



IBM Software Group

## **WebSphere® Commerce Feature Pack 2**

### *Business Component Services*

#### *Technical overview*



@business on demand.

© 2007 IBM Corporation  
Updated May 15, 2007

Welcome to the WebSphere Commerce Feature Pack 2 Business Component Services Technical Overview presentation.

## Agenda

- Architecture overview
- Component façade
- Service binding
- Client library
- Design Pattern Toolkit



The agenda for this presentation is to discuss the architecture overview, the component façade, the service binding, the client library and the Design Pattern Toolkit.

## Section

# ***Architecture overview***



This section discusses the architecture overview.

## Architecture overview

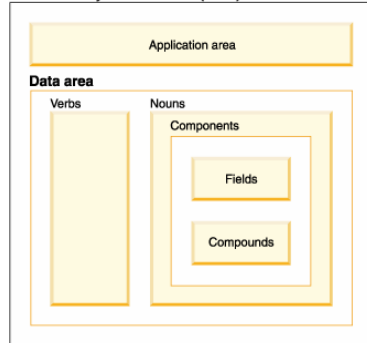
- Open Applications Group Integration Specification (OAGIS) services (Business Object Document)
  - ▶ Noun represents the business object
  - ▶ Verb represents the action to perform
  - ▶ Implemented by Service Data Objects (SDOs)
- Component façade implementing the services
  - ▶ Services are implemented as commands
- Service binding of the client to the component façade
  - ▶ Can be a local Java™ call or a remote Web service
- Client library provides Java™ friendly and Web enabled methods to invoke the service

The Business Component Services architecture is comprised of four parts: OAGIS based services, the Component Façade, the Service binding and the Client library.

Since the introduction of the Sales Center, **OAGIS** has been used as the XML notation to represent services. Within Feature Pack 2, focus has been placed on how OAGIS is used to represent services to retrieve data, update data or run business processes. An OAGIS message contains a noun which is the business object that is part of the service and a verb which represents the action to perform. Instead of using XML marshalling and unmarshalling code, Service Data Objects (SDO) are leveraged to implement the services. The **component façade** is the session bean that implements the services and provides the Web service interface. It is the entry point into the WebSphere Commerce system to run your business logic. The **service binding** defines how the communication between the client and the service is performed. In the case of Portal Integration, remote Web service calls are used since WebSphere Portal and WebSphere Commerce are on different machines. However, in the Web 2.0 Store where the struts presentation layer resides in the same Java application as the service itself, you would not want to use a Web service call but rather a local Java call. The purpose of the **client library** is to provide Java friendly and Web enabled APIs that simplify client development when communicating with the service.

## Business Object Document

Business Object Document (BOD)



- Application area
  - ▶ Contains application context information (language, store and so on)
- Verb
  - ▶ Get / Show
  - ▶ Process / Acknowledge
  - ▶ Change / Respond
  - ▶ Sync / ConfirmBOD
- Noun
  - ▶ Logical representation of the business object

The Business Object Document (BOD) is an open standard for a common horizontal message architecture developed by the Open Applications Group. BODs are business messages exchanged between software applications or components. A BOD is a message structure that contains an application area and a data area to operate on. The application area of the BOD describes the application context to associate with the processing. The application area associates specific application context information to control the processing and indicate application context when returning the result. In the context of a WebSphere Commerce service, the application area is where store, language and other session type data is found. This information is common across all component facades and is used during processing. The data area contains the service operation (Verb) and the business object of the operation (Noun). The verb indicates what operation to perform or the response of the service operation. In the case of a request, it contains details of the operation and in the case of a response, it contains information pertaining to the response. The noun represents the data contained in the business object pertaining to the request or response.

## OAGIS verbs (1)

### ■ Get / Show

- ▶ *Get* verb retrieves the business objects
- ▶ XPath expression used for search criteria
- ▶ *Show* response includes business objects that match the search criteria

### ■ Process / Acknowledge

- ▶ *Process* is used when client needs to submit, cancel or invoke logic against an existing business object
- ▶ *Acknowledge* response contains unique identifier for new or changed business objects



OAGIS comes with a set of 12 predefined non-deprecated verbs. In this Feature Pack, 8 of the 12 verbs are used where one verb represents the request and one verb represents the response. The 8 verbs are Get, Show, Process, Acknowledge, Change, Respond, Sync and ConfirmBOD.

Get requests are requests for data that use the Get verb, and return a *Show* response. The **Get** verb indicates the search criteria used to retrieve the business objects along with paging options for paged requests. The **Show** response includes the business objects that match the search criteria.

Following the OAGIS model for read requests, the best practice is for every read request to be represented as a Get Noun document where Get is the action to perform and the Noun is the business object or objects to return. As a response to the Get request, the Show document is returned with the requested data. The search expressions are XPath expressions which represent a query against the respective noun to indicate the data to be returned. As part of the extension to the XPath syntax, there is a way to pass control information to indicate the amount of data to return, ordering and other search result control information. Read requests use XPath expressions to define the response data to be returned.

**Process** is used when the client is submitting a new business object, canceling an existing business object, or invoking processing logic against an existing business object. The **Acknowledge** response includes the shell of the business object that was processed, containing at a minimum the unique identifier for the new or changed business object. On Process requests, actions are user-defined. Example process actions may include: 'Submit', 'Register', or 'Release'. The action must indicate the business operation that needs to take place on the included Noun. There is no predefined list of actions that can be associated with the Process verb, because the action is very specific on the supported business process associated with the business object.

## OAGIS verbs (2)

- Change / Respond
  - ▶ *Change* notifies the master repository to change the business object they own
  - ▶ *Respond* response contains the unique identifier for the new or changed business object
- Sync / ConfirmBOD
  - ▶ *Sync* synchronizes the business object across systems
  - ▶ *ConfirmBOD* indicates whether processing was successful

**Change** is used to notify the system that owns the business object that they should change a business object that they own. The Change message contains the verb and the noun. The noun contains the information to uniquely identify itself and the attributes to change. For example, when adding a contact to a customer's contact list (address book), the noun specified contains the contact information to add along with the information to uniquely identify the Person business object. The response to a Change request is a **Respond** BOD that contains a Response verb and a noun. The Noun refers to the data that was changed. The Respond verb contain the response criteria one for each action expression specified in the request.

**Sync** is used to notify interested parties the current state of the business objects that the system manages. Only the system that contains the master data record should be sending Sync requests. Sync is used to synchronize the business object or objects across systems and is initiated by the system that holds the master data record for the business object.

A common example of when sync is used is asynchronous updates by JMS. Sync is not limited to this usage but typically when backend systems are pushing updates, these updates are sent asynchronously and reliably. For example, the master data record for Member uses Sync BOD to broadcast updates. If SAP® is the master data record for Order, then a Sync BOD request is received when the order status changes. As a response to a Sync request, the **ConfirmBOD** response is used to indicate whether processing was successful. The confirm response contains the success or failure message as a result of the synchronization. The response will not include any information about the noun itself. The purpose of Sync is to synchronize the system so the Nouns currently in the system match the Nouns within the request message. For Sync requests, actions are user-defined. The action must indicate the business operation that needs to take place on the included Noun. There is no predefined list of actions that can be associated with the Sync verb.

