ThoughtWorks®
The art of heavy lifting.℠

Enterprise Integration Patterns with BizTalk Server 2004

Whitepaper

July 2004

Gregor Hohpe

Hsue-Shen Tham

Summary

Effective enterprise integration solutions rely on a number of important concepts, such as asynchronous messaging, orchestration, correlation, and long-running transactions. Unfortunately, these concepts are too often buried in vendor- and technology-specific vocabulary. Recently, design patterns have emerged as a way to capture and reuse design guidance outside of a specific technology context. Design patterns present known, good solutions to recurring design problems, described in a technology independent manner.

This whitepaper demonstrates how to use design patterns to effectively describe integration solution alternatives and guide design decisions. The patterns are applied to a simple example application, a loan broker scenario. Subsequently, the paper discusses implementation strategies for each pattern with BizTalk Server 2004. Finally, the paper guides the reader through a step-by-step implementation of the loan broker example.

The design patterns and the example application described in this paper are based on the book Enterprise Integration Patterns [EIP]. While a review of the book is helpful, it is by no means required (a summary of all integration design patterns can also be found at www.eaipatterns.com). The paper does, however, assume basic familiarity with BizTalk core concepts and features, for example as described in [CHAPPELL] or [BTSTUT].

Table of Contents

1 Enterprise Integration Patterns

As applications become more interconnected, enterprise integration has evolved from an esoteric niche market to an ubiquitous concern that permeates all types of application development. The broad success of Web services standards has further reduced the perceived gap between application development and integration.

Nevertheless, inherent properties such as network latency or limited control over applications drive important architectural differences between homogeneous applications and distributed integration solutions. These architectural differences necessitate design approaches that deviate from the familiar object-oriented design of monolithic applications. Therefore, understanding the underlying concepts is equally or even more important than studying a specific technology and its programming interfaces.

Software architecture remains a difficult area not at least because there is no "paint-by-numbers" approach where one can provide prescriptive step-by-step guidance to a complete solution. Rather, successful software architects use design principles and trade-offs to find the right solution between multiple alternatives. Unfortunately, it is difficult to describe these design concepts without diving into specific tool language and constructs. Also, developing a thorough understanding of these design principles often requires multiple years of experience. Even then, the principles remain rather abstract and it might not be obvious how to translate them into actionable guidance.

Luckily, design patterns have emerged as a useful way to narrow the gap between high-level design principles and tool-specific APIs. The notion of design patterns emerged from another discipline that does not provide for simple answers, the domain of building and town architecture. Christopher Alexander [ALEX] pioneered the idea of documenting design guidance in the form of patterns, proven solutions to recurring problems, over 25 years ago. A design pattern is not a firm rule that always applies. Rather, each pattern defines a specific context and discusses the forces that determine the right solution to the problem within this context. Because each pattern applies only within a defined context, a pattern is typically related to other patterns. A family of interrelated patterns forms a so-called pattern language.

Design patterns have long become a staple in the world of object-oriented design of application. The book Enterprise Integration Patterns [EIP] has recently applied the design pattern concept to enterprise integration solutions, with a particular focus on asynchronous messaging. The book describes a pattern language of 65 individual patterns, each of which is identified with a specific name and a visual representation, or icon. The patterns are organized into the following categories:

- **Channel Patterns** describe different types of message channels and show how to determine which channels a solution will need.
- **Message Construction Patterns** describe different ways messages can be used and how to form messages.
- **Routing Patterns** describe the routing of messages from the source to the destination.
- **Transformation Patterns** describe how to transform message content so that messages can be consumed.
- **System Management Patterns** show ways to test and monitor a running integration solution.

Each pattern is documented as a problem-solution pair as shown in the following example:

Message Filter

Problem: How can a component avoid receiving uninteresting messages?

Solution: Use a special kind of Message Router, a Message Filter, to eliminate undesired messages from a channel based on a set of criteria.

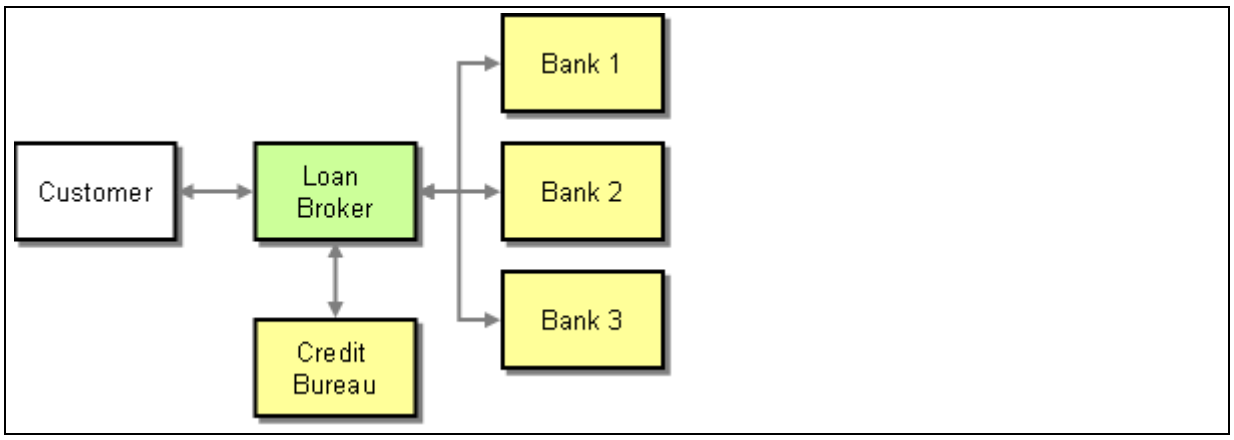
So how does one get started designing with patterns? Studying 65 patterns seems like a tall order. Fortunately, it is not required to know all patterns before starting to use them. This paper introduces the design patterns as they are used in the discussion of the example scenario. A Pattern Reference at the end of the paper summarizes all patterns mentioned in this paper. For a detailed description of all integration patterns visit http://www.eaipatterns.com.

2 Loan Broker Example

The best way to explain design principles and trade-offs is by means of a concrete example. For this paper we chose an example application from the domain of consumer lending: a loan broker providing interest rate quotes to a consumer.

2.1 Overview

In order to provide the best deal to the customer the loan broker requests quotes from multiple lenders (banks) and selects the best quote to pass back to the consumer. In order to speed up processing the loan broker also interacts with a credit bureau to determine the customer's credit worthiness. This way, the loan broker can retrieve the customer's credit rating once (the credit bureau typically charges a fee for each request) and pass the information along to each bank. The whole interaction is hidden from the customer, who simply provides his or her personal information and the terms of the desired loan. In return the customer receives the best interest rate the loan broker could obtain from the banks.



The Loan Broker Scenario

2.2 Business Requirements

The loan broker example is based on a simple set of business requirements as follows. The customer is expected to provide the following information to the loan broker:

- The customer's social security number (SSN; a unique identification number commonly used in financial transactions and identity theft)
- The desired loan amount in US Dollars
- The desired loan term in months

In return the loan broker replies with the following information:

- The interest rate of the best quote
- A unique Quote ID from the lender for future reference

Our example defines three banks, each of which specializes in a certain type of consumer. This allows each bank to streamline their processing and offer the best deal to customers in the target group. The banks expect the loan broker to pre-filter requests so that they are not inundated with quote requests for customers that do not match the bank's preferred customer profile. Each bank's preferences are expressed in terms of the customer's credit score rating and the duration of the customer's credit history. For our three banks, the preferences are as follows.

Bank 1 - Consumer Bank

This bank services a broad range of customers. Customers have to have a credit score of 500 or higher and have a credit history of at least 5 years.

Bank 2 - Exclusive Bank

This bank offers the best rates but specializes in top-end clientele. Exclusive Bank only services customers with a credit score of 700 and higher and a credit history of at least 10 years.

Bank 3 - Loan Shark

The name says it all. These guys will give a loan to just about anybody but charge a hefty premium.

The banks may use additional criteria internally to decide whether they will issue a quote. However, those criteria are not exposed to the loan broker. Even when a bank decides not to provide a quote it still has to reply to the loan broker stating that it is opting not to provide a quote. This way, the loan broker can distinguish a missing reply from the bank's decision not to provide a quote.

3 Designing with Patterns

This section describes how to design a solution that addresses the loan broker business requirements. The solution alternatives are expressed in the form of patterns. This allows us to discuss the solution without yet diving into the specifics of the technical implementation.

3.1 Loan Broker Tasks

Based on the business requirements we can break down the loan broker's responsibilities into the following tasks:

1. Receive requests from the customer
2. Obtain credit information from the credit bureau
3. Determine the appropriate banks to contact
4. Send a request for quote to each bank
5. Receive replies from each bank and determine the best quote
6. Pass the best quote back to the customer

Let's walk through these tasks one by one to see how we can use design patterns to express design considerations and trade-offs.

Step1: Receiving Requests

The loan broker should be able to process requests from multiple remote clients. Remote communication can involve a lot of complexity, including transport protocols, serialization, session management, and security. Separating the implementation of these functions from the actual business logic reduces complexity and improves opportunities for reuse. This consideration is the key driver behind the Service Interface pattern:

Service Interface **[ESP]**

Problem: How do you make pieces of your application's functionality available to other applications, while ensuring that the interface mechanics are decoupled from the application logic?

Solution: Design your application as a collection of software services, each with a Service Interface through which consumers of the application may interact with the service.

Choosing the Service Interface pattern leads to a few additional design decisions, such as whether the interaction is synchronous or asynchronous, how to correlate requests and responses, etc. In our example, we chose a synchronous interaction between customer and the loan broker. This implies that the customer waits until the loan broker returns a result. The main driver for this decision is simplicity. Because request and reply are part of a single interchange we do not have to worry about concepts such as *Correlation Identifier* [EIP] or *Return Address* [EIP]. Instead, these issues are addressed by the underlying transport protocol.

Step 2: Obtaining Credit Information

The banks require more information from the loan broker than the consumer provides. For example, the banks require the customer's credit score and the customer's credit history length. As a result the loan broker needs to gather additional information before it can pass the customer's request to the banks. The Content Enricher is a pattern that is tailor made for this purpose:

Content Enricher

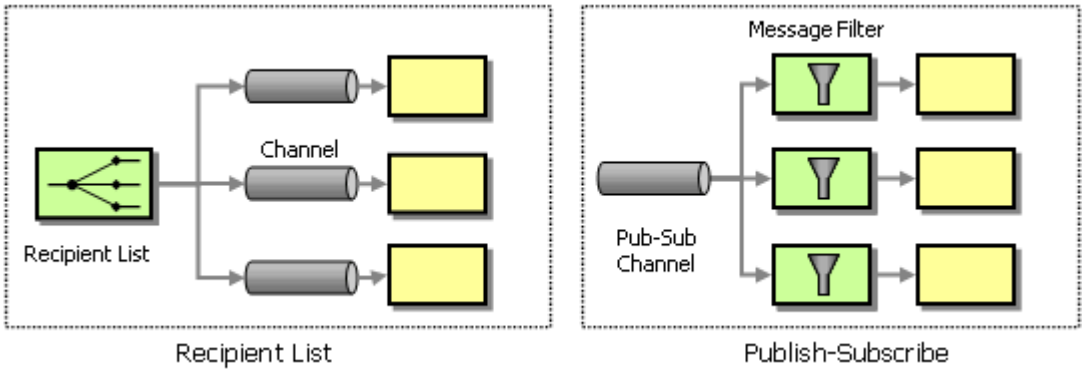
Problem: How do you communicate with another system if the message originator does not have all the required data items available?

Solution: Use a specialized transformer, a Content Enricher, to access an external data source in order to augment a message with missing information.

In our case the external data source is the credit bureau. The Content Enricher receives the quote request from the consumer, formulates a request to the credit bureau, and merges the credit bureau results with the original consumer request.

Step 3: Determine Appropriate Banks

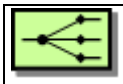
The business requirements mandate that the loan broker forward the enriched loan request only to appropriate banks for a quote. There are two fundamental approaches to implement this type of functionality (see figure):



Message Routing Options

We can either use a central Recipient List to forward the message to the appropriate channels or use a set of distributed Message Filters to eliminate unwanted messages.

A Recipient List pattern is a sophisticated form of the Message Router pattern [EIP]:



Recipient List

Problem: How do you route a message to a list of dynamically specified recipients?

Solution: Define a channel for each recipient. Then use a Recipient List to inspect an incoming message, determine the list of desired recipients, and forward the message to all channels associated with the recipients in the list.

The advantage of using a Recipient List is having central control over the message routing and the efficiency of routing messages only to those channels that are appropriate for the type of message. The main drawback is that the Recipient List is dependent on each potential recipient and their location. This can turn the Recipient List into a maintenance bottleneck.

A more open, but less controlled approach uses a Publish-Subscribe Channel and a series of Message Filters:

Publish-Subscribe Channel

Problem: How can the sender broadcast an event to all interested receivers?

Solution: Send the event on a Publish-Subscribe Channel, which delivers a copy of a particular event to each receiver.

Message Filter

Problem: How can a component avoid receiving uninteresting messages?

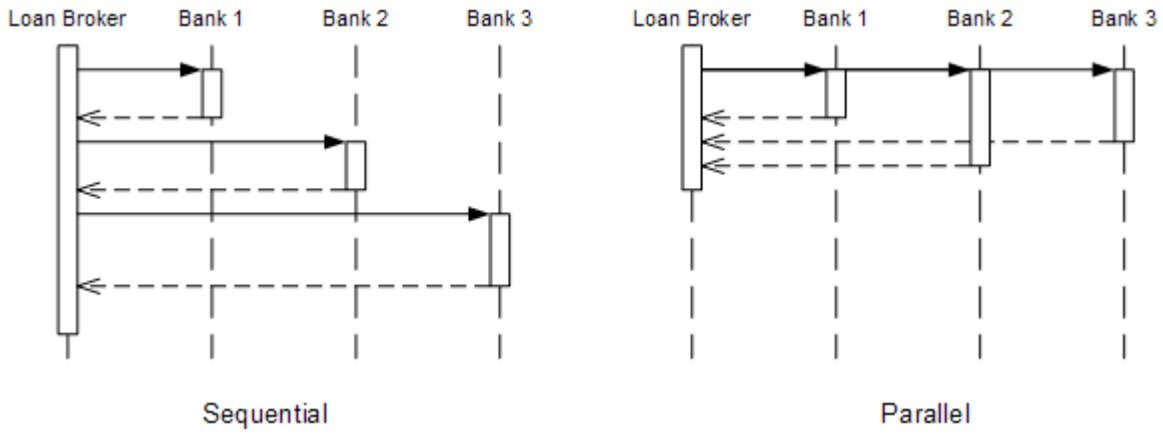
Solution: Use a special kind of Message Router, a Message Filter, to eliminate undesired messages from a channel based on a set of criteria.

