PDTool Developer’s Guide

An Open Source Asset for use with TIBCO® Data Virtualization

Project Name	AS Assets PDTool (Promotion and Deployment Tool)
Document Location	This document is only valid on the day it was printed. The source of the document will be found in the PDTool and PDToolRelease folder (https://github.com/TIBCOSoftware)
Purpose	Developer’s Guide

Revision History

Version	Date	Author	Comments
1.0	03/23/2011	Gordon Rose	Initial revision
1.0.1	8/1/2011	Mike Tinius	Revision due to architecture changes. Added text for creating a new VCS LifecycleListener.
1.1	08/15/2012	Mike Tinius	Added details about getting started with new PDTool projects in subversion.
1.2	09/01/2012	Mike Tinius	Added information on doing a VCS assessment prior to development.
2.0	08/20/2013	Mike Tinius	Updated documentation to Cisco format.
3.0	11/15/2013	Mike Tinius	Updated URL for Cisco subversion server
3.1	1/31/2014	Mike Tinius	Updated PDTool subversion project download section and debug w/https.
3.2	3/12/2014	Mike Tinius	Updated for open source release.
3.3	3/19/2014	Mike Tinius	Moved Eclipse specific info to Developer Startup Guide.
3.4	3/10/2015	Mike Tinius	Converted to Cisco format.
4.0	12/12/2017	Mike Tinius	Converted to Tibco format.

Related Documents

Name	Version

Supported Versions

Name	Version
TIBCO® Data Virtualization	7.0.4 or later

Table of Contents

1 Introduction

Purpose

The purpose of this document is to provide guidance on how to extend the functionality of the Promotion and Deployment Tool (“PDTool”).

This document covers the following topics:

1. **Promotion and Deployment Tool Overview** – Describes why scripts are needed for deployment.
2. **Extending the Promotion and Deployment Tool** – Describes the architecture and design principles.

Audience

This document is intended to provide guidance for the following users:

- **Developers** – provides guidance on how to develop new Modules of functionality.

Getting Started

To get started downloading the open source PDTool code, refer to the “Promotion and Deployment Tool Developer's Startup Guide - v4.0.pdf” for the details on how to do this and what software is required for development.

References

Product references are shown below. Any references to CIS or DV refer to the current TIBCO® Data Virtualization.

- TIBCO® Data Virtualization was formerly known as
  - Cisco Data Virtualization (DV)
  - Composite Information Server (CIS)

2 Promotion and Deployment Tool Overview

What is the Promotion and Deployment Tool "PDTool"?

The Promotion and Deployment Tool is a powerful ally of the CIS administrator, developer and deployment manager.

The goal of developing the tool is to provide a promotion/deployment tool that addresses 90% of CIS customers with an out-of-the-box solution, and beyond that, an extendable framework for the 10% of customers with unique requirements. The architecture of the tool allows it to support command-line and Ant deployments. It supports both VCS (primarily Subversion) and traditional CAR file based deployments. Finally, it supports both local and remote deployment.

Among the tasks included in version one are:

- Configure server attributes
- Backup Server
- Import .car files
- Restart CIS
- Manage groups
- Manage users
- Create, update, and delete resource owners
- Manage resource privileges
- Configure, enable, and re-introspect data sources
- Utilize the general attribute configuration interface
- Rebind procedures and views to new data sources
- Manage resources
- Remove resources
- Validate resources (existence test)
- Regression testing of resources

Recommended Development Tools

The recommended development environment for extending the PDTool is a robust Java and XML development tool, such as eclipse. Eclipse is the tool in which the PDTool has been developed, and the PDTool is delivered to customers both as a ready-to-go collection of jars, configuration files and scripts and as a collection of eclipse workspaces. For the development of new XML

Schema Definitions (XSDs), developers may wish to use a dedicated XML editing tool. Ultimately, any tool that can be used to edit text-based files such as Java classes and XML files is technically sufficient. Stylus Studio was used to modify XML Schemas. It also has a documentation generator that is very nice. The HTML documentation is included as part of the release documentation.

PDTool Artifact Types

In extending the tool, the developer will create and/or modify a variety of documents based on the following PDTool artifact types. Those artifact types are:

- XML Schema Definitions ("project XSDs") – The project XSDs represent types used and managed by the tool, such as GroupType, RelationalDatasourceType, etc.
- Java classes (self-explanatory)
- Shell scripts – Please note, the use of shell scripts is optional. The tool ships with pre-developed shell scripts for Windows and Linux.
- Properties files – Properties files consist of XML files used to store the information required for the PDTool to carry out the actions requested of it.
- Ant tasks and scripts – Please note that the use of Ant is optional.

Spring configuration files – Please note, the use of Spring is optional. The PDTool supports Spring dependency injection.

Promotion and Deployment Tool Eclipse Folders

The PDTool project is one big java project that contains several sub-folders which work in concert to provide a way to build the release for PDTool. Note: PDTool 6.2 is no longer supported but remains the project for now. However, no releases are being built for Data Virtualization server 6.2.

PDToolModules

The PDToolModules folder consists of Java classes generated from the project schema (PDToolModules.xsd) using JAXB. PDToolModules folder is a dynamic project with no dependencies on other projects. It is managed 100% via a schema editor of your choice. However, the one requirement that we have is that the tool can generate HTML documentation. For example, Stylus Studio can generate HTML documentation.

CISAdminApi7.0.0

The CISAdminAPI7.0.0 folder contains java classes generated by running the Java wsimport tool against the DV 7.0 web services admin API. CISAdminAPI7.0.0 is a static project with no dependencies on other projects. It provides CISAdminApi.jar to PDTool project which is the PDTool project for DV.

PDTool

The PDTool project contains the core modules of the tool, such as Datasource, Resource, User, etc. The PDTool code-base supports CIS 6.2. PDTool is a dynamic project and depends on the CisAdminAPI and PDToolModules projects. PDTool is the current baseline and supports CIS 6.2 and above as long as the CIS API does not change.

PDToolDocs

The PDToolDocs project contains all of the documentation for PDTool.

PDToolRelease

The PDToolRelease is a separate project that provides a central repository for a PDTool release. The 7.0.0 sub-folder provides a release for PDTool 7.0.0 and higher. The regression folder provides an automated testing facility for each PDTool release.

3 Creating Promotion and Deployment Tool Modules

This section provides a detailed look at the architecture of a PDTool module and describes the steps necessary to create or modify a PDTool module.

Overview of Promotion and Deployment Tool Modules

The PDTool module is where work is actually performed. PDTool modules are defined in the PDTool project, making it primary area of focus for a developer who wishes to add functionality to the PDTool. PDTool includes two main packages – com.tibco.ps.deploytool and com.tibco.ps.common. The "deploytool" package contains the classes associated with a PDTool module and the "common" package consists mostly of reusable code.

