# Stylized Architecture, Design Patterns, and Objects

**Robert T. Monroe, Drew Kompanek, Ralph Melton, and David Garlan**

***Abstract***

Software system builders are increasingly recognizing the importance of exploiting design knowledge in the engineering of new systems. One way to do this is to define an architectural style for a collection of related systems. The style determines a coherent vocabulary of system design elements and rules for their composition. By structuring the design space for a family of related systems a style can, in principle, drastically simplify the process of building a system, reduce costs of implementation through reusable infrastructure, and improve system integrity through style-specific analyses and checks.

Like architectural style, object-oriented design patterns attempt to capture and exploit design knowledge to ease the process of designing software systems and reusing proven designs. There are, however, significant differences in the roles and capabilities of architectural styles and object-oriented design patterns, as there are between architectural design and object-oriented design. In this paper we illustrate the relationship between software architecture and object-oriented design, as well as the relationship between architectural styles and design patterns. We begin by defining our terms and then proceed to compare and contrast the various approaches with examples.

This research was sponsored by the National Science Foundation under Grant Number CCR-9357792, by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA) under grant number F33615-93-1-1330,and by Siemens Corporate Research. The 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 U.S.~Government, or Siemens Corporation. The U.S.~Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon.

# 1 Introduction

A critical challenge for software engineering is to capture system designs and reuse established design knowledge when building new systems. One promising approach is to focus on the architectural level of system design. This level of design is concerned with the gross structure of a system as a composition of interacting parts. Key issues illuminated by an architectural design are the assignment of functionality to design elements, protocols of interaction between the elements, scaling and portability, and global system properties such as rates of processing, end-to-end capacities, and overall performance [SG96].

Architectural descriptions have long been recognized as an essential ingredient of a well-designed system. These descriptions tend to be informal and idiosyncratic, consisting of box-and-line diagrams that convey the essential system structure, together with prose that explains the meaning of the symbols. Nonetheless, they serve as a key component in the development of most systems, both by providing a critical staging point for determining whether a system can meet its essential requirements and by guiding implementors in system construction.

More recently architectural descriptions are being recognized as an appropriate vehicle for codification and reuse of design knowledge. Much of the power of architectural descriptions 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 allow engineers to quickly communicate their designs to others.

In many cases, such as those mentioned above, architectural idioms represent what has been termed an *architectural style* [AAG93]. As we describe later, an architectural style provides a design vocabulary (clients and servers, pipes and filters, blackboards and knowledge sources, etc.) together with rules for how elements in that vocabulary can be combined. In essence, a style provides a specialized design language for a specific class of systems. Principled use of architectural styles promises many benefits, including simplifying the process of designing and implementing a system by focusing the design space, reducing implementation costs through reusable infrastructure, and improving system integrity with style-specific analyses and checks.

**Object-oriented design and design patterns**

Another approach to describing system designs is to use an object-oriented paradigm. In its most simple form, object-oriented design provides a way for system designers to encapsulate data and behavior in discrete *objects* that provide explicit interfaces to other objects. Groups of objects interact by passing messages amongst themselves. Object-oriented design (OOD) has proven to be quite popular in practice, and sophisticated OOD methodologies provide significant leverage for designing software [i.e. Rum+91, SM92]. There is growing recognition, however, that although OOD provides a number of benefits, including ease of decomposing and partitioning system elements, functionality, and responsibility, it is not, by itself, particularly well suited to describing complex interactions between groups of objects. Likewise, although individual objects are often quite reusable at an implementation level, it can be difficult to capture and reuse common design idioms that involve multiple objects.

An increasingly popular approach to addressing these limitations of object-oriented design is to make use of design patterns. Although the principles underlying design patterns are not inherently tied to object-oriented design, much of the recent work in the area has focused on design patterns for composing objects [GHJV95, Pree95]. Like architectural styles, design patterns provide guidance for combining design elements in principled and proven ways.

In this paper we illustrate the relationship between software architecture and object-oriented design, as well as the relationship between architectural styles and design patterns. We begin by defining our terms and then proceed to compare and contrast the various approaches with examples.