In this alternative, each bank is free to subscribe to the Publish-Subscribe Channel without requiring any change inside the loan broker routing logic. The downside is that the routing logic is now spread out across a number of different Message Filter components. This can make it hard to determine the destinations for a specific message during test and debugging. Also, this approach is sometimes less efficient because the incoming message is sent to all filters just to be eliminated by those filters whose criteria do not match.

For our solution, we chose the implementation based on the Recipient List because it is the simpler solution and the business requirements do not call for rapid changes in banks or routing rules.

Step 4: Making Bank Requests

Now that the Recipient List determines the appropriate banks to contact with the quote request we have two choices on the timing of the actual requests. The loan broker can send the request to the first bank, wait for the reply, and then make the request to the second bank. This sequential style of interaction is simple because it avoids concurrent requests. However, the loan broker response times will likely be very slow because each bank is contacted in sequence. Alternatively, the loan broker could send concurrent, asynchronous requests to all eligible banks and process the replies as they come in. This parallel approach is slightly more complex but will improve the loan broker's response times. Instead of waiting for all three banks in sequence, the loan broker has to wait only for the slowest bank. If a quote request is sent to all three banks, this approach will be almost three times faster (see figure).

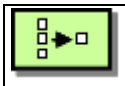


Comparing Sequential and Parallel Requests

Because the loan broker intends to provide swift service to its clients, we opt for the more efficient parallel interaction style.

Step 5: Processing Bank Replies

After the loan broker makes concurrent loan quote requests to all banks it needs to process the incoming replies. The business requirement states that the loan broker has to determine the lowest interest offer from all returned bank quotes. Condensing multiple messages into a single message is the specialty of the Aggregator pattern:



Aggregator

Problem: How do you combine the results of individual, but related messages so that they can be processed as a whole?

Solution: Use a stateful filter, an Aggregator, to collect and store individual messages until a complete set of related messages has been received. Then, the Aggregator publishes a single message distilled from the individual messages.

As described in [EIP], an Aggregator presents us with three primary design decisions:

Correlation. The loan broker needs to be prepared to service multiple customers' requests concurrently. This also means that bank reply messages arriving at the loan broker may belong to different customer's requests. The loan broker needs to correlate the proper replies belonging to the same customer into the same aggregate instance. We can choose from two primary implementation strategies for this correlation. We can use a synchronous request-reply protocol, such as SOAP over HTTP, which implements correlation inside the protocol. Alternatively, we can equip messages with *Correlation Identifiers* [EIP] and have the Aggregator perform explicit correlation. Our solution uses a synchronous communication style between the loan broker and the banks. This means that we do not have worry about correlation.

Completeness Condition. An Aggregator needs to be able to determine when enough replies have been received to process the aggregate. In this example, banks are required to return a reply even if they chose not to provide a quote. Therefore, the loan broker always receives the same number of replies as the number of requests it made. The synchronous request-reply protocol further simplifies this aspect because the number of connections opened by the recipient list tells the aggregator how many replies to expect.

Aggregation Algorithm. The aggregation algorithm determines the operation that the Aggregator performs once all reply messages have been received. In our case, the algorithm is straightforward; we choose the lowest interest rate of any bank quote within the aggregate.

Step 6: Reply to the Customer

Sending a reply message back to the customer is easy now that the Aggregator has determined the best quote. As mentioned above, we do not have to worry about correlation because the interaction between the customer and the loan broker is synchronous. This means that the connection from the customer to the loan broker will still be open by the time the Aggregator delivers the results.

It is likely, though, that the message format desired by the customers differs from the internal message format generated by the Aggregator. Therefore, we inject a *Message Translator* to perform the necessary transformation.

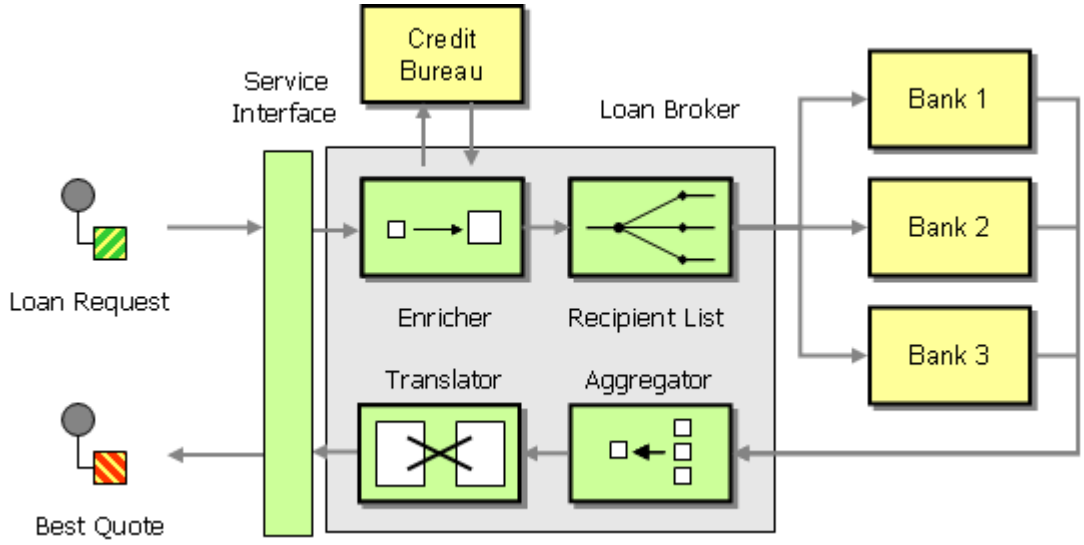
Message Translator

Problem: How can systems using different data formats communicate with each other using messaging?

Solution: Use a special filter, a Message Translator, between other filters or applications to translate one data format into another.

3.2 Solution Architecture

Describing the design alternatives and decisions with patterns enables us to visualize the loan broker solution architecture using the pattern images:



The Loan Broker Solution Architecture Expressed in Patterns

The loan broker receives requests via a *Service Interface*, and uses an *Enricher* to gather additional information required by the banks. A *Recipient List* sends request messages to the appropriate recipients while an *Aggregator* combines the replies into a single message. Finally, a *Message Translator* reformats the response into the form expected by the customer. By using patterns we are able to express our design decisions without having to dive deep into specific technologies or tools. The visual notation also allows us

to express the loan broker behavior in a simple, high-level picture. Furthermore, each pattern alerts us to additional design decisions or implementation strategies that we need to take into consideration. The following table summarizes the decisions.

Service Interface	
Interaction	Synchronous
Correlation	Not needed due to synchronous interaction
Recipient List	
Interaction	Synchronous
Request Sequencing	Parallel
Aggregator	
Correlation	Not needed due to synchronous interaction
Completeness Condition	Response received from all banks
Aggregation Algorithm	Lowest Interest Rate

3.3 Other Considerations

There are a number of topics we have not addressed yet, most prominently transactions and error handling. Essentially, the loan broker acts as a gatherer of information, that means, none of the loan broker's interactions with external parties require updates. Therefore, transactional support across the interactions is not required. The fact that all the interactions are read-only also means that the loan broker is an *Idempotent Receiver*.

Idempotent Receiver

Problem: How can a message receiver deal with duplicate messages?

Solution: Design a receiver to be an Idempotent Receiver--one that can safely receive the same message multiple times.

As a result, our primary form of error handling is to retry the action. If the loan broker cannot deliver a reply to the customer, the customer can simply retry the quote request.

4 Implementation Strategies

Now that the loan broker design is expressed in terms of technology independent patterns we can start mapping the patterns onto the implementation technology platform, BizTalk Server 2004. As we will find out, BizTalk Server 2004 already implements a number of the patterns, greatly simplifying the transition from pattern-based design to running example.

4.1 Basic Technology Choices

Before we start diving into the implementation of the individual patterns, we have to make a few technology decisions that will impact our implementation choices.

Web Services vs. Remoting vs. Messaging

The loan broker example requires remote communication between multiple systems, such as the customer, the loan broker, the credit bureau, and the banks. The Microsoft platform provides a number of tools and protocols for this purpose, including .NET Remoting, MSMQ, and Web services. Each of these technologies has unique features and advantages. For example, .NET Remoting provides a rich object-oriented interaction model whereas MSMQ excels in asynchronous, message-oriented communication. Web Services are based on platform-independent standards and enable interoperability across multiple vendor technologies.

Even though our example implements all components of the loan broker scenario using Microsoft technologies, in a real-world scenario it is likely that not all components would share the same platform. Therefore, we decide to connect all components using Web services.

Synchronous vs. Asynchronous Interaction

Traditionally, Web services have been associated with remote-procedure-call (RPC)-like interaction between service consumer and provider. RPC-style interaction is inherently synchronous, i.e. the consumer will block and wait for a response from the provider. However, Web services can also be used to implement asynchronous interaction where the consumer does not have to wait for the reply. Instead, the service provider can call the consumer ("call back") once the results of the operation are available.

Each alternative has pros and cons. Synchronous interaction is simple because it does not require the consumer to correlate the reply with the original request. It is also easier to test and debug because the consumer executes within a single thread of execution. The downside of synchronous interaction is twofold. First, the consumer is blocked while it is waiting for the reply and cannot perform any other tasks. Second, the consumer maintains an active connection to the provider for the duration of the whole interaction. This can cause contention for resources on the provider side because the service has to support many concurrent connections established by various consumers.

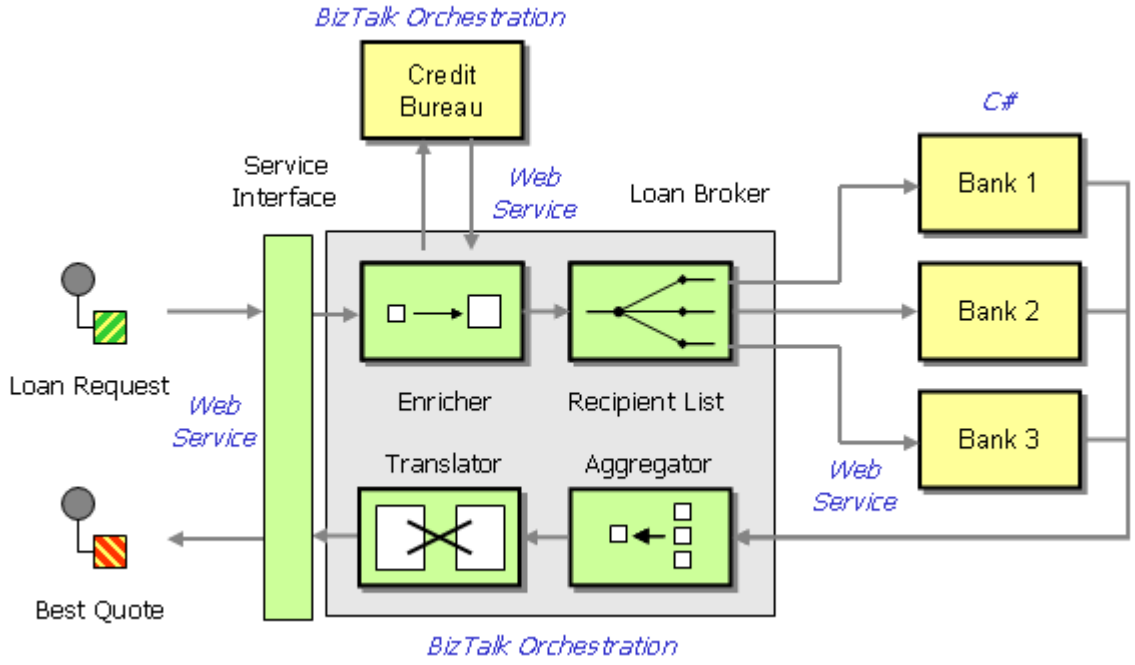
This example solution focuses primarily on the implementation of the loan broker component, such as the Recipient List and the Aggregator. Therefore, we choose a synchronous interaction between all components to keep the interaction model as simple as possible. For this example we do not have to worry too much about resource contention because the BizTalk Server Web services interface is implemented as an ASP.NET Web service running inside the Internet Information Services (IIS), which is designed to handle large numbers of concurrent client connections.

BizTalk Server Orchestration vs. C#

Because we use Web services as the communication technology we can choose from multiple implementation alternatives for each of the components. For example, we could easily implement the bank services as either a C# class or as a BizTalk Orchestration.

A BizTalk Orchestration is the natural choice for the loan broker component because the loan broker's task is to orchestrate the interaction between multiple external parties. The bank and the credit bureau implementations are essentially "dummies", so that either a C# or a BizTalk Orchestration is a viable choice. To demonstrate both alternatives we implement the credit bureau using a BizTalk orchestration and the banks as a set of C# classes.

The following figure adds the technology choices to the loan broker solution architecture.

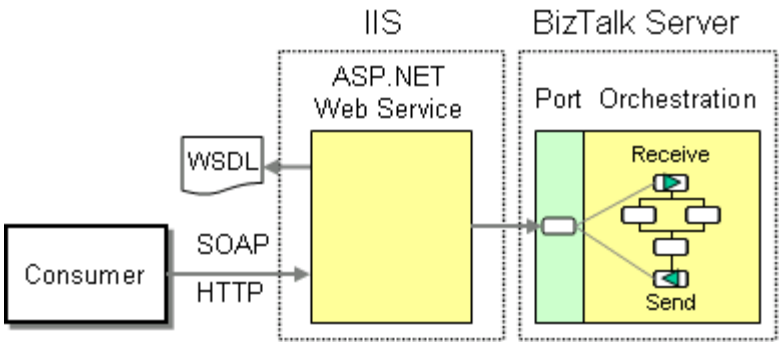


The Loan Broker Solution Architecture with Technology Choices

With the basic technology choices in place we can start to map each of the five patterns to the BizTalk implementation platform. It turns out that each pattern has a generic representation in BizTalk server. This allows us to discuss the mapping largely without referring to the specifics of the loan broker example, highlighting the reuse potential of the concepts represented by the design patterns.

