This series explores reusable assets such as recipes, software patterns, and models. The series shows how you can accelerate the development of SOA solutions. This fifth installment in the series explores the preferred data source pattern, which addresses consistency non-functional requirements when implementing reusable services. The preferred data source pattern is a microflow pattern for service aggregation. It was harvested from a real SOA engagement, and it has been reused in several other SOA applications and engagements. This article also demonstrates how you can use a Rational® Software Architect implementation of this pattern in a model-driven development environment to create a new service implementation.

View more content in this series

Introduction

Part 1 and Part 2 in this series introduced the SOA service implementation and optimize recipe, as well as an accompanying reference example. This recipe, which is available as a reusable asset, provides prescriptive guidance on how to use non-functional requirements to determine which architectural patterns are needed to build architecturally consistent applications and to provide architectural traceability and accountability in terms of the decisions made. The recipe also contains a reference example that shows how to use a model-driven development (MDD) approach that leverages the modeling capabilities of IBM Rational Software Architect to develop use case, analysis, design, and services models.

Part 3 of this series also showed how a legacy application can be surfaced using a top-down approach. The reference example accompanying the SOA implementation and optimization services recipe detailed how a reusable catalog service model was identified and specified using domain decomposition of the lookup item business process. The WS response template pattern was then applied to the catalog service model to provide a more flexible service model by allowing the requester fine-grained access to a course-grained interface. The pattern was used because it matched the nonfunctional requirements of flexibility and interoperability required for this service implementation. The controller (used to implement the service business logic) of the catalog service implementation was then used to connect to the legacy catalog application. This is a classic example of a legacy application, and it provides the core functionality necessary to
provision a service. However, this application needed to be surfaced as a service, and it needed to adhere to a different set of non-functional requirements that existed when the original application was created.

**Part 4** in this series focused on satisfying the other non-functional requirements for this new catalog service. These non-functional requirements are focused around service performance, service scalability, and traceability. In an SOA environment, where the number of users of the service is often not known at the time the service is created, the non-functional requirements of performance and scalability are fortunately often commonplace, making them easy to find. A well-known pattern that addresses performance of non-functional requirements is the requester side caching pattern, which was presented in **Part 4**.

Part 5 demonstrates one pattern to show how a set of separate services can be federated into a single service, namely the preferred data source pattern. The preferred data source pattern is a microflow pattern for service aggregation. It was harvested from an IBM SOA engagement, and it has been reused in several other SOA applications and engagements. The context mapping of patterns to non-functional requirements and of non-functional requirements to architectural decisions enables the SOA developer or architect to build architecturally consistent code that is compliant with the software development best practices and principles.

**Following the story so far**

In previous articles in the series, you set up a reusable asset specification (RAS) client in IBM Rational Software Architect. Using this client, you have access to a reusable asset called the **SOA implementation and optimization of services** recipe. An SOA reference example (detailed in **Part 2** of this series) has actually been integrated into the SOA implementation and optimization of services recipe for illustration purpose. Before you can use this recipe, you need to set up a project in Rational Software Architect to put all the other reusable assets referred to by the recipe. Typically these assets are use case, services, analysis, and design models. To create a simple project in Rational Software Architect, select **File > Project > Simple > Retail**.

The SOA reference example demonstrates how the SOA patterns work together in a composable manner and how they might interact with other patterns. This example is loosely based on a retail chain, and it focuses on a particular business service called Lookup Item. Lookup Item would typically be used in a retail scenario where an item would first be looked up in an inventory system before it could be sold. Typically Lookup Item is an example of a coarse-grained business service. This article also explains the corresponding IT services needed to provision this business service.

Business process decomposition is used to better understand the complex processes used to provision this business service. Figure 2 shows the intricacies of this business process. In particular, note that two other services are used in Figure 2: the **catalog** and **inventory** services:

- The catalog service is used to find the particular part number or SKU of the item that is to be looked up.
- The inventory service uses this SKU to see if the item of interest is present in the inventory system. It first looks in a local inventory system. Failing this, it searches in a warehouse...
inventory. The process could be continued by searching an outside supplier. You see this in more detail in Figure 4.