# 2 What is Software Architecture?

The architectural design of a software system is concerned with its gross structure and the ways in which that structure leads to the satisfaction of key system properties. In practice an architectural design fulfills two primary roles. First, architectural design provides a level of design abstraction at which software system designers can reason about overall system behavior: function, performance, reliability, etc. By abstracting away from implementation details, a good architectural description makes a system intellectually tractable, exposing the properties that are most crucial to its success. In this way an architectural design is often the key technical document used to determine whether a proposed new system will be able to meet its most critical requirements.

Second, an architectural design serves as the "conscience" for a system as it evolves over time. 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. By analogy to building blueprints, an architectural design makes explicit the "load-bearing walls" [PW92] of the software. Thus a well-documented architecture helps not only at design time, but also throughout the lifecycle of a system. To satisfy its multiple roles throughout the software lifecycle, an architectural description must be simple enough to permit system-level reasoning and prediction. Practically speaking, this limits architectural descriptions to a level of detail that can fit on a page or two. Consequently, to represent architectural designs, descriptions are usually hierarchical: atomic architectural elements at one level of abstraction are described by a more detailed architecture at a lower level.

Architectural descriptions are primarily concerned with the following four basic issues:

1) **System structure:** Architectural descriptions characterize a system's structure in terms of high-level computational elements and their interactions. That is, an architecture is concerned with the nature of a solution to a problem as a configuration of interacting components. It is specifically *not* about representing requirements (e.g., abstract relationships between elements of a problem domain), nor the details of implementations (e.g, algorithms and data structures).
2) **Rich abstractions for interaction:** Interactions between architectural components – often drawn as connecting lines – reflect a rich vocabulary for system designers. While interactions may be as simple as procedure calls or shared data variables, often they represent more complex forms of communication and coordination. 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).

3) **Global properties:** Since a principal use of an architectural design is to reason about overall system behavior, architectural designs are typically concerned with the entire system. The problems addressed by an architecture are usually system-level ones, such as end-to-end data rates and latencies, resilience of one part of a system to failure in other parts, or system-wide propagation of changes when one part of a system is modified (such as changing the platform on which the system runs).

# 3 Architectural Style

As with any design activity, a central question is how to leverage past experience to produce better designs. In current practice, the codification and reuse of architectural designs has occurred primarily through informal transmission of architectural idioms. For example, a system might be defined architecturally 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 the structure and underlying computational model of a system.

An important class of architectural idioms constitute what some researchers have termed "architectural styles." An architectural style characterizes a family of systems that are related by shared structural and semantic properties [AAG93]. More specifically, styles typically provide four things:

1) **A Vocabulary** of design elements – component and connector types such as pipes, filters, clients, servers, parsers, databases, etc.

2) **Design rules** – or constraints – that determine the permitted compositions of those elements. 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.

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

4) **Analyses** that can be performed on systems built in that style. Examples include schedulability analysis for a style oriented toward real-time processing [Ves94] and deadlock detection for client-server message passing [JC94]. A specific, but important, special case of analysis is system generation: many styles support application generators (e.g., parser generators), or lead to reuse of certain shared implementation base (e.g., user interface frameworks or 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, use of architectural styles 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 mechanisms and stub generation capability. Third, it is easier for others to understand a system's organization if conventionalized structures are used. For example, even without giving details, characterization of a system as a “client-server” organization immediately conveys a strong image of the kinds of pieces and how they fit together. Fourth, use of standardized styles supports interoperability. Examples include CORBA object-oriented architectures, the OSI 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, it is possible to analyze systems built in a pipe-filter style for throughput, latency, and deadlock-freedom. Such analyses might not be meaningful for an arbitrary, ad hoc architecture – or even one constructed in a different style. In particular, some styles make it possible to generate code directly from an architectural description. Sixth, it is usually possible (and desirable) to provide style-specific graphical depictions of design elements. This makes it possible to provide graphical renderings of designs that match engineers' domain-specific intuitions about how their designs should be visualized.