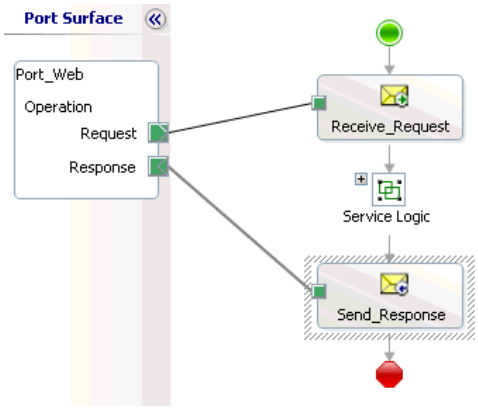
4.2 Service Interface with BizTalk Server 2004

A service has to provide a service interface to enable consumers to access the service. Straight out of the box, BizTalk Server 2004 provides the capabilities to expose an orchestration via a Web service interface as shown in the figure below.



Implementing Service Interface in BizTalk Server 2004

A BizTalk orchestration can be exposed as an ASP.NET Web service using the BizTalk Web Services Publishing Wizard. The wizard generates an ASP.NET Web service based on the orchestration's interface. The ASP.NET Web service acts as the Service Interface between the consumer and the orchestration, rendering the service contract in form of a WSDL document and acting as an entry point for incoming requests to the service. The orchestration receives Web service requests via a logical request-reply port and the Receive and Send shapes (see figure):



Implementing Service Interface in BizTalk Server 2004

The following steps are required to build and deploy an orchestration as a Web service:

1. **Define message schemas for inbound and outbound messages** – The inbound and outbound message schemas specify the format of the request and reply messages for the service.
2. **Define a logical request-response port –** Orchestrations interface with the outside world through logical ports that are independent of physical parameters such as IP addresses or file names. The logical ports are bound to physical ports at deployment time.
3. **Design the orchestration**
   - Create a message variable for the request message and a message variable for the reply message. Each variable uses the respective schema in defined in step 1.
   - Add a Receive shape to the orchestration and configure it to use the request message variable. Connect the Receive shape to the Request operation of the logical port.
   - Add a Send shape to the orchestration and configure it to use the reply message variable. Connect the Send shape to the Response operation of the logical port.

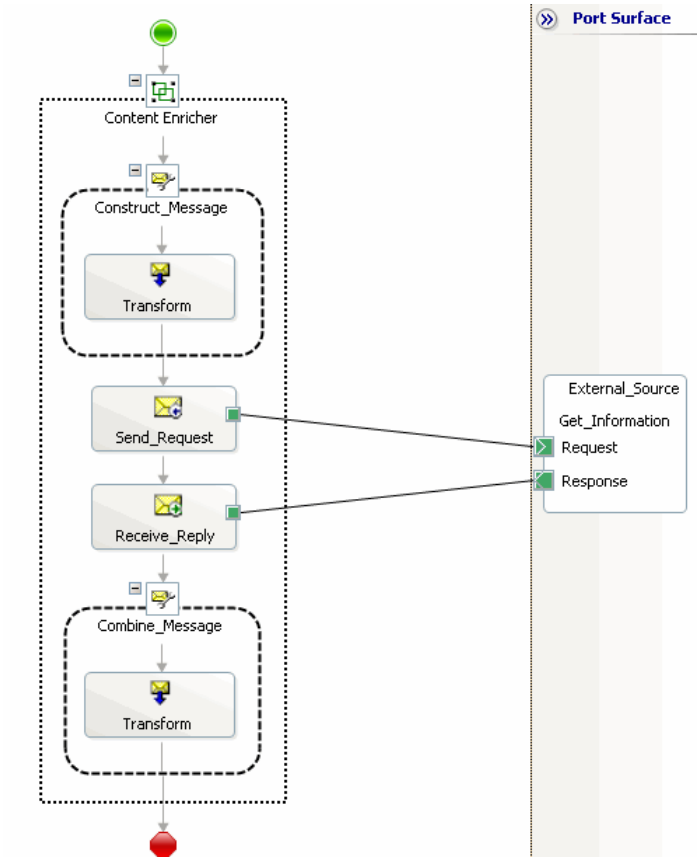
4. **Build and deploy the orchestration** – Before running the BizTalk Web Services Publishing Wizard, the orchestration has to be compiled and deployed into the Global Assembly Cache.
5. **Run the BizTalk Web Services Publishing Wizard** – Once deployed, the orchestration can be published as a Web service using the BizTalk Web Services Publishing Wizard. The BizTalk Web Services Publishing Wizard generates both the ASP.NET Web service and the physical Web port(s).
6. **Bind the orchestration's logical port to the physical port** – From the BizTalk Explorer, bind the orchestration's logical port to the physical port generated by the wizard.
7. **Enlist and start the orchestration** – Finally, enlist and start the orchestration in BizTalk Administrator, so that it can start processing request messages.

4.3 Content Enricher with BizTalk Server 2004

A Content Enricher's role is to collect additional information when an incoming message does not contain all the fields required by a subsequent processing step. In order to gather the missing information, the Content Enricher accesses an external data source. Consequently, the Content Enricher has to perform the following four steps:

- Construct a request message
- Send the request message to the external resource
- Receive a response from the external resource
- Combine the response from the external resource with the original message

Each step can be expressed as an action shape inside a BizTalk orchestration:



Implementing a Content Enricher in BizTalk Server 2004

The Transform shape transforms the incoming message into the format required by the external resource. The Transform shape always has to be embedded inside a Construct Message shape because it assigns a new message instance to the specified message variable. The Send and Receive shapes manage the interaction with the external resource. The shapes are connected to a logical port, which in turn can be bound to a variety of physical transport protocols, such as SOAP, MSMQ etc. Lastly, another Transform shape merges the response from the external resource with the original message to the format the required by the following processing step. A group shape wraps around the individual shapes to indicate that the shapes collaborate to perform a coordinated function.

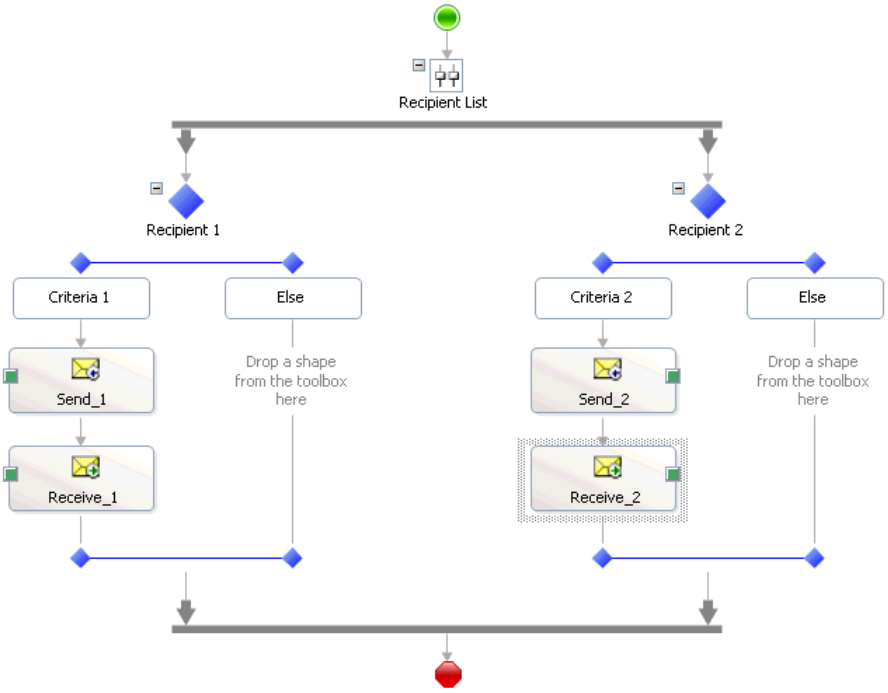
4.4 Recipient List with BizTalk Server 2004

As described in the previous section, a *Recipient List* takes a single message and forwards it to a number of recipients based on the message content or other criteria. Logically, the *Recipient List* consists of two steps: first, the *Recipient List* has to determine the list of appropriate destinations and then it has to forward the message to those destinations.

BizTalk Server 2004 provides a number of mechanisms to implement this type of functionality. Which option is the best fit depends on factors such as how 'dynamic' the list of recipients is (i.e., how often new recipients join or existing ones leave), how frequently the rules for message routing change, and what the mode of interaction between the recipients is (i.e., synchronous vs. asynchronous). Let's look at three alternative implementation options.

Parallel Actions

If the Recipient List interacts with the recipients in a synchronous manner, the Parallel Actions shape is a natural candidate. This shape allows the execution of multiple concurrent activities inside an orchestration. This way, the orchestration can make a synchronous request to each intended recipient simultaneously by using the Send and Receive shapes. Because the intend of a Recipient List is to forward an incoming message only to a select subset of recipients, each branch of the Parallel Actions shape contains a Decide shape that evaluates whether the message should be sent to the recipient represented by the respective branch (see figure). The Else branch of the Decide shape remains empty because no action is required if the recipient's criteria are not matched.

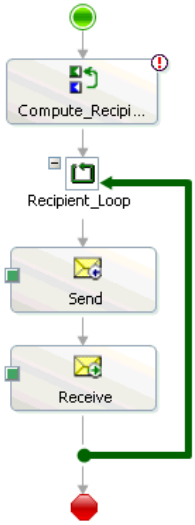


Implementing a Recipient List using the Parallel Actions shape

The primary limitation of this implementation strategy is the fact that the number of potential recipients is fairly static. When a new recipient is identified, the orchestration has to be modified by adding another branch to the Parallel Actions shape. Also, the routing logic is distributed across multiple Decide shapes, which can make debugging and maintenance of the routing rules more difficult.

Loop

A more flexible way to implement a *Recipient List* inside an orchestration is to use a Loop shape. In this case, an Expression shape computes the list of the intended recipients. Subsequently, a loop sends the messages to all recipients in the list (see figure).



Implementing a Recipient List using the Loop shape

This approach is more flexible because all the routing logic is contained in a single shape. For example, you could replace the Expression shape with a Call Rules shape to take advantage of the BizTalk Rules Engine. Also, if you connect the Send and Receive shapes to a dynamic port, you can more easily add new recipients without modifying the orchestration. The main limitation lies in the fact that the Loop shape sends the messages in a sequential order, which can be inefficient if the interaction between the Recipient List and the recipients is synchronous because the next request is made only after a response to the previous request has been received. Therefore, a loop is better suited for asynchronous communication with the recipients.

Message Box

A third option is to use the BizTalk message box, which provides publish-subscribe capabilities. This approach can result in functionality equivalent to a Recipient List but the implementation more closely resembles the *Publish-Subscribe Channel* plus *Message Filter* approach described earlier. The message box approach works best for one-way messages with a very dynamic set of recipients.

The following table summarizes the pros and cons of the implementation strategies.

	Parallel Actions	Loop	Message Box
Suitable for Synchronous Interaction	Yes	Somewhat (less efficient)	No
Easy to add / remove recipients	No	Yes	Yes
Routing Logic	Distributed	Consolidated	Distributed

4.5 Aggregator with BizTalk Server 2004

The role of an *Aggregator* is to collect a sequence of incoming messages and to consolidate them into a single message. An *Aggregator* is frequently used in conjunction with a *Recipient List*, a combination referred to as a *Scatter-Gather* [EIP]. In that context, the best implementation strategy for the Aggregator naturally depends on the implementation chosen for the Recipient List.

Scatter-Gather

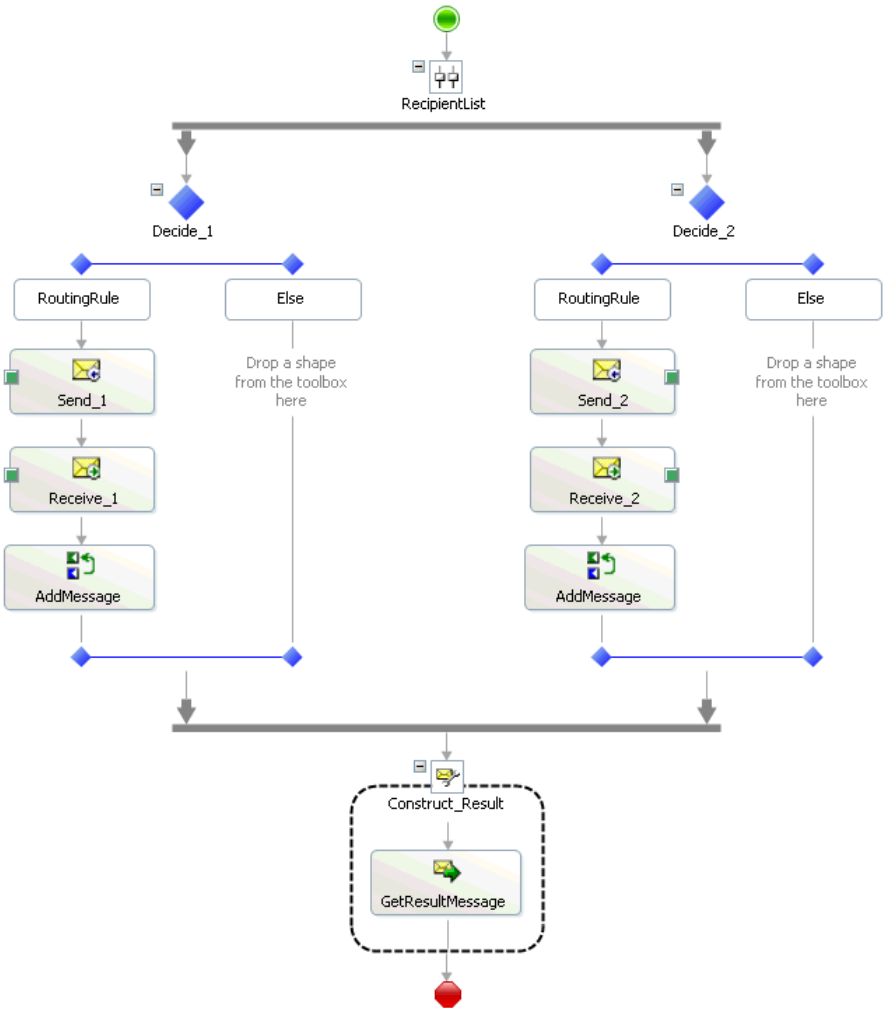
Problem: How do you maintain the overall message flow when a message needs to be sent to multiple recipients, each of which may send a reply?

Solution: Use a Scatter-Gather that broadcasts a message to multiple recipients and re-aggregates the responses back into a single message.

