



Architectural Styles, Design Patterns, and Objects

ROBERT T. MONROE, ANDREW KOMPANEK, RALPH MELTON, and DAVID GARLAN
Carnegie Mellon University

Architectural styles, object-oriented design, and design patterns all hold promise as approaches that simplify software design and reuse by capturing and exploiting system design knowledge. This article explores the capabilities and roles of the various approaches, their strengths, and their limitations.

Software system builders increasingly recognize the importance of exploiting design knowledge in the engineering of new systems. Several distinct but related approaches hold promise.

One approach is to focus on the architectural level of system design—the gross structure of a system as a composition of interacting parts. Architectural designs illuminate such key issues as scaling and portability, the assignment of functionality to design elements, interaction protocols between elements, and global system properties such as processing rates, end-to-end capacities, and overall performance.¹ Architectural descriptions tend to be informal and idiosyncratic: box-and-line diagrams convey essential system structure, with accompanying prose explaining the meaning of the symbols. Nonetheless, they provide a critical staging point for determining whether a system can meet its essential requirements, and they guide implementers in constructing the system. More recently, architectural descriptions have been used for codifying and reusing design knowledge. Much of their power comes from use of idiomatic architectural terms, such as “client-server system,” “layered system,” or “blackboard organization.”



These convey widespread if informal understanding of the descriptions and let engineers quickly communicate

Each approach has something to offer: a collection of representational models and mechanisms.

their designs to others. Such architectural idioms represent what have been termed architectural styles.²

The object-oriented paradigm offers another approach to describing system designs. In its simplest form, object-oriented design lets us encapsulate data and behavior in discrete objects that provide explicit interfaces to other objects; groups of objects interact by passing messages among themselves. OOD has proven to be quite popular in practice, and sophisticated OOD methodologies offer significant leverage for designing software,²⁻³ including ease of decomposing a system into its constituent elements and partitioning system functionality and responsibility among those elements. However, it is not by itself well suited to describing complex interactions between groups of objects. Likewise, although individual objects can often be reused in other implementations, capturing and reusing common design idioms involving multiple objects can be difficult.

Design patterns have become an increasingly popular choice for addressing OOD's limitations. Although the principles underlying design patterns are not inherently tied to OOD, much recent work in this area has focused on design patterns for composing objects.^{4,5} Like architectural styles, design patterns provide guid-

ance for combining design elements in principled and proven ways.

Each of these often complementary approaches to capturing software design knowledge and software designs themselves has both benefits and drawbacks. To effectively use these approaches, we need to understand their terminologies, capabilities, similarities, and differences. Further, we need to understand the roles that each can play in successful software design.

WHAT IS SOFTWARE ARCHITECTURE DESIGN?

In practice, an architectural design fulfills two primary roles. First, it provides a level of abstraction at which software system designers can reason about system behavior: function, performance, reliability, and so on. By abstracting away from implementation details, a good architectural description makes a system design intellectually tractable and exposes the properties most crucial to its success. It is often the key technical document used to determine whether a proposed new system will meet its most critical requirements.

Second, an architectural design serves as the "conscience" for a system as it evolves. By characterizing the crucial system design assumptions, a good architectural design guides the process of system enhancement—indicating what aspects of the system can be easily changed without compromising system integrity. As with building blueprints, a well-documented architectural design makes explicit the software's "load-bearing walls,"⁶ a fact that helps not only at design time but also throughout a system's life cycle. To satisfy its multiple roles over time, an architectural description must be simple enough to permit system-level reasoning and prediction; practically speaking, it should fit on a page or two. Consequently, it is usually hierarchical: atomic architectural elements at one level of abstraction

are often described by a more detailed architecture at a lower level.

Architectural descriptions are primarily concerned with the following basic issues:

- ◆ *System structure.* Architectural descriptions characterize a system's structure in terms of high-level computational elements and their interactions. That is, an architecture frames its design solution as a configuration of interacting components. It is specifically not about requirements (for example, abstract relationships between elements of a problem domain) nor implementation details (such as algorithms or data structures).

- ◆ *Rich abstractions for interaction.* Interactions between architectural components—often drawn as connecting lines—provide a rich vocabulary for system designers. Although interactions may be as simple as procedure calls or shared data variables, they often represent more complex forms. Examples include pipes (with conventions for handling end-of-file and blocking), client-server interactions (with rules about initialization, finalization, and exception handling), event-broadcast connections (with multiple receivers), and database accessing protocols (with protocols for transaction invocation).

- ◆ *Global properties.* Architectural designs typically describe overall system behavior. Thus the problems they address are usually system-level ones, such as end-to-end data rates and latencies, resilience of one part of the system to failure in another, or system-wide propagation of changes when one part of a system is modified (such as changing the platform on which the system runs).