Figure 1.0 below provides a detailed view of the technical architecture. The areas outlined in dashed red lines indicate a major functional area. Some of these areas are also Eclipse projects. The gray boxes within the functional areas represent the various aspects of the technical architecture. The yellow boxes represent those artifacts that are modified by the user. The red boxes represent Composite product provided scripts. The blue boxes represent software programs that are present in the user's environment. The green leaves represent the Spring framework. A complete explanation of the Deploy Command line scripts and Ant build file execution is covered in the "Promotion and Deployment Tool – User Guide". The Ant framework is not covered in the Developer's Guide as there is no additional development required. This component is considered to be complete and out-of-the-box. It is a generic orchestration framework for executing Ant build scripts. PDToolModules is only used for generating JAXB classes from XML Schema. The generation has been automated based on the XML Schema deployToolModules.xsd. The various java artifacts for the PDTool Project will be explained further below.

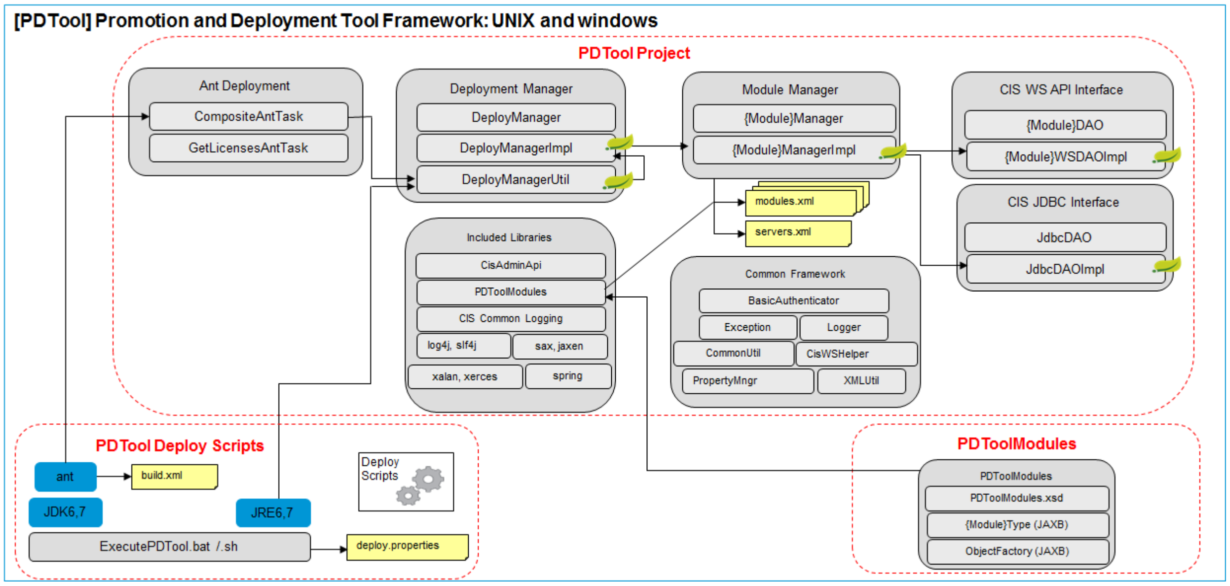


Figure 1.0 – Promotion and Deployment Tool Framework Architecture

Overview of the DeployManager Class

The entry point for all external clients to the functionality of the tool is the DeployManager class in the deploytool package. DeployManager is an example of the façade pattern, providing a single, simple interface to the PDTool. DeployManager includes method signatures in its interface for all of the methods of all modules, for example, the reIntrospectDatasource method of the DataSourcemanager module, etc.

DeployManagerImpl is the implementation of the DeployManager interface, delegating calls to it to the various modules in the PDTool. DeployManagerImpl represents the first instance of loose coupling that is found throughout the architecture of the PDTool. DeployManagerImpl uses factory-style calls, for example getDataSourceManager(), making it possible for the implementation of the getDataSourceManager() method – the DataSourceManagerImpl – to determine what the actual implementation of the DataSourceManager is. This is achieved by supporting Spring-based dependency injection. This architecture allows customers to completely replace any individual module implementations without disturbing any other aspect of the tool or how it operates. A default implementation for every manager is shipped with the tool.

Promotion and Deployment Tool Module Structure

The implementation of a module follows the CRUD paradigm and achieves another separation of responsibilities by focusing on processing and organizing the information required to perform an action while delegating the execution of the action to another class. The CRUD actions are ultimately performed by the module’s corresponding DAO class. Again, this architecture allows customers to change how an action is carried out without changing how it is invoked. The default DAO implementations use the web services API to carry out tasks – another implementation could take a different approach, such as using JDBC calls to call PDTool SQL procedures.

The PDTool module is clearly the focal point for a developer who wants to customize and/or extend the PDTool. The other key areas where a developer can extend the PDTool are shell scripts and Ant tasks and build scripts. Those topics are covered in later sections.

A PDTool module maps to a broad, common area of CIS functionality. The DataSource manager is one example, the UserManager another. In this section, we’ll analyze how the DataSource manager is constructed, making it clear in the process the steps required to create a new module as well as to modify or extend an existing module.

A module is made up of four classes. Assuming we’re writing a module that will be dedicated working with the ABC are of CIS functionality, then the classes will be:

ABCManager – ABCManger is an interface containing method signatures.

ABCManagerImpl – ABCManagerImpl is an implementation of the ABCManager interface. It delegates the work to the DAO class.

ABCManagerDao – ABCManagerDao defines the CRUD operations associated with each method of the module. It further delegates the work to an implementation class.

ABCManagerWSDAOImpl – Finally, an implementation class performs the actions defined by the DAO. In this example, the "WS" characters in the name indicate that this implementation uses the web services admin api. The implementation could just as easily use JDBC to call the procedures included in the PDTool that wrap the API calls.

Figure 1.1 below shows the interaction between the Module Manager implementation and the Web Service API DAO's.

