Oracle Utilities Application Framework Software Development Kit

Contents

Chapter 1. Oracle Utilities Application Framework Software Development Kit

Welcome to the Oracle Utilities Application Framework Software Development Kit.

The Oracle Utilities Software Development Kit is a set of utilities designed to build applications based on Oracle Utilities Application Framework, the application framework built by Oracle. It provides utilities for implementers to extend applications without compromising upgradeability. This document describes the Software Development Kit.

This document is divided into the following parts:

- The **User Guide** describes how to use the Software Development Kit to customize products.
- The **Developer Guide** presents information that aid the development process including technical references and standards.
- The **Packaging Guide** describes the procedures for taking developed code and data to the target environments.

User Guide

Overview

The Oracle Utilities Software Development Kit is a set of utilities designed to build applications based on Oracle Utilities Application Framework, the application framework built by Oracle. It provides utilities for base product developers and implementers to extend OUAF applications without compromising upgradeability. This document discusses the details of application development using Software Development Kit, including:

- The **Development Environment** section describes the environment that developers work on while using the Software Development Kit.
- The **Build Server** section describes the procedure for setting up a build server.
- The **Technical Architecture** section describes applications developed on framework. It describes the framework technical architecture at a high level and then describes its components in detail.
- The **Meta-data** is the core component of applications built on framework. The meta-data section describes the purpose, structure, and use of the meta-data tables.
- The **Development Process** section contains high level, quick reference guides on common tasks in building applications based on framework.

- The **Cookbook** section describes the development tasks in detail. The Development Process section contains links to specific sections in this section.
- The **Utilities** section describes the tools provided with Software Development Kit. These tools include batch programs and Perl scripts developed to automate several stages of the development process.

Converted COBOL Programs

Important:
As of Oracle Utilities Application Framework Release 4.3.0.0.0, all COBOL programs have been converted to Java. This version of the SDK therefore does *not* require a COBOL runtime or compiler. The term COBOL is still used to refer to metadata in some places, but in those and any other case, "COBOL" in this document implicitly means "Converted COBOL Program".

Development Environment

Overview

The App Server is the Development Environment

The Software Development Kit development environment is built on a standard app server install of the product being customized. Put another way, the app server is the development environment.

Source code is written and generated within the app server directory structure and executables are generated where the app server expects them and are therefore ready to be executed.

For example:

- UI code is written directly where the app server looks for them.
- The jar file for the Java programs is created directly where the app server looks for it.

Development App Server is Local, Not Shared

Each developer has a development app server in his workstation for each project. This means that a developer can code and unit test all within his workstation. This also means work-in-progress code contained within the developer's workstation.

Repository for Project

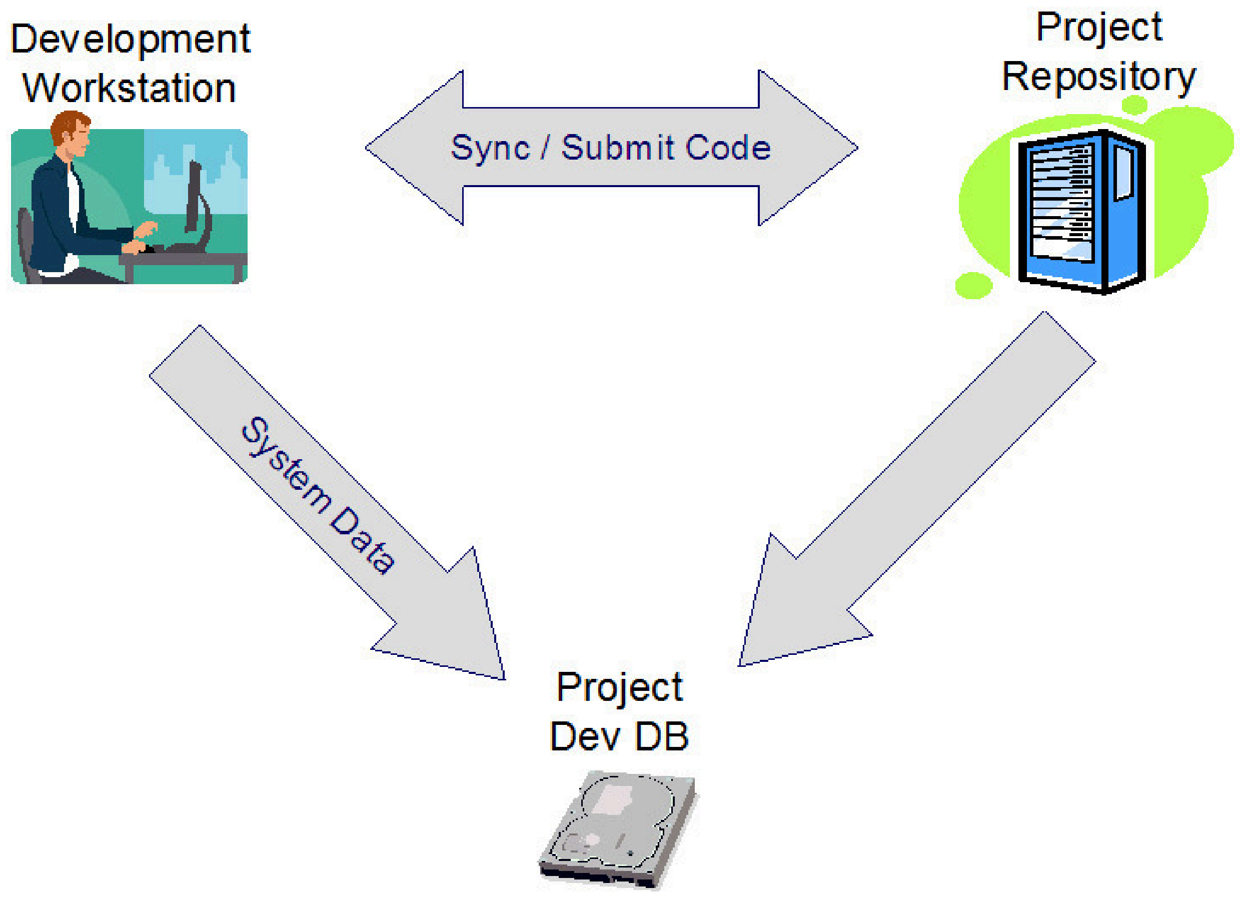
All finished code is submitted into the project repository. As such, developers synchronize with the project repository to get their local development environments current with the rest of the team.

The project repository is also set up as a development environment. When developers synchronize with the project repository, they get a development environment including configuration necessary for that project.

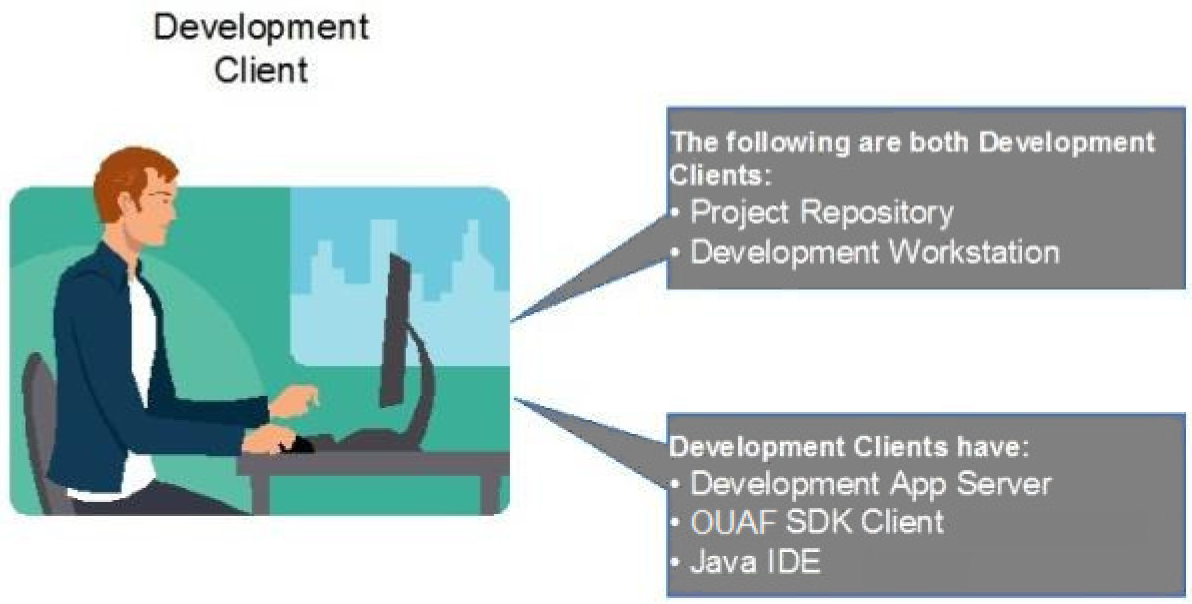
Components of the Software Development Kit

The following diagram illustrates the development environment.

Note:
Please see the installation guide for instructions on how to set up the Software Development Kit and its components.



Development Environment


Development Client
The following are both Development Clients:
• Project Repository
• Development Workstation
Development Clients have:
• Development App Server
• OUAF SDK Client
• Java IDE

Development Client

Project Development Database

Each project has a development database. This is a regular database install of the product that is being customized. System data for customizations are stored in this database. Development processes like code generation connect to this database. In addition, development app servers in development workstations connect to this database.

Project Repository

The project repository serves the following purposes:

- It is the central storage for all completed, unit-tested code.
- It provides the environment from which to build the latest state of the project.
- It provides the latest state of the project dev app server from which all developers can synchronize with.
- It is the source for CM Packaging.

  To support these purposes:

- It has to be accessible to all developers.
- It is set up as a development client, e.g., similar to a development workstation (see Development Workstation below).

Development Workstation

Developers write, generate, compile, and test code on development workstations. A development client is installed for each project that the developer works on.

The main components of a development client are the following:

- **Project Dev App Server**. Code is developed on and executables built into the project dev app server.
- **Software Development Kit Client**. This is the primary development tool of the Software Development Kit.
- **Eclipse SDK**. This is the Java development tool used in the Software Development Kit.

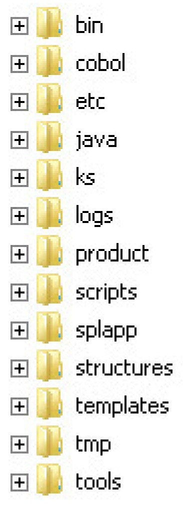
Directory Structure

The App Server Directory

As mentioned earlier, the app server is the development environment. Source code and executables are therefore placed within the directory structure of the app server.

Standard App Server Directory Structure

A typical application server will be installed with a directory structure similar to this:

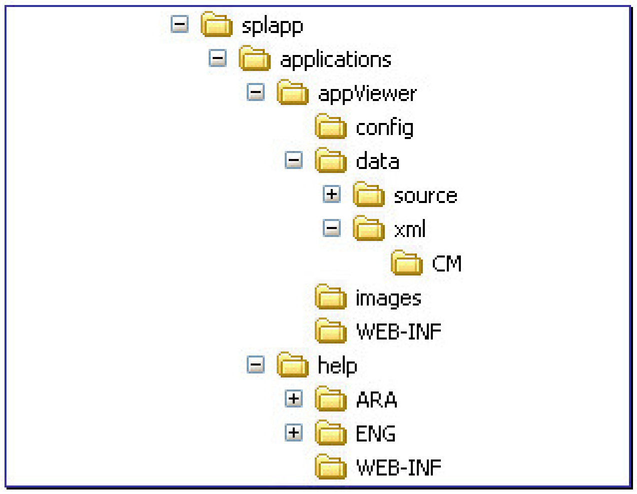


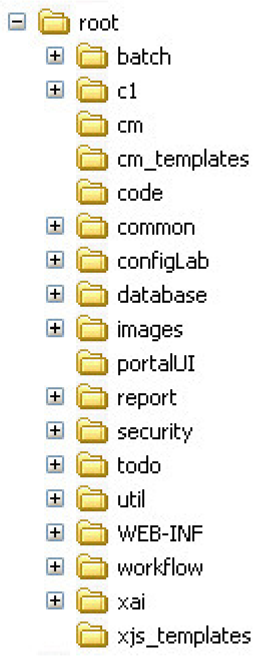
```
bin
cobol
etc
java
ks
logs
product
scripts
splapp
structures
templates
tmp
tools
```

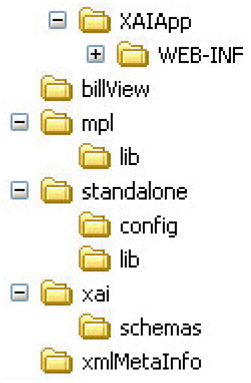
Within this structure only some of the subdirectories are of interest to a developer.

The `cobol` directory may still exist for some applications (e.g., CCB), but that is now obsolete as all COBOL programs have been converted to Java for edge applications running on OUAF 4.5.0.x.

The `splapp` directory contains all the main application files, including the deployed custom jars (`cm.jar`) and Web server files (e.g. JSP, Javascript).



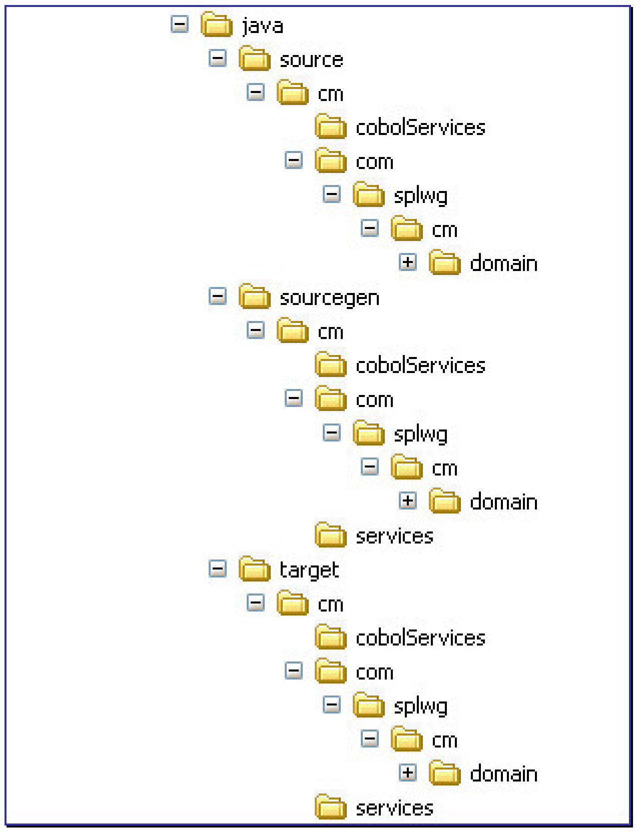




Additional Directories for Development

Java

A directory structure with a base directory of **java** is used for Java development, as shown in the following image.

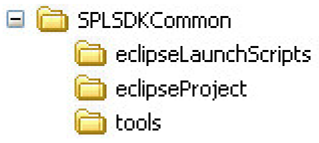


- **source** contains the code that the developer writes or generates that is submitted to the repository. Under this is the **com.splwg.cm.domain** folder, which contains the CM Java source code.
- **sourcegen** contains generated code that is necessary to build the project. All files in this structure are generated and therefore must not be modified manually in any way.
- **target** contains the runtime files created from **source** and **sourcegen**. The content of this directory is what is deployed as a jar file to the app server. All files in this structure are generated and therefore must not be modified manually in any way.
- The **cobolServices** folder contains any Converted COBOL service XML mapping files.

Project Configuration Information

Project information is stored in the SPLSDKCommon directory structure.

- **eclipseLaunchScripts** contains the Eclipse launch scripts for various tools.
- **eclipseProject** contains the project configuration information for Eclipse.
- **tools** contains the tools that are required for the project.



Pertinent Directories in the App Server

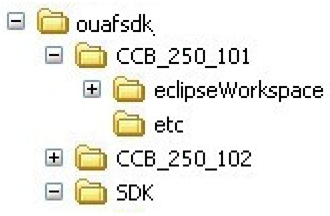
Item	Directory	Content
Java sources	java\source\cm\com\splwg\cm.domain	Java source code.
Web Application	splapp\applications\root\cm	UI code.
	splapp\applications\root\WEB-INF\lib	cm.jar file is deployed here.
Standalone Application	splapp\standalone\\lib	CM?*.jar file. cm.jar file is deployed here.

Item	Directory	Content
XAI Schemas	splapp\xai\schemas\CM*.xml	XAI schemas.
Service XML files	splapp\xmlMetaInfo\CM*.xml	Service XMLs for Java.
Eclipse Launch Scripts	SPLSDKCommon\eclipseLaunchScripts	Eclipse launch scripts.
Eclipse Project	SPLSDKCommon\eclipseProject	Project configuration information for Eclipse.
Software Development Kit Tools	SPLSDKCommon\tools	Tools required by the Software Development Kit.

Client Directory

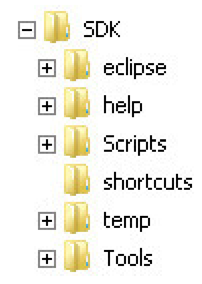
The Oracle Utilities Application Framework (OUAF) Software Development Kit client directory contains both the Software Development Kit itself and some project-specific information such as the Eclipse workspace.

The location of the Software Development Kit client is stored in the environment variable **SPLSDKROOT**.

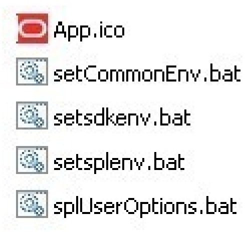


The Software Development Kit Client

The Software Development Kit client is installed in `SDK/<version>`.



The SDK folder has the following scripts at the top level:



Additionally, a copy of Eclipse is installed later into the `eclipse` directory.

Note:
Updates are unique versions of the Software Development Kit and therefore have their own directories.

Note:
A separate copy of Eclipse is installed per version of the Software Development Kit client because each version may have its own set of plug-ins and the plug-ins must be in the plugins directory of Eclipse.

All of these files are not meant to be executed directly by the developer and are intended to be executed through scripts in the **shortcuts** folder.

The `setsdkenv.bat` script specifies the OUAF SDK version version and installation folder.

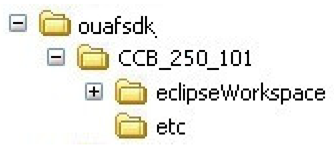
The `setplenv.bat` scriptsets up the environment variables needed to correctly run a development environment for the application server.

Note:
A copy of Eclipse is installed later into the `eclipse` directory. This folder is created when the user first runs the **startEclipse** shortcut script. A separate copy of Eclipse is installed per version

of the Software Development Kit client because each version may have its own set of plug-ins, and the plug-ins must be in the `plugins` directory of Eclipse.

Project Directories

Each project has its own directory.

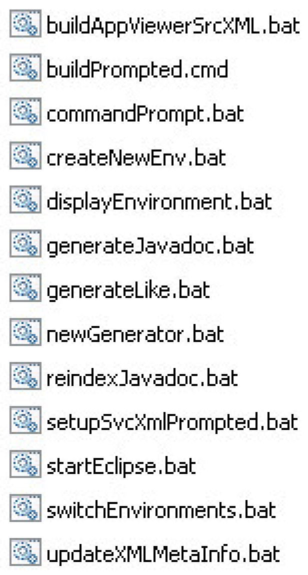


- **eclipseWorkspace** contains Eclipse workspace files.
- **etc** contains additional project-related files, including `setsplenv.bat`, which is used to set environment variables for the project. This script is executed before other scripts so that succeeding scripts operate on the project.
- **eclipseProject** (not shown above) exists only in development workstations (not in the project repository). It is a copy of the same directory in the app server.

Note:
setsplenv.bat is generated when the project is created/configured.

Shortcuts Directory

The shortcuts directory contains various scripts used in development.



buildAppViewerSrcXML.bat
buildPrompted.cmd
commandPrompt.bat
createNewEnv.bat
displayEnvironment.bat
generateJavadoc.bat
generateLike.bat
newGenerator.bat
reindexJavadoc.bat
setupSvcXmlPrompted.bat
startEclipse.bat
switchEnvironments.bat
updateXMLMetaInfo.bat

Except for `commandPrompt.bat`, `createNewEnv.bat`, and `startEclipse.bat`, all of these script are intended to be executed on the command line (see `commandPrompt.bat`).

The `commandPrompt.bat` script initializes the appropriate environment variables for a development environment so that OUAF SDK scripts, particularly those in the **shortcut** directory, can be executed on the Windows command prompt.

The `createNewEnv.bat` script allows a developer to set up another development environment for another application server in the workstation. The initial development environment is first created on installation of the OUAF SDK.

The `startEclipse.bat` script installs the Eclipse IDE and sets up the OUAF plugin in Eclipse if these are not yet installed. Otherwise, it initializes the appropriate environment variables for a development environment that the OUAF Eclipse plugin requires.

Synchronizing with the Project Repository

Developers synchronize the whole of the app server directory except for the following:

- java\sourcegen
- java\target
- splapp\xmlMetaInfo
- splapp\applications\root\WEB-INF\lib\cm.jar
- splapp\standalone\lib\cm.jar

- splapp\XAIApp\WEB-INF\lib\cm.jar
- logs\system

Versions

Version Number

The Software Development Kit version number comprises five period-delimited segments. The first four segments indicate the Framework (FW) version number, and the fifth specifies the SDK update number

The SDK update number starts with 1 (and increments by one) for each FW version. For example, an SDK version of 4.3.0.0.1 means that it is the first first SDK developed for FW 4.3.0.0, whereas 4.3.0.1.1 would be the first SDK version for FW 4.3.0 Service Pack 1.

Compatibility with Products

Generally, unless noted otherwise, an OUAF SDK product should only be used for its intended FW version. That general rule applies to all previous versions of the Oracle Utilities Framework SDK.

For example, OUAF SDK FW 4.2.0.2.4 is not compatible with OUAF SDK 4.3.0.0.1, and OUAF SDK 4.3.0.0.1 and FW 4.3.0.1.1 are never compatible.

Updates

Each new version of the OUAF SDK installs to its own folder, allowing you to use several version of the SDK on the same workstation. An OUAF SDK installation should never be used to overwrite an existing SDK installation with the intent of "upgrading" it. You should only develop using the version of the OUAF SDK that runs the same OUAF version that your application server runs.

New versions of the OUAF SDK can be installed and old versions uninstalled without affecting the application server. These options allow you to migrate your application servers to a newer version of the OUAF while allowing you to still refer to the original application server setup.

Moving Up to a New Update

Since an update of the OUAF SDK installs to its own folder, a new development workspace for your application server must be created as well. If it has not yet been created by the OUAF SDK ClientSetup program, use the `createNewEnv.bat` shortcut script to create one for your application server.

You will also need to install Eclipse through the `startEclipse.bat` shortcut script and create a new Eclipse project for your project. Since the CM Java sources are already present in the application server, you need only re-run the **Generate Artifact** launch configuration in Eclipse to generate code before you rebuild your project.

You should *not* run the Generate Artifact launch configuration in the Eclipse installation of the older OUAF SDK installation at this point.

Moving Up to a New Version of a Product

Moving to a new version of a product requires creating a new development environment suited for the new version. A new version is likely to be built on a new version of the Framework, which would mean that a new compatible version of the Software Development Kit is required.

The steps are as follows:

1. **Stabilize the project on the old version of the product.** Ensure that the project is in a stable state and that all developers have submitted all code to the repository.
2. **Prepare the database for the new project:**
    a. Copy the database of the project in the old version to a new database.
    b. Upgrade this newly created database to the new version of the product by following the database upgrade procedures of the product.
3. **Set up the repository for the new project:**
    a. Prepare a project repository as described in the installation documentation.
    b. Copy source code from the repository of the previous version into the project repository.
    c. Update code, if necessary, as specified in the documentation of the new version of the product.
    d. Build the entire project. This includes generation of code, compilation, generation of services, etc.
    e. Test the customizations.
4. **Set up development workstations.** At this point, developers can set up their workstations for the new project on the new version of the product. Each developer must follow the workstation setup procedure.

Product Single Fixes

When single fixes to products are released, the following should be done for projects for which the single fixes are required:

- Stabilize the project by making sure that the project is in a stable state and that all developers have submitted all code to the repository.
- Apply the single fix to the project repository.
- Each developer of the project must then synchronize with the project repository.

Build Server

Every enterprise has its own software development practices that cover how developers update code, how changes are tracked and tested and how new releases are created. We generally expect that whatever practices have historically worked within an organization will continue to work for the implementation of this application. However, a build methodology was developed that has worked well for managing concurrent changes to the application that is based on the following principles:

- All of the application should work all of the time. Therefore, changing one small part of the application requires that all of the application be retested.
- Bugs are more expensive to fix the longer they stay in a system. This principle has been proven time and time again in software engineering. This truth mostly owes to the fact that it is easiest to find the offending developer immediately after he or she broke the system and also that developer has less recollection of how and where the system was broken as time goes by.
- In a complex system malfunctions can occur "far away" from the points of code modification. It is unreliable to expect selective retesting based on what was *likely* to malfunction to find the all the places of actual malfunction.

Tailoring Your Oracle Utilities Application Implementation

This document describes the naming conventions and processes that must be followed to ensure a successful upgrade of the Oracle Utilities application base product release-on-release. The implementation team responsible for tailoring the Oracle Utilities application to meet specific customer needs must follow this guide to preserve their changes and ensure successful upgrades. Only the changes described in this document are considered as permitted for the tailoring of the base product. Any changes that do not conform to these rules may be overridden by the install utility during a base product upgrade.

Some naming conventions used in this document:

- `$SPLEBASE` (for UNIX) and `%SPLEBASE%` (for Windows) is the generic Oracle Utilities environment directory name.
- `$SPLENVIRON` (for UNIX) and `%SPLEBASE%` (for Windows) is the generic Oracle Utilities environment name.
- `$SPLDB` (for UNIX) and `%SPLDB%` (for Windows) is the database type.

Preserving Customer Changes

For any kind of a customer modification, the file's directory structure and naming conventions are defined in this section. The implementation team must follow these conventions to preserve the results of their work during a subsequent base product upgrade.

- The configuration parameters of the environment being upgraded are displayed (as default parameters) during the configuration stage of the install process. These parameters may be changed if new settings are preferred.
- The base product is shipped with examples of different kinds of modules that may be used by implementation teams. The examples can be found in the following directories:
    - `$SPLEBASE/splapp/applications/root/cm_templates` contains Oracle Utilities Application Framework Web file examples.
    - `$SPLEBASE/splapp/applications/root/<application product code>/cm_templates`. This directory contains Oracle Utilities application product Web file examples. The <application product code> varies by product; for example, the Oracle Utilities Customer Care and Billing, the <application product code> is `c1`.
    - `$SPLEBASE/scripts/cm_examples`. For batch script examples, this directory has two subdirectories: `FW` for Oracle Utilities Application Framework examples, and <application product code> for Oracle Utilities application product examples (e.g., `CCB` for Oracle Utilities Customer Care and Billing, `TAX` for Oracle Public Sector Revenue Management).

Note:
For simplicity, this document generally uses UNIX platform naming conventions. To apply these names to the Windows platform, use the Windows naming conventions "%" sign instead of the "$" sign, and backslashes ("\") instead of forward slashes ("/") as directory separators (e.g., `%SPLEBASE%\splapp\applications\root\cm_templates`).

Tailoring Web Files

Base product UI files are located in the directory `$SPLEBASE/splapp/applications/root`. Implementers may develop their own UI files under the directory `$SPLEBASE/splapp/applications/root/cm`. No specific naming conventions are enforced under this directory.

The root directory may be deployed in war file format for runtime environment (`SPLApp.war`). Use provided utilities to incorporate your **cm** directory into the `SPLApp.war` file.

Tailoring the CM Java Application

Implementers may write their own Java classes to extend the Oracle Utilities application functionality. All Java files should belong to the `com.splwg.cm` package. The CM Java application should be compiled into a jar file named `cm.jar`. The SDK Customer Modification packaging utilities will help build this file. The `cm.jar` is typically deployed into the following directories:

```
$SPLEBASE/splapp/applications/root/WEB-INF/lib

$SPLEBASE/splapp/applications/XAIApp/WEB-INF/lib

$SPLEBASE/splapp/businessapp/lib

$SPLEBASE/splapp/standalone/lib
```

Additional third-party jar files can be deployed by following the `cm*.jar` naming standard. Customers may use this option to deploy any additional functionality, interfaces with other applications, and so on. These will not be built by the SDK Customer Modification packaging utilities, but will be deployed into the application once it is supplied in jar format.

The root directory may be deployed in a war file format for the runtime environment (`SPLApp.war`). Use the provided utilities to incorporate your `/cm` directory into `SPLApp.war` file.

Important:
All `cm*.jar` files that need to be applied must be defined in `$SPLEBASE/structures/cm_jars_structure.xml`. If the file does not exist in the target environment, the sample `cm_jars_structure.xml.example` file can be copied from the SDK packaging's `/etc` folder.

Manual cm.jar deployment

The `cm.jar` file is usually deployed as part of the CM packaging process (`extractCMsource`, `applyCM`, `create_CM_release`, etc.), but in some cases it may be desirable to manually deploy the `cm.jar` file to one or more target environments.

CAUTION:
This should be done with care and should only be considered if the `cm.jar` components are self-contained and have no external dependencies.

To manually deploy `cm.jar`:

1. The SPLEBASE/structures/cm_jars_structure.xml must exist and should have at least the following:

   ```
   <?xml version="1.0" encoding="UTF-8"?>
   <jar_structure>
      <cm.jar>
     <source_dir_jar>@SPLEBASE@/etc/lib</source_dir_jar>
     <dest_folders>
    <dest_folder_1>@SPLEBASE@/splapp/applications/XAIApp/WEB-INF/lib</dest_folder_1>
    <dest_folder_2>@SPLEBASE@/splapp/applications/root/WEB-INF/lib</dest_folder_2>
    <dest_folder_3>@SPLEBASE@/splapp/businessapp/lib</dest_folder_3>
    <dest_folder_4>@SPLEBASE@/splapp/standalone/lib</dest_folder_4>
     </dest_folders>
     <child_jvm_path>@SPLEBASE@/splapp/standalone/lib</child_jvm_path>
      </cm.jar>
   </jar_structure>
   ```

   The <cm.jar> element identifies the jar file name, usually `cm.jar`, as defined here.

   Element <source_dir_jar> defines the source location of the abovementioned jar. The directory in the example above should work for most cases.

   The `dest_folder_` *n* elements point to the target locations where the jar will be placed. The directories in this example should work for all.

2. Manually copy the `cm.jar` to the directory specified in the `<source_dir_jar>` element, typically `$SPLEBASE/etc/lib`.

3. Run initialSetup.sh (or .bat on Windows) to do the rest. This will copy the `cm.jar` to the specified target locations and rebuild the war and ear files.

Positioning Custom Scripts

Customers and implementers may put their scripts under the directory `$SPLEBASE/scripts/cm`.

Replacing the Oracle Utilities Logo

Customers may want to replace the Oracle Utilities logo image on the Main menu with another logo image. To do this, put the logo <customer_logo_file>.gif file into the directory `$SPLEBASE/etc/conf/root/cm` and create a new "External" Navigation Key called **CM_logoImage**.

To replace the logo, run the Oracle Utilities application from the browser with the parameters:

```
http://<hostname>:<port>/cis.jsp?utilities=true&tools=true
```

From the **Admin** menu, select **Navigation Key**. Add the above Navigation Key with its corresponding **URL Override** path.

The syntax for the URL path is:

For Windows:

```
    http://<host name>:<port>/cm/<customer_logo_file>.gif
```

For UNIX:

```
    http://<host name>:<port>/spl/cm/<customer_logo_file>.gif
```

The root directory may be deployed in war file format for the runtime environment (`SPLApp.war`). Use the provided utilities to incorporate your `cm` directory into the `SPLApp.war` file.

Using the Implementation Version File

Implementers may keep the implementation version number in the `CMVERSION.txt` file in the `$SPLEBASE/etc` directory. This file is preserved by the install utility.

Tailoring XML Schema

Note:
This implementation option is applicable for Oracle Enterprise Taxation Management application only.

Implementers may generate their own XML schemas and store them in the directory `$SPLEBASE/ splapp/ xmlMetaInfo`. The implementation schemas must use the naming convention `CML*.xml`.

Tailoring Templates and User Exits

The templates delivered under the folder `$SPLEBASE/templates` can be overridden by the Application by creating a copy of the template file with the same name but prefixed by "cm.". The cm copy will be customized.

Since the templates can contain user exits (special statements that allow to import external files during the template processing). Those user exits can be overridden by creating a copy of the user exit file with the same name but prefixed by "cm_". The cm copy will be customized.

JUnit testing

JUnit is a Java framework that supports writing unit tests that help ensure your code works as desired, and existing code is not broken by new changes. It is often useful to create JUnit tests during development to verify that your code works as expected, and to keep and rerun the tests in the future to ensure that later changes in your (or someone else's code) don't unexpectedly break your code.

More information on JUnit testing philosophy is available at JUnit.org.

Note:
This document assumes that you use Eclipse. However you can choose to use different IDE but then you have to find how to achieve the equivalent functionality that Eclipse provides.

Assuming you have an existing JUnit test class, you can execute them directly within Eclipse by:

- Right-clicking on the class in Package Explorer

```
Run -> JUnit Test
```

All the tests for an application can be run from Eclipse by running the `com.splwg.AllTests` class in the "test" directory as a JUnit test.

Standard test cases

There are framework classes that are helpful for specific test cases:

Testing Searches

There is a convenient test superclass for search services, `com.splwg.base.api.testers.SearchTestCase`. This test class only requires that you override two methods:

- `String getServiceName()` - this method specifies the service name, eg CILCACCS, for the search
- `List getSearchTrials()` - this method should return a list of `SearchTrials`

A search trial describes information about a particular invocation of a search. You need to describe the inputs (the input fields and the search type), and then describe the expected output for that given input:

- Some expected rows, in the order expected

In order to properly test searches, the expected results is not required to contain every search result- if new rows are added by some other process, they will not cause the test to fail. The search results, however, must contain at least all of the expected results, in the relative order they are added.

- Possibly some prohibited rows, which the search should not find

In addition, there may be times when you want to guarantee that a certain row is definitely NOT found in the search result. This can be accomplished by adding a prohibitedRow, in the same manner as expected rows are added to the trial.

The search test FW will then use inputs from each search trial to execute the search, and compare the expected and prohibited results to the actual search results. It expects to find the expected rows in the order added, and should find all of them. Any different order or missing row results in a failure. What will not result in a test failure is if new rows have been added interspersed throughout the expected rows. These are fine. If a given search result row does not match the next expected result row, it is compared against all of the prohibited rows. If it matches any of them, the test fails.

The search framework will also examine the information about the search, and ensure that each search type (main, alternate, alternate2, ...) is executed at least once.

Here is a sample search test class:

```
package com.splwg.base.domain.batch.batchControl;


import com.splwg.base.api.lookup.SearchTypeLookup;

import com.splwg.base.api.testers.SearchTestCase;

import com.splwg.base.api.testers.SearchTestResult;
import com.splwg.base.api.testers.SearchTrial;


import java.util.ArrayList;

import java.util.List;


/**

 * @author bosorio

 * @version $Revision: #2 $

 */

public class BatchControlSearchService_Test

    extends SearchTestCase {


    //~ Methods ----------------------------------------------------------


    protected String getServiceName() {

        return "CILTBTCS";

    }


    /**

     * @see com.splwg.base.api.testers.SearchTestCase#getSearchTrials()

     */

    protected List getSearchTrials() {

        List list = new ArrayList();


        // Search using Main Criteria

        SearchTrial trial = new SearchTrial("Main search");

        trial.setSearchType(SearchTypeLookup.constants.MAIN);


        trial.addInput(BatchControlSearchService.INPUT_MAIN.BATCH_CD,

              "ADM");

        SearchTestResult expectedResult = trial.newExpectedResult();

        expectedResult.put(BatchControlSearchService.RESULT.BATCH_CD,

              "ADM");

        list.add(trial);


        // Search using Alternate Criteria
        trial = new SearchTrial("Search by description");
        trial.setSearchType(SearchTypeLookup.constants.ALTERNATE);

        trial.addInput(BatchControlSearchService.INPUT_ALT.DESCR,
              "AcCount D");
        expectedResult = trial.newExpectedResult();
        expectedResult.put(BatchControlSearchService.RESULT.BATCH_CD,
              "ADM");
        expectedResult.put(BatchControlSearchService.RESULT.DESCR,
              "Account debt monitor");
        list.add(trial);

        return list;
    }
}
```

Testing Maintenance Classes

There is a convenient test superclass for entity page maintenance, `com.splwg.base.api.testers.EntityPageServiceTestCase`. This test class requires several methods to be implemented to handle setting up the data and validating for each action (Add, Read, Change, Delete).