## Noun extension model

- **UserData element**
  - ▶ Business objects have UserData extension points for different complex types
  - ▶ Extension points are name-value pairs added to the complex type
  - ▶ Client needs to populate data to be passed to the service
- **Overlay**
  - ▶ Extension of an existing complex type (similar to inheritance in Java)
  - ▶ Replace usage of default complex type with the extension



From a noun perspective, because only 10% of the capability of OAGIS nouns are being used, Feature Pack 2 introduces nouns in the form of an extension model called UserData Elements. Those familiar with the WebSphere Commerce Messaging system or the MQ XML processing within WebSphere Commerce will notice that the default XML messages have the concept of user data which is a name value pair element that can be added to your XML syntax. These eventually get mapped to name value pairs when passed to the controller commands to run.

This concept has been reused when the nouns were defined. Within the many complex types of nouns, UserData elements have been added as an easy way to extend the data that is passed from the client to the server. If you want to add information to the noun, or to the complex type of the noun, you would populate these UserData elements. As a result, without any coding effort on the server side or regeneration of the Web service, this data would appear on the server side and could be processed immediately.

The drawback to the UserData element is that it just supports strings. It is a simple string name value pair parameter and may not represent the type of extension you want to build. Because of this, Feature Pack 2 introduces the use of Overlays. Overlays, a part of the OAGIS model, extend complex types similar to the inheritance model in Java. Within the Overlay model, you would extend your complex type defined within the noun and generate Java code to represent that complex type. More specifically, where you referenced the default WebSphere Commerce version of the complex type, you would reference your new version of the complex type and substitute it in. Within the client server framework, when the complex type is sent from the client to the server, regardless of whether it is a local Java call or a remote Web service, when the business logic runs it retrieves your version of the extended complex type. This model provides another way to add additional information, as it allows you to create structured Java objects to better represent the data that you want to flow from one system to another.



## Get request / Show response

```
<_cat:GetCatalogEntry xmlns:_cat="http://www.ibm.com/xmlns/prod/commerce/9/catalog"
xmlns:_wcf="http://www.ibm.com/xmlns/prod/commerce/9/foundation"
xmlns:oa="http://www.openapplications.org/oagis/9" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance">
  <oa:ApplicationArea xsi:type="wcf:ApplicationAreaType">
    <oa:CreationDateTime>2007-01-25T16:44:16.004Z</oa:CreationDateTime>
    <oa:BODID>341d6660-acbd-11db-a3a8-832d45b924b0</oa:BODID>
    <_wcf:BusinessContext>
      <_wcf:ContextData name="storeId">511</_wcf:ContextData>
    </_wcf:BusinessContext>
  </oa:ApplicationArea>
  <_cat:DataArea>
    <oa:Expression
      expressionLanguage="wcf:XPath">{ibmwcf.ap:WC_CatalogEntryDetailsProfile}/CatalogEntry[CatalogEntryIden-
tifier[(UniqueID='51100000117')]]</oa:Expression>
    </oa:Expression>
  </_cat:DataArea>
</_cat:GetCatalogEntry>
```

---

```
<_cat:ShowCatalogEntry xmlns:Oagis9="http://www.openapplications.org/oagis/9"
xmlns:_cat="http://www.ibm.com/xmlns/prod/commerce/9/catalog"
xmlns:_wcf="http://www.ibm.com/xmlns/prod/commerce/9/foundation"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <Oagis9:ApplicationArea xsi:type="wcf:ApplicationAreaType">
    <Oagis9:CreationDateTime>2007-01-25T17:01:18.297Z</Oagis9:CreationDateTime>
    <Oagis9:BODID>954527f0-acbf-11db-986f-827045b7cf58</Oagis9:BODID>
  </Oagis9:ApplicationArea>
  <_cat:DataArea>
    <Oagis9:Show/>
    <_cat:CatalogEntry catalogEntryTypeCode="ProductBean">
      <_cat:CatalogEntryIdentifier>
        <_wcf:UniqueID>51100000117</_wcf:UniqueID>
        <_wcf:ExternalIdentifier ownerId="7000000000000000002">
          <_wcf:PartNumber>TAWI-03</_wcf:PartNumber>
        </_wcf:ExternalIdentifier>
      </_cat:CatalogEntryIdentifier>
      <_cat:Description language="-1">
        <_cat:Name>Corrolus Wineglasses</_cat:Name>
        <_cat:Thumbnail>images/catalog/TAWI_03_sm.jpg</_cat:Thumbnail>
        <_cat:FullImage>images/catalog/TAWI_03.jpg</_cat:FullImage>
        <_cat:ShortDescription>Corrolus wineglasses</_cat:ShortDescription>
        <_cat:LongDescription>Corrolus wineglasses</_cat:LongDescription>
      </_cat:Description>
    </_cat:CatalogEntry>
  </_cat:DataArea>
</_cat:ShowCatalogEntry>
```

WebSphere Commerce Business Component Services

© 2007 IBM Corporation

Within the OAGIS request, the Get Business Object Document has two sections, the application section and the data section. The application section contains the application specific data. In this example, the only application data is the store ID of the store you want to fetch data for. The data area section of the Business Object Document has the verb and the noun that you want to act upon.

Get requests can be compared to an SQL statement where the table is the noun, the row is the access profile and the condition is the XPath expression. Since the way data is expressed is with an XPath expression, you do not need to provide the noun portion of the request, just the get verb with the expression of the query you want to run. This example shows the XPath notation that represents the catalog entry with a unique ID of 51100000117. The XPath notation represents the 'where' condition in SQL. The difference between the implementation is that there is no direct mapping between the XPath notation and the SQL that is run on the server side. This prevents SQL injection problems because in order to support the XPath expression the server has to have code to be able to represent the XPath expression in the corresponding implementation.

The bottom half of this example shows how an access profile can be used to indicate how much data you want returned as part of the Show response. In the expression for CatalogEntry, you want the data to be returned as a detailed profile so that you can present all of the detailed information about that catalog entry. With this notation, the client can specify how it uses the data, i.e. to populate a form or product page. If you change the access profile to summary and keep the same expression, a summary profile is returned such that enough information is available to display in table format. By using the access profile, you can control the amount of data without having to create a new service that represents a specific view of the data.

