

Understanding Software Connector Compatibilities Using A Connector Taxonomy

Nikunj R. Mehta and Nenad Medvidovic

Computer Science Department
University of Southern California
Los Angeles, CA 90089-0781, USA
{mehta, neno@usc.edu}

ABSTRACT

Software systems of today are frequently composed from prefabricated, heterogeneous components that provide complex functionality and engage in complex interactions. Software architecture research has revealed the importance of component interactions on our ability to perform software composition. Various development organizations have found interaction mismatches to be both difficult and a core issue behind failed software integration efforts. This paper is aimed at improving the understanding of component interactions, embodied in the notion of software connectors, in order to prevent such failures. Our previous work has resulted in a four-level classification framework for studying the characteristics of connectors. This paper presents a comprehensive taxonomy of software connectors based on the previously published connector classification framework. This taxonomy is used to explain various existing software connectors and study compatibility of connector characteristics, intended to prevent component interaction mismatches.

1 INTRODUCTION

A number of techniques have recently emerged to address the problem of consistently engineering large, complex software systems. The three most widely embraced efforts have been component-based software development standards (e.g., [21]), middleware platforms (e.g., [15,18]), and software architecture [17,20]. They present complementary, often overlapping approaches, centered around composing software systems from coarse-grained *components*.

Although components have been the predominant focus of researchers and practitioners, they address only one aspect of large-scale development. Another important aspect, particularly magnified by the emergence of the Internet and the growing need for distribution, is *interaction* among components. Component interaction is embodied in the notion of *software connectors*. Connectors manifest themselves in a software system as shared variable accesses, table entries, buffers, instructions to

a linker, procedure calls, networking protocols, pipes, SQL links between a database and an application, and so forth [20]. In large, and especially distributed systems, connectors become key determinants of system properties, such as performance, resource utilization, global rates of flow, scalability, reliability, security, evolvability, and so forth. Studies at various research organizations such as [5,6] have found component interaction mismatches to be a core issue underlying failure of large-scale software projects.

Software architecture-based approaches have come furthest in their treatment of connectors. They typically separate computation (components) from interaction (connectors) in a system. In principle, architectures do not assume component homogeneity, nor do they constrain the allowed connectors and connector implementation mechanisms. Several existing architecture-based technologies have provided support for modeling or implementing certain classes of connectors [2,19,22].

Despite this, the current level of understanding and support for connectors in architectures is insufficient. Most architectural approaches that have explicitly addressed connectors have either provided mechanisms for *modeling* arbitrarily complex connectors or *implementing* a small set of simple ones, but never both. Our previous work on classifying software connectors has helped further our understanding of what the fundamental building blocks of software interaction are and how they can be composed into more complex interactions [12]. Such understanding is necessary to guide software development and prevent expensive interaction mismatches.

This paper presents a continuation of that study. We provide a classification of interaction mechanisms employed in software systems. These mechanisms have been identified from an extensive study of software interactions mechanisms used in computer systems research as well as commercial software systems. The classification supports deeper understanding of existing connectors and their relationships. Further, the paper describes results of a connector compatibility analysis that helps to avoid combinations of connector characteristics that may

result in component interaction mismatches.

The remainder of the paper is organized as follows. Section 2 summarizes the connector taxonomy, while Section 3 provides a discussion of the compatibility between connectors based on the taxonomy. The paper concludes with a summary and a brief overview of future work.

2 CLASSIFYING CONNECTORS

Our connector classification framework is based on the following definition of connectors [20]:

Connectors mediate interactions among components; that is, they establish the rules that govern component interaction and specify any auxiliary mechanisms required.

Our previous work [12] discussed the overall structure of the classification framework used for the taxonomy of software connectors presented in this paper. Connector *types* discriminate among connectors based on the way in which four interaction *services*, namely *communication*, *coordination*, *conversion*, and *facilitation* are realized. Connector types are the level at which architects typically consider interactions when modeling systems. Accordingly, connector types are classified as *procedure call*, *event*, *data access*, *linkage*, *stream*, *arbitrator*, *adaptor*, and *distributor*. The architecturally relevant details of each connector type are captured through *dimensions*, and, possibly, further *sub-dimensions* as shown in Figure 1. Finally, the lowest layer in the framework is formed by the set of *values* a dimension (or sub-dimension) can take. Figure 1 depicts the complete connector taxonomy based on the four-layer classification framework. It is possible in principle to combine dimensions in an arbitrary fashion, although all combinations are not necessarily meaningful or correct. Section 3 identifies such incompatible combinations between connector dimensions.

