# Toward Boxology: Preliminary Classification of Architectural Styles

Mary Shaw and Paul Clements
Carnegie Mellon University
Pittsburgh, PA 15213

*Abstract: Software architects use a number of commonly-recognized "styles" to guide their design of system structures. We are beginning to classify these styles. We use a two-dimensional classification strategy with control and data issues as the dominant organizing axes. We position the major styles within this space and use finer-grained discriminations to elaborate variations on the styles. This provides a framework for organizing design guidance.*

*Keywords: software architecture, architectural styles, style classification/taxonomy*

## 1. Introduction

Software architecture is concerned with system structure—organization of the software, assignment of responsibilities to components, and assurance that the components' interactions satisfy the system requirements [Ga95, GS93, PW92]. Software developers recognize a number of distinct architectural styles. Many of these styles are defined informally and idiosyncratically. Our purpose here is to clarify the distinctions among styles as a first step in helping designers choose among the styles.

By *architectural style* we mean a set of design rules that identify the kinds of components and connectors that may be used to compose a system or subsystem, together with local or global constraints on the way the composition is done. *Components*, including encapsulated subsystems, may be distinguished by the nature of their computation and also by their packaging—the ways they interact with other components. To clarify the interactions we isolate their definitions in *connectors*. It is largely the interaction among components, mediated by connectors, that gives different styles their distinctive characteristics.

The style of a specific system is usually established by appeal to common knowledge or intuition. Architectures are usually expressed in box-and-line diagrams and informal prose, so the styles provide drawing conventions, vocabulary, and informal constraints (e.g., limiting topology or numbers of components of some type). Recently there has been some effort to identify and define styles more precisely and systematically [G+94, SG96, Sh96]. A few styles have been formalized or extensively analyzed [A+95, An91, Ni86]. Space does not permit us to offer primary definitions of specific styles here.


SIGSOFT 96 Workshop, San Francisco CA USA
© 1996 ACM 0-89791-867-3/96/10 ..$3.50

We are starting to organize and classify styles that appear in software descriptions so as to try to

- Establish a *uniform descriptive standard* for architectural styles—make the vocabulary used to describe styles more precise and shareable among software architects.
- Provide a systematic *organization to support retrieval* of information about styles.
- *Discriminate* among different styles—bring out significant differences that affect the suitability of a style for various tasks; show which styles are refinements of others.
- Set the stage for organizing advice on *selecting a style* for a given problem.

This classification is an intermediate step toward supporting architectural design decisions. Minimally, it will help the designer focus on important design issues by providing a checklist of topics to consider, thereby setting expectations for elements to include in the design. Eventually the classification should provide guidance for recognizing which styles are important candidates for shaping the solution. We would like to stimulate discussion of this classification at the workshop.

In this paper we classify a set of architectural styles that have been described previously (usually informally) in published literature. Table 1 shows the resulting classification. It is not complete, but it spans much of the diversity found in practice. Each row describes a style. Columns correspond to the feature categories as described below. Indented rows describe specializations of the styles in the primary rows they follow. Because this is a multidimensional classification, it is possible for a style to be a variant of more than one broader style; we handle this with cross-references.

We describe the styles in their pure forms, although they seldom occur that way. Real systems hybridize and amalgamate the pure styles, with the architect choosing useful aspects from several in order to accomplish the task at hand. Our classification does not impede this heterogeneity, but rather enhances the selection and blending process by making stylistic properties explicit. Understanding the pure forms is helpful in understanding or explaining the hybrids, and perhaps also in recognizing and eliminating unnecessary heterogeneity. Indeed, the classification activity can point out common styles that are hybrids of other styles, such as lightweight processes.

After looking through the table many readers will say, "But that's not what *I* mean by style X!". Indeed, it may not be. But it is, as far as we can tell, what *someone else* means. This is an indication that different readers use style names in different ways. A primary objective of this classification is to expose these differences and enable constructive discussion.

## 2. Classification strategy

A system designer's primary impression of an architecture often keys on the character of the interactions among components. Our classification strategy reflects this. The major axes of classification are the control and data interactions among components. We make finer discriminations within these dimensions.