## Process request / Acknowledge response

|  |   |
|--|---|
| <pre> _mbr:DataArea&gt;   &lt;oa:Process&gt;     &lt;oa:ActionCriteria&gt;       &lt;oa:ActionExpression actionCode="Register"         expressionLanguage="XPath"&gt;         /Person[1]       &lt;/oa:ActionExpression&gt;     &lt;/oa:ActionCriteria&gt;   &lt;/oa:Process&gt;   &lt;_mbr:Person&gt;     &lt;_mbr:PersonIdentifier /&gt;     &lt;_mbr:ParentOrganizationIdentifier /&gt;     &lt;_mbr:Credential&gt;       &lt;_mbr:LogonID&gt;test121622&lt;/_mbr:LogonID&gt;       &lt;_mbr:Password&gt;web1admin&lt;/_mbr:Password&gt;       &lt;_mbr:SecurityHint /&gt;     &lt;/_mbr:Credential&gt;     &lt;_mbr:ContactInfo&gt;       &lt;_wcf:ContactInfoIdentifier&gt;         &lt;_wcf:ExternalIdentifier&gt;           &lt;_wcf:PersonIdentifier /&gt;         &lt;/_wcf:ExternalIdentifier&gt;       &lt;/_wcf:ContactInfoIdentifier&gt;       &lt;_wcf:ContactName /&gt;       &lt;_wcf:Address&gt;         &lt;_wcf:City&gt;Toronto&lt;/_wcf:City&gt;       &lt;/_wcf:Address&gt;       &lt;_wcf:Telephone1 /&gt;       &lt;_wcf:Telephone2 /&gt;       &lt;_wcf:BestCallingTime&gt;Evening&lt;/_wcf:BestCallingTime&gt;       &lt;_wcf:EmailAddress1&gt;         &lt;_wcf:Value&gt;abc@123.com&lt;/_wcf:Value&gt;       &lt;/_wcf:EmailAddress1&gt;       &lt;_wcf:EmailAddress2&gt;         &lt;_wcf:Value&gt;abc@456.com&lt;/_wcf:Value&gt;       &lt;/_wcf:EmailAddress2&gt;       &lt;_wcf:Fax1 /&gt;       &lt;_wcf:Fax2 /&gt;     &lt;/_mbr:ContactInfo&gt;   &lt;/_mbr:Person&gt; &lt;/_mbr:DataArea&gt; </pre> | <pre> _mbr:DataArea&gt;   &lt;Oagis9:Acknowledge /&gt;   &lt;_mbr:Person&gt;     &lt;_mbr:PersonIdentifier&gt;       &lt;_wcf:UniqueID&gt;17002&lt;/_wcf:UniqueID&gt;       &lt;_wcf:DistinguishedName&gt;         uid=test121622,cn=users,dc=ibm,dc=com       &lt;/_wcf:DistinguishedName&gt;     &lt;/_mbr:PersonIdentifier&gt;     &lt;_mbr:ContactInfo&gt;       &lt;_wcf:ContactInfoIdentifier&gt;         &lt;_wcf:UniqueID&gt;34801&lt;/_wcf:UniqueID&gt;         &lt;_wcf:ExternalIdentifier&gt;           &lt;_wcf:ContactInfoNickName&gt;             test121622           &lt;/_wcf:ContactInfoNickName&gt;         &lt;/_wcf:ExternalIdentifier&gt;         &lt;_wcf:PersonIdentifier&gt;           &lt;_wcf:UniqueID&gt;17002&lt;/_wcf:UniqueID&gt;           &lt;_wcf:DistinguishedName&gt;             uid=test121622,cn=users,dc=ibm,dc=com           &lt;/_wcf:DistinguishedName&gt;         &lt;/_wcf:PersonIdentifier&gt;       &lt;/_wcf:ContactInfoIdentifier&gt;     &lt;/_mbr:ContactInfo&gt;   &lt;/_mbr:Person&gt; &lt;/_mbr:DataArea&gt; </pre> |
|--|---|

This example displays what the Process and Response messages look like from a data area perspective. The data area consists of the verb which is Process and the noun which is the action performed on the public part of the verb. On the left at the top, there is an action code of Register and it points to the first person within the request. Although the syntax of the OAGIS verb supports multiple expressions, for Feature Pack 2, only one action per verb is allowed. The second section on the left displays the noun information that is used to run the service operation. It includes the login ID, the password, the city and so on as expected when a user performs a register.

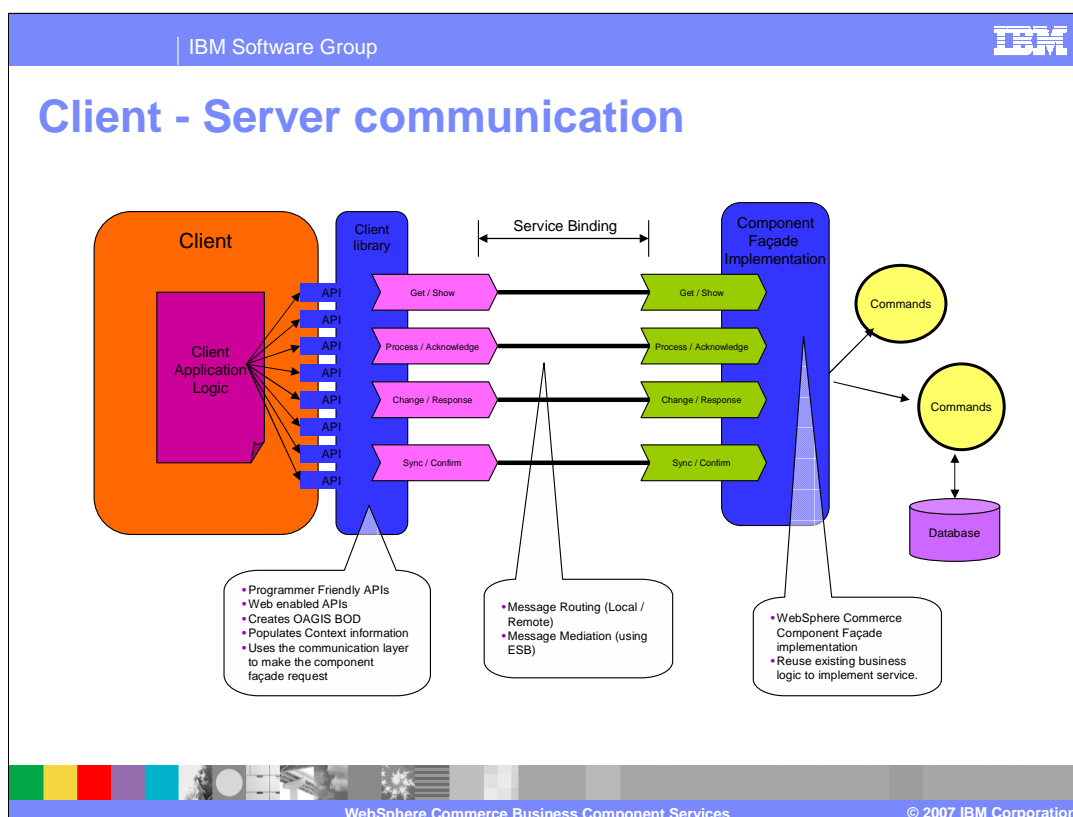
In response to the Process request, you get an Acknowledge response. It is a verb that has error information within the acknowledge. In this example, there is no error information to return to the client so the acknowledge is a placeholder verb element. The noun portion of the acknowledge has a unique ID and important information that the client may want to use to reference the noun. In this example, the distinguished name and the internal ID of the person being registered are returned.

## Service Data Objects (SDO)

- Java representation of the Service Objects
  - ▶ Interfaces and implementations that represent the structures defined in XSDs
  - ▶ Capability to create XML representation of Java objects
  - ▶ Has support of the Overlay extension model



SDO (Service Data Objects) is a framework for data application development, which includes an architecture and API. WebSphere Commerce utilizes the SDO capabilities for XML marshalling and unmarshalling, and code generation from XSD to static Java objects. In Java, the data objects of the service module, contained in the data object project, are represented as service data objects (SDOs). This module contains the XSDs of the data objects along with the WSDL of the services and the generated SDO code to be used by the component facade implementation and the Java client.



The following diagram displays the client server communication flow. The diagram can best be explained if read from right to left starting with the Component Façade.

### Component Façade

A component façade is used to group a set of related business services, such as catalog or order. Component façades are implemented as Java interfaces and their business services are implemented as methods on these interfaces. Each method in a component façade represents the request business object document (input argument to the method) and the corresponding response business object document (result of the method). The method name is the request verb name plus the noun name. For example, a member component façade might implement the Get verb on the Person noun which results in a `getPerson()` method on the façade. The component façade accepts OAGIS messages that contain a combination of verb and noun. It then calls WebSphere Commerce commands to provide the business logic associated with different services. WebSphere Commerce component façades accept four types of service request response combinations: Get/Show, Process/Acknowledge, Change/Respond and Sync/ConfirmBOD

### Service binding

The service binding resides between the client library and the services. Its responsibility is to provide the transport mechanism to pass data, using service data objects, between the client and the service. The transport mechanism can be Web services or local Java binding. The service binding can be leveraged with or without the client library. HTML requests use the Struts framework, which calls the client library. The requests are received as name-value pairs and are converted into the appropriate OAGIS messages. Java® Server Page (JSP) requests use the WebSphere Commerce foundation tag library, which calls the client library. SOAP requests already receive XML documents, and in these cases there is no need to use the client library. By leveraging the service binding, these XML handler assets can also be used outside of the WebSphere Commerce application and they still call the component façade.