Business process decomposition separates business services from business processes. A business analyst can think in terms of a Lookup Item service that would be part of some overall business process, without having to understand the intricacies of the service implementation. Business process decomposition enables you to understand the IT services that are needed to provision this process and its services.

In Part 3 and Part 4, you used an SOA pattern to help build the catalog service. By understanding the non-functional requirements of the catalog service, these requirements could be mapped, and the corresponding architectural decisions that need to be made to satisfy these requirements could be mapped to SOA patterns. This kind of mapping of non-functional requirements to patterns enables an SOA architect to create consistent architectures and to provide traceability and accountability in terms of the architectural decisions made. The two articles described how to engineer the catalog service using the following patterns:

- The Web services response template pattern to address service interoperability and flexibility
- The requester side caching pattern for service performance
- The aspect patterns for service traceability

In the case of the catalog service, an existing legacy catalog application existed, and you surfaced this as a service. In the case of the inventory catalog, no such existing application existed, requiring you to create a new service.

Creating a new service

In the reference example, there is an organization that manages multiple distributed inventories of items, each with its own Web service front end. From the use case analysis, one of the non-functional requirements dictates that the inventory service should provide a unified view across multiple, disparate data sources. In essence, the multiple inventories should appear to the client as a single inventory. In addition to federating the services, the requirements also indicate that an order of preference be implemented, so that certain preferred inventories are checked first. The alternate inventories are checked only when the preferred inventories are not found. The federated inventory service should then build up from multiple back-end data sources. The SOA reference example contains a single local inventory and a single remote inventory. The following reusable assets are available assets to build the inventory service:

- A model that represents an inventory service and related data representations: A service design model packaged as an RAS asset can be imported into a UML model project.
- The non-functional requirements for the inventory use case: The RequisitePro® requirement management file could also be made available as a RAS asset, but it is more likely to be available across a network for shared access.

Examine the steps outlined in the recipe (Figure 3) to help construct a new service. See Part 1 for details on how to get this recipe.
Figure 3. Review service model

The inventory service

The inventory service provides two operations. The first operation is a query to determine if the inventory contains enough units of a particular item to meet the quantity needed for an order. In the retail industry, the number of items available in stock is called the quantity on hand, or QoH. The second operation is to record a change in inventory levels, either because an item had been sold or new stock has arrived, which is called a stock movement. This service can also be visualized at two different levels of abstraction, similar to the catalog service above:

- The inventory service UML model: Inventory Service
- The inventory service specification (WSDL)

These are explained in more detail in the next section.

The inventory service model

Figure 4 shows the inventory service model. Following on the service description above, the model has two operations:

- The `getQoH()` operation. This operation takes a `QoHRequest` and returns `QoHResponse`.
- The `inventoryMovement()` operation. This operation takes an integer and returns a Boolean.

The inventory data model

The inventory data model (Figure 5) allows the request to specify the location of a source of inventory to be specified. An example might be a store or regional warehouse.

The next step in the creation of a new service is to review the non-functional requirements of the service (Figure 6).
Figure 6. Review the non-functional requirements of the service

Review of non-functional requirements

The nonfunctional requirements for the inventory service are tracked through Rational RequisitePro. Figure 7 shows the RequisitePro client inside Rational Software Architect, which
enables the architect to maintain and examine the tractability between the use cases, the functional requirements, and the non-functional requirements. Following is a summary of the high-level non-functional requirements for the inventory service:

- **Consistency**: The inventory services must provide a unified view into all of the disparate inventory services data sources that are in use.
- **Performance**: The inventory service must perform within acceptable time limits.
- **Traceability**: All invocations of the inventory service must be logged.
- **Scalability**: The implementation for the various inventory services must be scalable to deal with the appropriate load demands on the server.

The next step in creating the new service is to review the existing legacy analysis model (Figure 8). This typically details how the existing legacy application is being used or accessed.

