, 1984] or message-oriented middleware [Umar, 1997].

5.3.2 Layered System (LS) and Layered-Client-Server (LCS)

A layered system is organized hierarchically, each layer providing services to the layer above it and using services of the layer below it [Garlan and Shaw, 1993]. Although layered system is considered a "pure" style, its use within network-based systems is limited to its combination with the client-server style to provide layered-client-server.

Layered systems reduce coupling across multiple layers by hiding the inner layers from all except the adjacent outer layer, thus improving evolvability and reusability. Examples include the processing of layered communication protocols, such as the TCP/IP and OSI protocol stacks [Zimmerman, 1980], and hardware interface libraries. The primary disadvantage of layered systems is that they add overhead and latency to the processing of data, reducing user-perceived performance [Clark and Tennenhouse, 1990].

Layered-client-server adds proxy and gateway components to the client-server style. A proxy [Shapiro, 1986] acts as a shared server for one or more client components, taking requests and forwarding them, with possible translation, to server components. A gateway component appears to be a normal server to clients or proxies that request its services, but is in fact forwarding those requests, with possible translation, to its "inner-layer" servers. These additional mediator components can be added in multiple layers to add features like load balancing and security checking to the system.

Architectures based on layered-client-server are referred to as two-tiered, three-tiered, or multi-tiered architectures in the information systems literature [Umar, 1997].

LCS is also a solution to managing identity in a large scale distributed system, where complete knowledge of all servers would be prohibitively expensive. Instead, servers are organized in layers such that rarely used services are handled by intermediaries rather than directly by each client [Andrews, 1991].

5.3.3 Client-Stateless-Server (CSS)

The client-stateless-server style derives from client-server with the additional constraint of no session state allowed on the server component. Each request from client to server must contain all of the information necessary to understand the request, and cannot take advantage of any stored context on the server. Application state is kept entirely on the client.

These constraints improve the properties of visibility, reliability, and scalability. Visibility is improved because a monitoring system does not have to look beyond a single request datum in order to determine the full nature of the request. Reliability is improved because it eases the task of recovering from partial failures [Waldo et al., 1994]. Scalability is improved because not having to store state between requests allows the server component to quickly free resources and further simplifies implementation.

The disadvantage of client-stateless-server is that it may decrease network performance by increasing the repetitive data (per-interaction overhead) sent in a series of requests, since that data cannot be left on the server in a shared context.

5.3.4 Client-Cache-Stateless-Server (C$SS)

The client-cache-stateless-server style derives from the client-stateless-server and cache styles via the addition of cache components. A cache acts as a mediator between client and server in which the responses to prior requests can, if they are considered cachable, be reused in response to later requests that are equivalent and likely to result in a response identical to that in the cache if the request were to be forwarded to the server. An example system that makes effective use of this style is Sun Microsystems' NFS [Sandberg et al., 1985].

The advantage of adding cache components is that they have the potential to partially or completely eliminate some interactions, improving efficiency and user-perceived performance.

5.3.5 Layered-Client-Cache-Stateless-Server (LC$SS)

The layered-client-cache-stateless-server style derives from both layered-client-server and client-cache-stateless-server through the addition of proxy and/or gateway components. Two examples of systems that use an LC$SS style are the Internet domain name system (DNS) and the World Wide Web's HTTP [Fielding et al., 1999].

The advantages and disadvantages of LC$SS are just a combination of those for LCS and C$SS. However, note that we don't count the contributions of the CS style twice, since the benefits are not additive if they come from the same ancestral derivation.

5.3.6 Remote Session (RS)

The remote session style is a variant of client-server that attempts to minimize the complexity, or maximize the reuse, of the client components rather than the server component. Each client initiates a session on the server and then invokes a series of services on the server, finally exiting the session. Application state is kept entirely on the server. This style is typically used when it is desired to access a remote service using a generic client (e.g., TELNET [Postel and Reynolds, 1983]) or via an interface that mimics a generic client (e.g., FTP [Postel and Reynolds, 1985]).

The advantages of the remote session style are that it is easier to centrally maintain the interface at the server, reducing concerns about inconsistencies in deployed clients when functionality is extended, and improves efficiency if the interactions make use of extended session context on the server. The disadvantages are that it reduces scalability of the server, due to the stored application state, and reduces visibility of interactions, since a monitor would have to know the complete state of the server.

5.3.7 Remote Data Access (RDA)

The remote data access style [Umar, 1997] is a variant of client-server that spreads the application state across both client and server. A client sends a database query in a standard format, such as SQL, to a remote server. The server allocates a workspace and performs the query, which may result in a very large data set. The client can then make further operations upon the result set (such as table joins) or retrieve the result one piece at a time. The client must know about the data structure of the service to build structure-dependent queries.