# 4 Object-Oriented Design and Software Architecture

The object-oriented design paradigm provides another abstraction for designing software. In its simplest form, an object-oriented design allows system designers to encapsulate data and behavior in discrete *objects* that provide explicit interfaces to other objects. A message passing abstraction is used as the glue that connects the objects and defines the communication channels in a design. Although object-oriented design 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 examples that follow 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.

## 4.1 Examples

To illustrate the different capabilities of style-based software architecture design and state-of-the-practice object-oriented design, consider the case of the simple system presented in figures 1-5. Figures 1 and 2 use common architectural notations (see ADL breakout box) to present an architectural view of the system. Figures 3-5 describe progressively more refined versions of the same system using the popular Object Modeling Technique (OMT) object-oriented design notation [Rum+91].

Figure 1 presents an architectural view of the system. The architecture for this system 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 the pipe-and-filter style using the Wright notation [AG96], which is based on the CSP notation and calculus. 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 of the design elements in the graphical depiction of the system are 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. For a more detailed description of the pipe and filter style please see [AG94].

The example system has two primary components (labelled as stage 1 and stage 2), each of which perform a transformation on a stream of data and then send the transformed data to the next component downstream. The components interact via the "pipe" protocol specified in figure 2.

**Software Architecture Description Languages (breakout box)**

A variety of architectural design languages (ADL's) have been created to provide software architects with notations for specifying and reasoning about architectural designs. These ADL's focus on various aspects of architectural design and analysis and vary in flavor from rather informal to highly formal. For example, the UniCon system [SDK+95] focuses on compilation of architectural descriptions and modules into executable code; Rapide [LAK+95] emphasizes behavioral specification and the simulation of architectural designs; the Aesop System [GAO94] supports the explicit encoding and use of a wide range of architectural styles; and various Domain Specific Software Architecture languages support architectural specification tailored to a specific application domain [Tra94]. In addition to the ADLs described above, which were developed specifically for describing software architectures, a number of more general formal specification languages have been used to describe software architectures. Examples include Z [Spiv89], Communicating Sequential Processes (CSP) [Hoa85], and the Chemical Abstract Machine [IW95].

There is an emerging realization within the software architecture research community of the considerable overlap between these notations, particularly with respect to the structural aspects of a software architecture specification. The notations used to express the architectural diagrams and style specification in the examples of sections 4.2 and 5.1 reflect terminology and notations commonly found in these architecture description languages.


**[GAO94]** David Garlan, Robert Allen, and John Ockerbloom. Exploiting style in architectural design environments, In *Proceedings of SIGSOFT '94*, pages 179-185, ACM Press, December 1994.

**[Hoa85]** C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

**[IW95]** Paola Inverardi and Alexander Wolf. Formal specification and analysis of software architectures using the chemical abstract machine model, *IEEE Transactions on Software Engineering*, April 1995.

**[LAK+95]** David C. Luckham, Larry M. Augustin, John J. Kenney, James Veera, Doug Bryan, and Walter Mann. Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering*, April 1995.

**[SDK+95]** Mary Shaw, Robert Deline, Daniel Klein, Theodore Ross, David Young, and Gregory Zelesnik. Abstractions for software architecture and tools to support them. *IEEE Transactions on Software Engineering*, April 1995.

**[Spiv89]** J.M. Spivey. *The Z notation : a reference manual*, Prentice Hall, 1989

**[Tra94]** Will Tracz. DSSA frequently asked questions. *Software Engineering Notes*, 19(2):52-56, April 1994.

For simplicity, the figures show only two transformations and system input and output are ignored.

Three interesting observations about this architectural design, especially with respect to the OMT-based design of the same system in figures 3-5, follow: First, the protocol of interaction between the filters is rich, explicit, and well-specified. The Wright specification of figure 2 is associated with the pipe connector between the two filters (and, in fact, 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 describing the interactions amongst components this capability is very 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 provide 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 (e.g. the upstream pipe reading from the pipe). Finally, because there is a rich notion of connector semantics built into the style definition, it is possible to evaluate the design to determine emergent system-wide properties such as deadlock-freedom (provided that the system contains no cycles), throughput rates, and potential system bottlenecks.