Fundamentally, the implementation choices for the Aggregator resemble those of the Recipient List, that is, you can use the Parallel Actions shape or a Loop shape. The trade-offs are also quite analogous. A Parallel Actions shape works well for synchronous interaction but is less flexible. A Loop shape works better with asynchronous messaging and is more dynamic. If the Recipient List is implemented using a Parallel Actions shape it makes sense to include the Aggregator functionality in the same shape.

An Aggregator typically collects message data from multiple messages into a collection. Once all necessary messages have been received, the Aggregator computes the resulting message from the collection. One potential challenge is that BizTalk orchestrations do not directly support collections. However, this function can be easily implemented in a C#

class that can be called from the orchestration via the Expression shape (see the AddMessage shapes in the figure).



Implementing an Aggregator using the Parallel Actions shape

When used in this context, an Aggregator C# class needs to implement a simple interface with two methods:

```
public interface Aggregator
{
    void AddMessage(XmlDocument document);
    XmlDocument GetResultMessage();
}
```

The semantics of the methods are rather straightforward. AddMessage() adds a received message into the aggregate whereas GetResultMessage() returns the aggregated response message. BizTalk messages are actually XML documents. That's why the methods take parameters of type System.Xml.XmlDocument.

An Aggregator used in a more dynamic context (for example, in a Loop) also needs to implement the following method:

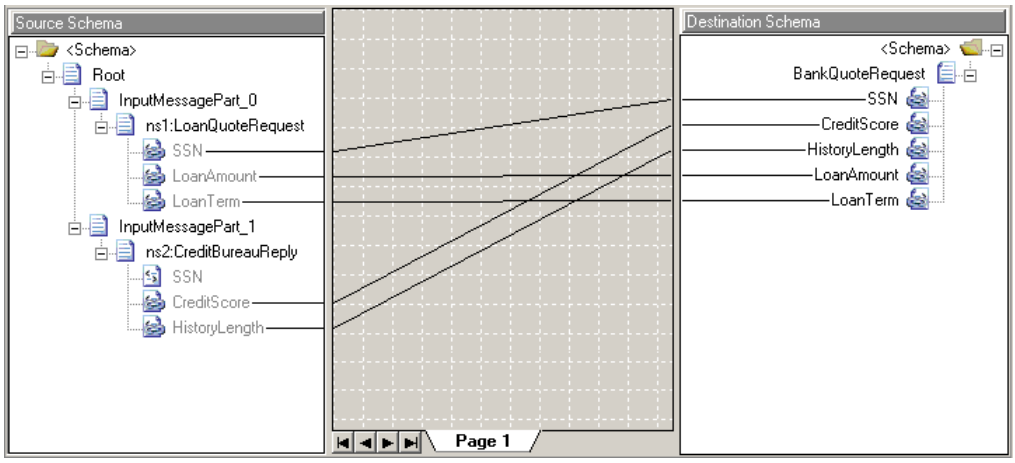
```
    bool IsComplete();
```

This method indicates whether the Aggregator has received a sufficient stream of input messages in order to render the result message. When using the Parallel Actions shape this method is typically not needed because the synchronization point at the bottom of the shape ensures that all parallel branches have completed before the GetResultMessage() method is called. An IsComplete() method might still be useful in those cases where timeouts or exceptions could terminate some branches without receiving a message.

The implementation of these methods depends on the Completeness Condition and the Aggregation Algorithm specified in the Aggregator pattern. In the loan broker example, the implementation is quite straightforward as the Aggregator only has to store the best bank quote received so far.

4.6 Message Translator with BizTalk Server 2004

A Message Translator converts messages of one format into another format. Message translation is a very common function in integration solutions because most existing systems define proprietary data and message formats that are not compatible with other systems' formats. BizTalk Server 2004 includes a visual BizTalk Mapper editor that is integrated into the Visual Studio .NET development environment. The BizTalk Mapper loads the source and target schema(s) and allows the user to map fields from one schema to the other via drag-and-drop. More complex transformation functions can be implemented by injecting standard or custom functoids.



5 Implementing the Example

The previous section describes how to implement the general patterns used in the loan broker scenario with BizTalk Server 2004. Based on these generic implementations, this section provides detailed step-by-step guidance on how to implement the loan broker functionality in BizTalk Server 2004. This section assumes that you are familiar with the basic concepts of BizTalk server (see [CHAPPELL]) and have basic working knowledge with the BizTalk development environment, for example by working through the excellent BizTalk tutorial [BTSTUT].

We decide to implement the credit bureau and bank services first because these services are self-contained, that is, they have no external dependencies. Also the implementation of these components is straightforward because both services are just mock-ups that simulate drastically simplified behavior. For simplicity's sake, we implement all components of the loan broker scenario inside a single Visual Studio solution.

5.1 Designing the Credit Bureau

The first external party that the loan broker interacts with is the credit bureau. A credit bureau performs credit history check services for third parties by keeping records of individuals' credit history based on their identification. The bureau provides a measure of the consumer's credit worthiness in a form of a numerical credit score. This credit score and the duration of the credit history are key factors used by lenders to determine the risk that a consumer presents.

The loan broker utilizes this service by submitting a request containing the applicant's social security number (SSN) to the credit bureau service. In return, it expects the credit score and the length of the credit history for the individual.

In reality, credit bureaus use complex mathematical models to compute the credit score. For simplicity sake, our example simulates the credit bureau's behavior by just generating a random number.

We utilize the Service Interface pattern to define the contract and interaction style for the service. The decision to expose the service as a web service gives us the option to choose from multiple implementation technologies, such as C#, Java or as a BizTalk server orchestration. To take advantage of BizTalk 2004 Web service capabilities, we choose to implement the service as a BizTalk Server orchestration and then expose it as a Web service.

Implementation

The following steps are involved in constructing the credit bureau. As expected, these steps very much resemble the steps described in the section Service Interface with BizTalk Server 2004.

1. Define the message schemas for all inbound and outbound messages
2. Define the logical request-response port as input and output to the orchestration
3. Define message variables in the orchestration with types referencing the message schemas
4. Add a Receive shape in the Designer to receive request messages
5. Construct a reply message using a Transform shape and a BizTalk map
6. Send the reply message to the response port using a Send shape
7. Build and deploy the orchestration into the Global Assembly Cache (GAC)

8. Publish the orchestration as a Web service using BizTalk Web Services Publishing Wizard
9. Bind the logical port to the generated physical web port
10. Enlist and start the orchestration

Step 1: Define Message Schemas

We start by creating the schemas for inbound request and reply messages for the credit bureau service. The schemas define the message format for inbound and outbound messages used for the credit bureau service's conversation. When the orchestration is published using the BizTalk Web Services Publishing Wizard, these schemas define the SOAP request and reply message format.

In BizTalk 2004, we define the schemas using the BizTalk Schema Editor in Visual Studio .NET. Begin by making a new empty BizTalk Server project and adding two new XML Schema files to the project in Visual Studio .NET. Name the schemas 'CreditBureauRequest' and 'CreditBureauReply'. Edit the schemas and add the following elements according to the table:

CreditBureauRequest.xsd

Namespace: http://www.microsoft.com/biztalk/creditbureau

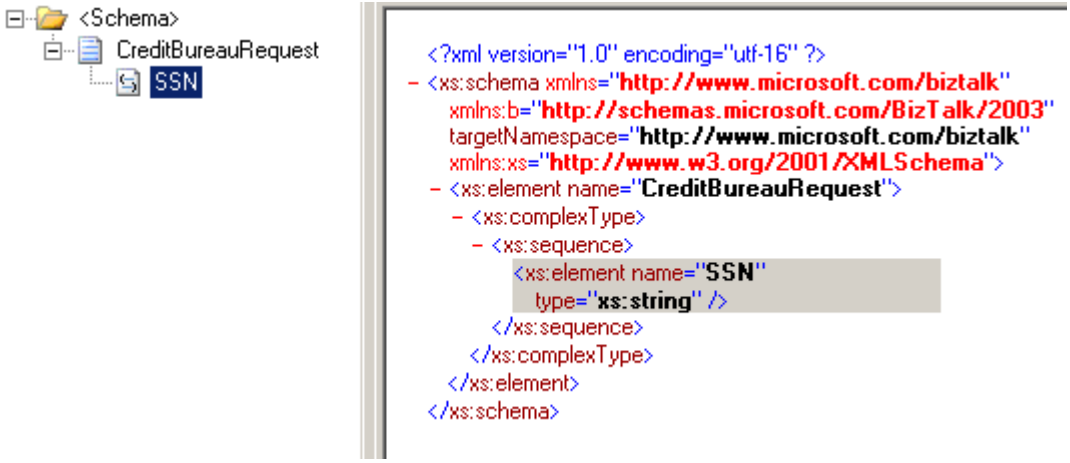
Name	Element	Type
CreditBureauRequest	Root Node	-
SSN	Child Field Element	xs:string

CreditBureauReply.xsd

Namespace: http://www.microsoft.com/biztalk/creditbureau

Name	Element	Type
CreditBureauReply	Root Node	-
SSN	Child Field Element	xs:string
CreditScore	Child Field Element	xs:int
HistoryLength	Child Field Element	xs:int

The request message schema should look as follows in the BizTalk Schema Editor:



Editing a Schema in the BizTalk Schema Editor

Note that the schemas do not have to include a correlation ID because the service is accessed only in a synchronous request-reply pattern.

Step 2: Define Logical Request-Response Ports

In BizTalk, ports are the point of entry and exit to an orchestration process. Logical request-response ports are defined at design time, representing logical locations for receiving and sending messages.

Add a new Orchestration to the project in Visual Studio .NET and name it 'CreditBureauProcess'. In BizTalk Orchestration Designer, add a new configured port to the orchestration. In the Port Configuration Wizard configure the port as a public request-response port according to the values specified in the following table:

Port Configuration

Properties	Value
Name	Port_CreditBureau
Existing or New Port Type	New
Port Type	PortType_CreditBureau
Communication Pattern	Request-Response
Access Restrictions	Public – no limit
Port direction of Communication	I'll be receiving a request and sending a response
Port Binding	Specify Later – the physical port will be created later by BizTalk Web Services Publishing Wizard

Rename the Operation of the port from Operation_1 to GetCreditScore.

The access restriction for the port needs to be public so that it can be exposed as a Web port when deployed as a Web service using BizTalk Web Service Wizard.

Step 3: Design the Orchestration

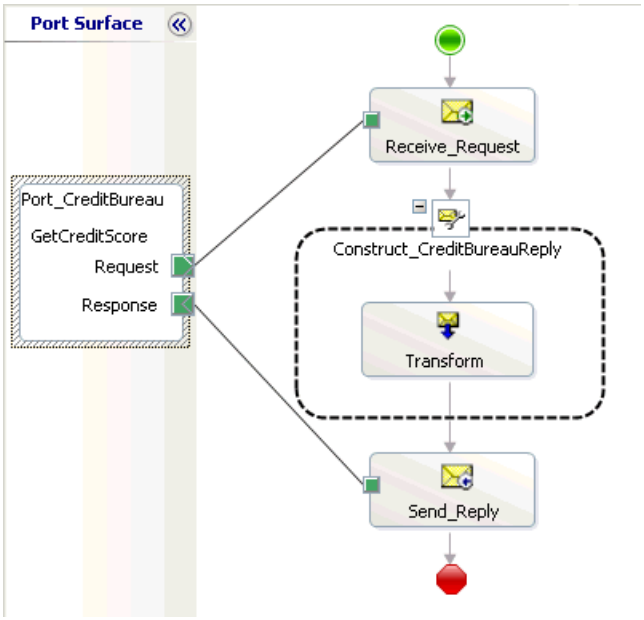
First we define message variables that are used by the orchestration. Each message variable references a specific message schema as the message type. The message variables represent message instances of the message schema type.

To add message variables in BizTalk 2004, switch to the Orchestration View in Visual Studio .NET, right click on the Message node and select "New Message". The credit bureau orchestration requires only the following two message variables:

Message Variables

Identifier	Message Type
CreditBureauRequestMsg	Schemas – CreditBureauRequest
CreditBureauReplyMsg	Schemas – CreditBureauReply

Once the variables are defined, we can start building up the rest of the orchestration.



Credit Bureau Orchestration

BizTalk orchestration designs typically start with a Receive shape connected to an incoming request port. A request message comes in through the request port and goes into the Receive shape to initiate the BizTalk orchestration process.

In the Orchestration Designer, add a Receive Shape from the toolbox to the Designer surface. Set the properties of the Receive Shape as follows:

Receive shape

Properties	Value
Name	Receive_Request
Activate	True
Message	CreditBureauRequestMsg
Operation	Connect the Receive shape to Port_CreditBureau Request operation

We do not need to use correlation sets in this case because the service is only accessible via synchronous request-reply.

Since we get a request, we will need to generate a reply message. As mentioned before, the credit bureau is just a 'dummy' service that generates a random credit score and history length. We simulate this process using a BizTalk Map created using the BizTalk Mapper in Visual Studio .NET. The BizTalk Mapper facilitates the transformation from one or more message schemas to another type of schema. To generate the reply for the credit bureau service, we map some of the existing fields in the original request message to equivalent fields in the reply schema, while the credit score and history length fields are generated randomly using scripting functoids.

In order to add a transformation map to the orchestration, add a Construct Message shape from the toolbox after the Receive shape. Again, edit and configure the shape with the following values:

Construct Message shape

Properties	Value
Name	Construct_CreditBureauReply
Message Constructed	CreditBureauReplyMsg

Then add a Transform shape into the Construct Message shape and rename it to 'Transform'. Double click on the Transform shape to initiate the Transform Configuration Wizard. Step through the wizard using the following configuration values:

Transform shape

Properties	Value
New or Existing Map	New
Fully Qualified Map Name	CreditBureau.GenerateRandomReply
Source	CreditBureauRequestMsg
Destination	CreditBureauReplyMsg

By default, when you specify that you want to create a new map, the checkbox "When I Click OK, launch the BizTalk Mapper" is checked. Click OK to launch the BizTalk Mapper to edit the map. Select the Scripting functoids from the Advanced Functoids tab of the Mapper toolbox and set the Script Type to Inline C#.

