Host Integration Server 2004 | Microsoft patterns & practices proven practices for predictable results

Integration Patterns

Contents

Aliases

Data Flow Architecture

Context

You have an integration solution that consists of several financial applications. The applications use a wide range of formats—such as the Interactive Financial Exchange (IFX) format, the Open Financial Exchange (OFX) format, and the Electronic Data Interchange (EDI) format—for the messages that correspond to payment, withdrawal, deposit, and funds transfer transactions.

Integrating these applications requires processing the messages in different ways. For example, converting an XML-like message into another XML-like message involves an XSLT transformation. Converting an EDI data message into an XML-like message involves a transformation engine and transformation rules. Verifying the identity of the sender involves verifying the digital signature attached to the message. In effect, the integration solution applies several transformations to the messages that are exchanged by its participants.

Problem

How do you implement a sequence of transformations so that you can combine and reuse them independently?

Forces

Implementing transformations that can be combined and reused in different applications involves balancing the following forces:

- Many applications process large volumes of similar data elements. For example, trading systems handle stock quotes, telecommunication billing systems handle call data records, and laboratory information management systems (LIMS) handle test results.

- The processing of data elements can be broken down into a sequence of individual transformations. For example, processing XML messages typically involves a series of XSLT transformations.
- The functional decomposition of a transformation *f(x)* into *g(x)* and *h(z)* (where *f[x]=g[x]?h[z]*) does not change the transformation. However, when separate components implement *g* and *h,* the communication between them (that is, passing the output of *g[x]* to *h[z]*) incurs overhead. This overhead increases the latency of a *g(x)?h(z)* implementation compared to an *f(x)* implementation.

Solution

Implement the transformations by using a sequence of filter components, where each filter component receives an input message, applies a simple transformation, and sends the transformed message to the next component. Conduct the messages through *pipes* [McIlroy64] that connect filter outputs and inputs and that buffer the communication between the filters.

The left side of Figure 1 shows a configuration that has two filters. A source application feeds messages through the pipe into filter 1. The filter transforms each message it receives and then sends each transformed message as output into the next pipe. The pipe carries the transformed message to filter 2. The pipe also buffers any messages that filter 1 sends and that filter 2 is not ready to process. The second filter then applies its transformation and passes the message through the pipe to the sink application. The sink application then consumes the message. This configuration requires the following:

- The output of the source must be compatible with the input of filter 1.
- The output of filter 1 must be compatible with the input of filter 2.
- The output of filter 2 must be compatible with the input of the sink.

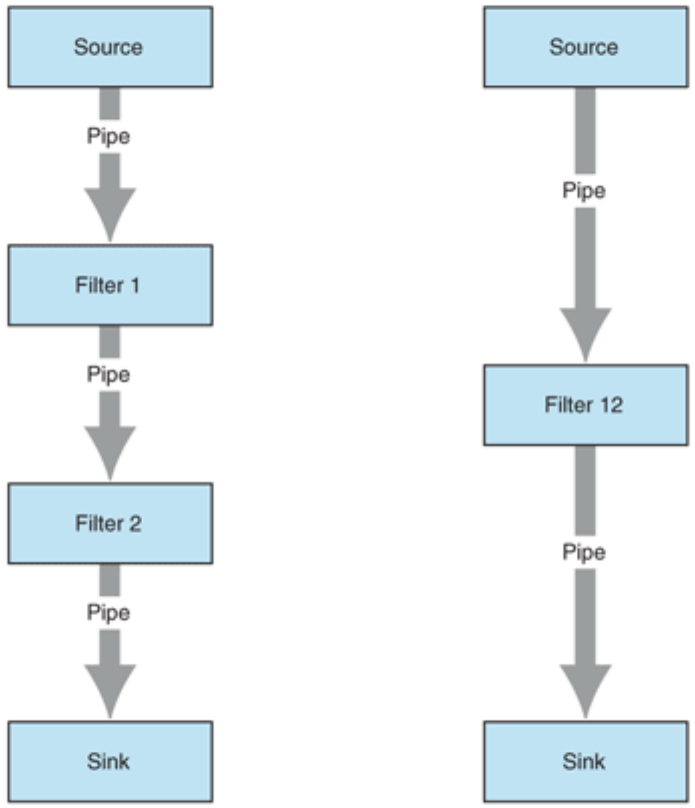

Figure 1. Using Pipes and Filters to break processing into a sequence of simpler transformations

The right side of Figure 1 shows a single filter. From a functional perspective, each configuration implements a transfer function. The data flows only one way and the filters communicate solely by exchanging messages. They do not share state; therefore, the transfer functions have no side effects. Consequently, the series configuration of filter 1 and filter 2 is functionally equivalent to a single filter that implements the composition of the two transfer functions (filter 12 in the figure).