In case your maintenance doesn't support add and delete, e.g. it's read and change only, then implement this method:

```
    protected boolean isReadAndChangeOnly() {
        return true;
    }
```

The test framework will only exercise the *read* action.

Your maintenance test class must provide the name of the service being tested, eg:

```
    protected String getServiceName() {
        return "CILTBTCP";
    }
```

Testing Add on Maintenance Class

First, in order to test an add, we need the data to add. This is provided in the method `protected PageBody getNewEntity()`. Here is an example:

```
protected PageBody getNewEntity() {
        PageBody body = new PageBody();
        body.put(Maintenance.STRUCTURE.BATCH_CD, "ZZTEST2");
        body.put(Maintenance.STRUCTURE.PROGRAM_NAME, "ZZPROG");
        body.put(Maintenance.STRUCTURE.ACCUM_ALL_INST_SW, Boolean.FALSE);
        body.put(Maintenance.STRUCTURE.DESCR, "Test service");
        body.put(Maintenance.STRUCTURE.LAST_UPDATE_DTTM,
             LAST_UPDATE_TIMESTAMP);
        body.put(Maintenance.STRUCTURE.LAST_UPDATE_INST, BigInteger.ZERO);
        body.put(Maintenance.STRUCTURE.NEXT_BATCH_NBR, BigInteger.ZERO);

        ItemList itemList =  body.newItemList
             (Maintenance.STRUCTURE.list_BCP.name);
        ListBody listBody = itemList.newListBody();

        listBody.put(Maintenance.STRUCTURE.list_BCP.BATCH_CD, "ZZTEST2");
        listBody.put(Maintenance.STRUCTURE.list_BCP.SEQ_NUM,
             BigInteger.valueOf(10));
        listBody.put(Maintenance.STRUCTURE.list_BCP.BATCH_PARM_NAME,
             "param1");
        listBody.put(Maintenance.STRUCTURE.list_BCP.BATCH_PARM_VAL, "val1");
        listBody.put(Maintenance.STRUCTURE.list_BCP.REQUIRED_SW,
             Boolean.FALSE);
        listBody.put(Maintenance.STRUCTURE.list_BCP.DESCR50, "Parameter 1");
        listBody.prepareToAdd();

        return body;
}
```

(This may look like an awful lot of typing, but any IDE like e.g. Eclipse that offers code-completion will make this kind of code entry very quick).

If the maintenance performs some server-side "defaulting" (changing of the data), and the result after the add differs from the data above, you will need to override `protected PageBody getNewReadEntity(PageBody original)`. This method gets the original data from the method above, and allows manipulation to bring it to the expected form after a read from the database.

In order to actually perform the read, the read header should be specified in `protected abstract PageHeader getReadHeader()`. For example:

```
    protected PageHeader getReadHeader() {
        PageHeader header = new PageHeader();
        header.put(Maintenance.HEADER.BATCH_CD, "ZZTEST2");
        return header;
    }
```

Testing Change on Maintenance Class

Next, a new read is performed (using the same read header above), and you can perform a change to the page body in the method:

```
protected PageBody changedPageBody(PageBody original)
```

Here is an example:

```
    protected PageBody changedPageBody(PageBody original) {
        original.put(Maintenance.STRUCTURE.ACCUM_ALL_INST_SW, Boolean.TRUE);

        ItemList list = original.getList("BCP");
        ListBody param = (ListBody) list.getList().get(0);
        param.put(Maintenance.STRUCTURE.list_BCP.DESCR50,
             "Changed parameter 1");
        param.prepareToChange();

        return original;
    }
```

A read is performed after the above changes are sent, and the results are compared.

Testing Delete on Maintenance Class

Finally, a delete is issued on the data, and it is verified that the entity no longer exists.

Test default actions on Maintenance Class

In addition, all defaults that are registered for a page maintenance must also be tested. This should be done through separate tester methods for each default, calling the FW support method `public PageBody executeDefault(PageBody pageBody, String defaultValue)`:

```
   public void testDefaultChg() {

       PageBody input = new PageBody();


       // TODO populate inputs for default

       // e.g.

       input.put(Maintenance.STRUCTURE.FK, "FK CODE");

       PageBody output = executeDefault(input, Maintenance.DEFAULTS.CHG);


       // TODO compare the outputs

       //  e.g.

       assertEquals("FK Description",

            output.get(Maintenance.STRUCTURE.FK_DESCR));

   }
```

Here is an example to test the default on a field under a list.

```
public void testDefaultAlogrithm() {

       PageBody input = new PageBody();


       ItemList itemList = input.newItemList

            (Maintenance.STRUCTURE.list_MRRA.name);

       ListBody listBody = itemList.newListBody();

       listBody.put(Maintenance.STRUCTURE.list_MRRA.MRR_ACTN_ALG_CD,

            "MRRCRESVCCC");


       PageBody output = executeDefault(input, Maintenance.DEFAULTS.AAD);

       ItemList outList = output.getList
```

```
            (Maintenance.STRUCTURE.list_MRRA.name);

        ListBody body = (ListBody) outList.getList().get(0);

        assertEquals(body.get(Maintenance.STRUCTURE.list_MRRA.MRRA_DESCR),

            "Create Service Customer Contact");

    }
```

The input page body should be populated with the expected inputs for the default action, while the output should be compared against the expected output.

Testing Entity Page Maintenance Classes

There is a convenient test superclass for entity page maintenance, `com.splwg.base.api.testers.EntityListPageTestCase`.

This test class requires several methods to be implemented to handle setting up the data and validating for each action (Add, Read, Change, Delete).

The maintenance test class must provide the name of the service being tested, eg:

```

    protected String getServiceName() {

        return "CILTBTCP";

    }
```

Testing Add on Entity Page Maintenance Class

First, in order to test an add, we need the data to add. This is provided in the method `protected void populateRowForAdd(ListBody row)`. Here is an example:

```

protected void populateRowForAdd(ListBody row) {

        row.put("DESCR50", "description");

        row.put("XAI_IN_SVC_ID", "$");

}
```

We also need to know the ID field, and an example ID, eg

```
    protected String getMainHeaderField() {
        return "NT_DWN_TYPE_CD";
    }

    protected StringId getTestId() {
        return new NotificationDownloadType_Id("FOO");
    }
```

Testing Change on Entity Page Maintenance Class

Also, a change is attempted, using the same keyed row given by the testId method above.

```
    protected void populateChangedRow(ListBody row) {
        row.put("DESCR50", "changed description");
        row.put("XAI_IN_SVC_ID", "#");
    }
```

The Comparisons

After the adds and changes above (also a delete is done), the state of the row is compared against the new row. By default, the framework implementations should work fine, and you don't need to do anything. However, in the rare case, you may need to override the following methods:

```
protected void compareAddedRow(ListBody originalListBody,
    ListBody newListBody)
protected void compareChangedRow(ListBody originalListBody,
    ListBody newListBody)
```

Test default actions on Entity Page Maintenance Class

In addition, all defaults that are registered for a page maintenance must also be tested. This should be done through separate tester methods for each default, calling the FW support method `public PageBody executeDefault(PageBody pageBody, String defaultValue)`:

```
    public void testDefaultChg() {
        PageBody input = new PageBody();
```

```
       // TODO populate inputs for default

       // e.g.

       input.put("FK", "FK CODE");

       PageBody output = executeDefault(input, "CHG");


       // TODO compare the outputs

       //  e.g.

       assertEquals("FK Description", output.get("FK_DESCR"));

   }
```

Another example for testing the default on the field which in on the list.

```
public void testDefaultAlogrithm() {

       ItemList itemList = new ItemList();

       itemList.setName("MRRA");

       List list = new ArrayList();

       itemList.setList(list);

       ListBody listBody = new ListBody();

       listBody.put("MRR_ACTN_ALG_CD", "MRRCRESVCCC");

       list.add(listBody);


       PageBody input = new PageBody();

       input.addList(itemList);


       PageBody output = executeDefault(input, "AAD");

       ItemList outList = output.getList("MRRA");

       List outputList = outList.getList();

       ListBody body = (ListBody) outList.getList().get(0);

       assertEquals(body.get("MRRA_DESCR"),

           "Create Service Customer Contact");

   }
```

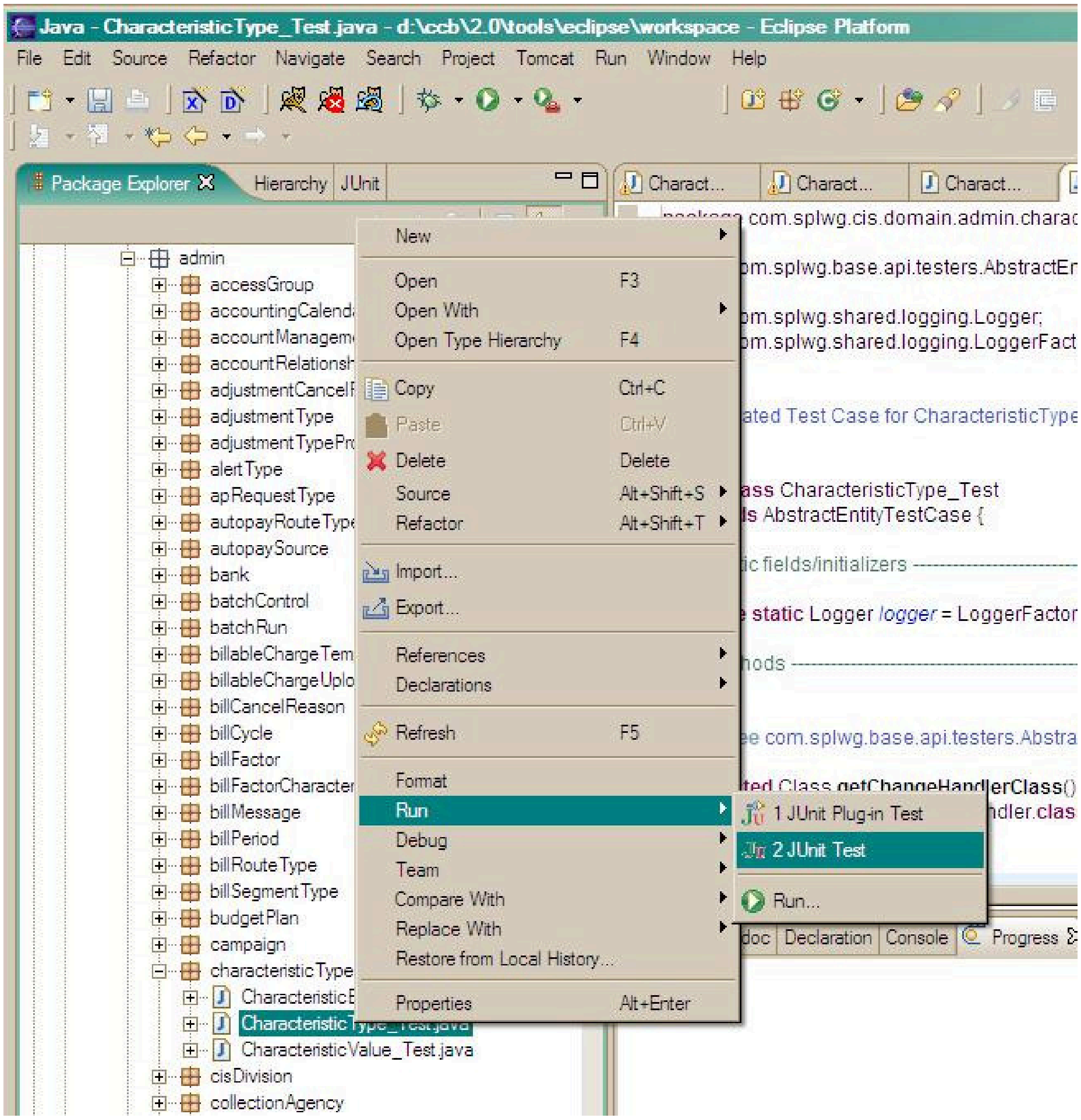
The input page body should be populated with the expected inputs for the default action, while the output should be compared against the expected output.

Testing Business Entity Validation

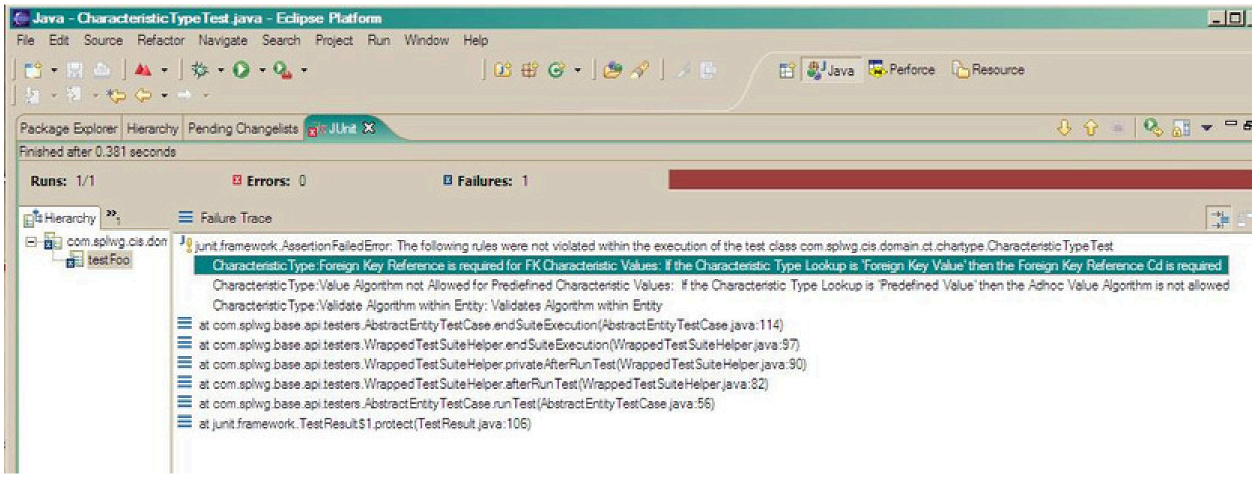
To test our validation, a test class needs to be created. The one-off generation process has created one for each of the existing entities in the system. The following is the one it created for the Characteristic Type entity:

```
public class CharacteristicType_Test extends AbstractEntityTestCase {


    private static Logger logger = LoggerFactory.getLogger(CharacteristicType_Test.class);


    /**

      * @see com.splwg.base.api.testers.AbstractEntityTestCase#getChangeHandlerClass()

      */

    protected Class getChangeHandlerClass() {

        return CharacteristicType_CHandler.class;

    }

}
```

This is a JUnit test case. Let's run it. From within Eclipse, right-click on the test class from within the Package Explorer.



The following image shows the resulting output from JUnit:


Java - CharacteristicTypeTest.java - Eclipse Platform
File Edit Source Refactor Navigate Search Project Run Window Help
Java Perforce Resource
Package Explorer Hierarchy Pending Changelists JUnit
Hierarchy
Failure Trace
com.splwg.cis.don
testFoo

```
Finished after 0.381 seconds
Runs: 1/1        Errors: 0        Failures: 1

junit.framework.AssertionFailedError: The following rules were not violated within the execution of the test class com.splwg.cis.domain.ct.chartype.CharacteristicTypeTest
    CharacteristicType:Foreign Key Reference is required for FK Characteristic Values: If the Characteristic Type Lookup is 'Foreign Key Value' then the Foreign Key Reference Cd is required
    CharacteristicType:Value Algorithm not Allowed for Prediefined Characteristic Values:  If the Characteristic Type Lookup is 'Predefined Value' then the Adhoc Value Algorithm is not allowed
    CharacteristicType:Validate Algorithm within Entity: Validates Algorithm within Entity
at com.splwg.base.api.testers.AbstractEntityTestCase.endSuiteExecution(AbstractEntityTestCase.java:114)
at com.splwg.base.api.testers.WrappedTestSuiteHelper.endSuiteExecution(WrappedTestSuiteHelper.java:97)
at com.splwg.base.api.testers.WrappedTestSuiteHelper.privateAfterRunTest(WrappedTestSuiteHelper.java:90)
at com.splwg.base.api.testers.WrappedTestSuiteHelper.afterRunTest(WrappedTestSuiteHelper.java:82)
at com.splwg.base.api.testers.AbstractEntityTestCase.runTest(AbstractEntityTestCase.java:56)
at junit.framework.TestResult$1.protect(TestResult.java:106)
```

As we see, the tests failed and told us that none of our three validation rules where validated. This is, of course true, but some explanation is necessary. When we run entity test cases, the framework looks up the change handler class being tested and collects all of its rules. Then it executes all the tests in the test class (basically every method starting with "test*"). At the end of each test, it looks to see if the last rule violated was one of the rules we are testing. At the end of all the tests, if there are still validation rules that weren't violated, the framework complains. At a minimum, the goal from this point is to create tests that violate each of our rules at least once. Preferably, tests should be created to violate the rules for all additional conditions that we can think of that might compromize the state of the entity.

Let's start fixing our tests with the third rule above the "Foreign Key Reference is required for FK Characteristic Values" rule. With a little head-scratching we determine that this is a "RequireRule" and we replace it as shown below:

```
public static ValidationRule
        foreignKeyReferencesRequiredForFkCharValueRule(){
    return RequireRule...someFactoryMethod...(
        "CharacteristicType:Foreign Key Reference is required for FK
             Characteristic Values",
        "If the Characteristic Type Lookup is 'Foreign Key Value' then the
             Foreign Key Reference Cd is required",
      ... some fancy stuff ....
        fkReferencesRequiredForFKCharacteristicValueMessage);
}
```

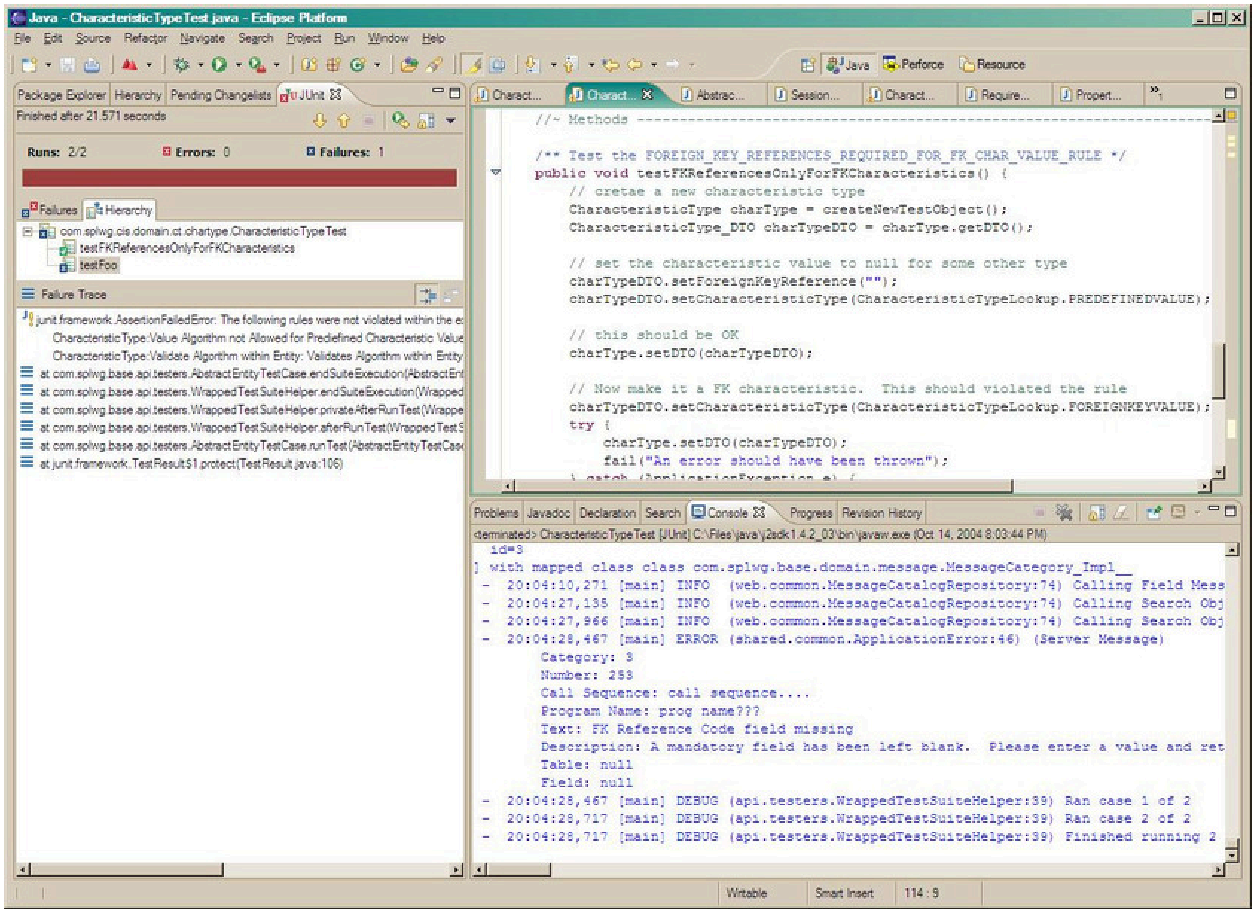
Here's the test that was added to the test class to test it:

```
/** Test foreignKeyReferencesRequiredForFkCharValueRule  */
public void testFKReferencesOnlyForFKCharacteristics() {
    // create a new characteristic type
    CharacteristicType charType = createNewTestObject();
    CharacteristicType_DTO charTypeDTO = charType.getDTO();

    // set the characteristic value to null for some other type
    charTypeDTO.setForeignKeyReference("");
    charTypeDTO.setCharacteristicType
          (CharacteristicTypeLookup.PREDEFINEDVALUE);

    // this should be OK
    charType.setDTO(charTypeDTO);

    // Now make it a FK characteristic.  This should violated the rule
    charTypeDTO.setCharacteristicType
          (CharacteristicTypeLookup.FOREIGNKEYVALUE);
    try {
        charType.setDTO(charTypeDTO);
        fail("An error should have been thrown");
    } catch (ApplicationException e) {
        // Make sure the correct rule was violated.
        VerifyViolatedRule
           (CharacteristicType_CHandler.
           foreignKeyReferencesRequiredForFkCharValueRule());
    }
}
```

Note:

Important note: Both a valid test *AND* an invalid test were added to the above method.

Finally, when the test is rerun, we have one less validation rule needing to be violated.


Java - CharacteristicTypeTest.java - Eclipse Platform
File Edit Source Refactor Navigate Search Project Run Window Help
Java Perforce Resource
Package Explorer Hierarchy Pending Changelists JUnit
Charact... Charact... Abstrac... Session... Charact... Require... Propert...
Failures Hierarchy
Failure Trace
Problems Javadoc Declaration Search Console Progress Revision History
Writable Smart Insert 114 : 9

```
Finished after 21.571 seconds

Runs: 2/2        Errors: 0        Failures: 1

com.splwg.cis.domain.ct.chartype.CharacteristicTypeTest
    testFKReferencesOnlyForFKCharacteristics
    testFoo

junit.framework.AssertionFailedError: The following rules were not violated within the e
    CharacteristicType:Value Algorithm not Allowed for Predefined Characteristic Value
    CharacteristicType:Validate Algorithm within Entity: Validates Algorithm within Entity
at com.splwg.base.api.testers.AbstractEntityTestCase.endSuiteExecution(AbstractEnt
at com.splwg.base.api.testers.WrappedTestSuiteHelper.endSuiteExecution(Wrapped
at com.splwg.base.api.testers.WrappedTestSuiteHelper.privateAfterRunTest(Wrappe
at com.splwg.base.api.testers.WrappedTestSuiteHelper.afterRunTest(WrappedTestS
at com.splwg.base.api.testers.AbstractEntityTestCase.runTest(AbstractEntityTestCase
at junit.framework.TestResult$1.protect(TestResult.java:106)
```

```
    //~ Methods -----------------------------------------------------------------------

    /** Test the FOREIGN_KEY_REFERENCES_REQUIRED_FOR_FK_CHAR_VALUE_RULE */
    public void testFKReferencesOnlyForFKCharacteristics() {
        // cretae a new characteristic type
        CharacteristicType charType = createNewTestObject();
        CharacteristicType_DTO charTypeDTO = charType.getDTO();

        // set the characteristic value to null for some other type
        charTypeDTO.setForeignKeyReference("");
        charTypeDTO.setCharacteristicType(CharacteristicTypeLookup.PREDEFINEDVALUE);

        // this should be OK
        charType.setDTO(charTypeDTO);

        // Now make it a FK characteristic.  This should violated the rule
        charTypeDTO.setCharacteristicType(CharacteristicTypeLookup.FOREIGNKEYVALUE);
        try {
            charType.setDTO(charTypeDTO);
            fail("An error should have been thrown");
```

```
<terminated> CharacteristicTypeTest [JUnit] C:\Files\java\j2sdk1.4.2_03\bin\javaw.exe (Oct 14, 2004 8:03:44 PM)
  id=3
] with mapped class class com.splwg.base.domain.message.MessageCategory_Impl__
 -  20:04:10,271 [main] INFO  (web.common.MessageCatalogRepository:74) Calling Field Mess
 -  20:04:27,135 [main] INFO  (web.common.MessageCatalogRepository:74) Calling Search Obj
 -  20:04:27,966 [main] INFO  (web.common.MessageCatalogRepository:74) Calling Search Obj
 -  20:04:28,467 [main] ERROR (shared.common.ApplicationError:46) (Server Message)
        Category: 3
        Number: 253
        Call Sequence: call sequence....
        Program Name: prog name???
        Text: FK Reference Code field missing
        Description: A mandatory field has been left blank.  Please enter a value and ret
        Table: null
        Field: null
 -  20:04:28,467 [main] DEBUG (api.testers.WrappedTestSuiteHelper:39) Ran case 1 of 2
 -  20:04:28,717 [main] DEBUG (api.testers.WrappedTestSuiteHelper:39) Ran case 2 of 2
 -  20:04:28,717 [main] DEBUG (api.testers.WrappedTestSuiteHelper:39) Finished running 2
```

Iterate Until Done

Test handleChange / handleAdd / etc code

Although there is no way to enforce testing of any coding in any of the methods

```
HandleRegisteredChange

     (BusinessEntity changedBusinessEntity,

      RegisteredChangeDetail changeDetail)


handleAdd(BusinessEntity newBusinessEntity)


handleChange(BusinessEntity changedEntity, DataTransferObject oldDTO)
```

It is still imperative that this code should also be exercised AND verified when testing the change handler. Please ensure that every path through these methods is exercised and the results verified.

In general, there is a specific set of classes or functionality that is required to have explicitly defined tests.

- Every entity (and entity extension) class must have each of its validation rules explicitly tested. That is, each rule should fail once, with an explicit acknowledgement of the failed rule expected.
- Every service must have a test.
    - Searches must test each search type once.
        - "Page" services must test their complete cycle that are available.
        - Queries must test read
        - Maintenance classes must test add/change/read/delete
- Every maintenance extension must have a test class
- Every algorithm implementation must have a test

Note:
Currently, the above "must have" tests may still not completely cover all the cases. For example, one search type may have several inputs, which trigger different code or queries to be executed. The testing FW as is can not know this, so only requires a single test case for that search type. However, it is strongly recommended that each specialized case possible be modeled with a test case, in order to achieve complete code coverage.

Note:
In addition, there is a desire to assure that each business component or business entity method has been tested. Currently these tests are not required. However, after a complete build server run, any business component methods or business entity methods that have not been explicitly tested will be reported.

Testing for Warnings

In both maintenance classes and entity change handlers, there is the possibility of issuing a warning. This code should be tested just as well as any other entity validation or default action.

Maintenance Classes

Here is complete valid example of verifying that a maintenance default action issues a proper warning.

```
public void testDefaultDEFAULT_FOR_ZONE_HNDL() {

    PageBody input = new PageBody();

    input.put(ContentZoneMaintenance.STRUCTURE.ZONE_CD, "CI_AFH");
```

```
        // test the default and expect to get a warning
        try {
            executeDefault(input, "ZH");
            fail("Should have a warning");
        } catch (ApplicationWarning e) {
            verifyWarningContains(e,
               MessageRepository.deleteZoneParametersWarning());
        }

        disableWarnings();

        // test the default and do not expect to get a warning or error
        PageBody output = executeDefault(input, "ZH");
        assertEquals(Boolean.TRUE, output.get("DELETE_SW"));
    }
```

Note:
By default, warnings are enabled, thus nothing special need be done. But you should put the normal try/catch block around the default execution, and catch an application warning. Once inside the catch block, you should verify that the warning(s) is/are valid expected ones. (This comparison is only done via the message category and message number. Thus, if there are parameters to the message construction, it matters not their values, since it may be difficult to get the values.) You should then retry the default with warnings disabled.

Entity tests

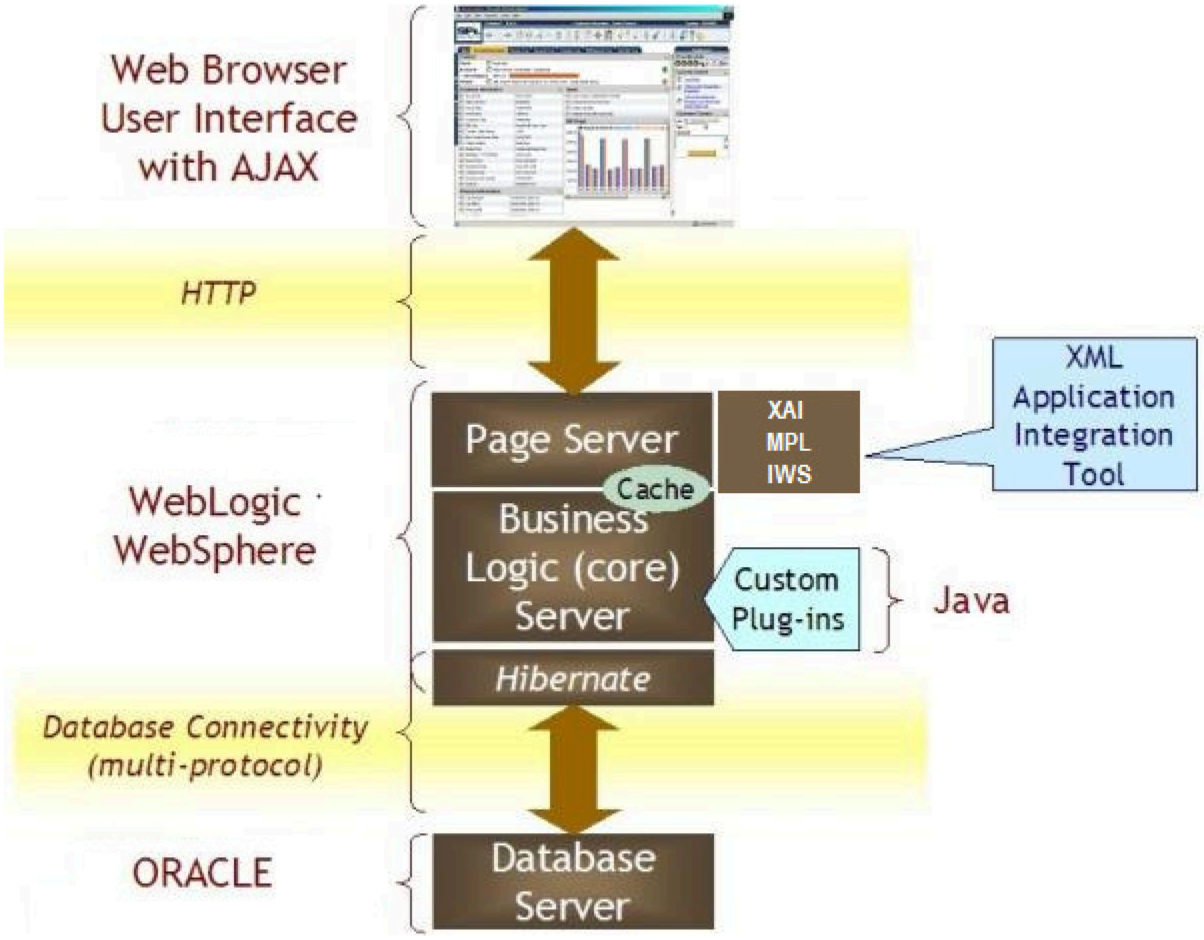
There was no current use of warnings in entity tests that I could easily "improve", so for now I use a slightly contrived example. (This is slightly contrived, because Installation is a special record, and the change below is not actually allowed in the application do to some records on the Adjustment Type table, and a valdiation on Installation.)

```
    public void testChangeBillSegmentFreeze() {
        Installation installation = getValidTestObject();
        Installation_DTO instDto = (Installation_DTO) installation.getDTO();
        instDto.setBillSegmentFreezeOption

            (BillSegmentFreezeOptionLookup.FREEZE_AT_WILL);

        installation.setDTO(instDto);


        instDto.setBillSegmentFreezeOption

            (BillSegmentFreezeOptionLookup.FREEZE_AT_BILL_COMPLETION);

        installation.setDTO(instDto);

        verifyWarningsContain

            (MessageRepository.changeBillSegmentFreezeWarning());

    }
```

Again, by default warnings are enabled, so nothing need be stated at the outset. Additionally, the conversion of warnings to an exception occurs at a later point, so there is no ApplicationWarning to catch. Instead, after the offending statement (in this case the setDTO method) you should just verify that the current warnings contain the specified message.

Technical Background

Technology Overview


Web Browser User Interface with AJAX
HTTP
WebLogic WebSphere
Page Server
Cache
Business Logic (core) Server
XAI
MPL
IWS
XML Application Integration Tool
Custom Plug-ins
Java
Hibernate
Database Connectivity (multi-protocol)
ORACLE
Database Server

Technology Overview of the OUAF System

Information is presented in a Web browser using HTML and JavaScript (not Java, e.g., no applets). The browser communicates with a Web Application Server via HTTP.

The Web Application Server is divided into several logical tiers: presentation services, business logic, and data access. Inbound HTTP requests are handled by Java Servlets in the presentation layer, which may in turn invoke data service objects. In turns, these objects may route control to Java-based business entities, which use the Hibernate ORM framework for data access and persistence.

Various static data (control tables for drop-downs, language-specific messages/labels, etc.) are cached in the presentation layer of the Web Application Server. The presentation layer makes use of XSLT technology to create HTML for the browser.

As the browser may need several "pages" to show all the information relating to a particular business entity, a JavaScript "model" is used to manage the data in toto, and the Internet Explorer XMLHTTP object is used to send the data to the server as an XML document. Data is provided to the browser as literal JavaScript. The specialized portal and dashboard areas use server-side XSLT technology to render the final HTML directly on the server. The HTML for grids in the browser is created using client-side MSXML XSLT transforms.

This kind of architecture is described as Asynchronous Javascript and XML (AJAX).

Portability

The system is highly portable to various hardware platforms, as web application servers are pure Java applications and run on myriad operating systems, including Windows clients, servers, and many versions of UNIX.

In principle, any compliant Java 2 Enterprise Edition container can host the application.

Distribution

The various logical components can be distributed to as many machines as desired. In particular, the web application server architecture is stateless, so many parallel server machines can be utilized given an appropriate load-balancing architecture.

OUAF Web Services

The OUAF system makes heavy use of web services, which are data access and update services ultimately implemented in Java, accessing Oracle databases. Each service invocation represents a distinct database transaction.

There are three kinds of services: Page, List, and Search.