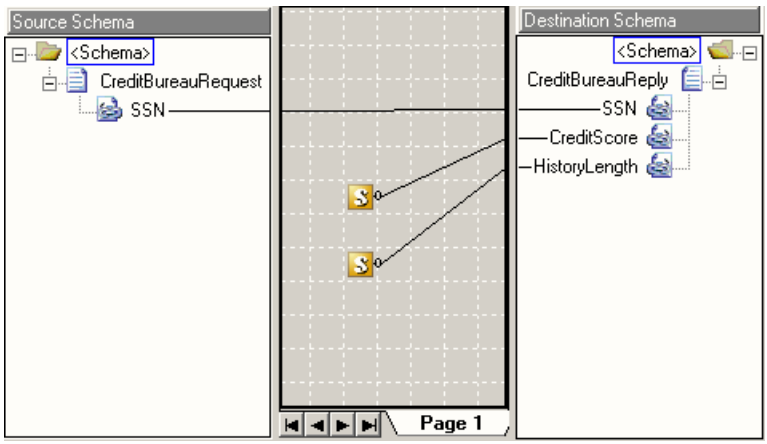
CreditBureau.GenerateRandomReply Map

Target	Functoids	Source
SSN	-	SSN

CreditScore	Inline C# Scripting Functoid:
	`public int GetCreditScore()`
	`{`
	`  Random random = new Random();`
	`  return (int)(random.Next(600) + 300);`
	`}`	

HistoryLength	Inline C# Scripting Functoid:
	`public int GetHistoryLength()`
	`{`
	`  Random random = new Random();`
	`  return (int)(random.Next(19) + 1);`
	`}`	

The configured Map looks as follows:



Generating Credit Bureau Reply message

Finally, send the reply message constructed off via the response port. In the Orchestration Designer surface, add a Send shape after the Construct Message shape and edit as follows:

Send shape

Properties	Value
Name	Send_Reply
Message	CreditBureauReplyMsg
Operation	Connect the Send shape to Port_CreditBureau Response operation

Step 4: Build and Deploy the Orchestration

Only assemblies with a strong name key can be deployed to the Global Assembly Cache (GAC). Generate a strong name key file from the Visual Studio .NET command prompt using the following command:

```
sn –k LoanBroker.snk
```

In the BizTalk project, change the Active Solution Configuration to Deployment and open the project's properties page. Assign the strong name key file under Common Properties/Assembly/Assembly Key File property. At the same time change the Configuration Properties/Deployment/Redeploy property to 'true' as this allows us to redeploy the orchestration to the GAC.

Finally build the project and deploy in Visual Studio .NET.

Step 5: Run BizTalk Web Services Publishing Wizard

Once deployed to the GAC, the orchestration can then be published as a Web service using the BizTalk Web Services Publishing Wizard. The BizTalk Web Services Publishing Wizard inspects the credit bureau orchestration assembly and generates an ASP.NET Web service and physical web ports.

To keep the project's files together, create a new folder "CreditBureau" under your solution directory. Create a virtual folder from the IIS Admin Console and map it to the new folder. Next, start the wizard, and select the CreditBureau assembly. Check the following options: Support Unknown SOAP Headers, Allow Anonymous Access, and Create BizTalk Receive Locations. Set the Project Location to the new virtual folder.

Step 6: Bind Logical Port to Physical Port

At design time, the actual physical ports for the orchestration are usually not known because they depend on the specific deployment environment. In BizTalk 2004, this problem is easily eliminated as BizTalk 2004 allows late binding for the ports, that is, we can define a logical port in the orchestration and then bind it later to an actual physical port through BizTalk Explorer.

The credit bureau orchestration uses only a single logical port that needs to be bound to the physical Web port generated by the wizard. This task is achieved using the BizTalk Explorer in Visual Studio .NET. In BizTalk Explorer, double click the Credit Bureau orchestration to instigate the configuration dialog. In the configuration dialog, bind the inbound port “Port_CreditBureau” to the generated Web port.

Step 7: Enlist and Start the Orchestration

The final step is to enlist and start the orchestration in BizTalk Explorer.

To verify that the credit bureau Web service is up and running navigate your Web browser to the following URL (replace *virtfoder* with the name of the virtual folder you created in Step 5):
http://localhost/*virtfolder*/CreditBureau_CreditBureauProcess_Port_CreditBureau.asmx

You should be able to see a list of all operations supported by the Web service. In our case the only defined operation is GetCreditScore.

5.2 Designing the Bank

After interacting with the credit bureau, the loan broker needs to interact with multiple banks to obtain loan quotes. Similar to the credit bureau, the banks are self-contained services and generate a random interest rate quote based on the client’s data.

We simulate the three different banks, each of which uses different rules governing the generation of the interest rate quote. Each of the banks is distinguished by different bank names and we use a simple formula for loan quote calculation that is parameterized by the following parameters:

- RatePremium, each bank has a minimum premium rate, which determines their profit margin.
- MaxLoanTerm, the maximum loan term that the bank is willing to engage

In our scenario:

Bank 1 - Consumer Bank, services a broad range of customers. The bank charges an average rate premium at 2.0% and willing to accept a maximum loan term of 48 months.

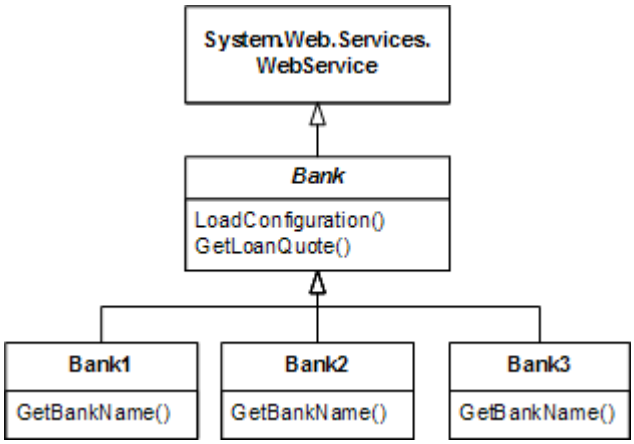
Bank 2 - Exclusive Bank offers the best rates but specialized in top-end clientele. Exclusive bank only charges a 1.8% rate premium and is willing to engage a maximum loan term of 60 months.

Bank 3 - Loan Shark offers the highest rate where they will give a loan to just about anybody but charge a hefty premium rate at 4.0% and a maximum loan term of 72 months.

We design and implement the banks as C# ASP.NET web services. For a detailed description of implementing Web services using ASP.NET please refer to [SOINET].

The banks expose a web method that allows the client to pass a quote request message and retrieve a quote reply message in return. All the banks are very similar, the only differences being the *BankName*, *RatePremium* and *MaxLoanTerm,* which regulate the

interest rate generation process. In order to avoid code duplication, we use an abstract base class to encapsulate the calculation logic for the quote. The class retrieves the required parameters from a shared configuration file.



Class diagram of Bank implementation

Step 1: Define Message Format

To keep all project files together create a folder under the solution directory and then map a new IIS virtual folder to this folder using the IIS Administrative Console. Next, add a new Visual C# ASP.Net Web Service project to the solution and specify the virtual folder as the project location.

For simplicity's sake we use a common message structure for all banks. We define the message formats for bank quote request message and bank quote reply message as structs in the abstract class, Bank. The Bank class inherits from WebService so that it is accessible as an ASP.NET Web service.

Add the following class to the project:

```
public abstract class Bank : System.Web.Services.WebService
{
   public struct BankQuoteRequest
   {
      public int SSN;
      public int CreditScore;
      public int HistoryLength;
      public decimal LoanAmount;
      public int LoanTerm;
   }

   public struct BankQuoteReply
   {
      public double InterestRate;
      public string QuoteID;
      public int ErrorCode;
   }
}
```

For the reply message, besides the interest rate, a quote ID and an error code are included in the message. The quote ID specifies which bank quoted the interest rates, while the error code equals 0 if a valid interest quote is provided or 1 if the bank decided not to provide a quote.

Step 2: Load Configuration Parameters

Each bank has to load its own specific set of parameters from the configuration file. The bank names are different from bank to bank, making the name a good candidate to distinguish the parameters.

Add an abstract method GetBankName() to the base class. Each bank class will implement this method and return the correct bank name. The base class constructor uses the bank name returned by this method to load the correct parameters *BankName*, *RatePremium* and *MaxLoanTerm* from the configuration file.

```
public abstract class Bank : System.Web.Services.WebService
{
  . . .
  // abstract method
  protected abstract string GetBankName();
  public const double PRIME_RATE = 2.0;
  private string bankName;
  private double ratePremium;
  private int maxLoanTerm;

  public Bank()
  {
    string bankInstance = GetBankName();
    bankName = ConfigurationSettings.AppSettings[bankInstance + "_BankName"];
    ratePremium = Double.Parse(
                ConfigurationSettings.AppSettings[bankInstance + "_RatePremium"]);
    maxLoanTerm = Int32.Parse(
                ConfigurationSettings.AppSettings[bankInstance + "_MaxLoanTerm"]);
  }
}
```

Add the bank parameters to the configuration file, Web.config:

```
<appSettings>
  <add key="Bank1_BankName" value="Consumer Bank" />
  <add key="Bank1_RatePremium" value="2.0" />
  <add key="Bank1_MaxLoanTerm" value="48" />
  <add key="Bank2_BankName" value="Exclusive Bank" />
  <add key="Bank2_RatePremium" value="1.8" />
  <add key="Bank2_MaxLoanTerm" value="60" />
  <add key="Bank3_BankName" value="Loan Shark " />
  <add key="Bank3_RatePremium" value="4.0" />
  <add key="Bank3_MaxLoanTerm" value="72" />
</appSettings>
```

Step 3: Bank Quote Calculation

Since now each of the banks is initialized with its own premium rates and maximum loan term that it can handle, we can add a generic quote calculation method into the abstract class:

```
[WebMethod]
[SoapDocumentMethod(ParameterStyle = SoapParameterStyle.Bare)]
public BankQuoteReply GetLoanQuote(
    [XmlElementAttribute(ElementName="GetBankQuoteRequest")]
    BankQuoteRequest request)
{
  BankQuoteReply reply = new BankQuoteReply();
  Random random = new Random();
  if (request.LoanTerm <= maxLoanTerm)
  {
    reply.InterestRate = PRIME_RATE + ratePremium +
        (double)(request.LoanTerm/12)/10 + (double)random.Next(10)/10;
    reply.ErrorCode = 0;
  }
  else
  {
    reply.InterestRate = 0.0;
    reply.ErrorCode = 1;
  }
  reply.QuoteID = String.Format("{0}-{1:00000}", bankName,    random.Next(100000));
  return reply;
}
```

Note that the loan quote calculations shown are just a trivial example as in real life the calculations are far more complex.

Step 4: Adding the Individual Banks

Finally, we add the individual bank classes, which override and inherit from the abstract *Bank* class. Each concrete bank class implements only a single method, which overrides the abstract method *GetBankName*.

Add a new Web service to the project. Change the generated class to inherit from the Bank class and add the following attribute to the class to set the namespace:

```
[WebService(Namespace="http://www.microsoft.com/biztalk/bank")]
```

The only method the class has to implement is GetBankName:

```
protected override string GetBankName()
{ return "Bank1"; }
```

The code below shows the code for Bank1 (Bank2 and Bank3 follow the same principle):

```
[WebService(Namespace="http://www.microsoft.com/biztalk/bank")]
public class Bank1 : Bank
{
    // generated code
    ...
    protected override string GetBankName()
    { return "Bank1"; }
}
```

In the end, compile and build the project. Ensure the Bank Web service is up and running by navigating with the Web browser to the following URLs (again *virtfolder* is the name of the virtual folder created in Step 1):

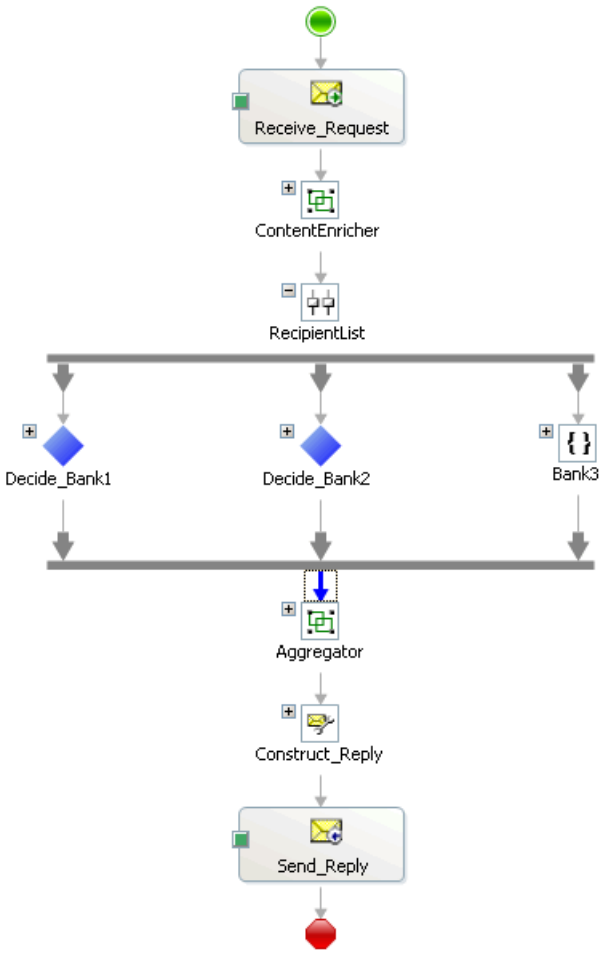
http://localhost/*virtfolder*/Bank1.asmx

http://localhost/*virtfolder*/Bank2.asmx

http://localhost/*virtfolder*/Bank3.asmx

5.3 Designing the Loan Broker

As described in the section Designing with Patterns, the loan broker functionality can be expressed as the collaboration of the following five patterns: Service Interface, Content Enricher, Recipient List, Aggregator, and Message Translator. The section Implementation Strategies discussed in general terms how each pattern can be implemented with BizTalk Server 2004. The following section applies the loan broker-specific context to the patterns and steps through the specific implementation of each component. The figure below depicts a high-level view of the loan broker orchestration.



Loan Broker Process Overview

Step 1: Define message schemas

In order to receive requests from customers the loan broker has to implement the Service Interface pattern. The required steps for this portion closely resemble the credit bureau implementation as both services implement the same pattern.

Add a new Empty BizTalk Server Project to the solution and name it 'LoanBroker'. Add two new schemas to the new project:

LoanQuoteRequest.xsd

Namespace: http://www.microsoft.com/biztalk/loanbroker

Name	Element	Type
LoanQuoteRequest	Root Node	-
SSN	Child Field Element	xs:integer
LoanAmount	Child Field Element	xs:decimal
LoanTerm	Child Field Element	xs:int

LoanQuoteReply.xsd

Namespace: http://www.microsoft.com/biztalk/loanbroker

Name	Element	Type
LoanQuoteReply	Root Node	-
SSN	Child Field Element	xs:integer
LoanAmount	Child Field Element	xs:decimal
InterestRate	Child Field Element	xs:double
QuoteID	Child Field Element	xs:string

Step 2: Define logical request-response ports

Add a new Orchestration to the project in Visual Studio .NET, name it LoanBrokerProcess, and set the Transaction Type to 'Long Running'.