Comparing the two configurations illustrates their tradeoffs:

- The two-filter configuration breaks the transformation between the source and the sink into two simpler transformations. Lowering the complexity of the individual filters makes them easier to implement and improves their testability. It also increases their potential for reuse because each filter is built with a smaller set of assumptions about the environment that it operates in.
- The single-filter configuration implements the transformation by using one specialized component. The one hop that exists

between input and output and the elimination of the interfilter communication translate into low latency and overhead.

In summary, the key tradeoffs in choosing between a combination of generic filters and a single specialized filter are reusability and performance.

In the context of pipes and filters, a transformation refers to any transfer function that a filter might implement. For example, transformations that are commonly used in integration solutions include the following:

- Conversion, such as converting Extended Binary Coded Decimal Interchange Code (EBCDIC) to ASCII
- Enrichment, such as adding information to incoming messages
- Filtering, such as discarding messages that match a specific criteria
- Batching, such as aggregating 10 incoming messages and sending them together in a single outgoing message
- Consolidation, such as combining the data elements of three related messages into a single outgoing message

In practice, the transfer function corresponds to a transformation that is specific enough to be useful, yet simple enough to be reused in a different context. Identifying the transformations for a problem domain is a difficult design problem.

Table 1 shows the responsibilities and collaborations that are associated with pipes and filters.

Table 1: Responsibilities and Collaborations of Pipes and Filters

Responsibilities	Collaborations
A filter takes a message from its input, applies a transformation, and sends the transformed message as output.	A filter produces and consumes messages.
A pipe transports messages between filters. (Sources and sinks are special filters without inputs or outputs.)	A pipe connects the filter with the producer and the consumer. A pipe transports and buffers messages.

Example

Consider a Web service for printing insurance policies. The service accepts XML messages from agency management systems. Incoming messages are based on the ACORD XML specification, an insurance industry standard. However, each agency has added proprietary extensions to the standard ACORD transactions. A print request message specifies the type of document to be generated, for example, an HTML document or a Portable Document Format (PDF) document. The request also includes policy data such as client information, coverage, and endorsements. The Web service processes the proprietary extensions and adds the jurisdiction-specific information that should appear on the printed documents, such as local or regional requirements and restrictions. The Web service then generates the documents in the requested format and returns them to the agency management system.

You could implement these processing steps as a single transformation within the Web service. Although viable, this solution does not let you reuse the transformation in a different context. In addition, to accommodate new requirements, you would have to change several components of the Web service. For example, you would have to change several components if a new requirement calls for decrypting some elements of the incoming messages.

An implementation that is based on *Pipes and Filters* provides an elegant alternative for the printing Web service. Figure 2 illustrates a solution that involves three separate transformations. The transformations are implemented as filters that handle conversion, enrichment, and rendering.

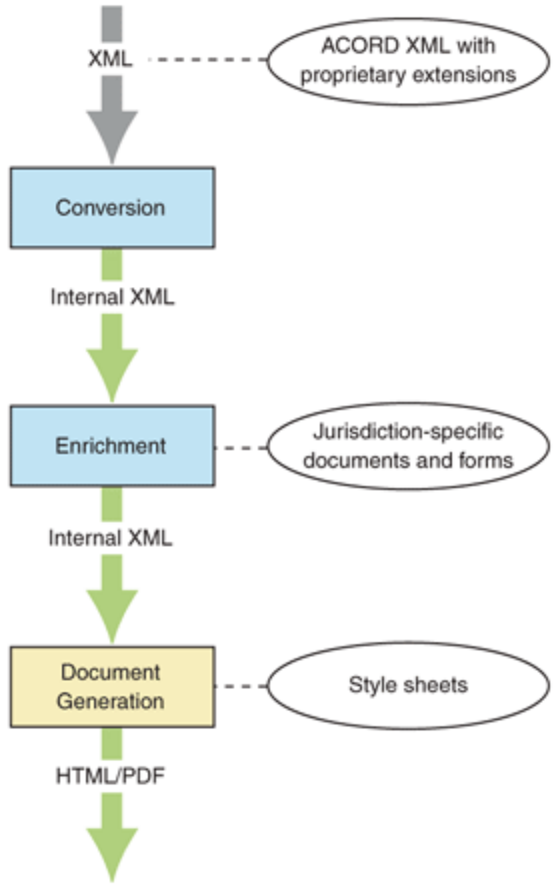

Figure 2. Printing Web service that uses Pipes and Filters

The printing service first converts the incoming messages into an internal vendor-independent format. This first transformation lowers the dependencies on the proprietary ACORD XML extensions. In effect, changing the format of the incoming messages only affects the conversion filter.