2.1 Procedure Call

Procedure call (PC) connectors model the flow of control among components through various *invocation* techniques (*coordination*) and perform transfer of data among the interacting components through the use of *parameters* (*communication*). They are characterized by several dimensions. PC *entry points* signify the provision of logic required to process the call. *Blocking* semantics of PCs determine the rules of simultaneous calling and processing of a procedure. *Accessibility* of PC is at times controlled for the purposes of encapsulation. A further encapsulation measure is the use of a high *fan-in* or *fan-out* where fan-in refers to the number of procedures that call this procedure, and fan-out refers to the number of procedures called by a procedure. Examples of PC connectors include functions, procedures, object-oriented methods, fork and exec in

POSIX environments, callback invocation in event-based systems, and operating system calls. Higher-order connectors, such as remote procedure calls, can also be composed “on top of” a procedure call by adding *facilitation* and *conversion* services.

2.2 Event

Event connectors are similar to procedure call connectors in that they model the flow of control among components (*coordination*). In this case, the flow is precipitated with an event. Messages containing a description of the event can be generated upon the occurrence of a single event or a specific pattern of events. The contents of an event message can be structured to contain information about the event and other application-specific information (*communication*). Event-based distributed systems rely on the notion of time manifested as *causality* as well as *synchronicity* [8,9]. Further, various event *notification* mechanisms and *delivery* semantics are possible, which also depend on the *cardinality* of producers and consumers. Other dimensions of an event connector type include *priority* and *mode* of origin. An example of this connector type GUI events in a windowing application, where user actions serve as the events that activate the system. Some events, such as interrupts, page faults, and traps, are triggered by hardware and then processed by software.

2.3 Stream

Streams are used to perform transfers of large amounts of data between autonomous processes (*communication*). Streams are also used in client-server systems with data transfer protocols to deliver results of computation. Streams have been employed in formal architectural models to represent connectors with fairly complex protocols of usage [2]. Streams may provide *unidirectional* or *bidirectional* data transfer. Further, streams often keep *state* that represents the data being transferred and provide *delivery* semantics to guarantee the data transfer. Streams also *format* the data for efficient transfer suitable to the current platform. *Cardinality* is used to distinguish the two ends of a stream, namely the sender and receivers. Streams can be combined with other connector types, such as data access connectors, to provide composite connectors for performing database and file storage access, and event connectors, to multiplex the delivery of a large number of events. Examples of stream connectors are UNIX pipes, TCP/UDP communication sockets, and proprietary client-server protocols.

2.4 Distributor

Distributed systems require identification of component locations and interaction paths to them based on symbolic names. Distributor connectors perform the identification of interaction paths and