As a contrast to the stylized architectural design presented in figures 1 and 2, let us now consider the three object-oriented designs of the same system presented in figures 3-5. These diagrams present progressively more sophisticated descriptions of the system. The first OMT diagram, given 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.

There are a few important points to notice about this design. The first is that 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 that implies the upstream filter is capable of invoking any public method of the downstream filter. Although objects can be sophisticated entities in this object-oriented design paradigm (OMT), the connections between the objects are relatively impoverished for use in architectural descriptions.

A second important point is that 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 that it can provide based on which aspects of the interface a requestor is using and the type of connection between the two objects. Finally, it is difficult to determine emergent system properties with an impoverished vocabulary of connections and interface constraints. For example, the ability to invoke any method of an associated object at any time makes it very difficult to determine the dataflow characteristics and deadlock-freedom that are relatively easily calculated using the software architecture and architectural style constructs described earlier.

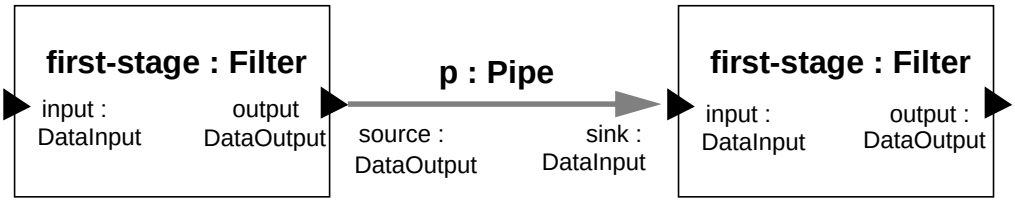



**Figure 1: Architectural description of a simple system described using the *pipe-and-filter* style.**

---

**Style** pipe-and-filter

**Interface Type** DataInput = ($\overline{\text{read}}$ Æ→ (data?x → DataInput
[] end-of-data → $\overline{\text{close}}$ → √))
⌈⌉ ($\overline{\text{close}}$ → √)

**Interface Type** DataOutput = $\overline{\text{write}}$ → DataOutput ⌈⌉ $\overline{\text{close}}$ → √

**Connector** Pipe
**Role** Source = DataOutput
**Role** Sink = DataInput
**Glue** = $\text{Buf}_{<>}$

where
$\text{Buf}_{<>}$ = Source.write?x → $\text{Buf}_{<x>}$ [] Source.close → $\text{Closed}_{<>}$
$\text{Buf}_{s<x>}$ = Source.write?y → $\text{Buf}_{<y>s<x>}$
[] Source.close → $\text{Closed}_{s<x>}$
[] Sink.read → $\overline{\text{Sink.data!x}}$ → $\text{Buf}_{s}$
[] Sink.close → Killed
$\text{Closed}_{s<x>}$ = Sink.read → $\overline{\text{Sink.data!x}}$ → $\text{Closed}_{s}$
[] Sink.close → √
$\text{Closed}_{<>}$ = Sink.read → $\overline{\text{Sink.end-of-data}}$ → 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: Specification of the pipe-and-filter style using the Wright architecture description language [AG96].**

Figure 4 attempts to address some of the issues raised by the Figure 3 design by making the pipe connector a first-class object. In this diagram we have introduced a *pipe* class that is used to connect two filters. 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 need to call in order 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. Unfortunately, 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, both of the filters can make arbitrary use of either the write_to() or read_from() methods. In order to maintain proper directionality of dataflow, however, we want to be able to specify that the upstream filter, annotated by the *source* role, will only make use of the write_to() method and the downstream filter, annotated by the *sink* role, will only make use of the read_from() method. The directionality and well defined pipe behavior is thus lost, along with the design analyses and assurances that go with it. 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 object-oriented design notions.

