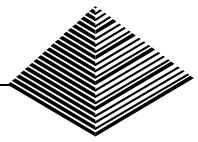
5



Moving From Qualities to Architecture

Architectural Styles

with Mary Shaw

We must recognize the strong and undeniable influence that our language exerts on our ways of thinking and, in fact, delimits the abstract space in which we can formulate—give form to—our thoughts.
— Niklaus Wirth

The previous chapter explained, in general terms, how architectures affect quality attributes. Qualities provide general guidelines and goals for a system but, as we will see many times in the remainder of this book, qualities by themselves are too vague to enable the design of a system.

This chapter discusses the use of recurring architectural *patterns* in achieving architectural aspects of software quality. Patterns occur in all phases of design. In this book, we will identify and discuss two kinds of patterns: system patterns (which are manifested as *architectural styles*, defined in Chapter 2 and discussed here) and design patterns (discussed in Chapter 13). A third kind of pattern, code patterns, will not be treated in depth because it is not by and large architectural in nature, except insofar as this pattern can be shared across architectural components by means of coding templates. Templates will be discussed in Chapter 13 and illustrated in Chapter 11.

What all patterns have in common is that they are predesigned "chunks" that can be tailored to fit a given situation and about which certain characteristics are known. Each pattern represents a package of design decisions that has already been made and can be reused, as a set. What is most different about them is their scale and hence the time of development when each is applied. System patterns tend to be applied by an architect, design patterns usually occur within the confines

Note: Mary Shaw is a professor at the School of Computer Science, Carnegie Mellon University.

of an architectural component, and code patterns live in the implementor's toolkit. We begin by discussing architectural styles.

5.1 Introducing Architectural Styles

An architectural style in software is in some ways analogous to an architectural style in buildings such as Gothic or Greek Revival or Queen Anne. It consists of a few key features and rules for combining those features so that architectural integrity is preserved. An architectural software style is determined by the following:

- A set of component types (e.g., data repository, a process, a procedure) that perform some function at runtime
- A topological layout of these components indicating their runtime interrelationships
- A set of semantic constraints (for example, a data repository is not allowed to change the values stored in it)
- A set of connectors (e.g., subroutine call, remote procedure call, data streams, sockets) that mediate communication, coordination, or cooperation among components.

Thus, a style is not an architecture any more than the term *Gothic* determines exactly what a building looks like. Rather, a style defines a class of architectures; equivalently, it is an abstraction for a set of architectures that meet it. Styles are usually ambiguous (intentionally so) about the number of components involved. For example, a pipe-and-filter stream may have two filters connected by a pipe or 20 filters connected by 19 pipes. Styles may be ambiguous about the mechanism(s) by which the components interact, although some styles (main-program-and-subroutine, for example) bind this explicitly. Styles are always ambiguous about the function of the system: One of the components may be a database, for example, but the kind of data may vary.

Mary Shaw and David Garlan have cataloged a set of architectural styles that they have observed by examining a collection of existing systems. Their motivation was the observation that architectural abstractions for complex systems exist, but we do not study or catalog them, as is common in other engineering disciplines.

Styles not only occur regularly in system designs, but they recur in slightly different forms. Style catalogs can help us recognize when two styles are similar, as Figure 5.1 illustrates. In this figure styles are categorized into related groups. For example, an event system is a substyle of independent components. Event systems themselves have two substyles: implicit and explicit invocation. Style catalogs also tell us the circumstances in which it is appropriate to apply a style.

In the remainder of this section, we'll take a brief tour of the architectural

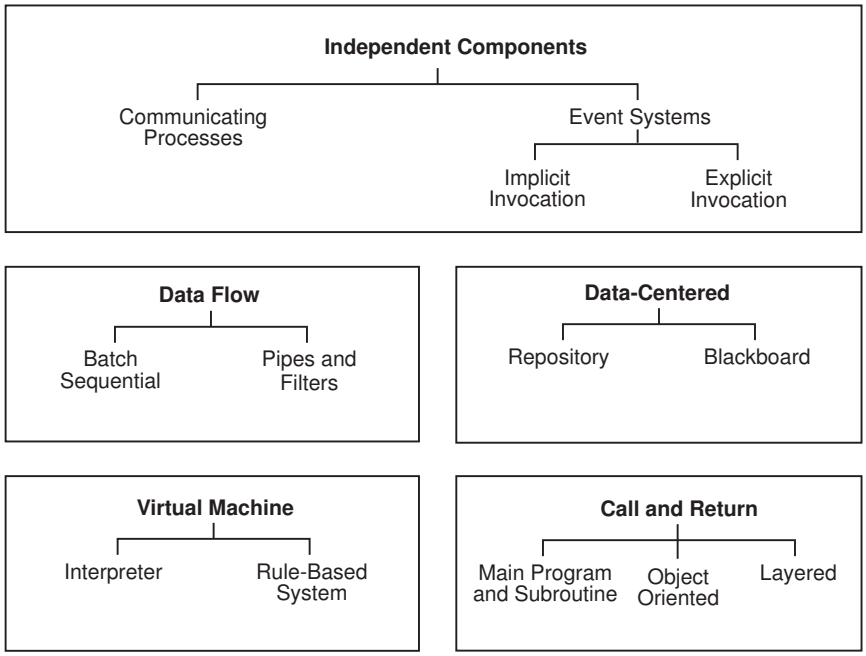


FIGURE 5.1 A small catalog of architectural styles, organized by is-a relations.

style zoo, stopping at each style type shown in Figure 5.1. In the next section, we'll take a closer look at how these styles are related to each other and will produce a refinement of the family tree of Figure 5.1.

DATA-CENTERED ARCHITECTURES

Data-centered architectures have the goal of achieving the quality of integrability of data. The term *data-centered architectures* refers to systems in which the access and update of a widely accessed data store is an apt description. A sketch of this style is shown in Figure 5.2. A client runs on an independent thread of control. The shared data that all of the clients access (and update) may be a passive repository (such as a file) or an active repository (such as a blackboard). Although the data-centered style shown here has a passive repository (you can tell this because no control enters it), it can actually occur in a slightly different guise. At its heart, it is nothing more than a centralized data store that communicates with a number of clients. The means of communication (sometimes called the coordination model) distinguishes the two subtypes: repository (the one shown) and blackboard. A blackboard sends notification to subscribers when data of interest changes, and is thus active. A blackboard differs from Figure 5.2

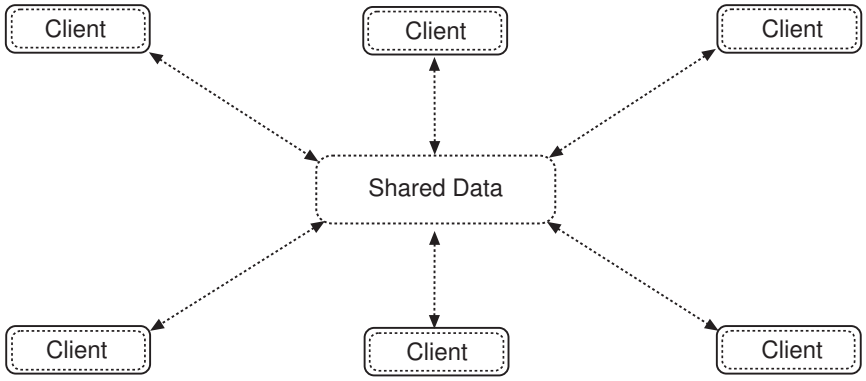


FIGURE 5.2 The data-centered style.

in that it would be drawn with control arrows emanating from the shared data.

Data-centered styles are becoming increasingly important because they offer a structural solution to integrability. Many systems, especially those built from preexisting components, are achieving data integration through the use of blackboard mechanisms. They have the advantage that the clients are relatively independent of each other, and the data store is independent of the clients. Thus, this style is scalable: New clients can be easily added. It is also modifiable with respect to changing the functionality of any particular client because other clients will not be affected. Coupling among clients will lessen this benefit but may occur to enhance performance.