### Client library

The client library provides Java-friendly libraries that build the service requests that the WebSphere Commerce service accepts. Clients of a WebSphere Commerce service are advised to use the client library to build the service request. The client library hides all the complexity of Web services requests, reducing the cost of development and maintenance of the client. Client libraries provide a mechanism for the client to easily switch between Web services and local Java calls depending on the deployment without needing to change the client code. Clients use this library to build service requests from a Java application, WebSphere Portal Server, or any other Web-enabled application. This client composes request Java objects and initiates the service communication.

## Section

# ***Component façade***



This section discusses the Component Façade.

## Component façade

- Independent of protocol
  - Receives and returns OAGIS messages
- Java class with supported service represented as methods on the class
  - Methods delegate object to a common processor
  - Processing of service request delegates which commands perform the business logic
- Deployed as an Enterprise Java Bean



A component facade is independent of the protocol. It is used to group a set of related business objects and provide support for the services associated with those objects. The methods on the component facades represent operations, also known as verbs, on WebSphere Commerce nouns. For example, the member component facade implements the Get verb on the Person noun. The component facade implementation is a Java interface with a one-to-many mapping of noun to interfaces. A noun should be implemented by only one facade interface. The methods on the facade interface match the Business Object Documents (BODs) supported for that noun.

The same component facade is used regardless of how the request enters the system. Each presentation layer must have the associated presentation framework and configuration in place to bind the component to that presentation. In the case of Web services, the Web services deployment descriptors and assets are required to enable the component for Web services. In the case of Web enablement for Struts or the WebSphere Commerce foundation tag library, the client library must have the required framework methods to integrate with the presentation layer and delegate the request to the component. All presentation-specific logic is contained at the presentation layer and only common logic that is independent of the presentation layer is part of the component facade.

As for the Java component facade implementation, the component consists of a Java bean that delegates to the business object document (BOD) processor to process the request. The BOD processor is a runtime environment that manages the request. This BOD processor performs the appropriate runtime setup and then delegates to commands to implement the request. The commands are part of the component facade and represent the business logic associated with the OAGIS request. The commands can call other commands and access or update the database depending on the request.

These services are independent of the component and should be decoupled from the business logic. After the processor takes control of the request, it enforces some of the quality of service requirements specified by the application.

## Business Object Document commands

- Each service request has a corresponding command
  - ▶ Get services follow the get design pattern to fetch and return data
  - ▶ Process, Change and Sync can use existing WebSphere Commerce features to re-use existing implementations
- Why commands?
  - ▶ Pluggable framework to change the implementation



The service command is the entry point on the component facade where the Business Object Document (BOD) begins to be processed by the business logic. The service command is a command that extends the WebSphere Application Server (not the WebSphere Commerce Server) command framework, and only the top-level service controller command needs to extend a specific implementation. The service controller command is an instance BOD command and should extend the abstract BOD commands. The service command breaks down the request into a series of business tasks and call the appropriate commands to complete these tasks.

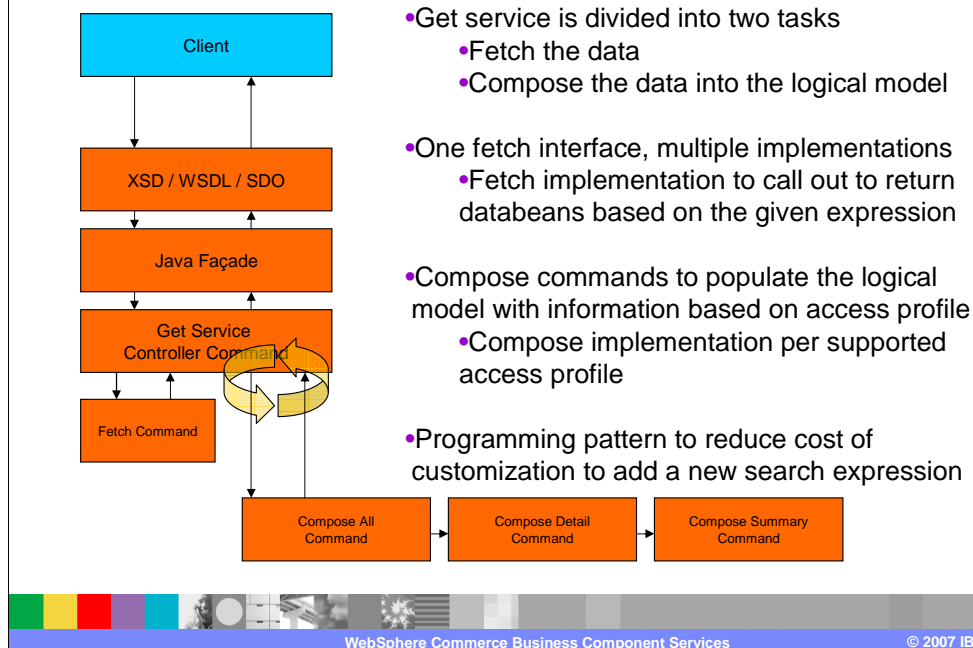
These business task commands are WebSphere Application Server commands and the only condition placed on the implementation is that it follows the WebSphere command pattern. No additional knowledge of WebSphere Commerce implementation or interfaces is required. For business task commands, the command only needs to extend the WebSphere command classes but uses the `com.ibm.commerce.foundation.server.commands.CommandFactory` class to instantiate the command implementation.

The command factory is used to create an instance of the command. The command factory determines through configuration the appropriate implementation for a particular command. The implementation instantiated can depend on a variety of information, but at a minimum it depends on the interface of the command to instantiate. Due to the common nature of how the requests are structured, the command factory can be used to abstract these tasks and determine the appropriate implementation. Although at a minimum the command factory requires the interface name of the command to instantiate, it does support accepting a `CommandKey` object to instantiate an implementation. The command key is a way to add an additional condition to determine whether the default task implementation should be used or, based on this particular condition, another implementation should be used. The default implementation is always used in the case where the condition does not correspond to an implementation.

For example, to support searching, the Get design pattern recommends separating the logic into a fetch task and a compose task. The implementation of each of these tasks depends not only on the interface name of the task but either an "XPath key" or "Access Profile" for the fetch and compose tasks. This condition allows new search expressions or access profiles to be added with minimal customization. Rather than extending an existing task to add support for this functionality, the condition is used to find the new implementation that is associated with this new behavior.

There are different command keys depending on the type of command to run. For the Fetch command, a Get command key is required to append the XPath expression as part of the key to retrieve the implementation. The same applies for specific action command keys to pick particular implementations based on more than just the command interface.

