

- "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]

The prose is usually accompanied by a box-and-line diagram that depicts the intended system architecture. In these diagrams, shapes suggest differences among the components, but there is little discrimination among the lines—that is, among different kinds of interactions. The descriptions are highly specific to the systems they describe, especially the labeling of components. The descriptions and their underlying vocabulary are highly informal, but they nevertheless appear to communicate effectively.

We studied sets of such descriptions and found a number of patterns that recur regularly. Some of these patterns govern the overall style that organizes the components; others identify the character of a component interface or an abstraction for component interaction. A few of the patterns (e.g., objects) have been carefully refined [Booch 86], but others are still used quite informally, even unconsciously. Nevertheless, the idiomatic patterns are widely recognized. System designs often appeal to several of these idioms, combining them in various ways. Inspecting descriptions of actual systems shows that the motivations for using different idioms are often not carefully separated, and the interactions of the idioms are correspondingly obscure.

Garlan and Shaw [Garlan and Shaw 93] describe several common architectural styles. This is not, of course, an exhaustive list; it offers rich opportunities for both elaboration and structure. By abstracting from the details of individual examples, we can identify underlying patterns. These idiomatic patterns differ in four major respects: the underlying intuition behind the idiom, 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 [Lane 90]; having uniform descriptions of the available architectures should simplify this task. Popular architectural patterns include:

Pipeline

<i>System model</i>	mapping data streams to data streams
<i>Components</i>	filters (purely computational, local processing)
<i>Connectors</i>	data streams (ASCII data streams for unix pipelines)
<i>Control structure</i>	data flow

Data abstraction (object-oriented)

<i>System model</i>	localize state maintenance
<i>Components</i>	managers (e.g., servers, objects, abstract data types)
<i>Connectors</i>	procedure call (method invocation is essentially procedure call with dynamic binding)
<i>Control structure</i>	decentralized, usually single thread

Implicit invocation (event-based)

<i>System model</i>	independent reactive processes
<i>Components</i>	processes that signal significant events without knowing recipients of signals

<i>Connectors</i>	automatic invocation of processes that have registered interest in events
<i>Control structure</i>	decentralized; individual components are not aware of recipients of signal

Repository (includes databases and blackboard systems)

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

Interpreter

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

Main program and subroutines

<i>System model</i>	call and definition hierarchy
<i>Components</i>	procedures
<i>Connectors</i>	procedure calls
<i>Control structure</i>	single thread

Layered

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

Patterns for component packaging and interaction

Recognizing idioms for overall organization patterns is not enough. The patterns rely on different kinds of components, and those components interact in different ways. A full architectural description language must therefore distinguish among kinds of components and interactions. Yet the tools at the disposal of programmers are the procedure and data references that languages allow them to export from modules.

Conventional views of system composition do not acknowledge these differences. Some simply view systems as collections of compilation units that import and export names of procedures and data, using constructs of the underlying programming language for the interaction. Others choose one organizational pattern and support it exclusively; this often involves restricting components to a particular form and interactions to one specialized abstraction.

A major shortcoming of both conventional methodology and conventional languages is that people don't generally recognize the distinctions (or lack thereof) that are causing trouble. If a

software developer is not sensitive to differences in component packaging, he or she may inadvertently select incompatible components. Consider, for example, the difference in unix between filters and procedures (system calls). Many of the same functions are provide in both forms. Even though a formal specification might indicate that both versions computer the same function (e.g., sort), the two versions are not interchangeable because they interact with their data in different ways.

The situation is exacerbated because the connectors are invisible—the abstractions used for system style are not evident in the design but are hard-coded in procedure calls. Furthermore, component packaging comes in many types, which are not often discriminated. Even the copious work on reuse ignores the needs of systems that use multiple patterns, because it fails to deal explicitly with different forms of component packaging and different abstractions for component interaction (connection).

The common organization styles listed in the previous section distinguish among different kinds of components and connectors. This establishes a requirement that an architectural language support those distinctions by labeling, checking, and analysis. We [refs] have argued elsewhere that interactions should have the same first-class status that components do.

<i>ComponentType</i>	<i>Intuition</i>	<i>Player Types supported</i>
Module	Conventional compilation unit	RoutineDef, RoutineCall, GlobalDataDef, GlobalDataUse, ReadFile, WriteFile
Computation	Pure function	RoutineDef, RoutineCall, GlobalDataUse
SharedData	Fortran common with import	GlobalDataDef, GlobalDataUse
SeqFile	unix file	ReadNext, WriteNext
Filter	unix filter	StreamIn, StreamOut
Process	unix process	RPCDef, RPCCall
SchedProcess	real-time process	RPCDef, RPCCall, Segment, Trigger
General	anything goes	All (that is, any player type is allowed)

Table 1: Component types supported by UniCon

Unicon, our prototype architectural language, takes the first steps [Shaw et al 94]. *Components* and *connectors* are the primary entities in the language. Each has a type, a specification, and an implementation. Specifications include the elements of interaction—*players* for components and *roles* for connectors. The types of components currently supported are listed in Table 1, the types of connectors in Table 2. The specification of a connector, called a protocol, determines how components will interact. It establishes the roles that must be played by various components, and it identifies the player types that can fill these roles. Subsystems are then constructed as nonprimitive components by defining the components and connectors to be used and the manner in which they will be configured.

