Service Based Objects (SBO's) in Documentum

Documentum Business Object Framework which was introduced from Documentum 5.3 plays a key role in most of the current Documentum implementations. Service based Object is one of the important member of Documentum BOF family. Lets try to see what makes Service Based Objects very popular and how can you implement it.

What is a SBO

In simple terms SBO in Documentum can be compared to session beans of J2EE environment. SBO enable the developers to concentrate just on the business logic and all the other aspects will be managed for you by the server. This reduces the application code significantly and reduces lots of complexities. The biggest advantage of a BOF that its deployed in a central repository. The repository maintains this module and DFC ensures that he latest version of the code is delivered to the client automatically.

Service Based Objects are repository and object type in-depended that means the Same SBO can be used by multiple Documentum repositories and can It can retrieve and do operations on different object types. SBO's can also access external resources for example a Mail server or a LDAP server. Prior to the introduction of Documentum Foundation Services SBO's were commonly used exposed to expose documentum web services.

An SBO can call another SBO or by any Type based Objects. (Type Based Objects (TBO) are a different kind of Business Object types which I will explain in a different study note)

A very simple to understand example for a SBO implementation would be a Zip code Validator. Multiple object types might have Zip code across multiple repositories. So if this functionality is exposed as a SBO it can be used by the custom application irrespective of Object types and repositories. This Validator SBO can be used even by different TBO's for validations.

Here are some bullet points about SBO's for easy remembering

- SBO's are part of Documentum Business Object framework
- SBO's are not associated with any repositories
- SBO's are not associated with any Documentum object types.
- SBO information is stored in repositories designated as Global Registry.
- SBO's are stored in /System/Modules/SBO/<sbo_name> folder of repository. <sbo_name> is the name of SBO.
- Each folder in /System/Modules/SBO/ corresponds to a individual SBO

How to implement a SBO using Composer

The steps to create a SBO are these.

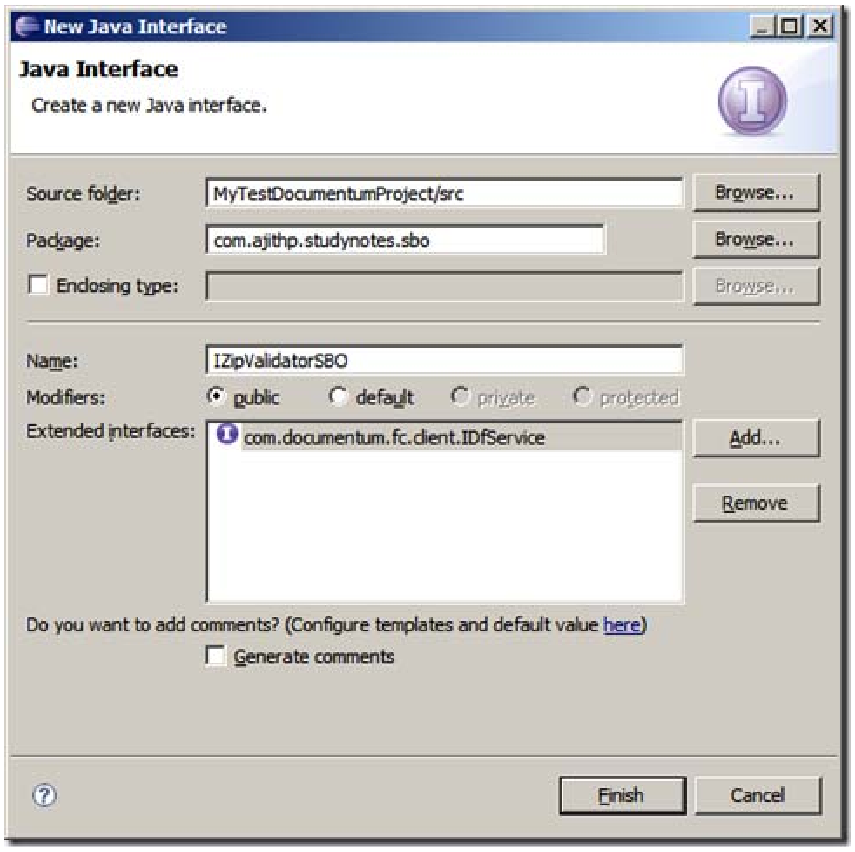
1) Create a interface that extends **IDfService** define your business method
2) Create the implementation class implement write your business logic, This class should extend **DfService** and implement the interface defined in Step 1
3) Create a jar file for the created Interface and another jar for the implementation class then create Jar Definitions
4) Create a SBO Module and Deploy your Documentum Archive using Documentum Composer (Application builder for older versions)

Lets see these steps with an Example SBO Zip Code Setter, I am not covering the steps using application builder here. The screenshots and the notes will give you an insight about how to use Documentum Composer to implement a Service Based Object in Documentum version 6 or above.

Step 1 : Create an interface and define your Business method

The first step is to create a interface which will define the business functionality. This interface should extend **IDfService** interface. Client application will use this interface to instantiate the SBO.

Click **New –> Interface** in Documentum Composer. Click on the Add button of Extended Interfaces and search for IDfService. Select IDfService and click OK



Source folder: MyTestDocumentumProject/src

Package: com.ajithp.studynotes.sbo

[ ] Enclosing type:

Name: IZipValidatorSBO

Modifiers: [✓ public] default private protected

Extended interfaces: com.documentum.fc.client.IDfService

Do you want to add comments? (Configure templates and default value here)

[ ] Generate comments

Now Add the Business method ValidateZipCode() to interface. The code should look like the following.

```
package com.ajithp.studynotes.sbo;


import com.documentum.fc.client.IDfService;

import com.documentum.fc.client.IDfSysObject;

import com.documentum.fc.common.DfException;


public interface IZipValidatorSBO extends IDfService {


 public void validateZipCode (IDfSysObject obj, String zipCode, String repository)throws DfException;


}
```

Step 2 : Create the implementation class

All the Service Based Object implementation classes should extend from **DfService** class and implement the Interface created in the first step. ***DfService class is an abstract class There are few methods which were abstract in 5.3 and has provided with a default implementation in 6.0 and later***

Method Name	Returns	More information
getVendorString()	String	This method's default implementation returns a empty String. Override to make changes to it.

getVersion()	String	This method returns a version which is not right, Override this method to return Major.minor version.
isCompatible()	boolean	The default implementation returns true if the version is an exact match

Lets see some other important methods of DfService Class before we move further.

Method Name	Returns	More information
getName()	String	This returns the fully qualified logical name of the service interface
getSession()	IDfSession	This method returns IDfsession Object for the docbase name which is passed as argument to this method. You have to make sure that you call releaseSession() after you are done with the operation that involves session.
releaseSession()		Releases the handle to the session reference passed to this method.
getSessionManager ()	IDfSessionManager	Returns the session manager.

Managing repository sessions in SBO

As We saw the the previous table its always good practice to release the repository session as soon as you are done with its use.

So the ideal way to do this should be like this.

```
// Get the session

IDfSession session = getSession(repoNam);

try {

// do the operation with session

} catch (Exception e){

// Process the exception

}finally {

// release the session

releaseSession(session)

}
```

Transactions in SBO

Another important thing is to know is how to handle transactions in SBO. Note that only session manager transactions can be used in a SBO. System will throw an Exception when a session based transaction used within a SBO.

beginTransaction() will start a new Transaction and use **commitTransaction()** to commit it or **abortTransaction()** to abort a transaction. Always ensure that you are not beginning a transaction where another transaction is active. You can use **isTransactionActive()** to find out whether a transaction is active or not.