## Get request design pattern



WebSphere Commerce Business Component Services

© 2007 IBM Corporation

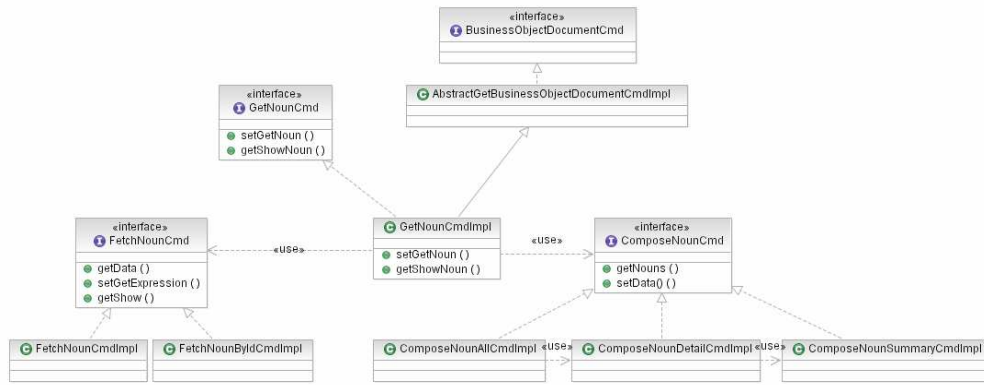
The Get service design pattern is used for retrieving and displaying information from Web services. It is divided into two tasks: fetch and compose. The Fetch command returns a list of data beans that matches the expression. The Get command then takes that list and for each data bean, it calls a Compose task to transform the data bean into the appropriate logical model (noun).

An access profile can be used to scope the response data. For example, to return a specific view of the data (for example a search view of a catalog entry), or to return the data in all languages for authoring purposes. As an extension to the XPath syntax, the access profile name-value pair should be added as a prefix to the XPath expression in curly brackets ({}). You must always specify the access profile. When the Get command calls the Compose command, it uses the access profile of the request as the key to select the appropriate Compose implementation. Because the access profile is just a superset of another access profile, the Compose command delegates to the parent access profile to first populate the logic model and add any required information. This results in a chain of Compose commands being called for the data bean, each populating a set of the logical model.

To customize the amount and type of data returned in the response, you can use different Compose tasks without changing the Fetch task by using a different access profile as the key. Supporting a new access profile consists of creating a new Compose command for that access profile and registering the implementation. Also, this chaining pattern ensures that customization of the compose command at a particular access profile is reflected in all dependent access profiles. If the customization adds additional data at the summary access profile, then all child access profiles also include this summary data. If you override one command, all dependent code gets this new behavior.



## Get request design pattern



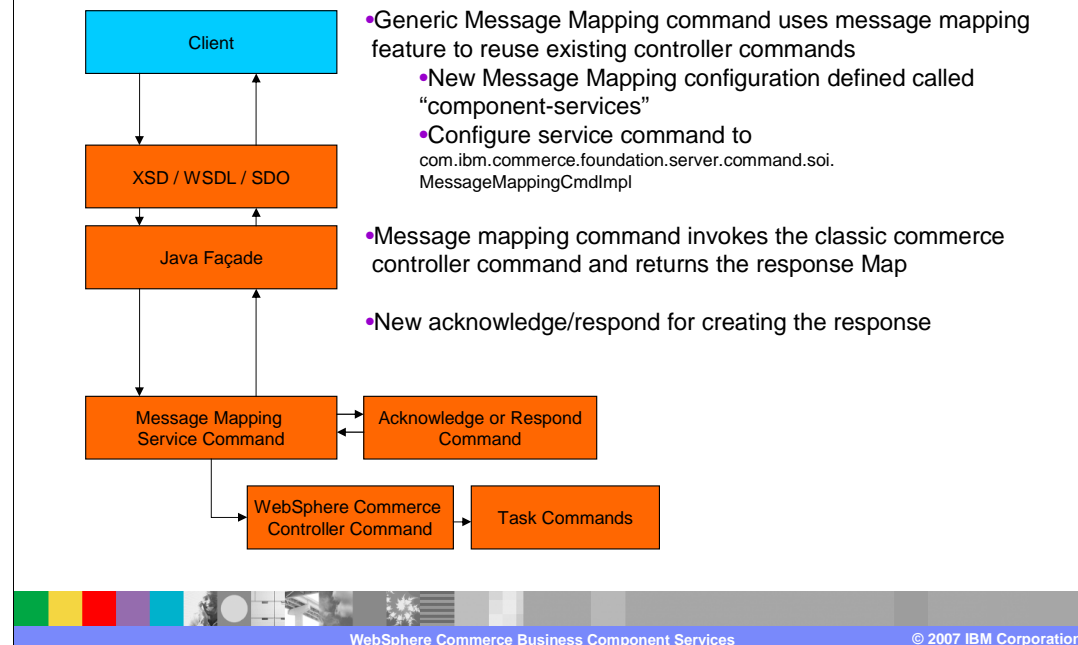
WebSphere Commerce Business Component Services

© 2007 IBM Corporation

The class diagram below demonstrates the classes involved to create a Get service. One master controller command is created for a Get service. This command, using the `com.ibm.commerce.foundation.server.util.oagis.SelectionCriteriaMapper` utility class, extracts the Get information from the request BOD and breaks the work into two tasks, delegating to the command framework to run a Fetch command and a Compose command. The Fetch command fetches the data, while the Compose command composes the response.

The main principle behind this design pattern is customization. This pattern promotes reuse of the Fetch command, so supporting a new search expression is a matter of implementing a new Fetch command to return a list of populated databases that represent that new expression. By using the XPath and command configuration, you can associate one XPath expression with one command implementation, or multiple XPath expressions with the same command implementation. You do not need to modify the Compose tasks just to support a new search expression.

## Process or Change request



WebSphere Commerce contains many controller commands that can be exposed as services. The Process and Change BOD command is just a command that converts the existing BOD request into a set of name-value pairs and delegates to an existing WebSphere Commerce controller command. However, the existing message mapping feature of WebSphere Commerce provides this feature and already addresses some of the additional customization aspects of extending the noun and adding more information. You need to customize only the controller command where the name-value pairs exist. You do not need to write new code to read the BOD request.

In order to do this, the service request is mapped to a generic message-mapping service command called `MessageMappingCmdImpl`. This generic command uses the message-mapping configuration defined under the name "component-services" to convert the SDO into a set of name-value pairs and a WebSphere Commerce command implementation. Then it implements the command and responds to the request.

In the message mapping configuration, this generic command looks for two control attributes to determine the code to call to generate the response. The attributes are `responseCommand` and `errorCommand`. The generic command uses the specified implementation to generate the response to the request. This implementation is the code hook to generate the success or error response. These response generators must follow the `MessageMappingResponseCmd` interface.

## Component façade customization

- Add new search expression
  - ▶ Add new Fetch command to support the next expression
- Add more data to Business Object
  - ▶ Add information to UserData elements or use the Overlay approach
  - ▶ Extend Compose to handle additional data
- Add business logic
  - ▶ Add new command to implement the business logic
  - ▶ Update business logic by extending existing commands



There are three ways to customize the component façade. You can add a new search expression, add more data to the Business Object or update the business logic.

To add a new search expression, you should know that search expressions performed by WebSphere Commerce services use an expression language known as XML Path Language, also called XPath. If you are new to XPath, you must familiarize yourself with the notation before proceeding with this customization task. WebSphere Commerce has also extended the notation for XPath to allow control over the display, ordering of the results and selection of business logic command implementations.

There are two methods for adding more data to the noun. The simplest method is to leverage the UserData extension points that the component nouns provide. The alternative is to take advantage of the overlay methodology. Overlay extensions allow users to have their extensions appear within the OAGIS or WebSphere Commerce complex types. In order to add elements, a user must extend the OAGIS or WebSphere Commerce complex types within their own namespace. By doing this, it is possible for users to enforce additional restrictions or add additional elements to the OAGIS or WebSphere Commerce complex types.