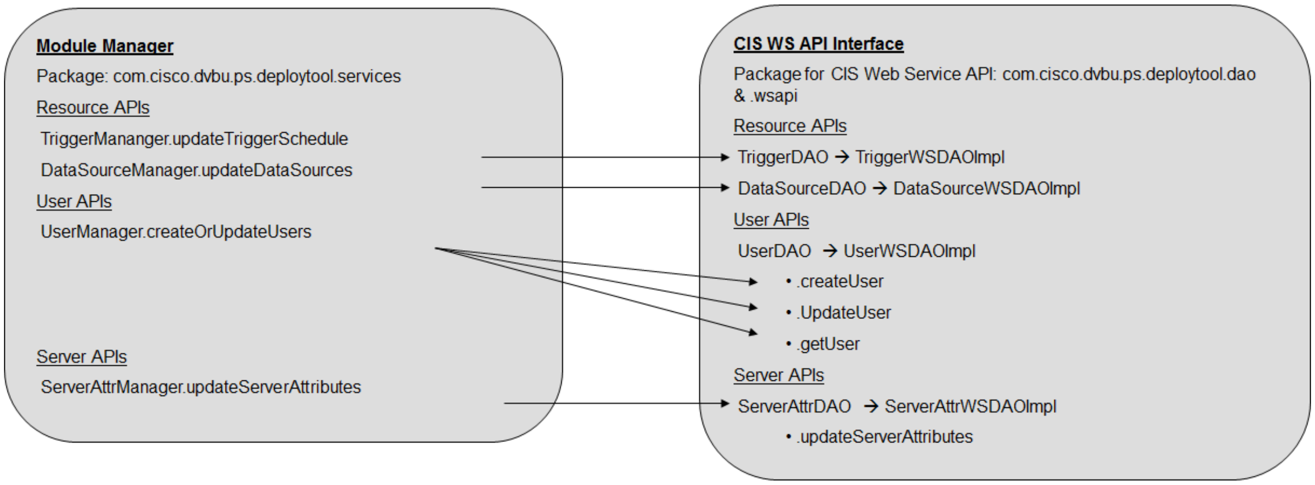

Module Manager
Package: com.cisco.dvbu.ps.deploytool.services
Resource APIs
TriggerMananger.updateTriggerSchedule
DataSourceManager.updateDataSources
User APIs
UserManager.createOrUpdateUsers
Server APIs
ServerAttrManager.updateServerAttributes
CIS WS API Interface
Package for CIS Web Service API: com.cisco.dvbu.ps.deploytool.dao & .wsapi
Resource APIs
TriggerDAO → TriggerWSDAOImpl
DataSourceDAO → DataSourceWSDAOImpl
User APIs
UserDAO → UserWSDAOImpl
• .createUser
• .UpdateUser
• .getUser
Server APIs
ServerAttrDAO → ServerAttrWSDAOImpl
• .updateServerAttributes

Figure 1.1 – Module Manager Interaction with DAO

Studying a Live Example

Now we'll look at a live example, the DataSourceManager module.

DataSourceManager.java

Below you'll see the workings of the DataSourceManager interface. Little explanation is required. Please note the generateDataSourcesXML method, which makes it trivial for customers to extract the metadata for multiple datasources, hand-code modifications such as port, host, etc., and then use the modified XML as an input to a task such as updateDataSource.

```
package com.tibco.ps.deploytool.services;

import com.tibco.ps.common.exception.CompositeException;

public interface DataSourceManager {

    public void updateDataSources(String serverId, String dataSourceIds,
        String pathToDataSourceXML, String pathToServersXML);

    //
    // -- snip other work methods, such as enableDataSource, for the sake of brevity
    //

    public void generateDataSourcesXML(String serverId, String startPath,
        String pathToDataSourceXML, String pathToServersXML) throws CompositeException;

}
```

DataSourceManagerImpl.java

The primary job of the implementation class is to validate and organize the inputs provided in property files and on the command-line and invoke the appropriate work methods. There are four areas you should focus on in the code from DataSourcemanagerImpl.java below:

1) Note that all work methods, such as updateDataSources (1a), are funneled through a common method, dataSourceAction (1b). This greatly simplifies the construction of the manager.
2) poupulateRelationalDataSourceTypeAttributes is an example of a method responsible for reading the module XML input files and populating attributes for processing by the DAO.
3) Note the use of one of the many JAXB classes based on the PDTool schemas.
4) Note the ability to use any DAO class to perform the work. If Spring dependency injection were used, by the time the DataSourceManagerImpl class were instantiated, the dataSourceDAO would be valued and therefore not null.

```
package com.tibco.ps.deploytool.services;

import java.math.BigInteger;
// -- snip additional imports

public class DataSourceManagerImpl implements DataSourceManager{
    private static Log logger = LogFactory.getLog(DataSourceManagerImpl.class);

    private DataSourceDAO dataSourceDAO = null;

    public void updateDataSources(1a)(String serverId, String dataSourceIds,
        String pathToDataSourceXML, String pathToServersXML) throws CompositeException {

          // -- snip logging call
        dataSourceAction(1b) (DataSourceDAO.action.UPDATE.name(), serverId,
            dataSourceIds, pathToDataSourceXML, pathToServersXML);
    }

     // -- snip intervening methods for sake of brevity
    private void poupulateRelationalDataSourceTypeAttributes(2) (
        DataSourceChoiceType(3) dataSourceChoiceType, Attribute attribute) {

        if (attribute.getName().equalsIgnoreCase("urlIP")) {
            dataSourceChoiceType.getRelationalDataSource().setHostname(attribute.getValue());
        } else if (attribute.getName().equalsIgnoreCase("urlPort")) {
            // -- snip remaining if..elseif for sake of brevity
        }
    }

    public DataSourceDAO getDataSourceDAO() {

       if(dataSourceDAO(4) == null){
           dataSourceDAO = new DataSourceWSDAOImpl();
       }
       return dataSourceDAO;
    }

    public void setDataSourceDAO(DataSourceDAO dataSourceDAO) {
       this.dataSourceDAO = dataSourceDAO;
    }

}
```

DataSourceDAO.java

In the model implementation, the DAO class is a simple interface. Note the single method, takeDataSourceAction.

```
package com.tibco.ps.deploytool.dao;

import com.tibco.ps.common.exception.CompositeException;
// -- snip additional imports

public interface DataSourceDAO {

        public static enum action {UPDATE,ENABLE,REINTROSPECT};

        public ResourceList takeDataSourceAction(String actionName,
        String dataSorucePath, AttributeList dataSoruceAttributes,String serverId,
        String pathToServersXML) throws CompositeException;
}
```

DataSourceWSDAOImpl.java

Finally, the work is carried out using whatever available approach the developer prefers to use. In this case, the work is carried out by making calls to the web services admin api. Note how elegantly simple the DAO implementation class is.

```
public class DataSourceWSDAOImpl implements DataSourceDAO {

        private static Log logger = LogFactory.getLog(DataSourceWSDAOImpl.class);

        public ResourceList takeDataSourceAction(String actionName,
            String dataSorucePath, AttributeList dataSoruceAttributes,String serverId,
            String pathToServersXML) throws CompositeException {

            ResourceList returnResList = null;

            // -- read target server properties from xml and build target
             //   server object based on target server name
            CompositeServer targetServer = WsApiHelperObjects.getServer(
                 serverId, pathToServersXML);

            // -- Construct the resource port based on target server name
            ResourcePortType port = CisApiFactory.getResourcePort(targetServer);

            try {

                 if (actionName.equalsIgnoreCase(DataSourceDAO.action.UPDATE.name())){
                     returnResList = port.updateDataSource(dataSorucePath,
                         DetailLevel.FULL, null, dataSoruceAttributes);

                 } else if (actionName.equalsIgnoreCase(DataSourceDAO.action.ENABLE.name())){
                     // -- Snip rest if else if structure
                 }

                // -- Snip loggin call

            } catch (UpdateDataSourceSoapFault e) {
                // -- Snip other exceptions and handling code
            }
                 return returnResList;
        }
}
```

Steps to Create a Promotion and Deployment Tool Module

At this point, it's relatively easy to outline the steps required to create a new PDTool module:

1. Define the area of functionality to be managed by the module

2. Determine the work methods of the module.
3. Define the DAO for the module. This should be a simple interface.
4. Implement the DAO using the technology of choice.
5. Define the module interface.
6. If needed, define supporting project XSDs.
7. Define the module implementation. This should ideally funnel all calls to do work through a single method.
8. Update the DeployManager and DeployManagerImpl classes to reflect the new actions.