Another important point is if your SBO doesn't start a transaction don't commit it or abort it in the SBO Code instead if you want to abort the transaction use **setTransactionRollbackOnly()** method.

Other important points

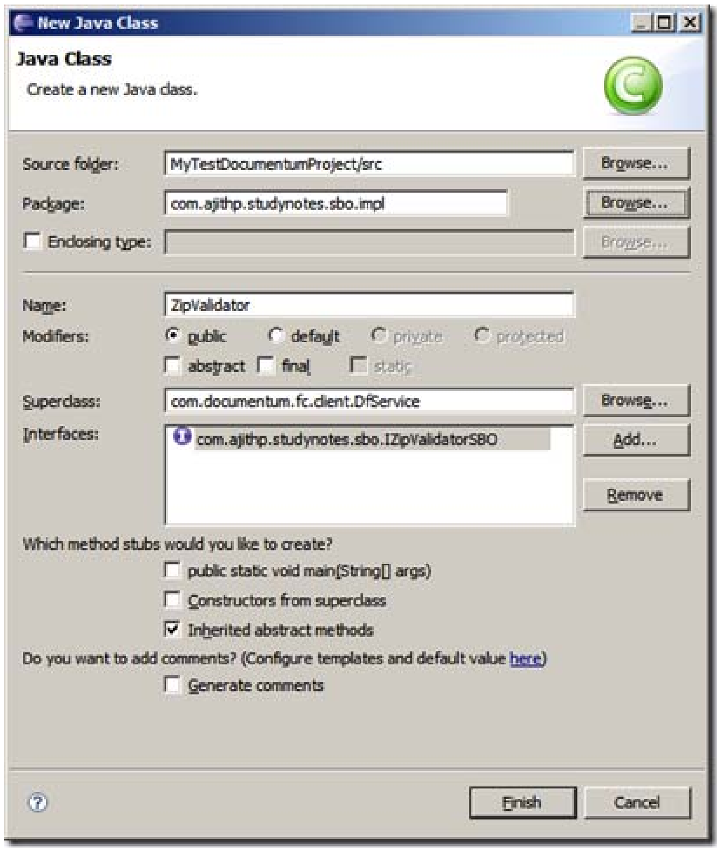
1) Since SBO’s are repository independed do not hard code the repository names in the methods. Either pass the repository name as method parameter or have it as a variable in SBO and use a setter method to populate it after instantiating

2) Always try to make SBO’s stateless (Its a pain to manage state full SBO’s).

3) Don’t reuse SBO, Always create a new instance before a operation.

Now lets see how to code our ZipSetterSBO

Click on **New –> Class**, Click on the Browse button of Superclass and Search and Select DfService and in the Interfaces search for the Interface created in the previous step and Click OK. Also select the option *Inherited Abstract Methods* in Which method stubs would you like to create.


New Java Class

Java Class

Create a new Java class.

Source folder: MyTestDocumentumProject/src

Package: com.ajithp.studynotes.sbo.impl

Enclosing type: [ ]

Name: ZipValidator

Modifiers: [✓ public] default private protected

[ ] abstract [ ] final [ ] static

Superclass: com.documentum.fc.client.DfService

Interfaces: com.ajithp.studynotes.sbo.IZipValidatorSBO

Which method stubs would you like to create?

[ ] public static void main(String[] args)

[ ] Constructors from superclass

[✓] Inherited abstract methods

Do you want to add comments? (Configure templates and default value here)

[ ] Generate comments

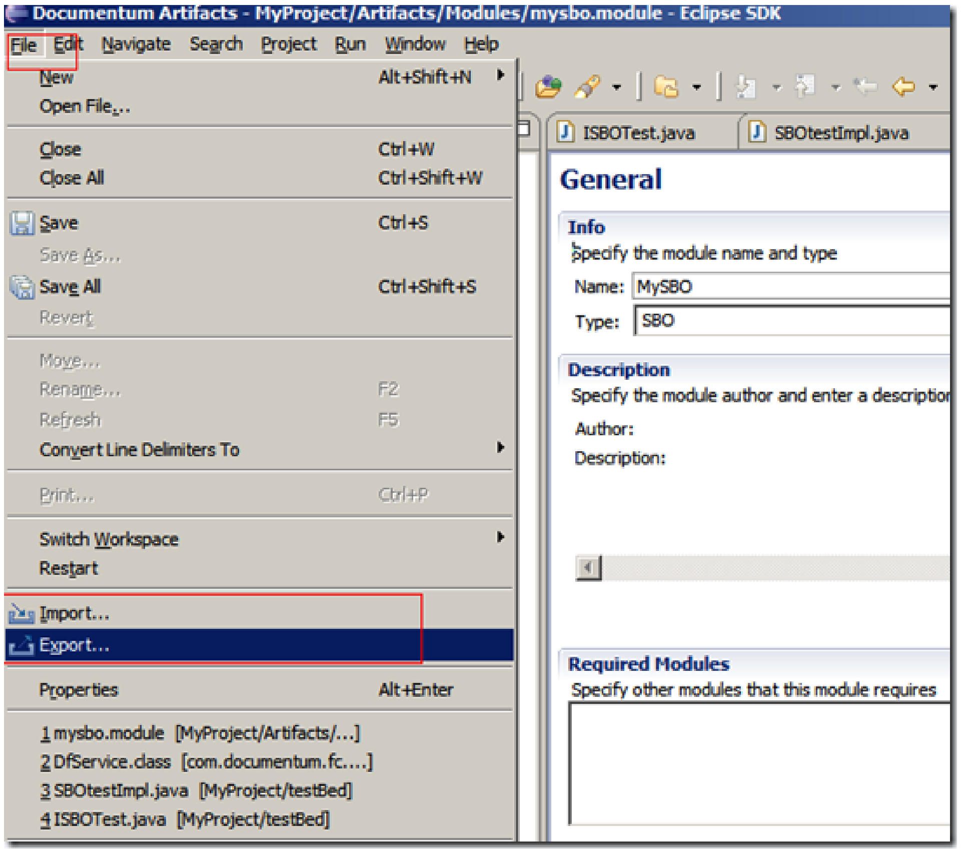
I had Overriden method getVersion() for the illustration purpose. See the code sample for the inline comments.

```
package com.ajithp.studynotes.sbo.impl;


import com.ajithp.studynotes.sbo.IZipValidatorSBO;

import com.documentum.fc.client.DfService;
import com.documentum.fc.client.IDfSession;

import com.documentum.fc.client.IDfSysObject;

import com.documentum.fc.common.DfException;

public class ZipValidator extends DfService implements IZipValidatorSBO {


public static final String versionString = "1.0";

// overriding the default

public String getVersion() {

return versionString ;

  }


public void validateZipCode (IDfSysObject obj, String zipCode, String repository) throws DfException {

     IDfSession session = getSession(repository);


     try {

     if (isValidUSZipcode(zipCode)){


         obj.setString("zipcode",zipCode);

         obj.save();

 }

     } catch (Exception e){

 /* Assuming that transaction is handled outside the code and this says DFC to abort the transaction

  in case of any error */

 getSessionManager().setTransactionRollbackOnly();

 throw new DfException();

     } finally {

 releaseSession(session);

    }


  }

private boolean isValidUSZipcode(String zipCode){

// implement your logic to validate zipcode.

// or even call a external webservice to do that

 // returning true for all zip codes

 return true;


  }
}
```

Step 3 : Generate Jar files and Create Jar Definitions

The next step in SBO creation is to create Jar files which will hold the interface and the implementation classes. These jar files are required to deploy your SBO.