Note that when the clients are built as independently executing processes, what we have is a client-server style that belongs in the independent-component section of the style catalog. Thus, we see that styles are not rigidly separated from each other.

DATA-FLOW ARCHITECTURES

Data-flow architectures have the goal of achieving the qualities of reuse and modifiability. The data-flow style is characterized by viewing the system as a series of transformations on successive pieces of input data. Data enter the system and then flows through the components one at a time until they are assigned to some final destination (output or a data store). The style has two subtypes, batch sequential and pipe-and-filter. In the batch sequential style, processing steps, or components, are independent programs, and the assumption is that each step runs to completion before the next step starts. Each batch of data is transmitted as a whole between the steps. The typical application for this style is classical data processing. This style is illustrated in Figure 5.3.

The pipe-and-filter style emphasizes the incremental transformation of data by successive components. This is a typical style in the UNIX family of operat-

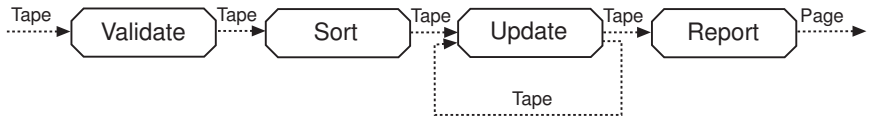


FIGURE 5.3 The batch sequential style.

ing systems. Filters are stream transducers. They incrementally transform data (stream to stream), use little contextual information, and retain no state information between instantiations. Pipes are stateless and simply exist to move data between filters.

Both pipes and filters are run nondeterministically until no more computations or transmissions are possible. Constraints on the pipe-and-filter style indicate the ways in which pipes and filters can be joined. A pipe has a source end that can only be connected to a filter's output port and a sink end that can only be connected to a filter's input port.

Pipe-and-filter systems, like all other styles, have a number of advantages and disadvantages. Their advantages principally flow from their simplicity—the limited ways in which they can interact with their environment. This simplicity means that a pipe-and-filter system's function is no more and no less than the composition of the functions of its primitives. There are no complex component interactions to manage. The pipe-and-filter style also simplifies system maintenance and enhances reuse for the same reason—filters stand alone, and we can treat them as black boxes. Also, both pipes and filters can be hierarchically composed: Any combination of filters, connected by pipes, can be packaged and appear to the external world as a filter. Finally, because a filter can process its input in isolation from the rest of the system, a pipe-and-filter system is easily made parallel or distributed, providing opportunities for enhancing a system's performance without modifying it.

Pipe-and-filter systems also suffer from some disadvantages. Because of the way that a problem is broken down in this style, a batch mentality is implicitly encouraged. So, interactive applications are difficult to create in this style. Also, filter ordering can be difficult: There is no way for filters to cooperatively interact to solve a problem. Performance in such a system is frequently poor for several reasons, all of which stem from the isolation of functionality that makes pipes and filters so modifiable; these reasons are listed below:

- Filters typically force the lowest common denominator of data representation (such as an ASCII stream). If the input stream needs to be transformed into tokens, every filter pays this parsing/unparsing overhead.
- If a filter cannot produce its output until it has received all of its input, it will require an input buffer of unlimited size. A sort filter is an example of a filter that suffers from this problem. If bounded buffers are used, the system could deadlock.

- Each filter operates as a separate process or procedure call, thus incurring some overhead each time it is invoked.

VIRTUAL MACHINE ARCHITECTURES

Virtual machine architectures have the goal of achieving the quality of portability. Virtual machines are software styles that simulate some functionality that is not native to the hardware and/or software on which it is implemented. This can be useful in a number of ways: It can allow one to simulate (and test) platforms that have not yet been built (such as new hardware), and it can simulate "disaster" modes (as is common in flight simulators and safety-critical systems) that would be too complex, costly, or dangerous to test with the real system.

Common examples of virtual machines are interpreters, rule-based systems, syntactic shells, and command language processors. For example, the Java language is built to run on top of the Java Virtual Machine, which allows the language to be platform independent. Virtual machines have the structure shown in Figure 5.4. The figure shows three kinds of data: the program being interpreted, the program's data (such as the values of variables assigned in the execution of the program), and the internal state of the interpreter (such as the values of registers or the current statement being executed). The interpretation engine selects an instruction from the program being interpreted, updates its internal state, and based on the instruction, potentially updates the program's data.

Executing a program via an interpreter adds flexibility through the ability to interrupt and query the program and introduce modifications at runtime, but there is a performance cost because of the additional computation involved in execution.

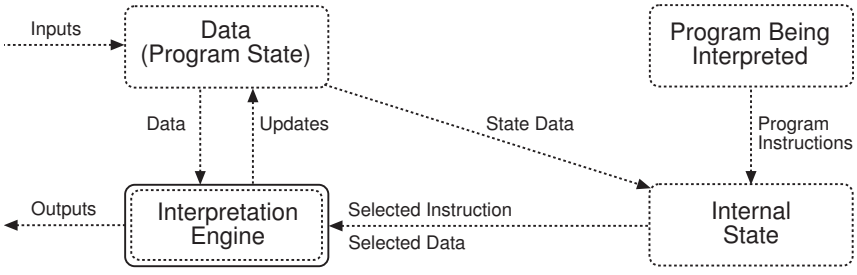


FIGURE 5.4 The virtual machine style.

CALL-AND-RETURN ARCHITECTURES

Call-and-return architectures have the goal of achieving the qualities of modifiability and scalability. Call-and-return architectures have been the dominant architectural style in large software systems for the past 30 years. However, within this style a number of substyles, each of which has interesting features, have emerged.

Main-program-and-subroutine architectures, as shown in Figure 5.5, is the classical programming paradigm. The goal is to decompose a program into smaller pieces to help achieve modifiability. A program is decomposed hierarchically. There is typically a single thread of control and each component in the hierarchy gets this control (optionally along with some data) from its parent and passes it along to its children.

Remote procedure call systems are main-program-and-subroutine systems that are decomposed into parts that live on computers connected via a network. The goal is to increase performance by distributing the computations and taking advantage of multiple processors. In remote procedure call systems, the actual assignment of parts to processors is deferred until runtime, meaning that the assignment is easily changed to accommodate performance tuning. In fact, except that subroutine calls may take longer to accomplish if it is invoking a function on a remote machine, a remote procedure call is indistinguishable from standard main program and subroutine systems.

Object-oriented or *abstract data type systems,* as shown in Figure 5.6, are the modern version of call-and-return architectures. The object-oriented paradigm, like the abstract data type paradigm from which it evolved, emphasizes the

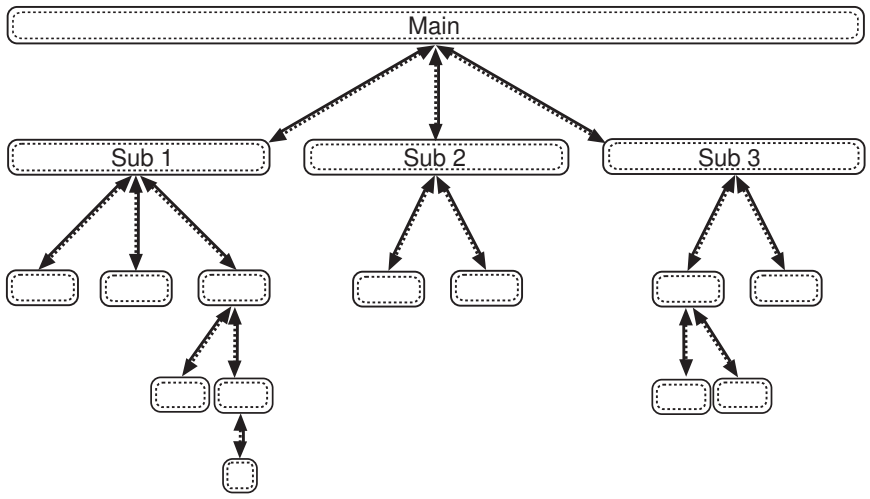


FIGURE 5.5 The main-program-and-subroutine style.

