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.

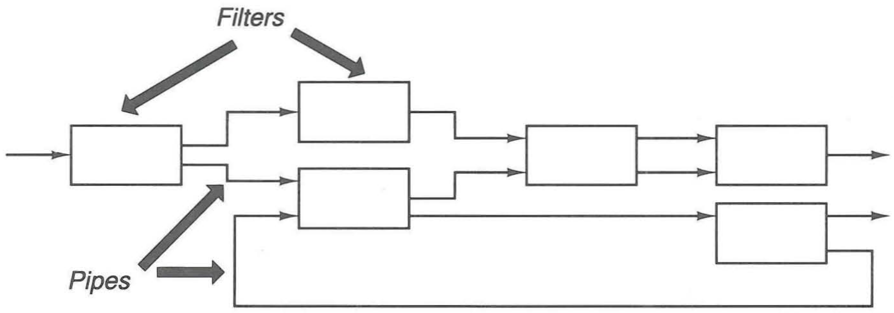

FIGURE 2.2 Pipes and Filters

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.