Some Patterns for Software Architectures

Chapter ??

by
Mary Shaw
Carnegie Mellon University

Abstract

Software designers rely on informal patterns, or idioms, to describe the architectures of their software systems—the configurations of components that make up the systems. At the first PLoP, I identified seven patterns that guide high-level system design and discussed the way they guide the composition of systems from particular types of components [Shaw 95]. This paper extends the descriptions of those patterns (plus one) in response to the discussion at the conference. Most significantly, it adds information on the kinds of problems each pattern handles best.

Design patterns for software architectures

Software designers describe overall system architectures using a rich vocabulary of abstractions. Although the descriptions and the underlying vocabulary are imprecise and informal, designers nevertheless communicate with some success. They depict the architectural abstractions both in pictures and words.

“Box-and-line” diagrams often illustrate system structure. These diagrams use different shapes to suggest structural differences among the components, but they make little discrimination among the lines—that is, among different kinds of interactions. The architectural diagrams are often highly specific to the systems they describe, especially in the labeling of components.

The diagrams are supported by prose descriptions. This prose uses terms with common, if informal, definitions:

> "Camelot is based on the client-server model and uses remote procedure calls both locally and remotely to provide communication among applications and servers." [Spector 87]

> "Abstraction layering and system decomposition provide the appearance of system uniformity to clients, yet allow Helix to accommodate a diversity of autonomous devices. The architecture encourages a client-server model for the structuring of applications." [Fridrich 85]

> "We have chosen a distributed, object-oriented approach to managing information." [Linton 87]

> "The easiest way to make the canonical sequential compiler into a concurrent compiler is to pipeline the execution of the compiler phases over a number of processors. ... A more effective way [is to] split the source code into many segments, which are concurrently processed through the various phases of compilation [by multiple compiler processes] before a final, merging pass recombines the object code into a single program." [Seshadri 88]

> "The ARC network [follows] the general network architecture specified by the ISO in the Open Systems Interconnection Reference Model. It consists of physical and data layers, a network layer, and transport, session, and presentation layers." [Paulk 85]

We studied sets of such descriptions and found a number of abstractions that govern the overall organization of the components and their interactions. A few of the patterns (e.g., object organizations [Booch 86 and blackboards [Nii 86]) have been carefully refined, but others are still used quite informally, even unconsciously. Nevertheless, the architectural patterns are widely recognized. System designs often appeal to several of these patterns, combining them in various ways.

Garlan and Shaw [Garlan and Shaw 93] describe several common patterns for architectures. This is not, of course, an exhaustive list; it offers rich opportunities for both elaboration and structure. These idiomatic patterns differ in four major respects: the underlying intuition behind the pattern, or the system model; the kinds of components that are used in developing a system according to the pattern; the connectors, or kinds of interactions among the components; and the control structure or execution discipline. By using the same descriptive scheme, we improve our ability to identify significant differences among the patterns. Once the informal pattern is clear, the details can be formalized [Allen and Garlan

94]. Further, choosing the architecture for a system should include matching characteristics of the architecture to properties of the problem [Jackson 94, Lane 90]; having uniform descriptions of the available architectures should simplify this task.

Systems are composed from identifiable *components* of various distinct types. The components interact in identifiable, distinct ways. Components roughly correspond to compilation units of conventional programming languages and other user-level objects such as files. Connectors mediate interactions among components; that is, they establish the rules that govern component interaction and specify any auxiliary implementation mechanism required. Connectors do not in general correspond individually to compilation units; they manifest themselves as table entries, instructions to a linker, dynamic data structures, system calls, initialization parameters, servers that support multiple independent connections, and the like. A pattern is based on selected types of components and connectors, together with a *control structure* that governs execution. An overall *system model* captures the intuition about how these are integrated [Shaw et al 95].