Resource Casting

CIS contains many different types of resources. The basic structure returned by the common functions *getResourcesFromPath()* and *getImmediateResourcesFromPath()* is a ResourceList which is defined as a List<Resource>. The Resource type is commonly used throughout the Deploy Tool. However, each type of resource can extend Resource to contain any number of additional objects. Therefore, it is necessary to cast the correct Resource type onto the structure. By casting an individual resource from a resource list, it will be possible to utilize the specific get and set methods for that resource type. The following is a list of valid Resource Types:

Class	ResourceType
Resource	= Generic
ContainerResource	= ResourceType.*CONTAINER*
DataSourceResource	= ResourceType.*DATA_SOURCE*
DefinitionSetResource	= ResourceType.*DEFINITION_SET*
LinkResource	= ResourceType.*LINK*
ProcedureResource	= ResourceType.*PROCEDURE*
TableResource	= ResourceType.*TABLE*
TreeResource	= ResourceType.*TRIGGER*
TriggerResource	= *ResourceType.TABLE*

```
ResourceList resourceList = new ResourceList();

resourceList.getResource().addAll(DeployManagerUtil.getDeployManager().getResourcesFromPath(
serverId, startPath, resourceTpe, DetailLevel.FULL.name(), pathToServersXML).getResource());

// Loop over the list of resources
for (Resource resource : resourceList.getResource()) {
         // Process the resource if it is a TABLE type
        if (table.getType().name().equalsIgnoreCase("TABLE")) {
                // Cast resource to a the TableResource type
                TableResource tableResource = (TableResource)resource;

                [code removed]
        }
}
```

4 Lab – Create a New PDTool Module

In this section, you will create a simple, new module based on the existing GroupManager module. The exercise will largely consist of navigating and studying a PDTool module in order to understand and use sections of the code to create your own module.

Please note – you should not check the Java code you create in this lab to the SVN repository.

Creating the DAO Interface

It is easiest to begin the creation of a new module with the simultaneous creation of the DAO interface and DAO implementation. The reason is that it is the information required by the DAO implementation to perform its work that will dictate the interface and implementation of the module manager. In effect, you start in the middle and work your both up and down.

1. Add a new interface called <YourNames>GroupDAO to the package com.tibco.ps.deploytool.dao, as in JohnsGroupDAO (hint – right-click on the package, select *New* and then *Interface*. Supply the name and click *OK*.)

2. Copy and paste *only* the line below for from GroupDAO.java to your interface.

```
public static enum action {CREATEORUPDATE,CREATE, UPDATE,
    DELETE, ADDUSER, REMOVEUSER};
```

At this point, all you are defining are the actions the DAO will take. You will need this enum in the implementation classes.

Creating the DAO Implementation

The DAO implementation is where the rubber meets the road. We'll use the CIS web services admin API to carry out the work of our module.

1. Add a new class called <YourNames>GroupDAOImpl to the package com.tibco.ps.deploytool.dao.wsapi, as in JohnsGroupDAOImpl (hint – right-click on the package, select *New* and then *Class*. Supply the name and click *OK*.)

2. Implement the <YourNames>GroupDAOImpl interface you created in the previous section, as in

   ```
   public class JohnsGroupDaoImpl implements JohnsGroupDAO {
   ```

3. Copy and the body of the class into your new class (we are only interested in one of the actions, but it's not necessary for this exercise to edit the takeDataSourceAction to reflect that. More on the other methods in the class below and again later when we create the manager interface and implementation).

In addition to the fact that developing the modules in this order makes sense, it also allows you to see how simple it is to perform work in the PDTool. Note that the steps are basically to use the CisApiFactory to get a reference to the appropriate web service port of the server on which the work is to be performed and then to call one of the port’s operations with a simple set of arguments.

The DAO also includes some helper methods – namely getGroup and getDomainMemberReferenceList. The getAllGroups method corresponds to the generateGroupsXML. Since each DAO will have a method for generating XML, you will want to pay attention to the implementation of this method.

Let’s point out a few things about how getGroup and getAllGroups work.

- The basic logic of the lookup methods is similar to the logic of the work method – get a reference to the appropriate web services port of the server identified by serverId and, in this case, call the appropriate operation of the User port to retrieve group information.
- The return types of the methods are JAXB classes built using the PDTool XSD schemas.

Lastly, a few more things to note:

- Every implementation class (both Manager and DAO) defines a private logger variable
- There is only one method for performing CRUD operations in the DAO implementation. The code of this method branches based on the action to be performed.
- This implementation of the DAO uses the web services API. However, because the DAO interface and implementation are separate, the implementation could easily take another approach.
- In this case, the work method does not return a value. That will vary by DAO.
- Any exception that occurs will be propagated upwards.

Completing the DAO Interface

We know from having “implemented” our DAO (in reality, there are no method signatures in our interface yet) what the method signatures of our DAO interface must look like. That was determined by the arguments required by the web services API calls we used.

1. Copy the body of the GroupDAO class to the body of <YourNames>GroupDao.

As you can see, the interface is extremely simple. We have one method used for performing all of the actions that the DAO can perform, and two methods associated with the lookup methods of the DAO.

Creating the Manager Interface

We now know exactly what the manager interface should look like. Use the following steps to create the interface for your module manager.

2. Add a new interface called <YourNames>GroupManager to the package com.tibco.ps.deploytool.services, as in JohnsGroupManager (hint – right-click on the package, select *New* and then *Interface*. Supply the name and click *OK*.)

3. Copy and paste the method signatures for generateGroupsXML and createOrUpdateGroups from GroupManager.java to your interface.

Please note, each module will have a generate<Abc>XML, where <Abc> represents the area of functionality that the module addresses. So, it is important to include that method signature in our interface. The generate XML methods make it easier for users to create input files for the PDTool.

Also, if you chose to start the development of the module with the development of the manager interface, then arriving at the correct method signatures would be somewhat of an iterative process. You would start with obvious arguments – for instance, the work of the method would be carried out on a server, so you would at least need to capture the id of the server and the path to the servers XML file. As you develop the code that performs the work, you would then discover the additional information required and refactor the higher level classes to reflect that.