In the BizTalk Orchestration Designer, add a new configured port to the orchestration. Configure the port using the Port Configuration Wizard as a public request-response port according to the configuration values defined in the following table:

Port Configuration

Properties	Value
Name	Port_LoanBroker
Existing or New Port Type	New
Port Type	PortType_LoanBroker
Communication Pattern	Request-Response
Access Restrictions	Public – no limit
Port direction of Communication	I'll be receiving a request and sending a response
Port Binding	Specify Later – the physical port will be created later by BizTalk Web Services Publishing Wizard

Then rename the Operation of the port from Operation_1 to GetLoanQuote. Similar to the credit bureau implementation, as the orchestration will be exposed as a Web service, we need to ensure the access restriction on the port is set to public access.

Step 3: Define message variables for Loan Broker schemas

Subsequently, we define the message variables to hold messages of the schema types defined in Step 1.

To add message variables, go to Orchestration View in Visual Studio .NET, right click on Message node and select “New Message”. At this moment, define the following message variables. They are:

Message Variables

Identifier	Message Type
LoanQuoteRequestMsg	Schemas – LoanQuoteRequest
LoanQuoteReplyMsg	Schemas – LoanQuoteReply

Step 4: Receiving Requests

A request message comes in through the request port and goes into the Receive shape to initiate the BizTalk orchestration process. Each incoming message represents a loan quote request and needs to instantiate a new orchestration instance. Therefore, we set the Activate property of the Receive shape to true.

In the Orchestration Designer, add a Receive Shape from the toolbox to the Designer surface. Edit the properties of the Receive Shape as follows:

Receive shape

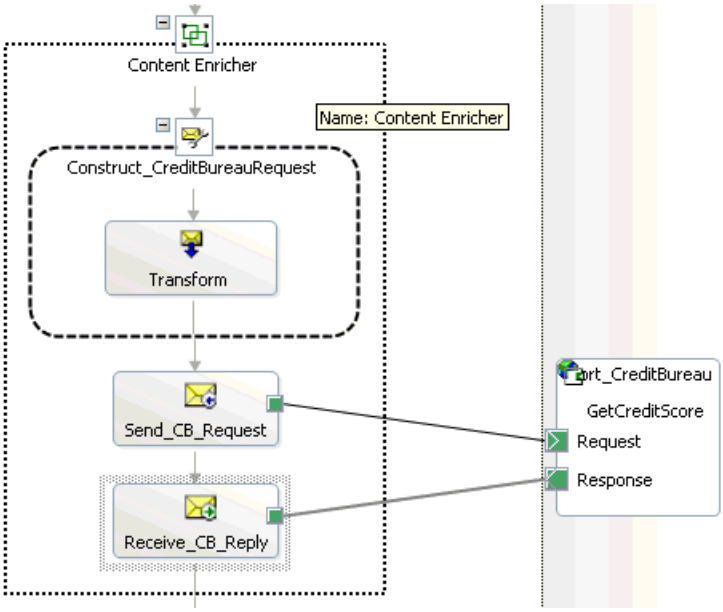
Properties	Value
Name	Receive_Request
Activate	True
Message	LoanQuoteRequestMsg
Operation	Connect the Receive shape to Port_LoanBroker Request operation

So far, our orchestration can receive loan quote request messages and stores the request message in the LoanQuoteRequestMsg variable. We can now use the data in this variable to contact the credit bureau and the banks.

Step 5: Interaction with Credit Bureau

The information in the original loan broker request message does not contain enough information to generate a request message for a loan quote to the bank. The fields in the loan broker request message such as SSN, LoanAmount and LoanTerm only form part of the Bank request message because the banks also require the client’s credit score and history information.

To augment the message in the orchestration, the loan broker needs to interact with the credit bureau to obtain the additional information. This is where the *Content Enricher* pattern plays a big role.



Interaction with the Credit Bureau using a Content Enricher

Complete the following steps to implement the Content Enricher pattern with the Credit Bureau:

a. Add a Group shape in the Orchestration Designer after the Receive shape and name it 'Content Enricher'. The purpose of the Group shape is to visually group the credit bureau interaction steps together.

b. In the Solution Explorer, add a Web reference to the credit bureau's Web service. Name the reference 'CreditBureau'.

c. Define message variables for the Credit Bureau Web Message request and response types.

Message Variables

Message Variable	Message Type
CreditBureauRequestMsg	Web Message Types – LoanBroker.CreditBureau.CreditBureau_CreditBureauProcess_Port_CreditBureau_.GetCreditScore_request
CreditBureauReplyMsg	Web Message Types – LoanBroker.CreditBureau.CreditBureau_CreditBureauProcess_Port_CreditBureau_.GetCreditScore_response

d. Next, construct the Credit Bureau Request message based on the information provided in the Loan Broker request message. From the toolbox, add a Construct Message shape within the Content Enricher Group.

Construct Message shape

Properties	Value
Name	Construct_CreditBureauRequest
Message Constructed	CreditBureauRequestMsg

Add a Transform shape into the Construct Message shape and rename it to 'Transform'. After renaming the Transform shape, double click on it to initiate the Transform Configuration Wizard. Step through the wizard using the following configuration values:

Transform shape

Properties	Value
New or Existing Map	New
Fully Qualified Map Name	LoanBroker.LoanQuoteRequestToCreditBureauRequest
Source	LoanQuoteRequestMsg
Destination	CreditBureauRequestMsg.CreditBureauRequest

Then Click OK to launch the BizTalk Mapper and edit the map as follows:

CreditBureauRequestMsg.CreditBureauRequest Map

Target	Functoids	Source
SSN	-	SSN

e. Add a new Configured Port to the port surface as point of access to the Credit Bureau. In the Port Configuration Wizard, configure the port with the following settings:

Port Configuration

Properties	Value
Name	Port_CreditBureau
Existing or New Port Type	Existing (from the Credit Bureau Web reference)
Port Type	Web Port Types - LoanBroker.CreditBureau.CreditBureau_CreditBureauProcess_Port_CreditBureau_.CreditBureau_CreditBureauProcess_Port_CreditBureau
Port Binding	Specify Now (The remaining port properties are configured automatically based on the Web reference)

f. Add a Send shape below the Construct Message shape to send a request off synchronously to the Credit Bureau Port. This is followed by a Receive shape to retrieve the response from the Web service.

Send shape

Properties	Value
Name	Send_CB_Request
Message	CreditBureauRequestMsg
Operation	Connect the Send shape to Port_CreditBureau Request operation

Receive shape

Properties	Value
Name	Receive_CB_Reply

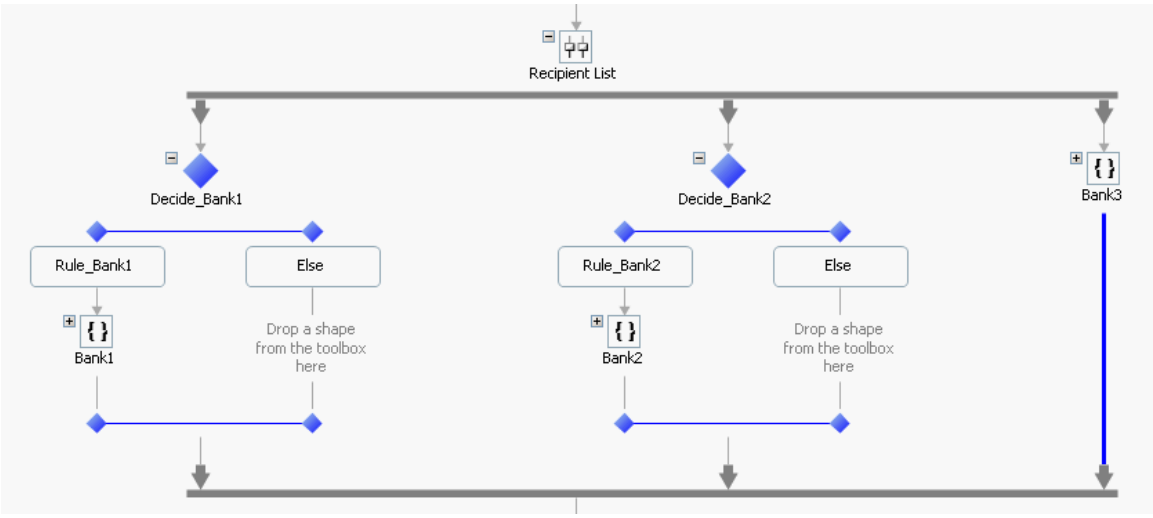
Activate	False
Message	CreditBureauReplyMsg
Operation	Connect the Receive shape to Port_CreditBureau Response operation

No correlation set is needed because the interaction is via synchronous request-reply.

Note that the generic Content Enricher pattern includes a Transform shape to merge the original request message with the response from the external resources (the credit bureau in this case). We do not include this step here because the recipient list in the next step uses an array of transformers to create the bank request messages.

Step 6: Interaction with the Banks

After retrieving the required information from the credit bureau, the loan broker has sufficient information to interact with the banks. Next, we implement the Recipient List pattern to route a bank request message to the appropriate banks. As each of the banks has a minimum requirement on the credit score, the Recipient List filters out the messages that do not meet the bank's credit score requirement. Once filtered, the loan broker constructs the bank request messages and passes them to the qualifying banks to obtain a loan quote.



Recipient List Implementation

The Recipient List implementation uses a combination of a Parallel Actions shape and a set of Decide shapes. Complete the following steps to implement the Recipient List:

a. Add a Parallel Actions shape after the Content Enricher and rename it to 'Recipient List'.

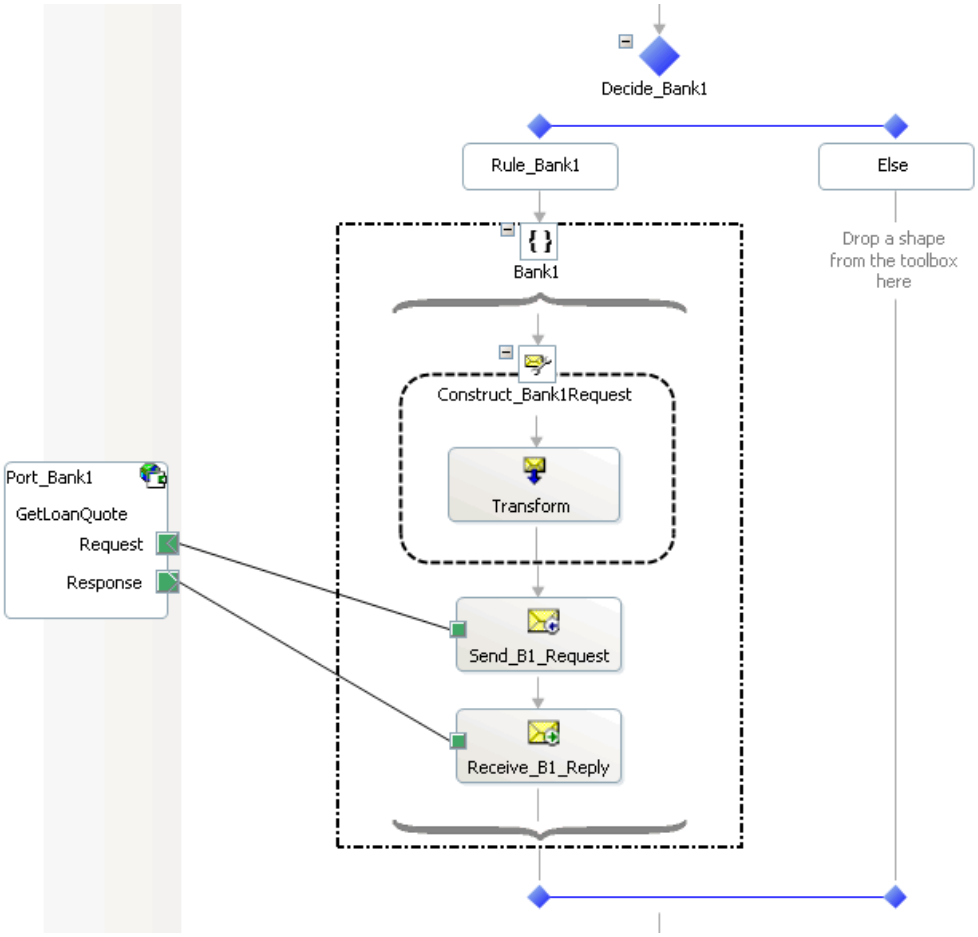
b. In the Solution Explorer, add a Web reference to the three banks' Web services. Name the references as Bank1, Bank2, and Bank3.

c. Promote all fields in the CreditBureau's reply message schema to Distinguished Fields. Promoting schema fields to Distinguished Fields allows the fields to be referenced from an XLANG expression. The Decide shape uses an XLANG expression to evaluate qualifying banks based on the credit score and history length values.

   In the Solution Explorer, expand the Web Reference node and traverse down to Credit Bureau's Reference.map. Open the Reference.xsd file and edit it with BizTalk

Schema Editor. In BizTalk Schema Editor, promote the CreditScore and HistoryLength fields of the CreditBureauReply schema as Distinguished Fields.

BANK1 PARALLEL BRANCH



Bank1 Interaction

d. Begin building the Bank1 parallel branch by adding a Decide shape to the first branch of the Parallel Actions shape. Rename it to 'Decide_Bank1', then select Rule_1 and rename it to 'Rule_Bank1'. Under the Rule_Bank1 Expression property, enter the following XLANG expression:

```
CreditBureauReplyMsg.GetCreditScoreResult.CreditScore >= 500 &&
CreditBureauReplyMsg.GetCreditScoreResult.HistoryLength >= 5
```

Note that the 'GetCreditScoreResult' from the expression is actually a part of the multi-parts message of the Web service message reply.

e. Add a Scope shape underneath the Rule_Bank1 shape. Rename the Scope to 'Bank1' and set the Transaction Type to 'Long Running'.

f. In the Orchestration View, expand the Bank1 node and select the Messages node. Add a two new message variables so that they are local under the Bank1 scope:

Message Variable	Message Type
Bank1RequestMsg	Web Message Types – LoanBroker.Bank1.Bank1_.GetLoanQuote _request
Bank1ReplyMsg	Web Message Types – LoanBroker.Bank1.Bank1_.GetLoanQuote _response

g. Next, construct the Bank1 request message using the Loan Broker request message and the Credit Bureau reply message. Insert a Construct Message shape into the Bank1 scope.