**Figure 8. Review the existing legacy analysis model**

---

**Review of the existing catalog design model**

Both the local and remote inventory services have the same interface. In the simplified example, there is a single method on the service to federate: `getQoH()`. This method accepts a request object that specifies information about an item that might be in the inventories. The return object indicates the quantity on hand and the specific location ID. The UML diagram in Figure 9 shows the service interface, its implementing component, and the messages that are passed in the main method.
Figure 9. Inventory service design

The current state of the design puts the responsibility on all of the clients to navigate the various back-end services. A disadvantage of this design is that when new services are added or endpoints are changed, they must be updated and changed in each of the clients. In retail applications, this could mean simultaneously updating clients in hundreds of store outlets. An advantage of this design is that each store can independently decide which inventory services to search and in what order. The scenario for this article, however, imposes the functional requirements that all clients of the services are expected to search the various inventories in exactly the same way and that there is a single point of management.

Applying the appropriate patterns

Part 3 showed how to apply the WS response template pattern to a model of the service to make the service more flexible. This Part 5 considers the requester side caching pattern and the aspect logging pattern.

SOA is about business agility. One way to achieve this kind of agility is to identify the business components that are core to the business. After these business components have been identified, the business processes and services associated with these business components can be identified and specified. Business process decomposition for these business processes and services can reveal the reusable IT services needed to provision them. The input and output messages of these IT services and the domain objects that these services manipulate also need to be specified. From an IT perspective, these services can be modeled as use cases, and their functional and non-functional requirements can be traced using a tool like Rational RequisitePro. With all the moving parts, how can the complexity of this kind of environment be managed? In particular how can you build out these services to ensure that the non-functional requirements of these services are satisfied?
Patterns are a way to help manage this complexity. Patterns are a repeatable solution to a problem in context. They typically are described by a pattern specification. Pattern specifications include a forces section, which outlines when the pattern should be used. Feasibility for a solution is ultimately determined by non-functional requirements, such as scalability, performance, security, transactionality, maintainability, and interoperability. By mapping these non-functional requirements to the forces section in the pattern specification, you can achieve traceability and accountability in terms of the architectural decisions.

This series of articles addresses four SOA patterns. Each of these SOA patterns satisfies certain non-functional requirements typical to an SOA application. Following is a list of these patterns and a brief description of the non-functional requirements that they each address:

- The **WS response template pattern specification** provides service interface flexibility, interoperability, and maintainability. See Resources for the WS response template pattern specification.
- The **requestor side caching pattern specification** improves service performance. See Resources for the requester side cache pattern specification.
- The **preferred data source pattern** is a microflow pattern for service aggregation.
- The **logging pattern** provides service invocation traceability. See Resources for more on model-driven development.

Before examining these patterns in detail, it is helpful to consider an n-tier architecture, as outlined below. A three-tiered architecture has a presentation tier, a business tier, and a persistence tier. In an SOA environment when you are dealing with the implementation of reusable IT services, you can separate the business tier further into a service layer, a controller layer, and an entity/object management layer. This layering is shown in Figure 1. This figure illustrates the layering in the context of an application server container.

The service layer is responsible for specifying the operation used in that service and for specifying the content of the message these operations use. The service layer is also responsible for serializing and de-serializing messages in and out of the container. The controller layer is responsible for implementing the business logic of the service. The controller layer can achieve this by invoking other services, other controllers, or an entity management layer. The entity management layer is responsible for entity management. The entity management layer ensures the transaction integrity of that object within the container.