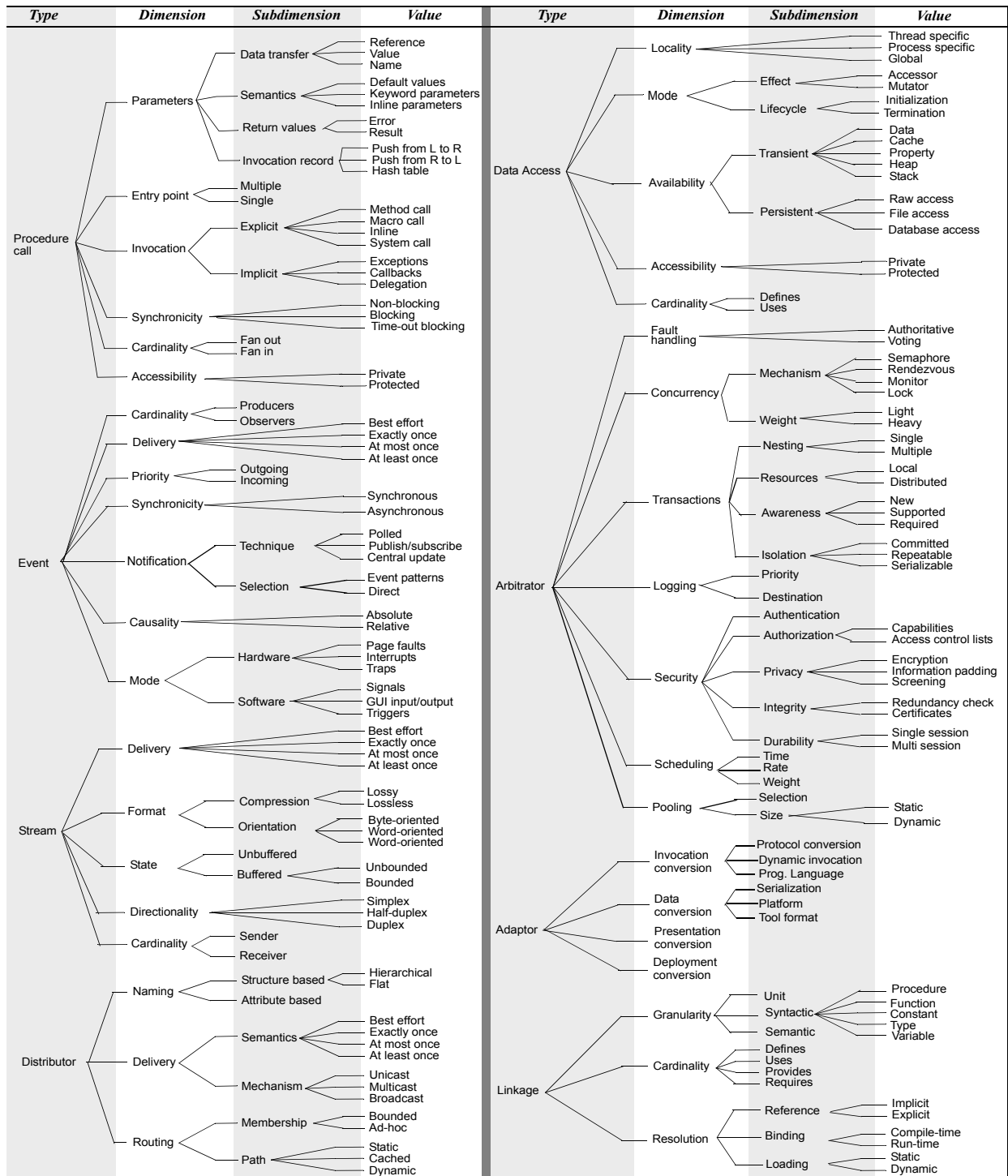


Figure 1. Connector Taxonomy.

subsequent *routing* of communication and coordination information among components along these paths (*facilitation*). They never exist by themselves, but provide assistance to other connectors, such as streams or procedure calls. Distributed systems exchange information using distributor connectors to direct the data flow. Domain Name Service (DNS) [13], and routing protocols belong to this connector type.

2.5 Data Access

Data access (DA) connectors allow components to access data maintained by a data store component [17] (*communication*). *Locality* of data being accessed determines the scope of DA usage. DA *mode* determines the kind of access to the data, i.e. read or write. Further, DA may also require preparation of the data store before and clean-up after access has been completed. *Accessibility* of DA is at times controlled for the purposes of encapsulation. The *cardinality* of

readers and writers also affects the access semantics. The data can be made *available* from a persistent or transient store, in which case the data access mechanisms will vary. Examples of persistent data access include database query mechanisms, such as SQL, and file I/O. Examples of transient data access includes heap and stack memory access, and information caching.

2.6 Arbitrator

When components are aware of the presence of other components but cannot make assumptions about their needs and state, arbitrators streamline system operation and resolve any conflicts (*facilitation*), and redirect the flow of control (*coordination*). For example, multi-threaded systems that require shared memory access use concurrency control to guarantee consistency and atomicity of operations. Arbitrators can provide facilities to negotiate service levels and mediate interactions requiring guarantees for isolation levels, reliability, and security. They also provide resource management services such as scheduling and pooling.