The advantages of remote data access are that a large data set can be iteratively reduced on the server side without transmitting it across the network, improving efficiency, and visibility is improved by using a standard query language. The disadvantages are that the client needs to understand the same database manipulation concepts as the server implementation (lacking simplicity) and storing application context on the server decreases scalability. Reliability also suffers, since partial failure can leave the workspace in an unknown state. Transaction mechanisms (e.g., two-phase commit) can be used to fix the reliability problem, though at a cost of added complexity and interaction overhead.

5.4 Mobile Code Styles

Mobile code styles use mobility in order to dynamically change the distance between the processing and source of data or destination of results. These styles are comprehensively examined in Fuggetta et al. [1998]. A site abstraction is introduced at the architectural level, as part of the active configuration, in order to take into account the location of the different components. Introducing the concept of location makes it possible to model the cost of an interaction between components at the design level. In particular, an interaction between components that share the same location is considered to have negligible cost when compared to an interaction involving communication through the network. By changing its location, a component may improve the proximity and quality of its interaction, reducing interaction costs and thereby improving efficiency and user-perceived performance.

In all of the mobile code styles, a data element is dynamically transformed into a component. Fuggetta et al. [1998] use an analysis that compares the code's size as a data element to the savings in normal data transfer in order to determine whether mobility is desirable for a given action. This would be impossible to model from an architectural standpoint if the definition of software architecture excludes data elements.

Style	Derivation	Net Perform.	UP Perform.	Efficiency	Scalability	Simplicity	Evolvability	Extensibility	Customiz.	Configur.	Reusability	Visibility	Portability	Reliability
VM						±		+				–	+	
REV	CS+VM			+	–	±		+	+			–	+	–
COD	CS+VM		+	+	+	±		+		+		–		
LCODC$SS	LC$SS+COD	–	++	++	+4+	+±+	++	+		+	+	±	+	+
MA	REV+COD		+	++		±		++	+	+		–	+	

5.4.1 Virtual Machine (VM)

Underlying all of the mobile code styles is the notion of a virtual machine, or interpreter, style [Garlan and Shaw, 1993]. The code must be executed in some fashion, preferably within a controlled environment to satisfy security and reliability concerns, which is exactly what the virtual machine style provides. It is not, in itself, a network-based style, but it is commonly used as such when combined with a component in the client-server style (REV and COD styles).

Scripting languages are the most common use of virtual machines, including general purpose languages like Perl [Wall et al., 1996] and task-specific languages like PostScript [Adobe, 1985]. The primary benefits are the separation between instruction and implementation on a particular platform (portability) and ease of extensibility. Visibility

is reduced because it is hard to know what an executable will do simply by looking at the code. Simplicity is reduced due to the need to manage the evaluation environment, but that may be compensated in some cases as a result of simplifying the static functionality.

5.4.2 Remote Evaluation (REV)

In the remote evaluation style [Fuggetta et al., 1998], derived from the client-server and virtual machine styles, a client component has the know-how necessary to perform a service, but lacks the resources (CPU cycles, data source, etc.) required, which happen to be located at a remote site. Consequently, the client sends the know-how to a server component at the remote site, which in turn executes the code using the resources available there. The results of that execution are then sent back to the client. The remote evaluation style assumes that the provided code will be executed in a sheltered environment, such that it won't impact other clients of the same server aside from the resources being used.

The advantages of remote evaluation include the ability to customize the server component's services, which provides for improved extensibility and customizability, and better efficiency when the code can adapt its actions to the environment inside the server (as opposed to the client making a series of interactions to do the same). Simplicity is reduced due to the need to manage the evaluation environment, but that may be compensated in some cases as a result of simplifying the static server functionality. Scalability is reduced; this can be improved with the server's management of the execution environment (killing long-running or resource-intensive code when resources are tight), but the management function itself leads to difficulties regarding partial failure and reliability. The most significant limitation, however, is the lack of visibility due to the client sending code instead of standardized queries. Lack of visibility leads to obvious deployment problems if the server cannot trust the clients.

5.4.3 Code on Demand (COD)

In the code-on-demand style [Fuggetta et al., 1998], a client component has access to a set of resources, but not the know-how on how to process them. It sends a request to a remote server for the code representing that know-how, receives that code, and executes it locally.