Architectural patterns

We turn now to patterns for some of the major architectural abstractions. The purpose of each of these patterns is to impose an overall structure for a software system or subsystem that

- is appropriate to the problem the system or subsystem is solving
- clarifies designer's intentions about the organization of the system or subsystem
- provides a paradigm that will help establish and maintain internal consistency
- allows for appropriate checking and analysis
- preserves information about the structure for reference during later maintenance

In practice, a designer adopts one or more of these patterns to shape the design. Patterns may be used in combination either by providing complementary views during initial design -- as both a repository and an interpreter [Garlan and Shaw 93] -- or by elaborating a component of one pattern using some other pattern -- as a layered system in which some layers are elaborated as pipelines and others as data abstractions. This progressive elaboration can be continued repeatedly until the architectural issues are resolved, at which point conventional programming techniques take over.

The description of each pattern includes notes on

- **Problem:** What problem the pattern addresses. That is, what characteristics of the application requirements lead the designer to select this pattern?
- **Context:** What aspects of the setting (computation environment or other constraints on the implementation) constrain the designer in the use of this pattern?
- **Solution:** The system model captured by the pattern, together with the components, connectors, and control structure that make up the pattern.
- **Diagram:** A figure showing a typical pattern, annotated to show the components and connectors.
- **Significant Variants:** For some patterns, notes some major variants of the basic pattern.
- **Examples:** References to examples or more extensive overviews of systems that apply this pattern

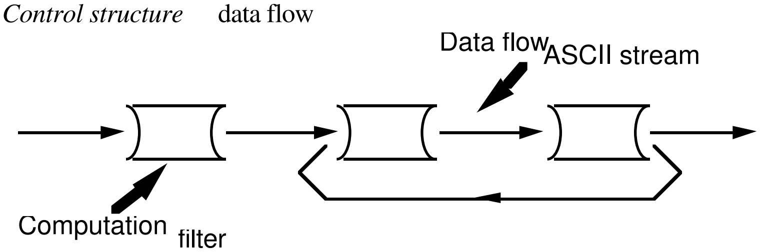
The pipeline architectural pattern

Problem: This pattern is suitable for applications that require a defined series of independent computations to be performed on ordered data. It is particularly useful when each of the computations can be performed incrementally on a data stream. In such cases the computations can, at least in principle, proceed in parallel; when this is possible it can reduce the latency of the system.

Context: The pattern relies on being able to decompose the problem into a set of computations, or *filters*, that transform one or more input streams *incrementally* to one or more output streams. The usual implementation embeds each transformation in a separate process and relies on operating system operations, or *pipes*, to stream the data from one process to another. The analysis is simplest if the filters do not interact except via the explicitly defined pipes.

Solution:

System model	data flow between components, with components that incrementally map data streams to data streams
Components	filters (purely computational, local processing, asynchronous)
Connectors	data streams (ASCII data streams for unix pipelines)
Control structure	data flow



Significant Variants: This pattern is commonly mentioned by unix programmers, who often use it for prototyping. Note that the pattern calls for "pure" filters, with local processing and little state. Unix "filters", however, often consume the entire input stream before producing output. This still works for systems without loops, but it interrupts the smooth flow of information through the system and can cause starvation if the data flow topology includes loops.

Examples: [Allen and Garlan 92] [Bach 86] [Barbacci et al 88] [Delisle and Garlan 90] [Seshadri 88] [Kahn 74]

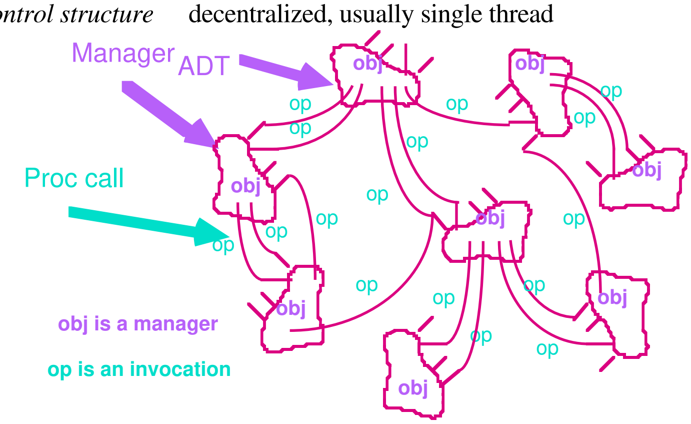
The data abstraction architectural pattern