A Page service defines all the data needed to display data on a single tab menu (e.g. across all child tab pages). The data structure is logically a tree, with a root object containing attributes as field/value pairs, and recursively containing lists of similarly structured objects. The typical maximum nesting depth is four levels of contained lists. Page service names end with the letter "P", e.g. CILCACCP. Page services may be called in five primary "modes": Read, Change, Add, Copy, Delete, and Default.

List services define a list of objects, possibly containing nested lists. In addition to being accessible independently for list-oriented data, they can be used to flesh out lists contained in page services where more data is available than can fit in the (fixed-size) buffer. List services do not support database updates.

Search services are used to support ad-hoc user searches for data. The results are structurally similar to List services. The input is a set of criteria and a search mode, with values "MN", "AL", "A2", A3", etc.

Service requests can return a normal result, or create an error or warning. A warning displays a message with a list of warning lines, and offers the choice of proceeding (which triggers the same call with a flag set to suppress warnings), or cancel. Errors create descriptive messages. Search and List services can only create errors, not warnings, while all Page services except Copy can create warnings and errors.

SPL Service XML Metainfo Files

The OUAF system represents the structure of a service using an XML document (loosely akin to an XML schema). Every service is defined with a single XML document, which is generated based on the Java class information.

Service XML documents are created by using annotation-based metainformation combined with system metadata (stored in the database).

Example using Page Service

The following excerpt of the CIPBSTMP.xml file will motivate the discussion.

```
<?xml version="1.0"?>

<!-- Service CIPBSTMP -->


<page service="CILBSTMP">
```

The root element of the document has the tag "page" to reflect that this is a page service, and describes the service name as an attribute.

```
   <pageHeader>

      <string name="STM_ID" size="12"/>

   </pageHeader>
```

The <page> element contains exactly one <pageHeader> and one <pageBody>. The <pageHeader> contains any number of "singleton" fields. This one is a string field named "STM_ID" in the browser, and with the same name in the original Java source. The field contains up to 12 characters (this is the "business rule" length, not physical storage).

Other types of singleton fields include <bigInteger>, <bigDecimal>, <money>, <date>, <time>, <dateTime>, and <boolean>.

The number-related fields (<bigInteger>, <bigDecimal>, and <money>) have a "precision" attribute, which describes the maximum number of digits that can be represented. Further, <bigDecimal> and <money> include the "scale" attribute, describing the number of decimal digits appearing after the decimal point. Thus, an element like this <bigDecimal precision="4" scale="2"/> can represent numbers in the range +/-99.99.

Continuing with the page service example, we have this section:

```
<pageBody>

  <row actionFlag="ROW_ACTION_FLG">

     <string name="BATCH_CD"

             size="8"/>

     <bigInteger name="BATCH_NBR"

                 precision="10"/>
```

The <pageBody> element contains <row> elements, singleton fields, and <list> elements, in any order. Here we have another <string> field, as well as a <bigInteger> (an integer value with no decimal fraction), this one holding (up to) 10 digits. This means numbers in the range +/-9,999,999,999 can be represented.

The <row> element reflect the java entity (row). In terms of the infoset and browser the fields in the <row> element are effectively merged into the containing <pageBody>, along with fields from any sibling <row> elements.

The "actionFlag" attribute names the field that contains a flag that determines the server action that should occur against the row.

```
     <string name="STM_CNST_ID"

             size="10"/>

     <date name="STM_DT" />

     <string name="STM_ID"

             size="12"

             isPK="true"/>

     <string name="STM_STAT_FLG"

             size="2"/>

     <bigInteger name="VERSION"

              precision="5"/>

  </row>
```

The "isPK" attribute marks fields that are part of the logical prime key of the "main" object/table for this page service.

We then see a <list> element:

```
<list name="STM_DTL" size="30" service="CILBSTDL" userGetMore="false">
```

The <list> element describes an elaborately structured array of objects. The element contains exactly one <listHeader> and <listBody>. Every list within a service buffer has a unique name attribute. The number of possible list body objects is given by the "size" attribute. In the event a list service exists independently for the list, it is named by the "service" attribute. Finally the "userGetMore" attribute switches the system into one of two modes:

userGetMore="false" means the system does not require the consent of the user in order to fetch more elements, in the event that the physical list buffer is filled to capacity with more elements available in the database. The system will autonomously call the corresponding list service (if it exists) in order to fetch the missing elements. In this way clients making one logical page service call may result in one physical page and several list service invocations.

userGetMore="true" means the system requires the affirmative consent of the user (e.g. via a "get more" button in the browser) to continue fetching available data. The list buffer is truncated.

```
<listHeader lastIndex="STM_DTL_COLL_CNT"
            actionFlag="LIST_ACTION_FLG"
            moreRows="MORE_ROWS_SW"
            alertRowIndex="ALERT_ROW">
     <string name="STM_ID"
        size="12"/>
     <string name="LAST_STM_DTL_ID"
        size="12"/>
</listHeader>
```

The <listHeader> element has a "lastIndex" attribute giving the field name that holds the number of elements actually returned, an "actionFlag" describing the operation to be performed on the list (e.g. change, delete), the "moreRows" attribute naming the field that holds the boolean that indicates whether more data remains un-retrieved in the database for the current list, and the "alertRowIndex" attribute, naming the field that holds an index into the list to describe the location of a validation error, used to select the correct item in a browser when presenting the error to the user.

In addition, a <listHeader> can contain any number of singleton fields. These are typically keys describing how to access this list, and logical "cursor" fields describing how to continue fetching more items.

```
  <listBody>
    <row actionFlag="ROW_ACTION_FLG2">
      <bigInteger name="VERSION"
                  precision="5"/>
      <string name="STM_DTL_ID"
              size="12"
              isPK="true"/>
      <string name="STM_CNST_DTL_ID"
              size="10"/>
      <string name="STM_ID"
              size="12"/>
    </row>
    <string name="STM_CNST_DTL_DESCR"
            size="50"/>
  </listBody>
</list>
```

This finishes the <list> element. Some more singleton elements appear before finishing the <pageBody> and <page>:

```
      <string name="STM_CNST_DESCR"
              size="50"/>
      <boolean name="ACTION_GENERATE_SW" />
   </pageBody>
</page>
```

Example Using Search Service

List service XML files essentially contain a <list> element as the document root, and will not be described further. Search service XML files are very similar to those for page services. Here is an example illustrating the differences:

```
<?xml version="1.0"?>

<!-- XML Java/Tuxedo mapping CIPCACCS

  Automatically generated by makeXMLMap Sat Nov 10 09:08:30 2001

  Source copybooks: CICCACCS CICCACCH -->

<search name="ACCT"

        service="CILCACCS"

        size="300">

   <searchHeader lastIndex="ACCT_COLL_CNT"

                 actionFlag="SRCH_ACTION_FLG"

                 searchByFlag="SEARCH_BY_FLG">
```

The <search> element is the root of the document, and contains a <searchHeader> and <listBody>. The <search> element is similar to the <list> element described above, and includes name, service, and size attributes. The <searchHeader> includes the "lastIndex" attribute, which gives the name of the field holding the number of returned elements, "actionFlag" which names the field containing the search action flag, and "searchByFlag" which names the field holding the search "mode". The <searchHeader> further contains singleton fields describing search criteria. These are adorned with extra attributes describing whether they are distinguished criteria that should always be populated from the search client (optional, defaults to "false"), and a criteria group designation.

```
      <string name="ACCT_ID"

              size="10"

              isCriteriaExtract="true"

              criteriaGroup="MN"/>

      <string name="ENTITY_NAME"

              size="50"

              criteriaGroup="AL"/>

   </searchHeader>
```

The <listBody> was described previously, and describes the structure of the elements matching the search criteria. In addition, the "isReturn" attribute describes fields that should be returned as the result data when a particular result row is selected (optional, defaults to "false").

```
   <listBody>
      <row actionFlag="ROW_ACTION_FLG">
         <string name="ACCT_ID"
                 size="10"
                 isReturn="true"/>
         <string name="ENTITY_NAME"
                 size="50"/>
         <string name="ACCT_REL_DESCR"
                 size="50"/>
         <string name="NAME_TYPE_FLG"
                 size="4"/>
      </row>
   </listBody>
</search>
```

Server Architecture Overview

The Java server is logically divided into several distinct layers, with different responsibilities. Form the point of view of a request from the browser, there are a handful of general-purpose data-centric servlets that can handle any service requests. These servlets handle HTTP requests and transform them into data objects and commands for further processing. Once an appropriate service is identified for handling a request, its metainfo is used to build a rich Java data structure from the (string-based) browser data representation. This data, in turn, forms the input to a Java service class.

In our terminology, a page service is a self-contained piece of server functionality that principally acts upon a particular "root" Java entity object (and table) and its child entities (tables). The framework automates the process of mapping data to and from these entities by making use of the service XML metainfo. These Java objects are also known as domain objects. These objects are made persistent via Hibernate, which offers the simpler HQL query language as an alternative to "raw" SQL. In the simplest cases, no human-written imperative code need be written to implement a page service.

For updates, validation is handled via both automatic logic determined by database metadata, including application-level referential integrity checking, and hand-coded validations implemented by change-handler classes.

For presentation-level requests (e.g. HTML constructs) the server uses XSL/T to create HTML from our UI metadata structure. Since UI layouts are fairly static the web presentation layer uses caching to help optimize performance.

The development process is geared to keeping generated and human-maintained artifacts completely separate. For instance, we generate superclasses that contain generated code that is e.g. required by Hibernate or the service framework, and programmers will implement subclasses that implement methods for unusual or extra behavior.

The artifact generation is driven principally by parsing special markup in Java sources known as annotations, combined with metadata held in the database (principally relating to tables, fields, and constraints). On a modern machine (eg. 3 GHz P-4) artifact generation for the entire system takes only a few minutes.

Client Architecture Overview

Introduction

The OUAF browser client uses many novel mechanisms in order to support the system design goals of high system performance, including low latency and high throughput. The core design principle is that the system is stateless, meaning only the browser client itself is aware of the session state, that is the application's context--what data is being viewed/modified and all other information related to a user more typically associated with session state on the server. This document discusses the important design points that implement a stateless architecture.

Client Architecture Discussion

Note:
This page is related to fixed pages in the application.

The web browser client communicates with the Web Application Server via HTTP, and receives two kinds of dynamic content:

- Views - HTML entities that describe how things look
- Model - Pure data in a convenient representation

The views are served as HTML that contain labels and HTML <SELECT> elements (drop-downs) that are localized to the user's language. Since drop-downs are fairly static, the view objects are cached on the browser via the HTTP 1.1 Cache-Control directive in order to avoid repeatedly accessing the Web Application Server for the same content.

The web server creates HTML using XSLT technology. The technique is as follows: the original metadata that defines HTML documents is copied from the database into XML files residing in the server. These files are shipped as part of a OUAF system deployment. At runtime the server converts these UI metadata documents into HTML in two steps. First, the logical structure is converted into a nearly-final HTML structure, lacking only language-specific information (labels and <select> lists), via XSLT, using one of a handful of standard XSLT template files. The result object is then transformed again, in order to inject language-specfic elements, creating the final HTML.

As a special case, for performance reasons the HTML for grids (lists and search windows) is created using client-side XSLT (MSXML).

Data is provided via servlets, of which there are only a few. The data is represented as literal JavaScript, which happens to be very convenient to handle by the browser (since it includes a native JavaScript parser). The model takes the form of a "tree" of JavaScript object nodes and values, with a distinguished root node. The model is converted into an XML document string when submitted to the server, using the HTTP POST method. This is convenient for the Web Application Server because it is equipped with powerful Java-based XML parsers.

Here is a table of servlets, and a brief description of each:

Servlet Name	HTTP Method	Usage
loginInfo	GET	Provides useful global data at login, retained for life of session, such as a map of all system URLs and a definition of menu structures.
pageRead	GET	Returns a data model object given one or more key/value pairs.
pageChange	POST	Accepts a data model representing modifications that should take place against an existing database entity, returns modified model.

Servlet Name	HTTP Method	Usage
pageAdd	POST	Accepts a data model representing a new entity that should be inserted into the database, returns new model.
pageDefault	POST	Accepts a data model describing a triggered "default" operation, returns a model object containing default values.
pageDelete	POST	Accepts a data model representing a database entity to be deleted, returns nothing.
pageCopy	POST	Accepts a data model representing a model that should be duplicated, returns the duplicate.
listRead	GET	Accepts key/value pairs describing a list of database entities, returns said list.
Search	GET	Accepts key/value pairs containing search criteria, returns list of objects satisfying said criteria.
StringSort	POST	Allows locale-sensitive sorting of strings. Used for sorting strings in grids when user clicks on the column header.

It is important to remember that the servlets deal with "pure" model data, and have no visible representation. Since actual business logic and database access resides in the app server, the servlets take the role of dispatchers and most servlets accept a "service" parameter describing which app server back-end service to invoke.

The model data is combined with the view on the browser client whenever the model changes or the view needs to be refreshed. This is done by a name-matching scheme where every HTML element that shows a model value has a name that "picks out" a corresponding value from the current model. All such HTML fields must include the string "data" in their HTML class.

The simplest case is showing a value from the "root" object, in which case the field name, also referred to as the "JS name" simply matches the model's attribute name.

There is more complexity in the case of lists. Every list in the model has a unique name, regardless of nesting depth, so a JS name that combines the list name with the property name suffices to uniquely identify a section of the model.

There are two sub cases of displaying properties of lists. The first is where the desired index into the list is known (e.g. grids). In this case, the JS name combines the list name, index, and property name as follows: <LIST_NAME>:POSITION$ELEMENT_PROPERTY, e.g. ACCTS:3$ACCT_ID. Note indexes are always 0-based in the browser (in accordance with JavaScript arrays). This example refers to the fourth element of the ACCTS list, and retrieves the ACCT_ID.

The other case is where the desired index is inferred as the "current" index (e.g. scrolls). Every list in the model keeps track of its current position index, which is used when no external index is provided. Hence the JS name ACCTS$ACCT_ID refers to the ACCT_ID property of the currently selected/visible element (presumably in a scroll) of the ACCTS list.

In the rare case that header fields should be displayed, they can be accessed using the JS name pattern LIST_NAME#HEADER_PROPERTY.

Generally, whenever a value is changed in an HTML element or focus is moved after making a change, the system attempts to "commit" the change back to the model. This involves several steps:

- Validating the input for syntax, possibly according to the current locale (e.g. dates)
- Identifying the relevant model node to receive the change
- Updating the node
- Marking the node dirty

The last step is important because we wish to know whether the user has made any changes to the model.

Saving changes to the server generally involves the following steps:

Identifying the servlet to be invoked (e.g. pageChange vs. pageAdd) depending on whether the model represents a new or already persistent entity.

Converting the model tree into a literal XML string representation.

Submitting the XML string to the Web Application Server via the appropriate servlet using the POST method, including a parameter describing the service. This is accomplished via the Internet Explorer XMLHTTP ActiveX object, using a synchronous calling mechanism.

The servlet, constituting a core piece of the Web Application Server "web presentation" layer, retrieves the service parameter, obtains the XML-based metainfo describing the service, and converts the XML request string into a Java object tree, ready to be passed to the "data service" layer.

The data service layer converts the Java object tree into a Jolt data buffer.

The relevant app service is invoked passing in the data buffer.

The result buffer is converted into a Java object tree.

The Java object tree is converted into a literal JavaScript representation (as needed) and transmitted to the browser.

The returned data (literal JavaScript) is mapped into a new live JavaScript object model.

The user interface is refreshed with the new data.

The portal screens use zone configuration to create HTML. Each distinct zone within the portal is created by a Java object known as a zone type. The zone type is responsible for acquiring data and rendering the displayed HTML. The portal framework takes care of common features such as zone expand/collapse.

SPL Client API

Overview

The OUAF system offers a large number of useful JavaScript functions in the client (browser). These allow manipulation of widgets, data, and triggering requests to the Web Application Server to view another page and/or object.

This document discusses functions that developers of user exit functions may wish to use.

Client API Discussion

JavaScript Invocation Context

To use client-side JavaScript functions effectively you have to understand the way JavaScript partitions the client window into independent object spaces (via iframes), and the way common JavaScript framework code is made available to those spaces.

While this may be old hat for experienced browser-side JavaScript programmers, it is important to remember that every window or iframe contains an independent object space, with a separate space of global variables and objects. Adding, modifying, or deleting objects (or classes) in one iframe has no effect on any objects in other iframes. However, it is possible to refer to objects that "live" in a different iframe with variables in a different iframe. This is somewhat dangerous; if the iframe that instantiated the object (where its prototype lives) goes away (by being closed, or having its href modified) the object may no longer be able to carry out any operations. However, "value" objects such as strings or numbers may safely be shared across frames, even if the creating frame closes.

An iframe can have JavaScript code defined directly into its HTML page, or it can include JavaScript code that exists in a separate file. The latter technique allows the creation of standard "library" code that is available to many iframes, without the network or development overhead of copying the same functions into hundreds/thousands of files.

The subtle point to remember in the foregoing is that even though the same JavaScript file may be included in different iframes, each definition is completely independent of every other. If the included JavaScript defines a global, then every iframe gets its own separate global variable binding.

Since there are some commonly used objects, most iframes define and initialize global variables to reference these objects. The "main" object usually refers to the main iframe, containing cisMain.jsp. This iframe is never reloaded during a client session, and holds the central model object. The model itself is usually available as the "model" global in most iframes.

Data Representation and Localization

The browser handles localization. The client browser object model is logically divided into a display ("view") and data ("model") layer. The responsibility for localizing the data values to the user's locale rests with the display layer, not the model layer. All code that retrieves model values and prepares them for display makes use of formatting code, for instance to display dates in the user's preferred fashion. Similarly, all user input is parsed in the context of the current locale to convert the data into the internal format that is stored in the model (and passed to Servlete Container/Java/database).

The internal format for model properties is to use strings (not JavaScript numbers or dates) for everything except boolean values.

The functions that are responsible for converting model data values into localized displayable ones are named convertInternalXYZToLocal(internalValue), while the reverse conversions are named convertLocalXYZToInternal(localValue). Note that latter can fail and include extensive validation, possibly triggering error message alerts.

Core JavaScript Classes

These classes are defined in cis.js, which is included by the main frame.

CisModel

The CisModel class plays two roles. The first role is to provide the metadata that describes the currently loaded model instance. The methods that serve this role are static methods, e.g., defined directly on the CisModel prototype object. These methods are accessible using the syntax CisModel.*function(params)*, assuming you are in the main frame. If not, use main.CisModel.*function(params)*. Instance methods and variables are, of course, accessible through any instance of CisModel, e.g. model.pageData or model.getValue('ACCT_ID').

Data representation

The data stored in instances of CisModel uses an internal system representation, not a localized representation. This means code that manipulates the model is unaware of the user's locale and display preferences. For instance, date values are always stored using the ISO 8601 string representation YYYY-MM-DD, and numbers are always stored as strings, not JavaScript numbers (because the required precision may exceed that of JavaScript's native number type).

The data takes the form of a tree of data nodes of two classes, DataElement (the singleton node class holding data attributes as JavaScript properties), and List, which manages an array of DataElement instances. Every list has a unique name property, regardless of its position in the tree (e.g. independent of nesting depth), making it possible to uniquely identify and retrieve any list by its name. The DataElement instances each keep a "dirty" flag to mark whether the user modified any properties, representing work that needs to be persisted to the database.

Every element instance is always in one of three logical states:

Persistent

New/Dirty

New/Clean (e.g. a "phantom")

Phantom elements are used to populate otherwise empty lists, to give a starting point in which to enter data. Unlike other new objects, phantoms are initially clean so they don't participate in persistence operations until explicitly modified.

Many CisModel methods act as starting points for recursive implementation through the data tree, and methods with the same or similar names are available on the DataElement class.

Navigation

Many methods accept list names as arguments in order to operate a unique list instance within the model data tree. Since it is possible for lists to be nested inside other lists, the system assumes the intended list is the one identified by the "current" list positions in the ancestor branch.

CisModel Instance Variables

- pageData

Used to access the root data node (e.g. model.pageData), an instance of the DataElement class.

Static methods

- parseNames(fieldName)

This function accepts a string containing a JS field name (the id of an HTML element) and cracks it into its constituent components: the list name, index, and property. The result is an object with three attributes, property and listName (which are null if missing), and position, which is actually a function that takes the current list position as an integer argument. The reason for this is to allow the calculation of the current position (no index) and a fixed index. If the list fragment is missing the listName and position are both null.

CisModel Instance Methods

The most commonly accessed instance of CisModel is the central model containing the data for the current page, typically available through the "main" global variable in most frames. (Recall again that all such globals refer to the same object; hence changes made with code running in one iframe are visible from any other iframe.)

- getValue(fieldName)

This accessor method returns the data value corresponding to the provided fieldName. The field name string will be cracked into constituent pieces using the static parseNames() method, in order to identify the instance of DataElement within the data "tree" that contains the desired property, and then to retrieve the property.

- getOriginalValue(fieldName)

If a model property has been modified, but the change has not yet been committed (e.g. with the Save button), the system tracks the originally retrieved value of the property. This accessor method can be used to retrieve the original value. Useful for implementing certain business rules having to do with logical state transitions.

- canSetValue(fieldName, value)

This method answers a boolean indicating whether the model is capable of accepting the given value for the provided fieldName. The method would answer false if buffer capacity limits would be exceeded were the change to be accepted.

- setValue(fieldName, value)

This setter modifies the property identified by fieldName to hold the given value. Defaulting will not be triggered.

- setLocalValueWithDefault(fieldName, localValue)

Sets the property identified by the given fieldName to the specified localValue. This method does not handle conversion errors, so the provided localValue should have already passed syntactic validation.

- setValueWithDefault(clientWindow, fieldName, value, element, afterFieldUpdateFunction, continuation, forceDefault, skipDefault, successFunction)

Sets the property identified by the given fieldName to the specified value. This method may trigger a default, and therefore requires further parameters:

clientWindow - the window object containing the element triggering the default.

element - the HTML element that is attempting to accept the value.

afterFieldUpdateFunction - a thread-safe continuation to be run after the value has been changed

continuation - a function to be run whether or not the value can be accepted (may be null)

forceDefault - an optional boolean that forces the default checking code to run, bypassing a shortcut execution path (presumeably to obtain a desired side-effect from defaulting) skipDefault - an optional boolean that bypasses the default triggering logic

successFunction - continuation to execute if the value is accepted by the default logic

Note:
Note this function is not "thread-safe", in the sense that it cannot be safely called e.g. in a loop that may issue several calls to this function. The workaround is to make use of the continuation function to "schedule" the follow-up operation.

- setListPosition(listName, newPosition)

Sets the list identified by the given listName to the position (a zero-based integer value) given by newPosition. Useful when you want to display a particular element in a scroll.

- getList(listName)

Answers the list object (instance of List) with the given name.

- replaceWithNewList(listName, sourceModel)

Typically called from the default handler callback, this method replaces the entire contents of the given list in the receiver (e.g. the model whose method is being called) with the list as provided in sourceModel. All elements in the list are considered new, and are eligible to be added to the database when the Save button is used.

- hasRealElements(listName)

Answer a boolean indicating whether the indicated list contains any persistent or dirty elements (not merely a single phantom).

- getElement(fieldName)

Returns the data element (instance of DataElement) corresponding to the given field name. The method resolves the list name and position index, if any, in order to navigate to the appropriate data element.

- markAsNew()

Mark the model "clean" by recursively clearing all dirty flags throughout the tree structure. Note this is a very "sensitive" method and should only rarely be needed.

DataElement

Instances of DataElement play the role of the nodes in the data model tree. They have properties corresponding to business attributes, and also define the tree structure by holding references to their parent data element and list(s) of children elements. The distinguished root DataElement instance in the core model instance is accessed using with the "pageData" property.

DataElement Instance Variables

- _isDirty

This boolean flag indicates whether the business attributes (stored as JavaScript object properties) have been modified since the DataElement was created (either using persistent database attributes or as a new object that will be added later). This attribute should not be modified directly.

- _originalIndex

An integer representing the position of this DataElement in its containing list when it was first retrieved from the server. For non-persistent data elements this value is -1.

DataElement Instance Methods

- set(property, value)

Set the given property to the given value. This apparently simple method can trigger a variety of side effects, including intelligently converting key attributes to uppercase (which is the default unless turned off with the tabMenu.shouldNotAutoUppercase property). The method returns a boolean indicating whether the setter succeeded; reasons for failure include if the existing value is identical to the current value (this is needed to avoid needlessly marking the data element dirty), and if the mutation capacity would be exceeded.

Further, if the modified attribute represents a key value, the change is propagated recursively through all child data elements by matching the attribute names.

In addition, the original, unmodified, property value is retained for future reference, and can be accessed using the originalValue function.

- originalValue(property)

Returns the original, unmodified value for the given property name. This is useful for user exit code implementing business rules that depend not only on the new attribute value but also the original value.

- isNew()

Answers a boolean indicating whether this DataElement instance is new, e.g. created by the user. Persistent instances answer false. The method uses the _originalIndex attribute to decide.

- isPersistent()

The logical opposite of isNew(); answers true only for persistent data elements. Useful for avoiding excessive use of logical not (!) operators, thus clarifying the intentions of the code.

- isDirty()

Answers a boolean indicating whether the message receiver data element *or any descendant child data element* is marked with the _isDirty flag.

- list(listName)

Answers the list (instance of List) corresponding to the given list name.

- clearDirtyFlag()

Set the receiver's _isDirty flag to false. Potentially dangerous because it subverts the systems automatic dirty tracking system.

- canBecomeDirty(property, value)

Answers a boolean indicating whether the receiver can accept the given value for the given property. The method answers false for various reasons, including if the proposed value matches the previous vaule, the receiver is not dirty and its parent list may not become dirty, and the property is not known to the metadata.

List

Instances of List represent a collection of DataElement instances, held by a parent DataElement. Every List is held by a DataElement, but the pageData "root" node has no parent list (it's held directly by the model).

List Instance Variables

- parentElement

  The instance of DataElement that contains this list.

- name

  The name of the list.

- position

  The integer index (0-based) of the "current" element.

- header

  The JavaScript object representing the list header. This is rarely accessed.

- elements

  A JavaScript array containing the current list of elements (instances of DataElement).

List Instance Methods

- size()

Answers the number of elements held by the receiver. Does not include elements scheduled for deletion.

- currentElement()

Answer the DataElement instance from the elements collection referred to by the currentPosition instance variable.

- isDirty()

Answer a boolean indicating whether any element is dirty. The test is recursive, and answers true if any descendant has the _isDirty flag set.

- markElementsAsNew()

This convenience method marks all elements in the collection as new, setting the _isDirty flag to true and setting the originalIndex to -1 for every DataElement. The method acts recursively on all descendant lists and elements.

- realSize()

Answers the number of elements, disregarding any phantom element.

- hasRealElements()

Answers true if the list contains at least one non-phantom element.

Free Functions

top.js

This is the set of "free" functions available in top.js, which is included by cis.jsp. You typically access these functions using the top.xyz() syntax, assuming your code is running in an iframe nested under cis.jsp.

The trend has been to de-emphasize the use of functions at this level, and migrate them to the main level. In the future we plan to eliminate the distinction between the top and main frames.

- getNavigationKeyForService(service)

Answers the navigation key corresponding to the given service (string). Since there may be several nav keys for the same service, the last one is answered. You always get the correct response for tab menus.

- getURL(navigationKey, withoutLanguage)

Answer the URL (string) corresponding to the given navigation key (also a string). If the withoutLanguage boolean is false, the user's language code is appended to the URL as a GET parameter.

- getFieldLevelSecurityInfo(navigationKeyOrService)

Answer the field-level security meta-data for the tab menu given by the navigation key or service. The result takes the form of a simple JavaScript object, with security types as properties, and values as arrays of all related authentication levels. For example, assume that adjustment maintenance has field-level security defined for a user for security type "ADJAMT", with two associated authentication levels, "1", and "3". To retrieve the object defining all field-level security info for the service:

var info = top.getFieldLevelSecurityInfo('adjustmentMaint')

or

var info = top.getFieldLevelSecurityInfo('CILAADUP')

(Note the literal JavaScript representation of the result object would be {ADJAMT: ['1', '3']})

To determine the array of authentication levels associated with this service and security type in one step:

var authenticationLevels = top.getFieldLevelSecurityInfo('adjustmentMaint')['ADJAMT']

or

var authenticationLevels = top.getFieldLevelSecurityInfo('CILAADUP')['ADJAMT']

- getMain()

This heavily used function returns a reference to the window constituting the "main" iframe, containining cisMain.jsp and the core model. The typical usage is top.getMain(), but many iframes define a global variable "main" for convenience.

- tabMenu()

This function returns a reference to the current tabMenu iframe window. The typical usage is top.tabMenu(). Many iframes define a global variable "topMenu" for convenience.

- tabPage()

This function returns a reference to the current tabPage iframe window. The typical usage is top.tabPage(). Many iframes define a global variable "topPage" for convenience.

- model()

This convenience accessor method returns a reference to the core model held in the "main" frame.

- openPage(navigationKey, tabName, keys, extraPageState, keepMemento, forceOpen)

This is a convenience function for the same function defined in cis.js. See the description there for a fuller description.

- getUser()

Returns the user id of the current user.

- getUser()

Returns the user id of the current user.

- getLanguage()

Returns the language code of the current user.

cis.js

The cis.js file contains the bulk of the core framework functions. In addition to defining the major framework classes (CisModel, DataElement, List, etc.) it contains a number of important functions that are described here. These functions are typically invoked by navigating to the main level, e.g. main.xyz(), where the global main has been defined to point to the frame containing cisMain.jsp.

- array_remove_element(array, element)

Remove the indicated element from the given array. Do nothing if it is not found. If the element appears more than once in the array, remove only the first one. Uses simple object comparison (==).

- array_index_of(array, element)

Return the index (0-based) of the given element in the given array. Answer -1 if the element cannot be found. If the element appears multiple times, answer the first (lowest) index. Uses simple object comparsion (==).

- array_remove(array, index)

Compensate for missing functionality in the built-in JavaScript Array class in early versions of JScript. Answers a new array instance with the element at the given index removed. The length of the new array is one less than the length of the given one.

- array_includes(array, element)

Answers a boolean indicating whether the given element is present in the given array. Uses simple object comparison (==).

- array_numeric_sort(array)

Sort the elements of the given array into numeric order, using a simple a < b comparison.

- arrayDo(array, oneArgClosure)

Perform a loop over the elements of the given array, applying the function given by oneArgClosure to each element in turn. The oneArgClosure function takes exactly one argument. Extremely useful for signalling the intention of looping structures. For example, consider this typical code:

```
var max = array.length;

for (var i = 0; i < max; i++) {

var element = array[i];

<do something to element>

}
```

This common structure can be simplified to the following:

```
arrayDo(array, oneArgFunction)
```

where oneArgFunction takes an array element as its parameter, and corresponds to the <do something to element> block above.

- arraySelect(array, selectClosure)

Returns a new array consisting of elements from the given array that return true when applied to the selectClosure (a function that takes one argument). For example, do this to find all numbers greater than 3 in an array:

```
var closure = function(each) {return each > 3};

var resultArray = arraySelect(array, closure);
```

- arrayReject(array, rejectClosure)

Returns a new array consisting of all elements from the given array except those that return true when applied to the rejectClosure (a function that takes one argument). For example, do this to find all numbers not greater than 3 in an array:

```
var closure = function(each) {return each > 3};

var resultArray = arrayReject(array, closure);
```

- arrayDetect(array, detectClosure)

Returns the first element in the given array that returns true when applied to detectClosure, a one-argument function. Return null if no element is found.

- arrayDetectIfFound(array, detectClosure, doClosure)

Similar to arrayDetect(), but proceeds to execute the one-argument function doClosure with the detected element.

- arrayDetectIndex(array, detectClosure)

Similar to arrayDetect(), but answers the index of the first element that satisfies the detectClosure function.

- arrayCollect(array, collectClosure)

Return a new array consisting of the results of applying the collectClosure one-argument function in turn to each element of array. For example, to double the values of an array holding numbers and store the result in a new array:

```
var closure = function(each) {return each * 2};
```

```
var resultArray = arrayCollect(array, closure);
```

- arrayCopy(array)

Answers a new array holding the same elements as the given array. Also known as a "shallow copy".

- arrayContains(array, detectClosure)

Returns a boolean indicating whether the given array contains an element that satisfies detectClosure, a one-argument function returning a boolean.

- arrayUniquePush(array, object)

Similar to array_push, but skips appending the given object if an identical object is already present in the array. Uses simple equality (==) for the comparison.

- configureMainButtons()

Enable or disable the buttons on the main button bar according to the current state of the system. This method should be triggered only if there is a reason to believe that circumstances have left the buttons in an inappropriate state.

- openPage(navigationKey, tabName, keys, extraPageState, keepMemento, forceOpen, extraLoadKeys)

Navigate the system to the tab menu identified by the given navigationKey. The particular tab page may be identified with the given tabName (string), which can also be identified as an integer index (0-based), otherwise the first tab page is used. To display a particular object on the tabMenu/tabPage populate the keys parameter with a key/value object holding the logical keys that identify the object. [The extraPageState parameter is deprecated]. The boolean keepMemento parameter identifies whether a memento should be stored for the current location, e.g. whether it will enter the history menu. If not provided, it defaults to true. The forceOpen boolean (defaults to false) controls whether the system should still show the tabMenu/tabPage if the desired object cannot be read (e.g. it was deleted from the database). The optional extraLoadKeys object is merged with keys prior to performing the read query, allowing the addition or overriding of key values.

- showMessageDescription(categoryNumberFieldName, messageNumberFieldName, shortDescriptionFieldName)

Open an alert dialog box showing a server message (the long message). The category and message numbers are given indirectly via the categoryNumberFieldName and messageNumberFieldName, which are property names in the model from which the actual numbers are fetched. An optional a short description field name can also be provided. The alertClientWindow is a reference to a window object that should act as "host" to the alert dialog box, in order to keep focus on the correct window when the dialog is dismissed.

- basicShowMessageDescription(categoryNumber, messageNumber, shortDescription, alertClientWindow)

Similar to showMessageDescription() as above, but directly accepts the desired category and message numbers rather than retrieving them from the model.

- showErrorMessage(categoryNumber, messageNumber, clientSubstitutions, element, alertClientWindow)

Show the server error message identified by the given category and message numbers, and relating to the HTMLElement given as element (optional), which should receive focus. The alertClientWindow parameter provides an optional reference to the "client" of this dialog that can be useful to preserve the correct focus when the dialog is dismissed. The clientSubstitutions parameter is private.

- restoreElements()

Frequently used in conjunction with setAvailableSignal(), this method clears flags that are used to implement synchronization control.

- doSave(specialActionField, successFunction, forceSave)

Submit a change request to the server using the current model. The specialActionField, if specified, is set to boolean true before submitting the model, without otherwise permanently changing the model. Execute the given successFunction if the save operation succeeds. A "clean" model is not submitted to the server unless the optional forceSafe boolean is true. The browser will refresh itself to show the version of the model as returned from the server, unless the operation resulted in a warning or error. In the first case a dialog shows the warning message line(s), and the user can choose to redo the save operation, ignoring further warnings. In the second case a descriptive error dialog is shown.

- doDelete()

Submit a delete request to the server using the current model. Clears the window if the operation succeeds.

- safelySetFocus(element)

Attempt to set focus to the given HTMLElement.

- updateElementFromModel(htmlElement)

Fetch the current value for the model attribute appropriate to the given htmlElement widget (using the name of the element) and display it.

- setAvailableSignal(aWindow)

Configure the system to accept input after processing a request for the server. Implemented by disabling the cisDisabled.css stylesheet. Typically used in conjunction with restoreElements().

- convertInternalDateTimeToLocal(value)

Convert the given internal date-time value to the user's localized format and answer it.

- convertLocalToInternal(htmlElement, value)

Using the datatype associated with the given html element (as described with its className), convert the given localized value into the internal system value and answer it.

- convertInternalMoneyToLocal(value)

Convert the given money value (a String) into its localized representation and answer it.

