2.2 PIPES AND FILTERS

In a pipe-and-filter style each component has a set of inputs and a set of outputs. A component reads streams of data on its inputs and produces streams of data on its outputs. This is usually accomplished by applying a local transformation to the input streams and computing incrementally, so that output begins before input is consumed. Hence components are termed *filters*. The connectors of this style serve as conduits for the streams, transmitting outputs of one filter to inputs of another. Hence the connectors are termed *pipes*.

Among the important invariants of the style is the condition that filters must be independent entities: in particular, they should not share state with other filters. Another important invariant is that filters do not know the identity of their upstream and downstream filters. Their specifications might restrict what appears on the input pipes or make guarantees about what appears on the output pipes, but they may not identify the components at the ends of those pipes. Furthermore, the correctness of the output of a pipe-and-filter network should not depend on the order in which the filters perform their incremental processing—although fair scheduling can be assumed. (See [AG92, AAG93] for in-depth treatment of this style and its formal properties.) Figure 2.2 illustrates this style.

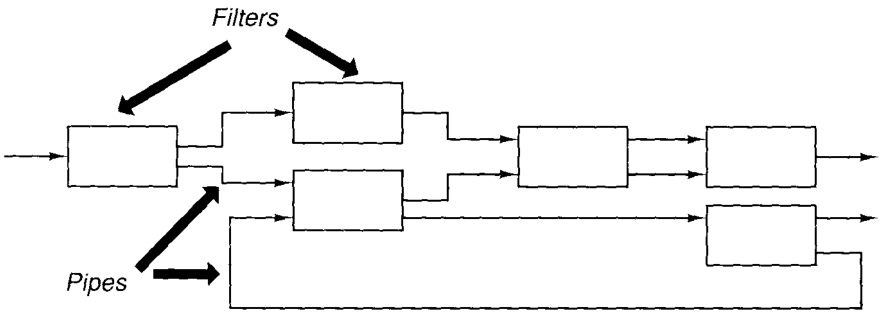


FIGURE 2.2 Pipes and Filters

Common specializations of this style include *pipelines*, which restrict the topologies to linear sequences of filters; bounded pipes, which restrict the amount of data that can reside on a pipe; and typed pipes, which require that the data passed between two filters have a well-defined type.

A degenerate case of a pipeline architecture occurs when each filter processes all of its input data as a single entity.[2] In this case the architecture becomes a *batch sequential* system. In these systems pipes no longer serve the function of providing a stream of data, and therefore are largely vestigial. Hence such systems are best treated as instances of a separate architectural style.

The best-known examples of pipe-and-filter architectures are programs written in the Unix shell [Bac86]. Unix supports this style by providing a notation for connecting components (represented as Unix processes) and by providing run-time mechanisms for implementing pipes. As another well-known example, traditionally compilers have been

[2] In general, we find that the boundaries of styles can overlap. This should not deter us from identifying the main features of a style with its central examples of use.

viewed as pipeline systems (though the phases are often not incremental). The stages in the pipeline include lexical analysis, parsing, semantic analysis, and code generation. (We return to this example in the case studies.) Other examples of pipes and filters occur in signal-processing domains [DG90], parallel programming [BAS89], functional programming [Kah74], and distributed systems [BWW88].

Pipe-and-filter systems have a number of nice properties. First, they allow the designer to understand the overall input/output behavior of a system as a simple composition of the behaviors of the individual filters. Second, they support reuse: any two filters can be hooked together, provided they agree on the data that are being transmitted between them. Third, systems are easy to maintain and enhance: new filters can be added to existing systems and old filters can be replaced by improved ones. Fourth, they permit certain kinds of specialized analysis, such as throughput and deadlock analysis. Finally, they naturally support concurrent execution. Each filter can be implemented as a separate task and potentially executed in parallel with other filters.

But these systems also have their disadvantages.[3] First, pipe-and-filter systems often lead to a batch organization of processing. Although filters can process data incrementally, they are inherently independent, so the designer must think of each filter as providing a complete transformation of input data to output data. In particular, because of their transformational character, pipe-and-filter systems are typically not good at handling interactive applications. This problem is most severe when incremental display updates are required, because the output pattern for incremental updates is radically different from the pattern for filter output. Second, they may be hampered by having to maintain correspondences between two separate but related streams. Third, depending on the implementation, they may force a lowest common denominator on data transmission, resulting in added work for each filter to parse and unparse its data. This, in turn, can lead both to loss of performance and to increased complexity in writing the filters themselves.