Our analysis of common architectural styles suggests that they are discriminated by these feature categories:

- 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
- what type of reasoning is compatible with the style

These categories form the basis of a descriptive classification that shows essential similarities and differences among styles. Features that distinguish styles also help us understand why a particular style is an appropriate solution for one type of problem and not for another.

### 2.1 Constituent parts: components and connectors

Components and connectors are the primary building blocks of architectures. A component is a unit of software that performs some function at run-time. Examples include programs, objects, processes, and filters. A connector is a mechanism that mediates communication, coordination, or cooperation among components. Implementations of connectors are usually distributed over many system components; often they do not correspond to discrete elements of the running system. Examples include shared representations, remote procedure calls, message-passing protocols, data streams, and transaction streams.

We focus on the *abstractions* used by designers in defining their architectures. In practice, most of these elements are ultimately *implemented* in terms of processes (as defined by the operating system) and procedure calls (as defined by the programming language). More abstract connectors include format conversions that allow two otherwise-incompatible components to share data and connectors augmented by performance monitoring, authentication, or audit-trail capabilities.

The allowable kinds of components and connectors are primary discriminators among styles. Selecting the types of constituent parts does not, however, uniquely identify the style. Control disciplines, data organizations, and the interaction of control and data all affect style distinctions. So do finer distinctions within types of components and connectors, some of which appear in Table 1. For example, both *program* and *transducer* refine *process; procedure calls* may be *local* or *remote*, and their binding may be *dynamic* or *static; batch data*, *data stream*, and *continuous refresh* are all forms of *data flow*. A taxonomic treatment of architectural components and connectors, filling out the conceptual framework begun here, appears elsewhere [K+96]. Components and connectors also guide the clustering criterion in Table 1.

### 2.2 Control issues

Control issues describe how control passes among components and how the components work together temporally. Control issues include:

- **Topology**: What geometric form does the control flow for the system take? Some architectures specify quite specific topologies. Within each general topology it may be useful to stipulate the direction in which control flows. The topology may be static or dynamic; this is determined by the binding time of the partner as described below.
- **Synchronicity**: How dependent are the components' actions upon each others' control states? In a *lockstep* system, the state of any component implies the state of all others. In *synchronous* systems, components synchronize regularly and often, but other state relationships are unpredictable. *Asynchronous* components are largely unpredictable in their interaction or synchronize once in a while, while *opportunistic* components such as autonomous agents work completely independently from each other in parallel.

  Lockstep systems can be *sequential* or *parallel*, depending on how many threads of control run through them. Other forms of synchronicity imply parallelism.
- **Binding time**: When is the identity of a partner in a transfer-of-control operation established? Some control transfers are pre-determined at program-*write* (i.e., source code) time, *compile* time, or *invocation* time (i.e., when the operating system initializes the process). Others are bound dynamically while the system is *running*.

### 2.3 Data issues

Data issues describe how data moves around a system. Data issues include:

- **Topology**: Data topology describes the geometric shape of the system's data flow graph. The alternatives are as for control topology
- **Continuity**: How continuous is the flow of data throughout the system? A *continuous*-flow system has fresh data available at all times; a *sporadic*-flow system has new data generated at discrete times. Data rates may be *high-volume* (in data-intensive systems) or *low-volume* (in compute-intensive systems).
- **Mode**: Data mode describes how data is made available throughout the system. It may be *passed* from component to component, or it is *shared* by making it available in a place accessible to all the sharers. If the components tend to modify it and re-insert it into the public store, this is a *copy-out-copy-in* mode. In some styles data is *broadcast* or *multicast* to specific recipients.
- **Binding time**: When is the identity of a partner in a transfer-of-control operation established? This is the data analogy of the same control issue,

### 2.4 Control/data interaction issues

Interaction issues describe the relationship between certain control and data issues.

- **Shape**: Are the control flow and data flow topologies substantially isomorphic to each other?
- **Directionality**: If the shapes are substantially the same, does control flow in the *same* direction as data or the *opposite* direction?