To update the business logic, existing WebSphere Commerce patterns are used where you register a new command implementation for a particular interface that implements you extended business logic

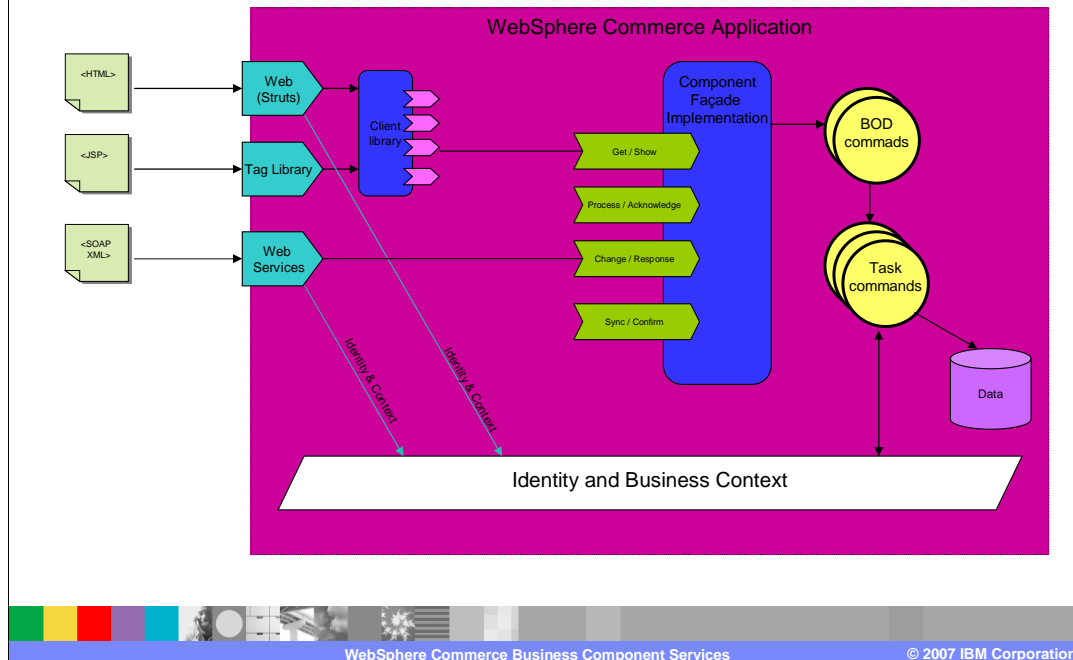
## Section

# ***Service binding***



This section discusses the WebSphere Commerce Service Binding.

## WebSphere Commerce service binding



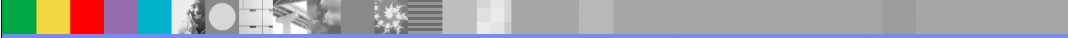
The service binding resides between the client library and the services. It provides the transport mechanism to pass data, using service data objects (SDOs), between the client and the service. This transport mechanism can be Web services or local Java binding. The Web service connects the client to the component facade implementation remotely. When the client and component facade implementation are deployed in separate applications, the client sends the service request using Web services as the transport layer. Because the client and component facade implementation are distinct applications, the Web service security is used for authorization. The local enterprise bean connects the client to the component facade implementation in the local JVM. When the client and component facade implementation are deployed within the same application, the client should communicate with the component facade implementation through a local EJB call. The client uses the component facade's local enterprise bean to invoke the method that matches the intended service.

IBM Software Group

IBM

Section

***Client library***



WebSphere Commerce Business Component Services

© 2007 IBM Corporation

This section discusses the client library.

## Client library

- Java utility class
  - ▶ Builds up request service objects and sends the request
  - ▶ Does not contain business logic
  - ▶ Provides programmer friendly convenience methods for Java applications
  - ▶ Provides Web enabled methods for Web applications
  - ▶ Deployed in any Java application



Client library for WebSphere Commerce services' primary purpose is to simplify and eliminate code on the client. The client library is essentially a Java layer to help Java applications integrate with your service architecture, with no additional code generation required. The client library already has support for session and authentication and provides Java-based clients a standardized mechanism to create the OAGIS SDO objects to represent service requests.

Each component provides a client library project to access the component. The client project contains the following two assets:

1. A constant file that is sharable between client and server
2. A client package that contains
  1. The abstract client Java class that contains common methods and methods required by the foundation framework
  2. A programming friendly/Web enabled Java class that extends from the abstract class to implement the Java friendly and Web enabled methods
  3. Noun specific exceptions to represent client errors and server errors

The subclass of the abstract client library class is a programmer friendly and Web enablement client library class. The programmer friendly client provides fine-grain methods to perform specific service actions. These methods build up the appropriate service request SDO and call the methods offered by the abstract client library class to perform the service request. The programmer friendly library should only build up the service request object. It must not send the message or contain any additional business logic. The responsibility of sending is part of the abstract client library class. The additional business logic is the responsibility of the business task command using the client library.

# Remote client deployment flow

The diagram illustrates the remote client deployment flow between a WebSphere Portal Server and a WebSphere Commerce Server.

**WebSphere Portal Server:**

- Contains a **Portlet** and a **Commerce Portlet**.
- The **Portlet** sets business context information to the client library.
- The **Commerce Portlet** is connected to the **Client library**.
- The **Client library** is secured by SSL or Firewall, which flows the identity information.
- Business Context Data is supplied by the caller.
- Authentication is passed as a wrapper with the business context from the client library (javax.security.auth.CallbackHandler).

**WebSphere Commerce Server:**

- Contains **Component Façade Implementation** blocks.
- The **Client library** sends requests to the **Component Façade Implementation** blocks.
- The **Component Façade Implementation** blocks return **commands**.

In the Struts framework, a browser request is routed to a servlet that acts as a controller. Using local Java calls, the controller calls the model for processing. The controller then dispatches the appropriate view to render data. The model encapsulates all business logic (implemented by following the command pattern) and data (implemented using JSP™ pages). The JSP pages retrieve data from the database using data beans, and then format the output.

WCSv602\_Business\_Component\_Services.ppt



## Web enablement

- Client library
  - ▶ Method that takes java.util.Map as input and java.util.Map as output
  - ▶ Input map contains URL parameters and String array of parameter values
- Web framework
  - ▶ Maps URL to a client library map based method
  - ▶ Invokes the method to run the service
  - ▶ Client look-up service to globally change client implementation

```
<action
  parameter="com.ibm.commerce.catalog.facade.client.CatalogFacadeClient"
  path="/ChangeProduct"
  type="com.ibm.commerce.struts.ComponentServiceAction">
  <set-property property="clientMethod" value="changeProduct" />
</action>
```

The guideline of how to enable the client library for the Web, is that each action should be represented as a separate method. The Web client library has a corresponding method for each Web URL request for that component. For example in the case of a shopping cart, there are URL requests to add an item, update an item and remove an item. These three actions should be represented as different methods on the client library. The map input method should call out to a protected build method to convert the Map input into the structured object. The map contains an array of strings and the key is the parameter name. A developer can generically update the set of name-value pairs that makes up a particular complex type of the request and not have to modify multiple methods. The map-based method must delegate to these methods to build up the complex type and piece together the complex objects to build the service request. This allows for only one code touch point when the customization wants to support additional name-value pairs instead of having to extend multiple methods. If a programming-friendly method exists that builds up the same message, the method delegates to that method.

The Struts presentation framework uses reflection to delegate the URL request parameters to the appropriate client library Web method. Struts has as part of its configuration a mapping between the URL request and the client library class and method to implement. The Struts framework also has a lookup service Struts plug-in that returns the right implementation for a specified client library. Since the mapping of client library class to URL can be rather lengthy, and more URLs can be added during upgrades, there is a single point of configuration that helps to globally control the class implementation. This way you can use the customized client library for all references of a particular client library without have to change every URL reference to that library. This client implementation lookup service provides that single point of custom configuration. The client libraries for the provided components: member, order, contract, catalog are already enabled for the Web.