After conversion, the printing service retrieves documents and forms that depend on the jurisdiction and adds them to the request message. This transformation encapsulates the jurisdiction-specific enrichment.

When the message contains all the information that comprises the final electronic document, a document generation filter converts the message to HTML or PDF format. A style sheet repository provides information about the appearance of each document. This last transformation encapsulates the knowledge of rendering legally binding documents.

In this example, the *Pipes and Filters* implementation of the printing Web service has the following benefits that make it preferable to implementing the Web service as a single monolithic transformation:

- **Separation of concerns.** Each filter solves a different problem.
- **Division of labor.** ACORD XML experts implement the conversion of the proprietary extensions into an internal vendor-independent format. People who specialize in dealing with the intricacies of each jurisdiction assist with the implementation of the filter that handles those aspects. Formatters and layout experts implement document generation.
- **Specialization.** Document-rendering is CPU intensive and, in the case of a PDF document, uses floating point operations. You can deploy the rendering to hardware that meets these requirements.
- **Reuse.** Each filter encapsulates fewer context-specific assumptions. For example, the document generator takes messages that conform to some schema and generates an HTML or PDF document. Other applications can reuse this filter.

Resulting Context

Using *Pipes and Filters* results in the following benefits and liabilities:

Benefits

- Improved reusability. Filters that implement simple transformations typically encapsulate fewer assumptions about the problem they are solving than filters that implement complex transformations. For example, converting a message from one XML encapsulation to another encapsulates fewer assumptions about that conversion than generating a PDF document from an XML message. The simpler filters can be reused in other solutions that require similar transformations.
- Improved performance. A *Pipes and Filters* solution processes messages as soon as they are received. Typically, filters do not wait for a scheduling component to start processing.
- Reduced coupling. Filters communicate solely through message exchange. They do not share state and are therefore unaware of other filters and sinks that consume their outputs. In addition, filters are unaware of the application that they are working in.
- **Improved modifiability**. A *Pipes and Filters* solution can change the filter configuration dynamically. Organizations that use integration solutions that are subject to service level agreements usually monitor the quality of the services they provide on a constant basis. These organizations usually react proactively to offer the agreed-upon levels of service. For example, a *Pipes and Filters* solution makes it easier for an organization to maintain a service level agreement because a filter can be replaced by another filter that has different resource requirements.

Liabilities

- Increased complexity. Designing filters typically requires expert domain knowledge. It also requires several good examples to generalize from. The challenge of identifying reusable transformations makes filter development an even more difficult endeavor.
- Lowered performance due to communication overhead. Transferring messages between filters incurs communication overhead. This overhead does not contribute directly to the outcome of the transformation; it merely increases the latency.
- Increased complexity due to error handling. Filters have no knowledge of the context that they operate in. For example, a filter that enriches XML messages could run in a financial application, in a telecommunications application, or in an avionics application. Error handling in a *Pipes and Filters* configuration usually is cumbersome.
- Increased maintainability effort. A Pipes and Filters configuration usually has more components than a monolithic implementation (see Figure 2). Each component adds maintenance effort, system management effort, and opportunities for failure.
- Increased complexity of assessing the state. The Pipes and Filters pattern distributes the state of the computation across several components. The distribution makes querying the state a complex operation.

Testing Considerations

Breaking processing into a sequence of transformations facilitates testing because you can test each component individually.

Known Uses

The input and output pipelines of Microsoft BizTalk Server 2004 revolve around *Pipes and Filters*. The pipelines process messages as they enter and leave the engine. Each pipeline consists of a sequence of transformations that users can customize. For example, the receive pipeline provides filters that perform the following actions:

- The filters decode MIME and S/MIME messages.
- The filters disassemble flat files, XML messages, and BizTalk Framework (BTF) messages.
- The filters validate XML documents against XML schemas.
- The filters verify the identity of a sender.

The BizTalk Pipeline Designer allows developers to connect and to configure these filters within the pipeline. Figure 3 shows a pipeline that consists of Pre-Assemble, Assemble, and Encode filters. The toolbox shows the filters than can be dropped into this configuration.