Creating the Manager Implementation

We're now ready to complete the development of our new module. Use the following steps to create the implementation of your module manager.

1. Add a new class called <YourNames>GroupManager Impl to the package com.tibco.ps.deploytool.services, as in JohnsGroupManagerImpl (hint – right-click on the package, select *New* and then *Class*. Supply the name and click *OK*.)

2. Copy and paste the two lines shown below from the top of the GroupmanagerImpl to your new class:

   ```
   private static Log logger = LogFactory.getLog(GroupManagerImpl.class);

   private GroupDAO groupDAO = null;
   ```

   Note that each implementation of a module manager includes private variables for a logger and for the module DAO. Be sure to change the name of the class passed to the getLog method to the name of the class you are creating.

3. Copy and paste the createOfUpdateGroups method into your code. Note that all calls are funneled to doGroupAction.

4. Copy and paste the doGroupAction into your code. Again, note that the doGroupAction method maps to the takeGroupAction method of your DAO.

Please note the use of getGroupDAO() as opposed to the use of the private field groupDao.

5. Add the getPrivileges and getGroups methods. These are helper methods. It should be fairly easy by now to understand how these methods work.

6. Lastly, add the getGroupDao and setGroupDao methods.

The use of getGroupDao makes it possible to use Spring dependency injection to use a DAO other than the default implementation provided with the PDTool.

Using Your New Module

As an optional exercise, update DeployManagerImpl to point it to the methods you have created and use the PDTool User Guide to modify the privileges for an existing group on your local CIS server.

Exploring Common Promotion and Deployment Tool Classes

As a second optional exercise, explore the following common classes to get a better understanding of the workings of the PDTool.

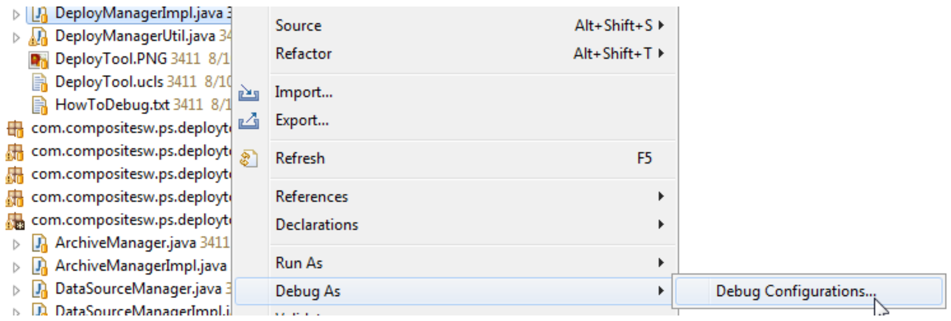
- com.tibco.ps.deploytool.util.DeployUtil
- All classes in the package com.tibco.ps.common.util
- All classes in the package com.tibco.ps.common.util.wsapi

5 How To Debug

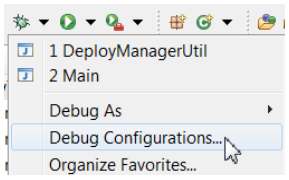
How to Debug in Eclipse

How to Debug:

1) Right click on DeployManagerUtil.java --> Debug As --> Configurations



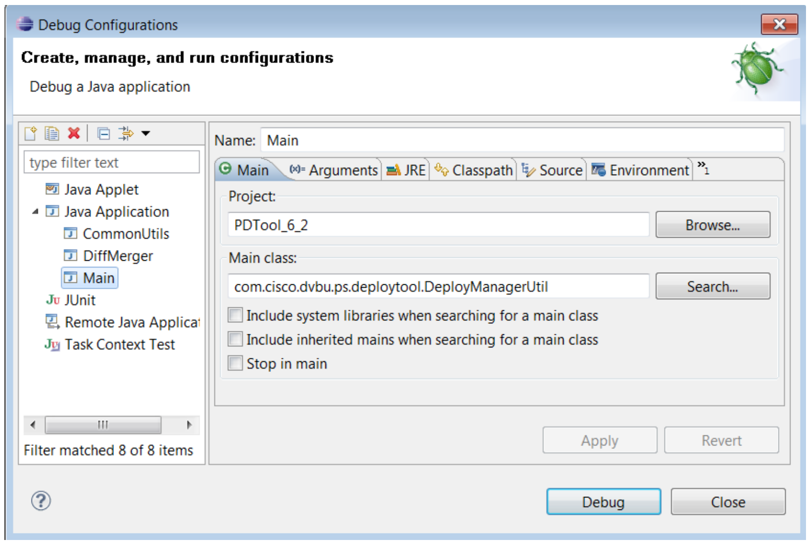
Or…Click the Debug icon and select Debug Configurations



2) Main Tab -->

When Debugging code line, set the Project name: PDTool

Set the Main class: com.tibco.ps.deploytool.DeployManagerUtil


Debug Configurations

Create, manage, and run configurations

Debug a Java application

Name: Main

Main | Arguments | JRE | Classpath | Source | Environment

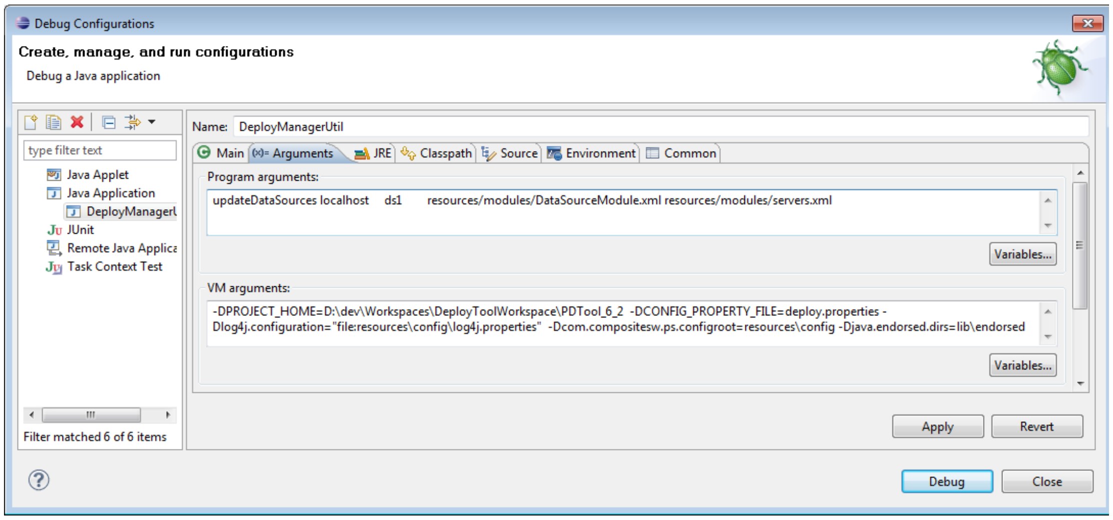
Project: PDTool_6_2

Main class: com.cisco.dvbu.ps.deploytool.DeployManagerUtil

☐ Include system libraries when searching for a main class
☐ Include inherited mains when searching for a main class
☐ Stop in main