Problem: This pattern is suitable for applications in which a central issue is identifying and protecting related bodies of information, especially representation information. When the solution is decomposed to match the natural structure of the data in the problem domain, the components of the solution can encapsulate the data, the essential operations on the data, and the integrity constraints, or *invariants*, of the data and operations.

Context: Numerous design methods provide strategies for identifying natural objects. Newer programming languages support various variations on the theme, so if the language choice or the methodology is fixed, that will strongly influence the flavor of the decomposition.

Solution:

System model	localized state maintenance
Components	managers (e.g., servers, objects, abstract data types)
Connectors	procedure call
Control structure	decentralized, usually single thread



Significant Variants: Classical objects (non-concurrent, interacting via procedure-like methods) are closely related. They differ largely in the use of inheritance to manage collections of related definitions and in the use of run-time binding for procedure calls (method invocation is essentially procedure call with dynamic binding).

Examples: [Booch 86] [GoF 95] [Linton 87] [Parnas 72]

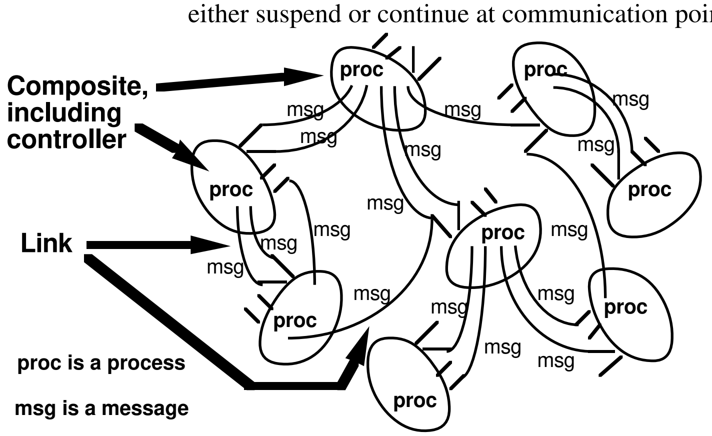
Problem: This pattern is suitable for applications that involve a collection of distinct, largely independent computations whose execution should proceed independently. The computations involve coordination of data or control at discrete points in time. As a result, correctness of the system requires attention to the routing and synchronization of the messages. Note that this pattern should be distinguished from data flow, which is generally taken to be of smaller granularity, higher regularity, and unidirectional flow. Although many other patterns may be *implemented* with message passing, the message-passing pattern is intended to apply when the essential character or the abstraction involves communication.

The communicating processes architectural pattern

Context: The selection of a communication strategy is often dictated by the communication support provided by the available operating system.

Solution:

System model	independent communicating processes
Components	processes that send and receive messages to/from explicitly selected recipients
Connectors	discrete messages (no shared data) with known communication partners.
Control structure	each process has its own thread of control, which may either suspend or continue at communication points



Significant Variants: Specific patterns of communication have proven useful in specific situations. The major points of variance include the topology of the communication network, the requirements on delivery, synchronization, and the number of recipients of each message (e.g., simple messages, broadcast, multicast).