# Table 1: A feature-based classification of architectural styles

| Style | Constituent parts | | Control issues | | | Data issues | | | | Control/data interaction | | Type of reasoning |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Components | Connectors | Topology | Synchronicity | Binding time | Topology | Continuity | Mode | Binding time | Isomorphic shapes | Flow directions | |
| **Data flow styles: Styles dominated by motion of data through the system, with no "upstream" content control by recipient** | | | | | | | | | | | | |
| Batch sequential [Be90] | stand-alone programs | batch data | linear | seq | r | linear | spor hvol | passed, shared | r | yes | same | Functional composition |
| Dataflow network [B+88] | transducers | data stream | arb | asynch | i, r | arb | cont lvol or hvol | passed | i, r | yes | same | |
| •Sub-styles | *Several substyles can be elaborated, following [A+95]* | | | | | | | | | | | |
| Closed loop control [Sh95] | embedded process, function | continuous refresh | fixed | asynch | w | fixed cyclic[1] | cont lvol | passed, shared | w | no | n/a | |
| **Call-and-return styles: Styles dominated by order of computation, usually with single thread of control** | | | | | | | | | | | | |
| Main program/subroutines [Pa72, Bo86] | procedures, data | procedure calls | hier | seq | w, c | arb | spor lvol | passed, shared | w, c, r | no | n/a | Hierarchy (local reasoning) |
| Information hiding systems [Pa72] | managers | procedure calls | arb | seq | w, c, r | arb | spor lvol | passed | w, c, r | yes | same | |
| •Abstract data types [Sh81] | managers | static procedure calls | arb | seq | w, c | arb | spor lvol | passed | w, c, r | yes | same | |
| •Classical[2] objects [Bo86] | managers (objects) | dynamic procedure calls | arb | seq | w, c, r | arb | spor lvol | passed | w, c, r | yes | same | |
| •Naive[3] client/server | programs | procedure calls or RPCs | star | synch | w, c, r | star | spor lvol | passed | w, c, r | yes | opposite | |
| **Interacting process styles: Styles dominated by communication patterns among independent, usually concurrent, processes** | | | | | | | | | | | | |
| Communicating processes [An91, Pa85] | processes | message protocols | arb | Any but seq | w, c, r | arb | spor lvol | any | w, c, r | possibly | if isomorphic, either | Nondeterminism |
| •Lightweight processes | lightweight processes | threads, (shared data[4]) | arb | ls/par, synch | w, c | arb | spor (th), cont (da) | passed, shared | w, c | no | n/a | |
| •Distributed objects | managers | remote proc calls | arb | ls/par, synch | w, c, r | arb | spor lvol | passed | w, c, r | no | n/a | |
| •Process-based naive client/server[3] | processes | request/reply messages | star | synch | w, c, r | star | spor lvol | passed | w, c, r | yes | opposite | |
| •Other sub-styles | *Several substyles can be elborated, following [An91]* | | | | | | | | | | | |
| Event systems [Ba86b, G+92, Ge89, HN86, He69, KP88, Re90] | processes | implicit invocation | arb | asynch, opp | i, r | arb | spor lvol | bdcast | i, r | no | n/a | |

# Table 1: A feature-based classification of architectural styles

| Style | Constituent parts | | Control issues | | | Data issues | | | | Control/data interaction | | Type of reasoning |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Components | Connectors | Topology | Synchronicity | Binding time | Topology | Continuity | Mode | Binding time | Isomorphic shapes | Flow directions | |
| **Data-centered repository styles:** Styles dominated by a complex central data store, manipulated by independent computations | | | | | | | | | | | | Data integrity |
| Transactional database [Be90, Sp87] | memory, computations | trans. streams (queries) | star | asynch, opp | w | star | spor lvol | shared, passed | w | possibly | if isomorphic, opposite | ACID[5] properties |
| •Client/server | managers, computations | transaction opns with history[3] | star | asynch. | w, c, r | star | spor lvol | passed | w, c, r | yes | opposite | |
| Blackboard [Ni86] | memory, computations | direct access | star | asynch, opp | w | star | spor lvol | shared, mcast | w | no | n/a | convergence |
| Modern compiler [SG96] | memory, computations | procedure call | star | seq | w | star | spor lvol | shared | w | no | n/a | invariants on parse tree |
| **Data-sharing styles:** Styles dominated by direct sharing of data among components | | | | | | | | | | | | |
| Compound document | editable documents | shared representation | n/a | n/a | n/a | hier | cont | shared | r | n/a | n/a | Representation |
| Hypertext | documents | internal refs. | | | | arb | cont | shared | w, c, r | | | |
| Fortran common, Jovial Compool | data structures | sharing (aliasing) | | | | arb | cont | shared | w, c | | | |
| Lightweight processes[4] | *See interacting processes style group. This style hybridizes processes and shared data, with emphasis on process* | | | | | | | | | | | |
| **Hierarchical styles:** Styles dominated by reduced coupling, with resulting partition of the system into subsystems with limited interaction | | | | | | | | | | | | |
| Layered [Fr85, LS79] | various | various | hier | any | any | hier | spor lvol, cont | any | w, c, i, r | often | same or opp | Levels of service |
| •Interpreter (Virtual machine) [HR85] | memory, state machine | direct data access | fixed hier | seq | w, c | hier | cont | shared | w, c | no | n/a | |