- updateField(event)

Update the value of the relevant HTML element based on the given event object. This may involve side effects such as updating the model.

- moneyToWholeInteger(amount)

This utility function helps you do simple arithmetic with money amounts. Since the precision of OUAF system monetary amounts can exceed that offered by the built-in JavaScript number types, we cannot perform arithmetic operations with those numbers without risking loss of precision and rounding problems. The solution is to eliminate the decimal point (if any) in the amounts in order to yield "pure" integers, which admit to exact arithmetic (for addition, subtraction, and multiplication) to very high orders of precision. The final result is then converted back into an internal money amount.

This function accepts an internal money amount (String) and returns a JavaScript "integer" representing the value.

- wholeIntegerToMoney(integer)

Reverse the moneyToWholeInteger operation to yield a monetary amount corresponding to the given integer.

- getInstallationData(key)
- Returns the installation data (string) associated with the given key.
- isUserModified()
- Returns a boolean indicating whether the user has made any changes to the model that should be preserved (e.g. the system will issue a warning if the changes are not saved).

Metadata Overview

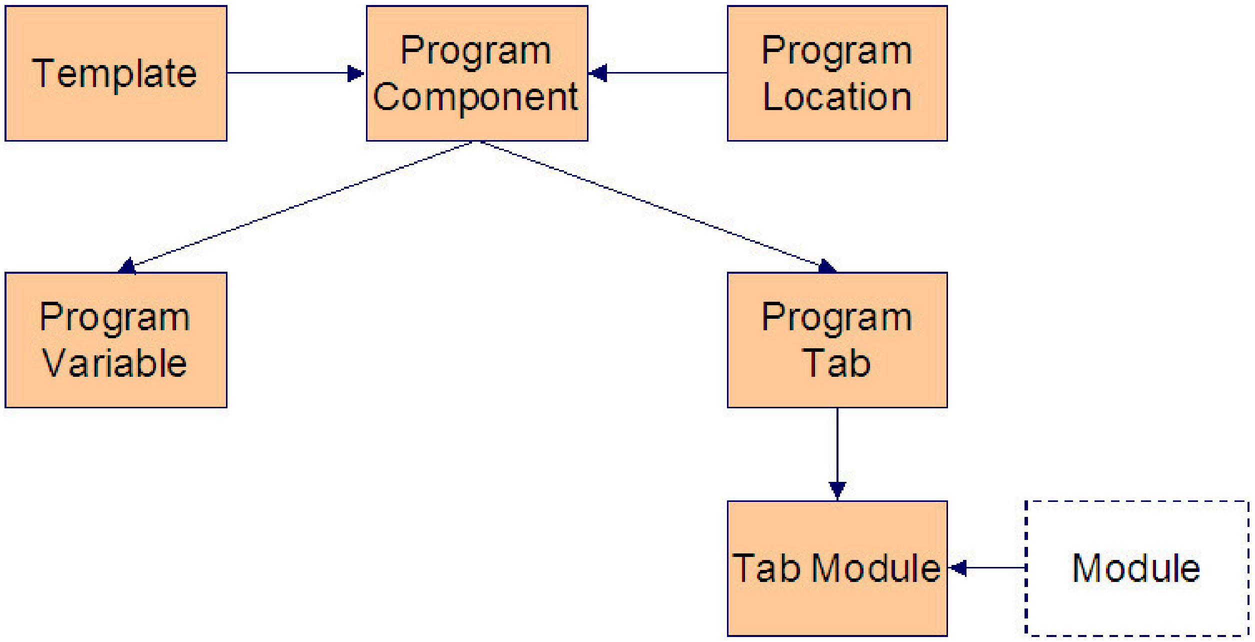
The generation of program components is dependent upon the Oracle Utilities Application Framework meta-data. The meta-data used by framework consists of program variables, program locations, program elements, menu options, navigation keys, tables and fields, and many more.

The meta-data itself can be split into distinct groupings. These groupings will be covered in more detail below.

The basic principle is that a developer enters meta-data for each component to be generated. The generation process applies the meta-data to the generator templates to create the final, deployable component, along with any necessary infrastructure changes. This chapter defines the framework metadata and its inter-relationships.

Generated Tab Menu Metadata

The following entity relationship diagram describes the metadata related to generated tab menus.



Generated Tab Menu Metadata ERD

Note:
The tab menu is represented by program component in the ERD above.

Every framework user interface (UI) transaction has a **tab menu**, which links together the different tab pages that are available on the transaction.

The Software Development Kit generator creates the tab menu using a specific template that is defined in the **template** meta-data. The tab menu's template is maintained from the UI Program Components Object View.

The generated tab menu resides in a certain physical directory in the server's file system. **Location**, the abstract name that represents the actual location of the tab menu, is entered in the UI Program Components Object View. The actual location information and the abstract name are maintained from the Locations Object View.

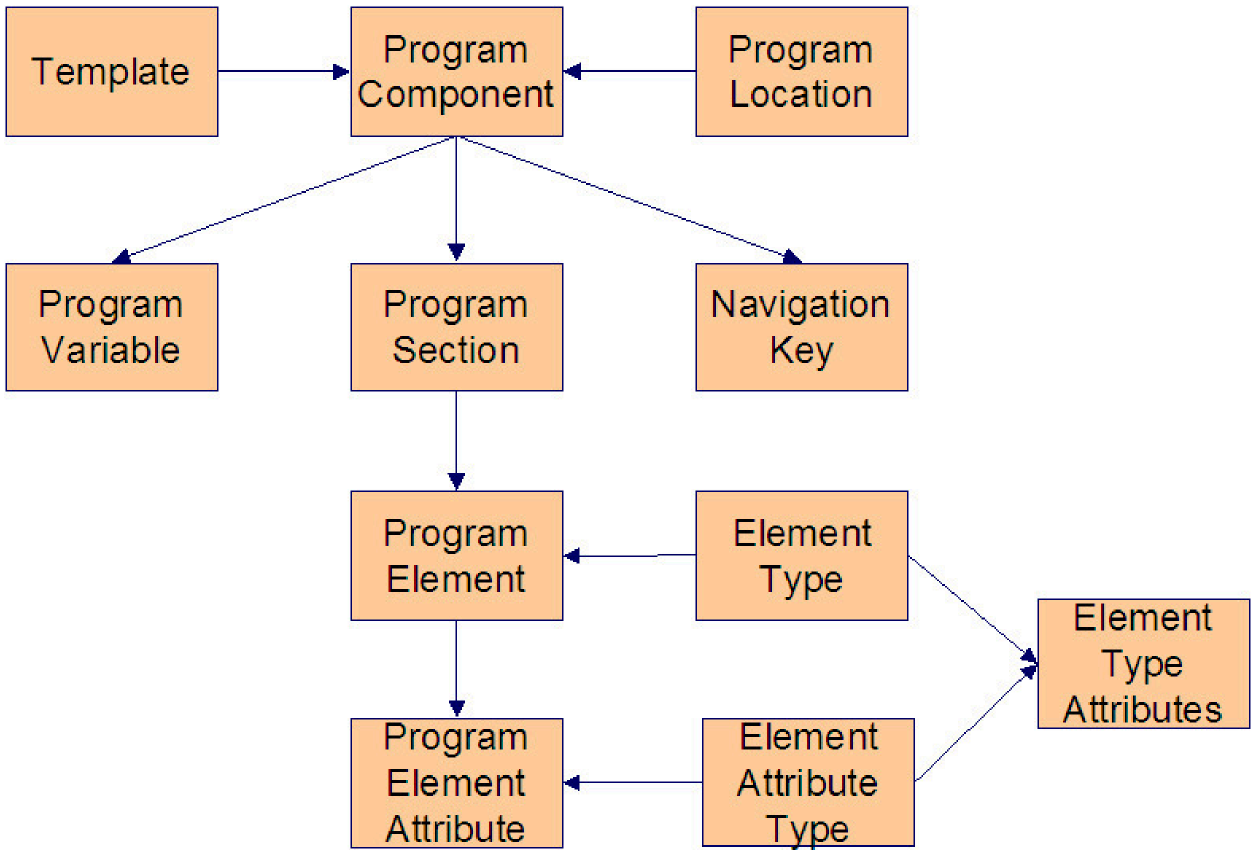
Tab menus may have one or more **program variables** that control its behavior and/or appearance. These variables are maintained from the Program Variable Collection of the UI Program Components Object View.

Tab menus also have one or more **program tabs**, which specify the labels and sequence of the tab pages in the transaction. These tabs are maintained from the Tab Menu Tabs Collection of the UI Program Components Object View.

The use of each tab may be restricted, based on the license key, as specified in **tab module**. This information is maintained from the Tab Module Collection of the Tab Menu Tabs Collection.

Generated UI Program Component Metadata

The following entity relationship diagram describes the meta-data related to generated UI programs.



Generated User Interface Program Component Meta-data ERD

Note:
The UI program component is represented by program component in the ERD above.

Each tab that is specified on the tab menu is linked to a particular **UI program component** - more commonly referred to as "UI page" or "tab page".

Every UI Program Component has a type (e.g. Search Page, List Grid, etc). The Software Development Kit generator uses this information to know how to create a certain type of program. The types are stored in the **template** meta-data table.

Just like tab menus, the generated tab page resides in a certain physical directory in the server's file system. **Location**, the abstract name that represents the actual location of the program, is specified in the UI Program Components Object View.

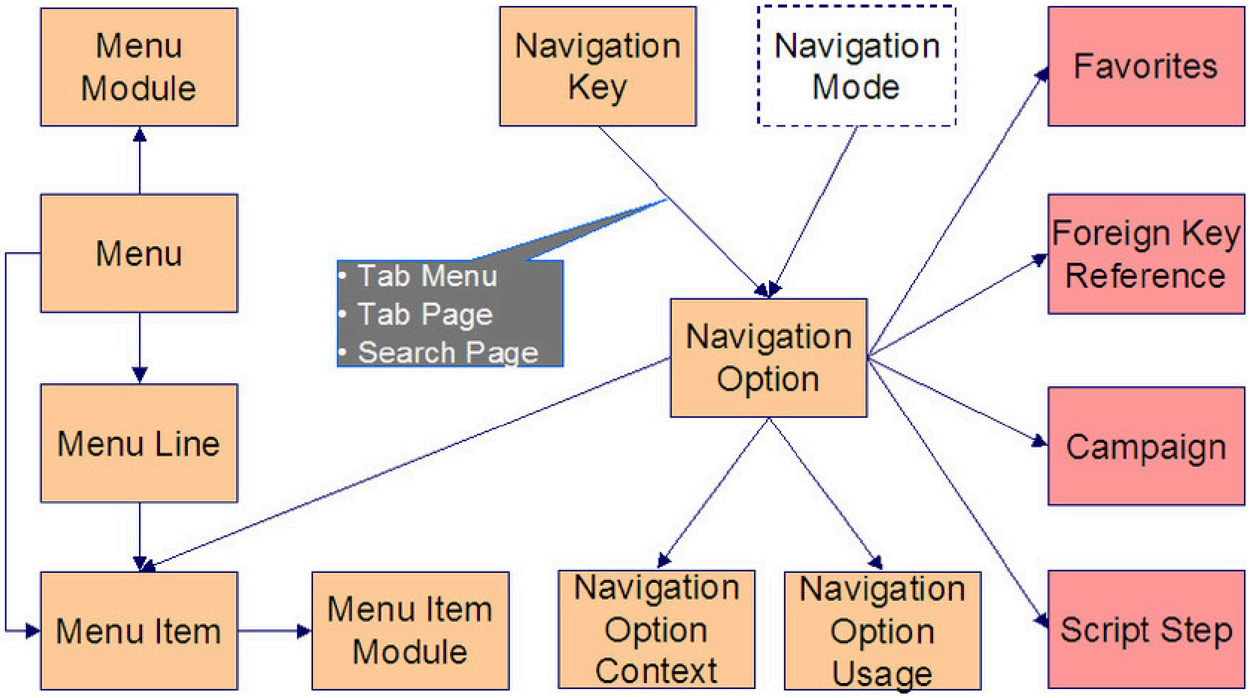
Tab pages may have one or more **program variables** that control its behavior and/or appearance. These variables are maintained from the Program Variable Collection of the UI Program Components Object View.

Each tab page has at least one **program section**. Each section has at least one element of a particular type. A **program element** may have one or more **element attributes** that control its behavior and/or appearance. For example, an element attribute may specify whether or not a field is hidden. Elements may be as simple as input text fields or buttons, and as complicated as trees, grids or graphs. The latter types of elements are actually contained in their own tab page and are referenced from the calling tab page. Please refer to UI Program Components Object View to see how these components are created and maintained.

When UI program components are referred to from other UI program components, instead of referencing these programs by their physical names, pseudo names/aliases (called **navigation keys**) are used. The navigation keys abstract the physical name and location of the program component from the application, making it easier to change and maintain such details. The link between this pseudo address and the actual location of the program is maintained from the Navigation Keys Object View.

Menu and Navigation Metadata

The following entity relationship diagram describes the metadata related to menus and navigation.



Menu and Navigation Metadata ERD

Transactions within framework are accessed through menus. The menu framework uses **navigation options** to define the information required in navigating between transactions. The most important attribute of a navigation option is the Target Navigation Key. This identifies the transaction the navigation option will navigate to.

A **navigation key** is a logical name/pseudo address for UI components. Its prime responsibility is to transform a logical address to a specific URL. The link between this pseudo address and the actual location of the program is maintained from the Navigation Key Object View.

The menu type defines how the menu is used. You have the following options:

- **Main** defines a menu that appears on the menu bar.
- **Admin** is a special type of **Main** menu as admin menu items can be grouped alphabetically or by functional group. Refer to the user documentation for more information about admin menu options.
- **Context** defines a context menu.
- **Submenu** defines a menu that appears when a menu item is selected. For example, the Main menu contains numerous submenus. Each submenu contains the navigation options used to open a page.

Table-Related Metadata

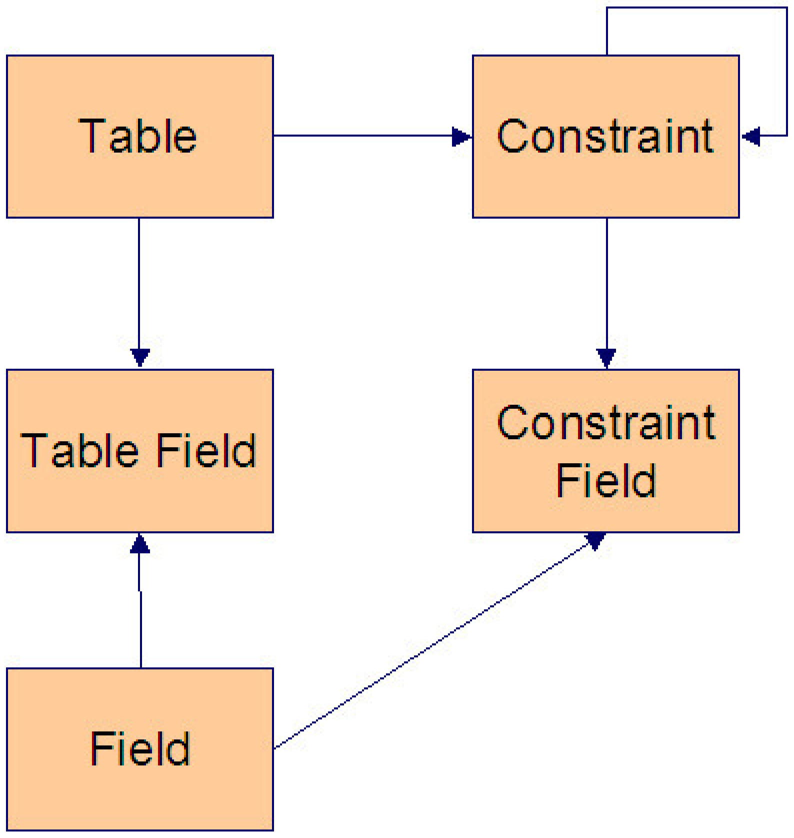


Table Metadata ERD

Table information is used for various purposes. Table information is stored in metadata. This includes which **fields** are on the table (**table field**) and business rules for the fields. **Constraints** define the keys of tables. Constraints also define relationships between tables. **Constraint fields** specify the fields involved in the keys or relationships between tables.

Multi-Language

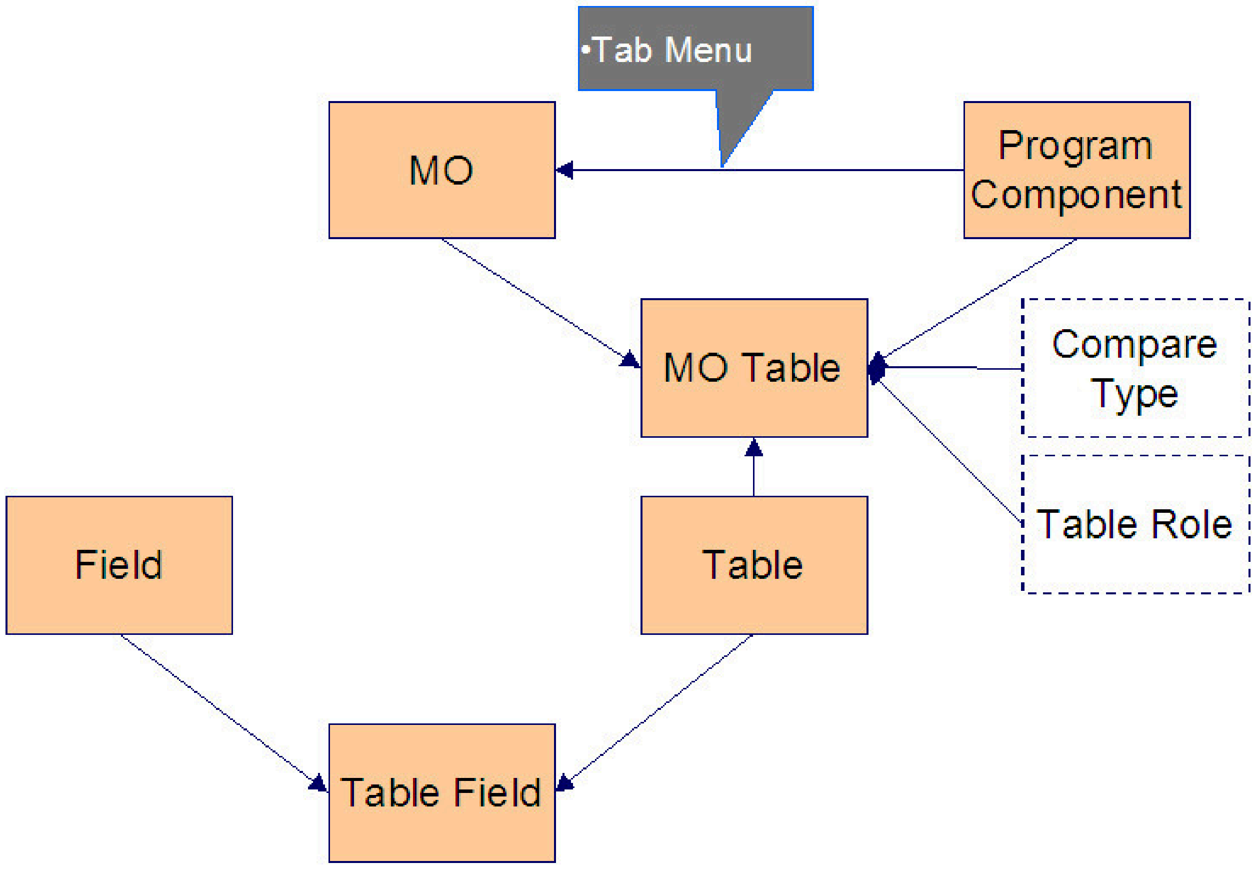
The framework product is available worldwide. This means that the product must be able to display information in many languages.

Some field and table information is language-dependent. Table, table / field and field all have child language tables that hold descriptions specific to each supported language. When field information is retrieved, the system returns not only the base field information, but also the descriptions associated with the user's language.

This process is also used for labels (fields with Work Switch set to **Y**). When a label is needed by the system, the work field information is obtained and the description (from the language table) is displayed on the UI.

Note:
Developers should not use plain text on the UI. All labels should be defined as work fields so that the system will recognize the field and obtain the correct, language-based description to display.

Maintenance Object Metadata



Maintenance Object ERD

A maintenance object (MO) represents a group of tables maintained within framework. These objects are primarily used by the ConfigLab functionality and by the archiving engine to process archiving or purging tasks.

MOs also provide structure from which the various program components (needed to maintain an object) can be created. To be specific, both the front-end user interface (UI) components and the back-end program components can be generated from the MO. The MO specifies the key program component, namely:

- UI Tab Menu

MOs have at least one associated table, e.g. a primary table, sometimes referred to as the root table. In most cases, there are child tables associated with the primary table. E.g. a language table, person name table (child of person table), etc. The Table Role specifies whether the table is a child or a primary.

The Compare Type indicates the comparison method that is used in the ConfigLab functionality. This field is not used for the purpose of building program components from MOs.

MO Table information is maintained from the Maintenance Object Tables Collection of the Maintenance Objects Object View.

Note:
Oracle recommends that customers using Oracle Utilities Application Framework version 4.2 or later and currently using ConfigLab switch to Configuration Migration Assistant (CMA) for their configuration data migration needs and retain ConfigLab for migration of master and transaction data migration. Also note that CMA functionality is not available to every Framework-based product. For details, including tips and requirements for moving from ConfigLab to CMA for configuration data migrations, see the "Configuration Migration Assistant" section in the *Oracle Utilities Application Framework Aministration Guide*.

Defining Generator Tools Metadata

Before generating new programs, you must create the metadata to be used by the generator tools.

Warning:
Please refer to the System Table Guide in the Database Administration Guide for the standard naming convention of each metadata object. Compliance to the standard naming conventions is critical in ensuring the ability to upgrade.

Please refer to the *Oracle Utilities Application Framework Administration Guide* for details about setting up each of these metadata tables.

- Fields (*Database Tools, Defining Fields*)
- Foreign Key References (*Database Tools, Defining Foreign Key References*)
- Lookup Tables (*Database Tools, Defining Lookups*)
- Navigation Options (*User Interface Tools, Defining Navigation Options*)
- Services (*Configuration Tools, Service Program*)
- Tables (*Database Tools, Defining Tables*)
- Menus (*User Interface Tools, Defining Foreign Key References*)
- Maintenance Objects (*Database Tools, Defining Maintenace Objects*)

Development Process

This chapter provides a quick reference for common development tasks. The details are described in the Cookbook chapter.

Hooking into User Exits

Hooking into UI Javascript User Exits

UI pages can have various events extended in order to add to or possibly override base product behavior. To create a Javascript user exit:

- Identify the page to extend.
- Create a JSP extension file (.xjs file) for the given page containing the necessary method for the given action.
- Identify user exits to code.
- Code the desired user exit logic into the JSP extension file.

Hooking into Java User Exits (interceptors)

Interceptors allow additional logic to be executed before or after the invocation of a service. To implement an interceptor:

- Identify the page to extend.
- Identify the interceptor interface to implement.
- Create an interceptor class.
- Code the desired logic into the interceptor class.
- Register the class in CMServiceConfig.xml.

Extending Business Entities

Business entities are the Java representation of persistent data in the system. These objects are transparently initialized and persisted into the database. Many entities are already defined by the base application but may be extended through customization. Likewise, new entities may be created which expose custom tables as business entities.

There are two kinds of hand-coded logic associated with business entities: logic that exposes useful methods to the outside world and logic that is used within the entity itself to perform validation and handle the cascading effects of its changes in state.

Logic exposed to outside callers is what is coded on the business entity's implementation class (the "Impl") class. These "business methods" are then generated onto the entity's "business interface" (e.g. the Person interface). The business interface is the contract that the entity has with other objects.

Quite another thing is how an entity validates and otherwise deals with its changes of state. This is event-driven logic that is not exposed to outside callers and never belongs on the business interface. This type of interface is commonly referred as a "specialization interface" rather than a "business interface" and is coded in change handlers. Unlike a business interface, which receives messages from other objects, a specialization interface is one that provides a mechanism purely for extension of some baseline behavior. In that spirit, the framework design clearly separates the two kinds of code.

Extending the Business Interface

- Create a new implementation class.
- Specify appropriate annotations for an extension implementation class.
- Code business methods.
- Generate artifacts.
- Create a JUnit test.
- Generate artifacts.
- Run JUnit tests.
- Deploy to runtime.
- Test in runtime.

Extending the Specialization Interface

- Create a new change handler.
- Specify appropriate annotations for a change handler.
- Code specialization interface.
- Generate artifacts.
- Create a JUnit test.
- Generate artifacts.
- Run JUnit tests.
- Deploy to runtime.
- Test in runtime.

Creating New Business Entities

Business entities are the object representation of persistent data in the database. To create a new business entity the tables, fields and other meta-data should have already been defined in corresponding meta-data tables. Likewise, the schema objects must already be in the database. Having completed these steps, the business entity is defined to the Java programming and runtime environment by:

Specifying the Business Interface

- Create a new implementation class.
- Specify appropriate annotations for an implementation class.
- Code business methods.
- Create a JUnit test.
- Generate artifacts.
- Run JUnit tests.
- Deploy to runtime.
- Test in runtime.

Specifying the Specialization Interface

- Create a new change handler.
- Specify appropriate annotations for a change handler.
- Code specialization interface, if any.
- Create a JUnit test.
- Generate artifacts.
- Run JUnit tests.
- Deploy to runtime.
- Test in runtime.

Extending Maintenance Classes

The topics in this section describe maintenance class extension procedures.

Maintenance extensions

Not all maintenance logic can go in the initial application's Maintenance. For instance, how can you retrieve the description of a foreign key whose table does not exist in that application?

An "extension" methodology exists whereby an existing page can have behavior added to it at predetermined plug-in points.

This is done by having a list of maintenance extensions that can be supplied for any given maintenance. At runtime, this list is kept and when a maintenance is initialized, new instances of its extensions are created. These extensions are called after any original maintenance behavior, and in the order of loaded applications. This means that the extensions should have no dependence on what other extensions have run, excepting the original maintenance having run.

To extend a maintenance:

- Create a new maintenance extension class.
- Specify the annotations required for a maintenance extension.
- Code desired logic in appropriate methods (see `AbstractMaintenanceExtension`).
- Generate artifacts.
- Code JUnit tests.
- Run JUnit tests.
- Deploy to runtime.
- Test in runtime.

Creating Business Components

Business Components provide a mechanism to provide non-persistent business logic (as opposed to business entities that add to persistent objects). An example business component is as follows:

```
/**
 * Component used to query for {@link Person} instances based on various
 * predefined criteria.
 *
 * @BusinessComponent
 *   (customizationReplaceable = false)
 */
public class PersonFinders_Impl
    extends GenericBusinessComponent
    implements PersonFinders
    /**
     * @param   nameType  a name type
     * @return  count of names by name type
     *
     * @BusinessMethod (customizationCallable = true)
     */
    public int findCountByNameType(Lookup nameType) {
        Query query = createQuery
            ("FROM PersonName name where name.nameType = :type");
        query.bindLookup("type", nameType);


        return (int) query.listSize();

    }
```

To add a new component:

- Create a new implementation class.
- Specify appropriate annotations for a business component implementation class.
- Code business methods.
- Specify appropriate annotations for business methods.
- Create a JUnit test.
- Generate artifacts.
- Run JUnit tests.
- Deploy to runtime.
- Test in runtime.

Plugging in Algorithms

Algorithms provide a powerful and flexible way of extending applications that use the Oracle Utilities Software Development Kit.

Algorithm spots in the application identify different areas that can be extended or customized by implementers. Each algorithm spot defines a set of inputs (typically via set- methods) and output (typically by get- methods).

During implementation, implementers can either re-use existing algorithm types or create new plug-in algorithm. To add a new plug-in algorithm, an implementer will follow these steps:

- Identify the plug-in spot.
- Create an algorithm component.
- Specify appropriate annotations for algorithm component.
- Code the desired logic into the invoke() method.
- Code methods to implement the algorithm spot interface.
- Create a JUnit test.
- Generate artifacts.
- Run JUnit tests.
- Deploy to runtime.
- Create a java class to perform a special plug-in action. This typically would be a modified version of an existing plug-in class. Refer to the algorithm spot definition for the various parameters that

are available. In writing it, look out for possible soft parameters that will add flexibility to the plug-in.
- Add an Algorithm Type to correspond to the new plug-in behavior. This includes naming the java class that was created in the previous step. In addition, the soft parameters that are expected by the algorithm are also defined here.
- Create Algorithm specifying the specific algorithm parameter values where applicable. If the algorithm type is flexible enough, it may end up being reused in multiple algorithms, each having a different set of soft parameter values.
- Add the algorithm to the appropriate control table's algorithms. With this step, the plug-in is available to the application.
- Test in runtime.

Creating Background Processes

To create a background process, there are three important classes that need to be created.

- An implementation of com.splwg.base.api.batch.BatchJob. This is the "driver" and should:
  - Include a "BatchJob" class annotation
  - Extend a generated superclass. In the case where the batch job is named "Foo", the generated superclass will be "Foo_Gen".
- An implementation of com.splwg.base.api.batch.ThreadWorker. This is responsible for processing the work distributed to a processing thread. By convention, this is coded as a static inner class within the BatchJob class implementation described above. If the file becomes excessively large, the worker can be split into its own source file. The worker class extends a generated abstract superclass. In the case of the "Foo" batch job, the worker should be named FooWorker and extend "FooWorker_Gen".
- At least one test class extending com.splwg.base.api.testers.BatchJobTestCase. This class will perform automated tests on the batch process. The runs are performed within the test thread and transaction and all changes are rolled back at the end of the test.

After creating the background process, a corresponding entry should be made in the Batch Control table referencing the created BatchJob's class.

An example batch Job is com.splwg.base.domain.todo.batch.BatchErrorToDoCreation and the test is BatchErrorToDoCreationTest.

Testing Background Processes

BatchJob classes can be tested with JUnit in two ways:

- Extending the BatchJobTestCase class and implementing abstract methods.
- Calling the submitBatchJob(SubmissionParameters) method in any ContextTestCase. This allows testing a mix of one or more background process and other business logic to be tested.

In both of these approaches, the normal commit and rollback logic of BatchJobs is subverted so that all updates performed by the batch process are rolled back when the test completes, either successfully or unsuccessfully. Therefore, these JUnit tests provide a safe way to test batch processes without making irreversible database updates.

Creating MOs and Maintenance Transactions

A typical development of a new MO and its corresponding maintenance transaction entails the following steps:

- Create database objects, e.g., tables, indexes, etc.
- Enter database type of meta-data using online application from the Admin Menu. This includes:
    - Field
    - Table
    - Table/Field
    - Constraints
- Enter MO meta-data using the online system from the Admin Menu.
- Create the entity, changeHandler, and maintenance impl (implementation) classes using Eclipse.
- Generate artifacts based on the impl classes using the Artifact Generator. The artifact generator must also be executed whenever annotations and/or meta-data are changed.
- Add business rules on either the entity or changehandler using Eclipse.
- Create business components, if necessary, in Eclipse.
- Create test classes and then execute JUnit tests in Eclipse.
- If necessary, update maintenance impl class annotation to include fields with derived values using Eclipse. Regenerate artifacts after changing annotation. This generates the service metainfo.
- Add business logic on maintenance impl classes using Eclipse.
- Create maintenance test classes and then execute JUnit tests in Eclipse.
- Create search impl classes using Eclipse.
- Create search test classes and then execute JUnit tests in Eclipse.
- Create a new Maintenance Object from the Admin Menu -> Maintenance Object. This would automaticaly create the Tab Menus and Tab Pages necessary for a new transaction. This will also create the appropriate navigation key for each program component.
- Create javascript user exits for UI program components (e.g. tab menu, tab page, list grid, etc.).
- Add security access to the new application service.

- Create Menu entry for new application service.
- Launch Tomcat server and test the new application service.

Building General Purpose Maintenance Classes

The steps for developing general-purpose maintenance classes are similar to those for MO-based maintenance classes, as described above, but without the need to rely on entity or MO metadata.

- Create the maintenance impl (implementation) classes using Eclipse.
- Generate artifacts based on the impl classes using the Artifact Generator. The artifact generator must also be executed whenever annotations and/or meta-data are changed.
- Create business components, if necessary, in Eclipse.
- Create test classes and then execute JUnit tests in Eclipse.
- If necessary, update maintenance impl class annotation to include fields with derived values using Eclipse. Regenerate artifacts after changing annotation. This generates the service metainfo.
- Add business logic on maintenance impl classes using Eclipse.
- Create maintenance test classes and then execute JUnit tests in Eclipse.
- Create javascript user exits for UI program components (e.g. tab menu, tab page, list grid, etc.).
- Add security access to the new application service.
- Create Menu entry for new application service.
- Launch Tomcat server and test the new application service.

Creating Javadocs for CM Source Code

Information about the Javadoc tool can be found at the following location: https://www.oracle.com/technical-resources/articles/java/javadoc-tool.html. Please refer to the documentation concerning how to write tags, troubleshoot warnings and errors, and any other Javadoc tool questions.

Some known warnings are generated as part of the CM Javadoc process. The following two warnings are safe to ignore:

- The product's annotations currently use tags that are unrecognized by the Javadoc tool. Currently, the Javadoc tool is reporting these as warnings. These warnings are safe to ignore. For the list of tags that are relevant, please refer to the reference guide. For example, the Javadoc tool emits the following warning when it encounters the product's EntityPageMaintenance annotation. It is safe to ignore.

  ```
  warning - @EntityPageMaintenance is an unknown tag
  ```

- The Javadocs tool may also generate warnings that appear from the generated artifacts. These are easily identifiable by looking at the path for the name "sourcegen." For example the following warning can be ignored since path name includes the "sourcegen" directory.

  C:\spl\CCB_PROJ1\java\sourcegen\cm\com\splwg\cm\domain\common\cmCisDiv \CmCisDivisionMaintenance_Gen.java:437

For all other warnings, please refer to the Javadoc documentation.

To generate Javadocs, run the utility script generateJavadoc.bat.

To integrate CM and the product's Javadocs, run the utility script reindexJavadoc.bat to recreate the indices to reflect current environment.

Generate CM Javadocs

Prerequisite: all artifacts need to be generated and the code needs to be compiled without errors.

The first step is to generate Javadocs for CM code. The standard behavior of the Javadoc tool is to create indices that show the packages and classes of the source code that the tool was run on. The resulting indices will only show links to the CM classes and not the product's. To recreate the indices so that they include both the CM and the product's Javadocs, follow the next step.

Recreate the Javadoc Indices

A utility script can recreate the Javadoc indices to include both the CM and the product's Javadocs. The script will scan the files in the Javadoc directory and recreate the indices based on the files that it finds.

Cookbook

Hooking into User Exits

Hooking into Maintenance Class User Exits

Maintenance extensions

Not all maintenance logic can go in the initial application's Maintenance. For instance, how can you retrieve the description of a foreign key whose table doesn't exist in that application?

Therefore, an "extension" methodology needs to exist whereby an existing page can have behavior added to it at predetermined plug-in points.

This is done by having a list of maintenance extensions that can be supplied for any given maintenance. At runtime, this list is kept and when a maintenance is initialized, new instances of its extensions are created. These extensions are called after any original maintenance behavior, and in the order of loaded applications. This means that the extensions should have no dependence on which other extensions have run, excepting the original maintenance having run.

Developing Maintenance Extensions

Maintenance extensions must use the same buffer structure as the original maintenance. The only change allowed is to add possible new default values. Thus a maintenance extension with its annotation might look like this:

```
/**
 * @version $Revision: #1 $
 * @MaintenanceExtension (serviceName = CILTALTP,
 *      newDefaults={ @JavaNameValue (value = TEST, name = test)
 *                  }
 *      )
 */
public class AlgorithmTypeMaintenanceExtension
    extends AlgorithmTypeMaintenanceExtension_Gen {
   }
```

The maintenance extension will have its superclass generated to give easy access to the STRUCTURE definition and HEADER and DEFAULT constants, as well as provide an easy hook for any future functionality that might need to be inserted.