The advantages of code-on-demand include the ability to add features to a deployed client, which provides for improved extensibility and configurability, and better user-perceived performance and efficiency when the code can adapt its actions to the client's environment and interact with the user locally rather than through remote interactions. Simplicity is reduced due to the need to manage the evaluation environment, but that may be compensated in some cases as a result of simplifying the client's static functionality. Scalability of the server is improved, since it can off-load work to the client that would otherwise have consumed its resources. Like remote evaluation, the most significant limitation is the lack of visibility due to the server sending code instead of simple data. Lack of visibility leads to obvious deployment problems if the client cannot trust the servers.

5.4.4 Layered-Code-on-Demand-Client-Cache-Stateless-Server (LCODC$SS)

As an example of how some architectures are complementary, consider the addition of code-on-demand to the layered-client-cache-stateless-server style discussed above. Since the code can be treated as just another data element, this does not interfere with the advantages of the LC$SS style. An example is the HotJava Web browser

[java.sun.com], which allows applets and protocol extensions to be downloaded as typed media.

The advantages and disadvantages of LCODC$SS are just a combination of those for COD and LC$SS. We could go further and discuss the combination of COD with other CS styles, but this survey is not intended to be exhaustive (nor exhausting).

5.4.5 Mobile Agent (MA)

In the mobile agent style [Fuggetta et al., 1998], an entire computational component is moved to a remote site, along with its state, the code it needs, and possibly some data required to perform the task. This can be considered a derivation of the remote evaluation and code-on-demand styles, since the mobility works both ways.

The primary advantage of the mobile agent style, beyond those already described for REV and COD, is that there is greater dynamism in the selection of when to move the code. An application can be in the midst of processing information at one location when it decides to move to another location, presumably in order to reduce the distance between it and the next set of data it wishes to process. In addition, the reliability problem of partial failure is reduced because the application state is in one location at a time [Fuggetta et al., 1998].

5.5 Peer-to-Peer Styles

Style	Derivation	Net Perform.	UP Perform.	Efficiency	Scalability	Simplicity	Evolvability	Extensibility	Customiz.	Configur.	Reusability	Visibility	Portability	Reliability
EBI				+	– –	±	+	+		+	+	–		–
C2	EBI+LCS		–	+		+	++	+		+	++	±	+	±
DO	CS+CS	–		+			+	+		+	+	–		–
BDO	DO+LCS	–	–				++	+		+	++	–	+	

5.5.1 Event-based Integration (EBI)

The event-based integration style, also known as the implicit invocation or event system style, reduces coupling between components by removing the need for identity on the connector interface. Instead of invoking another component directly, a component can announce (or broadcast) one or more events. Other components in a system can register interest in that type of event and, when the event is announced, the system itself invokes all of the registered components [Garlan and Shaw, 1993]. Examples include the Model-View-Controller paradigm in Smalltalk-80 [Krasner and Pope, 1988] and the integration mechanisms of many software engineering environments, including Field [Reiss, 1990], SoftBench [Cagan, 1990], and Polylith [Purtilo, 1994].

The event-based integration style provides strong support for extensibility through the ease of adding new components that listen for events, for reuse by encouraging a general event interface and integration mechanism, and for evolution by allowing components to be replaced without affecting the interfaces of other components [Garlan and Shaw, 1993]. Like pipe-and-filter systems, a higher-level configuring architecture is needed for the “invisible hand” that places components on the event interface. Most EBI systems also include explicit invocation as a complementary form of interaction [Garlan and Shaw, 1993]. For applications that are dominated by data monitoring, rather than data retrieval, EBI can improve efficiency by removing the need for polling interactions.

The basic form of EBI system consists of one event bus to which all components listen for events of interest to them. Of course, this immediately leads to scalability issues with regard to the number of notifications, event storms as other components broadcast as a result of events caused by that notification, and a single point of failure in the notification delivery system. This can be ameliorated though the use of layered systems and filtering of events, at the cost of simplicity.

Other disadvantages of EBI systems are that it can be hard to anticipate what will happen in response to an action (poor understandability) and event notifications are not suitable for exchanging large-grain data [Garlan and Shaw, 1993]. Also, there is no support for recovery from partial failure.

5.5.2 C2

The C2 architectural style [Taylor et al., 1996] is directed at supporting large-grain reuse and flexible composition of system components by enforcing substrate independence. It does so by combining event-based integration with layered-client-server. Asynchronous notification messages going down, and asynchronous request messages going up, are the sole means of intercomponent communication. This enforces loose coupling of dependency on higher layers (service requests may be ignored) and zero coupling with lower levels (no knowledge of notification usage), improving control over the system without losing most of the advantages of EBI.