## getData tag library

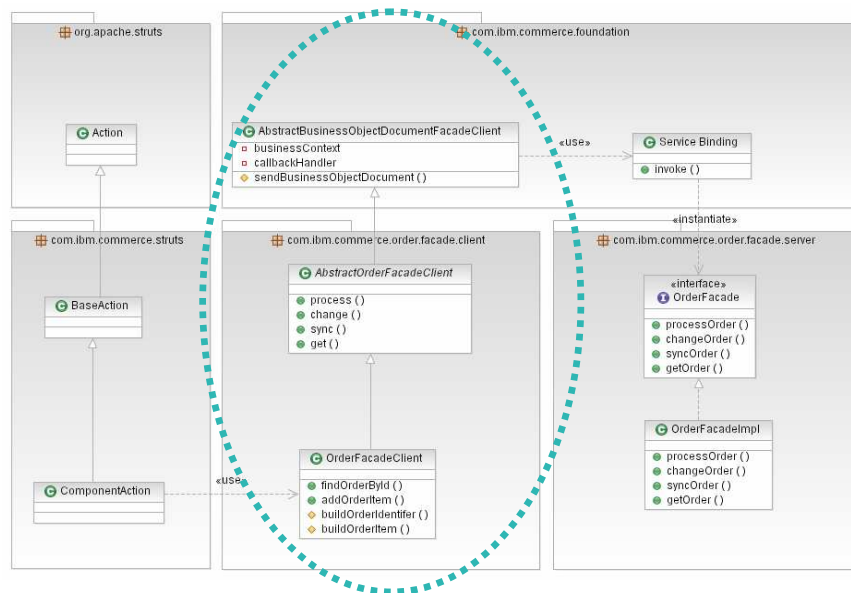
- Associates service data objects with a new scripting variable
- Retrieves SDO noun objects for use on a JSP
- Tag handler delegates to a configured client façade
- Accepts an instance of GetType and returns an instance of ShowDataAreaType
- JSP writer can compose the expression or delegate it to a configured expression builder
- JSP writer can override business context values for a request

```
<wcf:getData type="com.ibm.commerce.order.facade.datatypes.OrderType"
  var="order" expressionBuilder="findCurrentShoppingCart">
  <wcf:contextData name="storeId" data="10001" />
  <wcf:param name="accessProfile" value="WC_OrderDetailsProfile" />
</wcf:getData>
```

The `getData` tag retrieves service data objects from a WebSphere Commerce service and associates them with a newly declared scripting variable with a given ID. The `getData` tag also retrieves noun objects for use on specific JSP's. The `getData` tag handler delegates to a configured client façade whose method must accept an instance of `GetType` and return an instance of `ShowDataAreaType`. JSP writer can compose the Get expression or request that the expression composition be delegated to a configured expression builder and just pass in the name value parameters.

The XML snippet above gets the order object for the current shopping cart and assigns it to a variable called "order". It is important to note that the `param` tag and the `contextData` are sub-tags of the `getData` tag.

## Client library development



From a development perspective, there is an abstract business object document façade client that is created which should be used by all client APIs. Within this abstract implementation there is a generic send operation. This generic send operation uses the service binding concept to look at the configuration and determine how it should communicate with the service implementation either by using a Web service or a local EJB.

## Client library example

```
// =====
// Programmer friendly APIs that can be used by clients to perform
// more specific operations. These APIs is a convenient way to perform
// common operations clients of the SubsystemGroupName facade would perform.
// =====

protected NounNameType buildNounName(java.util.Map parameters) {
    NounNameType nounName = SubsystemGroupNameFactory.eINSTANCE.createNounNameType();
    String[] description = (String[]) parameters.get("description");
    if (description != null) {
        nounName.setDescription(description[0]);
    }
    return nounName;
}

protected java.util.Map buildResponse(AcknowledgeNounNameDataAreaType dataArea) {
    java.util.Map result = new java.util.HashMap();
    // populate response map ...
    return result;
}

public java.util.Map registerNounName(java.util.Map parameters) throws NounNameException {
    AcknowledgeNounNameType ack = super.processNounName(createProcessNounName("Register",
    buildNounName(parameters)));
    AcknowledgeNounNameDataAreaType dataArea = checkAcknowledgeNounName(ack.getDataArea());
    return buildResponse(dataArea);
}
```

An example of a Web enabled method is a `registerNounName` which takes a `Map` as input and returns a `Map` as output. The purpose of `registerNounName` is to build a Process Business Object Document with an action of Register and a noun that is a combination of the name-value pair found in the parameter map.

As part of the programming pattern, all the build type operations have been carved out and placed into protected methods such that extension is easier. To extend the build method you would extend the Java façade client and override the build method which will enable you to handle all the additional parameter data specified. You would never use your client to implement business logic. The whole purpose of the client is to build the request object and handle the response. The most it should have is parameter validation before it makes the service request.

## Client library deployment

- Configuration found in classpath
  - ▶ config/<component-id>/wc-component-client.xml
- Configuration types
  - ▶ Local Enterprise Java Bean
  - ▶ Generated Web service proxy with SOAP element binding
  - ▶ J2SE™ Web service

```
<_wcf:DevelopmentClientConfiguration xmlns:_wcf="http://www.ibm.com/xmlnl/prod/commerce/9/foundation"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.ibm.com/xmlnl/prod/commerce/9/foundation ../xsd/wc-component-client.xsd">
  <_wcf:InvocationService>
  <_wcf:InvocationBinding
    bindingImpl="com.ibm.commerce.foundation.internal.client.services.invocation.impl.JAXRPCWebServiceProxyInvocationBi
    ndingImpl">
  <_wcf:Property name="proxyClass" value="com.companyname.www.SubsystemGroupNameServicesPortTypeProxy" />
  </_wcf:InvocationBinding>
  </_wcf:InvocationService>
</_wcf:DevelopmentClientConfiguration>
```

The client library for a WebSphere Commerce service can be deployed in one of three possible environments:

- 1.The WebSphere J2EE™ environment where the generated JAX-RPC client proxy code is required to leverage the Web services capabilities provided by WebSphere Application Server,
- 2.A local J2EE environment using local EJB components, or
- 3.A J2SE environment for testing purposes

## Client customization

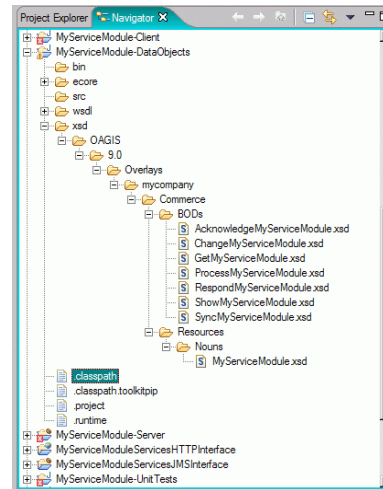
- Extend existing client
  - ▶ Add new service support by adding new methods
  - ▶ Add more data by overriding build methods which convert `java.util.Map` input into the Service Request object
  - ▶ Registering extended client implementation instead of the default client implementation



To extend an existing client you can either add more data to the current service request, handle new data in a service response, or add new behavior by building an entirely new service request.