| Key to column entries | |
|---|---|
| Topology | hier (hierarchical), arb (arbitrary), star, linear (one-way), fixed (determined by style) |
| Synchronicity | seq (sequential, one thread of control), ls/par (lockstep parallel), synch (synchronous), asynch (asynchronous), opp (opportunistic) |
| Binding time | w (write-time--that is, in source code), c (compile-time), i (invocation-time), r (run-time) |
| Continuity | spor (sporadic), cont (continuous), hvol (high-volume), lvol (low-volume) |
| Mode | shared, passed, bdcast (broadcast), mcast (multicast), ci/co (copy-in/copy-out) |

**Notes:**

1. Closed loop control establishes a controlling relation between an embedded process and a control function that responds to perturbations.
2. By "classical object" we mean objects as they originally emerged: non-concurrent, interacting via procedure-like methods. Objects are now often defined much more broadly, especially in their types of interactions.
3. True client/server systems maintain context that captures the current state of an ongoing series of actions. "Client/server" is sometimes used to describe systems that ignore this requirement and simply use components that call and define procedures or send request/reply messages amonge processes. We call the latter "naive client/server systems."
4. Lightweight processes may take advantage of the shared name space; they become a hybrid of communicating processes and shared data.
5. The ACID properties are atomicity, consistency, isolation, and durability.

## 2.5 Type of reasoning

Different classes of architectures lend themselves to different types of analysis. A system of components operating asynchronously in parallel yields to vastly different reasoning approaches (e.g., nondeterministic state machine theory) than a system that executes as a fixed sequence of atomic steps (e.g., function composition). Many analysis techniques compose their results from analysis of substructures, but this depends on the ability to combine sub-analyses. The fit of an analysis technique to an architecture is enhanced if the software organization matches the analysis organization—that is, if software substructure and analysis substructure are compatible. Thus, different architectural styles are good matches for different analysis techniques. Your choice of architecture may be influenced by the kinds of analysis you require.

# 3. Conclusion

Architectural styles have the potential to become the *lingua franca* of architecture-level design, in the same way that design patterns are moving to center stage in establishing the vocabulary and defining the solution space for finer-grained design problems. In order to capitalize on the shared experience represented by the repeated use of styles by system builders, it is necessary to establish a common vocabulary and a common descriptive framework for communicating about styles and the circumstances in which they are useful. We are working on such a descriptive framework.

# 4. Acknowledgments

This material has been developed over a period of several years. We particularly appreciate ongoing discussions with David Garlan, Rob DeLine, and Greg Zelesnik. Many others have offered useful advice, particularly our colleagues in the Composable Systems research group, the Software Architecture Reading Group, and the Garlan and Shaw's software architecture course. This work has profited from efforts of the Software Engineering Institute's Software Architecture Analysis Method (SAAM) group, especially Len Bass, Gregory Abowd, and Rick Kazman, who have provided a taxonomic classification of architectural elements (components and connectors), which provides the conceptual continuation of the work presented here. This work has been supported by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency, under grant F33615-93-1-1330 and by a grant from Siemens Corporation. It represents the views of the author and not of Carnegie Mellon University or any of the sponsoring institutions The Software Engineering Institute is supported by the U. S. Department of Defense.

# 5. Bibliography

[A+95] Gregory D. Abowd, Robert Allen, David Garlan. Formalizing Style to Understand Descriptions of Software Architecture. *ACM Transactions on Software Engineering and Methodology*, 4(4):319-364, Oct 1995.