2) Arguments Tab --> Set the Program Arguments:

This is where you put your PDTool method and parameters:



Name: DeployManagerUtil

Program arguments:

```
updateDataSources localhost    ds1      resources/modules/DataSourceModule.xml resources/modules/servers.xml
```

VM arguments:

```
-DPROJECT_HOME=D:\dev\Workspaces\DeployToolWorkspace\PDTool_6_2  -DCONFIG_PROPERTY_FILE=deploy.properties -
Dlog4j.configuration="file:resources\config\log4j.properties"  -Dcom.compositesw.ps.configroot=resources\config -Djava.endorsed.dirs=lib\endorsed
```

<u>example 1: PDTool orchestration</u>

ExecutePDTool.bat resources/properties/LabPD-Deploy.properties "" ""

<u>example 2: PDTool VCS workspace initialization</u>

vcsInitWorkspace user password

<u>example 3:   Generate Server Attributes</u>

generateServerAttributesXML localhost "/" resources/modules/getServerAttributeModule.xml resources/modules/servers.xml "READ_WRITE"

<u>example 4: Update Server Attributes</u>

updateServerAttributes localhost sa1 resources/modules/ServerAttributeModule.xml resources/modules/servers.xml

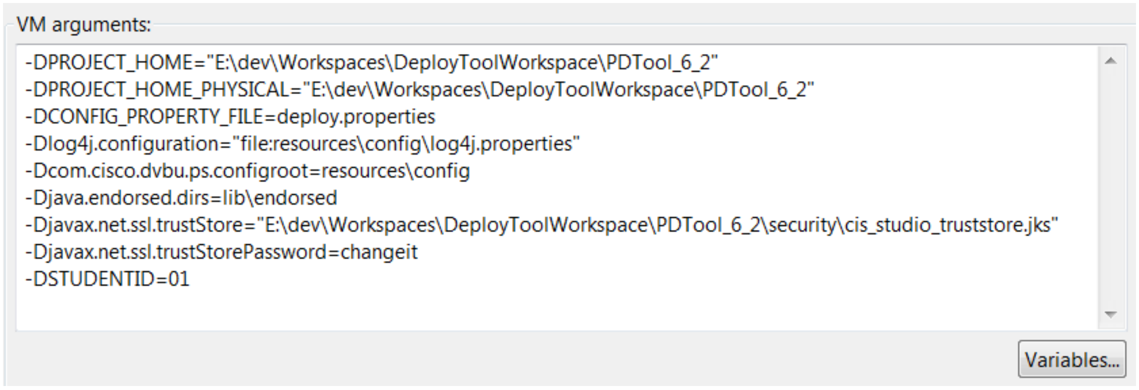
Paste the command into the window

Test different methods by pasting in different method signatures

3) Arguments Tab --> Set the VM Arguments:

This is static for all methods:

<u>For HTTP or HTTPS Connection:</u>



VM arguments:

```
-DPROJECT_HOME="E:\dev\Workspaces\DeployToolWorkspace\PDTool_6_2"
-DPROJECT_HOME_PHYSICAL="E:\dev\Workspaces\DeployToolWorkspace\PDTool_6_2"
-DCONFIG_PROPERTY_FILE=deploy.properties
-Dlog4j.configuration="file:resources\config\log4j.properties"
-Dcom.cisco.dvbu.ps.configroot=resources\config
-Djava.endorsed.dirs=lib\endorsed
-Djavax.net.ssl.trustStore="E:\dev\Workspaces\DeployToolWorkspace\PDTool_6_2\security\cis_studio_truststore.jks"
-Djavax.net.ssl.trustStorePassword=changeit
-DSTUDENTID=01
```

Variables...

-DPROJECT_HOME=<**PDTool-project-path**>
-DPROJECT_HOME_PHYSICAL=<**PDTool-project-path**>
-DCONFIG_PROPERTY_FILE=deploy.properties
-Dlog4j.configuration="file:resources\config\log4j.properties"
-Dcom.tibco.ps.configroot=resources\config
-Djava.endorsed.dirs=lib\endorsed
-Djavax.net.ssl.trustStore="<**PDTool-project-path**>\security\cis_studio_truststore.jks"
-Djavax.net.ssl.trustStorePassword=changeit
-DSTUDENTID=01

These lines are absolutely required for HTTPS but have no impact when using HTTP:

-Djavax.net.ssl.trustStore="<**PDTool-project-path**>\security\cis_studio_truststore.jks"
-Djavax.net.ssl.trustStorePassword=changeit

Note: If your Composite Server has been configured with a strong trust store then the following line should be used:

-Djavax.net.ssl.trustStore="<**PDTool-project-path**>\security\cis_studio_strong_truststore.jks"

When debugging code line

Set PROJECT_HOME to point to your PDTool directory.
Paste into the window
Click apply
Close
Paste into the window
Click apply
Close

4) Click on the Debug perspective to start
Set break points in the code

Click on the Debug icon.

6 Integrating a New VCS Type

In this section, you will learn how to integrate a new VCS type into PDTool and PDToolStudio. The first step is the assessment. Prior to coding it is important to assess the feasibility of integrating the VCS tool. Perform the assessment first and get help from the customer who is asking for the new VCS tool to be integrated. The customer will know the commands and can provide guidance.

Preparing

In preparation for integrating a new VCS type the following should be taken into consideration:

1. Determine a good 2-3 letter acronym for your VCS Server you are integrating to – you will use this throughout the scripts. For demonstration purposes “**ABC**” and “**abc**” will be used generically throughout this document.
2. A new VCS LifecycleListener will be created such as ABCLifecycleListener.java
3. The VCS Module implementation file “VCSManagerImpl.java” will be modified.
4. New sample configuration property files for PDTool (deploy.properties) and PDTool Studio (studio.properties) will be created.
5. Research the new VCS and understand how to perform the following commands:
    a. Refer to the “**Assessment**” below for a more detailed overview of what is required.
    b. Link a directory to the VCS repository
    c. Check-out a project folder from the VCS Repository to the local workspace
    d. Add, Delete, Commit or Submit commands
    e. Understand how usernames and passwords are passed
    f. Understand what VCS command line options need to be set
    g. Understand what Environment variables need to be set
    h. Understand how to construct the URL to talk to the VCS Server
    i. Understand if it requires any editors to pop up (you won’t want this on UNIX).
6. Install the required version of the VCS on both the Windows and Linux.
    a. Install the VCS server and command line on the Composite Linux machine
    b. Install the command line client on your Windows machine.

Assessment

It is recommend that the PS Consultant integrating the new VCS type extract this section of the document and send to the customer.   The customer is going to know the commands for the new VCS type much better than we are. The customer can help reduce the amount of time it takes for the initial research.   This also gives us a chance to engage with the customer as a partner in this integration work.