2.3 DATA ABSTRACTION AND OBJECT-ORIENTED ORGANIZATION

In the style based on data abstraction and object-oriented organization, data representations and their associated primitive operations are encapsulated in an abstract data type or object. The components of this style are the objects—or, if you will, instances of abstract data types. Objects are examples of a type of component we call a *manager* because it is responsible for preserving the integrity of a resource (here the representation). Objects interact through function and procedure invocations. Two important aspects of this style are (1) that an object is responsible for preserving the integrity of its representation (usually by maintaining some invariant over it), and (2) that the representation is hidden from other objects. Figure 2.3 illustrates this style.[4]

[3] This is true in spite of the fact that the pipe-and-filter style, like every style, has a set of devout followers—people who believe that all problems worth solving can best be solved using that particular style.

[4] We haven't mentioned inheritance in this description. While inheritance is an important organizing principle for defining the types of objects in a system, it does not have a direct architectural function. In particular, in our view, an inheritance relationship is not a connector, since it does not define the interaction between components in a system. Also, in an architectural setting inheritance of properties is not restricted to object types—but may include connectors and even architectural styles.

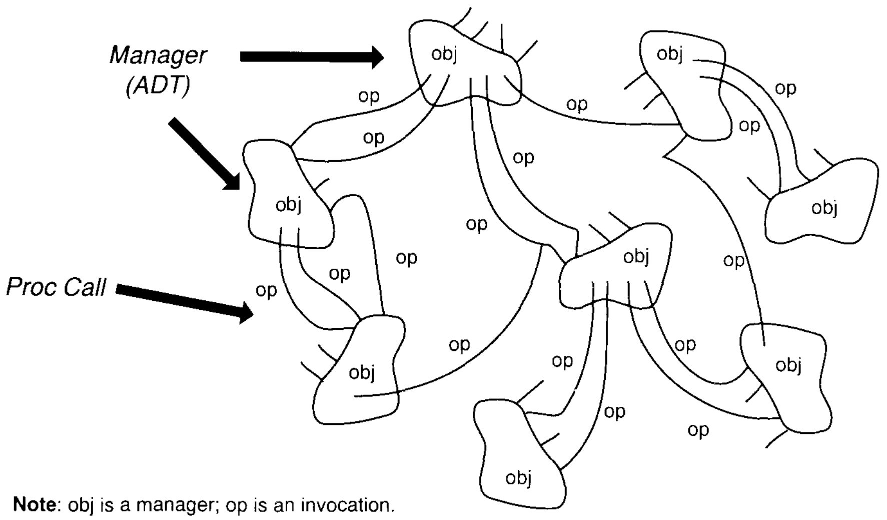


FIGURE 2.3 Abstract Data Types and Objects

The use of abstract data types, and increasingly the use of object-oriented systems, is, of course, widespread. There are many variations. For example, some systems allow "objects" to be concurrent tasks; others allow objects to have multiple interfaces [KG89, Har87b].

Object-oriented systems have many nice properties, most of which are well known. Because an object hides its representation from its clients, it is possible to change the implementation without affecting those clients. Additionally, the bundling of a set of accessing routines with the data they manipulate allows designers to decompose problems into collections of interacting agents.

But object-oriented systems also have some disadvantages. The most significant is that in order for one object to interact with another (via procedure call) it must know the identity of that other object. This is in contrast, for example, to pipe-and-filter systems, where filters do not need to know what other filters are in the system in order to interact with them. In object-oriented systems, then, whenever the identity of an object changes it is necessary to modify all other objects that explicitly invoke it. In a module-oriented language this manifests itself as the need to change the "import" list of every module that uses the changed module. There can also be side-effect problems: if A uses object B and C also uses B, then C's effects on B look like unexpected side effects to A, and vice versa.