Notifications are announcements of a state change within a component. C2 does not constrain what should be included with a notification: a flag, a delta of state change, or a complete state representation are all possibilities. A connector's primary responsibility is the routing and broadcasting of messages; its secondary responsibility is message filtering. The introduction of layered filtering of messages solves the EBI problems with scalability, while improving evolvability and reusability as well. Heavyweight connectors that include monitoring capabilities can be used to improve visibility and reduce the reliability problems of partial failure.

5.5.3 Distributed Objects

The distributed objects style organizes a system as a set of components interacting as peers. An object is an entity that encapsulates some private state information or data, a set of associated operations or procedures that manipulate the data, and possibly thread of control so that collectively they can be considered a single unit [Chin and Chanson, 1991]. In general, an object's state is completely hidden and protected from all other objects. The only way it can be examined or modified is by making a request or invocation on one of the object's publicly accessible operations. This creates a well-defined interface for each object, enabling the specification of an object's operations to be made public while at the same time keeping the implementation of its operations and the representation of its state information private, thus improving evolvability.

An operation may invoke other operations, possibly on other objects. These operations may in turn make invocations on others, and so on. A chain of related invocations is referred to as an action [Chin and Chanson, 1991]. State is distributed among the objects. This can be advantageous in terms of keeping the state where it is most likely to be up-to-date, but has the disadvantage in that it is difficult to obtain an overall view of system activity (poor visibility).

In order for one object to interact with another, it must know the identity of that other object. When the identity of an object changes, it is necessary to modify all other objects that explicitly invoke it [Garlan and Shaw, 1993]. There must be some controller object that is responsible for maintaining the system state in order to complete the application

requirements. Central issues for distributed object systems include: object management, object interaction management, and resource management [Chin and Chanson, 1991].

Object systems are designed to isolate the data being processed. As a consequence, data streaming is not supported in general. However, this does provide better support for object mobility when combined with the mobile agent style.

5.5.4 Brokered Distributed Objects

In order to reduce the impact of identity, modern distributed object systems typically use one or more intermediary styles to facilitate communication. This includes event-based integration and brokered client/server [Buschmann et al., 1996]. The brokered distributed object style introduces name resolver components whose purpose is to answer client object requests for general service names with the specific name of an object that will satisfy the request. Although improving reusability and evolvability, the extra level of indirection requires additional network interactions, reducing efficiency and user-perceived performance.

Brokered distributed object systems are currently dominated by the industrial standards development of CORBA within the OMG [1997] and the international standards development of Open Distributed Processing (ODP) within ISO/IEC [1995].

In spite of all the interest associated with distributed objects, they fare poorly when compared to most other network-based architectural styles. They are best used for applications that involve the remote invocation of encapsulated services, such as hardware devices, where the efficiency and frequency of network interactions is less a concern.

6 Related Work

I include here only those areas of research that describe, define, or embody software architectural styles. Other areas for software architectural research include architectural analysis techniques, architecture recovery and reengineering, tools and environments for architectural design, architecture refinement from specification to implementation, and case studies of deployed software architectures [Garlan and Perry, 1995].

6.1 Classification of Architectural Styles and Patterns

The area of research most directly related to this survey is the identification and classification of architectural styles and architecture-level patterns.

Shaw [1990] describes a few architectural styles, later expanded in Garlan and Shaw [1993]. A preliminary classification of these styles is presented in Shaw and Clements [1997] and repeated in Bass et al. [1998], in which a two-dimensional, tabular classification strategy is used with control and data issues as the primary axes, organized by the following categories of features: which kinds of components and connectors are used in the style; how control is shared, allocated, and transferred among the components; how data is communicated through the system; how data and control interact; and, what type of reasoning is compatible with the style. The primary purpose of the taxonomy is to identify style characteristics, rather than to assist in their comparison. It concludes with a small set of “rules of thumb” as a form of design guidance

Unlike this survey, the Shaw and Clements [1997] classification does not assist in evaluating designs in a way that is useful to an application designer. The problem is that the purpose of building software is not to build a specific shape, topology or component type, so organizing the classification in that fashion does not help a designer find a style

that corresponds to their needs. It also mixes the essential differences among styles with other issues which have only incidental significance, and obscures the derivation relationships among styles. Furthermore, it does not focus on any particular type of architecture, such as network-based applications. Finally, it does not describe how styles can be combined, nor the effect of their combination.

Buschmann and Meunier [1995] describe a classification scheme that organizes patterns according to granularity of abstraction, functionality, and structural principles. The granularity of abstraction separates patterns into three categories: architectural frameworks (templates for architectures), design patterns, and idioms. Their classification addresses some of the same issues as this survey, such as separation of concerns and structural principles that lead to architectural properties, but only covers two of the architectural styles described here. Their classification is considerably expanded in Buschmann et al. [1996] with more extensive discussion of architectural patterns and their relation to software architecture.