When adding a new VCS type/tool to PDTool, the following analysis must be performed to determine feasibility of integrating with PTDool and PDTool Studio:

1) Command Line interface (not GUI):

    a. Does the VCS tool have a command line interface similar to subversion? Many tools are patterned after subversion so it is a good baseline to start with.

    Candidate answer:

2) VCS Command:

    a. Determine the VCS base command for these operations for both Windows and UNIX.

    Candidate answer: Windows: ?, Unix: ?

3) Directory Execution Context:

    a. Need to know if the command can be executed from a top level directory and reference the folder or if the command needs to be executed from within the folder when checking out a resource.

        i. Checkin and checkout is performed on file system objects patterned after CIS folders and resources. Given a root folder such as cis_objects, it would contain /services/databases, /services/webservices and /shared. All CIS objects in CIS would be exported out and live in the file system for purposes of checkin/checkout.

        ii. The questions remains, what strategy does the VCS tool take when performing checkin and checkout. Can it execute from a top level folder such as cis_objects and simply reference the folder structure below it or does it have to actually cd into the folder structure in order to perform its command.

    Candidate answer:

4) VCS URL:

    a. Need to know how the URL is constructed to access the VCS server.

b. Need to know if the VCS server can be access via http and/or https.

Candidate answer:

5) VCS Authorization:

a. Need to know what authorization is required to access the VCS server.

b. Need to know how the user name and password is specified.

c. Need to know if credentials can be cached in the local file system so that no username and passwords has to be passed in.

Candidate answer:

6) VCS Environment Variables:

a. Need to know if there are any environment variables that need to be set prior to execution.

Candidate answer:

7) VCS Options:

a. Need to know what VCS options are relevant as general options and not specific to any command that should be included when executing the VCS command.

Candidate answer:

8) LifeCycleListener:

a. Determine the commands to execute, ADD, DELETE and COMMIT.

```
SVN Example:
//private static final String SVN = (File.separatorChar=='/')?"svn":"svn.exe";
private static final String SVN = VCS_EXEC;
private static final String[] SVN_ADD    = new String[] { SVN, "add", ""};
private static final String[] SVN_DELETE = new String[] { SVN, "delete", ""};
private static final String[] SVN_COMMIT = new String[] { SVN, "commit", "-m 
Autocommitting_preamble", ""};
```

Candidate answer:

```
//private static final String ABC = (File.separatorChar=='/')?"xxx":"xxx.exe";
private static final String ABC = VCS_EXEC;
private static final String[] ABC_ADD    = new String[] { ABC, "???", ""};
private static final String[] ABC_DELETE = new String[] { ABC, "???", ""};
private static final String[] ABC_COMMIT = new String[] { SVN, "???", "-m 
Autocommitting_preamble", ""};
```

b. Determine what list of messages sent to standard out should be ignored

```
SVN Example:
while((line = sr.readLine()) != null) {
        if (!line.contains("svn: warning")) result.append(line).append(LS);
}
```

Candidate answer:

9) Workspace/Repository Initialization:
    a. Link a directory to a central repository URL

```
SVN Example:
Link the VCS Repository URL and Project Root to the local workspace
svn import -m "linking workspace to the VCS repository" .
"${VCS_REPOSITORY_URL}/${VCS_PROJECT_ROOT}" ${SVN_OPTIONS} ${SVN_AUTH}
${VCS_WORKSPACE_INIT_LINK_OPTIONS}
```

Candidate answer:

    b. Checkout all resources from the central repository into the local repository

```
SVN Example:
Check out the repository to the local workspace
svn co "${VCS_REPOSITORY_URL}/${VCS_PROJECT_ROOT}" ${SVN_OPTIONS} ${SVN_AUTH}
${VCS_WORKSPACE_INIT_GET_OPTIONS}
```

Candidate answer:

10) VCS Check-in:

Typically this is a commit command or equivalent

Functionality required:

a. Check in a revision on a **folder** [HEAD or a specific revision number]
    a. What is the option to supply a comment or message?
    b. Can the full path be provided?
    c. What options are required?
    d. Can authorization be provided on command line?

```
SVN Example:
String fullResourcePath = (execFromDir+"/"+resourcePath).replaceAll("//", "/");
svn commit ${fullResourcePath} -m "${Message}" ${SVN_AUTH} ${VCS_OPTIONS}
${VCS_CHECKIN_OPTIONS}
```

Candidate answer:

b. Check out a revision on a **file** (resource) [HEAD or a specific revision number]

```
SVN Example:
String fullResourcePath =
(execFromDir+"/"+resourcePath+"_"+resourceType+".cmf").replaceAll("//", "/");
svn commit ${fullResourcePath} -m "${Message}"  ${SVN_AUTH} ${VCS_OPTIONS}
${VCS_CHECKIN_OPTIONS}
```

Candidate answer:

11) VCS Check-out:

Typically this is an "update" or "get" command or equivalent. Need to be able to update to HEAD or a revision.

Functionality required:

a. Check out a revision on a **folder** [HEAD or a specific revision number]
    a. What is the command to get the latest version [HEAD]?
    b. Can the full path be provided?
    c. What options are required?
    d. Can authorization be provided on command line?

```
SVN Example:
String fullResourcePath = (execFromDir+"/"+resourcePath).replaceAll("//", "/");
svn update ${fullResourcePath} -r ${Revision} ${SVN_AUTH} ${VCS_OPTIONS}
${VCS_CHECKOUT_OPTIONS}
```

Candidate answer:

b. Check out a revision on a **file** (resource) [HEAD or a specific revision number]

```
SVN Example:
String fullResourcePath =
(execFromDir+"/"+resourcePath+"_"+resourceType+".cmf").replaceAll("//", "/");
svn update ${fullResourcePath} -r ${Revision} ${SVN_AUTH} ${VCS_OPTIONS}
${VCS_CHECKOUT_OPTIONS}
```

Candidate answer:

c. Check out using a **tag** or **label**

```
SVN Example:
Not currently integrated.
```

Candidate answer:

Creating the VCS LifecycleListener

Currently there are three LifeCycleListeners implement which include:

- CVSLifecycleListener – **package** com.tibco.cmdline.vcs.spi.cvs
- P4LifecycleListener and – **package** com.tibco.cmdline.vcs.spi.p4
- SVNLifecycleListener – **package** com.tibco.cmdline.vcs.spi.svn

Step 1: Create a new Package → com.tibco.cmdline.vcs.spi.abc

Step 2: Select a LifecycleListener (e.g. SVNLifecycleListener), copy and paste into the new package.