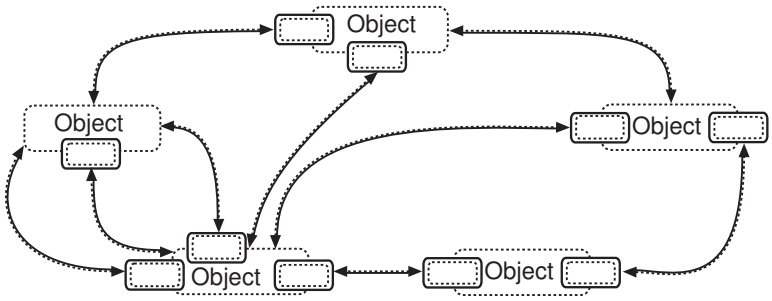


FIGURE 5.6 The object-oriented style.

bundling of data and the knowledge of how to manipulate and access that data. The goal is to achieve the quality of modifiability. This bundle is an encapsulation that hides its internal secrets from its environment. Access to the object is allowed only through provided operations, typically known as methods, which are constrained forms of procedure calls. This encapsulation promotes reuse and modifiability, principally because it promotes separation of concerns: The user of a service need not know, and should not know, anything about how that service is implemented. The main features that distinguish the object-oriented paradigm from abstract data types are inheritance (the hierarchical sharing of definitions and code) and polymorphism (the ability to determine the semantics of an operation at runtime).

When the object abstractions form components that provide black-box services and other components that request those services, this is a *call-based client-server style* (as opposed to a process-based client-server style, which we will see in the next family).

Layered systems, as shown in Figure 5.7, are ones in which components are assigned to layers to control intercomponent interaction. We first saw layers in Chapter 3. In the pure version of this substyle, each level communicates only with its immediate neighbors. The goal is to achieve the qualities of modifiability and, usually, portability. The lowest[1] layer provides some core functionality, such as hardware, or an operating system kernel. Each successive layer is built on its predecessor, hiding the lower layer and providing some services that the upper layers make use of. The upper layers often provide virtual machines themselves: complete sets of coherent functionality upon which an application, or a more complex virtual machine, can be built. Many of the case studies presented in this book make use of layered hierarchies for portability, modifiability, and ease of system parameterization. In practice, layered systems are frequently not "pure"; functions in one layer may talk to functions in layers other than its immediate neighbors. This is called *layer bridging,* and this practice is used where runtime

[1] Layered architectures are sometimes drawn as concentric circles. In that case, *lowest* becomes *innermost*. There is no conceptual difference.

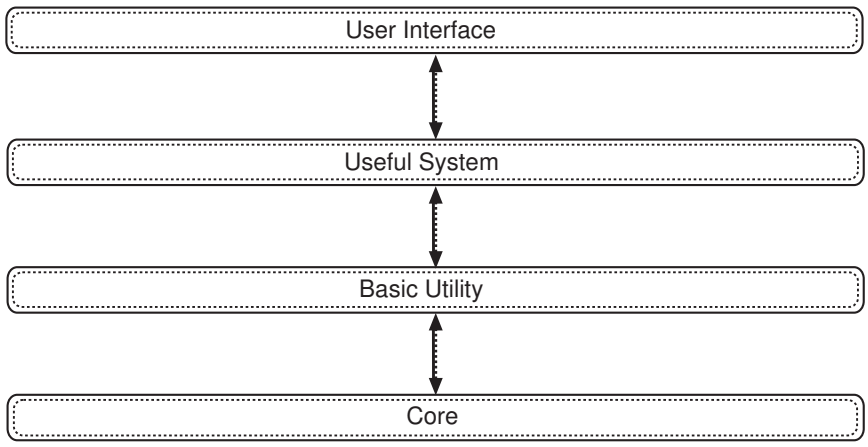


FIGURE 5.7 The layered style.

efficiency is of concern (and the overhead of sending a request through many layers of software cannot be absorbed). But layer bridging compromises the model. If a supposedly portable system is built on a virtual machine—an abstraction of the underlying hardware and software—layer bridging (the direct use of some kernel-level concept, bypassing the portability layer) will make the effort of porting the software greater. On the other hand, if the layering is pure, porting the system involves reimplementing only the portability layer, and this can be done once for *all* systems that are built on the virtual machine. For example, a program written in Java requires no porting effort to run it on a large variety of hardware and software platforms, because Java presents a uniform virtual machine abstraction on many platforms. The Java language developers do the porting once so that users of the abstraction need not do so.

INDEPENDENT COMPONENT ARCHITECTURES

Independent component architectures consist of a number of independent processes or objects that communicate through messages. All of these architectures have the goal of achieving modifiability by decoupling various portions of the computations. They send data to each other but typically do not directly control each other. The messages may be passed to named participants or, in the case of event systems using the publish/subscribe paradigm, may be passed among unnamed participants.

Event systems are a substyle in which control is part of the model. Individual components announce data that they wish to share (publish) with their environment—a set of unnamed components. These other components may register an

interest in this class of data (subscribe). If they do so, when the data appears, they are invoked and receive the data. Typically, event systems make use of a message manager that manages communication among the components, invoking a component (thus controlling it) when a message arrives for it. In this publish/subscribe paradigm, a message manager may or may not control the components to which it forwards messages. Components register types of information that they are willing to provide and that they wish to receive. They then publish information by sending it to the message manager, which forwards the message, or in some cases an object reference, to all interested participants.

This paradigm is important because it decouples component implementations from knowing each others' names and locations. As mentioned, it may involve decoupling control as well, which means that components can run in parallel, only interacting through an exchange of data when they so choose. This decoupling eases component integration, as has been seen in commercial software engineering tool integration environments and as will be shown in Chapters 13 and 16.

Besides event systems, the other substyle of independent components is the *communicating processes* style. These are the classic multiprocess systems such as we saw in Chapter 3, and will see again in Chapter 11. Of these, client-server is a well-known subtype. The goal is to achieve the quality of scalability. A server exists to serve data to one or more clients, which are typically located across a network. The client originates a call to the server, which works, synchronously or asynchronously, to service the client's request. If the server works synchronously, it returns control to the client at the same time that it returns data. If the server works asynchronously, it returns only data to the client (which has its own thread of control).

HETEROGENEOUS STYLES

Systems are seldom built from a single style, and we say that such systems are heterogeneous. There are three kinds of heterogeneity; they are as follows:

1. *Locationally heterogeneous*, meaning that a drawing of its runtime structures will reveal patterns of different styles in different areas. For example, some branches of a main-program-and-subroutines system might have a shared data repository.
2. *Hierarchically heterogeneous*, meaning that a component of one style, when decomposed, is structured according to the rules of a different style, as Figure 5.8 illustrates.
3. *Simultaneously heterogeneous*, meaning that any of several styles may well be apt descriptions of the system.