2.7 Adaptor

Adaptor connectors provide facilities to support interaction between components that have not been designed to inter-operate. Adaptors involve matching communication policies and interaction protocols among components (*conversion*). These connectors are necessary for inter-operation of components in heterogeneous environments, such as different programming languages or computing platforms. Conversion can also be performed to optimize component interactions for a given execution environment (e.g., LRPC [2]). Adaptors may employ transformations to match required services to the available facilities (e.g., virtual function tables used for dispatch of polymorphic method calls [4]) or support data interchange between tools (e.g., XMI [14]).

2.8 Linkage

Linkage connectors are used to tie the system components together and hold them in such a state during their operation and interaction. Linkage connectors enable the establishment of *ducts*, identified as primitives of connectors that form the channels for communication and coordination [12], which are then used by more functional connectors to enforce interaction semantics (*facilitation*).

Once ducts are established, a linkage connector may disappear from the system or remain in place to assist in the system's evolution. The *granularity* of linkage as well as the techniques used for *resolution* of the links impact the level of coupling in the system. *Cardinality* is used to distinguish the two ends of a

reference, namely the referencing and referenced element. Examples of linkage connectors are the C export mechanism and the Java dynamic class loader.

3 DETECTING MISMATCHES IN CONNECTOR COMPOSITION

The taxonomy of connectors described above can be used for synthesizing new connectors as well as analyzing compatibility of connector dimensions. Instantiating dimensions of a single connector type by choosing one or more values forms a simple connector *species*; on the other hand, using values of dimensions from different connector types leads to a composite ("higher-order") connector species. Many real-world scenarios force an architect to compose such higher-order connectors to satisfy the application requirements. Note that creating unprecedented connectors is not a trivial task. Just as component integration has presented tremendous challenges to software engineers, so too does the integration of heterogeneous connectors. At the least, it requires better understanding of the connectors' complementary, orthogonal, and incompatible characteristics as evident from some documented failures (e.g., [6]).

In the absence of a set of rules for defining composite connectors, it becomes difficult to guide the composition of connectors. However, if properly coupled with *use contracts* that specify the constraints under which a connector can be used, the taxonomy is likely to become a useful aid to software architects in defining their own connectors. Our connector taxonomy can help identify potential mismatches between incompatible connector dimensions and avoid such combinations. Figure 2 shows a compatibility matrix for the connector dimensions identified in the connector taxonomy of Figure 1, and provides guidance for which combinations are necessary for using a connector and which ones should be avoided. The sparse compatibility matrix suggests that the allowed design space of connectors appears to be very large based on our current understanding of connectors, but some pitfalls are known to exist and should be avoided.

There are four kinds of rules for combining connector dimensions: *cautions*, *requires*, *restricts* and *prohibits*. The *cautions* rule indicates that certain combinations of values for two connector dimensions, that are required to be used in tandem, may result in an unstable, or unreliable connector species. For example, a PC being invoked implicitly cannot have multiple entry points since an implicit invocation mechanism cannot choose between the entry points.

The *requires* rule indicates that the choice of a dimension requires another dimension to also be selected in a connector species. It also allows all

		Procedure call					Event					Data access				Stream				Linkage			Arbitrator					Adaptor			Distributor														
		Parameters	Entry point	Invocation	Synchronicity	Cardinality	Accessibility	Cardinality	Delivery	Priority	Synchronicity	Notification	Causality	Mode	Locality	Mode	Availability	Accessibility	Cardinality	Delivery	Format	Directionality	Cardinality	State	Granularity	Cardinality	Resolution	Fault handling	Concurrency	Transactions	Logging	Security	Scheduling	Pooling	Invocation	Data	Presentation	Deployment	Naming	Delivery	Routing				
Procedure call	Parameters	&	&											⊗										⊗																					
	Entry point		&																					⊗																					
	Invocation			⊗																				⊗																					
	Synchronicity			&	&																	⊗																							
	Cardinality			&	&																																								
Event	Cardinality																																												
	Delivery																																												
	Priority																																												
	Synchronicity																																												
	Notification																																												
	Mode																																												
Data access	Locality																																												
	Mode																																												
	Availability																																												
	Accessibility																																												
Stream	Delivery																																												
	Format																																												
	Directionality																																												
	Cardinality																																												
Linkage	Granularity																																												
	Cardinality																																												
	Resolution																																												
Arbitrator	Fault handling																																												
	Concurrency																																												
	Transactions																																												
	Logging																																												
	Security																																												
	Scheduling																																												
Adaptor	Invocation																																												
	Data																																												
	Presentation																																												
	Deployment																																												
Distributor	Naming																																												
	Delivery																																												
	Routing																																												