It appears that 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, 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 presented in figures 3 and 4. This time, the pipe construct has been broken into three interacting objects – a *pipe* object that controls dataflow and buffering, a *source* object that attaches to the upstream filter and provides only a write_to() interface to the pipe, and a corresponding *sink* object that attaches to the downstream filter and provides only a read_from() interface to the pipe. This solution addresses the problem of both filters having access to both read_from() and write_to() methods of the pipe 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 – i.e. only the source role may invoke the pipe's enqueue_data() method, only the sink role may invoke the pipe's dequeue_data() method, and the pipe object takes care of all queuing and buffering issues. There are further details to this protocol that can also be encoded in the pattern and its objects.

The advantage to the pattern approach here is that it allows us to 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

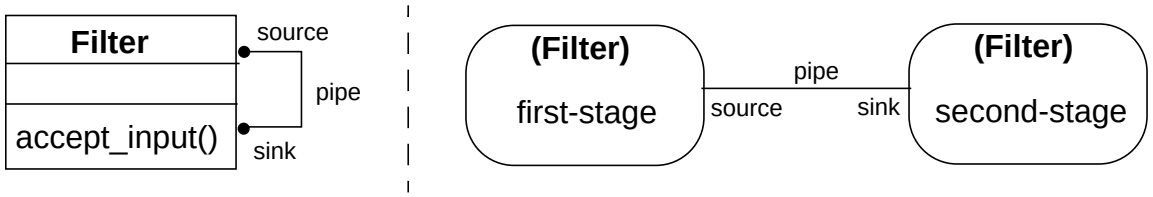



**Figure 3: Naive object-oriented design (OMT) specification of simple pipe-and-filter system architecture.**

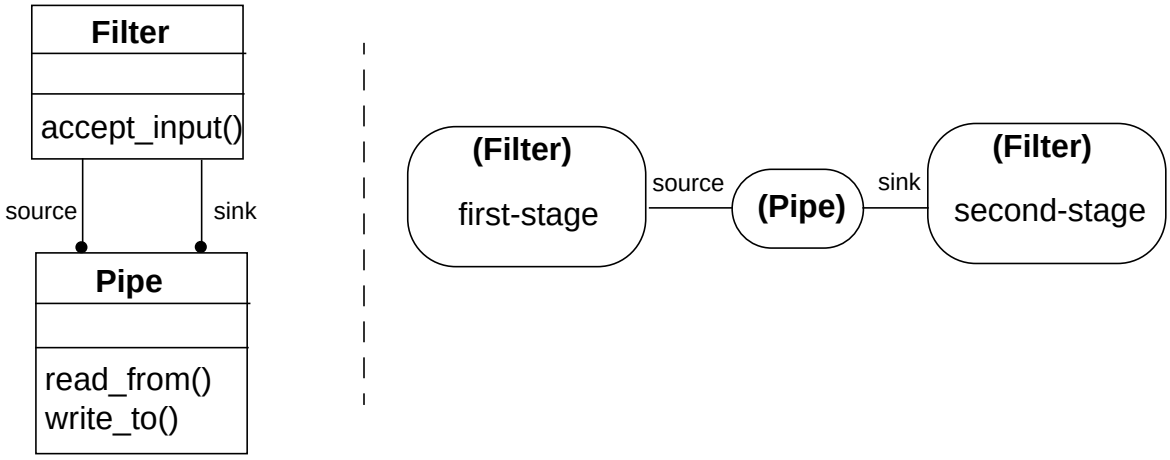



**Figure 4: OMT specification of simple pipe-and-filter system architecture. Pipe is now a first-class design entity.**

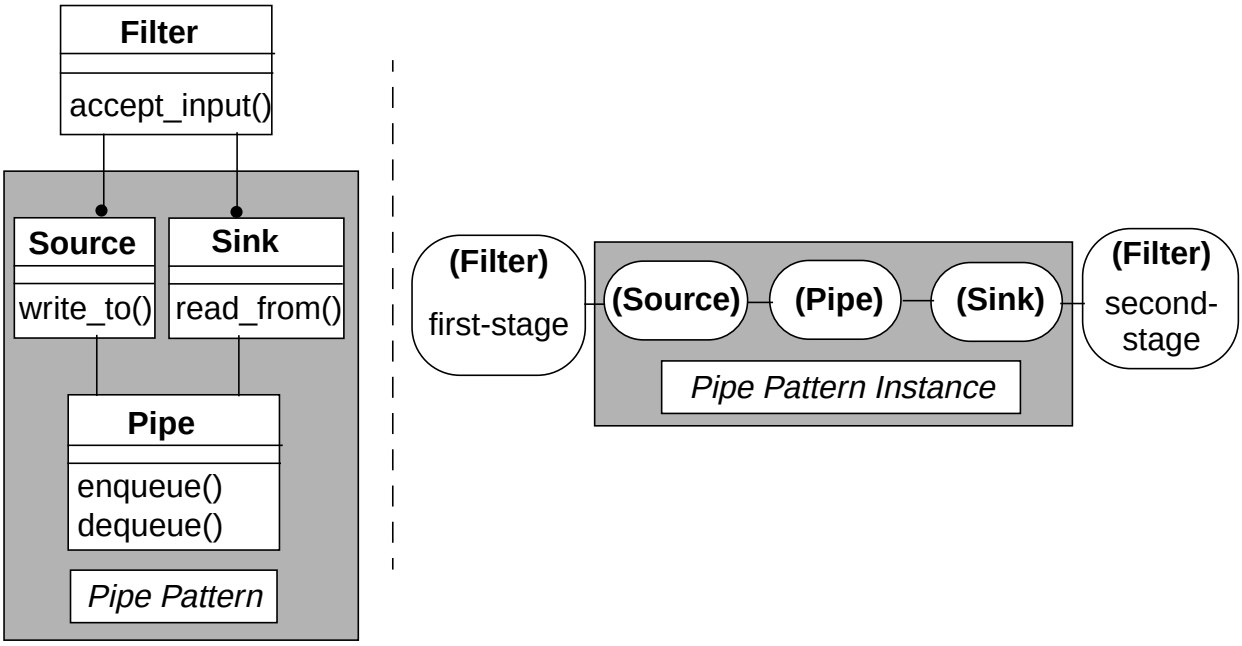



**Figure 5: OMT-based specification of simple pipe-and-filter system architecture. The pipe connector is represented as a design pattern. Connector interfaces (source and sink) are now first-class entities.**

sink objects satisfy in the Filter class, but doing so would have added constraints to the filter class that may not be generally appropriate, and may significantly decrease 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, connected together with a pattern specification significantly hinders the goal of simplicity. Although it turns out to be possible to 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 object-oriented design paradigm.