ARCHITECTURAL STYLE

As with any design activity, a central question is how to leverage past experience to produce better designs. In current practice, architectural designs have been codified and reused primari-

SOFTWARE ARCHITECTURE DESCRIPTION LANGUAGES

A variety of architectural design languages have been created to provide software architects with notations for specifying and reasoning about architectural designs. ADLs focus on various aspects of architectural design, and the analyses they support vary in flavor from rather informal to highly formal. Here are some examples:

- ◆ The UniCon system¹ focuses on compilation of architectural descriptions and modules into executable code.
- ◆ Rapide² emphasizes behavioral specification and the simulation of architectural designs.
- ◆ Wright³ provides a formal basis for specifying component interactions (via connectors) and architectural styles.
- ◆ The Aesop System⁴ supports the explicit encoding and use of a wide range of architectural styles.
- ◆ Various domain-specific software architecture languages⁵ support architectural specification tailored to a specific application domain.

In addition to the ADLs described above, which were developed specifically for describing software architectures, several more general formal specification languages have also been used. Examples include Z,⁶ Communicating Sequential Processes,⁷ and the Chemical Abstract Machine.⁸

The software architecture research community is realizing that these notations overlap considerably, particularly with respect to the structural aspects of a software architecture

specification. ACME is an emerging generic architecture description language that is designed to facilitate the interchange of architectural designs between different ADLs and toolsets.⁹

The notations used to express the architectural diagrams and style specifications in this article's examples reflect terminology and notations commonly found in these architecture description languages.

REFERENCES

1. M. Shaw et al., "Abstractions for Software Architecture and Tools to Support Them," *IEEE Trans. Software Eng.*, Apr. 1995, pp. 314-335.
2. D.C. Luckham et al., "Specification and Analysis of System Architecture using Rapide," *IEEE Trans. Software Eng.*, Apr. 1995, pp. 336-355.
3. R. Allen and D. Garlan, "Formalizing Architectural Connection," *Proc. 16th Int'l Conf. Software Eng.*, IEEE Computer Soc. Press, Los Alamitos, Calif., pp. 71-80.
4. D. Garlan, R. Allen, and J. Ockerbloom, "Exploiting Style in Architectural Design Environments," *Proc. SIGSOFT '94*, ACM Press, New York, 1994, pp. 179-185.
5. W. Tracz, "DSSA Frequently Asked Questions," *Software Eng. Notes*, Apr. 1994, pp. 52-56.
6. J.M. Spivey, *The Z Notation: A Reference Manual*, Prentice-Hall, Englewood Cliffs, N.J., 1989.
7. C.A.R. Hoare, *Communicating Sequential Processes*, Prentice-Hall, Englewood Cliffs, N.J., 1985.
8. P. Inverardi and A. Wolf, "Formal Specification and Analysis of Software Architectures Using the Chemical Abstract Machine Model," *IEEE Trans. Software Eng.*, Apr. 1995, pp. 373-386.
9. D. Garlan, R.T. Monroe, and D. Wile, "ACME: An Architecture Description and Interchange Language," tech. report, Carnegie Mellon Univ., Pittsburgh, 1996.

ly through informal transmission of architectural idioms. For example, a system architecture might be defined informally as a client-server system, a blackboard system, a pipeline, an interpreter, or a layered system. While these characterizations rarely have formal definitions, they convey much about a system's structure and underlying computational model.

An important class of architectural idioms constitutes what some researchers have termed architectural styles. An architectural style characterizes a family of systems that are related by shared structural and semantic properties.² An architectural style provides a specialized design language for a specific class of systems. Specifically, styles typically provide the following four things:

- ◆ A vocabulary of design elements: component and connector types such as pipes, filters, clients, servers, parsers, and databases.

- ◆ Design rules, or constraints, that determine which compositions of those elements are permitted. For example, the rules might prohibit cycles in a particular pipe-filter style, specify that a client-server organization must be an

n-to-one relationship, or define a specific compositional pattern such as a pipelined decomposition of a compiler.

- ◆ Semantic interpretation, whereby compositions of design elements, suitably constrained by the design rules, have well-defined meanings.

- ◆ Analyses that can be performed on systems built in that style. Examples include schedulability analysis for a style oriented toward real-time processing, and deadlock detection for client-server message passing. An important special case of analysis is system generation: many styles support application generators (for example, parser generators), or lead to reuse of a certain shared implementation base (such as user interface frameworks and support for communication between distributed processes).

The use of architectural styles has a number of significant benefits. First, it promotes design reuse: routine solutions with well-understood properties can be reapplied to new problems with confidence. Second, it can lead to significant code reuse: often the invariant aspects of an architectural style lend themselves to shared implementations. For example, systems described in a

pipe-filter style might reuse Unix operating system primitives to handle task scheduling, synchronization, and communication through pipes. Similarly, a client-server style can take advantage of existing RPC (remote procedure call) mechanisms and stub generation capabilities. Third, it is easier for others to understand a system's organization if conventionalized structures are used. For example, even without giving details, characterizing a system as a client-server organization immediately conveys a strong image of the kinds of pieces present and how they fit together. Fourth, use of standardized styles supports interoperability. Examples include CORBA object-oriented architectures, the OSI (Open Systems Interconnection) protocol stack, and event-based tool integration. Fifth, as we noted earlier, by constraining the design space, an architectural style often permits specialized, style-specific analyses. For example, we can analyze systems built in a pipe-filter style for throughput, latency, and freedom from deadlock, but this might not be meaningful for another system that uses a different style or an arbitrary, ad hoc architecture.