Symbol	Rule	Modality	Meaning
⊗	Cautions	Mandatory	The two dimensions are required simultaneously, but have restrictions on certain combinations
⊠	Prohibits	Optional	The combined use of the two dimensions is disallowed
⊙	Restricts	Optional	The combined use of the two dimensions has restrictions
&	Requires	Mandatory	The two dimensions are always required simultaneously

Figure 2. Connector Compatibility Matrix.

possible combinations of the given dimension and its required co-dimension. This chaining rule is the basis for constructing the base connector species. For example, an event connector that requires delivery semantics also needs a notification dimension, which in turn requires cardinality, synchronicity and mode. This chaining rule results in identification of dimensions that are mandatory in all species of a connector type, and those that are optional. In Figure 2, the mandatory dimensions are **bold-faced**, whereas the optional dimensions are regular-faced. Some dimensions are required only in other connector types and are *italicized* in the matrix.

The *restricts* rule is used to indicate that the two dimensions are not required to be used together at all times, and that there are certain combinations of their values that are invalid. This rule is used to prevent mismatches similar to the *cautions* rule. For example,

thread-specific data access cannot use heavy-weight concurrency (see the intersection of *Data access - Locality* with *Arbitrator - Concurrency*). On the other hand, the *cautions* rule mandates the use of a co-dimension even though some combinations of the values of the two dimensions are invalid.

Finally, the *prohibits* rule is used to exclude any combination of two dimensions from being used and indicates total incompatibility of the dimensions. For example, stream delivery cannot be built on transactional atomicity (see the intersection of *Stream - Delivery* with *Arbitrator - Transactions*).

While Figure 2 illustrates the constraints on the binary combinations of connector dimension it is our hypothesis based on our experience to date that the compatibility relations between dimensions are transitive. The compatibility rules can be successively applied to determine the n-ary compatibility between

dimensions. The transitivity of the binary relations described in this paper serves as a good starting point for analyzing the n-ary relations.

4 DISCUSSION AND CONCLUSIONS

Creating a taxonomy presents a tremendous intellectual challenge. It requires classifying the work done by hundreds of researchers and practitioners over several decades into a compact representation. We do not expect that our taxonomy is complete in its current form. Instead, it is intended to enable better and more complete understanding of software connectors and to evolve as that understanding improves. We believe that the comprehensive taxonomy of software connectors provided in this paper forms the necessary foundation for determining the characteristics of *any* connector.

During the process of creating the taxonomy and evaluating it on numerous examples over the last three years, we have found that all encountered connectors could be classified as belonging to one (in the case of simple connectors) or more (in the case of higher-order connectors) of the types. On the other hand, the dimensions, sub-dimensions, and values have evolved as our understanding of connectors evolved. Moreover, the sparser reflexive sections of the compatibility matrix (i.e. those that combine the dimensions of a connector type with the same dimensions) indicate the need for greater understanding of the given connector type (e.g., arbitrator and adaptor). On the other hand, the dimensions of a connector type with rich constraints indicate greater cohesion, and so a lower need for evolution (e.g., procedure call, event, and linkage).

Many issues remain venues of future work. We will be adding tool support for providing design guidance to architects that are composing their own connectors on the basis of the connector dimension compatibility rules. We also intend to further study connector properties, relationships, and tradeoffs in order to devise novel connectors to help automate programming tasks that currently require manual, but recurring solutions.