2.4 EVENT-BASED, IMPLICIT INVOCATION

In a system in which the component interfaces provide a collection of procedures and functions, such as an object-oriented system, components typically interact with each other by explicitly invoking those routines. Recently, however, there has been considerable interest in an alternative integration technique, variously referred to as *implicit invocation*, *reactive integration*, and *selective broadcast*. This style has historical roots in systems based on actors [Hew69], constraint satisfaction, daemons, and packet-switched networks.

The idea behind implicit invocation is that instead of invoking a procedure directly, a component can announce (or broadcast) one or more events. Other components in the system can register an interest in an event by associating a procedure with it. When the event is announced, the system itself invokes all of the procedures that have been registered for the event. Thus an event announcement "implicitly" causes the invocation of procedures in other modules.

For example, in the Field system [Rei90], tools such as editors and variable monitors register for a debugger's breakpoint events. When a debugger stops at a breakpoint, it announces an event that allows the system to automatically invoke procedures of those registered tools. These procedures might scroll an editor to the appropriate source line or redisplay the value of monitored variables. In this scheme, the debugger simply announces an event, but does not know what other tools or actions (if any) are concerned with that event, or what they will do when that event is announced.

Architecturally speaking, the components in an implicit invocation style are modules whose interfaces provide both a collection of procedures (as with abstract data types) and a set of events. Procedures may be called in the usual way, but a component can also register some of its procedures with events of the system. This will cause these procedures to be invoked when those events are announced at run time.

The main invariant of this style is that announcers of events do not know which components will be affected by those events. Thus components cannot make assumptions about the order of processing, or even about what processing will occur as a result of their events. For this reason, most implicit invocation systems also include explicit invocation (i.e., normal procedure call) as a complementary form of interaction.

Examples of systems with implicit invocation mechanisms abound [GKN92]. They are used in programming environments to integrate tools [Ger89, Rei90], in database management systems to ensure consistency constraints [Hew69, Bal86], in user interfaces to separate presentation of data from applications that manage the data [KP88, SBH+83], and by syntax-directed editors to support incremental semantic checking [HN86, HGN91].

One important benefit of implicit invocation is that it provides strong support for reuse. Any component can be introduced into a system simply by registering it for the events of that system. A second benefit is that implicit invocation eases system evolution [SN92]. Components may be replaced by other components without affecting the interfaces of other components in the system.

The primary disadvantage of implicit invocation is that components relinquish control over the computation performed by the system. When a component announces an event, it cannot assume other components will respond to it. Moreover, even if it does know what other components are interested in the events it announces, it cannot rely on the order in which they are invoked. Another problem concerns exchange of data. Sometimes data can be passed with an event, but in other situations event systems must rely on a shared repository for interaction. In these cases global performance and resource management can become critical issues. Finally, reasoning about correctness can be problematic, since the meaning of a procedure that announces events will depend on the context of bindings in which it is invoked. This is in contrast to traditional reasoning about procedure calls, which need only consider a procedure's pre- and post-conditions when reasoning about the functional behavior of its invocation.

2.5 LAYERED SYSTEMS

A layered system is organized hierarchically, each layer providing service to the layer above it and serving as a client to the layer below. In some layered systems inner layers are hidden from all except the adjacent outer layer, except for certain functions carefully selected for export. Thus in these systems the components implement a virtual machine at some layer in the hierarchy. (In other layered systems the layers may be only partially opaque.) The connectors are defined by the protocols that determine how the layers will interact. Topological constraints include limiting interactions to adjacent layers. Figure 2.4 illustrates this style.

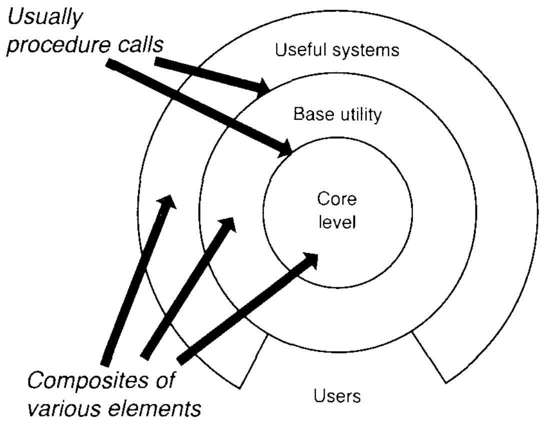


FIGURE 2.4 Layered Systems

The most widely known examples of this kind of architectural style are layered communication protocols [McC91]. In such applications each layer provides a substrate for communication at some level of abstraction. Lower levels define lower levels of interaction, the lowest typically being defined by hardware connections. Other application areas for this style include database systems and operating systems [BO92, FO85, LS79].