Zimmer [1995] organizes design patterns using a graph based on their relationships, making it easier to understand the overall structure of the patterns in the Gamma et al. [1995] catalog. However, the patterns classified are not architectural patterns, and the classification is based exclusively on derivation or uses relationships rather than on architectural properties.

6.2 Distributed Systems and Programming Paradigms

Andrews [1991] surveys how processes in a distributed program interact via message passing. He defines concurrent programs, distributed programs, kinds of processes in a distributed program (filters, clients, servers, peers), interaction paradigms, and communication channels. Interaction paradigms represent the communication aspects of software architectural styles. He describes paradigms for one-way data flow through networks of filters (pipe-and-filter), client-server, heartbeat, probe/echo, broadcast, token passing, replicated servers, and replicated workers with bag of tasks. However, the presentation is from the perspective of multiple processes cooperating on a single task, rather than general network-based architectural styles.

Sullivan and Notkin [1992] provide a survey of implicit invocation research and describe its application to improving the evolution quality of software tool suites. Barrett et al. [1996] present a survey of event-based integration mechanisms by building a framework for comparison and then seeing how some systems fit within that framework. Rosenblum and Wolf [1997] investigate a design framework for Internet-scale event notification. All are concerned with the scope and requirements of an EBI style, rather than providing solutions for network-based systems.

Fuggetta et al. [1998] provide a thorough examination and classification of mobile code paradigms. This survey builds upon their work to the extent that I compare the mobile code styles with other network-capable styles, and place them within a single framework and set of architectural definitions.

6.3 Middleware

Bernstein [1996] defines middleware as a distributed system service that includes standard programming interfaces and protocols. These services are called middleware because they act as a layer above the OS and networking software and below industry-specific applications. Umar [1997] presents an extensive treatment of the subject.

Architecture research regarding middleware focuses on the problems and effects of integrating components with off-the-shelf middleware. Di Nitto and Rosenblum [1999] describe how the usage of middleware and predefined components can influence the architecture of a system being developed and, conversely, how specific architectural choices can constrain the selection of middleware. Dashofy et al. [1999] discuss the use of middleware with the C2 style.

Garlan et al. [1995] point out some of the architectural assumptions within off-the-shelf components, examining the authors' problems with reusing subsystems in creating the Aesop tool for architectural design [Garlan et al., 1994]. They classify the problems into four main categories of assumptions that can contribute to architectural mismatch: nature of components, nature of connectors, global architectural structure, and construction process.

6.4 Design Methodologies

Most early research on software architecture was concentrated on design methodologies. For example, object-oriented design [Booch, 1986] advocates a way to structure problems that leads naturally to an object-based architecture (or, more accurately, does not lead naturally to any other form of architecture). One of the first design methodologies to emphasize design at the architectural level is Jackson System Development [Cameron, 1986]. JSD intentionally structures the analysis of a problem so that it leads to a style of architecture that combines pipe-and-filter (data flow) and process control constraints. These design methodologies tend to produce only one style of architecture.

There has been some initial work investigating methodologies for the analysis and development of architectures. Kazman et al. have described design methods for eliciting the architectural aspects of a design through scenario-based analysis with SAAM [1994] and architectural trade-off analysis viaATAM [1999]. Shaw [1995] compares a variety of box-and-arrow designs for an automobile cruise control system, each done using a different design methodology and encompassing several architectural styles.

6.5 Handbooks for Design, Design Patterns, and Pattern Languages

Shaw [1990] advocates the development of architectural handbooks along the same lines as traditional engineering disciplines. As discussed in Section 2.3, the object-oriented programming community has taken the lead in producing catalogs of design patterns, as exemplified by the "Gang of Four" book [Gamma et al., 1995] and the essays edited by Coplien and Schmidt [1995].

Software design patterns tend to be more problem-oriented than architectural styles. Shaw [1996] presents eight example architectural patterns based on the architectural styles described in [Garlan and Shaw, 1993], including information on the kinds of problems best suited to each architecture. Buschmann et al. [1996] provide a comprehensive examination of the architectural patterns common to object-based development. Both references are purely descriptive and make no attempt to compare or illustrate the differences among architectural patterns.

Tepfenhart and Cusick [1997] use a two dimensional map to differentiate among domain taxonomies, domain models, architectural styles, frameworks, kits, design patterns, and applications. In the topology, design patterns are predefined design structures used as building blocks for a software architecture, whereas architectural styles are sets of operational characteristics that identify an architectural family independent of application domain. However, they fail to define architecture itself.