Use Composers/Eclipse Create Jar option or command line jar command to create the jar file


Documentum Artifacts - MyProject/Artifacts/Modules/mysbo.module - Eclipse SDK
File Edit Navigate Search Project Run Window Help
New Alt+Shift+N
Open File...
Close Ctrl+W
Close All Ctrl+Shift+W
Save Ctrl+S
Save As...
Save All Ctrl+Shift+S
Revert
Move...
Rename... F2
Refresh F5
Convert Line Delimiters To
Print... Ctrl+P
Switch Workspace
Restart
Import...
Export...
Properties Alt+Enter
1 mysbo.module [MyProject/Artifacts/...]
2 DfService.class [com.documentum.fc....]
3 SBOtestImpl.java [MyProject/testBed]
4 ISBOTest.java [MyProject/testBed]
ISBOTest.java
SBOtestImpl.java
General
Info
Description
Required Modules

Specify the module name and type

Name: MySBO

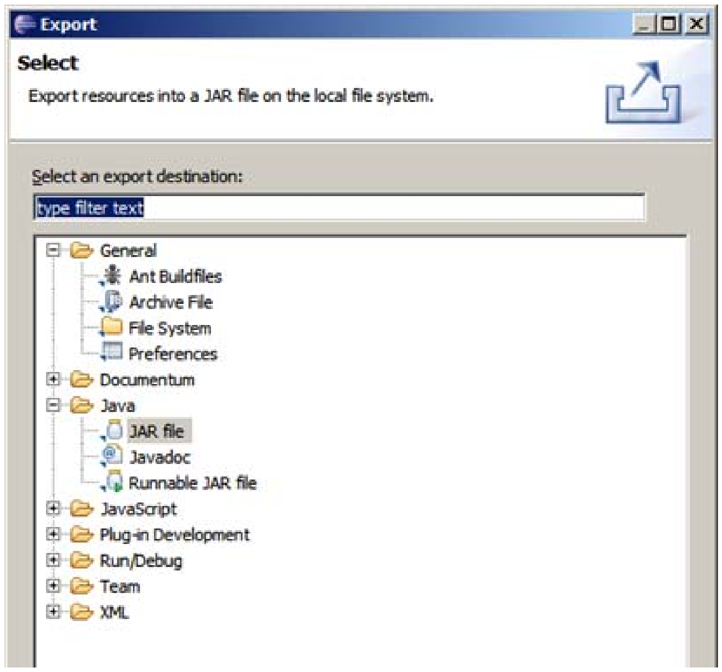
Type: SBO

Specify the module author and enter a descriptio

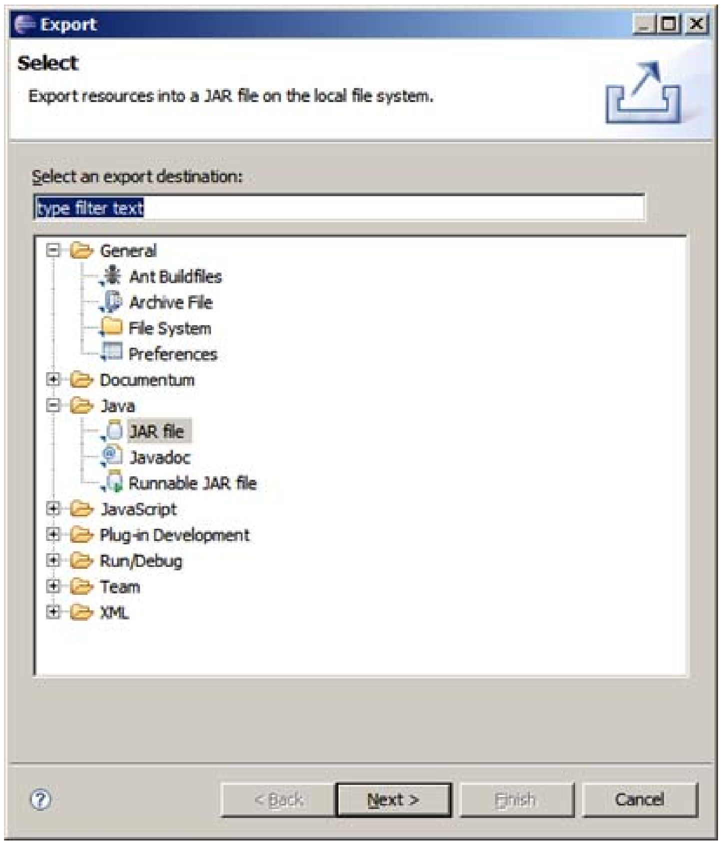
Author:

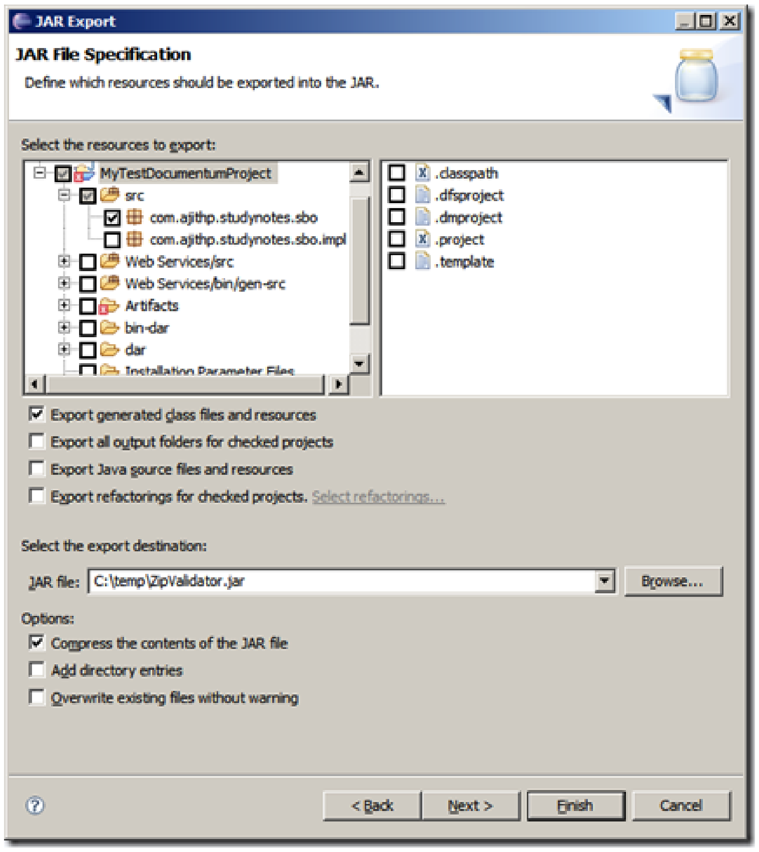
Description:

Specify other modules that this module requires



Export resources into a JAR file on the local file system.