Layered systems have several desirable properties. First, they support designs based on increasing levels of abstraction. This allows implementors to partition a complex problem into a sequence of incremental steps. Second, they support enhancement. As with pipelines, because each layer interacts with at most the layers below and above, changes to the function of one layer affect at most two other layers. Third, they support reuse. Like abstract data types, they allow different implementations of the same layer to be used interchangeably, provided they support the same interfaces to their adjacent layers. This leads to the possibility of defining standard layer interfaces upon which different implementors can build. (Good examples are the OSI ISO model and some of the X Window System protocols.)

But layered systems also have disadvantages. Not all systems are easily structured in a layered fashion. (We will see an example of this later in the case studies of Chapter 3.) And even if a system *can* logically be structured in layers, considerations of performance may require closer coupling between logically high-level functions and their lower-level implementations. Additionally, it may be quite difficult to find the right levels of abstraction. This is particularly true for standardized layered models. The communications community, for instance, has had some difficulty mapping existing protocols into the ISO framework because many of those protocols bridge several layers.

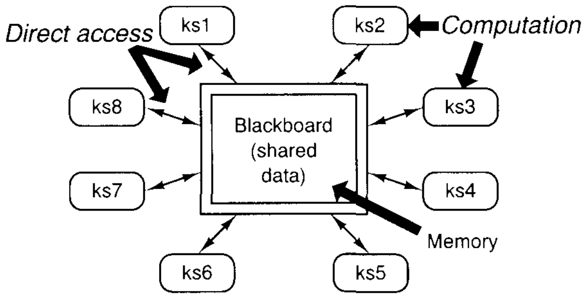
2.6 REPOSITORIES

In a repository style there are two quite distinct kinds of components: a central data structure represents the current state, and a collection of independent components operate on the central data store. Interactions between the repository and its external components can vary significantly among systems.

The choice of a control discipline leads to two major subcategories. If the types of transactions in an input stream trigger selection of processes to execute, the repository can be a traditional database. On the other hand, if the current state of the central data structure is the main trigger for selecting processes to execute, the repository can be a blackboard.

Figure 2.5 illustrates a simple view of a blackboard architecture. (We will examine more detailed models in the case studies.) The blackboard model is usually presented with three major parts:

1. **The knowledge sources:** separate, independent parcels of application-dependent knowledge. Interaction among knowledge sources takes place solely through the blackboard.
2. **The blackboard data structure:** problem-solving state data, organized into an application-dependent hierarchy. Knowledge sources make changes to the blackboard that lead incrementally to a solution to the problem.
3. **Control:** driven entirely by the state of the blackboard. Knowledge sources respond opportunistically when changes in the blackboard make them applicable.



Note: ks is a knowledge source.

FIGURE 2.5 The Blackboard

The diagram shows no explicit representation of the control component. Invocation of a knowledge source (ks) is triggered by the state of the blackboard. The actual locus of control, and hence its implementation, can be in the knowledge sources, the blackboard, a separate module, or some combination of these.

Blackboard systems have traditionally been used for applications requiring complex interpretations of signal processing, such as speech and pattern recognition. Several of these are surveyed by Nii [Nii86]. They have also appeared in other kinds of systems that involve shared access to data with loosely coupled agents [ACM90].

There are, of course, many other examples of repository systems. Batch-sequential systems with global databases are a special case. Programming environments are often organized as a collection of tools together with a shared repository of programs and program fragments [BSS84]. Even applications that have traditionally been viewed as pipeline architectures may be more accurately interpreted as repository systems. For example, as we will see later, while compiler architecture has traditionally been presented as a pipeline, the "phases" of most modern compilers operate on a base of shared information (symbol tables, abstract syntax tree, etc.).