Step 3: Rename the source file – Refactor → Rename SVNLifecycleListener.java to ABCLifecycleListener.java. This will rename the class and the method to your new name

Step 4: Modify ABCLifecycleListener

- Modify the variables containing the VCS commands for your version control system as needed

```
// VCS_EXEC contains the full path and command.  It is not required to be in the path.
private static final String SVN = VCS_EXEC;

private static final String[] SVN_ADD    = new String[] { SVN, "add", ""};
private static final String[] SVN_DELETE = new String[] { SVN, "delete", ""};
private static final String[] SVN_COMMIT = new String[] { SVN, "commit", "-m
Autocommitting_preamble", ""};
```

- Modify references to the VCS variables in the "handle()" method

```
    public void handle(File file, Event event, Mode mode, boolean verbose) throws
VCSException {
        switch(event) {
            case CREATE:
                switch (mode) {
                    case POST:
                        handle(file, SVN_ADD, 2, verbose);

System.out.println("CALLBACK: " + file.getPath());

                        break;
                    default: // do nothing
                }
                break;

            case DELETE:
                switch (mode) {
                    case PRE:
                        handle(file, SVN_DELETE, 2, verbose);
                        break;
                    default: // do nothing
                }
                break;
            default:  // do nothing
        }
    }

    public void checkinPreambleFolder(File file, boolean verbose) throws VCSException {
        handle(file, SVN_COMMIT, 3, verbose);
    }
```

- Modify the method **protected** String getErrorMessages as needed to check for error messages and warnings returned from your version control system. Depending on the VCS you are using, this may or may not be required.

```
protected String getErrorMessages(Process process) throws VCSException {
 StringBuilder result = new StringBuilder();

 BufferedReader sr = new BufferedReader(new InputStreamReader(process.getErrorStream()));
    try {
        String line = null;
        while((line = sr.readLine()) != null) {
//ADD LINES HERE TO CATCH WARNINGS AND ERRORS
            if (!line.contains("svn: warning")) result.append(line).append(LS);
        }
    }
    catch(IOException e) {
        throw new VCSException(e);
    }
    finally {
        try { sr.close(); } catch(IOException e) {throw new VCSException(e);}
    }
    return result.toString();
}
```

Modifying VCSManagerImpl.java

There are three sections that need modifying which include. Below is the roadmap on what needs to be modified. The best approach is to copy one of the sections in the same method like SVN, paste it into the NEW_VCS type. Modify the code according to the commands of your target VCS Server.

1. Workspace Initialization

```
// Initialize the VCS workspace
private void vcsInitWorkspaceCommon()throws CompositeException {

        ... code removed

        /*********************************************************
         * INIT VCS WORKSPACE FOR "NEW VCS TYPE" [NEW_VCS_TYPE]
         *********************************************************/
        if (vcsStruct.getVcsType().equalsIgnoreCase("NEW_VCS_TYPE")) {
                //Implement Initialization for NEW VCS TYPE HERE
        }
}
```

2. VCS Check-in

```
//*********************************************************************
// Execute a VCS Generalized Scripts  ** vcs_checkin_checkout_cvs vcs_checkin
//*********************************************************************
private void vcs_checkin_checkout__vcs_checkin() throws CompositeException {

        ... code removed

        /*******************************************
         * NEW_VCS_TYPE=New VCS Type
         *******************************************/
        if (vcsStruct.getVcsType().equalsIgnoreCase("NEW_VCS_TYPE")) {
                // Implement VCS Checkin here
        }
}
```

3. VCS Check-out

```
//********************************************************************
// Execute a VCS Generalized Scripts  ** vcs_checkin_checkout_cvs vcs_checkout
//*********************************************************************
private void vcs_checkin_checkout__vcs_checkout() throws CompositeException {

        ... code removed

        /*******************************************
         * NEW_VCS_TYPE=New VCS Type
         *******************************************/
        if (vcsStruct.getVcsType().equalsIgnoreCase("NEW_VCS_TYPE")) {
              // Implement VCS Checkout here
        }
}
```

4. Load VCS Constructor Variables

```
//----------------------------------------------------------------------
// loadVcs: Derived variables
//----------------------------------------------------------------------
private void loadVcs(String prefix, String user, String password) {

      ... code removed

      //----------------------------------------------------------------------
      // loadVcs: New VCS Type (new) specific settings
      //----------------------------------------------------------------------
      if (this.getVcsType().equalsIgnoreCase("NEW_VCS_TYPE")) {
            // Implement loadVCS here
      }
}
```

5. Set VCS Connection Property Environment Variables

```
//----------------------------------------------------------------------
// setVCSConnectionProperties: set environment variables
//----------------------------------------------------------------------
private void setVCSConnectionProperties(String vcsConnectionId, String
pathToVcsXML) {

      ... code removed

      //------------------------------------------------------------
      // setVCSConnectionProperties: New VCS Type (new) specific settings
      //------------------------------------------------------------
      if (vcsType.equalsIgnoreCase("ABC")) {
            oldProp = System.setProperty("ABC_ENV", envList);
      }
```

Creating new Configuration Property Files

In this section, you will learn what needs to be done to create new configuration property file examples. The property files for "deploy.properties" are different than "studio.properties". You will need to select a current example file, copy it and modify it. The following example files currently exist:

PDTool – deploy.properties example files:

- deploy_cvs_[win|unix].properties
- deploy_p4_[win|unix].properties
- deploy_svn_[win|unix].properties

- deploy_tfs_win.properties

PDTool Studio – studio.properties example files:

- studio_cvs_example.properties
- studio_p4_example.properties
- studio_svn_example.properties
- studio_tfs_example.properties

Implementation steps:

Step 1: For PDTool (deploy.properties),

1. select a suitable deploy tool file example
2. Copy the file
3. Paste the file
4. Rename to deploy_abc_example.properties – where "abc" is the 2-3 letter acronym of your chose.
5. Modify the property file using your VCS specific commands, options, and environment variables

Step 2: For PDTool Studio (studio.properties),

1. select a suitable studio property file example
2. Copy the file
3. Paste the file
4. Rename to studio_abc_example.properties – where "abc" is the 2-3 letter acronym of your chose.
5. Modify the property file using your VCS specific commands, options, and environment variables

Testing

You are finally ready for testing. Build the project. Consider first debugging in Eclipse to catch any wayward errors right off the bat. Once you are ready for command-line, utilize the UnitTest-VCSModule.properties for the command-line orchestration and VCSModule.xml for the XML configuration. How to execute:

Initialize Repository: ExecutePDTool.bat -vcsinit vcsuser vcspassword

Execute Commands: ExecutePDTool.bat -exec ..\resources\properties\UnitTest-VCSModule.properties

7 Conclusion

Concluding Remarks

The Promotion and Deployment Tool is a set of pre-built modules intended to provide a turn-key experience for promoting CIS resources from one CIS instance to another. The user only requires system administration skills to operate and support. The code is transparent to operations engineers resulting in better supportability. It is easy for users to swap in different implementations of a module using the Spring framework and configuration files.

How you can help!

Build a module and donate the code back to Tibco Professional Services for the advancement of the “***Promotion and Deployment Tool***”.