In addition to providing analytical support for connector composition, we are also currently investigating connector composition techniques in the context of architectural *styles*. Architectural styles represent a set of constraints on architectural elements used to define a family of systems [1]. We have proposed an approach for architectural composition, called Alfa [11], that consists of a characterization technique for styles, and style-based architectural primitives. Connectors are composed in Alfa based on the compatibility matrix presented in this paper. The initial prototype of Alfa's accompanying toolset, built

in Java, is already available from the Alfa web site [10]. This toolset supports four styles client-server, pipe-and-filter, C2, and push-based systems. We are currently studying additional styles using Alfa.

5 REFERENCES

1. G. D. Abowd, R. Allen and D. Garlan. Formalizing Style to Understand Descriptions of Software Architecture. *ACM Trans. on Software Engineering and Methodology*, 4(4): 319-364, October 1995.
2. R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, July 1997.
3. B. N. Bershad, T. E. Anderson, E. D. Lazowska and H. M. Levy. Lightweight Remote Procedure Call. *ACM Trans. on Computer Systems*, 8(1), 1990.
4. K. Driesen, U. Holzle and J. Vitek. Message Dispatch on Pipe-lined Processors. *ECOOP '95, Lecture Notes in Computer Sciences, volume 952*. Springer Verlag, 1995.
5. C. Gacek and B. Boehm. Composing Components: How does one detect architectural mismatches? *Workshop on Compositional Software Architectures*, Monterey, CA, USA, January 1998.
6. D. Garlan, R. Allen, and J. Ockerbloom. Architectural Mismatch, or, Why It's Hard to Build Systems out of Existing Parts. *ICSE '95*, Seattle, WA, USA, April 1995.
7. G. T. Heineman. Adaptation and Software Architecture. *ISAW '98*, Orlando, FL, USA, November 1998.
8. L. Lamport. Time, Clocks and the Ordering of Events in a Distributed System. *Comm. ACM*, 21(7), July 1978.
9. D. C. Luckham and J. Vera. An Event-Based Architecture Definition Language. *IEEE Trans. on Software Engineering*, September 1995.
10. Alfa web site. <http://sunset.usc.edu/~softarch/Alfa>.
11. N. R. Mehta and N. Medvidovic, Distilling Software Architectural Primitives from Architectural Styles, *Submitted to ICSE '03*, Portland, CA, USA.
12. N. R. Mehta, N. Medvidovic and S. Phadke, Towards a Taxonomy of Software Connectors. *ICSE '00*, Limerick, Ireland, May 2000.
13. P. Mockapetris, and K. J. Dunlap Development of the Domain Name System. *Symposium on Communications Architectures and Protocols*, 1988.
14. Object Management Group. XML Metadata Interchange (XMI). Proposal to the OMG OA & DTF RFP 3: Stream-based Model Interchange Format (SMIF). *OMG Document ad/98/10-05*. October 1998.
15. R. Orfali, D. Harkey, and J. Edwards. The Essential Distributed Objects Survival Guide. John Wiley & Sons, Inc., NY, USA, 1996.
16. D. E. Perry. Software Interconnection Models. *ICSE '87*, Monterey, CA, USA, May 1987.
17. D. E. Perry and A. L. Wolf. Foundations for the Study of Software Architectures. *ACM SIGSOFT Software Engineering Notes*, October 1992.
18. R. Sessions. COM and DCOM: Microsoft's Vision for Distributed Objects. John Wiley & Sons, Inc., NY, USA, 1997.
19. M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young and G. Zelesnik. Abstractions for Software Architecture and Tools to Support Them. *IEEE Trans. on Software Engineering*, April 1995.
20. M. Shaw and D. Garlan. Software Architecture: Perspectives on an Emerging Discipline. Prentice-Hall, 1996.
21. Sun Microsystems, Inc. Java 2 Enterprise Edition Specification v1.3. <http://java.sun.com/j2ee>
22. R. N. Taylor et al. A Component- and Message-Based Architectural Style for GUI Software. *IEEE Trans. on Software Engineering*, 22(6), 1996.