2.7 INTERPRETERS

In an interpreter organization a virtual machine is produced in software. An interpreter includes the pseudoprogram being interpreted and the interpretation engine itself. The pseudoprogram includes the program itself and the interpreter's analog of its execution state (activation record). The interpretation engine includes both the definition of the interpreter and the current state of *its* execution. Thus an interpreter generally has four components: an interpretation engine to do the work, a memory that contains the pseudocode to be interpreted, a representation of the control state of the interpretation engine, and a representation of the current state of the program being simulated (see Figure 2.6).

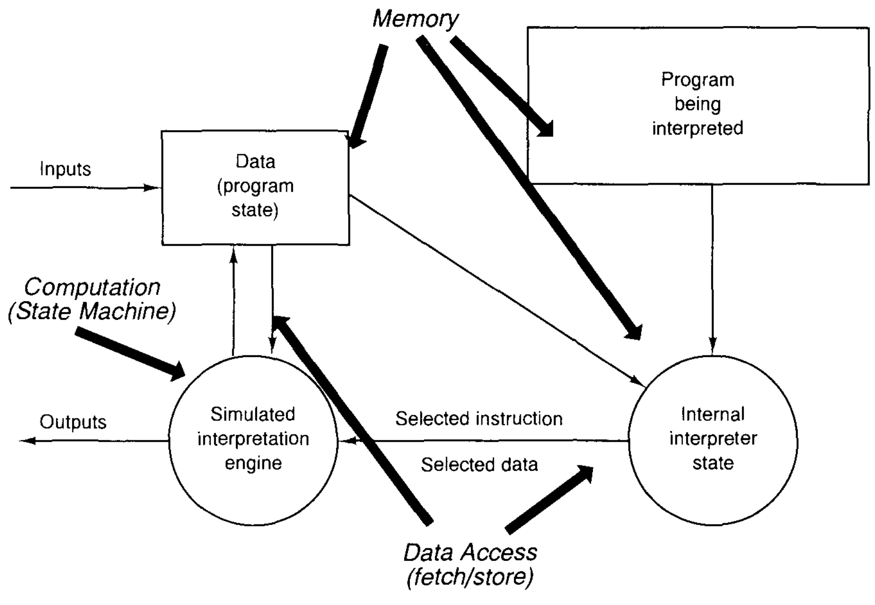

Memory
Program being interpreted
Inputs
Data (program state)
Computation (State Machine)
Outputs
Simulated interpretation engine
Selected instruction
Selected data
Internal interpreter state
Data Access (fetch/store)

FIGURE 2.6 Interpreter

Interpreters are commonly used to build virtual machines that close the gap between the computing engine expected by the semantics of the program and the computing engine available in hardware. We occasionally speak of a programming language as providing, for example, a "virtual Pascal machine."

2.8 PROCESS CONTROL

Another architectural style is based on process control loops. This system organization is not widely recognized in the software community; nevertheless it seems to quietly appear within designs dominated by other models. Unlike object-oriented or functional designs, which are characterized by the kinds of components that appear, control-loop designs are characterized both by the kinds of components involved and the special relations that must hold among them.

2.8.1 Process-Control Paradigms

Continuous processes of many kinds convert input materials to products with specific properties by performing operations on the inputs and on intermediate products. The values of measurable properties of system state (materials, equipment settings, etc.) are called the variables of the process. Process variables that measure the output materials are called the controlled variables of the process. The properties of the input materials, intermediate products, and operations are captured in other process variables. In particular, the manipulated variables are associated with things that can be changed by the control system in order to regulate the process. (Process variables should not be confused with program variables.) Figure 2.7 gives some useful definitions.

Process variables. Properties of the process that can be measured; several specific kinds are often distinguished.

Controlled variable. Process variable whose value the system is intended to control.

Input variable. Process variable that measures an input to the process.

Manipulated variable. Process variable whose value can be changed by the controller.

Set point. The desired value for a controlled variable.

Open-loop system. System in which information about process variables is not used to adjust the system.

Closed-loop system. System in which information about process variables is used to manipulate a process variable to compensate for variations in process variables and operating conditions.

Feedback control system. The controlled variable is measured, and the result is used to manipulate one or more of the process variables.

Feedforward control system. Some of the process variables are measured, and anticipated disturbances are compensated for without waiting for changes in the controlled variable to be visible.