6.6 Reference Models and Domain-specific Software Architectures (DSSA)

Reference models are developed to provide conceptual frameworks for describing architectures and showing how components are related to each other [Shaw, 1990]. The Object Management Architecture (OMA), developed by the OMG [1995] as a reference model for brokered distributed object architectures, specifies how objects are defined and created, how client applications invoke objects, and how objects can be shared and reused. The emphasis is on management of distributed objects, rather than efficient application interaction.

Hayes-Roth et al. [1995] define domain-specific software architecture (DSSA) as comprising: a) a reference architecture, which describes a general computational framework for a significant domain of applications, b) a component library, which contains reusable chunks of domain expertise, and c) an application configuration method for selecting and configuring components within the architecture to meet particular application requirements. Tracz [1995] provides a general overview of DSSA.

DSSA projects have been successful at transferring architectural decisions to running systems by restricting the software development space to a specific architectural style that matches the domain requirements [Medvidovic et al., 1999]. Examples include ADAGE [Batory et al., 1995] for avionics, AIS [Hayes-Roth et al., 1995] for adaptive intelligent systems, and MetaH [Vestal, 1996] for missile guidance, navigation, and control systems. DSSA emphasize reuse of components within a common architectural domain, rather than selecting an architectural style that is specific to each system.

6.7 Architecture Description Languages (ADL)

Most of the recent published work regarding software architectures is in the area of architecture description languages (ADL). An ADL is, according to Medvidovic and Taylor [1997], a language that provides features for the explicit specification and modeling of a software system's conceptual architecture, including at a minimum: components, component interfaces, connectors, and architectural configurations.

Darwin is a declarative language which is intended to be a general purpose notation for specifying the structure of systems composed of diverse components using diverse interaction mechanisms [Magee et al., 1995]. Darwin's interesting qualities are that it allows the specification of distributed architectures and dynamically composed architectures [Magee and Kramer, 1996].

UniCon [Shaw et al., 1995] is a language and associated toolset for composing an architecture from a restricted set of component and connector examples. Wright [Allen and Garlan, 1997] provides a formal basis for specifying the interactions between architectural components by specifying connector types by their interaction protocols.

Like design methodologies, ADLs often introduce specific architectural assumptions that may impact their ability to describe some architectural styles, and may conflict with the assumptions in existing middleware [Di Nitto and Rosenblum, 1999]. In some cases, an ADL is designed specifically for a single architectural style, thus improving its capacity for specialized description and analysis at the cost of generality. For example, C2SADEL [Medvidovic et al., 1999] is an ADL designed specifically to describe architectures developed in the C2 style [Taylor et al., 1996]. In contrast, ACME [Garlan et al., 1997] is an ADL that attempts to be as generic as possible, but with the trade-off being that it doesn't support style-specific analysis.

6.8 Formal Architectural Models

Abowd et al. [1995] claim that architectural styles can be described formally in terms of a small set of mappings from the syntactic domain of architectural descriptions (box-and-line diagrams) to the semantic domain of architectural meaning. However, this assumes that the architecture is the description, rather than an abstraction of a running system.

Inverardi and Wolf [1995] use the Chemical Abstract Machine (CHAM) formalism to model software architecture elements as chemicals whose reactions are controlled by explicitly stated rules. It specifies the behavior of components according to how they transform available data elements and uses composition rules to propagate the individual transformations into an overall system result.While this is an interesting model, it is unclear as to how CHAM could be used to describe any form of architecture whose purpose goes beyond transforming a data stream.

Rapide [Luckham and Vera, 1995] is a concurrent, event-based simulation language specifically designed for defining and simulating system architectures. The simulator produces a partially-ordered set of events that can be analyzed for conformance to the architectural constraints on interconnection. Le Métayer [1998] presents a formalism for the definition of architectures in terms of graphs and graph grammars.

7 Conclusions

Each architectural style promotes a certain type of interaction among components. When components are distributed across a wide-area network, use or misuse of the network drives application usability. By characterizing styles by their influence on architectural properties, and particularly on network-based application performance, we gain the ability to better choose a software design that is appropriate for the application.

The primary contributions of this survey are the development of a comprehensive set of definitions for software architecture and architectural styles, identification of the architectural properties that are most influenced by the constraints of each style, and a classification of styles according to those properties.

There are, however, a couple limitations with the chosen classification. First, the evaluation of all network-based styles against the generic notion of network communication, rather than a specific type of communication, leads to difficulty in accurate comparisons between styles that have different properties depending on the type of communication. For example, many of the good qualities of the pipe-and-filter style disappear if the communication is fine-grained control messages, and are not applicable at all if the communication requires user interactivity. Likewise, layered caching only adds to latency, without any benefit, if none of the responses to client requests are cachable. This type of distinction does not appear in the classification, and is only addressed informally in the discussion of each style. I believe this limitation can be overcome by creating separate classification tables for each type of communication problem. Example problem areas would include, among others, large grain data retrieval, remote information monitoring, search, remote control systems, and distributed processing.