This last form of heterogeneity recognizes that styles do not partition software architectures into nonoverlapping, clean categories. You may have noticed this already. The data-centered style at the beginning of this discussion was com-

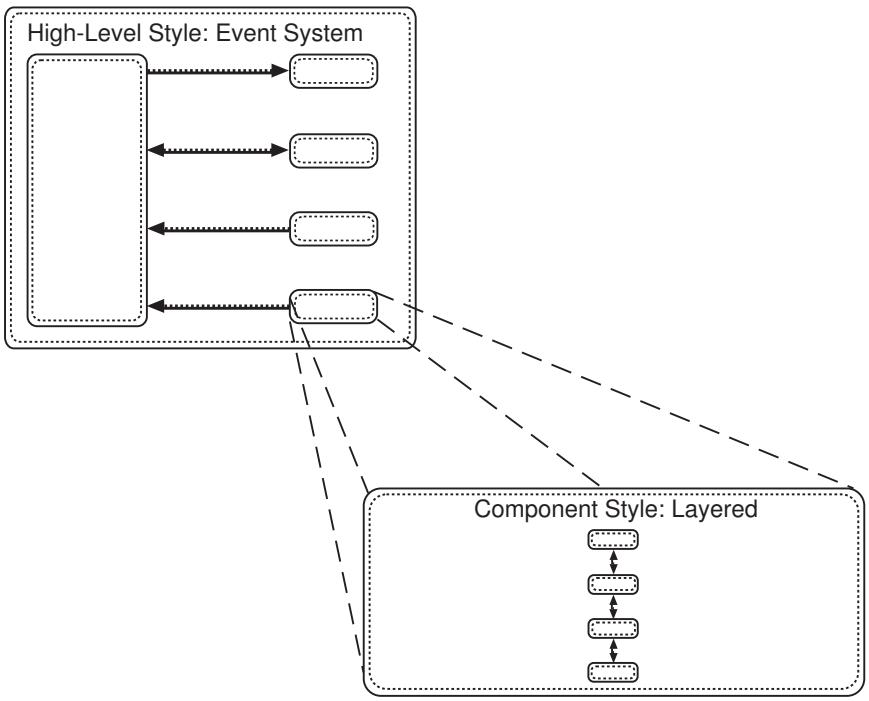


FIGURE 5.8 Hierarchical heterogeneity.

posed of thread-independent clients, much like an independent component architecture. The layers in a layered system may comprise objects or independent components or even subroutines in a main-program-and-subroutines system. The components in a pipe-and-filter system are usually implemented as processes that operate independently, waiting until input is at their port; again, this is similar to independent component systems whose order of execution is predetermined.
To see an illustration of simultaneous heterogeneity, consider the following A-7E architecture from Chapter 3:

- Its primary control mechanism is call-and-return; its uses structure forms layers with the Extended Computer as its portability layer. Hence, it is a member of the layered style of the call-and-return family.
- Its process structure reveals the classic multiprocess architecture of the communicating processes style in the independent components family.
- Its modular structure forms abstract data types and (except for the lack of inheritance) objects, suggesting an object-oriented style in the call-and-return family.
- Its Data Banker is a blackboard, conforming to the blackboard style of the

data-centered family.

- Some input data make a linear trip through the system, in through the Device Interface Module, through the Data Banker, combined with other values in the Function Drivers, and back out through the Device Interface Module again. It is a stretch, but it is possible to describe the A-7E architecture as a data-flow style as well.

In Chapter 11, we'll see another system that consists simultaneously of objects, layers, and communicating processes. In fact, commercial client-server systems such as those that communicate over a CORBA-like infrastructure (see Chapter 8) can often be described as layered object-based process systems, suggesting that this hybrid of three styles may in fact be useful enough as a descriptive vehicle that it deserves its own cage in the style zoo.

These styles are not the full set, or even the only set. They are an early effort at facilitating the architect's job by providing a more sophisticated vocabulary for discussing design alternatives. Other styles will emerge as the field matures.

Should this lack of definitive categorization discourage us? Probably not. Styles exist more as cognitive aids and communication cues than anything else. We can use them as shorthand to convey meaning in a more compact fashion than would otherwise be possible. And just as we know that architectures consist of many structures that do not necessarily resemble each other, it should not surprise us that more than one style may be teased out of (or engineered into) a given system.

In the next section, we take a closer look at styles and the discriminating factors among them. Then we use this information to help us choose specific styles to solve given problems.

5.2 Organizing Architectural Styles

Because an architecture is almost never constructed entirely from one style, an architect needs to understand the interrelationships among styles.

In general, understanding the ramifications of an architecture that is the result of combining styles requires a process of architectural analysis. We will discuss that in Chapter 9. For now, though, we can analyze the set of styles just introduced to see how they are described, what they have in common, and how they might be extended to admit other design possibilities. In this section we begin to organize and classify some of the styles that appear in software descriptions, many of which were introduced in the previous section. We do this for the following reasons:

- By bringing out significant differences that affect the suitability of a style for various tasks, the architect is empowered to make more informed selections.
- By showing which styles are variations of others, the architect can be more confident in choosing appropriate combinations of styles.

- The features used to classify styles will help the designer focus on important design issues by providing a checklist of topics to consider, thereby setting expectations for elements to include in the design.

We first introduce the features that we will use to describe and discriminate styles.

FEATURE CATEGORIES

A system designer's primary impression of an architecture often focuses on the character of the interactions among components. Thus, the major axes of classification are the control and data interactions among components. We can, however, make finer discriminations within these dimensions, such as the following:

- How control is shared, allocated, and transferred among the components
- How data is communicated through the system
- How data and control interact
- The type of reasoning the style permits
- The kinds of components and connectors that are used in the style

Constituent Parts: Components and Connectors. The allowable kinds of components and connectors are important discriminators among styles. Components and connectors are the primary building blocks of architectures. In this categorization, a component is a unit of software that performs some function at runtime. Examples include programs, objects, processes, and filters. A connector is a mechanism that mediates communication, coordination, or cooperation among components. Implementations of connectors may be distributed over many components; often they do not correspond to discrete elements of the running system. Examples include shared representations, remote procedure calls, message-passing protocols, and data streams.

Selecting the types of constituent parts does not, however, uniquely identify the style, which is why there are other feature categories. Further, some styles rely on finer distinctions within a type of component or connector. Some of these distinctions appear in Table 5.1; a local versus a remote procedure call is an example.

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

- *Topology.* What geometric form does the control flow for the system take? A pipeline often has a *linear* (nonbranching) or at least an *acyclic* control topology; a main-program-and-subroutines style features a *hierarchical* (tree-shaped) topology; some server systems have *star* (hub-and-spoke)

topologies; a style consisting of communicating sequential processes may have an *arbitrary* topology. Within each topology it may be useful to stipulate the direction in which control flows. The topology may be static or dynamic; this is determined by the binding time of the elements as described below.

- *Synchronicity.* How dependent are the components' actions upon each others' control states? In a *lockstep* system, the state of any component implies the state of all others; for instance, a batch sequential system's components are in lockstep with each other because one doesn't begin execution until its predecessor finishes. Same instruction, multiple data (SIMD) algorithms for massively parallel machines also work in lockstep. In *synchronous* systems, components synchronize regularly and often, but other state relationships are unpredictable. *Asynchronous* components are largely unpredictable in their interaction or synchronize once in a while, and *opportunistic* components such as autonomous agents work completely independently from each other in parallel. Lockstep systems can be *sequential* or *parallel*, depending on how many threads of control run through them. Other forms of synchronicity imply parallelism.
- *Binding time*. When is the identity of a partner in a transfer-of-control operation established? Some control transfers are predetermined at program *write-* (i.e., source code) time, *compile*-time, or *invocation*-time (i.e., when the operating system initializes the process). Others are bound dynamically while the system is *run*ning.

Data Issues. Data issues describe how data move around a system; they include the following:

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

Control/Data Interaction Issues. Interaction issues describe the relationship

between certain control and data issues.

- *Shape*. Are the control-flow and data-flow topologies substantially isomorphic?
- *Directionality.* If the shapes are substantially the same, does control flow in the *same* direction as data or the *opposite* direction? In a data-flow system such as pipe-and-filter, control and data pass together from component to component. However, in a client-server style, control tends to flow into the servers and data flow into the clients.

Type of Reasoning. Different classes of architectures lend themselves to different types of analysis. A system of components operating asynchronously in parallel yields to vastly different reasoning approaches (e.g., nondeterministic state machine theory) than a system that executes as a fixed sequence of atomic steps (e.g., function composition). Many analysis techniques compose their results from analysis of substructures, but this depends on the ability to combine subanalyses. Thus, different architectural styles are good matches for different analysis techniques. Your choice of architecture may be influenced by the kinds of analysis you require.