Examples: [Andrews 91] [Paulk 85]

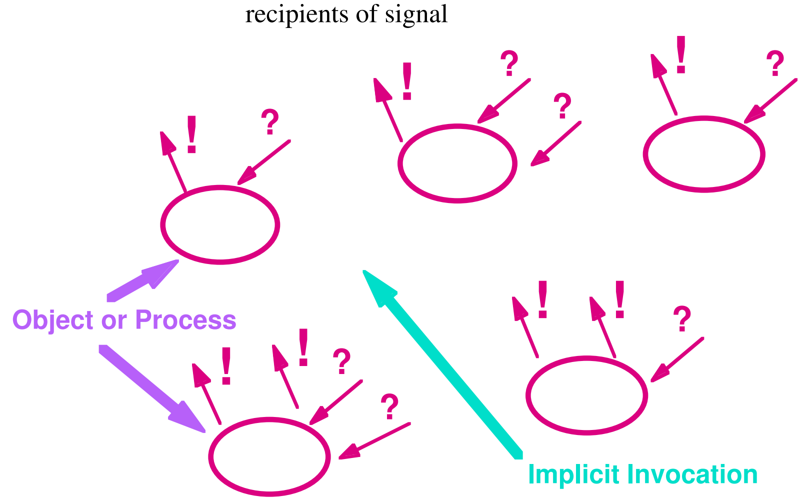
The implicit invocation architectural pattern

Problem: This pattern is suitable for applications that involve loosely coupled collection of components, each of which carries out some operation and may in the process enable other operations. These are often *reactive* systems. The pattern is particularly useful for applications that must be reconfigurable on the fly, either by changing a service provider or by enabling and disabling capabilities.

Context: Implicit invocation systems usually require an event handler that registers components' interest in receiving events and notifies them when events are raised. They differ in the kind and amount of information that accompanies the event, so it's important to choose one that fits the problem. Reasoning about the correctness of the system depends very heavily on reasoning about the kinds of events, the collection of components, and their collective effect; this is trickier than reasoning about correctness when the execution order is known.

Solution:

System model	independent reactive processes
Components	processes that signal significant events without knowing recipients of signals
Connectors	automatic invocation of processes that have registered interest in events
Control structure	decentralized; individual components are not aware of recipients of signal



Examples: [Balzer 86] [Garlan et al 92] [Gerety 89] [Habermann and Notkin 86] [Hewitt 69] [Krasner and Pope 88] [Reiss 90] [Shaw et al 83]

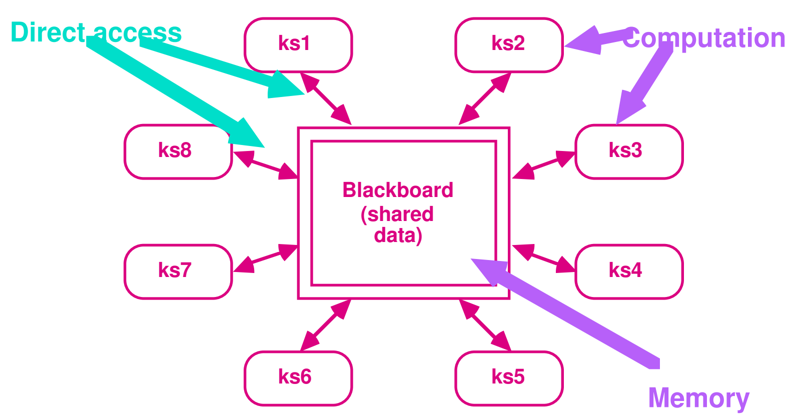
The repository architectural pattern

Problem: This pattern is suitable for applications in which the central issue is establishing, augmenting, and maintaining a complex central body of information. Typically the information must be manipulated in a wide variety of ways. Often long-term persistence may also be required. Different variants support radically different control strategies.

Context: Repositories often require considerable support, either an augmented runtime system (such as a database) or a framework or generator to process the data definitions.

Solution:

System model	centralized data, usually richly structured
Components	one memory, many purely computational processes
Connectors	computational units interact with memory by direct data access or procedure call
Control structure	varies with type of repository; may be external (depends on input data stream, as for databases), predetermined (as for compilers), or internal (depends on state of computation, as for blackboards)



Significant Variants: The repository pattern covers large centralized transaction-oriented databases, the blackboard systems used for some AI applications, and systems with predetermined execution patterns in which different phases add information to a single complex data structures (e.g., compilers). These variants differ chiefly in their control structure.