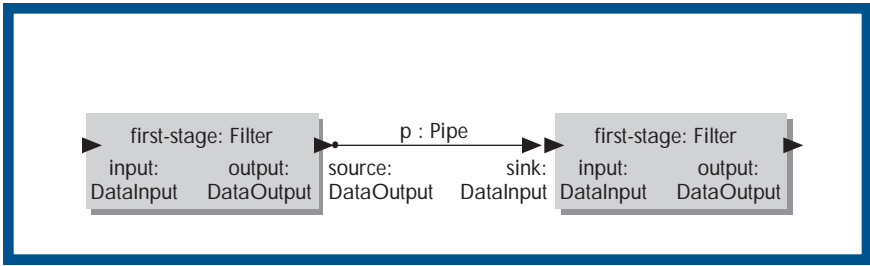


Figure 1. A simple system in the pipe-and-filter style is specified using an architectural notation.

```

Style pipe-and-filter
Interface Type DataInput = (read → (data?x → DataInput
                                [] end-of-data → close → √))
                                [] (close → √)
Interface Type DataOutput = write → DataOutput [] close → √

Connector Pipe
Role Source = DataOutput
Role Sink = DataInput
Glue = Buf<x>
where
  Buf<x> = Source.write?x → Buf<x> [] Source.close → Closed<x>
  Buf<S<x>> = Source.write?y → Buf<y>S<x>
              [] Source.close → Closed<S<x>>
              [] Sink.read → Sink.data!x → Buf<S<x>>
              [] Sink.close → Killed
  Closed<S<x>> = Sink.read → Sink.data!x → Closed<S<x>>
              [] Sink.close → √
  Closed<x> = Sink.read → Sink.end-of-data → Sink.close → √
  Killed = Source.write → Killed [] Source.close → √

Constraints
  ∀ c : Connectors • Type(c) = Pipe
  ∀ c : Components • Filter(c)
  where
    Filter(c:Component) = ∀ p : Ports(c) • Type(p) = DataInput
                        ∨ Type(p) = DataOutput

End Style

```

Figure 2. The system shown in Figure 1 is specified here using the Wright architecture description language.

OBJECT-ORIENTED DESIGN AND SOFTWARE ARCHITECTURE

The object-oriented design paradigm provides another abstraction for software design. In its simplest form, an OOD lets system designers encapsulate data and behavior in discrete objects that provide explicit interfaces to other objects. A message-passing abstraction is used as the glue that con-

nects the objects and defines the communication channels in a design. Although OOD concepts can be used to address some architectural design issues, and doing so is popular among software developers, there are significant differences between the capabilities and benefits of object-oriented approaches to design and the approaches provided by an emerging class of software architecture design

tools and notations. As the following examples illustrate, software architecture concepts allow an architect to describe multiple, rich interfaces to a component and to describe and encapsulate complex protocols of component interaction that are difficult to describe using traditional object-oriented concepts and notations.

To illustrate the different capabilities of style-based software architecture design and state-of-the-practice object-oriented design, consider the simple system presented in Figures 1 through 5. Figures 1 and 2 use common architectural notations (see the boxed text on architecture description languages on page 45) to present architectural views of the system. Figures 3 through 5 describe progressively more refined versions of the same system using the Object Modeling Technique OOD notation.³

In Figure 1, the system's architecture is described in a pipe-and-filter style that specifies the design vocabulary of components and connectors. In the pipe-and-filter style, all components are *filters* that transform a stream of data and provide specially typed input and output interfaces. All connectors in the style are *pipes* that describe a binary relationship between two filters and a data transfer protocol. Each pipe has two interfaces: a *source* that can only be attached to a filter's output interface, and a *sink* that can only be attached to a filter's input interface. Figure 2 provides a more formal definition of this style using the Wright notation.⁷ The Wright style specification describes the semantics of the design elements that can be used in the style (pipes and filters), along with a set of constraints that specify how the design elements can be composed when building systems in the pipe-and-filter style. There is a direct correlation between the graphical notation and the formal specification of the design elements. Each design element in the graphical depiction of the system is typed, and the type corresponds to the



type and protocol specifications given in the Wright specification. Thus, the graphical diagram actually has a firm semantic grounding for specification and analysis.

The sample system has two primary components, labeled stage 1 and stage 2, each of which transforms a data stream and then sends it to the next component downstream. The components interact via the pipe protocol specified in Figure 2. For simplicity, Figures 1 and 2 show only two transformations and ignore system input and output.