FIGURE 2.7 Process-Control Definitions

The purpose of a control system is to maintain specified properties of the outputs of the process at (sufficiently near) given reference values called the *set points*. If the input materials are pure, if the process is fully defined, and if the operations are completely repeatable, the process can simply run without surveillance. Such a process is called an open-loop system. Figure 2.8 shows such a system, a hot-air furnace that uses a constant burner setting to raise the temperature of the air that passes through. A similar furnace that uses a timer to turn the burner off and on at fixed intervals is also an open-loop system.

The open-loop assumptions are rarely valid for physical processes in the real world. More often, properties such as temperature, pressure, and flow rates are monitored, and their values are used to control the process by changing the settings of apparatus such as valves, heaters, and chillers. Such systems are called *closed-loop systems*. A home thermostat

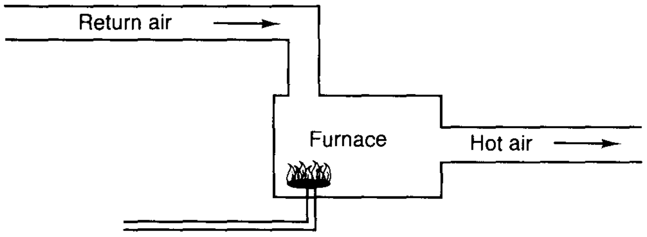


FIGURE 2.8 Open-Loop Temperature Control

is a common example: the air temperature at the thermostat is measured, and the furnace is turned on and off as necessary to maintain the desired temperature (the set point). Figure 2.9 shows the addition of a thermostat to convert Figure 2.8 to a closed-loop system.

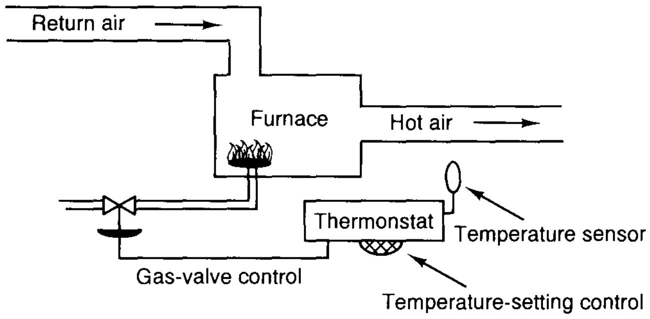


FIGURE 2.9 Closed-Loop Temperature Control

There are two general forms of closed-loop control. Feedback control, illustrated in Figure 2.10, adjusts the process according to measurements of the controlled variable. The important components of a feedback controller are the process definition, the process variables (including designated input and controlled variables), a sensor to obtain the controlled variable from the physical output, the set point (target value for the controlled variable), and a control algorithm. Figure 2.9 corresponds to Figure 2.10 as follows: the furnace with burner is the process; the thermostat is the controller; the return air temperature is the input variable; the hot air temperature is the controlled variable; the thermostat setting is the set point; and the temperature sensor is the sensor.

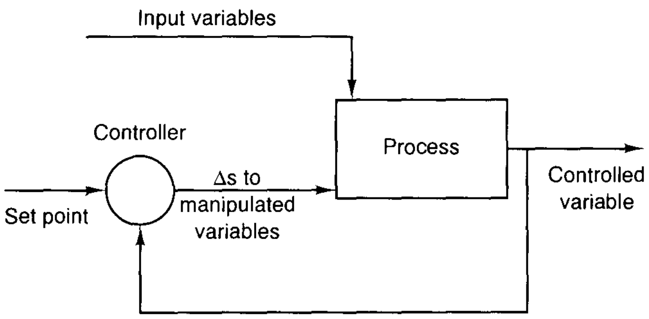


FIGURE 2.10 Feedback Control

Feedforward control, shown in Figure 2.11, anticipates future effects on the controlled variable by measuring other process variables whose values may be more timely; it adjusts the process based on these variables. The important components of a feedforward controller are essentially the same as for a feedback controller except that the sensor(s) obtains values of input or intermediate variables. It is valuable when lags in the process delay the effect of control changes.

