Software Architecture
Barber
Project Milestone #3
Architectural Style Discussion
from 1st Edition of *Software Architecture in Practice*

Software Architecture in Practice

Len Bass
Paul Clements
Rick Kazman

ADDISON–WESLEY

Boston • San Francisco • New York • Toronto • Montreal
London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

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.

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.

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.

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.

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.

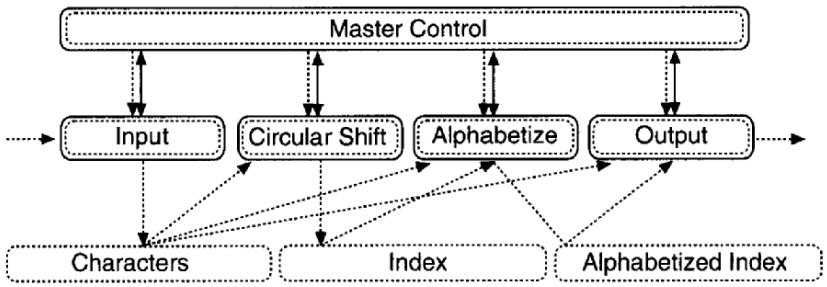
Parnas's two solutions—while identical with respect to functionality—differ in their support for the quality attributes of performance, modifiability, reusability, and extendibility. The original shared-memory solution (where shared information is stored in global variables, accessible by all components) 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 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 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

Shared-Memory Architecture



Abstract Data Type Architecture

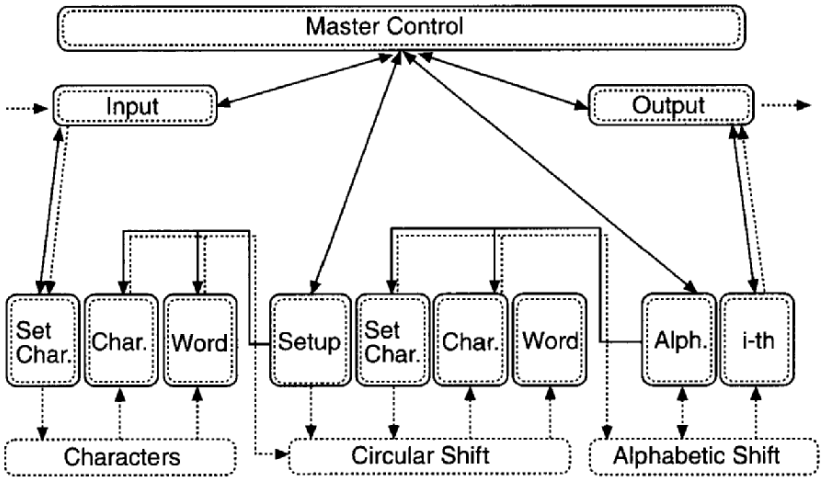


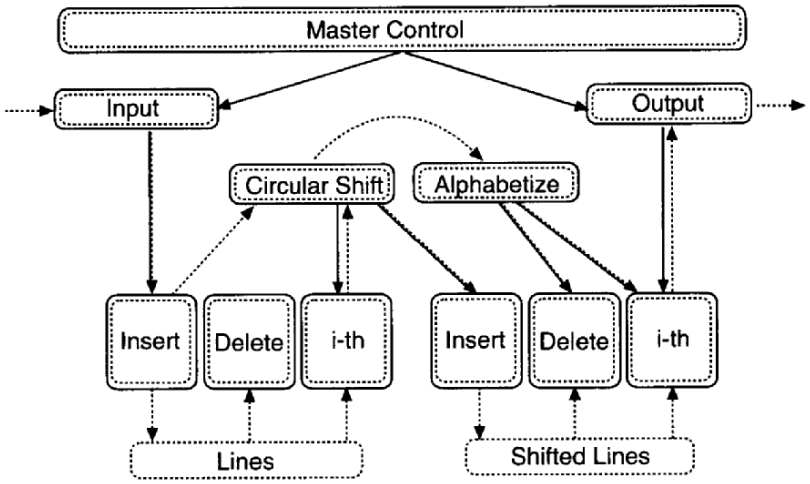
FIGURE 5.9 Parnas's KWIC solutions.

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,

Implicit Invocation Architecture



Pipe-and-Filter Architecture

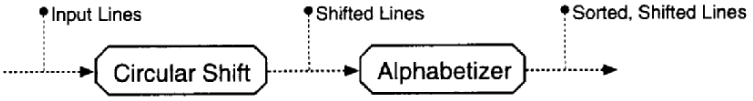


FIGURE 5.10 Shaw and Garlan's KWIC solutions.

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

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	–	+	–	+

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 components 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.