Construct Message shape

Properties	Value
Name	Construct_Bank1Request
Message Constructed	Bank1RequestMsg

Add a Transform shape into the Construct Message shape and rename it to 'Transform'. Double click on the shape to initiate the Transform Configuration Wizard. Note that there are two ways to initiate the creation of a BizTalk Map in Visual Studio .NET. The first way is by directly adding a BizTalk Map to the project and editing it using BizTalk Mapper, while the alternative is indirectly through the Transform Configuration Wizard. To map multiple source schemas to a target schema you have to use the latter method. Step through the wizard using the following configuration values:

Transform shape

Properties	Value
New or Existing Map	New
Fully Qualified Map Name	LoanBroker.LoanQuoteRequestToBank1Request
Source	LoanQuoteRequestMsg **and** CreditBureauReplyMsg.GetCreditScoreResult
Destination	Bank1RequestMsg.GetBankQuoteRequest

Then launch the BizTalk Mapper and edit the map as follows:

LoanBroker.LoanQuoteRequestToBank1Request Map

Target	Functoids	Source
SSN	-	LoanQuoteRequest.SSN
CreditScore		CreditBureauReply.CreditScore
HistoryLength		CreditBureauReply.HistoryLength
LoanAmount		LoanQuoteRequest.LoanAmount
LoanTerm		LoanQuoteRequest.LoanTerm

h. Add a new Configured Port for Bank1 to the Port Surface and configure it as follows.

Port Configuration

Properties	Value
Name	Port_Bank1
Existing or New Port Type	Existing (from the Bank1 Web reference)
Port Type	Web Port Types - LoanBroker.Bank1.Bank1_.Bank1

Port Binding	Specify Now (The remaining port properties are configured automatically based on the Web reference)

i. After the Construct Message shape, add a Send shape followed by a Receive shape.

Send shape

Properties	Value
Name	Send_B1_Request
Message	Bank1RequestMsg
Operation	Connect the Send shape to Port_Bank1 Request operation

Receive shape

Properties	Value
Name	Receive_B1_Reply
Activate	False
Message	Bank1ReplyMsg
Operation	Connect the Receive shape to Port_Bank1 Response operation

BANK2 PARALLEL BRANCH

j. The Bank2 branch is essentially identical to the Bank1 branch except for the rules inside the Decide shape and the external port. Add a Decide shape in the second Parallel Branch and rename it to ‘Decide_Bank2’. Select the Rule_1 shape and rename it to Rule_Bank2. Enter the following XLANG expression for the Rule_Bank2 shape:

```
CreditBureauReplyMsg.GetCreditScoreResult.CreditScore >= 700 &&
CreditBureauReplyMsg.GetCreditScoreResult.HistoryLength >= 10
```

k. Add a Scope shape underneath the Rule_Bank2 shape. Rename the Scope to 'Bank2' and set the Transaction Type to 'Long Running'.

l. In the Orchestration View, expand the Bank2 node and select the Messages node. Add a two new message variables so that they are local under the Bank2 scope:

Message Variable	Message Type
Bank2RequestMsg	Web Message Types – LoanBroker.Bank2.Bank2_.GetLoanQuote _request
Bank2ReplyMsg	Web Message Types – LoanBroker.Bank2.Bank2_.GetLoanQuote _response

m. Construct the Bank2 request message using the Loan Broker request message and the Credit Bureau reply message. From the toolbox, add a Construct Message shape after the Rule_Bank2.

Construct Message shape

Properties	Value
Name	Construct_Bank2Request
Message Constructed	Bank2RequestMsg

Add a Transform shape into the Construct Message shape and rename it to ‘Transform’. After renaming the Transform shape, double click on it to initiate the

Transform Configuration Wizard and step through the wizard using the following configuration values:

Transform shape

Properties	Value
New or Existing Map	New
Fully Qualified Map Name	LoanBroker.LoanQuoteRequestToBank2Request
Source	LoanQuoteRequestMsg and CreditBureauReplyMsg.GetCreditScoreResult
Destination	Bank2RequestMsg.GetBankQuoteRequest

Then launch the BizTalk Mapper and edit the map as follows:

LoanBroker.LoanQuoteRequestToBank2Request Map

Target	Functoids	Source
SSN	-	LoanQuoteRequest.SSN
CreditScore		CreditBureauReply.CreditScore
HistoryLength		CreditBureauReply.HistoryLength
LoanAmount		LoanQuoteRequest.LoanAmount
LoanTerm		LoanQuoteRequest.LoanTerm

Note that the map to create Bank2 request message is similar to Bank1 map because all bank implementations share a common schema for the bank messages. However, in real life each bank will likely have its own proprietary schema.

n. Add a new Configured Port to the Port Surface for Bank2

Port Configuration

Properties	Value
Name	Port_Bank2
Existing or New Port Type	Existing (from the Bank2 Web reference)
Port Type	Web Port Types - LoanBroker.Bank2.Bank2_.Bank2
Port Binding	Specify Now (The remaining port properties are configured automatically based on the Web reference)

o. After the Construct Message shape, add a Send shape follows by a Receive shape.

Send shape

Properties	Value
Name	Send_B2_Request
Message	Bank2RequestMsg
Operation	Connect the Send shape to Port_Bank2 Request operation

Receive shape

Properties	Value
Name	Receive_B2_Reply
Activate	False
Message	Bank2ReplyMsg
Operation	Connect the Receive shape to Port_Bank2 Response operation

BANK3 PARALLEL BRANCH

p. The Bank3 Parallel Branch does not need a Decide shape because Bank3 accepts any loan quote request regardless the credit score of the applicants. Add a new branch to the Parallel Actions shape. Then, add a Scope shape into the new branch. Name the Scope 'Bank3' and set its Transaction Type to 'Long Running'.

q. In the Orchestration View, expand the Bank3 node and select the Messages node. Add a two new message variables so that they are local under the Bank3 scope:

Message Variable	Message Type
Bank3RequestMsg	Web Message Types – LoanBroker.Bank3.Bank3_.GetLoanQuote _request
Bank3ReplyMsg	Web Message Types – LoanBroker.Bank3.Bank3_.GetLoanQuote _response

r. Construct the request message for Bank3 by adding a new Construct Message shape into the Scope Bank3.

Construct Message shape

Properties	Value
Name	Construct_Bank3Request
Message Constructed	Bank3RequestMsg

Next, add a Transform shape into the Construct Message shape and rename it to ‘Transform’. After renaming the Transform shape, double click on it to initiate the Transform Configuration Wizard and configure the shape using the following values:

Transform shape

Properties	Value
New or Existing Map	New
Fully Qualified Map Name	LoanBroker.LoanQuoteRequestToBank3Request
Source	LoanQuoteRequestMsg and CreditBureauReplyMsg.GetCreditScoreResult
Destination	Bank3RequestMsg.GetBankQuoteRequest

Then launch the BizTalk Mapper and configure the map as follows:

LoanBroker.LoanQuoteRequestToBank3Request Map

Target	Functoids	Source
SSN	-	LoanQuoteRequest.SSN

CreditScore	-	CreditBureauReply.CreditScore
HistoryLength	-	CreditBureauReply.HistoryLength
LoanAmount	-	LoanQuoteRequest.LoanAmount
LoanTerm	-	LoanQuoteRequest.LoanTerm

s. Add a new Configured Port to the Port Surface for Bank3

Port Configuration

Properties	Value
Name	Port_Bank3
Existing or New Port Type	Existing (from the Bank3 Web reference)
Port Type	Web Port Types - LoanBroker.Bank3.Bank3_.Bank3
Port Binding	Specify Now (The remaining port properties are configured automatically based on the Web reference)

t. After the Construct Message shape, add a Send shape follows by a Receive shape.

Send shape

Properties	Value
Name	Send_B3_Request
Message	Bank3RequestMsg
Operation	Connect the Send shape to Port_Bank3 Request operation

Receive shape

Properties	Value
Name	Receive_B3_Reply
Activate	False
Message	Bank3ReplyMsg
Operation	Connect the Receive shape to Port_Bank3 Response operation

Step 7: Building the Aggregator

So far we implemented the Content Enricher and the Recipient List to pass the loan quote request to the appropriate banks. Now all the banks' replies have to be consolidated into a single response message. The *Aggregator* pattern is the best candidate for the task.

Reviewing the decisions that we made in the Solution Architecture section, the implementation of the Aggregator is driven by the following design decisions:

- Correlation: no correlation is needed due to synchronous interaction. Also each orchestration instance will have its own instance of the aggregator class (see below) so that no correlation across instances is required.
- Completeness Condition: Aggregator state is considered complete once a response is received from all banks

- Aggregation Algorithm: Determine the lowest interest rate

To implement the Aggregator pattern we create a C# class that is referenced by Expression shapes, inserted after each of the Bank's Receive shapes. Each Expression shape adds a message to the message aggregate.

To implement this function complete the following steps:

Define a new schema and name it BestBankQuote with the specified elements:

BestBankQuote.xsd

Namespace: http://www.microsoft.com/biztalk/loanbroker

Name	Element	Type
BestBankQuote	Root Node	-
InterestRate	Child Field Element	xs:double
QuoteID	Child Field Element	xs:string
ErrorCode	Child Field Element	xs:int

Then add a new message variable to represent the schema:

Message Variables

Message Variable	Message Type
BestBankQuoteMsg	Schemas – BestBankQuote

As the name indicates, the BestBankQuote schema represents the best bank quote message from the message aggregator.

Add a new Visual C# Class Library project to the solution and name it BankQuoteAggregator. Begin the implementation of the C# message aggregator class by creating a new C# interface:

```
namespace BankQuoteAggregator
{
    public interface Aggregator
    {
        void AddMessage(XmlDocument document);
        XmlDocument GetResultMessage();
    }
}
```

In C#, BizTalk Message schemas are equivalent to XmlDocument types. The method AddMessage adds a received message into the aggregate, while GetResultMessage() returns a result message based on the aggregator's algorithm. In this case, the method returns the message with the lowest interest rate.

Next we create a data holder class to mimics the schema of the message. Creating this class allows us to separate the aggregator's logic from the XML parsing logic.

```
[Serializable]
internal class DataHolder
{
    private readonly double interestRate;
    private readonly string quoteID;
    private readonly int errorCode;

    public const int STATUS_OK = 0;
    public const int STATUS_ERROR = 1;

    public DataHolder(double interestRate, string quoteID, int errorCode)
    {
        this.interestRate = interestRate;
        this.quoteID = quoteID;
        this.errorCode = errorCode;
    }

    public double InterestRate { get { return interestRate; }}
 public string QuoteID { get { return quoteID; }}
    public double ErrorCode { get { return errorCode; }}
}
```

Note that the DataHolder class has to be serializable so that it can be used inside a long-running BizTalk transaction. Long running transactions need to be able to "dehydrate" (i.e. serialize) the state of an orchestration, including the state of any referenced C# class.

Next, create a helper class that translates XML messages into the instances of the DataHolder class and vice versa. This class acts as an *Assembler* [EAA]. If you want to avoid the XML parsing code altogether you can also use the xsd tool to create C# classes directly from the XML schema and have the XMLSerializer take care of parsing [SOINET].

```
internal class DataHolderAssembler
{
  public static DataHolder GetInstance(XmlDocument document)
  {
      DataHolder dataHolder;
      try
      {
         XmlElement root = document.DocumentElement;
         XmlNode node = root.SelectSingleNode("/*[local-
name()='BankQuoteReply']/*[local-name()='InterestRate']");
         string text = node.InnerText;
         double interestRateVal = Double.Parse(text);
         string quoteIDVal = root.SelectSingleNode("/*[local-
name()='BankQuoteReply']/*[local-name()='QuoteID']").InnerText;
         int errorCodeVal = Int32.Parse(root.SelectSingleNode("/*[local-
name()='BankQuoteReply']/*[local-name()='ErrorCode']").InnerText);
         dataHolder = new DataHolder(interestRateVal, quoteIDVal,
                                     errorCodeVal);
       }
       catch(Exception) { dataHolder = null; }
       return dataHolder;
  }
```

```
  public static XmlDocument AssembleResult(DataHolder quote)
  {
      XmlDocument document = new XmlDocument();
      XmlNode root = document.CreateNode(XmlNodeType.Element, "BestBankQuote",
          "http://www.microsoft.com/biztalk/aggregator");
      XmlElement interestRate = document.CreateElement("InterestRate");
      interestRate.InnerText = quote.InterestRate.ToString();
      XmlElement quoteID= document.CreateElement("QuoteID");
      quoteID.InnerText = quote.QuoteID;
      XmlElement errorCode = document.CreateElement("ErrorCode");
      errorCode.InnerText = quote.ErrorCode.ToString();
      root.AppendChild(interestRate);
      root.AppendChild(quoteID);
      root.AppendChild(errorCode);
      document.AppendChild(root);
      return document;
  }
}
```

Incoming XML documents have to conform to the bank quote reply message schema while the AssembleResult method creates an XML document instance that conforms to the BestBankQuote schema.

Finally, the BankQuoteAggregator class provides an implementation of the generic Aggregator interface:

```
[Serializable]
public class BankQuoteAggregator : Aggregator
{
  private DataHolder bestQuote;

  public void AddMessage(XmlDocument document)
  {
    DataHolder currentQuote = DataHolderAssembler.GetInstance(document);
    if (currentQuote != null && currentQuote.ErrorCode ==
        DataHolder.STATUS_OK)
    {
        if (bestQuote == null || bestQuote.InterestRate >
            currentQuote.InterestRate)
        {
            bestQuote = currentQuote;
        }
    }
  }

  public XmlDocument GetResultMessage()
  {
    if (bestQuote == null)
    {
        bestQuote = new DataHolder(0.0, "No Qualifying Bank Quotes", 1);
    }
    return DataHolderAssembler.AssembleResult(bestQuote);
  }
}
```

In this class, the extra effort of creating a data holder and an Assembler pays off. We can implement the Aggregator logic in a few lines of code that have next to no references to XML and messaging. When a new quote message is added, the BankQuoteAggregator class determines whether the rate quoted is lower than the best quote so far. If the new quote is lower, it is then assigned to be the lowest quote. If the bank sets the error flag, the message is simply ignored.