A second limitation is with the grouping of architectural properties. In some cases, it is better to identify the specific aspects of, for example, understandability and verifiability induced by an architectural style, rather than lumping them together under the rubric of simplicity. This is particularly the case for styles which might improve verifiability at the expense of understandability. However, the more abstract notion of a property also

has value as a single metric, since we do not want to make the classification so specific that no two styles impact the same category. One solution would be a classification that presented both the specific properties and a summary property.

Regardless, this initial survey and classification is a necessary prerequisite to any further classifications that might address its limitations.

8 Acknowledgments

I'd like to thank Nenad Medvidovic for providing me with a copy of some hard-to-obtain papers and pointing me in the direction of others. Peyman Oreizy's survey [Oreizy, 1998] provided a useful outline for structuring this survey. This version has benefited from comments on earlier drafts by Richard N. Taylor, Mark S. Ackerman, and David S. Rosenblum.

9 References

[Abowd et al., 1995] 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.

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

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

[Alexander, 1979] C. Alexander. The Timeless Way of Building. Oxford University Press, New York, 1979.

[Allen and Garlan, 1997] 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.

[Andrews, 1991] G. Andrews. Paradigms for process interaction in distributed programs. ACM Computing Surveys, 23(1), Mar. 1991, pp. 49-90.

[Barrett et al., 1996] 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.

[Bass et al., 1998] L. Bass, P. Clements, and R. Kazman. Software Architecture in Practice. Addison Wesley, Reading, Mass., 1998.

[Batory et al., 1995] 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.

[Berners-Lee, 1996] T. Berners-Lee. WWW: Past, present, and future. IEEE Computer, 29(10), October 1996, pp. 69-77.

[Berners-Lee et al., 1998] T. Berners-Lee, R. T. Fielding, and L. Masinter. Uniform Resource Identifiers (URI): Generic Syntax. Internet RFC 2396, Aug. 1998.

[Bernstein, 1996] P. Bernstein. Middleware: A model for distributed systems services. CACM, Feb. 1996, pp. 86-98.

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

[Boasson, 1995] M. Boasson. The artistry of software architecture. IEEE Software, 12(6), Nov. 1995, pp. 13-16.

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

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

[Buschmann et al., 1996] 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.

[Cagan, 1990] 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.

[Cameron, 1986] J. R. Cameron. An overview of JSD. IEEE Transactions on Software Engineering, 12(2), Feb. 1986, 222-240.

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

[Clark and Tennenhouse, 1990] 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.

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

[Coplien, 1997] J. O. Coplien. Idioms and Patterns as Architectural Literature. IEEE Software, 14(1), Jan. 1997, pp. 36-42.

[Dashofy et al., 1999] 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.

[DeRemer and Kron, 1976] 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.

[Di Nitto and Rosenblum, 1999] 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.

[Fielding et al., 1999] 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.]

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

[Fuggetta et al., 1998] A. Fuggetta, G. P. Picco, and G. Vigna. Understanding code mobility. IEEE Transactions on Software Engineering, 24(5), May 1998, pp. 342-361.

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

[Garlan and Ilias, 1990] 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.

[Garlan and Shaw, 1993] 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.

[Garlan et al., 1994] 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.

[Garlan and Perry, 1995] 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.

[Garlan et al., 1995] 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.

[Garlan et al., 1997] D. Garlan, R. Monroe, and D. Wile. ACME: An architecture description language. In Proceedings of CASCON'97, Nov. 1997.

[Ghezzi et al., 1991] C. Ghezzi, M. Jazayeri, and D. Mandrioli. Fundamentals of Software Engineering. Prentice-Hall, 1991.

[Hayes-Roth et al., 1995] 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.

[Inverardi and Wolf, 1995] 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.

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

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

[Kazman et al., 1994] 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.

[Kazman et al., 1999] 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.

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

[Krasner and Pope, 1988] 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.

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

[Le Métayer, 1998] D. Le Métayer. Describing software architectural styles using graph grammars. IEEE Transactions on Software Engineering, 24(7), Jul. 1998, pp. 521-533.

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

[Luckham et al., 1995] 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.

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

[Maes, 1987] P. Maes. Concepts and experiments in computational reflection. In Proceedings of OOPSLA '87, Orlando, Florida, Oct. 1987, pp. 147-155.