Table 5.1 shows a sample set of architectural styles, organized according to these feature categories. We will explore some of these styles in more detail in the next section.

5.3 Refinements of Styles

The classification scheme of Table 5.1 maps out a small part of the space of architectural styles, but it does not capture the richness found in practice, even among this select subset of styles. Each row can be elaborated on to capture more detailed distinctions. In this section we illustrate a refinement of Table 5.1 by partially elaborating on two well-known families of styles that are often used in system development. We do this because the general styles introduced in the previous sections leave many important design decisions unbound. They provide an *approach* to architectural design but are not sufficient in and of themselves to serve as a design. Refinements of those styles are not only possible but necessary before they can lead to an architecture. The refinements we present in this section are more special-purpose tools that can be kept in an architect's conceptual toolkit.

REFINING THE DATA-FLOW STYLE

Data-flow networks describe systems whose components operate on large, continuously available data streams. The components are organized in arbitrary topologies that stream the data with nontransforming connectors. Use of an arbi-

TABLE 5.1 A Feature-Based Classification of Architectural Styles

	Constituent parts		*Control issues*			*Data issues*				*Control/data interaction*	
Style	**Components**	**Connectors**	**Topology** [a]	**Synchronicity** [b]	**Binding time** [c]	**Topology** [a]	**Continuity** [d]	**Mode** [e]	**Binding time**	**Isomorphic shapes**	**Flow directions**
Data flow: *dominated by motion of data through the system, with no "upstream" content control by recipient*									***Type of Reasoning****: Functional composition*		
Batch sequential	Stand-alone programs	Batch data	Linear	Seq.	r	Linear	Spor. hvol.	Passed, shared	r	Yes	Same
Data-flow network	Transducers	Data stream	Arb.	Asynch.	i, r	Arb.	Cont. lvol. or hvol.	passed	i, r	Yes	Same
Pipes and filters	See Section 5.3.										
Other data-flow substyles	See Section 5.3.										
Call-and-return: *dominated by order of computation, usually with single thread of control*									***Type of Reasoning****: Hierarchy (local reasoning)*		
Main program/ subroutines	Procedures, data	Procedure calls	Hier.	Seq.	w, c	Arb.	Spor. lvol.	Passed, shared	w, c, r	No	n/a
Abstract data types	Managers	Static calls	Arb.	Seq.	w, d	Arb.	Spor. lvol.	Passed	w, c, r	Yes	Same
Objects	Managers (objects)	Dynamic calls	Arb.	Seq.	w, c, r	Arb.	Spor. lvol.	Passed	w, c, r	Yes	Same
Call-based client-server	Programs	Calls or RPC	Star	Synch.	w, c, r	Star	Spor. lvol.	Passed	w, c, r	Yes	Opposite
Layered	Various	Various	Hier.	Any	any	Hier.	Spor. lvol., cont.	Any	w, c, i, r	Often	Same or opp.

TABLE 5.1 A Feature-Based Classification of Architectural Styles *Continued*

	Constituent parts		***Control issues***			***Data issues***				***Control/data interaction***	
Style	**Components**	**Connectors**	**Topology** [a]	**Synchronicity** [b]	**Binding time** [c]	**Topology** [a]	**Continuity** [d]	**Mode** [e]	**Binding time**	**Isomorphic shapes**	**Flow directions**
Independent components: dominated by communication patterns among independent, usually concurrent, processes									***Type of Reasoning:*** Nondeterminism		
Event systems	Processes	Signals	Arb.	Asynch.	w, c, r	Arb.	Spor. lvol.	Multicast	w,c,r	Yes	Same
Communicating processes	Processes	Message protocols	Arb.	Any but seq.	w, c, r	Arb.	Spor. lvol.	Any	w,c,r	Possibly	If isomorphic, either
Communicating processes substyles	See Section 5.3.										
Data-centered: dominated by a complex central data store, manipulated by independent computations									***Type of Reasoning:*** Data integrity		
Repository	Memory, computations	Queries	Star	Asynch. opp.	w	Star	Spor. lvol.	Shared passed	w	Possibly	If isomorphic, opposite
Blackboard	Memory, computations	Direct access	Star	Asynch. opp.	w	Star	Spor. lvol.	Shared mcast	w	No	n/a
Virtual machine: characterized by translation of one instruction set into another									***Type of Reasoning:*** Levels of service		
Interpreter	Memory, state machine	Direct data access	Fixed hier.	Seq.	w, c	Hier.	Cont.	Shared	w,c	No	n/a

a. Hier. (hierarchical), arb. (arbitrary), star, linear (one-way), fixed (determined by style).

b. Seq. (sequential, one thread of control), Synch. (synchronous), Asynch. (asynchronous), opp. (opportunistic).

c. w (write-time—that is, in source code), c (compile-time), i (invocation-time), r (runtime).

d. Spor (sporadic), cont. (continuous), hvol. (high-volume), lvol. (low-volume).

e. mcast (multicast)

trary topology makes modification of the system more difficult because interactions among components are unconstrained. Thus, it is natural to wish to place restrictions on the topologies and analyze the results.

Abowd, Allen, and Garlan analyzed what they call the *pipe-and-filter* style by way of formalizing the semantics of architectural styles (as opposed to cataloging their construction, as we have done). They identify three (overlapping) variations of the pipe-and-filter style as follows:

1. Systems without feedback loops or cycles (acyclic)
2. Pipelines (linear)
3. Systems with only fan-out components

Their overall pipe-and-filter style corresponds to the data-flow network style of Table 5.1: The components are elements that asynchronously transform input into output with minimal retained state (i.e., transducers). The transducers are connected in various topologies by high-volume data-flow streams.

The pipeline substyle can be seen in Table 5.2 to be a specialization of data-flow network—its data and control topology are restricted from *arbitrary* in the general form to *linear* in the specialized form, but the classifications are otherwise identical. The fan-out and acyclic substyles similarly differ from the general form only by imposing different topological restrictions.

UNIX pipes and filters, a specialization not treated by Abowd and colleagues but widely used elsewhere, can be seen to be a subspecialization of the pipeline style. The hook-ups can only be specified at the time a program—script, in this case—is written or when the command is given to the operating system. Further, its components are those that accept ASCII streams, not generalized data streams.

The classification shows that all of these styles (acyclic, pipelines, fan-out, UNIX pipes and filters) comprise a family that we have called *data-flow network*, in which the members are distinguished mainly by topological restriction. Table 5.2 shows the relationships among the major styles and their family members.

REFINING THE COMMUNICATING PROCESSES STYLE

Andrews analyzed and cataloged a family of styles based on processes communicating with each other via message passing. This family corresponds to the communicating processes style in Table 5.1. Communicating processes are used to achieve the goals of modifiability and scalability, but performance and configuration constraints affect how these goals are achieved. Andrews identifies eight variants that occur in practice and satisfy different goals. The next paragraphs show how each variant is obtained by refining (specializing) the basic communicating processes style.