In order to reference the BankQuoteAggregator classes in the orchestration, the BankQuoteAggregator classes have to be built using a strong key and deployed to the GAC.

a. Open the AssemblyInfo.cs file and assign a fixed version number to the assembly.

```
[assembly: AssemblyVersion("1.0.0")]
```

b. Assign a strong key in AssemblyInfo.cs

```
[assembly: AssemblyKeyFile("LoanBroker.snk")]
```

c. Build and deploy to the GAC using the gacutil.exe tool

```
gacutil -I BankQuoteAggregator.dll
```

Switch back to the Orchestration Designer. In the LoanBroker project add a reference to the BankQuoteAggregator project. Next create a new global variable in the Orchestration View under the Variables node:

Variables

Property	Value
Name	QuoteAggregator
Type	<.NET Class> and then select the BankQuoteAggregator.BankQuoteAggregator class
Use Default Constructor	True

Now we can add the banks' reply messages to the QuoteAggregator using the AddMessage method.

a. Insert a new Scope shape after the Receive shape inside the Bank1 Scope. Name it 'Atomic_Scope_1' and set the transaction type to 'Atomic'. An atomic transaction is needed in this scenario because the QuoteAggregator variable is accessed by the multiple parallel activities. If the access to the QuoteAggregator is not encapsulated within an atomic transaction scope, we are bound to run into a shared data update problem (actually, the orchestration compiler will catch this problem and flag a build error).

b. Add an Expression shape inside the new scope and name it Aggregate_Bank1. In the Expression Editor insert the following expression:

```
QuoteAggregator.AddMessage(Bank1ReplyMsg.GetLoanQuoteResult);
```

c. Repeat for Bank2 and Bank3, with the following values:

   Bank2 Parallel Branch

   - Scope shape as 'Atomic_Scope_2' with atomic transaction type
   - Expression shape as 'Aggregate_Bank2' with the following expression:

```
QuoteAggregator.AddMessage(Bank2ReplyMsg.GetLoanQuoteResult);
```

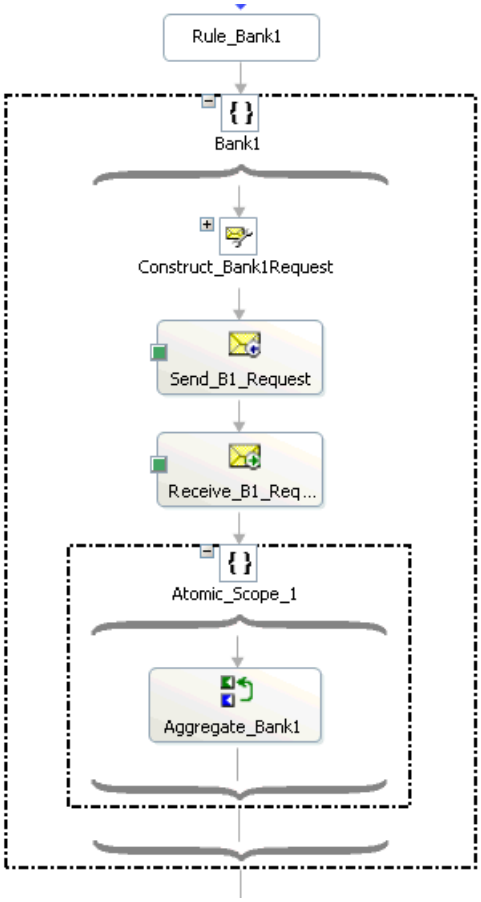
   Bank3 Parallel Branch

   - Scope shape as 'Atomic_Scope_3' with atomic transaction type

- Expression shape as ‘Aggregate_Bank3’ with the following expression:

```
QuoteAggregator.AddMessage(Bank3ReplyMsg.GetLoanQuoteResult);
```

The Bank1 Parallel Branch now looks as follows:



Bank1 Branch with Aggregator

Next we retrieve the best quote from the QuoteAggregator, once all the messages received are consolidated.

d. Add a new Group shape after the Parallel Actions shape and name it ‘Aggregator’

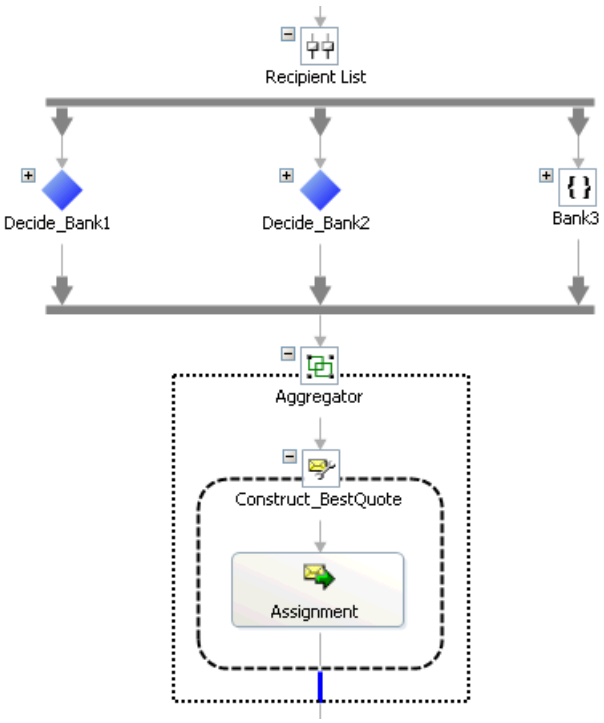
e. Add a new Construct Message shape into the ‘Aggregator’ Group and set with the specified values:

Construct Message shape

Properties	Value
Name	Construct_BestQuote
Message Constructed	BestBankQuoteMsg

f. Next insert a Message Assignment shape within the Construct Message shape and rename it to Assignment. Insert the following XLANG expression into the assignment:

```
BestBankQuoteMsg = QuoteAggregator.GetResultMessage();
```



Implementation of Aggregator

Step 8: Construct the Loan Broker Reply Message

The last step remaining in the Loan Broker orchestration is to implement the Message Translator pattern to construct a loan quote reply message based on the LoanQuoteReplyMsg schema. Two data sources are needed to construct the reply message. The SSN and the loan amount are copied from the original request message while the interest rate and the quote ID are copied from the BestBankQuoteMsg variable.

Add a new Construct Message shape after the Aggregator Group and configure it with the following settings:

Construct Message shape

Properties	Value
Name	Construct_Reply
Message Constructed	LoanQuoteReplyMsg

Add a Transform shape into the Construct Message shape and rename it to 'Transform'. Configure the Transform shape using the following values:

Transform shape

Properties	Value
New or Existing Map	New
Fully Qualified Map Name	LoanBroker.BestBankQuoteToLoanQuoteReply
Source	BestBankQuoteMsg, LoanQuoteRequestMsg
Destination	LoanQuoteReplyMsg

Then Click OK to launch the BizTalk Mapper and edit the map as follows:

LoanBroker.BestBankQuoteToLoanQuoteReply Map

Target	Functoids	Source
SSN	-	LoanQuoteRequest.SSN
LoanAmount	-	LoanQuoteRequest.LoanAmount
InterestRate	-	BestBankQuote.InterestRate
QuoteID	-	BestBankQuote.QuoteID

The last step in the orchestration is to add a Send shape right after the Construct Message shape to send the constructed reply message back to the client. In the Orchestration Designer surface, add a Send shape and configure it as follows:

Send shape

Properties	Value
Name	Send_Reply
Message	LoanQuoteReplyMsg
Operation	Connect the Send shape to Port_LoanBroker Response operation

Step 9: Build and Deploy

Finally, build the Loan Broker orchestration with a strong name key file and then deploy the orchestration to the GAC. Create a new virtual folder from the IIS Administration Console. Publish the orchestration as a Web service using the BizTalk Web Service Publishing Wizard specifying the virtual folder as the project location. Subsequently bind the LoanBroker orchestration's logical port to the web port created by the wizard. Enlist and start the orchestration.

5.4 Putting It All Together

The previous sections describe how to build and deploy the credit bureau, the banks and the loan broker component. To test the complete system we create a simple test client in C#.

a. Add a new Visual C# Console Application project to the solution and name it TestClient
b. Add a Web reference to the loan broker Web service to the project. Name the reference 'LoanBroker'
c. Add a new class 'TestClient' and add the following code:

```
using LoanBroker;
class TestClient
{
  public static void PrintReply(LoanQuoteReply reply)
  {
    Console.WriteLine("SSN : " + reply.SSN);
    Console.WriteLine("INTEREST RATE : " + reply.InterestRate);
    Console.WriteLine("LOAN AMOUNT : " + reply.LoanAmount);
    Console.WriteLine("QUOTE ID : " + reply.QuoteID);
  }

  [STAThread]
  static void Main(string[] args)
  {
    LoanQuoteRequest request = new LoanQuoteRequest();
    request.SSN = "1000";
    request.LoanAmount = new decimal(50000.0);
    if (args.Length > 0)
      request.LoanAmount = decimal.Parse(args[0]);
    request.LoanTerm = 48;
    if (args.Length > 1)
      request.LoanTerm = int.Parse(args[1]);

    LoanBroker_LoanBrokerProcess_Port_LoanBroker service =
        new  LoanBroker_LoanBrokerProcess_Port_LoanBroker();
    LoanQuoteReply reply = service.GetLoanQuote(request);
    TestClient.PrintReply(reply);
    Console.ReadLine();
  }
}
```

d. Execute the test client:

```
LoanBrokerTest 50000 55
```

You will see a response similar to the following:

```
SSN : 1000
INTEREST RATE : 6.1
LOAN AMOUNT : 50000
QUOTE ID : Loan Shark-04652
```

6 Conclusions

This whitepaper demonstrated how a patterns-based approach can decompose an integration scenario into a collection of reusable patterns. The patterns allowed us to discuss design alternatives in a technology-neutral fashion. As a final step, the patterns can be realized in the desired target platform technology.

The simple loan broker example only introduced a handful of patterns. Many more integration patterns can be found in [EIP] and [PAG]. In order to fit the discussion within the constraints of a white paper we had to make some simplifying assumptions. For example, the loan broker does not implement any error handling strategies. For example, if one bank is unavailable or a request times out we might simply use the best quote provided by the other banks.

One might also consider replacing the synchronous interaction between the components with an asynchronous request-reply message-exchange pattern. In this case, the service consumer makes a service request but does not actively wait for the service to respond. Instead, when the service has computed the response the service calls the consumer with the results. This style of interaction is generally better suited for long-running services. We might even consider inserting artificial delays into some of the services to assess the impact of a slow-running service on overall system behavior.

Even when using the synchronous request-reply interaction style we could use the asynchronous versions of the client proxy to make many concurrent requests to the loan broker. This would allow us to examine the system behavior under load.

7 Patterns Reference

This section lists all patterns referenced in the paper.

Service Interface [ESP]

Problem	Solution
How do you make pieces of your application's functionality available to other applications, while ensuring that the interface mechanics are decoupled from the application logic?	Design your application as a collection of software services, each with a Service Interface through which consumers of the application may interact with the service.

Content Enricher

Problem	Solution
How do you communicate with another system if the message originator does not have all the required data items available?	Use a specialized transformer, a Content Enricher, to access an external data source in order to augment a message with missing information.

Recipient List

Problem	Solution
How do you route a message to a list of dynamically specified recipients?	Define a channel for each recipient. Then use a Recipient List to inspect an incoming message, determine the list of desired recipients, and forward the message to all channels associated with the recipients in the list.

Publish-Subscribe Channel

Problem	Solution
How can the sender broadcast an event to all interested receivers?	Send the event on a Publish-Subscribe Channel, which delivers a copy of a particular event to each receiver.

Message Filter

Problem	Solution
How can a component avoid receiving uninteresting messages?	Use a special kind of Message Router, a Message Filter, to eliminate undesired messages from a channel based on a set of criteria.

Aggregator

Problem	Solution
How do you combine the results of individual, but related messages so that they can be processed as a whole?	Use a stateful filter, an Aggregator, to collect and store individual messages until a complete set of related messages has been received. Then, the Aggregator publishes a single message distilled from the individual messages.

Scatter-Gather

Problem	Solution
Problem: How do you maintain the overall message flow when a message needs to be sent to multiple recipients, each of which may send a reply?	Use a Scatter-Gather that broadcasts a message to multiple recipients and re-aggregates the responses back into a single message.

Message Translator

Problem	Solution
How can systems using different data formats communicate with each other using messaging?	Use a special filter, a Message Translator, between other filters or applications to translate one data format into another.

8 About the Authors

Gregor Hohpe

Gregor leads the Enterprise Integration practice at ThoughtWorks, Inc., a specialized provider of application development and integration services. Gregor is a widely recognized thought leader on asynchronous messaging architectures and co-author of the seminal book "Enterprise Integration Patterns" (Addison-Wesley, 2004). Gregor speaks regularly at technical conferences around the world and maintains the Web site www.eaipatterns.com.

Hsue-Shen Tham

Shen is a senior consultant with ThoughtWorks, based in Melbourne, Australia. He has architected and developed large custom applications using both J2EE, and .NET based technologies and has successfully deployed integration solutions using a variety of enterprise integration tools.

9 References

[EIP] Hohpe, Gregor, Bobby Woolf, Enterprise Integration Patterns. Addison-Wesley, 2004

[EAA] Fowler, Martin, Patterns of Enterprise Application Architecture, Addison-Wesley, 2003

[ESP] Enterprise Solution Patterns, Microsoft Patterns & Practices, http://msdn.microsoft.com/architecture/patterns

[PAG] Patterns and Practices, Integration Patterns, Microsoft Patterns & Practices, http://msdn.microsoft.com/architecture/patterns

[SOINET] Implementing Service-Oriented Integration with ASP.NET, Microsoft Patterns & Practices, http://msdn.microsoft.com/architecture/patterns/default.aspx?pull=/library/en-us/dnpag/html/implsoiwithnet.asp

[SOIBTS] Implementing Service-Oriented Integration with BizTalk Server 2004, Microsoft Patterns & Practices, http://msdn.microsoft.com/architecture/patterns/default.aspx?pull=/library/en-us/dnpag/html/implsoiwithbts.asp

[CHAPPELL] Understanding BizTalk Server 2004, http://www.msdn.microsoft.com/library/default.asp?url=/library/en-us/BTS2004IS/htm/understanding_abstract_syfs.asp?frame=true

[BTSTUT] BizTalk Server 2004 Tutorial, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/introduction/htm/ebiz_ref_tut_intro_tegk.asp

[MEP] SOAP Spec

[ALEX] Alexander, Christopher, A Pattern Language – Towns, Building, Construction, Oxford University Press, 1977