We can make three observations about this architectural design, especially with respect to the OMT-based design of the same system in Figures 3 through 5. First, the protocol of interaction between the filters is rich, explicit, and well specified. The Wright specification in Figure 2 is associated with the pipe connector between two filters (and with all connectors of type pipe). This specification defines the protocol for transmitting data through a pipe, the ordering behavior of the pipe, and the various interfaces that the pipe can provide to its attached filters. Because a primary focus of software architecture is to describe interactions among components, this capability is important. Second, both the components and connectors—filters and pipes in this style—have multiple, well-defined interfaces. As a result, a pipe can limit the services that it provides to the filters on each end. Likewise, a filter can specify whether each of its interfaces will provide input or output, as well as the type of data passing through. In this example, the upstream filter can only write to the pipe, and the downstream filter can only read from the pipe, preventing inappropriate access to connector functionality (such as the upstream pipe reading from the pipe). Finally, because there is a rich notion of connector semantics built into the style definition, we can evaluate the

design to determine emergent systemwide properties such as freedom from deadlock (provided that the system contains no cycles), throughput rates, and potential system bottlenecks.

In contrast to the stylized architectural design shown in Figures 1 and 2, Figures 3 through 5 present different OODs of the same system in progressively more sophisticated descriptions. The first OMT diagram, in Figure 1, provides a simple class diagram that says each filter may be associated with other filters by a pipe association. Each pipe association has a source and a sink role to indicate directionality. The instance diagram in Figure 1 depicts the example system using this class structure.

The association between the first-stage and second-stage filters is not truly a first-class entity like the Filter class and

is therefore not capable of supporting an explicit, sophisticated protocol description like the pipe in the architectural example. Rather, this is a generic association, implying that the upstream filter can invoke any public method of the downstream filter. Although objects can be sophisticated entities in the OMT paradigm, the vocabulary for determining interactions between objects is relatively impoverished for use in architectural descriptions.

Any object that can send a message to another object can request that the target object invoke any of its public methods. There is effectively a single, flat interface provided by all objects to all objects. As a result, it is difficult for an architectural object to limit the services it can provide based on which aspects of the interface a requester is

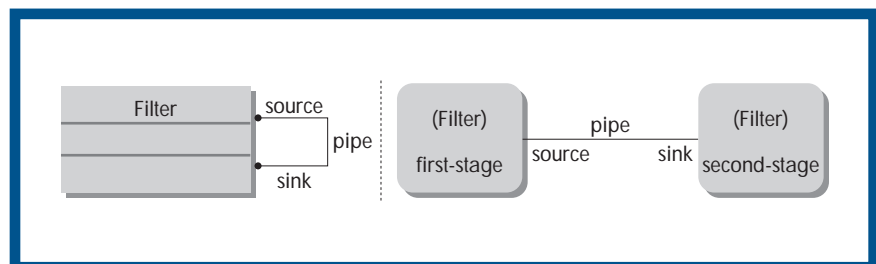


Figure 3. The same system shown in Figure 1 is depicted here using a naive object-oriented notation (OMT).

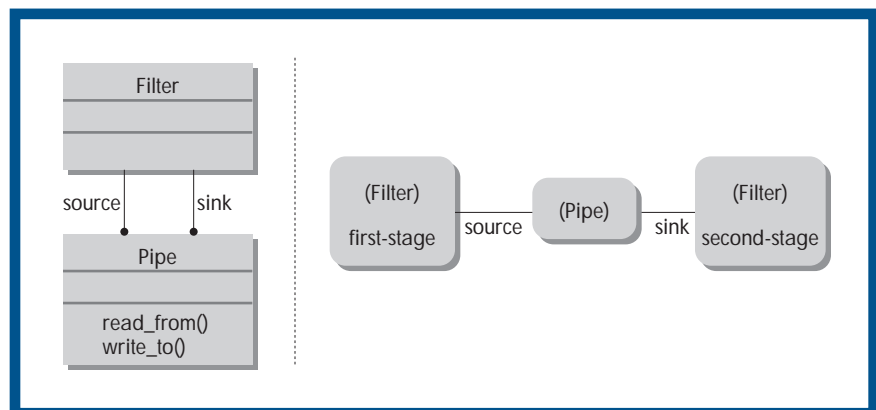


Figure 4. An OMT specification is used to define the same system architecture shown in Figure 1. Pipe is now a first-class design entity.



using and the type of connection between the two objects.

Finally, it is difficult to determine emergent system properties with an

The pattern approach lets us describe relatively complex protocols of interactions between objects.

impoverished vocabulary of connections and interface constraints. For example, the ability to invoke any method of an associated object at any time makes it difficult to determine dataflow characteristics and freedom from deadlock, which are both calculated relatively easily using the software architecture and architectural style constructs described earlier.

Figure 4 shows an attempt to address some of the issues raised by the design in Figure 3. It does so by making the pipe connector a first-class object. In this diagram, we use a pipe object to connect two filter objects. Using the OMT notation, it is now possible to add behavioral semantics to the Pipe class by associating dynamic and functional models with it. The Pipe class also introduces two new methods, `read_from()` and `write_to()`, that filters must call to send data on the pipe or read data from it.

One effect of placing a pipe entity between two filters is that the upstream filter no longer knows which downstream filter is receiving and processing its data. As a result, the upstream filter no longer has access to the downstream filter's methods. It can only access the pipe that connects them, ensuring a significant degree of independence from the downstream filter and transferring communication

responsibility to the pipe.