<i>ConnectorType</i>	<i>Intuition</i>	<i>Role Types and the Players they support</i>
Pipe	unix pipe	Source (accepts StreamOut of Filter, ReadNext of SeqFile) Sink (accepts StreamIn of Filter, WriteNext of SeqFile)
FileIO	unix operations between process and file	Reader (accepts ReadFile of Module) Readee (accepts ReadNext of SeqFile) Writer (accepts WriteFile of Module) Writee (accepts WriteNext of SeqFile)
ProcedureCall	inter-module procedure call	Definer (accepts RoutineDef of Computation or Module) Caller (accepts RoutineCall of Computation or Module)
DataAccess	shared data (between compilation units within a process)	Definer (accepts GlobalDataDef of SharedData or Module) User (accepts GlobalDataUse of SharedData, Computation, or Module)
RemoteProcCall	remote procedure call	Definer (accepts RPCDef of Process or SchedProcess) Caller (accepts RPCCall of Process or SchedProcess)
RTScheduler	processes competing for processor cycles	Stimulus (accepts Trigger of SchedProcess) Action (accepts Segment of SchedProcess)

Table 2: Connector types supported by UniCon

This is only a first step, as new types must be added manually, no abstraction capabilities are yet available, and the enforcement of patterns of architectural style is still quite weak.

3. Alexander's patterns

Alexander's pattern language [Alexander et al 77] has helped to shape my views on software architecture over the past few years. I discovered it originally when I was trying to understand the reuse of design fragments that couldn't be captured in subroutine or object form. These fragments included

- program skeletons to be fleshed out, often with varying numbers of certain features
- housekeeping activities such as scheduling or synchronization that seem to be impossible to isolate in their own modules
- configurations of modules without reference to the computations in the modules

It was clear that these fragments are reused idiomatically just as algorithms and data structures are. However, they must be fleshed out with other (unrelated) code in order to be useful, so ordinary library mechanisms are not suitable. Alexander's patterns are of precisely this form.

Other influences include the use of carefully-structured prose rather than formal specifications to express the patterns, the discrimination among a variety of patterns and restriction on which ones can interact with which others, and recognition that it's acceptable for some patterns to operate at large scale while others operate in the small.

A major attraction of Alexander's pattern language is that forms are modified when they are combined, and the patterns describe how these changes take place. This is in stark contrast to most programming language support, in which the parts are completely, rigidly defined in advance and maintain their identity throughout (except for inter-module optimization, which is invisible to designers).

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. This research was supported by the Carnegie Mellon University School of Computer Science and Software Engineering Institute (which is sponsored by the U.S. Department of Defense), 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. The US Government is authorized to reproduce and distribute reprints for Government purposes, notwithstanding any copyright notation thereon. 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 Wright Laboratory, the Department of Defense, the United States Government, Siemens Corporation, or Carnegie Mellon University.

References

Allen and Garlan 94

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

Alexander et al 77

Christopher Alexander, Sara Ishikawa, Murray Silverstein, et al. *A Pattern Language*. Oxford University Press 1977.

Booch 86

Grady Booch. Object-Oriented Development. *IEEE Trans. on Software Engineering* SE-12, 2, February 1986, pp. 211-221.

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 and Mary Shaw. An Introduction to Software Architecture. In V. Ambriola and G. Tortora (eds), *Advances in Software Engineering and Knowledge Engineering*, vol. II, World Scientific Publishing Company, 1993.

Lane 90

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

Linton 87

Mark A. Linton. Distributed Management of a Software Database. *IEEE Software*, vol 4 no 6, November 1987.

Paulk 85

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

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 94

Mary Shaw. Procedure Calls are the Assembly Language of Software Interconnection: Connectors Deserve First-Class Status. *Proc. Workshop on Studies of Software Design*, Springer-Verlag 1994, to appear.

Shaw and Garlan 93

Mary Shaw and David Garlan. Characteristics of Higher-level Languages for Software Architecture. Manuscript, 1993.

Shaw et al 94

Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young, Gregory Zelesnik. Abstractions for Software Architecture and Tools to Support Them. Soon to be a CMU technical report.

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 CMU-CS-87-129, June 1987.