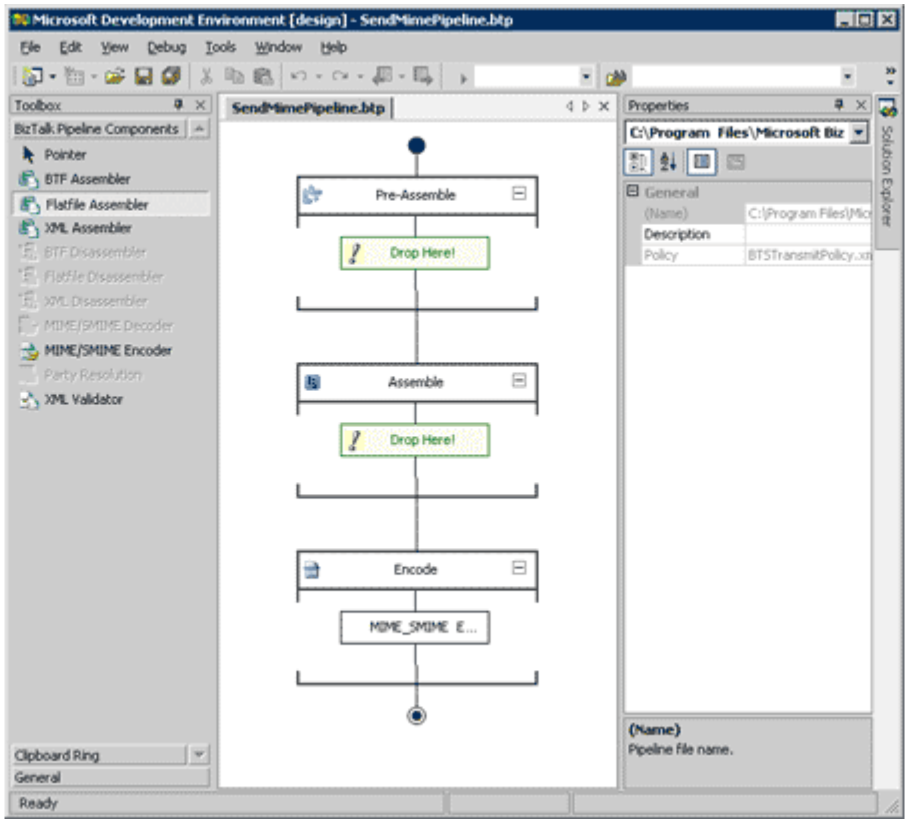

Figure 3. A Microsoft BizTalk Server2004 send pipeline in Pipeline Designer (Click the image to enlarge it)
Many other integration products use *Pipes and Filters* for message transformation. In particular, XML-based products rely on XSL processors to convert XML documents from one schema to another. In effect, the XSL processors act as programmable filters that transform XML.

Related Patterns

For more information about *Pipes and Filters*, see the following related patterns:

- Implementing Pipes and Filters with BizTalk Server 2004. This pattern uses the Global Bank scenario to show how you can use BizTalk Server 2004 to implement Pipes and Filters.
- *Pipes and Filters* [Shaw96, Buschmann96, Hohpe03].
- Intercepting Filter [Trowbridge03]. This version of *Intercepting Filter* discusses the pattern in the context of Web applications built

using the Microsoft .NET Framework. Developers can chain filters to implement preprocessing and post-processing tasks such as extracting header information and rewriting URLs.
- *In-band and Out-of-band Partitions* [Manolescu97]. This pattern remedies the lack of a component that has a global context in *Pipes and Filters* systems. The out-of-band partition is context-aware; therefore, it can configure the filters and handle errors.

Acknowledgments

[Buschmann96] Buschmann, Frank; Regine Meunier, Hans Rohnert, Peter Sommerland, and Michael Stal. *Pattern-Oriented Software Architecture*. John Wiley & Sons Ltd, 1996.

[Hohpe04] Hohpe, Gregor and Bobby Woolf, *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley, 2004.

[Manolescu97] Manolescu, Dragos. "A Data Flow Pattern Language," in *Proceedings of the 4th Pattern Languages of Programming*, September 1997, Monticello, Illinois.

[McIlroy64] The fluid-flow analogy dates from the days of the first UNIX systems and is attributed to Douglas McIlroy; see http://cm.bell-labs.com/cm/cs/who/dmr/mdmpipe.html.

[Trowbridge03] Trowbridge, David; Dave Mancini, Dave Quick, Gregor Hohpe, James Newkirk, and David Lavigne. *Enterprise Solution Patterns Using Microsoft .NET*. Microsoft Press, 2003. Also available on the *MSDN Architecture Center* at: http://msdn.microsoft.com/en-us/library/ms998469.aspx.

[Shaw96] Shaw, Mary, and David Garlan, *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

Start | **Previous** | **Next**

exist.