One-Way Data Flow Through Networks of Filters. This is a version of the

TABLE 5.2 Specializations of the Data-Flow Network Style

	Constituent parts		*Control issues*			*Data issues*			
Style [a]	**Components**	**Connectors**	**Topology**	**Synchronicity** [b]	**Binding time** [c]	**Topology**	**Continuity** [d]	**Mode**	**Binding time**
Data-flow network	Transducers	Data stream	Arbitrary	Asynch.	i, r	Arbitrary	Cont. lvol. or hvol.	Passed	i, r
▪ Acyclic	Transducers	Data stream	Acyclic	Asynch	i, r	Acyclic	Cont. lvol. or hvol.	Passed	i, r
▪ Fanout	Transducers	Data stream	Hierarchy	Asynch	i, r	Hierarchy	Cont. lvol. or hvol.	Passed	i, r
▪ Pipeline	Transducers	Data stream	Linear	Asynch	i, r	Linear	Cont. lvol. or hvol.	Passed	i, r
– UNIX pipes and filters	Transducers	ASCII stream	Linear	Asynch	i	Linear	Cont. lvol. or hvol.	Passed	i

a. Control/data interaction for all styles is: Isomorphic shapes—yes; Flow directions—same.

b. Asynch. (asynchronous).

c. i (invocation-time), r (runtime).

d. Cont. (continuous), hvol. (high-volume), lvol. (low-volume).

data-flow network substyle implemented with communicating processes. In this version the implementation with messages intrudes on the data-flow abstraction. A piece of data enters the system and makes its way through a series of transformations, each transform accomplished by a separate process. The series need not be linear; Andrews gives an example of a tree of processes forming a sorting network. To analyze this substyle, we note how it differs from both main styles that it resembles. To cast it as a specialization of data-flow networks, we (1) restrict its data and control topologies to one-way flows and (2) relax its data-handling requirements from continuous to sporadic. To cast it as a specialization of communicating processes, we restrict its topologies from arbitrary to one way and its synchronicity to asynchronous. That is, this style lies within the intersection of two major styles: the data-flow network and communicating processes.

Requests and Replies Between Clients and Servers. Clients and servers, a popular style, already shown in Table 5.1. It can be seen to be a specialization of the communicating processes style in which the topologies, synchronicity, and mode are restricted from the general form. This is the naive form, which ignores the usual requirement to maintain state for an ongoing sequence of interactions between the client and the server.

Back-and-Forth (Heartbeat) Interaction Between Neighboring Processes. A heartbeat algorithm causes each node in the process graph to send information out and then gather in new information. An example of applying this algorithm is to discover the topology of a network. On each "beat," each process (representing a processor) communicates with every other process it can, broadcasting its idea of the topology. Between beats, every process assimilates the information just sent to it, combining it with its current idea of the layout. The computation terminates when a completion condition has been met. Andrews proposes two variations of this substyle, depending on whether shared memory is used. We model this form of process interaction by restricting the synchronicity of the communicating processes style to lockstep-parallel (although asynchronous versions are possible) and reflecting the shared-data–distributed-data choice by describing the data and control topologies appropriately.

Probes and Echoes in Graphs. Probe/echo computations work on (incomplete) graphs. A probe is a message sent by a process to a set of successors; an echo is the reply. Probe/echo algorithms can be used to compute a depth-first search on a graph, discover network topologies, or broadcast using neighbors. The probe/echo substyle can be described as specializing the communicating processes style by restricting the topologies to an incomplete graph, synchronicity to asynchronous, data mode to passed, and flow directions to same.

Broadcasts Between Processes in Complete Graphs. Broadcast algorithms use a distinguished process to send a message to all other processes. An example

is to broadcast the value of a central clock in a soft real-time system. Modeling the broadcast style in our classification simply requires restricting the data topology to star (for that portion of the computation involved in the broadcast) and the data mode to broadcast; the control topology remains arbitrary.

Token Passing Along Edges in a Graph. Token-passing algorithms use tokens (a special kind of message) to convey temporal rights to the processes receiving the tokens. Token passing is used, for instance, in algorithms to compute the global state of a distributed asynchronous system or to implement distributed mutual exclusion of a shared resource. Token passing is a refinement of the communicating processes style that restricts the synchronicity to asynchronous, data mode to passed, and flow direction to same. The topologies remain arbitrary, and the continuity remains sporadic low volume.

Coordination Between Decentralized Server Processes. In this model, identical servers are replicated to increase the availability of services (for example, in case of the failure or backlog of a single server). The essence of the algorithm is to provide the appearance to clients of a single, centralized server. This requires that the servers coordinate with each other to maintain a consistent state. One server cannot change the "mutual" state without agreement of a sufficient majority of the others. This weighted voting scheme is implemented by passing multiple tokens among the servers. Architecturally, this algorithm is identical to the token-passing substyle discussed previously.

Replicated Workers Sharing a Bag of Tasks. Unlike decentralized servers that maintain multiple copies of data, this style provides multiple copies of computational elements. The replicated-workers style is a primary tool for SIMD machine programmers. Parallel divide-and-conquer is one of its manifestations. One process can be the administrator, generating the first problem and assigning subproblems. Other processes are workers, solving the subproblems (and generating and administering further subproblems as necessary). Subsolutions bubble back up a hierarchical path until the original administrator can assemble the solution to the global problem. To see SIMD algorithms as a substyle of communicating processes, we restrict the topologies to hierarchical, the synchronicity to synchronous, mode to passed or shared (depending on whether shared data is used) and flow direction to same.

Table 5.3 on the next page summarizes these descriptions.

5.4 Using Styles in System Design

Which style should you choose to design a system, then, if more than one will do? The answer (of course) is that it depends on the qualities that most concern you.

TABLE 5.3 Specializations of the Communicating Processes Style

	Control issues			Data issues				Control/data interaction	
Style [a]	**Topology** [b]	**Synchronicity** [c]	**Binding time** [d]	**Topology**	**Continuity** [e]	**Mode** [f]	**Binding time**	**Isomorphic shapes**	**Flow directions**
Interacting process: *dominated by communication patterns among independent, usually concurrent, processes*									
Communicating processes	Arb.	Any but seq.	w, c, r	Arb.	Spor. lvol.	Any	w, c, r	Possibly	If isomorphic, either
One-way data flow, networks of filters	Linear	Asynch.	w, c, r	Linear	Spor. lvol.	Passed	w, c	Yes	Same
Client/server request/reply	Star	Synch.	w, c, r	Star	Spor. lvol.	Passed	w, c	Yes	Opposite
Heartbeat	Hier.	Ls./par.	w, c, r	Hier. or star	Spor. lvol	Passed shared ci./co.	w, c	No	Same
Probe/echo	Incomplete graph	Asynch.	w, c, r	Incomplete graph	Spor. lvol.	Passed	w, c	Yes	Same
Broadcast	Arb.	Asynch.	w, c, r	Star	Spor. lvol.	Bdcast.	w, c	No	Same
Token passing	Arb.	Asynch.	w, c, r	Arb.	Spor. lvol.	Passed	w, c	Yes	Same
Decentralized servers	Arb.	Asynch.	w, c, r	Arb.	Spor. lvol.	Passed	w, c	Yes	Same
Replicated workers	Hier.	Synch.	w, c, r	Hier.	Spor. lvol.	Passed shared	w, c	Yes	Yes

a. The Constituent parts for all styles are: Components—processes; Connectors—message protocols.

b. Hier. (hierarchical), arb. (arbitrary), star, linear (one-way).

c. Seq. (sequential, one thread of control), ls./par. (lockstep parallel), synch. (synchronous), asynch. (asynchronous), opp. (opportunistic).

d. w (write-time—that is, in source code), c (compile-time), i (invocation-time), r (runtime).

e. Spor. (sporadic), lvol. (low-volume).

f. Bdcast. (broadcast), Mcast (multicast), Ci./co. (copy-in/copy-out).

If you ask an architect to tell you about the architecture for a system, odds are that the answer will be couched in terms of the architectural solution to the most difficult design problem. If the system had to be ultrareliable and achieving this would be problematic, you would probably first hear about the fault-tolerant redundant warm-restart aspects of the architecture. Perhaps the system had to have high performance as well, but if that were not problematic, you would not hear the solution for that at first. If the architect knew that the system was going to live for a long time, growing and being modified constantly, you would hear about the layered objects and abstract data types used to insulate the system from change. If the system also had to be secure but used an off-the-shelf solution, such as encryption/decryption, you wouldn't hear about the solution for that until later.