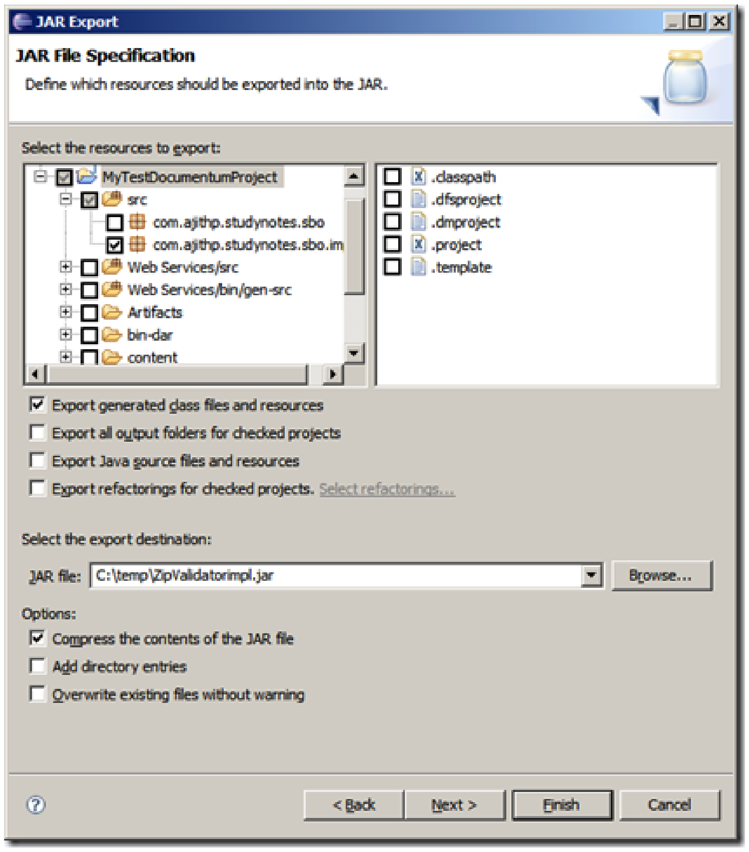
[✓] Export generated class files and resources
[ ] Export all output folders for checked projects
[ ] Export Java source files and resources
[ ] Export refactorings for checked projects. Select refactorings...

Select the export destination:

JAR file: C:\temp\ZipValidator.jar

Options:
[✓] Compress the contents of the JAR file
[ ] Add directory entries
[ ] Overwrite existing files without warning

Selecting the sbo package to create the interface jar


JAR Export

JAR File Specification

Define which resources should be exported into the JAR.

Select the resources to export:

- [✓] MyTestDocumentumProject
  - [✓] src
    - [ ] com.ajithp.studynotes.sbo
    - [✓] com.ajithp.studynotes.sbo.im
  - [ ] Web Services/src
  - [ ] Web Services/bin/gen-src
  - [ ] Artifacts
  - [ ] bin-dar
  - [ ] content

- [ ] .classpath
- [ ] .dfsproject
- [ ] .dmproject
- [ ] .project
- [ ] .template

- [✓] Export generated class files and resources
- [ ] Export all output folders for checked projects
- [ ] Export Java source files and resources
- [ ] Export refactorings for checked projects. Select refactorings...

Select the export destination:

JAR file: C:\temp\ZipValidatorimpl.jar

Options:

- [✓] Compress the contents of the JAR file
- [ ] Add directory entries
- [ ] Overwrite existing files without warning

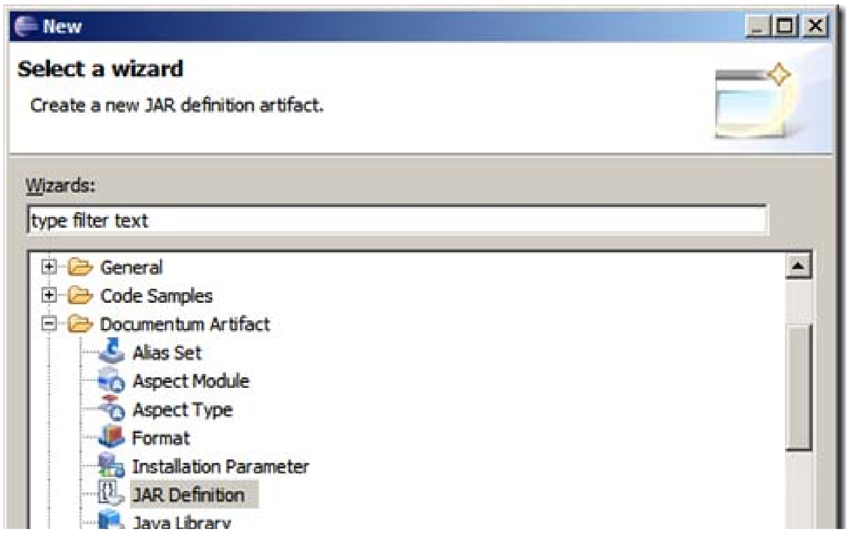
Selecting the com.ajithp.studynotes.sbo.impl for implementation.

Look at the Composers Export Jar screenshots for Interface and implementation (Refer Eclipse Documentation for more details). I think the figures posted above are self explanatory.

The Command line to create a Jar file is **jar cf <name_of_jar>** Please look at the Java Documentation for more details on switches and options of Jar command.

The creation of Jar Definitions are new step added in Composer.

1) In Composer change the perspective to Documentum Artifacts Click **New –> Other –> Documentum Artifacts –> Jar Definition**


New

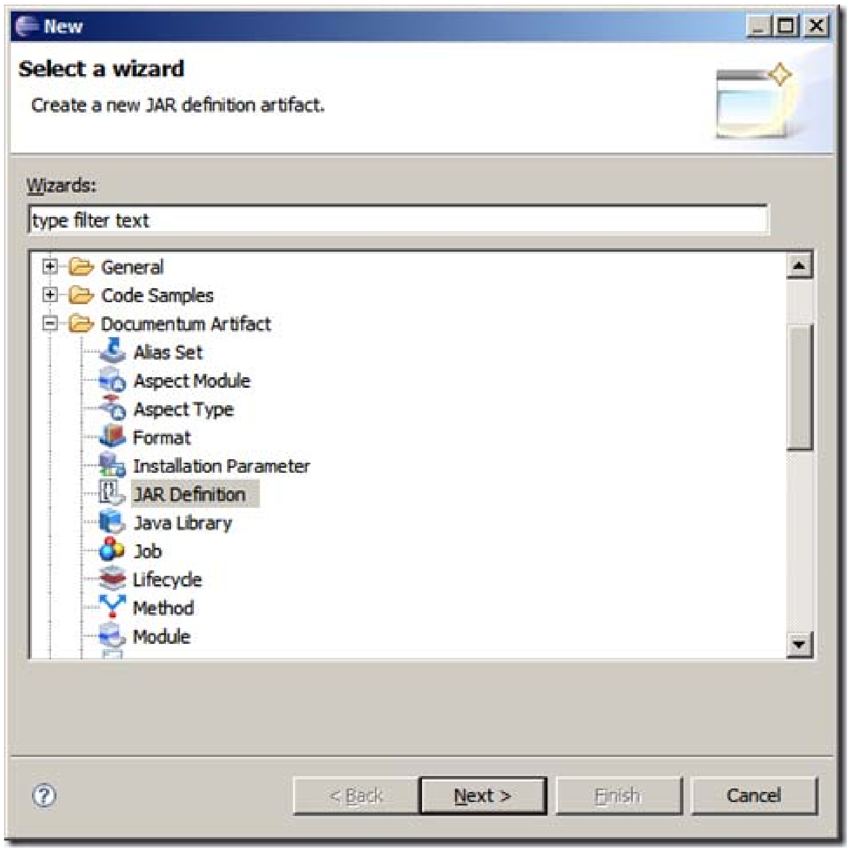
Select a wizard

Create a new JAR definition artifact.