However, there is still a significant limitation to this design. Because the pipe object has to offer its full method interface to both of its attached filters, either filter can use the `write_to()` or `read_from()` methods. To maintain proper dataflow direction, however, we must be able to specify that the upstream filter, annotated by the source role, will use only the `write_to()` method, and that the downstream filter, annotated by the sink role, will use only the `read_from()` method. Unfortunately, the OMT notation does not let us formally specify these constraints. The directionality and well-defined pipe behavior are thus lost, along with the design analyses and assurances that go with them. It is certainly possible to create filters that abide by this protocol, but it is difficult to specify and enforce this constraint generally and explicitly using standard OOD notions.

Design patterns. An object-oriented approach to specifying an architectural pipe connector for use in pipe-and-filter style systems, along with rules for how a pipe can be properly instantiated in a design, apparently will require the cooperation of multiple objects. The emerging concept of *design patterns* addresses this issue.

Figure 5 presents a third and final revision of the simple pipe-and-filter architecture. This time, the pipe construct has been broken into three interacting objects:

- ◆ a pipe object controls dataflow and buffering,
- ◆ a source object attaches to the upstream filter and provides only a `write_to()` interface to the pipe, and
- ◆ a corresponding sink object attaches to the downstream filter and provides only a `read_from()` interface to the pipe.

This solution solves the problem of both filters having access to both `read_from()` and `write_to()` methods by

providing intermediary objects with limited interfaces.

By itself, however, this design does not completely mitigate the problem of access to inappropriate methods. It simply shifts the problem from the filter objects accessing inappropriate pipe methods to the source and sink objects improperly accessing pipe methods. Because the pipe, source, and sink methods are all encapsulated by the pipe-connector pattern, however, it is possible to describe a protocol by which the three objects agree to interact according to an appropriate pipe protocol; that is,

- ◆ the pipe object takes care of all queuing and buffering issues,
- ◆ only the source role may invoke the pipe's `enqueue_data()` method, and
- ◆ only the sink role may invoke the pipe's `dequeue_data()` method.

Further details of this protocol can also be encoded in the pattern and its objects.

The pattern approach lets us describe relatively complex protocols of interactions between objects that we want to encapsulate, but don't want to encapsulate within a single class. We could have described many of the constraints that the source and sink objects satisfy in the Filter class, but doing so would have added constraints to the class that may not be generally appropriate, and might have significantly decreased reusability. It would also have spread the interaction protocol among a wider variety of constructs, when we really want to be able to encapsulate it to clarify the design and ease the process of reasoning about the design. The need to use three different types of objects, interconnected with a pattern specification, significantly hinders the goal of simplicity. Although we could model a pipe connection using OMT and design patterns, much of the simplicity and elegance that came from specifying a simple type-annotated arrow with the architectural notation is lost when connectors are no longer first-class entities, as in the OOD paradigm.



Summary. As these examples illustrate, architectural designs involve abstractions that may not necessarily be best modeled as a system of objects, at least in the narrow sense of objects as encapsulated data types that interact through method invocation. This point is not limited to dataflow styles such as pipe-and-filter. We can easily make similar arguments about architectural design done in a layered style, a client-server-based style, a distributed-database style, or many other styles of architectural design.

Given that architectural styles can describe a broad range of different design families, it is tempting to view object-oriented design as a style of architectural design in which all components are objects and all connections are simple associations or aggregations (to use the OMT vocabulary). Indeed, it is possible to define object-based architectural styles that provide the typical primitive system construction facilities supported by many OOD toolsets. This view is quite reasonable for the subset of OOD that deals with architectural abstractions. There are, on the other hand, a number of design issues addressed directly by OOD that are generally considered outside the scope of architectural design. Examples include ways of modeling problem domains and requirements, and implementation issues such as designing data structures and algorithms. These concerns are relevant to software development and should probably be considered when a system architecture is being designed; it should not, however, be necessary to directly express and address all of them in an architectural description.

Architectural design is concerned with composing systems from components, and the interactions between these components. Such compositions provide an abstract view of a system, so that the designer can do system-level analyses and reason about system integrity constraints. Examples include throughput rates and freedom from deadlock. These distinctive aspects of architectural design highlight several