TABLE 5.4 Rules of Thumb for Choosing an Architectural Style

Style	When to use
Data-flow	It makes sense to view your system as one that produces a well-defined easily identified output that is the direct result of sequentially transforming a well-defined easily identified input in a time-independent fashion. Integrability (in this case, resulting from relatively simple interfaces between components) is important.
▪ Batch sequential	▪ There is a single output operation that is the result of reading a single collection of input and the intermediate transformations are sequential.
▪ Data-flow network	▪ The input and output both occur as recurring series, and there is a direct correlation between corresponding members of each series.
– Acyclic	– ...and the transformations involve no feedback loops.
– Fanout	– ...and the transformations involve no feedback loops, and an input leads to more than one output.
– Pipeline, UNIX pipe-and-filter	– The computation involves transformations on continuous streams of data. – The transformations are incremental; one transformation can begin before the previous step has completed.
▪ Closed-loop control	▪ Your system involves controlling continuing action, is embedded in a physical system, and is subject to unpredictable external perturbation so that preset algorithms go awry.
Call-and-return	The order of computation is fixed, and components can make no useful progress while awaiting the results of requests to other components.
▪ Object-oriented/ abstract data type	▪ Overall modifiability is a driving quality requirement. ▪ Integrability (in this case, via careful attention to interfaces) is a driving quality requirement.
– Abstract data types	– There are many system data types whose representation is likely to change.
– Objects	– Information-hiding results in many like modules whose development time and testing time could benefit from exploiting the commonalities through inheritance.

TABLE 5.4 Rules of Thumb for Choosing an Architectural Style *Continued*

Style	When to use
– Call-and-return-based client-server	–Modifiability with respect to the production of data and how it is consumed is important.
▪ Layered	▪ The tasks in your system can be divided between those specific to the application and those generic to many applications but specific to the underlying computing platform. ▪ Portability across computing platforms is important. ▪ You can use an already-developed computing infrastructure layer (operating system, network management package, etc.).
Independent component	Your system runs on a multiprocessor platform (or may do so in the future). Your system can be structured as a set of loosely coupled components (meaning that one component can continue to make progress somewhat independently of the state of other components). Performance tuning (by reallocating work among processes) is important. Performance tuning (by reallocating processes to processors) is important.
▪ Communicating processes	▪ Message passing is sufficient as an interaction mechanism.
– Lightweight processes	– Access to shared data is critical to meet performance goals.
– Distributed objects	– The reasons for the object-oriented style and the interacting process style all apply.
– One-way data-flow, networks of filters	– The reasons for the data-flow network style and the interacting process style all apply.
– Client-server request/reply	– The tasks can be divided between instigators of requests and executors of those requests or between producers and consumers of data.
– Heartbeat	– The overall state of the system must be assessed from time to time (as in a fault-tolerant system) and the components are working in lockstep with each other. – Availability is a driving requirement.
– Probe/echo	– The topology of the network must be assessed from time to time (as in a fault-tolerant system).
– Broadcast	– All of the components need to be synchronized from time to time. – Availability is an important requirement.
– Token passing	– it makes sense for all of the tasks to communicate with each other in a fully connected graph. – The overall state of the system must be assessed from time to time (such as in a fault-tolerant system), but the components are asynchronous .
– Decentralized servers	– Availability and fault tolerance are driving requirements and the data or services provided by the servers are critical to the system's functionality.

TABLE 5.4 Rules of Thumb for Choosing an Architectural Style *Continued*

Style	When to use
– Replicated workers	– The computation may be solved by a divide-and-conquer approach using parallel computation.
▪ Event systems	▪ You want to decouple the consumers of events from their signalers. ▪ You want scalability in the form of adding processes that are triggered by events already detected/signaled in the system.
Data-centered	A central issue is the storage, representation, management, and retrieval of a large amount of related long-lived data.
▪ Transactional database/ repository	▪ The order of component execution is determined by a stream of incoming requests to access/update the data, and the data is highly structured. ▪ A commercial database that suits your requirements is available and cost effective.
▪ Blackboard	▪ You want scalability in the form of adding consumers of data without changing the producers. ▪ You want modifiability in the form of changing who produces and consumes which data.
Virtual machine	
▪ Interpreter	▪ You have designed a computation but have no machine to run it on.

Styles are like that. A style can serve as the primary description of a system in an area where discourse and thought are most important to meet uncertainty. Other styles may well apply and be fruitful. But a good rule of thumb is "first things first." Start with the architectural structure that provides the most leverage on the qualities (including functionality) that you expect to be most troublesome. From there, consider a style appropriate to that structure that addresses the qualities. At that point, other structures and styles can come into play to help address secondary issues.

We expect that the distinctions established in the classification and refinements of Section 5.2 and Section 5.3 provide a framework for offering design guidance of the general form of If your problem has characteristic *x*, consider architectures with characteristic *y*. However, organizing this information is a major undertaking for each problem domain. In the interim, we can at least state rules of thumb, as in Table 5.4.

5.5 Achieving Quality Goals with Architectural Styles

This section illustrates how different architectural styles lead to different quality attributes by showing how a single system, designed four different ways, differs

in its outcome. This example, originally provided by Parnas and modified by Shaw and Garlan, is a case study that shows different architectural alternatives for a Key Word In Context (KWIC) system. One of the main points of this analysis is to show how one system could be designed in a variety of ways. The functionality was the same in each case; what changed was the system's fitness with respect to a portfolio of quality attributes.

A KWIC system takes a set of text lines as input, produces all circular shifts of these lines, and then alphabetizes the results. A KWIC system is primarily used to create an index that is quickly searchable because every key word can be looked up alphabetically even if it does not appear at the beginning of the original index phrase.

The following is an example of the input and output of a KWIC system:

Input: Sequence of lines
An Introduction to Software Architecture
Key Word in Context

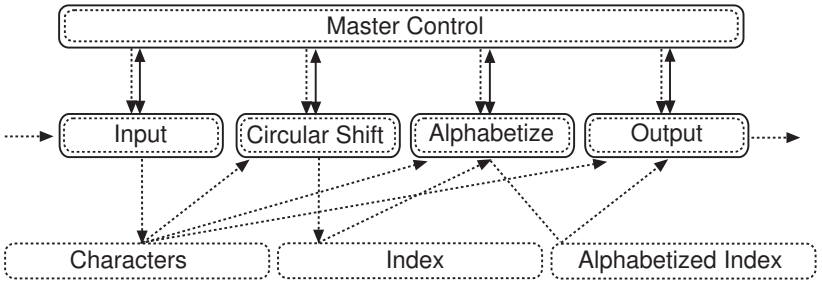
Output: Circularly shifted, alphabetized lines (ignoring case)
An Introduction to Software Architecture
Architecture an Introduction to Software
Context Key Word in
in Context Key Word
Introduction to Software Architecture an
Key Word in Context
Software Architecture an Introduction to
to Software Architecture an Introduction
Word in Context Key

This case study was originally used by Parnas to make an argument for the use of information-hiding as a design discipline. In Shaw and Garlan's analysis, they discuss four possible architectures for this system: the original "straw-man" architecture, described by Parnas, the improved information-hiding (abstract data type) architecture, an implicit invocation architecture, and a UNIX-style pipe-and-filter architecture.

Parnas's original solutions are shown in Figure 5.9.