You must use the constants on the STRUCTURE or HEADER structure definitions to reference input header fields or which output fields to populate.

The maintenance extension can then override any methods needed to provide its functionality. Some examples of methods available are:

```
    /**
     * Process a default
     * @param defaultValue the raw string value of the default (can compare
     *     against DEFAULTS constants)
     * @param item the item to be modified with default values
     */
    public void processDefault(String defaultValue, DataElement item) {}

    /**
     * Process the data after the whole add (root and chidren) action is
     *     done.
     * @param originalItem the input item
     */
    public void afterAdd(DataElement originalItem) {}

    /**
     * Process the data after the whole read (root and children) action is
     *     done.
     * @param result the output item
     */
    public void afterRead(DataElement result) {}


    /**
     * Process the data after an element of the given list has been read.
     * @param listName the list name
     * @param outputElement the output element
     * @param sourceEntity the just read entity
     */
    public void afterPopulateElement(String listName,
        DataElement outputElement, BusinessEntity sourceEntity) {}

    /**
     * Process the data after an element of the given list has been changed.
     * @param listName the list name
     * @param inputElement the input element
     * @param changedEntity the changed entity
     */
    public void afterChangeElement(String listName,
        DataElement inputElement, BusinessEntity changedEntity) {}


    // ...
```

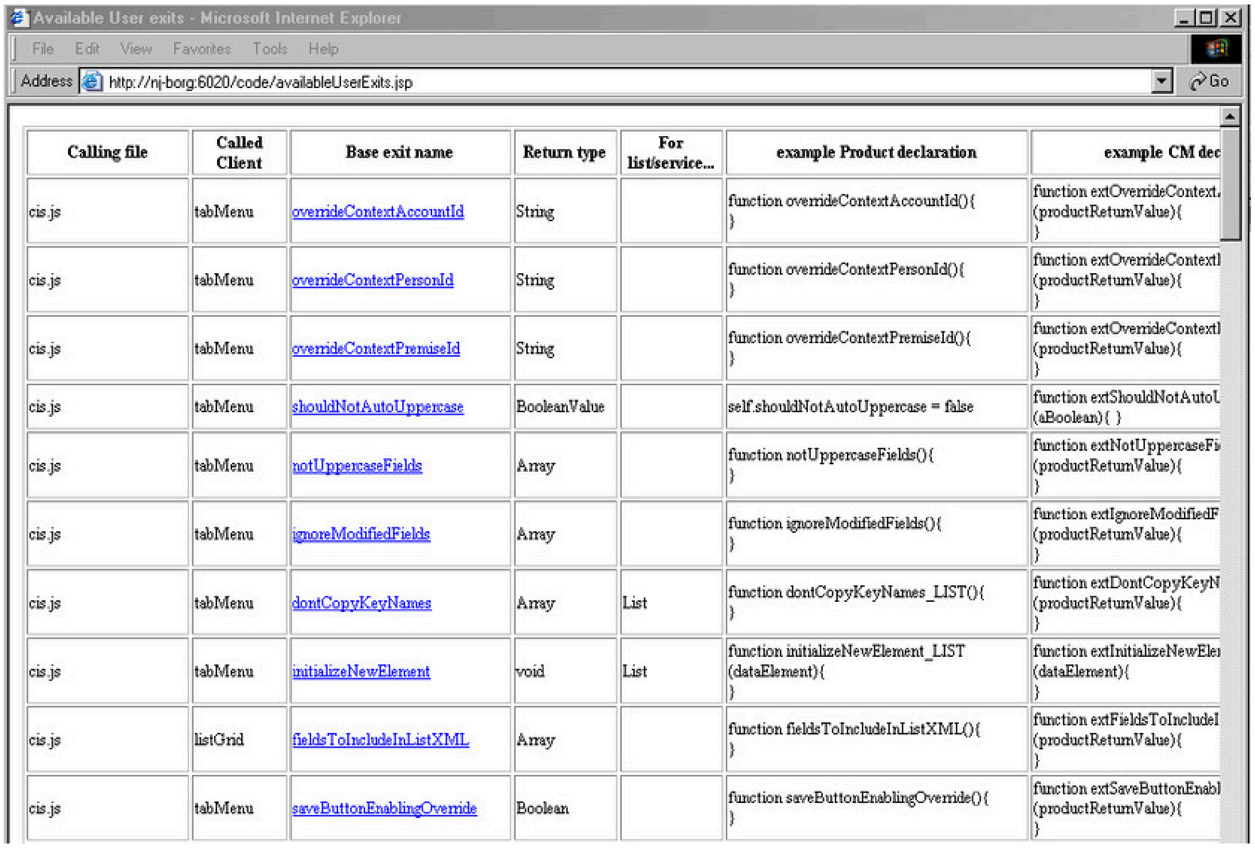
The complete list can be found in the hierarchy of the extension class (e.g., `AbstractMaintenanceExtension`) http://www.python.org/

Hooking into UI Javascript User Exits

The client-side external user exits are designed to give implementers flexibility and power to extend the base package user interface. Implementers have the ability to add additional business logic without changing base html files. These user exits were developed such that developers can create an include-like file based on external user exit templates.

There are two types of client user exits available. There are process-based user exits that wrap the similar product user exit code with pre- and post- external user exit calls, and there are also data-based user exits that simply allow the implementer to add/delete data from the product returned data.

Both types of external user exit are only called if the function exists in the implementer's external include JSP file. All available user exits are listed online in the system through the relative URL: /code/availableUserExits.jsp, with definition examples and links to the Framework code that executes the call.


Available User exits - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://nj-borg:6020/code/availableUserExits.jsp
Go

Calling file	Called Client	Base exit name	Return type	For list/service...	example Product declaration	example CM dec
cis.js	tabMenu	overrideContextAccountId	String		function overrideContextAccountId(){ }	function extOverrideContext. (productReturnValue){ }
cis.js	tabMenu	overrideContextPersonId	String		function overrideContextPersonId(){ }	function extOverrideContextI (productReturnValue){ }
cis.js	tabMenu	overrideContextPremiseId	String		function overrideContextPremiseId(){ }	function extOverrideContextI (productReturnValue){ }
cis.js	tabMenu	shouldNotAutoUppercase	BooleanValue		self.shouldNotAutoUppercase = false	function extShouldNotAutoU (aBoolean){ }
cis.js	tabMenu	notUppercaseFields	Array		function notUppercaseFields(){ }	function extNotUppercaseFi (productReturnValue){ }
cis.js	tabMenu	ignoreModifiedFields	Array		function ignoreModifiedFields(){ }	function extIgnoreModifiedF (productReturnValue){ }
cis.js	tabMenu	dontCopyKeyNames	Array	List	function dontCopyKeyNames_LIST(){ }	function extDontCopyKeyN (productReturnValue){ }
cis.js	tabMenu	initializeNewElement	void	List	function initializeNewElement_LIST (dataElement){ }	function extInitializeNewEle (dataElement){ }
cis.js	listGrid	fieldsToIncludeInListXML	Array		function fieldsToIncludeInListXML(){ }	function extFieldsToIncludeI (productReturnValue){ }
cis.js	tabMenu	saveButtonEnablingOverride	Boolean		function saveButtonEnablingOverride(){ }	function extSaveButtonEnabl (productReturnValue){ }

Miscellaneous How-To's

The following are some how-to examples of typical behavior utilizing some of the standard user exits.

The examples are written for cases of modifying new CM transaction pages, where the function definitions are put into "extended JavaScipt" files (.xjs) that are meant to contain JavaScript user exits directly for a page.

If, on the other hand, an implementer wishes to modify the behavior of a shipped product page, each of the functions below have a corresponding "ext" function that can defined in a /cm/extXXX.jsp file corresponding to the desired page that will fire after any product function call (see above an example of hiding the Sequence column in the algorithm maintenance page).

How Do I Control the Initial Focus Within Tab Pages/Grids/Search Pages?

The system automatically places the initial focus on an appropriate widget (generally input fields) within a Tab Page/Search Page/Grid.

By default it will place focus on the first enabled field with a data class defined on it. (If input fields do not have the Field Name / Table Name defined within Meta Data they will have no data class)

If there are no fields satisfying this criteria within the tab page it will continue to look (recursively) into all the contained frames (e.g. list grids etc.)

If no field is found then no element receives focus.

You can override the default behavior at each level via the provision of a focusWidgetOverride() function within the user exit file which will return the Name of the Field to receive the focus or null.

If null is returned it will ignore all fields within this document and continue to search in lower level documents.

E.G.

From within a Tab Page (If you want focus to go on to a sub document)

```
function focusWidgetOverride() {
  return null;
}
```

From within a List Grid

```
function focusWidgetOverride() {
  return "TD_TYPE_DRLKY:0$TBL_NAME";
}
```

from within a Search Page

```
function focusWidgetOverride() {
return "LAST_NAME";
}
```

Note:
These functions can be as simple or complicated as you want. You could conditionally return a field name or null and this code will run each time the window loads. Also, if a tab page has a popup window or a search window open as it is loading then the initial focus will not be set to the tab page but stay with the popup window

How Do I Mark Fields that Won't Make the Model Dirty?

In certain windows, we have a concept of a "locator" field, which typically acts as a filter on some lists of the object you're looking at. Examples are user group's filter on description, and several IB windows' filter by date.

With the warning about loss of data when throwing away a dirty model, this results in the use of locator fields giving this warning, which wouldn't be expected. In order to avoid this warning on locator fields, you can add a function like the one that follows that enumerates the locator fields:

```
function ignoreModifiedFields(){

    return ['START_DTTM']

}


You can include any number of fields in the array, e.g.

return ['FIELD_1', 'FIELD_2', 'FIELD_3']
```

How Do I Control the Triggering of Defaults After a Search?

If a search returns multiple fields and more than one of these fields can trigger default, then it might be more efficient to only have one of these fields trigger the defaulting.

This is accomplished by creating a new function called **overrideDefaultTriggersFor_SEARCHGROUP** within the tab page that contains the search, where **SEARCHGROUP** is the name of the searchGroup you want to override.

The function must return an object with the triggering field(s) are attributes with a **true** value.

For example

```
function overrideDefaultTriggersFor_SRCH1() {

var triggers = {};


triggers["ACCT_ID"] = true;

            triggers["SA_ID"]=true;
return triggers;

}
```

How Do I Avoid Automatically Setting Fields to Uppercase?

Model attributes that are also key fields are automatically coerced to be in uppercase. You can block this behavior on a field-by-field basis by defining the notUppercaseFields() function in your TabMenu's user exit file to return an array of field names that should not be converted.

Example:

```
function notUppercaseFields() {

   return ['ELEM_ATT$AT_NAME']

}
```

You can also provide a "global" override for an entire TabMenu by setting the shouldNotAutoUppercase variable to true:

```
var shouldNotAutoUppercase = true;
```

How Can I Force the Save Button to be Enabled?

The save button usually synchronizes itself to the state of the model such that if it hasn't been "dirtied" the button is disabled. You may wish to control the state of the save button e.g. because a save should always/never be allowed.

Simply define the function saveButtonEnablingOverride() on your TabMenu user exit file to return a boolean indicating whether the save button should be enabled. You can simply return a literal boolean, or perform any desired processing to determine the return value.

Example:

```
function saveButtonEnablingOverride() {

  return false;

}
```

How Can I Override the Processing After a Change/Add?

If you need to intervene in the processing after the system successfully completes a Change or Add operation, define the function privatePostChangeSucceeded() or privatePostAddSucceeded() in your TabMenu user exit file. The function should return a boolean to indicate whether the system should refresh the UI with the newly returned server data. You'd want to return false if e.g. you navigate to a different TabMenu.

Example :

```
function privatePostAddSucceeded() {

    var model = parent.model;

    var modeFlag = model.getValue('COMPL_NAV_MODE_FLG');

    var navKey = model.getValue('COMPL_NAV_KEY');

    var complSw = model.getValue('CMPLT_CLICKED_SW');

    if (complSw && model.getValue('ENRL_STATUS_FLG') == '30') {

        if (modeFlg && navKey){

             if (modeFlag == 'G') {

                parent.tabPage.gotoContext(navKey);

                return false;

             } else if(modeFlag == 'A') {

                parent.tabPage.addContext(navKey);

                return false;

             }

        }

   }

   return true;

}
```

How Do I Prevent the System from Setting Focus to a Widget After an Error?

When a service receives an error and shows a message after calling a back-end service, the browser attempts to set focus to the relevant widget in error. If you don't need this behavior, you can define the TabMenu variable dontSetFocusOnError to boolean "true.

Example:

```
var dontSetFocusOnError = true;
```

How Do I Prevent Attributes From Being Copied into New List Elements?

The system automatically copies key fields (based on name matching) from a parent list element into new child elements (e.g. created by using the scroll '+' button), in order to keep their prime keys consistent. If you want to inhibit this operation for certain fields, define the TabMenu function dontCopyKeyNames_<LIST NAME> to return an array of fields that should not be copied into new elements of the list named LIST_NAME

Example:

```
function dontCopyKeyNames_ENRL_FLD() {

   return ['SEQ_NUM']

}
```

How Do I Customize New List Elements?

When you use '+' button on a grid or scroll you get a new, empty list element. If you want to customize the object, define a function in the TabMenu's user exit file named initializeNewElement_<LIST_NAME>(newElement).

Example:

```
function initializeNewElement_ENRL_LOG(newElement) {

    newElement.set('ENRL_LOG_TYPE_FLG', 'USER');

    newElement.set('USER_INFO', parent.model.getValue('CURRENT_USER_INFO'));

}
```

How Can I Get My Sequence Numbers to Default Properly on My List Grid?

If you are working with a List Grid that uses some type of sequence field (e.g. SEQNO, LINE_SEQ, SORT_SEQ), there is a handy bit of technology that you can use that will cause the UI to do this job for you.

Just follow the steps below and you'll have the problem solved in no time. The sequence field will be populated in your "empty line" and any elements that are added from then on will have an appropriate value in the sequence field. If the user edits the sequence field at any point, the next element added to the list will incorporate the change without any problems.

Note:
The default Sequence Number functionality will default the next nearest tens value from the highest sequence. The defaulting will do nothing after the sequence reaches the highest number it can hold.

- In the user exit file of the **Tab Menu** - **not** the main Page or the List Grid - copy this JavaScript code:

```
function initializeNewElement_LIST_NAME(newElement) {
      var myListName = "LIST_NAME";
      var myListSeqName = "FIELD_NAME";
      var myListMaxSeq = 999;
      defaultSequenceNumber(myListName,myListSeqName,myListMaxSeq,newElement)
}

</SCRIPT>
<SCRIPT src="/zz/defaultSequenceNumber/defaultSequenceNumber.js"></SCRIPT>
<SCRIPT>
```

- For **LIST_NAME**, substitute your List Grid's list name. Be careful not to lose that underscore [_] just in front of **LIST_NAME** in the first line! Remember that JavaScript is case-sensitive and make sure that you use all UPPERCASE letters as shown here.
- For **FIELD_NAME**, substitute the name of your sequence field, whatever that might be in your List. Don't lose the quotes ["] ! Again, use all UPPERCASE letters.

How Do I Override the Tab Page Shown After an Error in a List (Grid/Scroll)?

When the system receives an error (e.g. after a Save) it attempts to set focus on the relevant widget, which might require flipping to a different tab page. If the error relates to a list (grid or scroll) the system might not choose the tab page you'd prefer. In that event you can control the tab page that should be opened by defining the TabMenu function overrideErrorTabPage_<LIST_NAME>().

Example:

```
function overrideErrorTabPage_BPA() {

    return 'bussProcessAssistantStepPage';

}
```

How Do I Disregard Unwanted Criteria from a Search Triggered by a Search Button?

When a search button (currently implemented as an IMG) is pushed, the system launches a search and "pulls" all applicable criteria values from the current model. It might be that certain criteria fields should be ignored in a particular case. You can define the function addIgnoreFieldsFor_<triggerFieldName>() on a tab or search page's user exit file to specify fields to ignore whenever the IMG button named triggerFieldName is pushed on that page.

The function takes a single argument, fields, and you should add key/value pairs where the key is a field name to ignore, and the value is true.

Example:

```
addIgnoreFieldsFor_ADDRESS1_SRCH = function(fields) {

  fields['CITY_SRCH'] = true

}


addIgnoreFieldsFor_PER_ID = function(fields) {

  fields['ENTITY_NAME_SRCH'] = true

}
```

How Do I Disregard Unwanted Search Result Columns?

When you accept the result of a NOLOAD search the system tries to populate the selected search result row into the current model. Sometimes this doesn't make sense e.g. because there is no corresponding attribute for a display-only column. You can exclude a column from being returned as part of a search result by defining the search client's (Tab Page or Search window) function ignoreResultColumns() in the corresponding page's user exit file. Return an object with keys specifying attributes and values all set to boolean "true".

Example:

```
function ignoreResultColumns() {

    return { ADDRESS1: true, CITY: true, POSTAL: true };

}
```

Since Searches can be shared by many search clients, it is possible that some clients want to get a specific column, but others don't. In that case, define the TabMenu function ignoreResultColumnsFor_<service> as above.

Example:

```
function ignoreResultColumnsFor_CILCCOPS() {

    return {CONT_OPT_TYPE_CD: true}

}
```

How Do I Format a Value Based on a Given Format?

If you need to format a value based on a given format, for example, on Person ID Number, if you select ID Type as SSN (999-99-9999), you can always format the Person ID Number before committing it to the server.

To do so, you can call the formatValue javascript function.

- In the user exit file of the tab page include the following lines:

```



</SCRIPT>
```

```
<SCRIPT src="/zz/formatValue/formatValue.js"></SCRIPT>

<SCRIPT>
```

- Now, you can start using the function to format a value. To use this function, you need to pass in both the **value** and the **format** into the function.

```

var phFormat = myData.getValue(pureListName + 'PHONE_TYPE_FORMAT');

if (pureFieldName == 'PHONE') {

  updateField(pureListName + 'PHONE' ,

        formatValue(myData.getValue(pureListName + 'PHONE'), phFormat));

}
```

Hooking into Java User Exits (interceptors)

Create a class implementing any of the following Interceptor Java Interfaces whenever processing is required before or after the invocation of a service. The CMServiceConfig.xml file contains the mapping between services and corresponding classes that implement pre/post processing plug-ins. The files should reside in the same directory as the service xml files, that is, in the <classpath>/services folder. This can be arranged by placing the files in the web application server's WEB-INF/classes/services folder, or placing them in an existing jar file.

Note:
Note: CM interceptors defined on the CMServiceConfig.xml override base product interceptors on the same service and action.

To implement an interceptor:

- Creating a class implementing any of the Interceptor Java Interfaces.
- Register the class in CMServiceConfig.xml.

Example

The following is a sample interceptor and configuration file, where one interceptor class implements all four interfaces.

Configuration file CMServiceConfig.xml:

```

<ServiceInterceptors

    <Service name="CMLTBTCP">
```

```
        <Interceptor action="add">

            com.splwg.cis.interceptortest.InterceptorTest

        </Interceptor>

        <Interceptor action="change">

            com.splwg.cis.interceptortest.InterceptorTest

        </Interceptor>

        <Interceptor action="delete">

            com.splwg.cis.interceptortest.InterceptorTest

        </Interceptor>

        <Interceptor action="read">

            com.splwg.cis.interceptortest.InterceptorTest

        </Interceptor>

    </Service>

</ServiceInterceptors>
```

Class com.splwg.cm.interceptortest.InterceptorTest:

```
package com.splwg.cm.interceptortest;


import com.splwg.base.api.serviceinterception.IAddInterceptor;

import com.splwg.base.api.serviceinterception.IChangeInterceptor;

import com.splwg.base.api.serviceinterception.IDeleteInterceptor;

import com.splwg.base.api.serviceinterception.IReadInterceptor;

import com.splwg.base.api.service.PageBody;

import com.splwg.base.api.service.PageHeader;

import com.splwg.base.api.service.RequestContext;


public class InterceptorTest implements IAddInterceptor, IChangeInterceptor,

        IDeleteInterceptor, IReadInterceptor {


    public PageBody aboutToAdd(RequestContext context, PageBody in) {

        System.out.println("aboutToAdd: " + in);

        return null;

    }


    public void afterAdd(RequestContext context, PageBody added) {
        System.out.println("afterAdd: " + added);
    };

    public PageBody aboutToChange(RequestContext context, PageBody in) {
        System.out.println("aboutToChange: " + in);
        return null;
    };

    public void afterChange(RequestContext context, PageBody changed) {
        System.out.println("afterChange: " + changed);
    };

    public boolean aboutToDelete(RequestContext context, PageBody in) {
        System.out.println("aboutToDelete: " + in);
        return true;
    }

    public void afterDelete(RequestContext context, PageBody in) {
        System.out.println("afterDelete: " + in);
    }

    public PageBody aboutToRead(RequestContext context, PageHeader in) {
        System.out.println("aboutToRead: " + in);
        return null;
    }

    public void afterRead(RequestContext context, PageBody result) {
        System.out.println("afterRead: " + result);
    }
}
```

Maintaining General-Purpose Maintenance Classes

While most page maintenance classes are actually Entity-based (see Maintaining MO's, below), it is sometimes necessary to write a general-purpose maintenance class for some specific purpose.

To develop such a Page Maintenance class, you need to create a hand-coded implementation class. This class subclasses a class (to be generated) with the same name (and package) as your class, but with the suffix "_Gen". For example, if your class is named `SamplePageMaintenance`, you'll subclass `SamplePageMaintenance_Gen`. Your class must include an annotation providing the metadata that describes the maintenance. This annotation is essentially a subset of the annotation for an entity page maintenance (aka MO maintenance), but leaves out details specific to the entity model (also known as the domain model). For example, the `RowField` annotation is not supported, since it links directly to an entity.

Here is an example of a simple `PageMaintenance` annotation:

```
@PageMaintenance (secured = false, service = CILABCDE,

      body = @DataElement (contents = { @DataField (name = DATA_FIELD1, overrideFieldName = FLD_NAME)

                    , @DataField (DATA_VALUE1)}),

      actions = { "read"},

      header = { @DataField (name = INPUT_FIELD1, overrideFieldName = FLD_NAME)

            , @DataField (name = INPUT_VALUE1)},

      headerFields = { @DataField (name = CONTEXT_NAME1, overrideFieldName = FLD_NAME)

            , @DataField (name = CONTEXT_VALUE1, overrideFieldName = FK_VALUE1)

            , @DataField (name = CONTEXT_NAME2, overrideFieldName = FLD_NAME)

            , @DataField (name = CONTEXT_VALUE10, overrideFieldName = FK_VALUE1)},

      modules = { "foundation"})
```

This example doesn't use any lists, but they are described and supported as are any entity maintenance classes. By definition, any lists here are unmapped--that is, they are not populated by the framework from the entity model.

The supported actions are `read`, `change`, `add`, and `delete`. You can leave out the actions annotation completely if you intend to support all four of these actions. Otherwise, it's useful to declare what methods you'll support, so the framework can create an appropriate error message when an unsupported method is invoked.

You must implement one or more of the following action methods.

```
protected DataElement read(PageHeader header)

protected void change(DataElement item)

protected PageHeader add(DataElement item)

protected PageHeader copy(PageHeader header)

protected void delete(DataElement item)
```

The body of the implementation is completely up to you. The available API is largely the same as for entity page maintenance, e.g. you have a current session/transaction in which to execute queries, can access the entity model, etc. You are expected to throw `ApplicationError` or `ApplicationWarning` Java exceptions, as appropriate (e.g. via `addError()` and `addWarning()`), unless a serious or unforeseen problem occurs, in which case you should throw a `LoggedException`, or simply let the underlying Java runtime exception "bubble up".

The usual implementation of the read method would be to retrieve one or more parameters from the input page header, and construct a `DataElement` holding the desired return values, including any lists (which may be recursive).

For the `change` method, the usual behavior would be to examine the provided `DataElement` object, perform some operation, and return a different data element to hold the "changed" values.

The add method is similar to change in that it accepts an input `DataElement`, and returns the "newly added" `DataElement` instance (which should be a different instance than the input).

The `delete` method accepts a `DataElement`, but returns nothing after the conclusion of the operation.

Maintaining MOs

Maintaining Maintenance Classes for MOs

For a new MO, use the Eclipse plugin to create the skeletal class structure for a new maintenance object class.

For other services not linked to a MO, you will need to write a new maintenance subclass and create the annotation.

To develop an Entity Page Maintenance class, you need to create a hand-coded implementation class. This class must include an annotation providing the metadata that describes the maintenance. In addition, the business entities that "back" the maintenance must already have been created.

Let's take a look at a simple maintenance annotation example to illustrate its properties:

```
@EntityPageMaintenance (

      program = CIPTBTCP,

      service = CILTBTCP,

      entity = batchControl,
      copy = true,

      body = @DataElement (

          contents = {@DataField (DEFAULT_FOR_FLG)

                                , @RowField (name=foo, entity=batchControl)

                                , @ListField(name=BCP, owner=foo, property = parameters)

                           }

             ),

      lists = {@List (name = BCP, size = 50, copy = true,

                               program=CIPTBCPL, constantName="CI-CONST-CT-MAX-FIFTY-COLL",

                               body = @DataElement (contents = {@RowField (batchControlParameter)}))

              }

     )
```

First we see that this tag is an EntityPageMaintenance, meaning it is a page maintenance for a single entity *root* object. Here it is a batch control, but account, person, premise, etc. would also be examples. The idea here is that, by default, the maintenance framework tries to read, save, and delete the *tree* of data that starts with an instance of batch control. (Another kind of page maintenance is EntityListPageMaintenance, where you maintain a list of entities without a single *root* object. It has slightly different attributes than those discussed below.)

Next we list some attributes of the top-level annotation. The required `program` attribute gives the equivalent Converted COBOL Program page module name that we're replacing.

The `service` is the name of the page service that we are implementing. This is required so the Framework knows that it should route requests for this service directly to this Java class.

The required `entity` property names the entity that this maintenance uses for its root. It should match an entity that is defined within the system, else the maintenance obviously can't work!

The copy attribute signals that certain copy fields should be defined in the service. Making the framework explicitly aware of these fields is preferable to "dumb" coding of these fields.

Now we hit the two major structural elements, the `body` and `lists` attributes.

The `body` attribute always resolves to a DataElement, which is simply a way to organize the collection of "rows", "loose fields", and lists that belong to a particular level in the service data structure. These contents are simply held in the `contents` array, which here starts with a simple datafield, DEFAULT_FOR_FLG. Note that you simply reference the field name, and the generator uses the field metadata to infer its type and size. The next element of the `contents` array in this example is RowField. This is essentially a way of naming a reference to the properties of a single entity/table, including its language fields, if appropriate. You need to specify its `entity` and name. Here I use a dummy name, "foo". Finally we have a ListField, which consists of a reference to a list structure that is defined in a separate tag (`lists`). Here we merely name the referenced list by name, provide the owner which matches the name of the "parent" row, and the property, which tells the system how to access the list from the parent. Here we deduce that the `getParameters`() method on a batch control will yield the desired child list.

List Maintenance Classes

Writing a new list maintenance class requires you to create a new class, that provides an annotation with metadata, and lets you implement any user exits you need.

The class should subclass a generated class with the same name, but with suffix **_Gen**.

The annotation is `@ListService`, and is the same annotation structure that is used for lists within PageMaintenance. The service name should end with "L".

If there are child lists, you need to declare them with the `lists` annotation, just like for PageMaintenance.

You should specify any query criteria (from clause, where clause, order by clause) in the List Service annotation, see MaintenanceListFilter on how to implement that.

The default test superclass simply tests that at least one result row is returned.

Here is an example for testing this list maintenance:

```
package com.splwg.base.domain.batch.batchRun;


import com.splwg.base.api.service.ListHeader;

import com.splwg.base.api.testers.ListServiceTestCase;


import java.math.BigInteger;


public class ThreadListInquiry_Test

    extends ListServiceTestCase {

//~ Methods

//-------------------------------------------------------------------------------

    public String getServiceName() {

        return "CILTBTHL";

    }
    /**
     * @see com.splwg.base.api.testers.ListServiceTestCase#getReadListHeader()
     */
    protected ListHeader getReadListHeader() {
        ListHeader header = new ListHeader();
        header.put("BATCH_CD", "TD-BTERR");
        header.put("BATCH_NBR", BigInteger.valueOf(1));
        header.put("BATCH_RERUN_NBR", BigInteger.valueOf(0));

        return header;
    }
}
```

Maintenance List Filters

A given list on a maintenance may not need to return all the data in the list. Instead, a filter can be applied to return a subset of the data. You get the main list by default. And now, you can modify the SQL that will be used for the list retrieval by writing HQL to filter the list. This HQL goes into the `@List` annotation's "fromClause" and "whereClause" properties. This is written as an HQL filter HQL, where the main table (and its language alias, if one exists) already "exists" in the background, and can be referenced by the alias "this" (and "thisLang" for the language row). New entities can be added to the 'from' clause, and a 'where' clause can be specified. A select clause should not be specified (instead results can be added in the bindList user exit - see below), and neither should an order by (the order by is specified separately).

Additionally, if there are extra values that can be retrieved via a join, the loose data fields can be specified as `@ListDataField` , with an hqlPath property specifying the hql path to select the result. And finally, you can bind parameters and also specify extra results into the query in the bindList user exit specific for the given list.

List Maintenance Get More

When a list is too large to send to the UI in one shot, there is the ability to "get more" rows.

An @List can simply have its order set (separate from any other Maintenance List Filter properties). Everything else is automated by the FW. There is no need to add special **LAST_** fields to the annotation/ service, nor even to add the parent's PKs.

The order of a maintenance's list can be given in two ways:

- The order can come from the domain _Impl class. However, this is limited as it may only use fields on the entity table itself (not even language properties).
- The order can be specified on the @List annotation as a string "orderBy", written in hql form, using the special entity alias "this" to refer to the row (and "thisLang" to refer to the language row if one exists), and including any other aliases available from the fromClause property.

The list will retrieve rows in chunk sizes given by the size property on the @List annotation.

An example of using this filtering to join extra information is available on the class `MaintenanceObjectMaintenance`. Another example on a `ListService`, is available on the class `NavigationOptionMenuList`.

Besides using a MaintenanceListFilter and knowing how to deal with list get mores, lists in a page maintenance will automatically retrieve (and cache) the language row associated with the main row of the list. This helps the n+1 select problem (only a single SQL is issued, instead of the main one, plus an extra one for each of the rows' language row), and also provides the ability to have short hand for the orderBy property of a list. If the order is simply by a language property, then you can reference it by `thisLang`.property, without having to supply a filterClause.

Maintaining Maintenance Objects

A maintenance object is a group of tables maintained together within the system.

Note:
For detailed information about Maintenance Objects, please refer to user document *Framework Administration, Database Tools, Defining Maintenance Object Options*

Maintaining Database Meta-data

Maintaining Fields

Field represents a column on a database table. A Field must exist before it can be defined on a Table.

Note:
For detailed information about fields, please refer to user document *Framework Administration, Database Tools, Defining Field Options.*

Maintaining Tables

Table represents a database table used to store data or a database view.

Note:
For detailed information about menus, please refer to user document *Framework Administration, Database Tools, Defining Table Options.*

Maintaining Java Classes

Maintaining Business Entities

Business Entity Background

A central framework concept is the Business Entity that allows for persistent data in the database to be interacted with as objects. In general there is at least one Business Entity class corresponding to each table in the system. Likewise an instance of one of these classes corresponds to a row in the database. Here are some things to remember about Business Entities:

- When you create a new instance of a Business Entity and the current transaction commits, a new row is committed to the database. Likewise when instances are deleted or changed, corresponding deletes or updates are performed on the database. There is no concept of a transient entity in our architecture; application logic is dealing with only persistent objects.
- When the actual insert, update and delete statements are issued to the database is controlled by the framework and may be deferred for performance reasons. The framework is, however, expected to issue DML with sufficient timeliness to maintain data consistency so that application code need not concern itself with when statements are actually executed.
- When you use the query language (HQL) the returned objects are Business Entities (or scalars in the case of "count" or other aggregate functions). These objects may be modified by application code and those changes will be persisted to the database.
- The way you change the properties on entities is via the Data Transfer Objects corresponding to the entity.

How Do I Create a New Business Entity Instance?

Creating a new entity is equivalent to inserting a new row into the database. The first thing you need is to have the framework create a new instance of the correct Data Transfer Object for you so that you can set properties for the new entity instance. This can be done via one of the standard framework methods accessible from the abstract superclasses of classes holding business logic. This method can be told which DTO to create by passing the business interface class for the entity. In the example below, we are creating a new Person_DTO.

```
Person_DTO personDTO = (Person_DTO) createDTO(Person.class);
```

Alternatively, if you find that you have a reference to an Id class, the appropriate DTO can be created via a method generated onto that Id class.

```
Person_Id aPersonId = new Person_Id("123467890");

Person_DTO personDTO = aPersonId.newDTO();
```

Now let's set some values for the new Person instance:

```
personDTO.setStateId(state.getId());

personDTO.setLanguageId(language.getId());
```

Finally we try to create a persistent instance based on these values. This is equivalent to doing the insert against the underlying table except that: (1) required validation occurs and (2) the timing of actual insert occurs at the discretion of the framework.

```
Person person = (Person) createBusinessEntity(personDTO);
```

That's it. When the current transaction is committed, a new person will be added to the database.

How Do I Change Values on an Existing Business Entity Instance?

There are really three steps:

- Ask the existing entity for its DTO.
- Change the appropriate values on the DTO.
- Call setDTO() on the entity instance.

```
Person_DTO dto = person.getDTO();
```

```
dto.setAddress1("invalid value");

person.setDTO(dto);
```

Necessary Change Handlers will fire to validate the change to this "person" object as well as other cascading events as specified in the entity's change handlers.

How Do I Delete a Business Entity Instance?

There are a number of ways to delete entities.

1. Delete an instance that you have a reference to:

   ```
   person.delete();
   ```

2. Delete an instance where you have only its Id:

   ```
   delete(personId);
   ```

3. Delete the results of a query

   ```
   Query query = createQuery("from Person person where exists ( "

       + " from PersonName as perName where person = perName.id.person and "

       + "perName.isPrimaryName = :systemBool and perName.entityName "

       + "like :name)");

    query.bindLikableString("name", "ABC", 64);

    query.bindBoolean("systemBool", com.splwg.base.api.datatypes.Bool.TRUE);


    long rowsDeleted = query.delete();
   ```

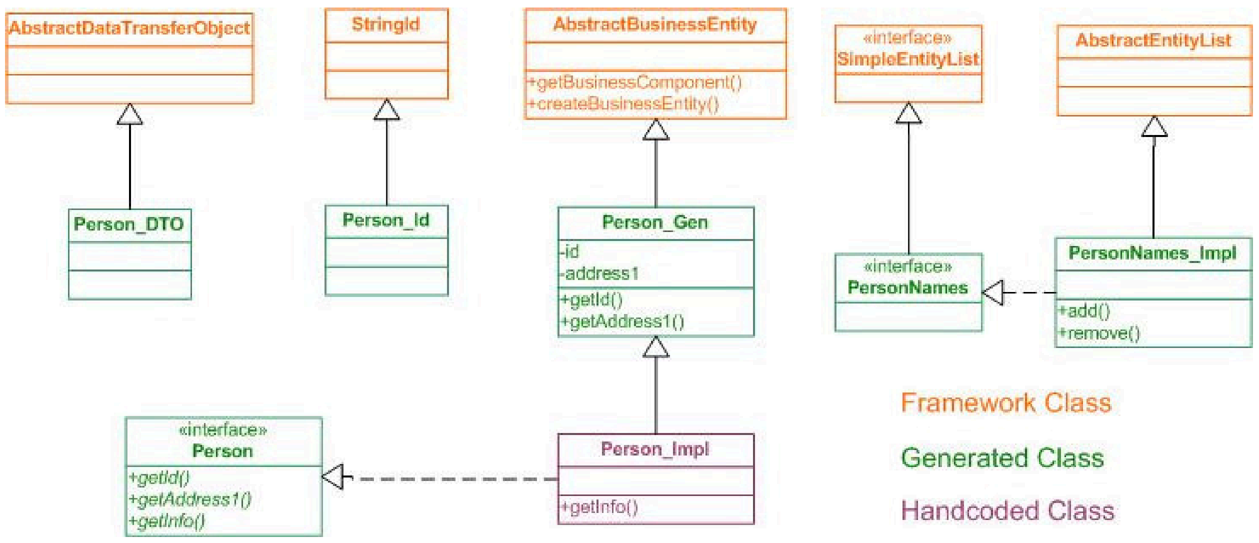
Persistent Classes

Behind the scenes, the persistence and validation mechanisms are quite complex and require the collaboration of many classes and pieces of configuration data. Thankfully, most of this complexity is hidden from the application programmer. Still, there are various classes that the application programmer will deal with:

- Framework Classes that act as an application programming interface. These API classes are directly referenced by application code.
- Generated Classes that are created for each business entity that serve two purposes:

    - They provide convenient methods (like property "getters" and "setters") based on the structure of the specific entity, it's fields, child collections and key structure for example.
    - They are necessary for the persistence mechanisms to work correctly.
- Handcoded classes that the application programmer is expected to write. Many of the handcoded classes are read by the artifact generator so the framework can "wire up" the handcoded functionality.