## 4.2 Discussion

The primary point that these examples illustrate is that 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 datatypes 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 object-oriented design (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 that OOD addresses directly although they 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 the development of software, and should probably be taken into consideration when developing an architecture for a system, it should not, however, be necessary to directly express and address all of them in an architectural description.

Architectural design is concerned with compositions of systems out of components and the interactions between them. These compositions provide an abstract view of a system that permits system level analyses and allows a designer to reason about system integrity constraints. Examples include deadlock-freedom and throughput rates. These distinctive aspects of architectural design highlight a number of important constrasts with object-oriented design. While both are concerned with the structure of a system, 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 in the examples 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 allow a designer to 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 it is possible to specify an OO-based architectural style, it is possible to use an object-oriented design 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.

# 5 Architectural Styles and Design Patterns

Two of the primary limitations of traditional object-oriented design, as discussed in the previous examples, are the difficulty involved in specifying how groups of objects interact and in specifying and packaging related collections of objects for reuse. As the example in Figure 5 illustrated, design patterns are one approach to mitigating 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. A number of notations for expressing these patterns have arisen over the past four or five years, most of which have focused on patterns for object-oriented design [e.g. GHJV95, Pree95]. The utility of design patterns, however, extends beyond object-oriented design. The fundamental requirements for specifying and reusing design patterns for software design appear to be that the design domain is well understood, supports the encapsulation of design elements, and has evolved a collection of well known and proven design idioms. Pattern languages then allow knowledgeable designers to codify proven designs, design fragments, and frameworks for subsequent reuse.