The WS response template described in Part 3 can be applied to the service layer (other patterns, such as a WS security pattern, that directly impact the service definition would be applied as this layer). The requester side caching pattern, which was described in Part 4, is applied at the controller layer. The aspect logging pattern, which was described in Part 4, is also applied at the controller layer, as is the preferred data source pattern. The session facade, message facade, and business delegate core Enterprise JavaBeans patterns belong in the controller tier, while the data management pattern of the entity and data access object core enterprise JavaBeans patterns belong in the entity layer. Figure 1 shows the n-tier layered architecture for service implementation. Figure 10 shows where these patterns are applied.
Mapping patterns to non-functional requirements and building the inventory service

In addition to the main functional requirement of providing inventory services, additional architectural decisions regarding non-functional requirements, such as data consistency, performance, and transactionality, can be addressed through the use of patterns. In a model-driven development environment, such as Rational Software Architect, the architect or developer can use an analysis model to apply the appropriate patterns graphically to satisfy these requirements into one common, federated service as additional requirements (both functional and non-functional). Following the reference example in the SOA implementation and optimize Services recipe, the federated inventory service (information service) has the following non-functional requirements, which can be addressed with the following patterns.

To build the catalog service, three patterns are used:

- **Unification/consistency**: The preferred data source (PDS) pattern
- **Performance**: The requestor side cache pattern
- **Scalability**: An EJB architecture leveraging well-known EJB patterns for creating a scalable implementation
- **Traceability**: The AspectJ based logging pattern for logging all service invocations

The construction of the federated inventory services is shown in Figure 11. You can see the preferred data source pattern used to federate the microflow. The requester side caching pattern could be used to enhance the federated search's performance. Both these patterns are applied at the controller layer.

**Figure 11. Building the virtual inventory service using patterns**

![Diagram of building the virtual inventory service using patterns](image)

Figure 12 shows building the local inventory service. The requester side caching pattern could again be used to enhance performance at the controller layer. The EJB patterns of the business
delegate and the session facade would also be applied at the controller layer. Finally, the data access object (DAO) pattern would be applied at the entity management layer.

Figure 12. Building the local inventory service using patterns

Mapping patterns to non-functional requirements

When talking about patterns (use the Gang of Four design patterns as an example), there are two different elements of interest. One element is the actual text that describes the pattern. This is what you would find in the book, such as in the chapter about the adapter pattern.

The other element is the design tooling elements that implement that pattern. For example, there are UML artifacts in Rational Software Architect that implement the different design patterns from the Gang of Four book. So, if you want to apply an adapter to one of your designs, you would pull that element off of the palette and make some manipulations in the tool to actually transform your design to use the pattern.

One result is called a pattern specification. The other result is called a pattern implementation.

What is described in this article is a pattern implementation.

As you can see, there is a fine balance between patterns and non-functional requirements, especially when you are engineering information services. You need to consider both the non-functional requirements of the data and of the service. As you consider the patterns used to build information services, it is very important to consider the data patterns. There have been two recently published data pattern specifications for building SOA informational services:

- The data federation pattern virtualizes data from multiple disparate information sources. The pattern creates an integrated view into distributed information without creating data redundancy, and while federating both structured and unstructured information. This pattern specification helps data and application architects make informed decisions about data architecture and document decision guidelines.
The data consolidation pattern specification helps data and application architects make informed architectural decisions and improve decision guidelines. In this article, you can see how to apply the pattern in the SOA context. The primary business driver for the data consolidation pattern (also referred to as the data population pattern) is to gather and reconcile data from multiple data sources before this information is needed. To do so, it extracts data from one or more sources, transforms that data into the desired target format, and loads it into a persistent data target. The pre-population of a persistent target is a key differentiator between this pattern and the data federation pattern.

The sidebar describes the preferred data source pattern in more detail.

**Using the preferred data source pattern**

The preferred information source pattern provides the capability for a client to retrieve information from a set of information sources, without the need to understand (at least at more than a high level) that there are multiple sources. One of the sources is identified as the preferred source, and the others are considered alternate sources, used only when the preferred source cannot provide the desired information. All sources are invoked in a fixed order.

**Non-functional requirements**

The preferred data source is used to satisfy the non-functional requirement of unification, where access to disparate data source requires a common point of access.

**Component instance diagram**

Figure 13 shows the component instances that represent the application of this pattern. A new component instance of the service specification, called the *combined service*, acts as a single point of access for all backend services. Only clients communicate with the combined service, which implements the preferred data source algorithm.