Some examples of the above classes are shown below.



Creating the Implementation Class

There is very little that needs to be done by application developers to create a basic business entity. In addition to the setup of the CI_MD_* tables describing the entity and its constraints only an implementation class (or "Impl" for short) needs to be added. In this case a developer added Person_Impl. The following is a simple example of an "Impl" class for the Person entity.

```
/**
 * @BusinessEntity
 *   (tableName = CI_PER,
 *    oneToManyCollections  = { @Child( collectionName = names,
 *                                 childTableName = CI_PER_NAME,
 *                                 orderByColumnNames = { SEQ_NUM } )
 *                            }
 *   )
 */
```

```
public class Person_Impl

    extends Person_Gen {

    /**

     * @return the UI Display "info" for this person

     */

    public String getInfo() {

        return "PrimaryName: " + getPrimaryName().getInfo();

    }

}
```

Important parts of the implementation class are described below:

- **The implementation class name must end with the suffix "_Impl".** For example, if the entity has a name of "person" then the implementation class name of "some.package.Person_Impl". It also means that the generated business interface will have a name of "some.package.Person".
- **A Class Annotation** which declares:
    - What table this entity represents
    - What the owned-child tables are and what they should be called
    - Other information. Please see the BusinessEntityAnnotation class for more details.
- **The class extends an abstract superclass having the suffix of "_Gen".** Continuing the example of an entity named "person", the implementation class would extend a not-yet-created abstract superclass named "some.package.Person_Gen". This superclass is created by the artifact generator based on metadata about the table and contains:
    - Getter methods for properties including parent objects and collections
    - The getDTO() and setDTO(...) methods that allow for properties to be changed
    - Access to standard framework methods like createQuery(...)
- **Business methods.** Any hand coded public methods are automatically exported onto the generated business interface (e.g. "some.package.Person"). Client code can then access the added business method as follows:

```
Person aPerson = some logic retrieving a person instance

String thePersonsInfo = aPerson.getInfo();
```

- **Constants.** Any hand coded public static final variables are automatically exported onto the generated business interface. This will be useful for constants related to the entity.
- Having created a new entity, it is likely that validation rules and other behaviors should be added to it. Please see Adding Change Handlers for more information.

Developing Change Handlers

The creation of Business Entities allows business logic to interact with rows in database tables as objects and in doing so allows business methods to be invoked on those objects to perform some business function. Quite another thing is how the entities *react* to proposed changes in their state. Outside callers have no business being exposed to the internal validations and cascading state changes within the objects that they interact with. Because of object encapsulation, they should not be exposed to such issues. Nonetheless, there needs to be a way to program the internal logic of entities. This is the reason for Change Handlers, to provide for objects to react to proposed changes in their state.

Change Handlers are classes that add behavior to entities. This behavior takes two forms.

- **Validation rules.** This allows for proposed changes to be validated against business rules. These rules are expected to be "side effect free" meaning that the validation does not change the state of the system. By calling side effect free validations only after all changes to entity state have been performed, the framework can avoid many complex scenarios where invalid data can "slip past" validations.
- **Cascading change logic.** This allows changes to this entity to cause changes to other entities.

Creating the Change Handler Class

A Business Entity may have more than one Change Handler. The framework will call each handler associated with an entity when an attempt is made to modify the state of the underlying entity. The following are the important parts of a Change Handler class:

- The class should extend the AbstractChangeHandler class and have a class name ending with "_ CHandler".
- The @ChangeHandler class annotation. This tells the framework which entity to attach the change handler to at runtime.
- Implement any "handle" methods. These are methods that can implement any cascading effects of the proposed change to the entity's state.
- Construct Validation Rules that are returned by public static methods on the change handler class. There should be one static method per rule. The reason for exposing these methods is to facilitate

testing (see below). Static methods are used instead of static variables to prevent timing problems associated with the static initialization of static variables.
- Return an array of the rules created above via the getValidationRules() method. The framework invokes this method at runtime to retrieve the rules.
- Make sure to run the artifact generator and rebuild source code after adding a Change Handler or modifying its annotation.

Testing the Change Handler Class

When adding behavior to an entity, it is desirable to do the following:

- Break the rules into modular pieces that can be independently maintained and tested.
- Test that each behavior works by creating a JUnit test for each distinct behavior.

The following steps are recommended when adding new change handlers so that the additional behavior is sufficiently tested.

- Add each rule to the change handler at once using instances of the PlaceHolderRule class. Use an appropriate RuleId and Description as self-documentation of what the rule is supposed to do.
- Add a new test class by extending AbstractEntityTestCase. This class should reference the change handler being added and will insure that each rule is violated by at least one test. The test class name should end with "Test".

- Run the test class as a JUnit test. The test class should complain that there was at least one rule that was not violated by the test class. For the rule that was not violated, add a test method to the test class and also add the "real" validation logic to the change handler class. Try executing the test class again. Continue implementing more test methods and rules until all rules are tested and the JUnit class completes successfully. Below is an example, test method for a rule that tests both a successful change and an unsuccessful change. It is important to insure that the validation error is thrown by the actual rule being tested.

```
public void testAddressOneLabelRequiredIfAddressOneIsAvailable() {

   //pass

   Country country = (Country) createQuery(

       "from Country country").firstRow();

   Country_DTO countryDto = country.getDTO();

   country.setDTO(countryDto);


   //fail
countryDto.setAddress1Available(Bool.TRUE);

countryDto.setLanguageAddress1("");

try {

    country.setDTO(countryDto);

    fail("A validation error should have been thrown");

} catch (ApplicationException e) {

  verifyViolatedRule(Country_Chandler

      .addressOneLabelRequiredIfAddressOneIsAvailable(), e);

}

}
```

- Add other test methods to test "handle" methods on the change handler as well as business methods that may have been added.

Validation Rules

Validation rules are the mechanism for describing to the runtime system how it should validate business entities. There are a few important characteristics of these rules:

- The coding style is **declarative**. That is, every attempt has been made so the programmer specifies **what** makes data valid, not **how** or **when** the validation should take place.
- Only in the case of "custom rules" does the programmer need to build the step-by-step logic specifying how the validation should take place.
- Validation rules are **side-effect free**. That is, they cannot change the persistent state of the system. This insures that all the validations are performed on the complete set of changes. Likewise, it allows for the startChange()/saveChanges() logic to safely defer the firing of rules until the end of the coherent set of changes.

The Rules

A number of useful rules are provided in the interest that the application programmer can use them with a minimum of programming. These are classes that implement ValidationRule and can be used by application logic:

- **ProtectRule** will protect one or more properties on an entity.
- **RequireRule** will require that a property be populated.
- **AllowRule** allows a value to be populated.
- **AllowAndRequireRule** both allows and requires that a property be populated.

- **DecimalRule** provides some common validations against decimal data types.
- **CustomRule** will create a rule out of a **CustomValidation** class implementing logic that just cannot be handled by existing rules.
- **RestrictRule** will restrict a property to a set of values

Each of the rules above provides **standard rules** that represent similarly configured rules that are used repeatedly in the system. These standard rules can be created via static "factory" methods on the rules themselves. Consider the following standard rule:

```
/**
 * Protect the dependant property when the primary property is equal to the supplied lookup value.
 *
 * @param ruleId a unique ruleId
 * @param description a description
 * @param primaryProperty the property that the condition depends on
 * @param dependantProperty the property that is protected when the condition is true
 * @param primaryLookupValue a {@link Lookup} value that the primary property must equal for the dependant property
 *                             to be protected
 * @return a new rule
 */
public static ProtectRule
   dependantPropertyWhenPrimaryMathesLookup
        (String ruleId,
         String description,
         SingleValueProperty primaryProperty,
         SingleValueProperty dependantProperty,
         Lookup primaryLookupValue)
```

What this rule does is prevent one property from being changed (the "dependant" property) when another property (the "primary" property) matches a certain value. An example would be the "freeze date/time cannot be changed when the status is 'frozen'". In this case, the dependant property would be the freeze date/time and the primary property would be the status. The lookup value of "frozen" would be passed in as the lookup value.

Custom Rules

There are situations when custom rules need to be coded. These are for situations too complex for a declarative rule. The process is as follows:

- Create a class that extends AbstractCustomValidation. Implement one or more of the abstract methods corresponding to various "events" that may occur with respect to the underlying entity.
- Within any change handler requiring this rule, instantiate a CustomRule passing in the class created above.

  For details on the "events" that can be processed by the custom validation, please refer to the JavaDocs.

  When coding a CustomValidation that will be used by a CustomRule. It is important to understand when these events fire.

- **Eager Validations** fire "immediately" when the underlying entity is changed (either via delete, setDTO(), or createEntity()).

  validateAdd() fires only on an add

  validateChange() fires only when an existing entity is changed

  validateAddOrChange fires **in addition to** either validateAdd() or validateChange()

  validateDelete fires when the entity is being deleted

  validateRegisteredChange() fires when "some other object is changed" (like a child). This can be any random entity instance that feels like notifying you regarding a change of state or, more commonly, the framework automatically registers a change when a child collection is manipulated. Your custom code can determine if a change has been made to a child it's interested in by calling the getChangeToList() method on the change detail passed in. You just pass in the class of your child collection and it passes back changes, if any.

- **Lazy Validations** fire when a "coherent set of changes" is complete.

  validateSave() can be used to implement validations that needs to be performed "at the end" of some set of changes. By default a set of changes is both started and saved within individual calls to setDTO or createBusinessEntity, etc. However, this can be controlled programmatically by calling the startChanges() and saveChanges() methods that are available from within all business objects (change handlers, entities, components, etc). Any type of change (add, change, deleted, register change) will trigger validateSave().

Conditions

The rules wouldn't be very useful if all you could do was *always* protect or require properties. This behavior is usually based on conditions. Rules take as input one or more Conditions (e.g. objects implementing the Condition interface). Right now, there are several conditions that can be used:

- **Equals**. This condition can compare properties to each other or to constants (lookup values, Strings, etc). Likewise, the size of a collection can be compared using Equals (e.g. determine personNames' size equals 0 would mean there are no names for a person). Finally, null values can be tested using a special constant value "Constant.NULL".
- **Not**. This is the basic boolean operator that can change the value of other conditions.
- **And**. This is the basic boolean operator that takes two child condtions, and return true if each of them are true. This is evaluated "lazily" and won't even evaluate the second condition if the first is false (a performance enhancement).
- **Or**. This is the basic boolean operator that takes two child conditions, and return true if either of them are true. This is evaluated "lazily" and won't even evaluate the second condition if the first is true (a performance enhancement).
- **GreaterThan** / **GreaterThanOrEquals**. This evaluates whether one property/constant is greater than (or greater than or equal to) to another property/constant.
- **LessThan** / **LessThanOrEquals**. This evaluates whether one property/constant is less than (or less than or equal to) to another property/constant.
- **Contains**. These are conditions for a collection of children- at least one element has condition x, at most 2 elements match condition y, etc). The child condition's properties should be referenced from the point-of-view of the child row.

Each of these conditions is accessible from the corresponding property or condition. There should be no reason in normal development to use the constructors for the conditions above. Instead, you could say, for instance

```
Condition isPrimaryName = PersonName.properties.isPrimaryName.isTrue();
```

or

```
Condition isAlias

   = PersonName.properties.nameType.isEqualTo(NameTypeLookup.ALIAS);
```

or

```
    Condition greaterThan

       = PersonName.properties.sequence.isGreaterThan(BigInteger.ZERO);
```

or

```
Condition hasOnePrimaryName

   = Person.properties.names.containsAtLeastOne(isPrimaryName);
```

or

```
Condition notAlias = isAlias.not();
```

Change Handler Helpers for Maintenance Objects

The Business Object based Maintenance objects have some standard validations. The Helper classes described below will help in reusing the validation code. The Change Handler Helpers have been created for the following objects:

- BO based MO
- Standard MO Log table
- Standard MO Log Parameter table

BO-Based MO

The `com.splwg.base.api.maintenanceObject.BOBasedMaintenanceObjectCHandlerHelper` can be used for the BO-based MO Change Handler validations.

The following standard validations are provided by this helper class:

- The Business Object cannot be changed
- The Business Object must be for the correct MO
- Status is required if the Business Object has a lifecycle
- Status must be a valid lifecycle status

The following methods are provided by this helper class:

- Adding log entries for entity creation and entity status change (if the MO does not already have a transition algorithm)

To use the validations, create an instance of the helper class in the MO Change Handler and get the validation rules from the Helper, as illustrated in the following sample code.

Change Handler Sample Code

```
public class OutboundCrossReferenceMessage_CHandler extends AbstractChangeHandler<OutboundCrossReferenceMessage>

{

private final BOBasedMaintenanceObjectCHandlerHelper helper = new BOBasedMaintenanceObjectCHandlerHelper<

OutboundCrossReferenceMessage> ( new MaintenanceObject_Id("O2-OXREFMSG"),

OutboundCrossReferenceMessage.properties,

OutboundCrossReferenceMessage.properties.lookOnBusinessObject() );

...

public void handleAddOrChange ( OutboundCrossReferenceMessage changedOutboundCrossReferenceMessage,

DataTransferObject< OutboundCrossReferenceMessage> oldDTO )

{

helper.handleAddOrChange(changedOutboundCrossReferenceMessage, oldDTO);

}

public ValidationRule[] getValidationRules()

{

return helper.getValidationRules();

}

}
```

Change Handler Junit Test Code

```
public void testBoCannotBeChanged()

{

startChanges();

OutboundCrossReferenceMessage_DTO dto = (OutboundCrossReferenceMessage_DTO)

createDTO(OutboundCrossReferenceMessage.class);

dto.setBusinessObjectId(new BusinessObject_Id("ZZ-OXREFBO"));

setDtoData(dto);

OutboundCrossReferenceMessage outboundCrossReferenceMessage = dto.newEntity();

saveChanges();

try

{

dto.setBusinessObjectId(new BusinessObject_Id("ZZ-CASE"));
outboundCrossReferenceMessage.setDTO(dto);

saveChanges();

fail("An error should have been thrown");

}

catch (ApplicationError e)

{

verifyViolatedRule(BOBasedMaintenanceObjectCHandlerHelper.boCannotBeChanged(

OutboundCrossReferenceMessage.properties, OutboundCrossReferenceMessage.properties

.lookOnBusinessObject()), e);

}

}
```

Standard MO Log Table

The `com.splwg.base.api.maintenanceObject.MaintenanceLogCHandlerHelper` can be used for the Standard MO Log Table Change Handler validations.

The following standard validations are provided by this helper class:

- Log entry cannot be deleted if of type:Created, Exception, Status Transition, Status Transition Error, System, User Details
- Validates the characteristic value possibly stored on the log entry
- Long description or message are required, but not both
- User details must provide long description

The following default methods for Add are provided by this helper class:

- Log dateTime to system date time
- User to current system user
- Log sequence to next highest number
- Status to the parent entity's status (will warn if specified input differs from parent status)

To use the validations, create an instance of the helper class in the MO Change Handler and get the validation rules from the Helper. First you must create a new Characteristic Entity in the Characteristic Entity Lookup for your object. This entity can be selected on any Characteristic Type to indicate that that characteristic type can be used in the log messages for this log. Sample code follows.

Change Handler Sample Code

```
public class OutboundCrossReferenceMessageLog_CHandler extends AbstractChangeHandler<OutboundCrossReferenceMessageLog>

{

private final MaintenanceLogCHandlerHelper helper = new MaintenanceLogCHandlerHelper<OutboundCrossReferenceMessageLog,

OutboundCrossReferenceMessage>(new MaintenanceObject_Id("O2-OXREFMSG"), OutboundCrossReferenceMessageLog.properties,

OutboundCrossReferenceMessageLog.properties.lookOnParentOutboundCrossReferenceMessage(),

CharacteristicEntityLookup.constants.OUTBOUND_CROSS_REFERENCE_MESSAGE_LOG);

public void prepareToAdd(DataTransferObject<OutboundCrossReferenceMessageLog> newDTO)

{

helper.prepareToAdd(newDTO);

}

public void prepareToChange(OutboundCrossReferenceMessageLog unchangedEntity,

DataTransferObject<OutboundCrossReferenceMessageLog> newDTO)

{

helper.prepareToChange(unchangedEntity, newDTO);

}

public ValidationRule[] getValidationRules()

{

return helper.getValidationRules();

}

}
```

Change Handler Junit Test Code

```
public void testLongDescRequiredIfLogTypeIsUserDetails()

{

startChanges();

OutboundCrossReferenceMessage_DTO dto = (OutboundCrossReferenceMessage_DTO) createDTO(OutboundCrossReferenceMessage.cla

ss);

dto.setBusinessObjectId(new BusinessObject_Id("ZZ-OXREFBO"));

outXrefMsgTest.setDtoData(dto);

OutboundCrossReferenceMessage outboundCrossReferenceMessage = dto.newEntity();

OutboundCrossReferenceMessageLog_DTO outboundCrossReferenceMessageLogDto1 = (OutboundCrossReferenceMessageLog_DTO)

createDTO(OutboundCrossReferenceMessageLog.class);

outboundCrossReferenceMessageLogDto1.setLogEntryType(LogEntryTypeLookup.constants.USER_DETAILS);

try

{

outboundCrossReferenceMessage.getLogs().add(outboundCrossReferenceMessageLogDto1, null);
saveChanges();

fail("An error should have been thrown");

}

catch (ApplicationError e)

{

verifyViolatedRule(MaintenanceLogCHandlerHelper

.longDescriptionIsRequiredIfLogTypeIsUserDetails

(OutboundCrossReferenceMessageLog.properties), e);

}

}
```

Standard MO Log Parameter Table

The `com.splwg.base.api.maintenanceObject.MaintenanceLogParameterCHandlerHelper` can be used for the Standard MO Log Parameter Table Change Handler.

The following standard methods on Add are provided by this helper class:

- Parameter sequence to next highest number

To use the Helper, create an instance of the helper class in the MO Change Handler and get the validation rules from the Helper, as illustrated in the following sample code.

Change Handler Sample Code

```


public class OutboundCrossReferenceMessageLogParameter_CHandler

extends AbstractChangeHandler<OutboundCrossReferenceMessageLogParameter>

{

private final MaintenanceLogParameterCHandlerHelper helper = new MaintenanceLogParameterCHandlerHelper<

OutboundCrossReferenceMessageLogParameter, OutboundCrossReferenceMessageLog>(new MaintenanceObject_Id("O2-OXREFMSG"),

OutboundCrossReferenceMessageLogParameter.properties,

OutboundCrossReferenceMessageLogParameter.properties.

lookOnParentOutboundCrossReferenceMessageLog());

...

public void prepareToAdd(DataTransferObject newDTO)

{

helper.prepareToAdd(newDTO);

}

public ValidationRule[] getValidationRules()
{
return helper.getValidationRules();
}
}
```

Additional Validations

Using Helper Class Validations Only

If only validations from the helper class are required, use the Change Handlers `getValidationRules()` method to return the `helper.getValidationRules()`. This will enforce all the validations in the Helper class on the Change Handler.

Sample code:

```

public ValidationRule[] getValidationRules() {
return helper.getValidationRules();
}
```

Using Helper and Change Handler Validations

Create an Array of ValidationRules in the `getValidationRules()` method of the Change Handler and pass this array to the `helper.getValidationRules()` method. The Helper class adds the rules passed to it to the standard set to provided validations.

Sample code:

```

public ValidationRule[] getValidationRules() {
return helper.getValidationRules(<Array of Validation Rules>);
}
```

Maintaining Business Components

Business Components are business objects having two important characteristics.

- They are **non-persistent** holders of business logic. That is, they are the place to put business logic not tied to a single business entity instance (e.g. a single "Account" or "Person".) This makes them analogous to "common routines".
- When allowed, implementations of business components may be replaced at runtime by custom classes implementing the same business interface. An example of this includes "info" logic.

Creating Business Components

Much like Business Entities, it is necessary to create an implementation (*_Impl) class containing the actual logic that is then processed by the artifact generator. Below is an example that would be created by hand:

```
/**
 * Component used to query for {@link Person} instances based on various
 * predefined criteria.
 *
 * @BusinessComponent
 *   (customizationReplaceable = false)
 */
public class PersonFinders_Impl
    extends GenericBusinessComponent
    implements PersonFinders
    /**
     * @param   nameType  a name type
     * @return  count of names by name type
     *
     * @BusinessMethod (customizationCallable = true)
     */
    public int findCountByNameType(Lookup nameType) {
        Query query = createQuery
            ("FROM PersonName name where name.nameType = :type");
        query.bindLookup("type", nameType);

        return (int) query.listSize();
    }
}
```

This example shows a "finder" component that is responsible for holding queries related to the "person" entity. These queries are not related to any *particular* person because, in that case, they would rightfully belong on the entity implementation class itself. Our (cooked up) example shows a single method that returns a count of PersonName instances by name type.

Let's look at various parts of the component:

- **@BusinessComponent** class annotation.
    - **customizationReplaceable** attribute specifies whether or not customers can replace this component at runtime. The default is **false**. If a component is "replaceable", its methods are assumed to be "customizationCallable".
- **GenericBusinessComponent** is extended which gives this class access to framework methods.
- **PersonFinders** is implemented. This is the name of the generated business interface. Any customized replacement of the business component would implement this interface as well.
- The business method **findCountByNameType**. For the method to be exported to the business interface (and therefore callable by other business objects), it must be public.
    - **@BusinessMethod** is an optional method-level annotation.
        - **customizationCallable** specifies that this method is part of the "supported" API. That is, our customers are entitled to call this method from their customizations and therefore, we must change this method with great reluctance in future release.

Component Replacement

Business Components provide a simple extension mechanism where base-package code can be made available to be replaced by customizations. For this to take place, two things must take place:

- A component is added as described above with the customizationReplaceable annotation attribute set to true.
- A replacement component is created that implements the business interface of the original component and also sets the **replacementComponent** attribute to **true**.

An example, replacement of the PersonFinders component is shown below. Component implementations are registered in the same order as the "application stack", that is "base" followed by "ccb" then followed by "cm". After the component is defined in one application, derived applications (higher on the stack) can replace the implementation.

```
package com.abcutilities.cis.customizations.person;

/**

 * @BusinessComponent

 *   (replacementComponent = true)

 */

public class CustomizedPersonFindersImpl

    extends GenericBusinessComponent

    implements PersonFinders {
```

```
    public Integer findCountByNameType(PerOrBusLookup nameType) {

        ... customized code ...

    }

    ...

}
```

Calling Components

Business Components are accessed via their business interfaces. Following is an example of how to call the above component from some other business object:

```

PersonFinders finders = PersonFinders.Factory.newInstance();

int count = finders.findCountByNameType(NameTypeLookup.constants.PRIMARY);

logger.info(count + " primary names found");
```

Maintaining Maintenance Classes, including collections

Maintaining Services

This defines services available in the system. These include user interface services as well as stand-alone XAI services. Use this transaction to introduce a new user interface or stand-alone XAI service.

Note:
For detailed information about service programs, please refer to user document *Framework Administration, XML Application Integration, Setting Up Your XAI Environment, Setting Up Your Registry, Service Program*.

Maintaining Foreign Key References

You need to setup foreign key references if you have characteristics whose valid values are defined in another table (e.g., if you use "foreign key reference" characteristic types).

Note:
For detailed information about foreign keys, see "Primary and Foreign Keys" in the *Oracle Utilities Application Framework Administration Guide*.

Maintaining Lookup Tables

Some special fields are defined as "lookups" in the system. These fields have a predefined set of values for which language-dependent descriptions are supplied to be displayed in the online system.

Note:
For detailed information about lookups, please refer to user document *Framework Administration, Database Tools, Defining Look Up Options*.

Maintaining Navigation Keys

Each location to which a user can navigate (e.g., transactions, tab pages, tab menus, online help links, etc.) is identified by a navigation key. A navigation key is a logical identifier for a URL.

Note:
For detailed information about navigation keys, please refer to user document *Framework Administration, User Interface Tools, Defining Navigation Keys*.

Maintaining Navigation Options

Every time a user navigates to a transaction, the system retrieves a navigation option to determine which transaction should open. Many navigation options are shipped with the base package and cannot be modified as these options support core functionality, but you may need to add additional navigation options to support your specific business processes.

Note:
For detailed information about navigation options, please refer to user document *Framework Administration, User Interface Tools, Defining Navigation Options*.

Maintaining User Interfaces

The configuration tools allow you to extend the front-end user interface. The main component of this is a UI Map, supported by Business Objects and Business Services.

Note:
For detailed information about user interfaces, please refer to user document *Framework Administration, Configuration Tools.*

Maintaining Menus

This metadata represents the root of a menu "tree". A menu contains a list of menu "lines", which, in turn, contains a list of menu "items". Lines can define navigation keys and/or associated actions, or further submenus.

Note:
For detailed information about menus, please refer to user document *Framework Administration, User Interface Tools, Defining Menu Options.*

Maintaining Application Security

Application security defines how a particular application service is used, namely:

- Which user groups can access the service
- What actions may be performed within the service

Note:
For detailed information on how to define application security, please refer to user document *Framework Administration, Defining Security & User Options.*

Maintaining UI Components (Translation)

You can use the override fields on some of the system data tables to modify and customize the labels, buttons, titles, tab names and messages on the standard user interface. This may be helpful to correct minor interface inconsistencies and inappropriate translations as well as to provide translations for any single fixes that you may have applied to your environment. (Single fixes release without translation, so you may need to translate any labels and descriptions for new UI components or messages.)

You can manually modify the descriptions or translations of the following items:

- Dialog titles
- Transaction titles and tab labels
- Field labels on tab pages
- Button labels
- Messages

Flushing Server and Client Caches

A great deal of information in the user interface changes infrequently, including field labels, menu items, and drop down lists. In order to avoid accessing the database every time this type of information is required by an end-user, the system maintains a cache of static information on the web server. Additionally, depending on how you set up the preferences on your Web browser, these items may also be cached in the browser.

After you make a change to a user interface item, such as a field label, you may need to flush the appropriate cache on the Web server as well as the client.

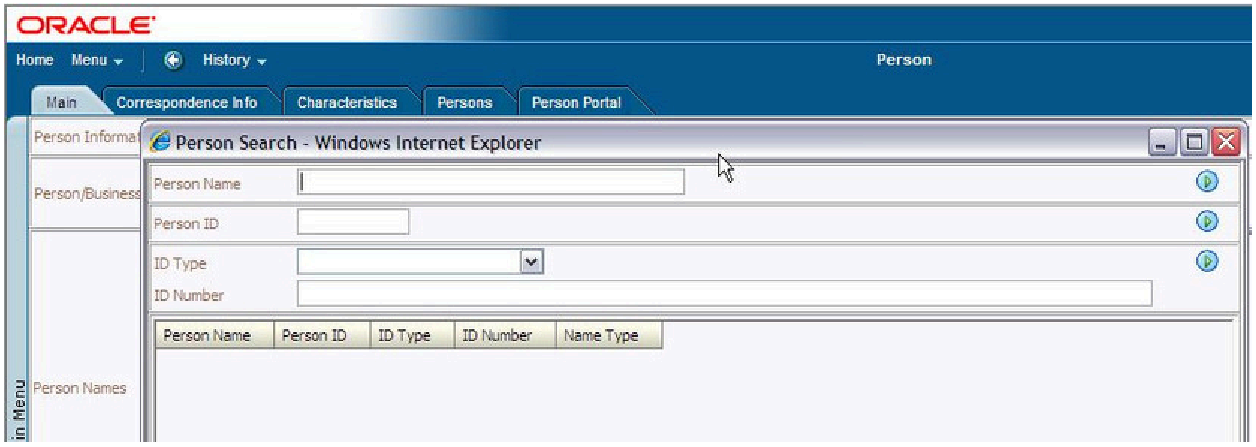
Note:
For information about flushing caches on the Web server, refer to the Caching Overview section in the Defining General Options chapter of the Oracle Utilities Application Framework Administration documentation.

User Language

You must log in as a user ID that has the same language as the items for which you want to modify the description. For example, if you want to modify a French message, you must log in with a user ID that is set to use French. The instructions in the following sections assume that you are logged in with a user ID that has the appropriate language set.

Modifying Dialog Titles

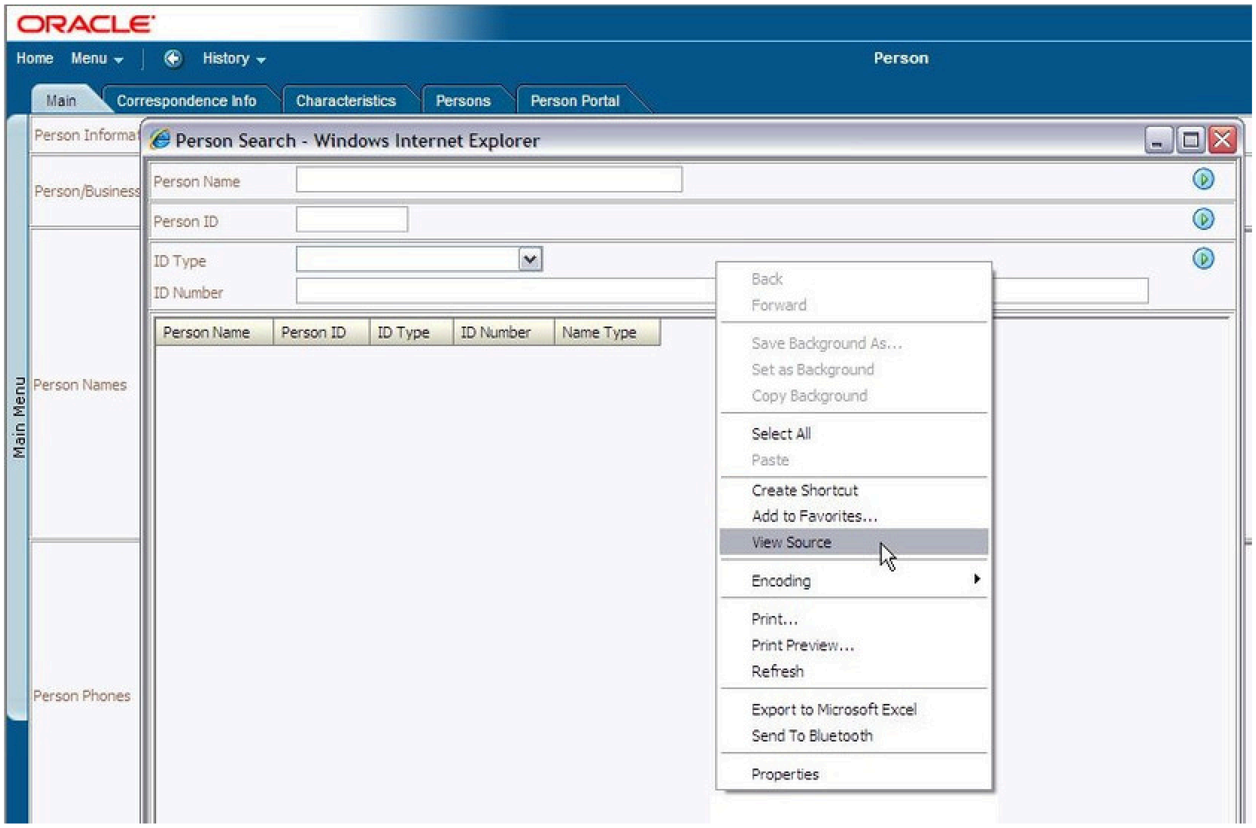
A dialog can be a search window or dialogs that provide additional functionality, such as the Start / Stop Confirmation Dialog or the Generate Bill dialog.



Dialog Title

To modify a dialog title:

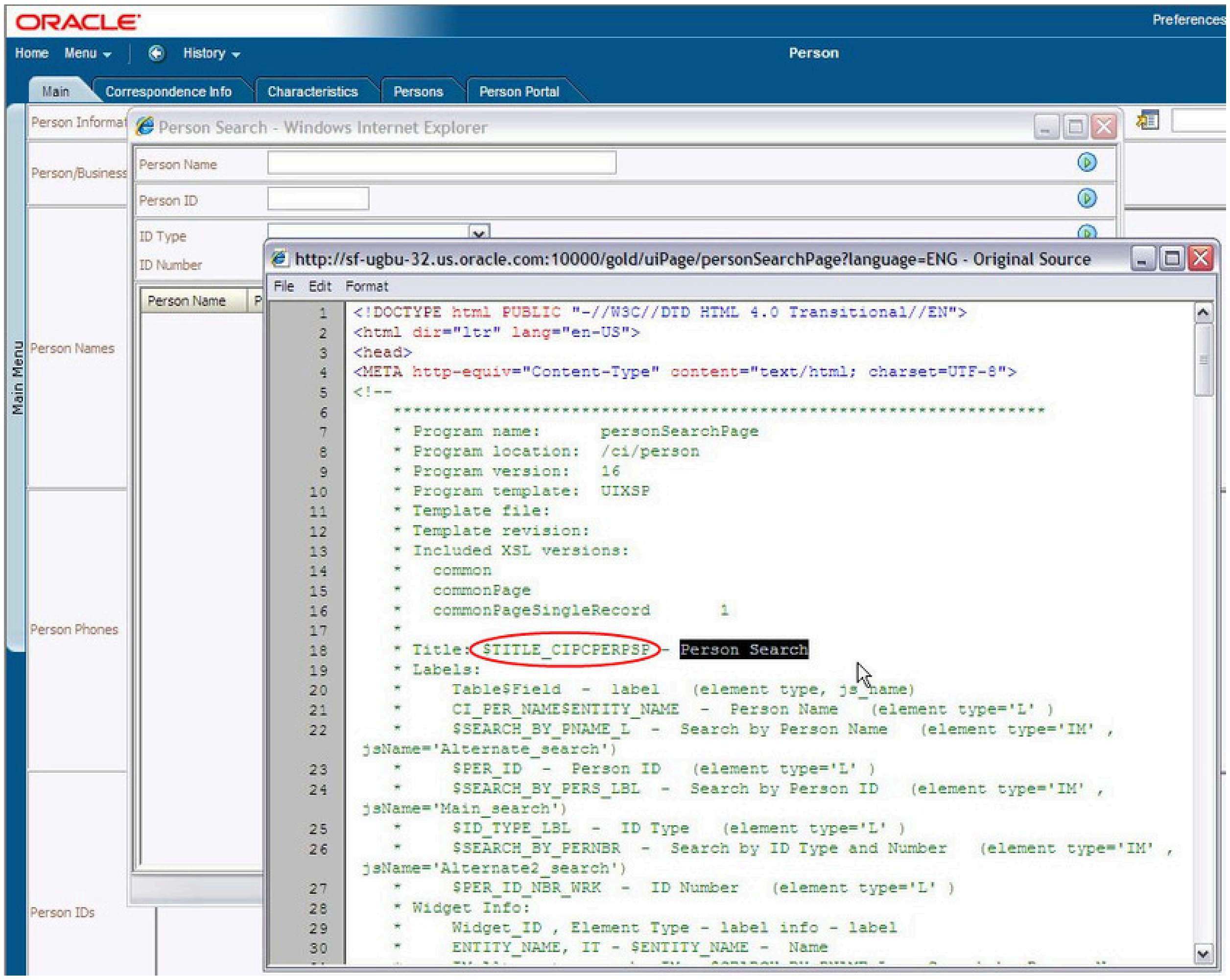
- Navigate to and open the dialog with the title that you want to change.
- Right-click near the top of the dialog and select View Source from the pop-up menu.