Architectural styles are closely related to design patterns in two ways. First, architectural styles can be viewed as kinds of patterns [Sha96] – or perhaps more accurately as pattern languages [Ker95]. 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. The second way that design patterns are related to styles is that for a given style there may exist a set of idiomatic uses of it. These idioms act as “micro architectures”, or architectural design patterns, that are designed to work within a specific architectural style. By providing a framework within which these patterns work, the designer using the pattern can leverage the broad descriptive and analytical capabilities provided by the style along with proven mechanisms for addressing specific design challenges in the form of design patterns.

In our view, patterns and architectural styles are 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). It is also important to note that patterns need not be architectural. Indeed, many patterns in emerging handbooks deal with solutions to lower level programming mechanisms, rather than system structuring issues.

**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<br>at least 1 async-output port<br>at least 0 sync-caller ports | processing-cost, rate, input-message-type(s), output-message-type(s) |
| Resource | Component for which processes contend. | exactly 0 async ports<br>at least 1 sync-callee port | resource-cost |
| Device | Sends messages into the system at a predefined rate. | exactly 0 sync-ports<br>exactly 0 async-input-ports<br>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<br>exactly 1 async-output role | message-type |
| async-msg-pass-rendevous | Like async-msg-pass, but requires rendevous before sending message. N-ary connector. | at-least 1 async-input role<br>exactly 1 async-output role | message-type |
| sync-request | Binary synchronous request channel, typed messages. | exactly 1 sync-caller role<br>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 can not 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.**

## 5.1 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 (RTP/C) is designed to assist architects putting together real-time multimedia systems that will run on a uniprocessor computer. Detailed descriptions of the semantics and analyses underlying the RTP/C style can be found in [Jef93]. 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 a well defined style such as the RTP/C style, however, relatively concrete design patterns play an important role. The well-defined RTP/C primitive design elements and guidelines form a language that can be used to capture more detailed, concrete solutions to specific problems. The style in this case provides a well understood and defined vocabulary framework for composing individual design elements in principled ways that support real-time analyses. Figure 7 provides two design patterns done in the RTP/C style – the “forked-memory” pattern and the “message-replicator” pattern. Each pattern is depicted here using the structure for pattern descriptions provided in [GHJV95]. The patterns have been trimmed of some details for simplicity and conservation of space. Along with a diagram, each pattern provides information describing its applicability, consequences of use, etc. The framework for specifying patterns given in [GHJV95] works well for architectural patterns as well as OO Patterns. The primary difference between architectural patterns and the OO patterns given in the [GHJV95] book and others like it is that architectural patterns address a a more specific set of design issues than the OO patterns. Specifically, they address the architectural issues enumerated in section 2. Just as OMT and objects are used to illustrate 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, a number of the design patterns in [GHJV95] appear to be quite applicable for architectural design. Examples include the Facade pattern (p. 185) that provides a single interface to a collection of objects, the Observer pattern (p. 293) that specifies a mechanism for maintaining consistency amongst objects (or components), and the Strategy pattern (p.315) that specifies a way 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 there are a number of patterns in [GHJV95] that do not address architectural issues. Examples include the Factory Method pattern (p.107) and the Flyweight pattern (p. 195). Both of these patterns deal with lower-level implementation issues than architectures generally specify.