Examples: [Ambriola 90] [Nii 86] [Barstow et al 84]

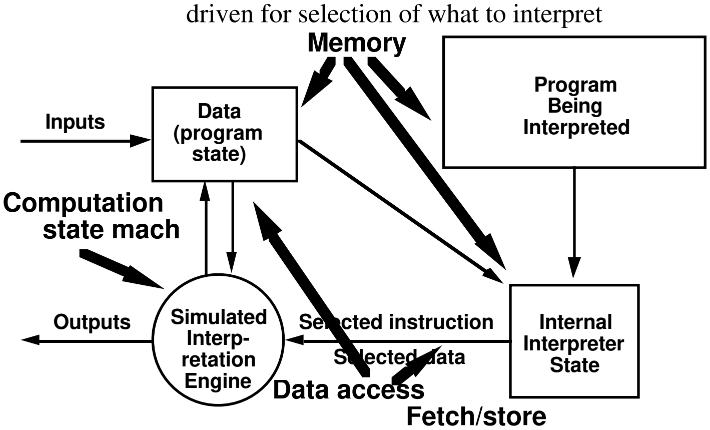
The interpreter architectural pattern

Problem: This pattern is suitable for applications in which the most appropriate language or machine for executing the solution is not directly available. The pattern is also suitable for applications in which the core problem is defining a notation for expressing solutions, for example as scripts. Interpreters are sometimes used in chains, translating from the desired language/machine to an available language/machine in a series of stages.

Context: The interpreter will most often be designed to bridge the gap between the desired machine or language and some (possibly virtual) machine or language already supported by the execution environment.

Solution:

System model	virtual machine
Components	one state machine (the execution engine) and three memories (current state of execution engine, program being interpreted, current state of program being interpreted)
Connectors	data access and procedure call
Control structure	usually state-transition for execution engine; input-driven for selection of what to interpret



Significant Variants: Expert systems are often implemented as interpreters for the collections of rules, or productions, that represent the expertise. Because the productions require a complex selection rule, specialized forms of interpreters have evolved.

Examples: [Hayes-Roth 85]

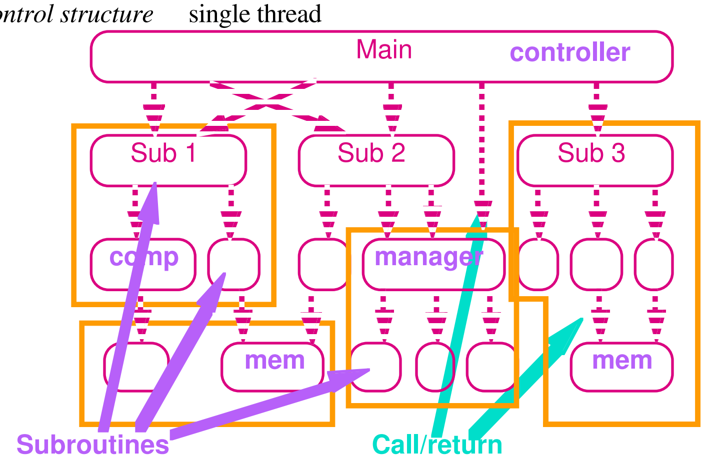
Problem: This pattern is suitable for applications in which the computation can appropriately be defined via a hierarchy of procedure definitions. It is usually used with a single thread of control.

The main program and subroutines architectural pattern

Context: Many programming languages provide natural support for defining nested collections of procedures and for calling them hierarchically. These languages often allow collections of procedures to be grouped into modules, thereby introducing name-space locality. The execution environment usually provides a single thread of control in a single name space.

Solution:

System model	call and definition hierarchy, subsystems often defined via modularity
Components	procedures and explicitly visible data
Connectors	procedure calls and explicit data sharing
Control structure	single thread



Significant Variants: The model of procedure call as the organizing principle is preserved across processes through the Remote Procedure Call (RPC). Although RPC is typically implemented by communication messages, the abstraction it presents is of single-threaded procedure call.