Parnas's two solutions—while identical with respect to functionality—differ in their support for the quality attributes of performance, modifiability, reusability, and extendibility. The original shared-memory solution (where shared information is stored in global variables, accessible by all components) has good performance but poor modifiability characteristics. Modifiability is poor because any change to the data format, for example, could potentially affect every component in the system. The abstract data type solution, on the other hand, has better support for modifiability because it hides implementation details (such as data formats) inside abstract data types. An information-hiding approach typically compromises performance somewhat because more interfaces are traversed, and it typically uses more space because information is not shared. It may be difficult

Shared-Memory Architecture



Abstract Data Type Architecture

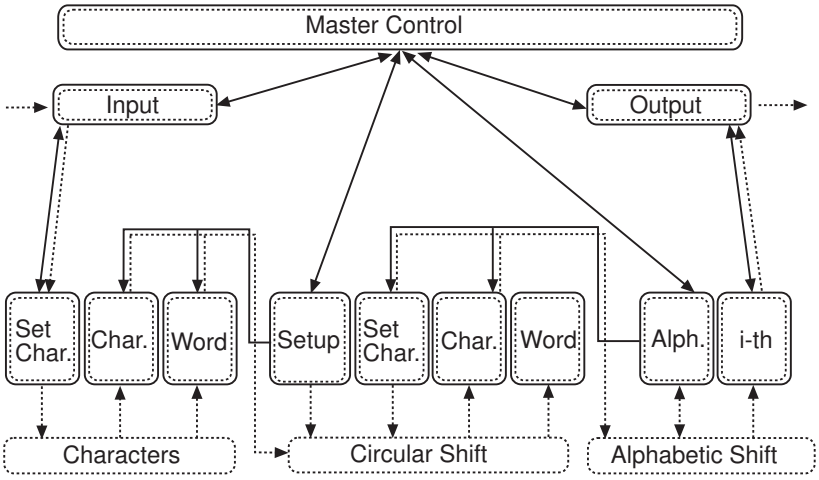


FIGURE 5.9 Parnas's KWIC solutions.

to extend the functionality of this solution because there are relatively complex interactions among the abstract data types.

Shaw and Garlan provide two different solutions to this case study: a solution using an implicit invocation architecture and one using a pipe-and-filter solution. These architectures are given in Figure 5.10.

In the implicit invocation architecture, calls to the Circular Shift function are made implicitly, by inserting data into the Lines buffer, as are calls to the Alphabetizer function. In the pipe-and-filter architecture, two filters provide the entire functionality: one to shift the input stream and one to sort the shifted stream. The implicit invocation architecture supports extendibility. If one wanted to add a new function to this architecture, that function would only need to be

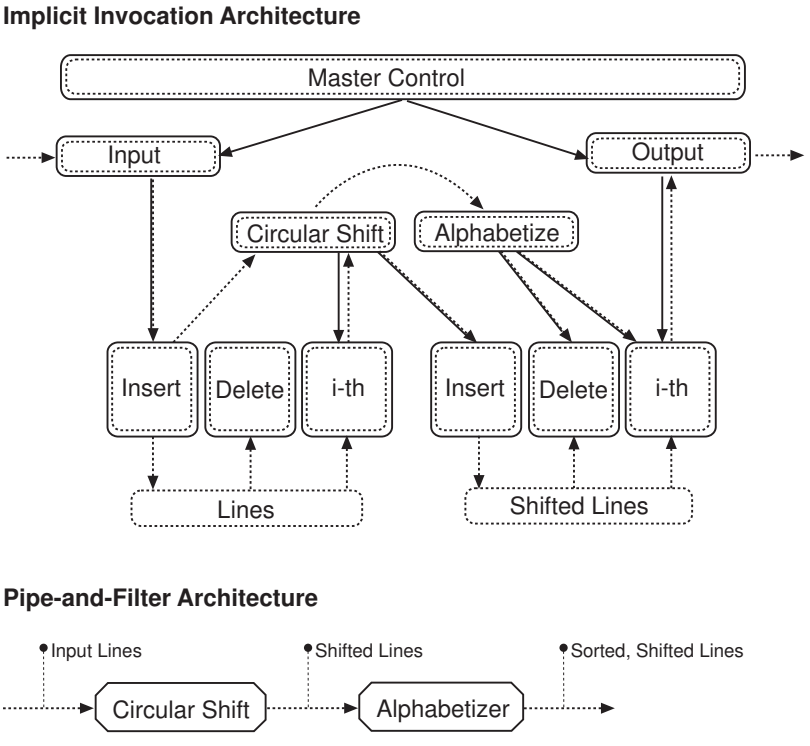


FIGURE 5.10 Shaw and Garlan's KWIC solutions.

registered against an event (such as the insertion of a new line into one of the buffers), and the function would be automatically executed whenever the event occurred.

On the other hand, this architecture offers poor control (what is the order of the new function with respect to any functions previously registered with the line insertion?) and poor space utilization (because data is replicated in the two Lines buffers. The pipe-and-filter architecture is intuitive and clean and offers the best

TABLE 5.5 Rankings of KWIC Architectures with Respect to Quality Attributes

	Shared data	Abstract data type	Implicit Invocation	Pipe-and-filter
Change in algorithm	–	–	+	+
Change in data representation	–	+	–	–
Change in function	–	–	+	+
Performance	+	+	–	–
Reuse	–	+	–	+

support for reuse of the four alternatives. The Circular Shift and Alphabetizer filters could be picked up and used, unchanged, in another system without affecting anything in their environments. However, this solution is less efficient than the others because each filter typically runs as a separate process and may incur some overhead parsing its input and formatting its output. Also, the pipe-and-filter solution may not be space efficient.

APPLYING PATTERNS TO ACHIEVE DESIRED ATTRIBUTES

So far we have briefly described each solution to the KWIC problem and provided a laundry list of costs and benefits. Shaw and Garlan, in their analysis, present a number of findings with respect to how each of the solutions accommodates a change in the underlying algorithms, a change in the way that data are represented, and a change in the function. They also evaluate each solution with respect to performance and support for reuse. Their results are presented in Table 5.5.

The problem with this table is that it is not repeatable. Or, at the least, it is not repeatable without deep knowledge of the various architectural styles, their costs, their benefits, and an understanding of how each style accommodates the problem at hand. This appears to be in conflict with one of the goals of designing with patterns and styles: that a novice should be able to easily apply expert knowledge, as encapsulated and represented by these designs. In practice, however, it is not that simple.

So, how can one go about systematically comparing these (and other) software architectures with respect to their overall satisfaction of a collection of quality attributes? We discuss an alternative technique for analyzing this system in Chapter 9.

We will also discuss the use of patterns in more detail in Chapter 13, where we will concentrate on the use of patterns as a design discipline. There we will look at how the rigorous application of a *small* number of *straightforward* patterns can dramatically simplify a software architecture.

5.6 Summary

This chapter has introduced a set of architectural styles. Styles occur in related groups, and styles in separate groups may actually be closely related. Styles give us a shorthand way of describing a system in ways that make sense, even though the same system may be described with equal fidelity by any of several styles. Styles represent a first-order approach at achieving a system's quality requirements via architectural means, and which style we use to describe a system depends on which qualities we are trying hardest to achieve.

Styles can be described by a set of features, such as the nature of their com-

ponents and connectors, their static topologies and dynamic control- and data-passing patterns, and the kind of reasoning they admit.

For now, we turn our attention to the primitive design principles that underlie architectural styles (and, as we shall see, design and code patterns). What is it that imparts portability to one design and efficiency to another and integrability to a third? We want to examine these primitives.

We will next turn our attention to these fundamental building blocks, which we call *unit operations*.

5.7 For Further Reading

Shaw and Garlan's catalog of styles may be found in [Shaw 96*a*]. The classification of styles in Section 5.2 is joint work by Mary Shaw and Paul Clements [Shaw 97]. Their paper contains many more substyles as well as a comprehensive list of references to authors who have written about each of the substyles mentioned in Tables 5.1 through 5.3.

Abowd, Allen, and Garlan [Abowd 95] provide a detailed discussion, and formal analysis, of data-flow styles. Andrews [Andrews 91] provides a detailed discussion of styles of communicating processes.

Parnas [Parnas 72] provided the original KWIC problem which Shaw and Garlan subsequently analyzed [Shaw 96*a*].

5.8 Discussion Questions

1. Section 5.4 gives rules of thumb for style selection. Pick a few of the styles not mentioned in that section and see if you can write rules of thumb for choosing those styles.
2. A large number of styles are designed to support the quality of modifiability. Give examples and styles of the different types of modification supported by them.
3. Styles, as they are commonly described, are the result of empirical observation, not a taxonomic organization from first principles. As a result, overlap is high: Objects can be cooperating processes that can be layered, and so on. Why is that? Hint: Think about what architectural structures from Chapter 2 are involved in the description of each style.