View Dialog Source

Note:
Many dialogs and windows have multiple source files; so if you can't locate the field you are looking for, try right clicking in a different area (closer to the label you want to modify). For example, if you right-click in the grid area of the Person Search illustrated above, you will open a different source file. If you already know the name of the field you want to modify, you can skip this step.

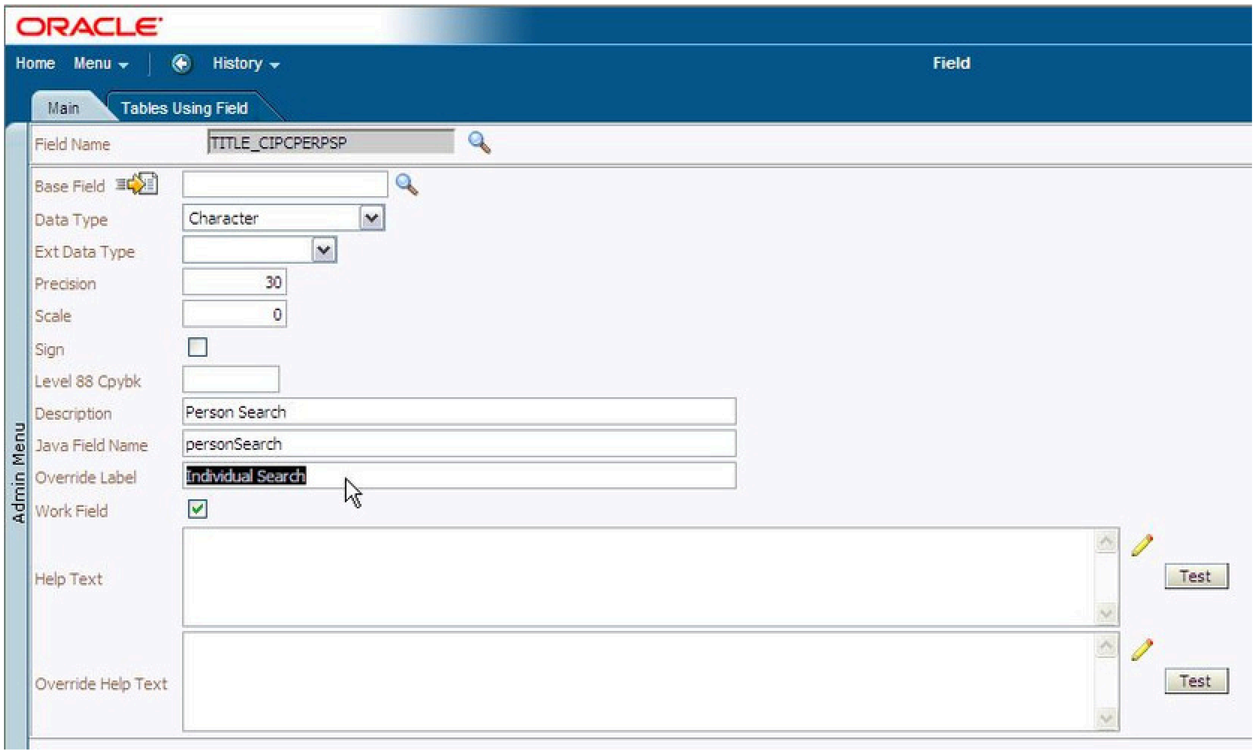
- In the displayed source file, locate the field name that has the value you want to modify. The field for the dialog title is clearly labelled and the current value of the field is displayed after the hyphen.



```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html dir="ltr" lang="en-US">
<head>
<META http-equiv="Content-Type" content="text/html; charset=UTF-8">
<!--
    *******************************************************************
    * Program name:       personSearchPage
    * Program location:   /ci/person
    * Program version:    16
    * Program template:   UIXSP
    * Template file:
    * Template revision:
    * Included XSL versions:
    *   common
    *   commonPage
    *   commonPageSingleRecord       1
    *
    * Title: $TITLE_CIPCPERPSP - Person Search
    * Labels:
    *     Table$Field  -  label   (element type, js_name)
    *     CI_PER_NAME$ENTITY_NAME  -  Person Name   (element type='L' )
    *     $SEARCH_BY_PNAME_L  -  Search by Person Name   (element type='IM' ,
 jsName='Alternate_search')
    *     $PER_ID  -  Person ID   (element type='L' )
    *     $SEARCH_BY_PERS_LBL  -  Search by Person ID   (element type='IM' ,
 jsName='Main_search')
    *     $ID_TYPE_LBL  -  ID Type   (element type='L' )
    *     $SEARCH_BY_PERNBR  -  Search by ID Type and Number   (element type='IM' ,
 jsName='Alternate2_search')
    *     $PER_ID_NBR_WRK  -  ID Number   (element type='L' )
    * Widget Info:
    *     Widget_ID , Element Type - label info - label
    *     ENTITY_NAME, IT - $ENTITY_NAME -  Name
```

Title Field Name

- To modify the field override via the application, navigate to **Admin Menu - Database - Field** in the Oracle Utilities Application Framework application.
- When the field search dialog appears, enter the name of the field as it appears in the source.
- Enter an Override Label with a title description to suit your needs and save your changes.



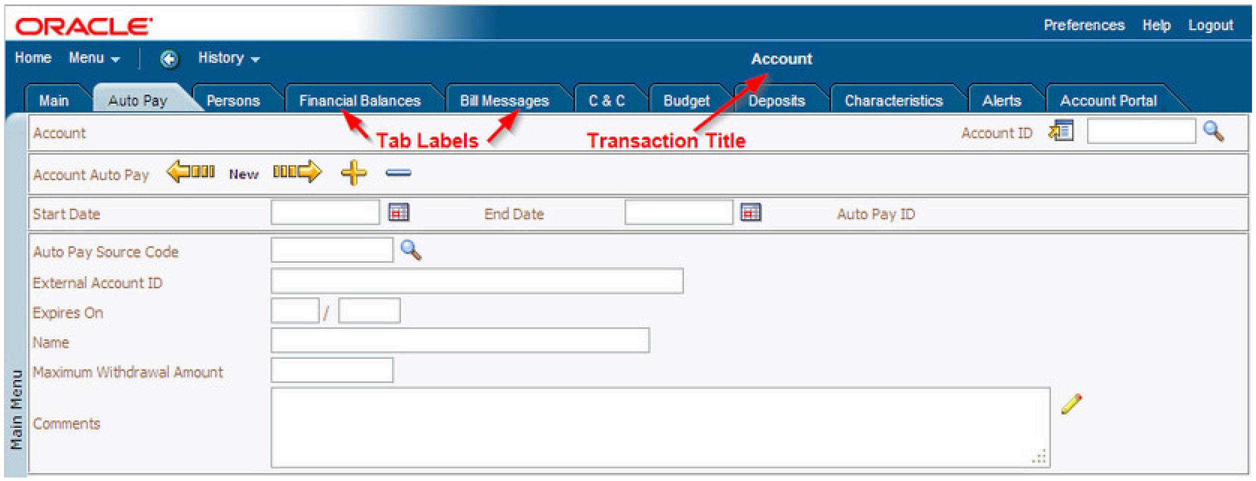
Field Name: TITLE_CIPCPERPSP
Base Field:
Data Type: Character
Ext Data Type:
Precision: 30
Scale: 0
Sign: ☐
Level 88 Cpybk:
Description: Person Search
Java Field Name: personSearch
Override Label: Individual Search
Work Field: ✓
Help Text:
Override Help Text:

Database - Field

• Flush the server and browser caches and verify that the new dialog title appears correctly.

Modifying Transaction Titles and Tab Labels

You can modify the transaction title and or the tab labels that appear on a transaction.

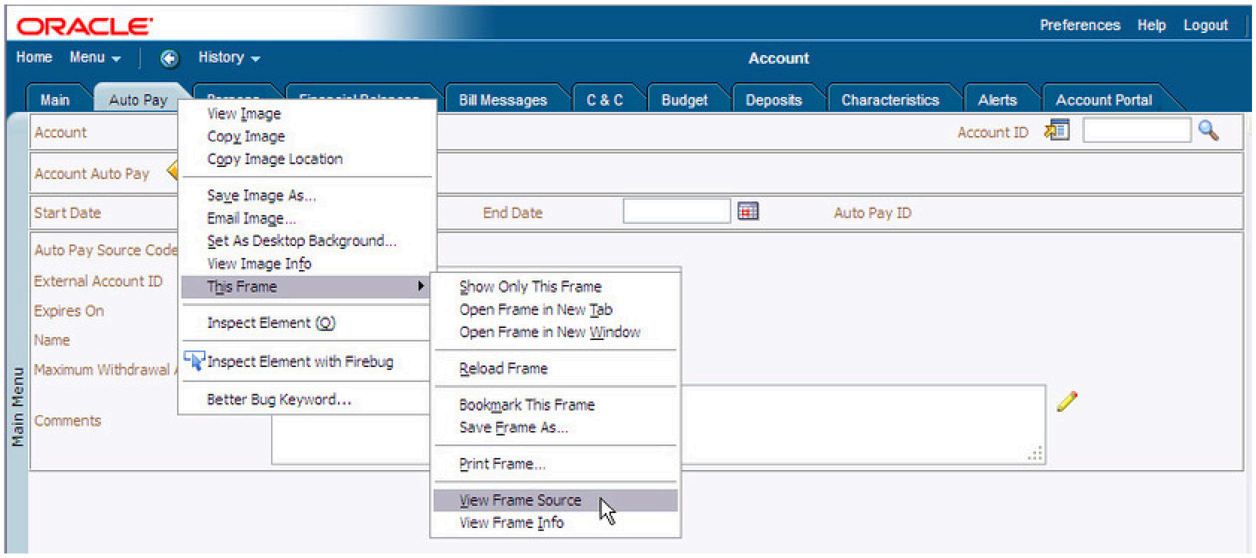


Account
Account ID
Account Auto Pay
Start Date
End Date
Auto Pay ID
Auto Pay Source Code
External Account ID
Expires On /
Name
Maximum Withdrawal Amount
Comments

Transaction Title and Tab Labels

To modify the transaction title and/or tab labels:

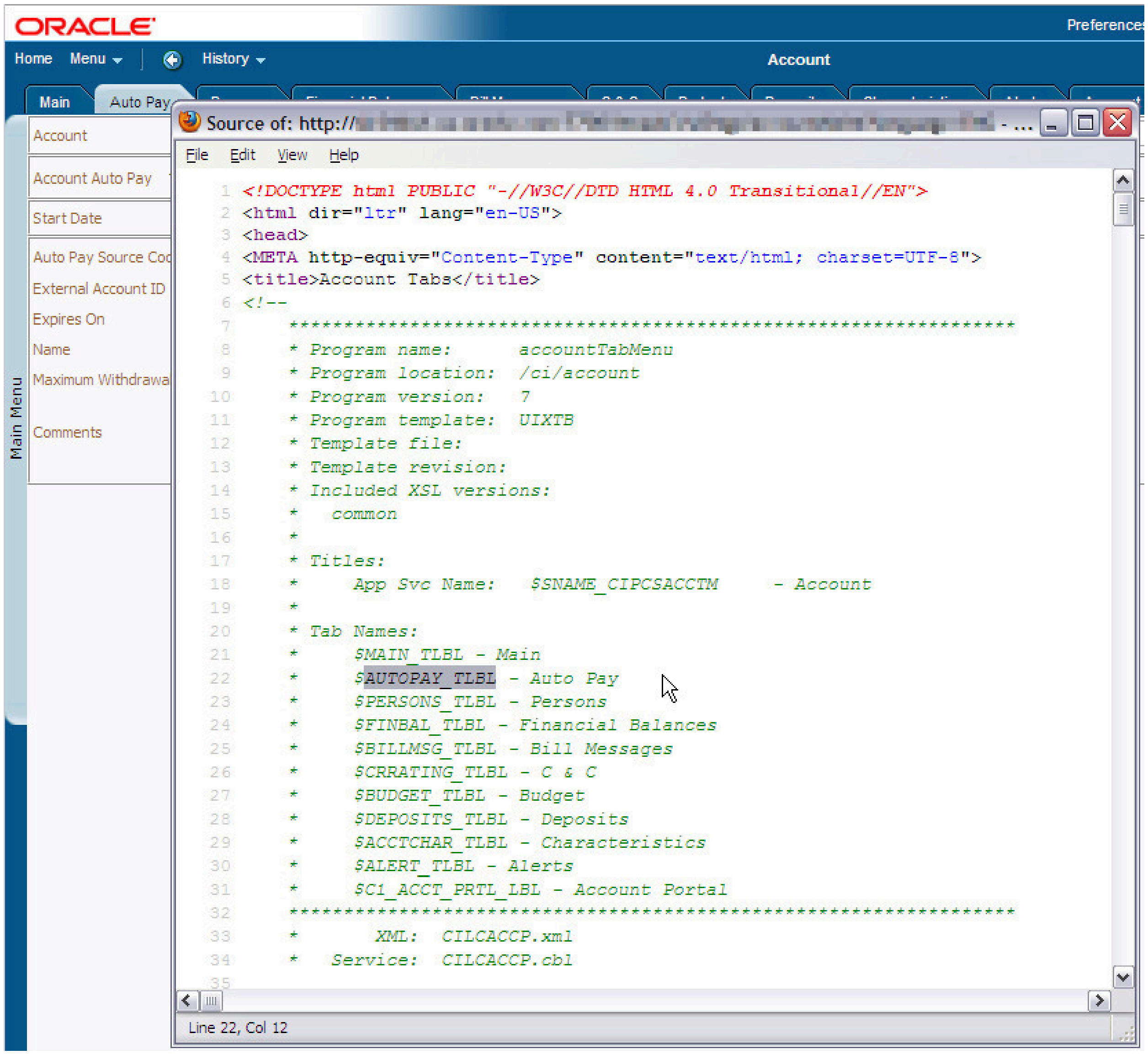
- Navigate to the transaction that has the title and/or tab name you want to modify.
- Right-click in the empty area to the right or left of the tab bar and select View Source from the drop-down menu.



View Transaction Title/Tab Source

Note:
Many dialogs and windows have multiple source files; so if you can't locate the field you are looking for, try right-clicking in a different area (closer to the label you want to modify). To view the source for the transaction title and tab bar, right-click directly to the right or left of the tab bar. If you already know the name of the field you want to modify, you can skip this step.

- In the displayed source file, locate the field name that has the value you want to modify. The fields for the transaction titles and tab labels are clearly labelled and the current values of the fields are displayed after the hyphens.



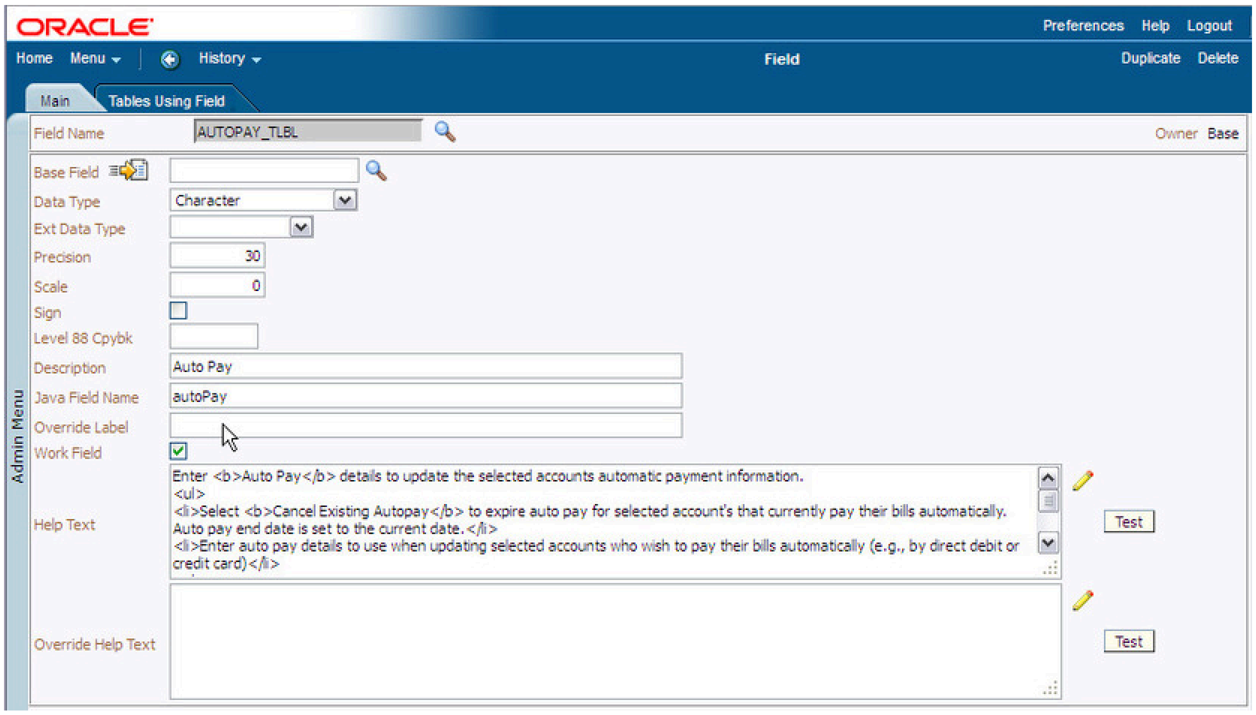
```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html dir="ltr" lang="en-US">
<head>
<META http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Account Tabs</title>
<!--
    ******************************************************************
    * Program name:      accountTabMenu
    * Program location:  /ci/account
    * Program version:   7
    * Program template:  UIXTB
    * Template file:
    * Template revision:
    * Included XSL versions:
    *   common
    *
    * Titles:
    *     App Svc Name:   $SNAME_CIPCSACCTM     - Account
    *
    * Tab Names:
    *     $MAIN_TLBL - Main
    *     $AUTOPAY_TLBL - Auto Pay
    *     $PERSONS_TLBL - Persons
    *     $FINBAL_TLBL - Financial Balances
    *     $BILLMSG_TLBL - Bill Messages
    *     $CRRATING_TLBL - C & C
    *     $BUDGET_TLBL - Budget
    *     $DEPOSITS_TLBL - Deposits
    *     $ACCTCHAR_TLBL - Characteristics
    *     $ALERT_TLBL - Alerts
    *     $C1_ACCT_PRTL_LBL - Account Portal
    ******************************************************************
    *       XML:  CILCACCP.xml
    *   Service:  CILCACCP.cbl
```

Transaction Title and Tab Field Names

Note:

Subsystem Name. If you modify the subsystem field description, your changes will appear on every transaction that is part of the subsystem.

- To modify the field override via the application, navigate to **Admin Menu - Database - Field** in the Oracle Utilities Application Framework application.
- When the field search dialog appears, enter the name of the field as it appears in the source.
- Enter an Override Label with a title or tab description to suit your needs and save your changes.



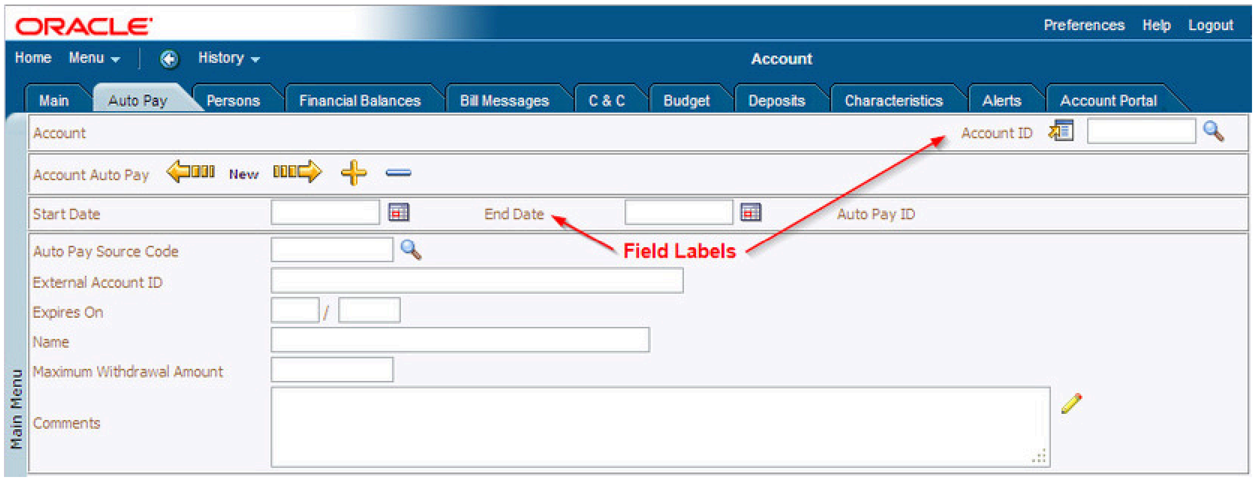
Field Name: AUTOPAY_TLBL
Owner: Base
Base Field:
Data Type: Character
Ext Data Type:
Precision: 30
Scale: 0
Sign: ☐
Level 88 Cpybk:
Description: Auto Pay
Java Field Name: autoPay
Override Label:
Work Field: ✓
Help Text:
```
Enter <b>Auto Pay</b> details to update the selected accounts automatic payment information.
<ul>
<li>Select <b>Cancel Existing Autopay</b> to expire auto pay for selected account's that currently pay their bills automatically.
Auto pay end date is set to the current date.</li>
<li>Enter auto pay details to use when updating selected accounts who wish to pay their bills automatically (e.g., by direct debit or
credit card)</li>
```
Override Help Text:

Database - Field

- Flush the server and browser caches and verify that the new field label appears correctly.

Modifying Field Labels on Pages

You can modify field labels that appear on transactions.



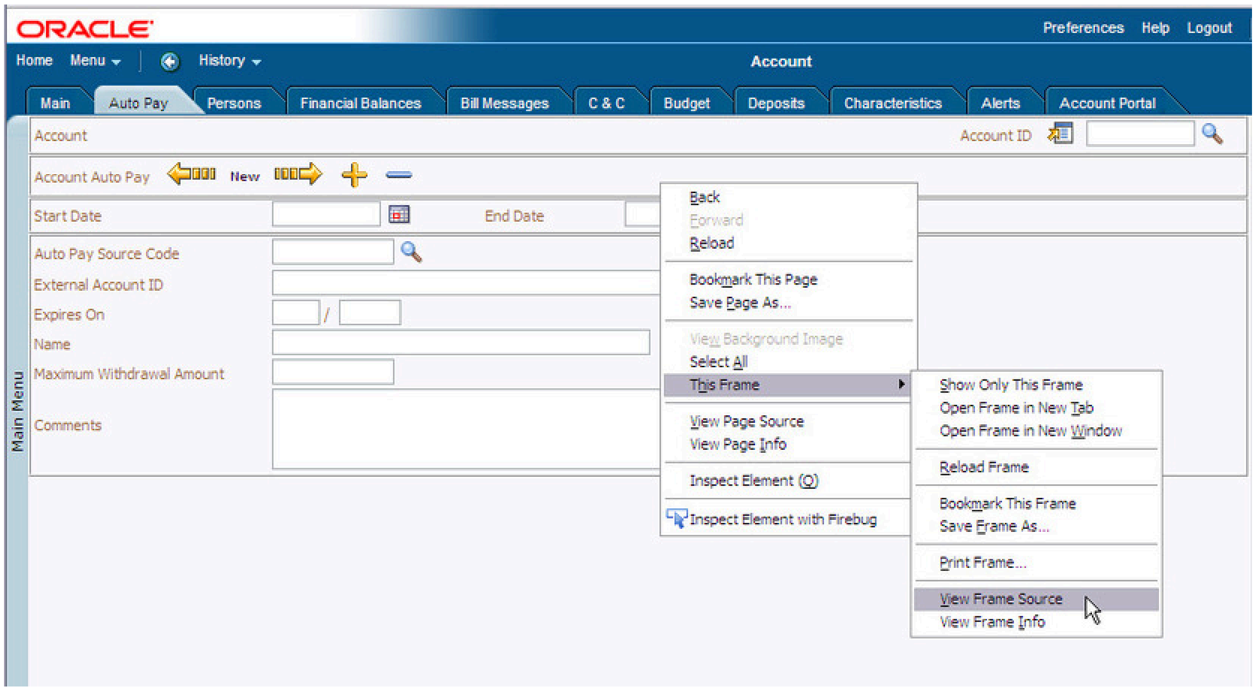
Field Labels

Warning:

Field labels may be reused! A field label may be reused on multiple transactions and tabs. If you override the field's label, your changes affect all pages and transactions on which that field label appears.

To modify the field labels that appear on transactions:

- Navigate to the transaction that has the field name you want to modify.
- Right-click in an empty area near the label and select View Source from the drop-down menu.

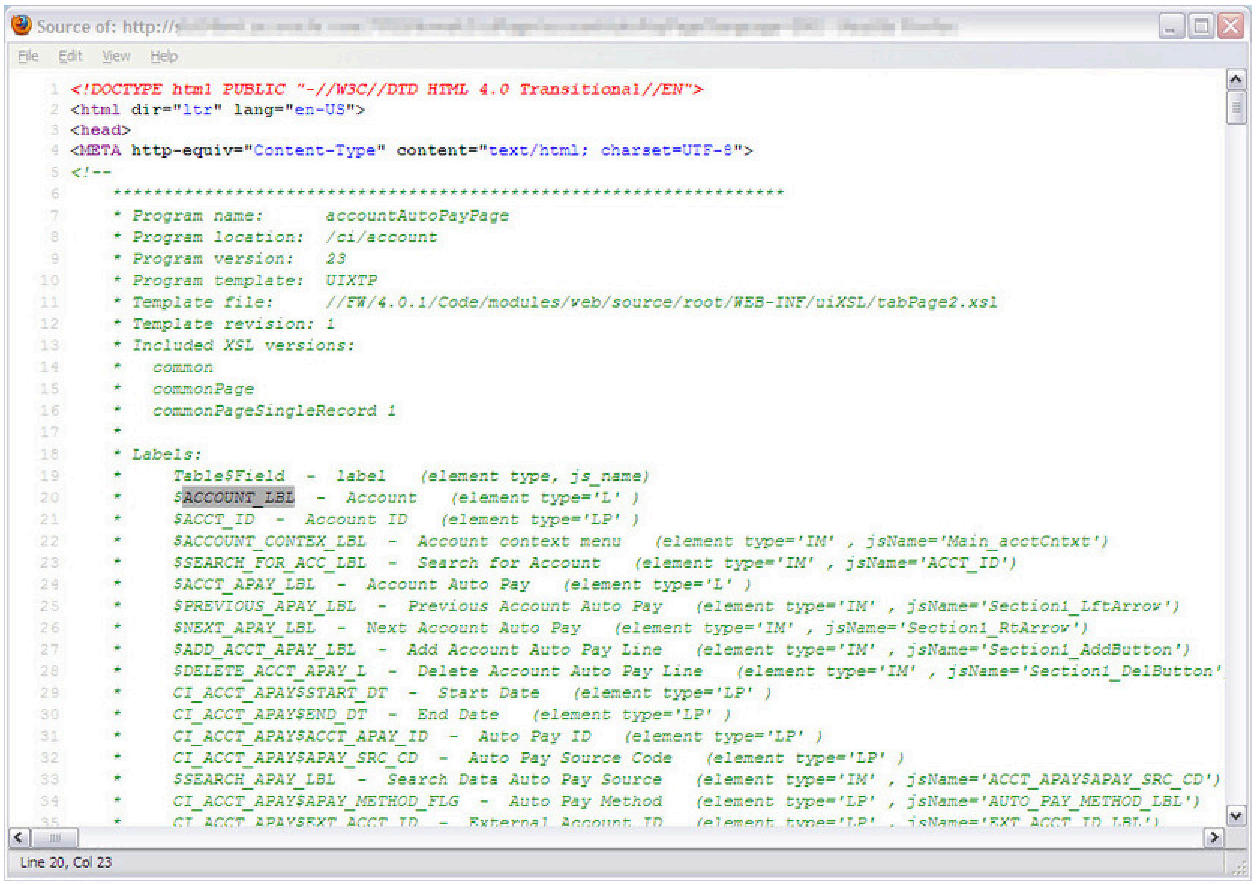


View Page Source

Note:

Many dialogs and windows have multiple source files; so if you can't locate the field you are looking for, try right-clicking in a different area (closer to the label you want to modify). If you already know the name of the field you want to modify, you can skip this step.

- In the displayed source file, locate the field name that has the value you want to modify. The fields for the labels are clearly identified and the current values of the fields are displayed after the hyphens.