Examples: [Parnas 72] [Spector 87]

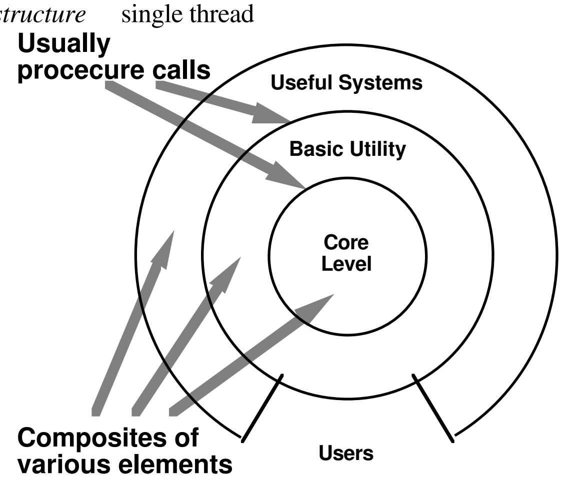
The layered architectural pattern

Problem: This pattern is suitable for applications that involve distinct classes of services that can be arranged hierarchically. Often there are layers for basic system-level services, for utilities appropriate to many applications, and for specific tasks of the application.

Context: Frequently, each class of service is assigned to a layer and several different patterns are used to refine the various layers. Layers are most often used at the higher levels of design, using different patterns to refine the layers.

Solution:

System model	hierarchy of opaque layers
Components	usually composites; composites are most often collections of procedures
Connectors	depends on structure of components; often procedure calls under restricted visibility, might also be client-server
Control structure	single thread



Significant Variants: (a) Layers may be transparent (interfaces from lower layers show through) or opaque (only the interface defined by this layer may be used by the next layer up). (b) Layered systems are often organized as chains of virtual machines or interpreters.

Examples: [Batory and O'Malley 91] [Fridrich 85] [Lauer and Satterthwaite 19] [Paulk 85]

Acknowledgments

The technical results reported here were developed jointly with various co-authors, especially David Garlan. A good share of the motivation has come from sources outside computer science, most notably Alexander's work on pattern languages and some conversations with Vic Vyssotsky on urban planning. Discussions of the paper at the 1994 and 1995 PLoP workshops showed me a number of ways to improve the content and presentation

This research was supported by the Carnegie Mellon University School of Computer Science, by a grant from Siemens Corporate Research, and by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA) under grant F33615-93-1-1330. Views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of any sponsor.

References

[Allen and Garlan 94] Robert Allen and David Garlan. Formalizing Architectural Connection. In *Proc 16th International Conference on Software Engineering*, 1994.

[Ambriola 90] V. Ambriola, P. Ciancarini, and C. Montangero. Software Process Enactment in Oikos. *Proc Fourth ACM SIGSOFT Symposium on Software Development Environments*, SIGSOFT Software Engineering Notes, December 1990.

[Andrews 91] Gregory. R. Andrews. Paradigms for Process Interaction in Distributed Programs. *ACM Computing Surveys* vol 23, no 1, March 1991.

[Bach 86] M. J. Bach. The Design of the UNIX Operating System. Software Series, Prentice-Hall 1986, sec 5.12, pp. 111-119.

[Balzer 86] Robert M. Balzer. Living with the Next Generation Operating System. Proc 4th World Computer Conf., September 1986.

[Barbacci et al 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*, April 1988.

[Barstow et al 84] D. R. Barstow, H. E. Shrobe, and E. Sandewall (eds). *Interactive Programming Environments*. McGraw-Hill 1984.

[Batory and O'Malley 91] D. Batory and S. O'Malley. The Design and Implementation of Hierarchical Software Systems Using Reusable Components. TR 91-22, Dept. of Computer Science, University of Texas, Austin, June 1991.