[Magee et al., 1995] 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.

[Magee and Kramer, 1996] 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.

[Medvidovic and Taylor., 1997] 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.

[Medvidovic, 1998] Architecture-based specification-time software evolution. Ph.D. Dissertation, University of California, Irvine, Dec. 1998.

[Medvidovic et al., 1999] 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.

[Monroe et al, 1997] 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.

[Moriconi et al., 1995] M. Moriconi, X. Qian, and R. A. Riemenscheider. Correct architecture refinement. IEEE Transactions on Software Engineering, 21(4), April 1995, pp. 356-372.

[Nii, 1986] H. Penny Nii. Blackboard systems. AI Magazine, 7(3):38-53 and 7(4):82-107, 1986.

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

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

[Oreizy et al., 1998] 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.

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

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

[Parnas, 1972] 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.

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

[Parnas et al., 1985] 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.

[Perry and Wolf, 1992] 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.

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

[Postel and Reynolds, 1985] J. Postel and J. Reynolds. File Transfer Protocol. *Internet STD 9, RFC 959,* October 1985.

[Pountain et al., 1994] D. Pountain and C. Szyperski. Extensible software systems. Byte, May 1994, pp. 57-62.

[Prieto-Diaz et al., 1986] R. Prieto-Diaz and J. M. Neighbors. Module interconnection languages. Journal of Systems and Software, 6(4), Nov. 1986, pp. 307-334.

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

[Python, 1970] 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>.

[Rasure and Young, 1992] 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.

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

[Rosenblum and Wolf, 1997] 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.

[Sandberg et al., 1985] 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.

[Shapiro, 1986] 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.

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

[Shaw et al., 1995] 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.

[Shaw, 1995] M. Shaw. Comparing architectural design styles. IEEE Software, 12(6), Nov. 1995, pp. 27-41.

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

[Shaw and Garlan, 1996] M. Shaw and D. Garlan. Software Architecture: Perspectives on an emerging discipline. Prentice-Hall, 1996.

[Shaw and Clements, 1997] 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.

[Sinha, 1992] A. Sinha. Client-server computing. CACM, July 1992, pp. 77-98.

[Sullivan and Notkin, 1992] 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.

[Tanenbaum et al.., 1985] A. S. Tanenbaum and R. van Renesse. Distributed Operating Systems. ACM Computing Surveys, 17(4), Dec. 1985, pp. 419-470.

[Taylor et al., 1996] 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.

[Tepfenhart and Cusick, 1997] W. Tephenhart and J. J. Cusick. A Unified ObjectTopology. IEEE Software, 14(1), Jan. 1997, pp. 31-35.

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

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

[Vestal, 1996] S. Vestal. MetaH programmer's manual, version 1.09. Technical Report, Honeywell Technology Center, April 1996.

[Waldo et al., 1994] 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.

[Wall et al., 1996] L. Wall, T. Christiansen, and R. L. Schwartz. Programming Perl, 2nd ed. O'Reilly & Associates, 1996.

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

[Zimmerman, 1980] H. Zimmerman. OSI reference model --- The ISO model of architecture for open systems interconnection. IEEE Transactions on Communications, 28, Apr. 1980, pp. 425-432.

10 Summary of Architectural Style Classification

Style	Derivation	Net Perform.	UP Perform.	Efficiency	Scalability	Simplicity	Evolvability	Extensibility	Customiz.	Configur.	Reusability	Visibility	Portability	Reliability
PF			±			+	+	+		+	+			
UPF	PF	–	±			++	+	+		++	++	+		
CS					+	+	+							
LS			–		+		+				+		+	
LCS	CS+LS		–		++	+	++				+		+	
CSS	CS	–			++	+	+					+		+
C$SS	CSS+$	–	+	+	++	+	+					+		+
LC$SS	LCS+C$SS	–	±	+	+++	++	++				+	+	+	+
RS	CS			+	–	+	+					–		
RDA	CS			+	–	–						+		–
VM						±		+				–	+	
REV	CS+VM			+	–	±		+	+			–	+	–
COD	CS+VM		+	+	+	±		+		+		–		
LCODC$SS	LC$SS+COD	–	++	++	+4+	+±+	++	+		+	+	±	+	+
MA	REV+COD		+	++		±		++	+	+		–	+	
RR			++		+									+
$	RR		+	+	+	+								
EBI				+	– –	±	+	+		+	+	–		–
C2	EBI+LCS		–	+		+	++	+		+	++	±	+	±
DO	CS+CS	–		+			+	+		+	+	–		–
BDO	DO+LCS	–	–				++	+		+	++	–	+	