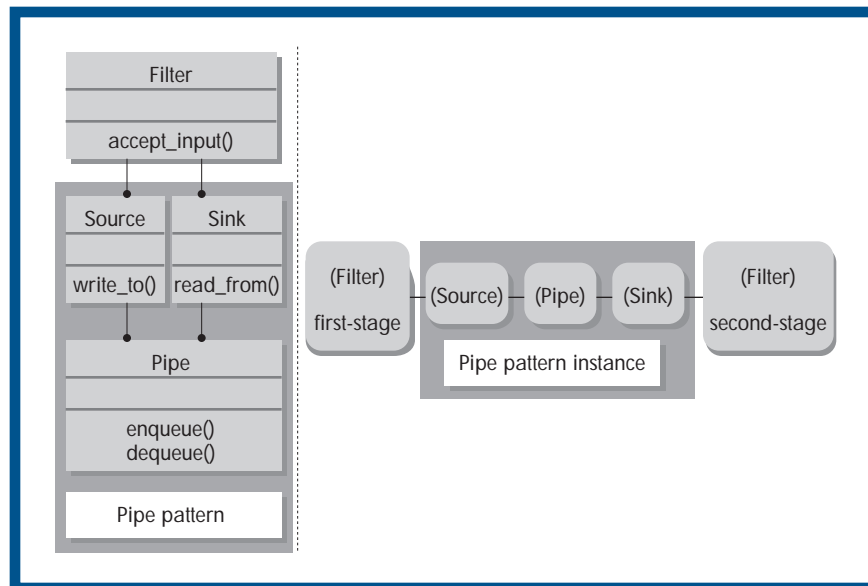


Figure 5. In this OMT-based specification of the system shown in Figure 1, the pipe connector is represented as a design pattern. Connector interfaces (source and sink) are now first-class entities.

important contrasts with object-oriented design. Although both are concerned with system structure in general, architectural design involves a richer collection of abstractions than is typically provided by OOD. These abstractions support the ability to describe new kinds of potentially complex system glue (or connectors). In addition to the pipe connector illustrated earlier, it is also possible to define n -ary connectors such as an event system, an RPC-based SQL query, or a two-phase-commit transaction protocol. Architectural abstractions also let a designer associate multiple interfaces with components and to express topological and other semantically based constraints over a design.




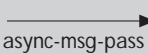
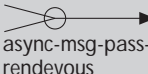
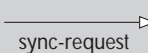
Thus neither architectural design nor object-oriented design subsumes the other. They are both appropriate at various times in the development process and they share some common notions and concepts. Just as you can specify an OO-based architectural style, you can use an OOD to implement or refine a sophisticated component or

connector in an architectural design. The fundamental issues that the two approaches address and the abstraction mechanisms that they provide, however, are not the same.

ARCHITECTURAL STYLES AND DESIGN PATTERNS

Two of the primary limitations of traditional OOD, as described in the previous examples, are the difficulty in specifying how groups of objects interact and in specifying and packaging related collections of objects for reuse. As Figure 5 shows, design patterns can mitigate these problems. The basic idea behind design patterns is that common idioms are found repeatedly in software designs and that these patterns should be made explicit, codified, and applied appropriately to similar problems. Several approaches to expressing these patterns have arisen over the past four or five years, most of which have focused on patterns for OOD.^{4,5} The utility of

Primitive vocabulary:

Primitive vocabulary	Informal description	Interface constraints	Properties
Components:		(ports define typed component interfaces)	
 Process	OS Process. Processes read input messages, send results to output interfaces.	at least 1 async-input port at least 1 async-output port at least 0 sync-callee ports	processing-cost, rate, input-message-type(s), output-message-type(s)
 Resource	Component for which processes contend.	exactly 0 async ports at least 1 sync-callee port	resource-cost
 Device	Send messages into the system at a predefined rate.	exactly 0 sync ports exactly 0 async-input ports at least 1 async-output port	output-rate, output-message-type
Connectors:		(roles define typed connector interfaces)	
 async-msg-pass	Asynchronous message channel for typed messages.	exactly 1 async-input role exactly 1 async-output role	message-type
 async-msg-pass-rendevous	Like async-msg-pass, but requires rendezvous before sending message. N-ary connector.	at least 1 async-input role exactly 1 async-output role	message-type
 sync-request	Binary synchronous request channel, typed messages.	exactly 1 sync-callee role exactly 1 sync-callee role	message-type

Design rules (list is a subset of all RTP/C style design rules):

- Async-msg-pass connectors may only connect (process, process) or (device, process) pairs of components.
- Sync-request connectors may only connect (process, resource) pairs of components.
- All processes must have an attached input interface.
- Each connector's input message type must match its output message type.
- ...

Style-based design analyses:

Analysis	Description
Message path typechecking	Insures only valid message types are passed along each message channel. Provides early detection of message type mismatch.
Rate calculation	Determines how often each process can be given control and resources.
Schedulability	Calculates whether this design could be scheduled on a uniprocessor with user-specified performance characteristics.
Repair heuristics	If the system cannot be scheduled, this analysis identifies bottlenecks and suggests likely repairs and improvements.

Figure 6. An informal specification of the Real-Time Producer/Consumer (RTP/C) style.

design patterns, however, extends beyond this. There are three fundamental requirements for specifying and reusing software design patterns: the design domain must be well understood, it must support the encapsulation of design elements, and it must have evolved a collection of well-known and proven design idioms. Pattern languages then let knowledgeable designers codify proven designs, design fragments, and frameworks for subsequent reuse.