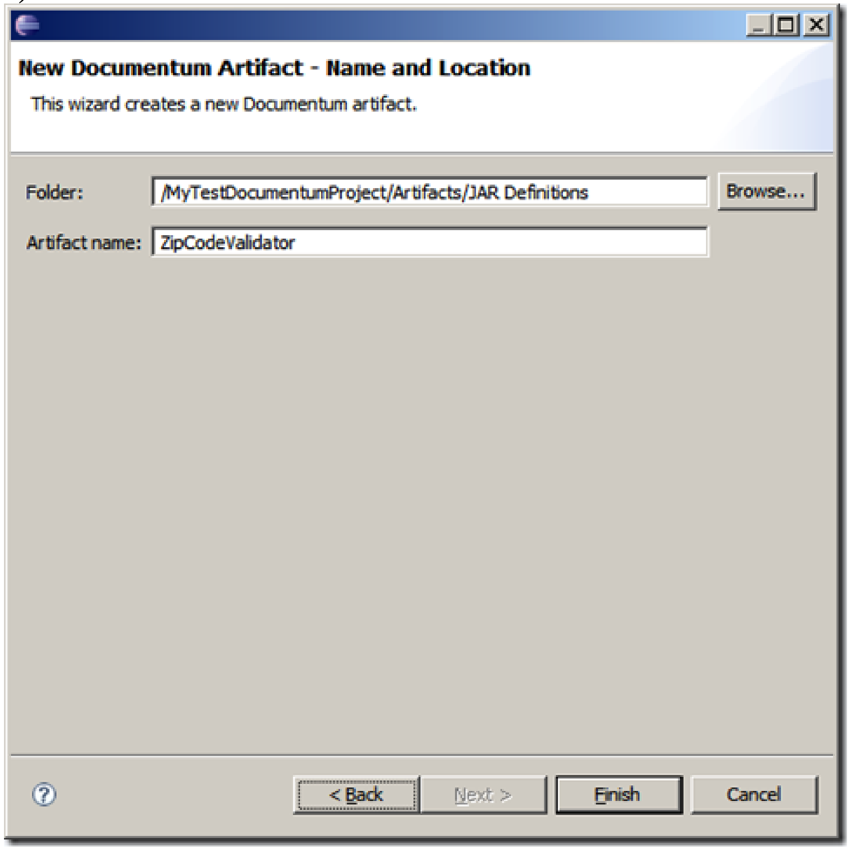
Wizards:

type filter text

- General
- Code Samples
- Documentum Artifact
  - Alias Set
  - Aspect Module
  - Aspect Type
  - Format
  - Installation Parameter
  - JAR Definition
  - Java Library



2) Click Next and Enter the name of for the Jar Definition and click Finish

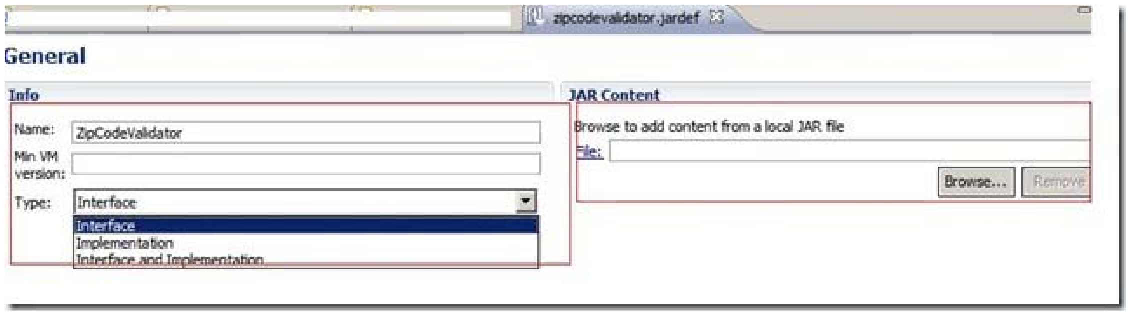

New Documentum Artifact - Name and Location
This wizard creates a new Documentum artifact.
< Back
Next >
Finish
Cancel
Browse...

Folder: /MyTestDocumentumProject/Artifacts/JAR Definitions

Artifact name: ZipCodeValidator

3) Select Type as **Interface** if the jar has only interface , **Implementation** if the jar has only implementation of interface or **Interface and Implementation** if the single jar file has both interface and implementation. click on the Browse button and browse to the jar created in the last step.

In Our case create two Jar Definitions The first one with type as Interface pointing to Jar Created for SBO and second one with type Implementation pointing to the implementation jar


zipcodevalidator.jardef

General

Info

Name: ZipCodeValidator

Min VM version:

Type: Interface

JAR Content

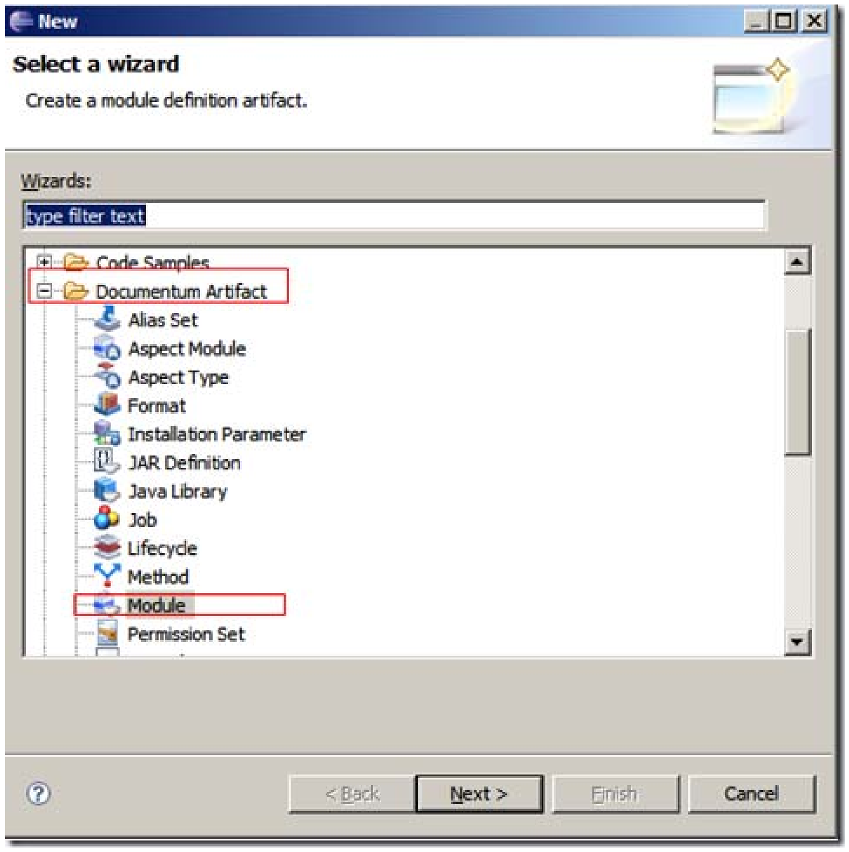
Browse to add content from a local JAR file

File:

Name the Interface jar def as zipcodevalidator and the implementation jardef as zipcodevalidatorimpl

Step 4 : Create a Module and Deploy the SBO

In Composer change the perspective to Documentum Artifacts then Click **New –> Other –> Documentum Artifacts –> Module**


New

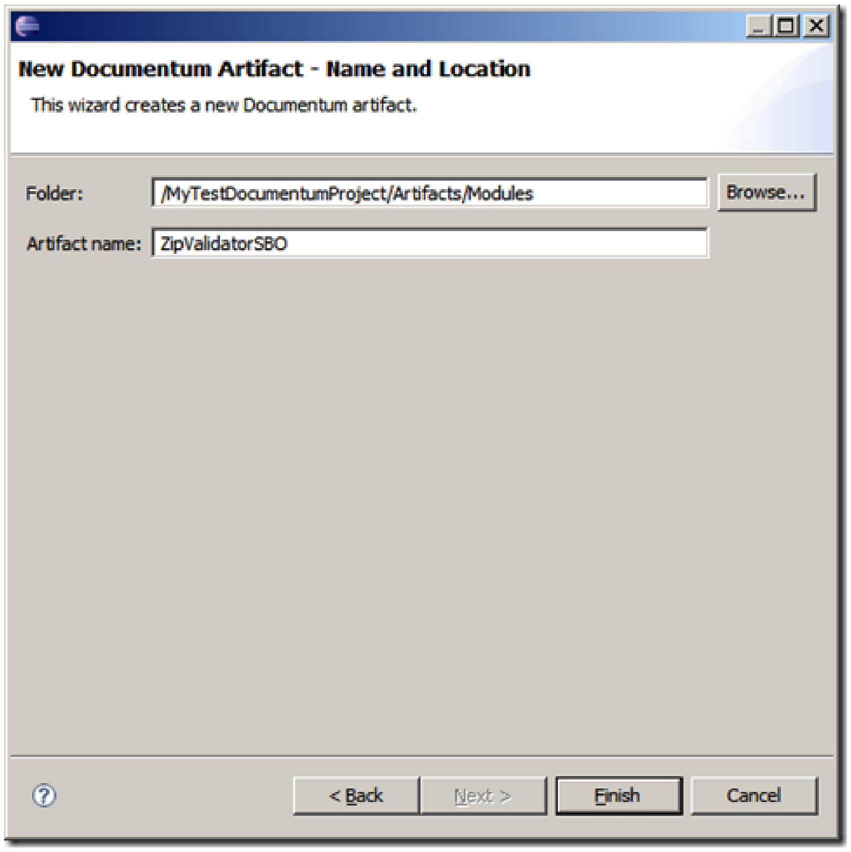
Select a wizard

Create a module definition artifact.

Wizards:

type filter text

Give a valid name and leave the default folder and Click Finish



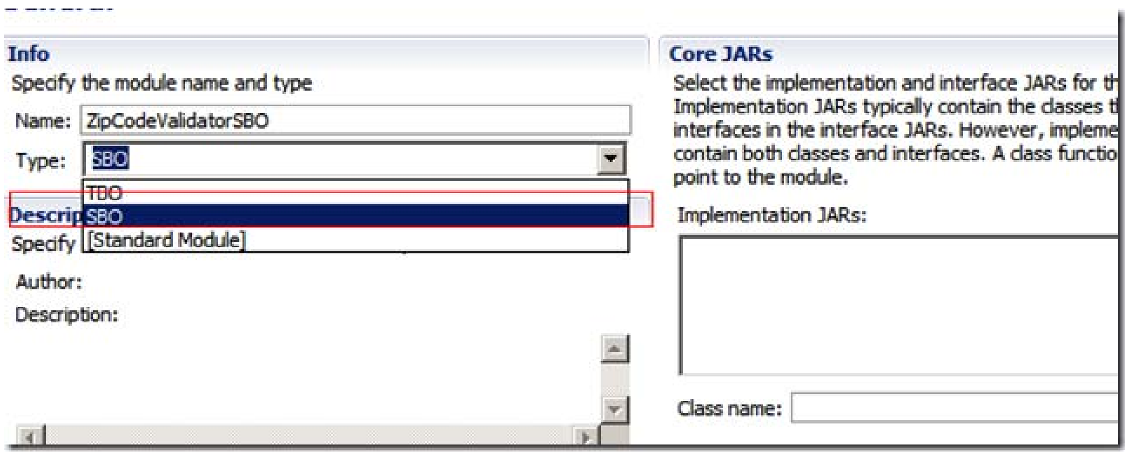
New Documentum Artifact - Name and Location

This wizard creates a new Documentum artifact.

Folder: /MyTestDocumentumProject/Artifacts/Modules

Artifact name: ZipValidatorSBO

In the Module edit window select SBO from the dropdown



Info

Specify the module name and type

Name: ZipCodeValidatorSBO

Type: SBO

Core JARs

Select the implementation and interface JARs for th

Implementation JARs typically contain the classes t

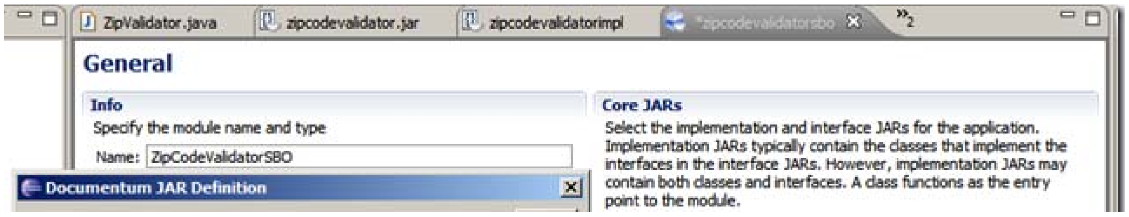
interfaces in the interface JARs. However, impleme

contain both classes and interfaces. A class functio

point to the module.

Implementation JARs:

Now Click on Add Section of Implementation Jars of Core Jars. A new pop up window will appear which will have list of all the Jar definitions set to Type Implementation and Interface and Implementation. Select the one you wanted to use for ZipCodeValidatorSBO that is ZipCodeValidatorImpl.