**Figure 13. Preferred data source component instance diagram**

**Sequence diagram**

Figure 14 shows a typical scenario in the form of a sequence diagram. In this scenario, a client uses the combined service to access the backend services. The combined service is responsible for invoking the backend services in the correct order and for returning the first qualified (non-null) response to the calling client. Note that for clarity, the synchronous returns are not shown.
Understanding the preferred data source pattern implementation

You can now use the pattern specification to create several implementations of the preferred data source pattern. Typically any implementation provides some level of automation for applying the pattern. The implementations this article describes use the model-driven development environment of Rational Software Architect. This section describes an implementation of the preferred data source pattern using the Rational Software Architect patterns engine.

This Rational Software Architect patterns engine implementation assumes a UML model representation of the service or interface that needs to be accelerated. The approach to using this pattern follows the model-driven development approach of instantiating the pattern parameters to specific UML model elements of the service or interface. After the pattern parameters are bound, additional UML elements are automatically created. A UML-to-Java™ transformation, invoked on the resulting model, generates the resulting Java implementation artifacts.

This implementation of the preferred data source pattern is created using the Rational Software Architect pattern engine. Creating a pattern with the Rational Software Architect pattern engine yields an Eclipse plug-in. This plug-in can be packaged as a reusable asset using the Reusable Asset Specification (RAS) (see Resources to learn more about the Reusable Asset Specification; see Part 1 to understand more about patterns and reusable assets). The result of this packaging is an RAS asset. This RAS asset, along with its associated meta-data, can then be deployed to a RAS server, such as the developerWorks RAS repository. In the next section, you will see how to access this pattern asset from the developerWorks RAS repository using a Rational Software Architect RAS client. This pattern asset can imported into Rational Software Architect, and Rational Software Architect functionality can be extended to allow a user to apply the requester side caching pattern to a model.
Applying the preferred data source pattern

The preferred data source pattern implementation can be imported into Rational Software Architect using the recipe. First, navigate to the **Apply patterns to a service implementation** section in the recipe and expand the section titled **Applying the preferred data source pattern**. Locate the asset under this step and select **Import**, as shown in Figure 15.

You can see how this pattern is applied and used by downloading and watching the **Flash file**.

**Figure 15. Import the preferred data source pattern implementation**

This installs the preferred data source pattern implementation into Rational Software Architect. After the pattern has been installed, it shows up in the pattern explorer, as shown in Figure 16.

**Figure 16. Pattern explorer showing the preferred data source pattern**

Next, apply the pattern. The following steps show how to navigate the recipe and import the legacy catalog model into Rational Software Architect. The requester side caching pattern is then applied to this model, and the corresponding cache-aware code is generated in a UML-to-Java transformation.

1. Open the UML Model of the service, and create the WSDL using a UML2WSDL transformation.
2. Create one primary WS implementation of the server using the generated WSD.
3. Create one or more alternative WS implementations using the same WSDL, and repeat as necessary.
4. Create a simple test implementation to test each services implementation. In general, each implementation of a method examines a value or property of an incoming parameter. Then, decide whether to return a null value (or empty list) or to return an instance of the...
object that the method is supposed to return. For example, given the getQoH() method, the implementation is as follows:

```java
public com.ibm.retail.inventory.QoHResponse getQoH(
    com.ibm.retail.inventory.QoHRequest qoHRequest)
throws java.rmi.RemoteException { System.out.println("Primary");
    if (qoHRequest.getLocationID() == 1) {
        com.ibm.retail.inventory.QoHResponse qoh =
            new com.ibm.retail.inventory.QoHResponse();
        qoh.setQuantityAvailable("some"); return qoh; }
    return null; }
```

An alternate implementation would be as follows:

```java
public com.ibm.retail.inventory.QoHResponse getQoH(
    com.ibm.retail.inventory.QoHRequest qoHRequest)
throws java.rmi.RemoteException { System.out.println("Secondary");
    if (qoHRequest.getLocationID() == 2) {
        com.ibm.retail.inventory.QoHResponse qoh =
            new com.ibm.retail.inventory.QoHResponse();
        qoh.setQuantityAvailable("lots"); return qoh; }
    return null; }
```

5. Create a virtual Web service implementation of this same WSDL. However, when creating this implementation, be sure the Web project is also a client of the same WSDL, in addition to being a server. Note that this yields a warning message during one of the wizard's steps (see Figure 18). Because this is a special case scenario, you can safely ignore this warning message.