Architectural styles relate closely to design patterns in two ways. First, architectural styles can be viewed as kinds of patterns⁸—or perhaps more accurately as pattern languages.⁹ Describing an architectural style as a

design pattern requires, however, a rather broad definition of the scope of design patterns. An architectural style is probably better thought of as a design language that provides architects with a vocabulary and framework with which they can build useful design patterns to solve specific problems—much as OMT provides a framework and notation for working with objects. Second, for a given style there may exist a set of idiomatic uses. These idioms act as microarchitectures, or architectural design patterns, designed to work within a specific architectural style. By providing a framework within which these patterns work, the designer using the pattern can leverage style's the broad

descriptive and analytical capabilities along with proven mechanisms for addressing specific design challenges in the form of design patterns.

We see patterns and architectural styles as complementary mechanisms for encapsulating design expertise. An architectural style provides a collection of building-block design elements, rules and constraints for composing the building blocks, and tools for analyzing and manipulating designs created in the style. Styles generally provide guidance and analysis for building a broad class of architectures in a specific domain, whereas patterns focus on solving smaller, more specific problems within a given style (or perhaps multiple styles).

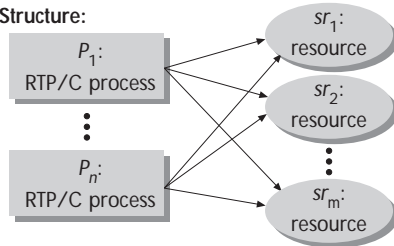
Shared-resource architectural pattern

Intent: Avoid deadlock when processes share common resources.

Motivation: System deadlock can occur when architectural components lock shared resources in an inappropriate order.

Applicability: Architectural designs done in the RTP/C style, where process components share resource components and freedom from deadlock is more important than run-time performance.

Structure:



Participants: N RTP/C process components, each connected to m or fewer RTP/C resource components. All connectors used are RTP/C sync-request connectors from processors to resources.

Collaborations: In order to avoid deadlock, a process P_k can only send a request on resources sr_i (locking sr_j) if $i > j$, where s_j is the highest numbered resource currently held by P_k .

Consequences: Using the ordered access protocol to prevent deadlock will not generally lead to optimal resource access or allocation. Other protocols may lead to better average-case performance.

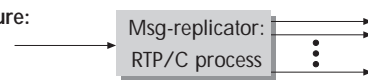
Message-Replicator architectural pattern

Intent: Send identical messages to a dynamically changing group of other components using a principled protocol.

Motivation: A component's output may need to be sent to a variable set of components. The set of receiving components may change as the system runs, and constraints on the order in which recipients receive the messages may be important (as in the case of a stock-quote and trading system).

Applicability: Architectural designs done in the RTP/C style where the set of applicable recipients for the output of a specific component may vary as the system runs.

Structure:



Participants: The Msg-replicator process is an RTP/C process component with a single input port and a variable array of output ports. There is a single async-msg-pass connector providing input and a set of async-msg-pass connectors that send output to the recipients.

Collaborations: When this pattern is instantiated the designer needs to select a protocol by which the messages will be sent to the outputs. Options include *sequentially*, whereby messages are written in a user-specified order to each output connector one at a time; *parallel*, whereby all messages are written to their output connectors concurrently; or a user-specified variation on one of these.

Consequences: The dynamic nature of this pattern can make some static analyses, such as dataflows and delivery guarantees, difficult or impossible to perform.

Figure 7. Two sample architectural design patterns in the RTP/C style.

It is also important to note that patterns need not be architectural. Indeed, many patterns in recent handbooks^{4,5} deal with solutions to lower-level programming mechanisms, rather than system-structuring issues.

Pattern and style examples. To illustrate the scope and purpose of architectural styles, as well as how they relate to design patterns, consider the architectural style specification given in Figure 6. This style, described as the Real-Time Producer/Consumer style, is designed to assist architects putting together real-time multimedia systems running on uniprocessor computers.¹⁰ Figure 6 provides an informal description of the RTP/C style, emphasizing the types of (primitive) design vocabulary used by designs constructed in the style, design rules and constraints that specify how the elements may be composed, and analyses that can be performed on the design. The RTP/C style definition describes a set of primitive building blocks and guidelines for putting together a fairly broad range of systems within a reasonably well understood domain.

Even with such a well-defined style, however, relatively concrete design patterns play an important role. The

RTP/C primitive design elements and guidelines form a language that can be used to capture more detailed, concrete solutions to specific problems. This style provides a well-understood and well-defined vocabulary framework for composing individual design elements in principled ways that support real-time analyses. Figure 7 shows two simplified design patterns done in the RTP/C style—the forked-memory pattern and the message-replicator pattern. Along with a diagram, each pattern provides information describing its applicability, consequences of use, and so on. We have shown these patterns using the structure provided in a 1995 book by Erich Gamma and his colleagues.⁴ This framework works well for architectural patterns as well as for OO patterns, with the primary difference being that architectural patterns address a more specific set of design issues (as described earlier under “What is software architecture?”) than do OO patterns. Just as OMT and objects are used to show the design patterns in most OOD patterns handbooks, the vocabulary and rules of architectural style can be used to specify architectural design patterns.