## Developing WebSphere Commerce services

- Client library
  - ▶ Java code to build OAGIS style request and process OAGIS style response
  - ▶ Contains methods to integrate with presentation framework
- Data objects
  - ▶ Generated SDO from XML schema
- Component façade
  - ▶ Services are implemented as commands
  - ▶ Calls common processor to perform services associated with processing



The WebSphere Commerce service module contains all the assets used by the WebSphere Commerce service. If you are creating new WebSphere Commerce services, you need to create a new service module for each logical grouping of nouns you plan to support. The Design Pattern Toolkit and the ComponentProjects design pattern generate the base code for the new service module from a simple XML file. The service modules can be generated by describing the service module in a specialized XML syntax. This allows you to start directly with the service module implementation without having to spend hours with the setup and configuration of a service module.

After you generate a new service module, it contains the following projects:

The **data objects project** contains the XSDs, the WSDL, the EMF generated model, and the generated SDO code. The purpose of the data objects project is to contain the data objects that make up the service. The data objects project should refer only to the Foundation-DataObjects.jar and should include and extend only objects contained in that project. The **client library project** is a set of APIs that are exposable to any client, regardless of whether that client is WebSphere Commerce or some other application. The **component facade project** contains the WebSphere Commerce controller commands that implement the services along with other commands to provide the tasks associated with the service implementation.


After the pattern is applied, there is one JUnit Test class that is created to send and receive empty service messages. After applying the pattern and adding the projects to the WebSphere Commerce application, the Test case should run successfully where empty Services messages are sent and received. These projects contain the transport enablement of the service module for HTTP and JMS. The component facade project uses these projects to enable Web service support for both transports.

IBM Software Group

IBM

Section

***Design Pattern Toolkit***



WebSphere Commerce Business Component Services

© 2007 IBM Corporation

This section discusses the Design Pattern Toolkit.



## Design Pattern Toolkit

- Code generation pattern based on Design Pattern Toolkit (DPTK)
  - ▶ <http://www.alphaworks.ibm.com/tech/dptk>
- Improves development experience by automating repeatable steps to build a service
- Generates projects and assets for developing a component
  - ▶ Data objects project with XSD and EMF files
  - ▶ Client project with known required methods implemented
  - ▶ Server EJB project with deployment descriptors and template command implementation to start developing
  - ▶ HTTP and JMS Web service enablement projects
  - ▶ Unit test project for JUnit test cases

```
<commerce-component
  name="SubsystemGroupName"
  company="CompanyName"
  packageprefix="com.mycompany.commerce"
  namespace="http://www.mycompany.com/xmlns/prod/commerce/9"
  nlsprefix="myco" type="SOI">

  <noun
    name="NounName"
    get="true" process="true" change="true" sync="true"/>

</commerce-component>
```

The Design Pattern Toolkit (DPTK) is an Eclipse-enabled template engine for generating applications based on customizable, model-driven architecture transformations. WebSphere Commerce uses the DPTK plug-in for creating WebSphere Commerce service modules from a simple XML file. By describing the service module in a specialized XML syntax, the service modules can be generated. This allows you to start directly with the service module implementation without having to spend hours with the setup and configuration of a service module.

## Next steps after pattern is applied

- Define the Noun
- Generate the SDO
- Implement the service
  - ▶ Implementing the starter commands created
- Implement client methods
  - ▶ Web enablement methods
  - ▶ Get data tag
  - ▶ Programmer friendly Java methods



After the pattern is applied you need to perform the following steps:

1. Define your noun – you will receive a starter XSD with no content in the noun so that you decide what your business object will look like for that service.
2. Generate your SDO with the Rational® tools.
3. Implement the business logic - as part of the pattern generator you receive some of the skeleton commands which need to be implemented.
4. Implement the client method to integrate with the Portal infrastructure within the tag library.

## References

- WebSphere Commerce Feature Pack 2

- ▶ <http://publib.boulder.ibm.com/infocenter/wchelp/v6r0m0/index.jsp?topic=/com.ibm.commerce.base.doc/concepts/covwhatsnewinthisrelease.htm>

- Service Data Objects

- ▶ <http://www-128.ibm.com/developerworks/java/library/j-sdo/>

- Design Pattern Toolkit

- ▶ <http://www.alphaworks.ibm.com/tech/dptk>



For more information regarding WebSphere Commerce Feature Pack 2, Service Data Objects or the Design Pattern Toolkit, visit the sites indicated in the presentation.

## Feedback

### Your feedback is valuable

You can help improve the quality of IBM Education Assistant content to better meet your needs by providing feedback.

- Did you find this module useful?
- Did it help you solve a problem or answer a question?
- Do you have suggestions for improvements?

[Click to send e-mail feedback](#)



You can help improve the quality of IBM Education Assistant content by providing feedback.

# Trademarks, copyrights, and disclaimers

The following terms are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both:

|                 |                        |          |          |           |
|-----------------|------------------------|----------|----------|-----------|
| IBM             | CICS                   | IMS      | MQSeries | Tivoli    |
| IBM (logo)      | Cloudscape             | Informix | OS/390   | WebSphere |
| e[logo]business | DB2                    | Series   | OS/400   | xSeries   |
| AIX             | DB2 Universal Database | Lotus    | pSeries  | zSeries   |

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are registered trademarks of Microsoft Corporation in the United States, other countries, or both.

Intel, ActionMedia, LANDesk, MMX, Pentium and ProShare are trademarks of Intel Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Linux is a registered trademark of Linus Torvalds.

Other company, product and service names may be trademarks or service marks of others.

Product data has been reviewed for accuracy as of the date of initial publication. Product data is subject to change without notice. This document could include technical inaccuracies or typographical errors. IBM may make improvements and/or changes in the product(s) and/or program(s) described herein at any time without notice. Any statements regarding IBM's future direction and intent are subject to change or withdrawal without notice, and represent goals and objectives only. References in this document to IBM products, programs, or services does not imply that IBM intends to make such products, programs or services available in all countries in which IBM operates or does business. Any reference to an IBM Program Product in this document is not intended to state or imply that only that program product may be used. Any functionally equivalent program, that does not infringe IBM's intellectual property rights, may be used instead.

Information is provided "AS IS" without warranty of any kind. THE INFORMATION PROVIDED IN THIS DOCUMENT IS DISTRIBUTED "AS IS" WITHOUT ANY WARRANTY, EITHER EXPRESS OR IMPLIED. IBM EXPRESSLY DISCLAIMS ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT. IBM shall have no responsibility to update this information. IBM products are warranted, if at all, according to the terms and conditions of the agreements (e.g., IBM Customer Agreement, Statement of Limited Warranty, International Program License Agreement, etc.) under which they are provided. Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products in connection with this publication and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. IBM makes no representations or warranties, express or implied, regarding non-IBM products and services.

The provision of the information contained herein is not intended to, and does not, grant any right or license under any IBM patents or copyrights. Inquiries regarding patent or copyright licenses should be made, in writing, to:

IBM Director of Licensing  
IBM Corporation  
North Castle Drive  
Armonk, NY 10504-1785  
U.S.A.

Performance is based on measurements and projections using standard IBM benchmarks in a controlled environment. All customer examples described are presented as illustrations of how those customers have used IBM products and the results they may have achieved. The actual throughput or performance that any user will experience will vary depending upon considerations such as the amount of multiprogramming in the user's job stream, the I/O configuration, the storage configuration, and the workload processed. Therefore, no assurance can be given that an individual user will achieve throughput or performance improvements equivalent to the ratios stated here.

© Copyright International Business Machines Corporation 2007. All rights reserved.

Note to U.S. Government Users - Documentation related to restricted rights-Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract and IBM Corp.