It appears that 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

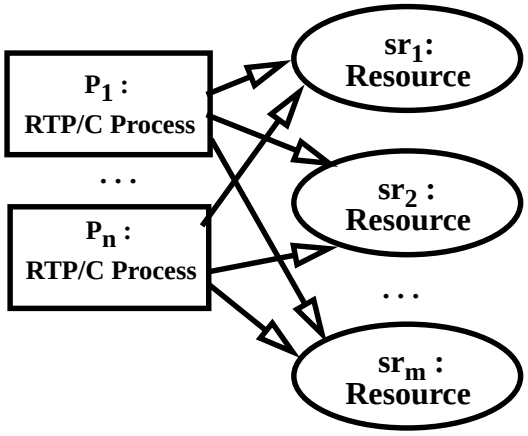
### *Shared-Resource* Architectural Pattern

**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 deadlock-freedom 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 processes to resources.

**Collaborations:** In order to avoid deadlock, a process $P_k$ can only send a request on resources $sr_i$ (locking $sr_i$) 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 resources access or allocation. Other protocols may lead to better average-case performance.

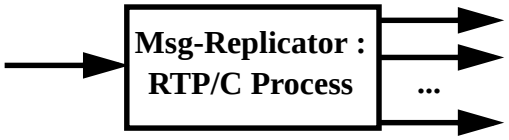
### *Message-Replicator* Architectural Pattern

**Intent:** Send identical messages to a dynamicly 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 style 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 example architectural design patterns in the RTP/C style**

instances and patterns of common architectural design idioms. As a result, the constructs and concepts underlying architectural style are comparable to those underlying an object-oriented design methodology like OMT, rather than a set of design patterns such as those given in [GHJV95]. A specific architectural style is better thought of as a language for building patterns than an instance of a design pattern itself.

# 6 Conclusion and Future Work

Architectures, architectural style, objects, and design patterns represent complementary aspects of design. Although there is certainly some overlap among the issues and aspects of software design that these four approaches address, none completely subsumes the other. Each has something to offer in the way of a complete ensemble of representational models and mechanisms.

# 7 Acknowledgments

This work has been strongly influenced by David Garlan and Robert Allen, whom the authors gratefully acknowledge.

# 8 References

**[AAG93]** Abowd, G., Allen, R. and Garlan, D. Using Style to Give Meaning to Software Architecture. In *Proc. of SIGSOFT '93: Foundations of Software Engineering*, Software Engineering Notes 118(3), pp 9-20, ACM Press, Dec. 1993.

**[AG94]** Robert Allen and David Garlan. Formalizing architectural connection. In *Proceedings of the 16th International Conference on Software Engineering*, pages 71-80, Sorrento, Italy, May 1994.

**[AG96]** Robert Allen and David Garlan, *The Wright Architectural Description Language*, Technical Report, Carnegie Mellon University. To be published, summer 1996.

**[GHJV95]** Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Design*. Addison-Wesley, 1995.

**[JC94]** G.R. Ribeiro Just and P.R. Freire Cunha. Deadlock-free configuration programming. In *Proceedings of the Second International Workshop on Configurable Distributed Systems.* March, 1994.

**[Jef93]** Kevin Jeffay. The Real-Time Producer/Consumer Paradigm: A paradigm for the construction of efficient, predictable real-time systems. *Proc.of the 1993 ACM/SIGAPP Symposium on Applied Computing,* ACM Press, February 1993, pp. 796-804.

**[Ker95]** N.L. Kerth. Caterpillar's fate: A pattern language for transformations from analysis to design. In James O. Coplien and Douglas C. Schmidt, editors, *Pattern Languages of Program Design*. Addison-Wesley, 1995

**[Pree95]** Wolfgang Pree. *Design Patterns for Object-Oriented Software Development*, Addison-Wesley, 1995.

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

**[Rum+91]** James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, William Lorensen, *Object-oriented modeling and design*, Prentice Hall, 1991.

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

**[Sha96]** Mary Shaw. Some patterns for software architectures. In *Proceedings of the Second Pattern Languages of Program Design Workshop.* To appear, 1996.

**[SM92]** Sally Shlaer and Steve Mellor, *Object lifecycles : modeling the world in states*, Yourdon Press, 1992.

**[Ves94]** Steve Vestal. Mode changes in real-time architecture description language. In *Proceedings of the Second International Workshop on Configurable Distributed Systems*, March 1994.