General

Info

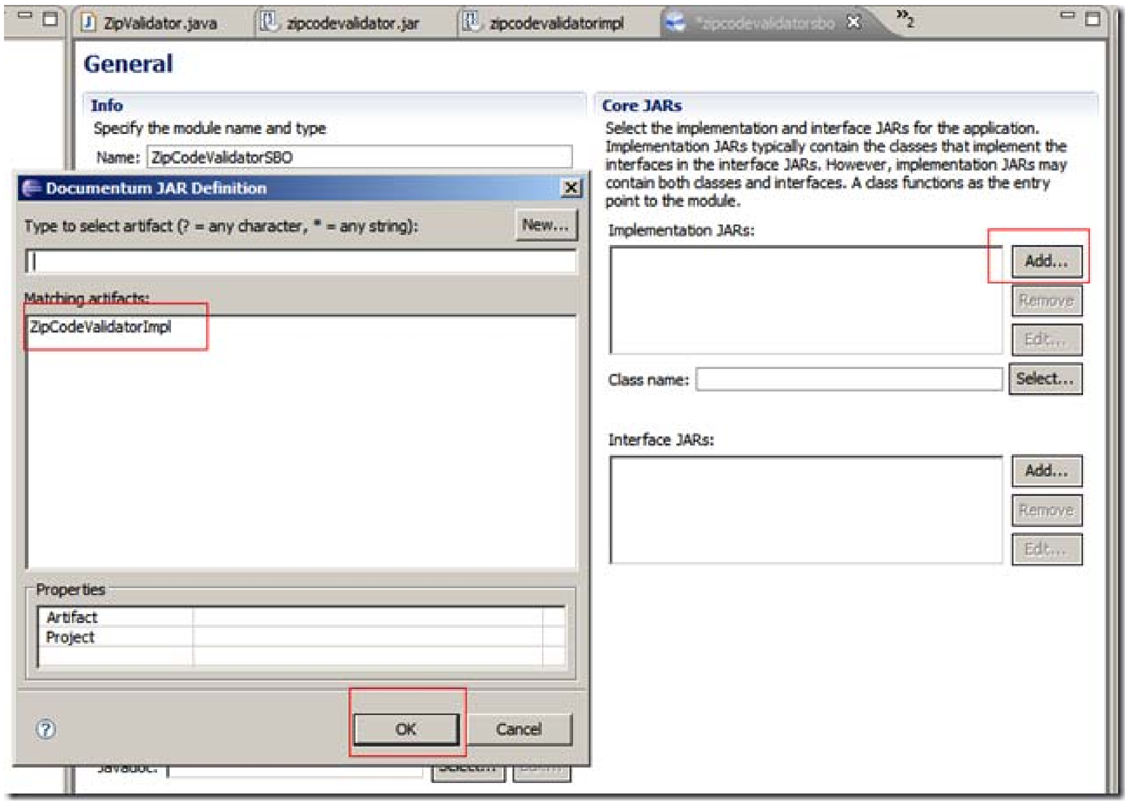
Specify the module name and type

Name: ZipCodeValidatorSBO

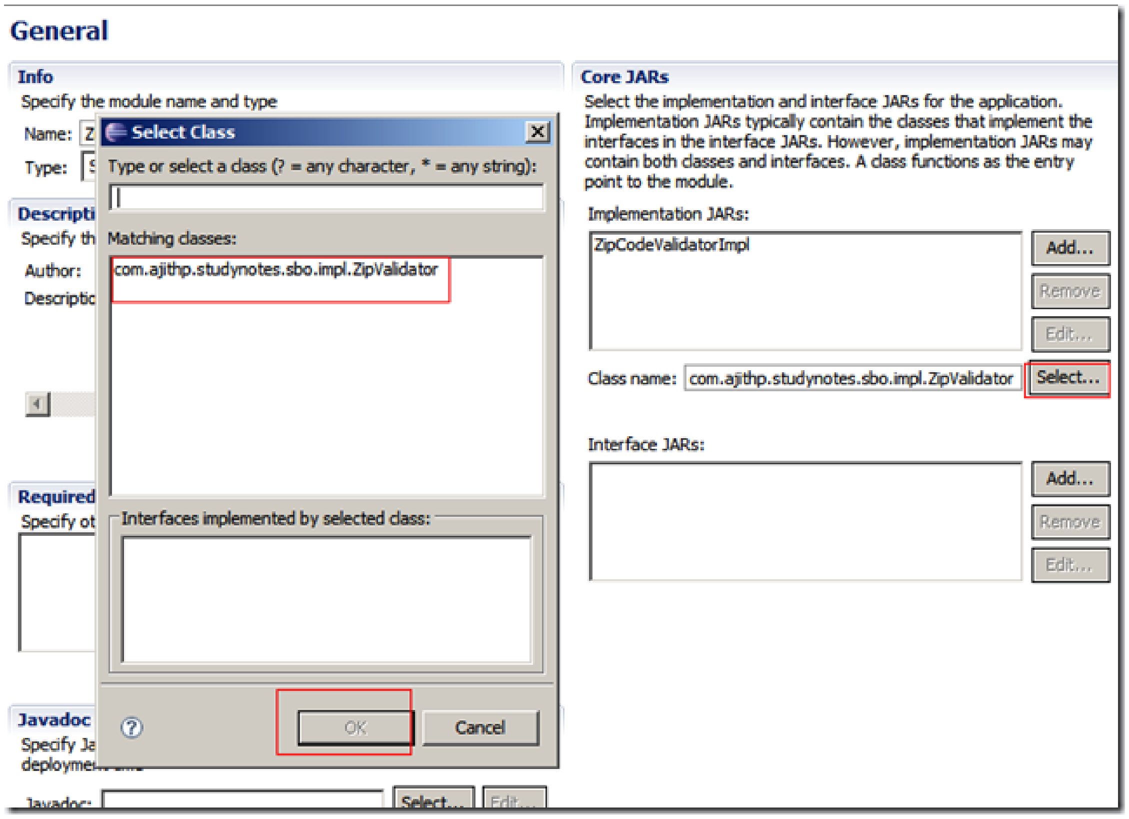
Documentum JAR Definition

Core JARs

Select the implementation and interface JARs for the application. Implementation JARs typically contain the classes that implement the interfaces in the interface JARs. However, implementation JARs may contain both classes and interfaces. A class functions as the entry point to the module.

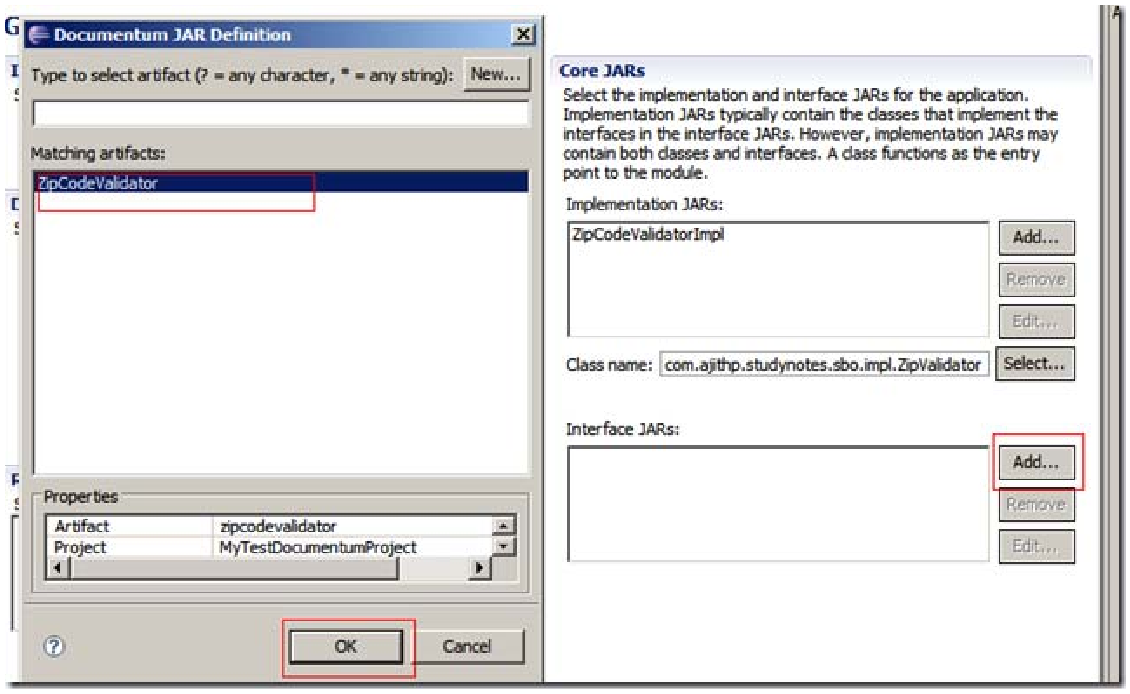

ZipValidator.java
zipcodevalidator.jar
zipcodevalidatorimpl
*zipcodevalidatorsbo
General
Info
Specify the module name and type
Name: ZipCodeValidatorSBO
Documentum JAR Definition
Type to select artifact (? = any character, * = any string):
New...
Matching artifacts:
ZipCodeValidatorImpl
Properties
Artifact
Project
OK
Cancel
Core JARs
Select the implementation and interface JARs for the application. Implementation JARs typically contain the classes that implement the interfaces in the interface JARs. However, implementation JARs may contain both classes and interfaces. A class functions as the entry point to the module.
Implementation JARs:
Add...
Remove
Edit...
Class name:
Select...
Interface JARs:
Add...
Remove
Edit...