It follows, then, that OMT and the design patterns notations from the OOD patterns handbooks can be used

to specify architectural patterns also. In fact, several of the design patterns that Gamma and his colleagues describe appear to apply to architectural design.⁸ Examples include the Facade pattern that provides a single interface to a collection of objects, the Observer pattern that specifies a mechanism for maintaining consistency among objects (or components), and the Strategy pattern that specifies how to separate algorithmic choices from interface decisions. None of the listed patterns are limited to being only architectural patterns. All have applicability at lower levels of design (such as detailed design or implementation code). In addition to the architectural patterns listed here, several patterns in the Gamma et al. book, for example, fail to address architectural issues. The Factory Method and Flyweight patterns. Both of these patterns, for instance, deal with lower-level implementation issues than architectures generally specify.

Thus, architectural design patterns and object-oriented design patterns are simply instances of the more general class of all design patterns. Unlike design patterns proper, however, an architectural style provides a language and framework for describing families of well-formed software architectures.



The role of style is to provide a language for expressing both architectural instances and patterns of common architectural design idioms. As a result, the constructs and concepts underlying architectural style are comparable to those underlying an OOD methodology like OMT, rather than a set of

design patterns such as those given by Gamma and his colleagues.⁴ A specific architectural style is better thought of as a language for building patterns than as an instance of a design pattern itself.

Architectures, architectural styles, objects, and design patterns cap-

ture complementary aspects of software design. Although the issues and aspects of software design addressed by these four approaches overlap somewhat, none completely subsumes the other. Each has something to offer in the way of a collection of representational models and mechanisms. ♦

ACKNOWLEDGMENTS

We thank Robert Allen for his helpful comments. This research was sponsored by the National Science Foundation under grant no. CCR-9357792 and a graduate research fellowship; by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF; by the Advanced Research Projects Agency under grant no. F33615-93-1-1330; and by Siemens Corporate Research.

REFERENCES

1. M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice-Hall, Englewood Cliffs, N.J., 1996.
2. G. Abowd, R. Allen, and D. Garlan, "Using Style to Give Meaning to Software Architecture," *Proc. SIGSOFT '93: Foundations Software Eng.*, ACM, New York, 1993. Also in *Software Eng. Notes*, Dec. 1993, pp. 9-20.
3. J. Rumbaugh et al., *Object-Oriented Modeling and Design*, Prentice-Hall, Englewood Cliffs, N.J., 1991.
4. E. Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Design*, Addison-Wesley, Reading, Mass., 1995.
5. W. Pree, *Design Patterns for Object-Oriented Software Development*, Addison-Wesley, Reading, Mass., 1995.
6. D. Perry and A. Wolf, "Foundations for the Study of Software Architecture," *ACM Software Eng. Notes*, Vol. 17, No. 4, Oct. 1992, pp. 40-52.
7. R. Allen and D. Garlan, "Formalizing Architectural Connection," *Proc. 16th Int'l Conf. Software Eng.*, IEEE Computer Soc. Press, Los Alamitos, Calif., 1994, pp. 71-80.
8. M. Shaw, "Some Patterns for Software Architecture," in *Pattern Languages of Program Design, Vol. 2*, J. Vlissides, J. Coplien, and N. Kerth, eds., Addison-Wesley, Reading, Mass., 1996, pp. 255-269.
9. N.L. Kerth, "Caterpillar's Fate: A Pattern Language for Transformations from Analysis to Design," in *Pattern Languages of Program Design*, J.O. Coplien and D.C. Schmidt, eds., Addison-Wesley, Reading, Mass., 1995.
10. K. Jeffay, "The Real-Time Producer/Consumer Paradigm: A Paradigm for the Construction of Efficient, Predictable Real-Time Systems," *Proc. 1993 ACM/SIGAPP Symp. Applied Computing*, ACM Press, New York, 1993, pp. 796-804.



Robert T. Monroe is a doctoral candidate in the Department of Computer Science at Carnegie Mellon University. He holds an MS in computer science from Carnegie Mellon and a BS from the University of Michigan. His research interests include software design tools, software architecture, and languages for expressing software design expertise. He is a member of the IEEE Computer Society and ACM.



Ralph Melton is a graduate student in the Department of Computer Science at Carnegie Mellon University. He holds a BS from Stanford University. His research interests include software architecture and the use of formal methods to describe design fragments and their composition.



Andrew Kompanek is a research programmer with the ABLE research group in Carnegie Mellon University's School of Computer Science. He recently received his BS in mathematics and computer science from Carnegie Mellon. His current work includes the design and development of visualization and automated layout tools for software architectures.



David Garlan is associate professor of computer science at Carnegie Mellon University, where he heads the ABLE Project. His research focuses on software architecture, the application of formal methods to the construction of reusable designs, and software development environments. He completed his PhD at Carnegie Mellon, and holds a BA from Amherst College and an MA from the University of Oxford, England. He is member of the IEEE Computer Society and ACM.

Address questions about this article to Robert Monroe, School of Computer Science, Carnegie Mellon University, 5000 Forbes Ave., Pittsburgh, PA 15213; bmonroe+@cs.cmu.edu.