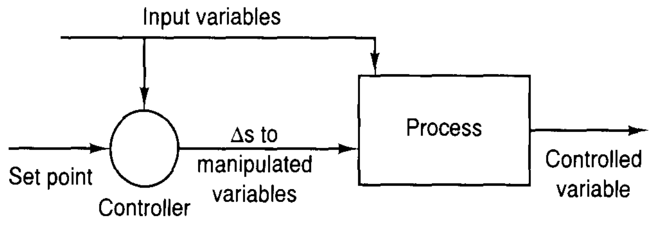


FIGURE 2.11 Feedforward Control

These are simplified models. They do not deal with complexities such as properties of sensors, transmission delays, and calibration issues. They ignore the response characteristics of the system, such as gain, lag, and hysteresis. They don't show how to combine feedforward and feedback or how to choose which process variables to manipulate. Chemical engineering provides excellent quantitative models for predicting how processes will react to various control algorithms; indeed there are a number of standard strategies [P+84, Section 22]. These are mentioned in Section 3.4.3, but a detailed discussion is beyond the scope of this book.

2.8.2 A Software Paradigm for Process Control

We usually think of software as algorithmic: we compute outputs (or execute continuous systems) solely on the basis of the inputs. This normal model does not allow for external perturbations; if noninput values of a computation change spontaneously, this is regarded as a hardware error. The normal software model corresponds to an open-loop system, and in most cases it is entirely appropriate. However, when the operating conditions of a software system are not completely predictable—especially when the software is operating a physical system—the purely algorithmic model breaks down. When the execution of a software system is affected by external disturbances—forces or events that are not directly visible to or controllable by the software—then a control paradigm should probably be considered for the software architecture.

An architectural style for software that controls continuous processes can be based on the process-control model, incorporating the essential parts of a process-control loop:

1. **Computational elements:** separate the process of interest from the control policy.
   - *Process definition*, including mechanisms for manipulating some process variables.
   - *Control algorithm* for deciding how to manipulate process variables, including a model for how the process variables reflect the true state.
2. **Data elements:** continuously updated process variables and sensors that collect them.
   - *Process variables*, including designated input, controlled, and manipulated variables, and knowledge of which can be sensed.

- *Set point*, or reference value for controlled variable.
- *Sensors* to obtain values of process variables pertinent to control.

3. **The control loop paradigm:** establishes the relation that the control algorithm exercises. It collects information about the actual and intended states of the process, and tunes the process variables to drive the actual state toward the intended state.

The two computational elements separate issues about desired functionality from issues about responses to external disturbances. For a software system, we can bundle the process and the process variables; that is, we can regard the process definition together with the process variables and sensors as a single subsystem whose input and controlled variables are visible in the subsystem interface. We can then bundle the control algorithm and the set point as a second subsystem; this controller has continuous access to current values of the set point and the monitored variables. For a feedback system, this will be the controlled variable. There are two interactions between these major systems: the controller receives values of process variables from the process, and the controller supplies continuous guidance to the process about changes to the manipulated variables.

The result is a particular kind of dataflow architecture. The primary characteristic of dataflow architectures is that the components interact by providing data to each other, each component executing when data is available. Most dataflow architectures involve independent (often concurrent) processes and pacing that depend on the rates at which the processes provide data for each other. The control-loop paradigm assumes further that data related to process variables is updated continuously. Moreover, unlike many dataflow architectures, which are linear, the control-loop architecture requires a cyclic topology. Finally, the control loop establishes an intrinsic asymmetry between the control element and the process element.

2.9 OTHER FAMILIAR ARCHITECTURES

There are numerous other architectural styles and patterns. Some are widespread, and others are specific to particular domains. While a detailed treatment is beyond the scope of this chapter, we briefly note a few of the important categories.

- **Distributed processes:** Distributed systems have developed a number of common organizations for multiprocess systems [And91]. Some can be characterized primarily by their topological features, such as ring and star organizations. Others are better characterized in terms of the kinds of interprocess protocols that are used for communication (e.g., heartbeat algorithms).

  One common form of distributed system architecture is a *client-server* organization [Ber92]. In these systems a server represents a process that provides services to other processes (the clients). Usually the server does not know in advance the identities or number of clients that will access it at run time. On the other hand, clients know the identity of a server (or can find it out through some other server) and access it by remote procedure call.

- **Main program/subroutine organizations**: The primary organization of many systems mirrors the programming language in which the system is written. For languages without support for modularization, this often results in a system organized