[Booch 86] Grady Booch. Object-Oriented Development. *IEEE Tr. on Software Engineering*, February 1986, pp. 211-221.

[Delisle and Garlan 90] Norman Delisle and David Garlan. Applying Formal Specification to Industrial Problems: A Specification of an Oscilloscope. *IEEE Software*, September 1990.

[Fridrich 85] 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).

[Garlan and Shaw 93] David Garlan & Mary Shaw. An Introduction to Software Architecture. In Ambriola & Tortora (eds), *Advances in Software Engineering and Knowledge Engineering*, vol. II, World Scientific Pub Co., 1993.

[Garlan et al 92] David Garlan, Gail E. Kaiser, and David Notkin. Using Tool Abstraction to Compose Systems. *IEEE Computer*, vol 25, June 1992.

[Gerety 89] C. Gerety. HP Softbench: A New Generation of Software Development Tools. TR SESD-89-25, Hewlett-Packard Software Engineering System Division, Ft Collins CO, November 1989.

[GoF 95] E. Gamma, R. Helm. R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley 1995.

[Habermann and Notkin 86] Nico Habermann and David Notkin. Gandalf: Software Development Environments. *IEEE Tr on Software Engineering,* vol SE-12, December 1986.

[Hayes-Roth 85] F. Hayes-Roth. Rule-Based Systems. *Comm. ACM* vol 28, September 1985.

[Hewitt 69] Carl Hewitt. Planner A Language for Proving Theorems in Robots. *Proc First Int'l Joint Conf. in Artificial Intelligence*, 1969.

[Jackson 94] Michael Jackson. Problems, Methods, and Specializations (A contribution to the Special Issue on Software Engineering in the Year 2001). *IEE Software Engineering Journal*, November 1994.

[Kahn 74] G. Kahn. The semantics of a simple language for parallel programming. *Information Processing*, 1974.

[Krasner and Pope 88] 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, August/September 1988.

[Lane 90] Thomas G. Lane. Studying Software architecture Through Design Spaces and Rules. *Carnegie Mellon University Technical Report* , September 1990.

[Lauer and Satterthwaite 79] Hugh C. Lauer and Ed. H. Satterthwaite. Impact of MESA on System Design. Proc Third Int'l Conf. on Software Engineering, May 1979.

[Linton 87] Mark A. Linton. Distributed Management of a Software Database. *IEEE Software*, 4, 6, November 1987.

[Nii 86] H. Penny Nii. Blackboard Systems. *AI Magazine* 7(3):38-53 and 7(4):82-107.

[Paulk 85] Mark C. Paulk. The ARC Network: A Case Study. *IEEE Software*, vol 2 no 3, May 1985, pp. 62-69

[Parnas 72] David L. Parnas. On the Criteria to be Used in Decomposing Systems into Modules. *Comm. ACM* vol 15, December 1972.

[Reiss 90] S. P. Reiss. Connecting Tools Using Message Passing in the Field Program Development Environment. *IEEE Software*, July 1990.

[Seshadri 88] V. Seshadri et al. Semantic Analysis in a Concurrent Compiler. *Proceedings of ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*.

[Shaw 95] Mary Shaw. Patterns for Software Architectures. In Coplien & Schmidt (eds),*Pattern Languages of Program Design,* Addison-Wesley 1995, pp. 453-462.

[Shaw et al 83] Mary Shaw, Ellen Borison, Michael Horowitz, Thomas Lane, David Nichols, and Randy Pausch. Descartes: A Programming-Language Approach to Interactive Display Interfaces. *Proc SIGPLAN '83: Symposium on Programming Language Issues in Software Systems*, ACM SIGPLAN Notices, vol 18, June 1983.

[Shaw et al 95] Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young, Gregory Zelesnik. Abstractions for Software Architecture and Tools to Support Them. *IEEE Tr. on Software Engineering*, May 1995.

[Spector 87] 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, June 1987.