Click on the Select Button near pointing to Class name and Select the implementation class. In this case ZipValidator


General
Info
Specify the module name and type
Name:
Type:
Description
Author:
Required
Javadoc
Select Class
Type or select a class (? = any character, * = any string):
Matching classes:
com.ajithp.studynotes.sbo.impl.ZipValidator
Interfaces implemented by selected class:
OK
Cancel
Core JARs
Select the implementation and interface JARs for the application. Implementation JARs typically contain the classes that implement the interfaces in the interface JARs. However, implementation JARs may contain both classes and interfaces. A class functions as the entry point to the module.
Implementation JARs:
ZipCodeValidatorImpl
Add...
Remove
Edit...
Class name: com.ajithp.studynotes.sbo.impl.ZipValidator
Select...
Interface JARs:
Add...
Remove
Edit...

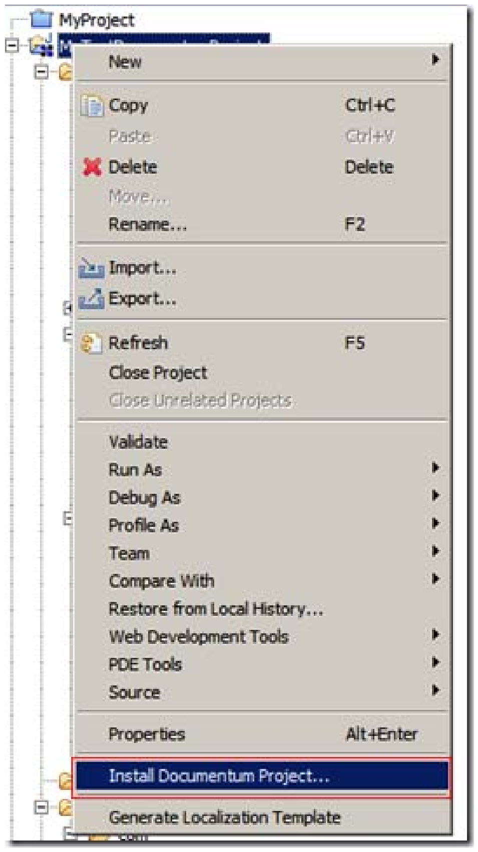
Now Click on Add Section of Interface Jars of Core Jars. A new pop up window will appear which will have list of all the Jar definitions set to Type Interfaces and Interface and Implementation. Select the one you wanted to use for

ZipCodeValidatorSBO that is ZipCodeValidator.

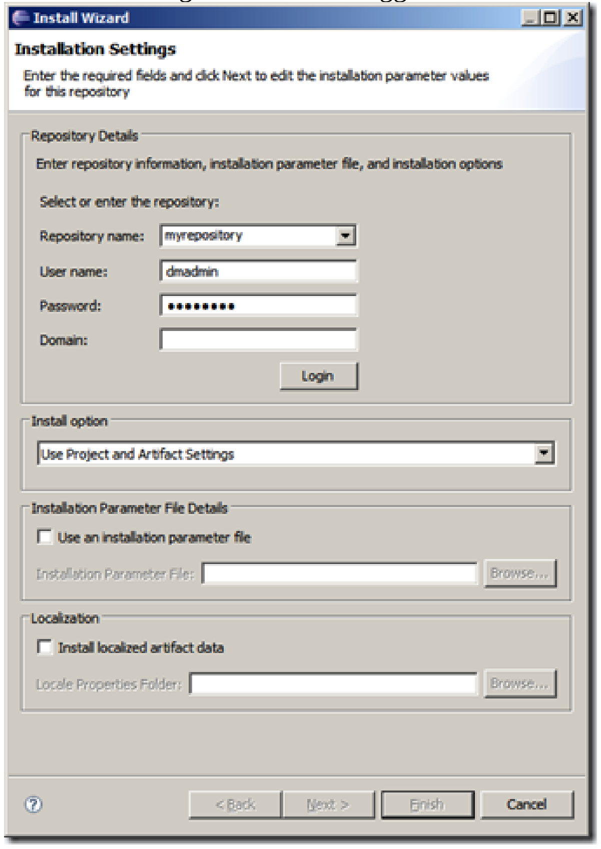

Documentum JAR Definition
Type to select artifact (? = any character, * = any string):
New...
Matching artifacts:
ZipCodeValidator
Properties
Artifact zipcodevalidator
Project MyTestDocumentumProject
OK
Cancel
Core JARs
Select the implementation and interface JARs for the application. Implementation JARs typically contain the classes that implement the interfaces in the interface JARs. However, implementation JARs may contain both classes and interfaces. A class functions as the entry point to the module.
Implementation JARs:
ZipCodeValidatorImpl
Add...
Remove
Edit...
Class name: com.ajithp.studynotes.sbo.impl.ZipValidator
Select...
Interface JARs:
Add...
Remove
Edit...

For more details of other options refer to Documentum Composer Manual. Save the Module.

Now right click on the project and install the Documentum project



Click on the Login button after logged in Click on Finish to start the installation.



Install Wizard

Installation Settings

Enter the required fields and click Next to edit the installation parameter values for this repository

Repository Details

Enter repository information, installation parameter file, and installation options

Select or enter the repository:

Repository name: myrepository

User name: dmadmin

Password: ••••••••

Domain:

Install option: Use Project and Artifact Settings

Installation Parameter File Details

- [ ] Use an installation parameter file

Installation Parameter File:

Localization

- [ ] Install localized artifact data

Locale Properties Folder:

Look at the Documentum composer documentation to know more about the Installation options.

How to use SBO from a Client Application

follow the below steps to instantiate a SBO from a client application.

1) Get the Local client
2) Create a login info and populate the login credentials.
3) Create a **IDfSessionManager** object
4) Use the **newService ()** from the Client Object to create a SBO instance

```
// create client

  IDfClient myClient = DfClient.getLocalClient();

  // create login info

  IDfLoginInfo myLoginInfo = new DfLoginInfo();

  myLoginInfo.setUser("user");

  myLoginInfo.setPassword("pwd");

  // create session manager

  IDfSessionManager mySessionManager = myClient.newSessionManager();

  mySessionManager.setIdentity("repositoryName", myLoginInfo);

  // instantiate the SBO

  IZipValidatorSBO zipValidator = (IZipValidatorSBO) myClient.newService( IZipValidatorSBO.class.getName(), mySessionManager);
```

```
// call the SBO service

zipValidator.validateZipCode(obj, zipCode, "repositoryName");
```