**Figure 17. Create a virtual Web service implementation of the virtual proxy**
6. Open the design model for the virtual service (or create a new if none exists).
7. In this model, create a UML package that corresponds to the main implementation package of the virtual service’s implementation (such as com.ibm.retail.inventory).
8. In this package, visualize the service interface (IInventory) and the service locator class (IInventoryLocator).
9. Add a new instance of the preferred data source pattern to the model.
10. Bind the service interface to the pattern, such as InventoryService.
11. Bind all the methods that the router manages, such as getQoH().
12. Bind the locator visualization to the pattern. The result is the creation of two new classes in the model:
   - A router implementation that intercepts requests to the service and routes them to the primary and alternate services as necessary.
   - A servlet class that initializes and enables the updating of the primary and alternate service list during runtime.
13. Using the UML-to-Java transformation, generate the code for these two classes into the virtual service Web project. The result should be the creation of two new classes in the main package: a new servlet entry in the Web deployment descriptor and a runtime configuration file (sources.xml) in which the actual endpoints of the services are placed. If the service definition specified has exceptions, the router class needs to implement them.

14. Provide an implementation of the virtual Web service to delegates of the newly created router. See the following code as an example:

```java
public com.ibm.retail.inventory.QoHResponse getQoH(
    com.ibm.retail.inventory.QoHRequest qoHRequest)
    throws java.rmi.RemoteException {
    InventoryServicePrefSvcRouter router =
    InventoryServicePrefSvcRouter.getInstance(); return router.getQoH(qoHRequest); }
```

15. Edit the sources.xml file. Change the name of the service to `InventoryService`, and then update each of the endpoint elements with the actual endpoint information. You can obtain this actual endpoint information by testing and examining the primary and alternate Web service implementations.

```xml
<facades service="InventoryService"> <source
    endpoint="http://localhost:9080/LocalInventory/services/InventoryServicePort"/> <source
    endpoint="http://localhost:9080/RemoteInventory/services/InventoryServicePort"/> <facades>
```

16. Before testing the application, initialize the service router. Right-click the initialization servlet node under the deployment descriptor, and select Run.
What have you accomplished?

In this example, you examined a system's requirements to federate a number of inventory services. You also provided a single contact point that gives preference to an ordered set of services when finding items. Then you applied an existing pattern to the set of design artifacts that are being developed for this system.

Figure 21. Preferred data source pattern applied to the reference example

Conclusion

Through the use of a model-driven development process, core functionality has been leveraged and reused in a form that makes it consistent and of higher quality. The WS preferred data source pattern is a well-documented pattern, and it commonly occurs in enterprise and SOA environments. The actual pattern implementation described in this article provides a consistent and easy way to introduce this architectural mechanism to development teams with some degree of assurance that each implementation is consistent. Consistent application of architectural mechanisms not only promotes easier maintenance, but it helps ensure higher quality, because discovered defects can be fixed for all applications.

This pattern can be combined with other patterns in the SOA implementation and optimize services recipe to solve other functional and non-functional requirements. The requestor side cache pattern from the example can be added to the preferred data source implementation, speeding up the performance of slow services. The AspectJ based logging patterns can be used to log or audit service usage without significantly affecting the preferred data or caching functionality.
Downloadable resources

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Final interchange project for the article</td>
<td>requesterSideCachingPatternCatalogApp.zip</td>
<td>149KB</td>
</tr>
<tr>
<td>Applying the preferred data source pattern</td>
<td>PDS.swf</td>
<td>2955KB</td>
</tr>
</tbody>
</table>

© Copyright IBM Corporation 2007
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)