[An91] Gregory R. Andrews. Paradigms for Process Interaction in Distributed Programs. *ACM Computing Surveys*, 23(1):49-90, Mar 1991.

[B+88] M. R. Barbacci, C. B. Weinstock, and J. M. Wing. Programming at the Processor-Memory-Switch Level. *Proc 10th Int'l Conf on Software Engineering*, Apr 1988.

[Ba86b] Robert M. Balzer. Living with the Next Generation Operating System. *Proc 4th World Computer Conf.*, Sep 1986.

[Be90] Laurence J. Best. *Application Architecture: Modern Large-Scale Information Processing*. Wiley, 1990.

[Bo86] Grady Booch. Object-Oriented Development. *IEEE Tr. Software Engineering*, Feb 1986, pp. 211-221.

[Fr85] Marek Fridrich and William Older. Helix: The Architecture of the XMS Distributed File System. *IEEE Software*, vol 2, no 3, May 1985 (pp.21-29).

[G+92] David Garlan, Gail Kaiser, and David Notkin. Using Tool Abstraction to Compose Systems. *IEEE Computer*, 25(6), June 1992.

[G+94] David Garlan, Robert Allen, and John Ockerbloom. Exploiting Style in Architectural Design Environments. *Proc. Second ACM SIGSOFT Symposium on Foundations of Software Engineering*, Dec 1994.

[Ga95] David Garlan (ed). First International Workshop on Architectures for Software Systems, Workshop Summary. *ACM Software Engineering Notes* 20(3), July 1995, pp.84-89.

[Ge89] C. Gerety. HP Softbench: A New Generation of Software Development Tools. *TR SESD-89-25*, Hewlett-Packard Software Engineering System Division, Ft. Collins CO, Nov1989.

[GS93] David Garlan and Mary Shaw. An Introduction to Software Architecture. In Ambriola & Tortora (eds), *Advances in Software Engineering & Knowledge Engineering*, vol. II, World Scientific Pub Co., 1993, pp.1-39.

[He69] Carl Hewitt. Planner A Language for Proving Theorems in Robots. *Proc First Int'l Joint Conf. in Artificial Intelligence*, 1969.

[HR85] Frederick Hayes-Roth. Rule-Based Systems. *Communications of the ACM*, vol 28, no 9, Sep 1985, pp.921-932.

[K+96] Rick Kazman, Paul Clements, Gregory Abowd, Len Bass. Classifying Architectural Elements. Unpublishd manuscript.

[KP88] G. Krasner and S. Pope. A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. *Journal of Object Oriented Programming*, vol 1, Aug/Sep 1988.

[LS79] Hugh C. Lauer and Ed. H. Satterthwaite. Impact of MESA on System Design. *Proc Third Int'l Conf. on Software Engineering*, May 1979.

[Ni86] H. Penny Nii. Blackboard Systems. *AI Magazine* 7(3):38-53 and 7(4):82-107.

[Pa72] David L. Parnas. On the Criteria to be Used in Decomposing Systems into Modules. *Comm. ACM* vol 15, Dec1972.

[Pa85] Mark C. Paulk. The ARC Network: A Case Study. *IEEE Software*, vol 2 no 3, May 1985, pp. 62-69

[PW92] Dewayne E. Perry and Alexander L. Wolf. Foundations for the Study of Software Architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40-52, Oct 1992.

[Re90] S. P. Reiss. Connecting Tools Using Message Passing in the Field Environment. *IEEE Software*, 7(4):57-66, Jul 1990.

[SG96] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall 1996.

[Sh81] Mary Shaw (ed). *Alphard: Form and Content*. Springer-Verlag, 1981.

[Sh95] Mary Shaw. Beyond Objects: A Software Design Paradigm Based on Process Control. *ACM Software Engineering Notes*, 20(1), Jan 1995.

[Sh96] Mary Shaw. Some Patterns for Software Architectures. *Proceedings of Second Workshop on Pattern Languages for Programming*, Addison-Wesley 1996.

[Sp87] Alfred Z. Spector et al. Camelot: A Distributed Transaction Facility for Mach and the Internet - An Interim Report. *Carnegie Mellon University Computer Science Technical Report*, Jun 1987.