Source of: http://

File Edit View Help

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html dir="ltr" lang="en-US">
<head>
<META http-equiv="Content-Type" content="text/html; charset=UTF-8">
<!--
      ******************************************************************
      * Program name:      accountAutoPayPage
      * Program location:  /ci/account
      * Program version:   23
      * Program template:  UIXTP
      * Template file:     //FW/4.0.1/Code/modules/veb/source/root/WEB-INF/uiXSL/tabPage2.xsl
      * Template revision: 1
      * Included XSL versions:
      *   common
      *   commonPage
      *   commonPageSingleRecord 1
      *
      * Labels:
      *      Table$Field  -  label   (element type, js_name)
      *      $ACCOUNT_LBL  -  Account   (element type='L' )
      *      $ACCT_ID  -  Account ID   (element type='LP' )
      *      $ACCOUNT_CONTEX_LBL  -  Account context menu   (element type='IM' , jsName='Main_acctCntxt')
      *      $SEARCH_FOR_ACC_LBL  -  Search for Account   (element type='IM' , jsName='ACCT_ID')
      *      $ACCT_APAY_LBL  -  Account Auto Pay   (element type='L' )
      *      $PREVIOUS_APAY_LBL  -  Previous Account Auto Pay   (element type='IM' , jsName='Section1_LftArrow')
      *      $NEXT_APAY_LBL  -  Next Account Auto Pay   (element type='IM' , jsName='Section1_RtArrow')
      *      $ADD_ACCT_APAY_LBL  -  Add Account Auto Pay Line   (element type='IM' , jsName='Section1_AddButton')
      *      $DELETE_ACCT_APAY_L  -  Delete Account Auto Pay Line   (element type='IM' , jsName='Section1_DelButton'
      *      CI_ACCT_APAY$START_DT  -  Start Date   (element type='LP' )
      *      CI_ACCT_APAY$END_DT  -  End Date   (element type='LP' )
      *      CI_ACCT_APAY$ACCT_APAY_ID  -  Auto Pay ID   (element type='LP' )
      *      CI_ACCT_APAY$APAY_SRC_CD  -  Auto Pay Source Code   (element type='LP' )
      *      $SEARCH_APAY_LBL  -  Search Data Auto Pay Source   (element type='IM' , jsName='ACCT_APAY$APAY_SRC_CD')
      *      CI_ACCT_APAY$APAY_METHOD_FLG  -  Auto Pay Method   (element type='LP' , jsName='AUTO_PAY_METHOD_LBL')
      *      CI_ACCT_APAY$EXT_ACCT_ID  -  External Account ID   (element type='LP' , jsName='EXT_ACCT_ID_LBL')
```

Line 20, Col 23

Field Label Names and Values

Note:

Table-specific Fields. Note that some labels may be specific to the table on which they appear, while other labels are generic throughout the application. If a field label is specific to a table, the table name appears before the $ in the field list.

- If the label is table-specific, navigate to **Admin Menu - Database - Table** in the Oracle Utilities Application Framework application and search for the name of the table.

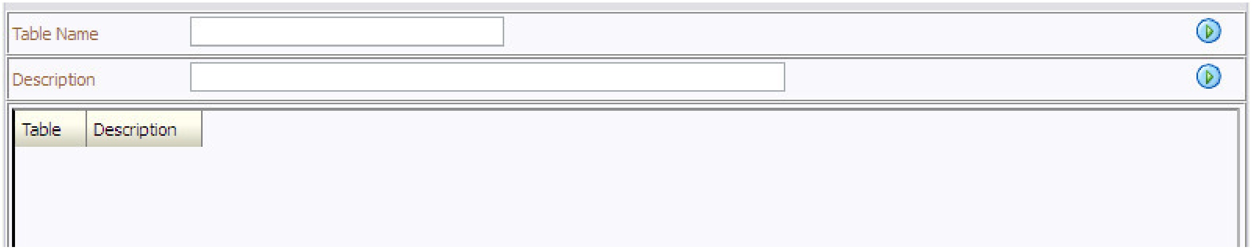


Table Name	
Description	

Table	Description

Search for Table

- Navigate to the Table Field tab and scroll to the field whose label you wish to modify.

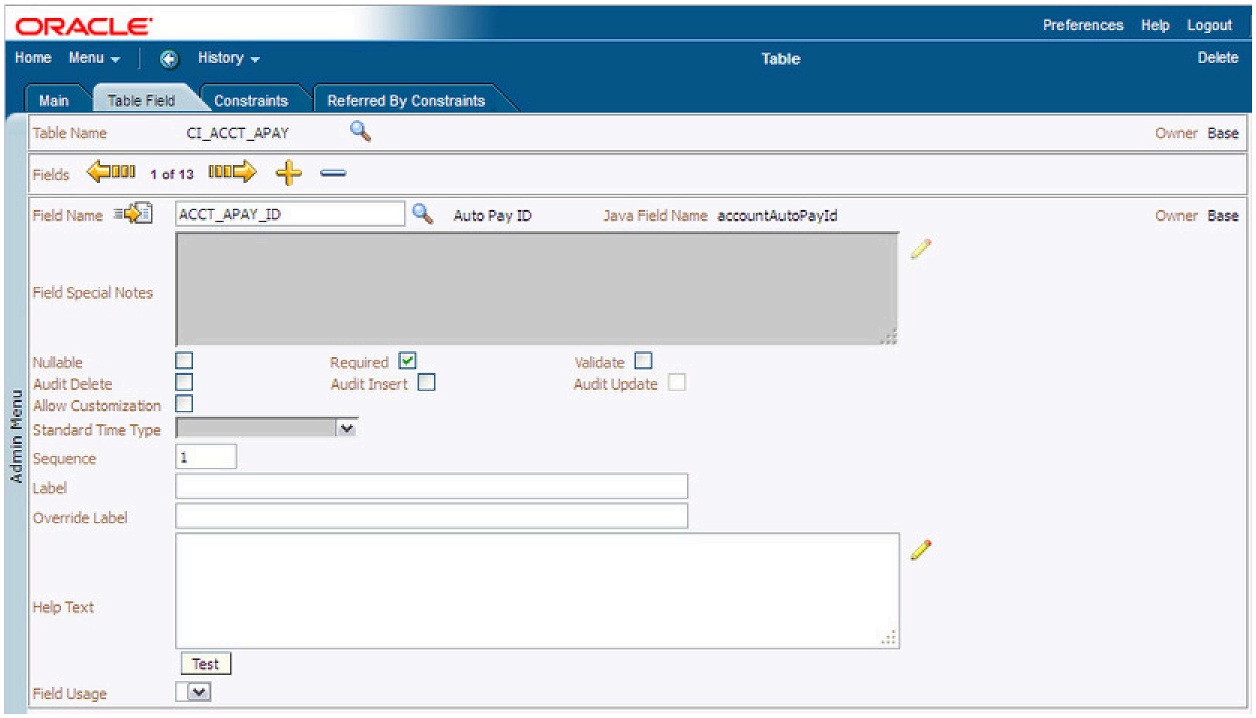


Table Name CI_ACCT_APAY Owner Base

Fields 1 of 13

Field Name ACCT_APAY_ID Auto Pay ID Java Field Name accountAutoPayId Owner Base

Field Special Notes

Nullable ☐ Required ✓ Validate ☐

Audit Delete ☐ Audit Insert ☐ Audit Update ☐

Allow Customization ☐

Standard Time Type

Sequence 1

Label

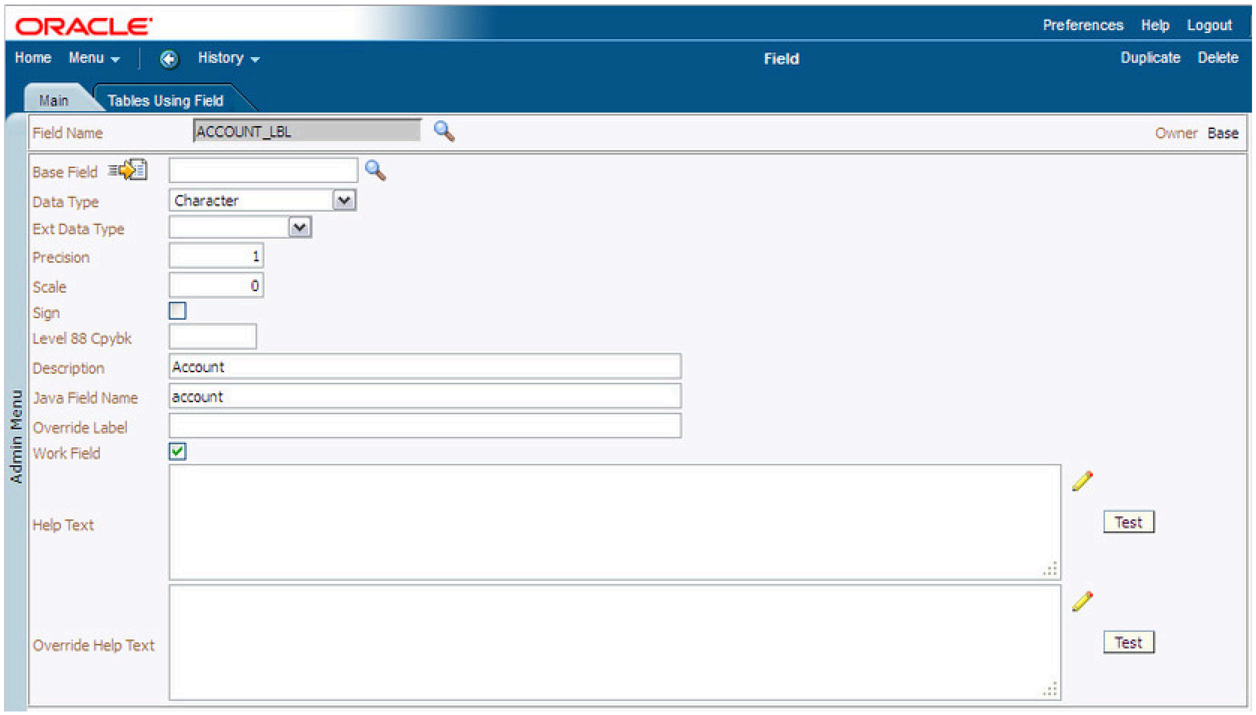
Override Label

Help Text

Field Usage

Table Field

- Enter an Override Label to suit your needs and save your changes.
- If the label is not table-specific, navigate to **Admin Menu - Database - Field** and search for the field name.
- When the field appears, enter an Override Label to suit your needs and save your changes.



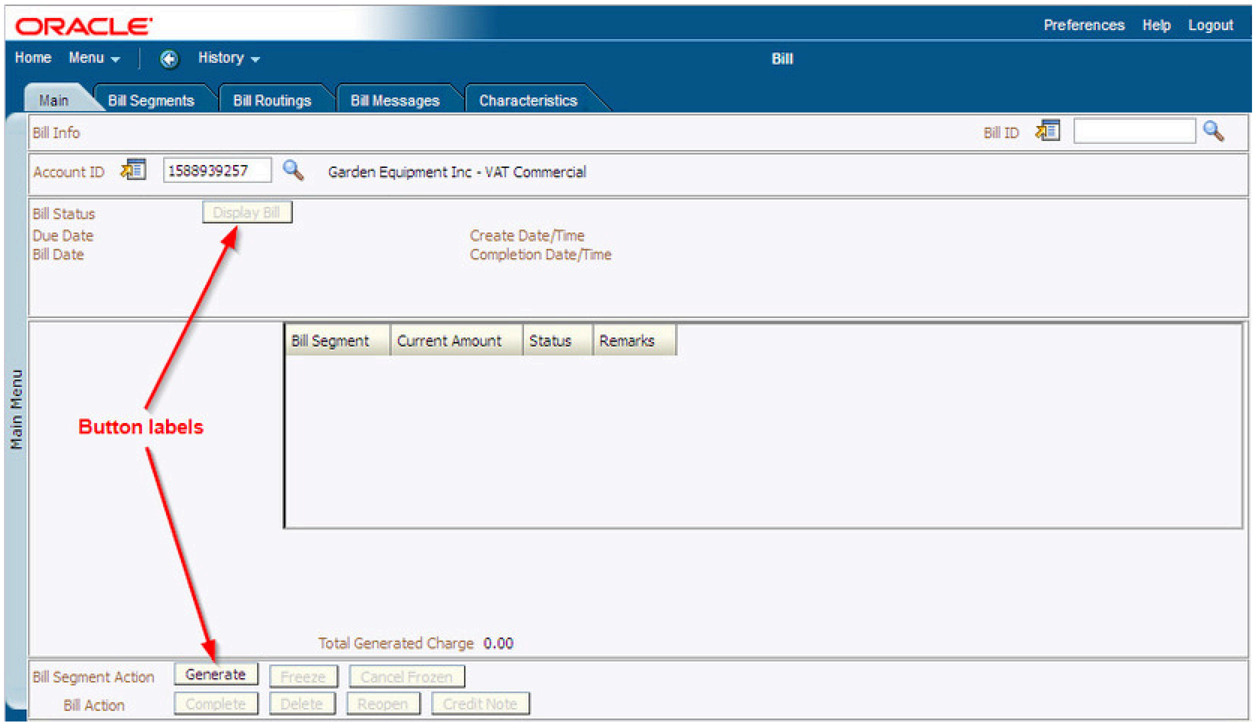
Field Name	ACCOUNT_LBL
Owner	Base
Base Field	
Data Type	Character
Ext Data Type	
Precision	1
Scale	0
Sign	☐
Level 88 Cpybk	
Description	Account
Java Field Name	account
Override Label	
Work Field	✓
Help Text	
Override Help Text	

Database - Field

- Flush the server and browser caches and verify that the new field label appears correctly.

Modifying Button Labels

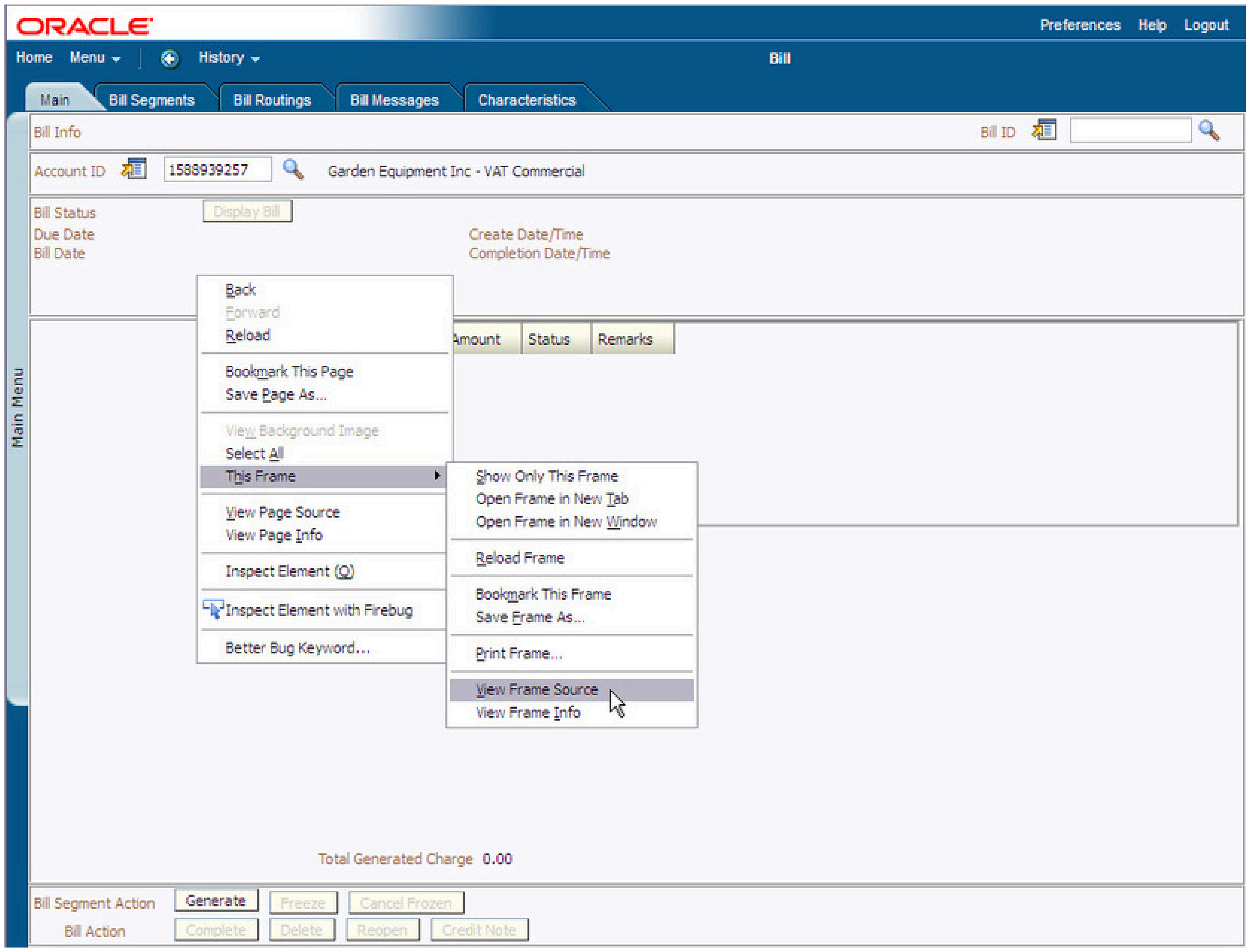
Button labels are just like field labels; they are stored in the field table. You can modify button labels just like you can field labels.


ORACLE
Preferences Help Logout
Home Menu History
Bill
Main Bill Segments Bill Routings Bill Messages Characteristics
Main Menu
Bill Info
Bill ID
Account ID 1588939257 Garden Equipment Inc - VAT Commercial
Bill Status
Display Bill
Due Date
Bill Date
Create Date/Time
Completion Date/Time
Bill Segment Current Amount Status Remarks
Button labels
Total Generated Charge 0.00
Bill Segment Action Generate Freeze Cancel Frozen
Bill Action Complete Delete Reopen Credit Note

Button Labels

To modify button labels:

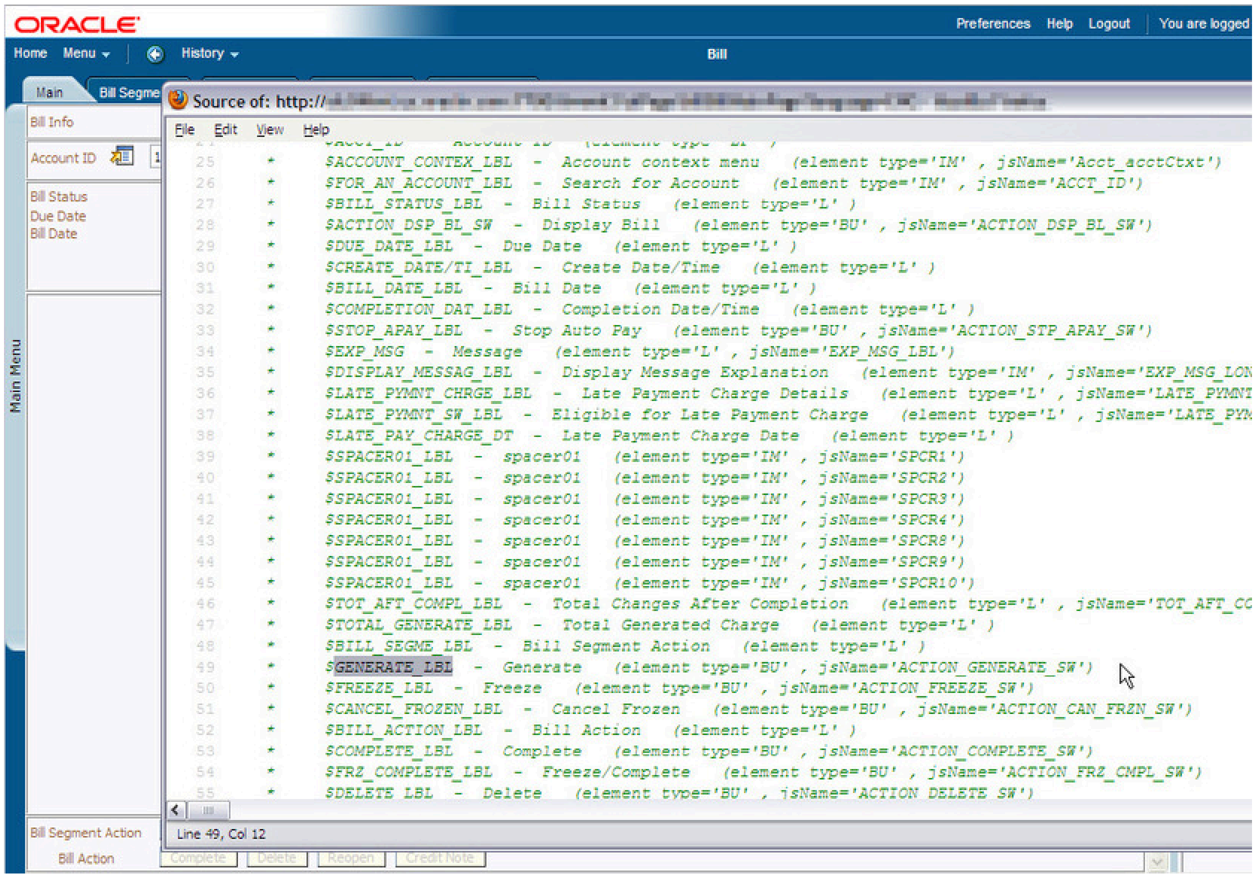
- Navigate to the transaction that has the button label you want to modify.
- Right-click in an empty area near the label and select View Source from the drop-down menu.



View Page Source

Note:
Many dialogs and windows have multiple source files; so if you can't locate the field you are looking for, try right clicking in a different area (closer to the label you want to modify). If you already know the name of the field you want to modify, you can skip this step.

- In the displayed source file, locate the field name that has the value you want to modify. The fields for the labels are clearly identified and the current values of the fields are displayed after the hyphens.



Source of: http://[redacted]

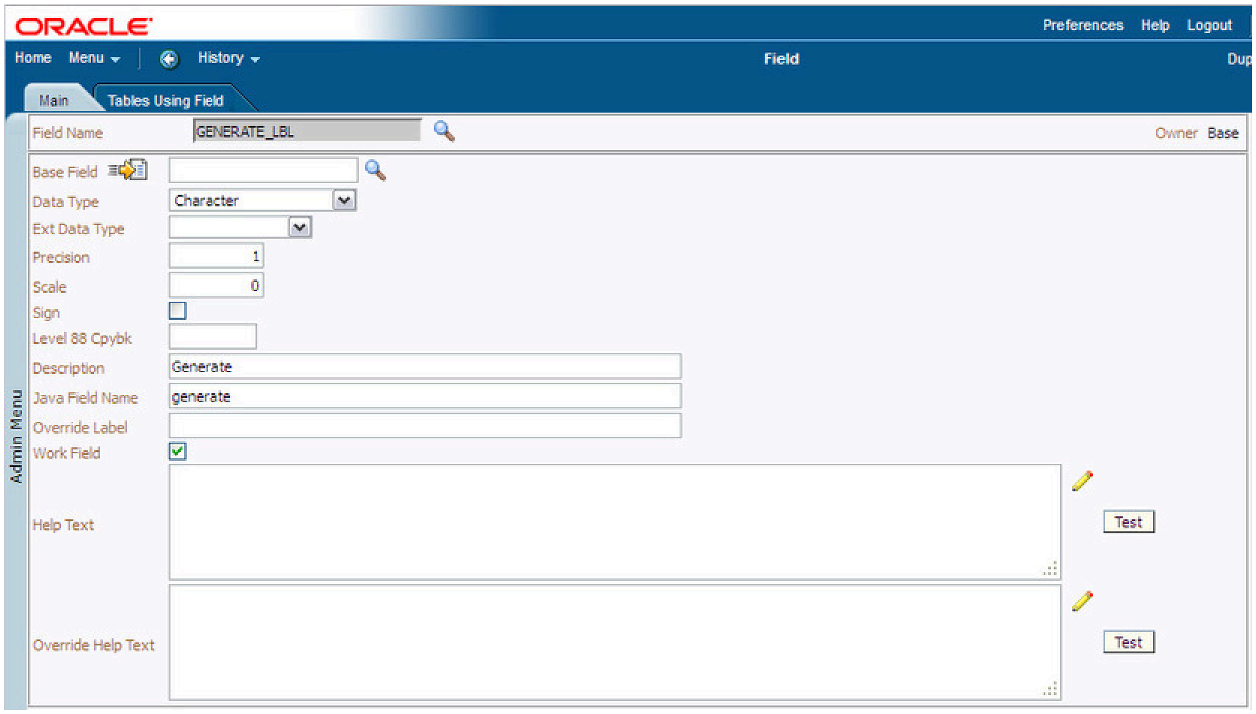
File Edit View Help

```
*    $ACCOUNT_CONTEX_LBL  -  Account context menu   (element type='IM' , jsName='Acct_acctCtxt')
*    $FOR_AN_ACCOUNT_LBL  -  Search for Account   (element type='IM' , jsName='ACCT_ID')
*    $BILL_STATUS_LBL  -  Bill Status   (element type='L' )
*    $ACTION_DSP_BL_SW  -  Display Bill   (element type='BU' , jsName='ACTION_DSP_BL_SW')
*    $DUE_DATE_LBL  -  Due Date   (element type='L' )
*    $CREATE_DATE/TI_LBL  -  Create Date/Time   (element type='L' )
*    $BILL_DATE_LBL  -  Bill Date   (element type='L' )
*    $COMPLETION_DAT_LBL  -  Completion Date/Time   (element type='L' )
*    $STOP_APAY_LBL  -  Stop Auto Pay   (element type='BU' , jsName='ACTION_STP_APAY_SW')
*    $EXP_MSG  -  Message   (element type='L' , jsName='EXP_MSG_LBL')
*    $DISPLAY_MESSAG_LBL  -  Display Message Explanation   (element type='IM' , jsName='EXP_MSG_LON
*    $LATE_PYMNT_CHRGE_LBL  -  Late Payment Charge Details   (element type='L' , jsName='LATE_PYMNT
*    $LATE_PYMNT_SW_LBL  -  Eligible for Late Payment Charge   (element type='L' , jsName='LATE_PYM
*    $LATE_PAY_CHARGE_DT  -  Late Payment Charge Date   (element type='L' )
*    $SPACER01_LBL  -  spacer01   (element type='IM' , jsName='SPCR1')
*    $SPACER01_LBL  -  spacer01   (element type='IM' , jsName='SPCR2')
*    $SPACER01_LBL  -  spacer01   (element type='IM' , jsName='SPCR3')
*    $SPACER01_LBL  -  spacer01   (element type='IM' , jsName='SPCR4')
*    $SPACER01_LBL  -  spacer01   (element type='IM' , jsName='SPCR8')
*    $SPACER01_LBL  -  spacer01   (element type='IM' , jsName='SPCR9')
*    $SPACER01_LBL  -  spacer01   (element type='IM' , jsName='SPCR10')
*    $TOT_AFT_COMPL_LBL  -  Total Changes After Completion   (element type='L' , jsName='TOT_AFT_CC
*    $TOTAL_GENERATE_LBL  -  Total Generated Charge   (element type='L' )
*    $BILL_SEGME_LBL  -  Bill Segment Action   (element type='L' )
*    $GENERATE_LBL  -  Generate   (element type='BU' , jsName='ACTION_GENERATE_SW')
*    $FREEZE_LBL  -  Freeze   (element type='BU' , jsName='ACTION_FREEZE_SW')
*    $CANCEL_FROZEN_LBL  -  Cancel Frozen   (element type='BU' , jsName='ACTION_CAN_FRZN_SW')
*    $BILL_ACTION_LBL  -  Bill Action   (element type='L' )
*    $COMPLETE_LBL  -  Complete   (element type='BU' , jsName='ACTION_COMPLETE_SW')
*    $FRZ_COMPLETE_LBL  -  Freeze/Complete   (element type='BU' , jsName='ACTION_FRZ_CMPL_SW')
*    $DELETE_LBL  -  Delete   (element type='BU' , jsName='ACTION_DELETE_SW')
```

Line 49, Col 12

Field Label Names and Values

- Navigate to **Admin Menu - Database - Field** in the Oracle Utilities Application Framework application and search for the field name.
- When the field appears, enter an Override Label to suit your needs and save your changes.



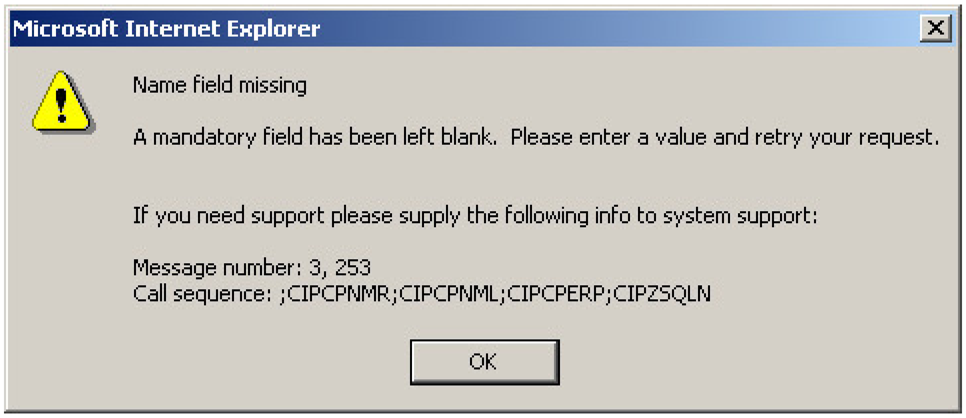
Field Name: GENERATE_LBL
Owner: Base
Base Field:
Data Type: Character
Ext Data Type:
Precision: 1
Scale: 0
Sign: ☐
Level 88 Cpybk:
Description: Generate
Java Field Name: generate
Override Label:
Work Field: ✓
Help Text:
Override Help Text:

Database - Field

• Flush the server and browser caches and verify that the new field label appears correctly.

Modifying Messages

You can modify the message text and description for messages, such as error, warning and validation messages. The following example shows a validation message:



Name field missing

A mandatory field has been left blank. Please enter a value and retry your request.

If you need support please supply the following info to system support:

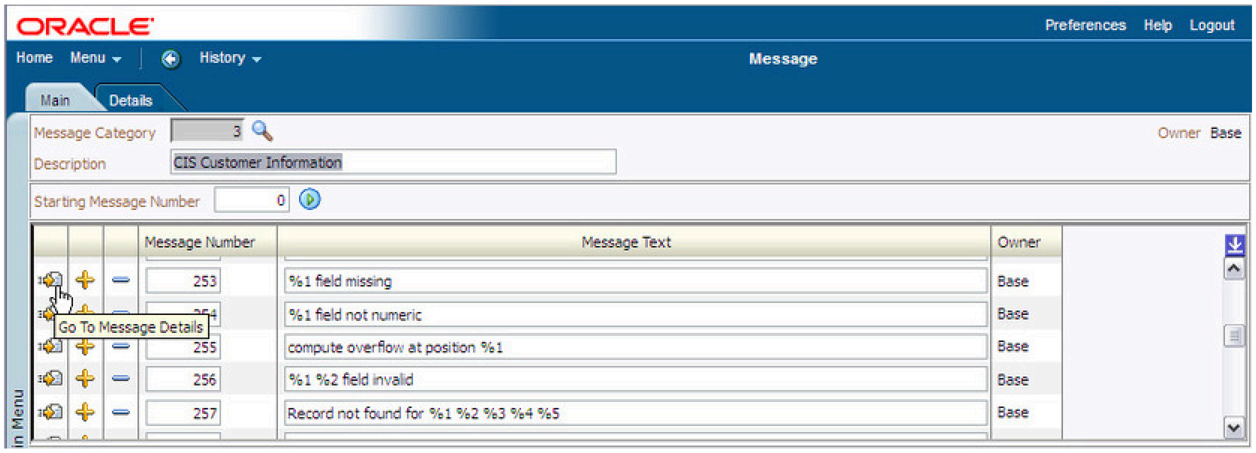
Message number: 3, 253
Call sequence: ;CIPCPNMR;CIPCPNML;CIPCPERP;CIPZSQLN

Message

To edit messages, you need to know the message category and number. The category is the part of the message number that appears before the comma. In the example message above, the category is 3. The number is the part of the message number that appears after the comma. In the example message above, the message number is 253.

To edit the message text or description:

- Navigate to **Admin Menu - System - Message**.
- Specify the message category in the search dialog.
- Specify the starting message number and click the search icon.

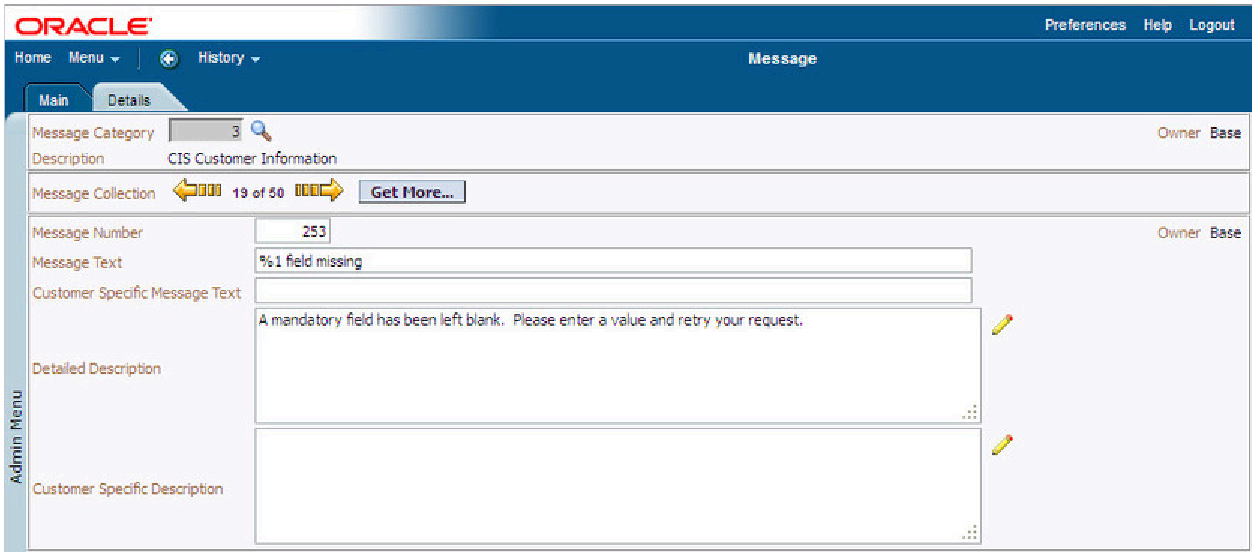


Message Category	3
Description	CIS Customer Information
Owner	Base
Starting Message Number	0

Message Number	Message Text	Owner
253	%1 field missing	Base
254	%1 field not numeric	Base
255	compute overflow at position %1	Base
256	%1 %2 field invalid	Base
257	Record not found for %1 %2 %3 %4 %5	Base

System - Message

- Click the go to button for the message you want to edit. You are transferred to the Details tab for that message.



Message Category 3 Owner Base
Description CIS Customer Information
Message Collection 19 of 50
Message Number 253 Owner Base
Message Text %1 field missing
Customer Specific Message Text
Detailed Description A mandatory field has been left blank. Please enter a value and retry your request.
Customer Specific Description

Message Details

- Enter the customer specific message text and description as appropriate for your needs.

Note:

Message Variables. Messages may have one or more variables. Variables are indicated by a percent sign (%) followed by a number. A value is substituted for the variable before the message is displayed. Do not modify the message variables and make sure that your custom message contains the same number of variables as the original.

- Save your changes.

If possible, you can attempt to verify that the message was changed correctly. However, it is not always easy to determine and duplicate the situations where a specific message may appear.

Note:

For more information about system messages, please refer to user document *Framework Administration, User Interface Tools, Defining System Messages*.

Plugging in Algorithms

The following will illustrate the steps to create a new plug-in algorithm. This example will create a new Adhoc characteristic validation algorithm that is very similar to a delivered plug-in.

Creating Algorithm Spot Implementation Class

Review Algorithm Spot Definition

The algorithm spot definition identifies the purpose of the algorithm spot and the required methods per implementation. It may also help to look at existing implementations of the relevant algorithm spot.

The relevant algorithm spot in this example is AdhocCharacteristicValueValidationAlgorithmSpot in com.splwg.base.domain.common.characteristicType.

Create Algorithm Component Implementation

Copy the existing numeric validation plug-in "AdhocDateValidationAlgComp_Impl and name it as " AdhocDateAgeValidationiiiAlgComp_Impl" where iii is your initials.

Modify the annotation to replace the last Date Format soft parameter with two decimal parameters (ageFrom and ageTo).

In addition, modify the validateDateInRange method to check that the age (given date less the system's current date / 365.25) will be greater than the soft parameter ageFrom (if non-zero), and will be less than the ageTo (if non-zero). Make sure that negative numbers are allowed so that this plug-in can be used to compare against some future "expiration date" kind of scenarios.

Generate and build the java classes.

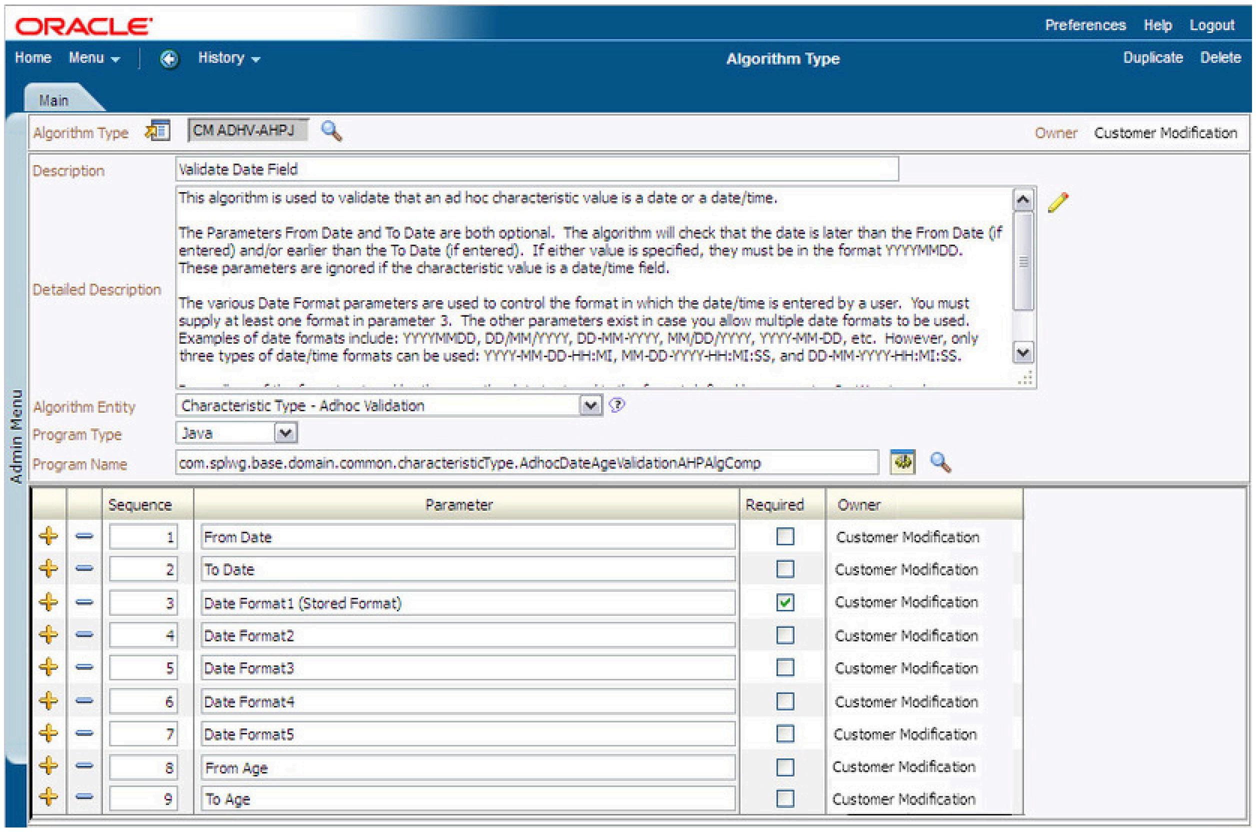
Note:
The various "Adhoc characteristic value validation" algorithms that come with the Oracle Utilities Software Development Kit are good references for algorithm plug-ins.

Add Algorithm Type

Add a new algorithm type copying most of the entries for ADHV-DTD:

- Algorithm Type: **CM ADHV-iiiJ** where **iii** is your initials.
- Description: **Validate Date Field (Age)**
- Long Description: **<Copy ADHV-DTD description here>. The Parameters From Age and To Age are optional decimals. The algorithm will check the "age" (current system date less the characteristic date / 365.25) is not less than the From Age (if non-zero) and is not more than To Age (if non-zero).**
- Algorithm Entity: **Char Type - Adhoc Value Validation**

- Program Type: **Java**
- Program name: **com.splwg.cm.domain.common.characteristicType.AdhocDateAgeValidationiiiAlgComp** where iii is your initials.
- Parameters:
    - Sequence: **1**, Parameter: **From Date**, Required: **Not Checked**
    - Sequence: **2**, Parameter: **To Date**, Required: **Not Checked**
    - Sequence: **3**, Parameter: **Date Format1 (Stored Format)**, Required: **Checked**
    - Sequence: **4**, Parameter: **Date Format2**, Required: **Not Checked**
    - Sequence: **5**, Parameter: **Date Format3**, Required: **Not Checked**
    - Sequence: **6**, Parameter: **Date Format4**, Required: **Not Checked**
    - Sequence: **7**, Parameter: **Date Format5**, Required: **Not Checked**
    - Sequence: **8**, Parameter: **Age From**, Required: **Not Checked**
    - Sequence: **9**, Parameter: **Age To**, Required: **Not Checked**



Algorithm Type: CM ADHV-AHPJ   Owner: Customer Modification

Description: Validate Date Field

Detailed Description:
This algorithm is used to validate that an ad hoc characteristic value is a date or a date/time.

The Parameters From Date and To Date are both optional. The algorithm will check that the date is later than the From Date (if entered) and/or earlier than the To Date (if entered). If either value is specified, they must be in the format YYYYMMDD. These parameters are ignored if the characteristic value is a date/time field.

The various Date Format parameters are used to control the format in which the date/time is entered by a user. You must supply at least one format in parameter 3. The other parameters exist in case you allow multiple date formats to be used. Examples of date formats include: YYYYMMDD, DD/MM/YYYY, DD-MM-YYYY, MM/DD/YYYY, YYYY-MM-DD, etc. However, only three types of date/time formats can be used: YYYY-MM-DD-HH:MI, MM-DD-YYYY-HH:MI:SS, and DD-MM-YYYY-HH:MI:SS.

Algorithm Entity: Characteristic Type - Adhoc Validation

Program Type: Java

Program Name: com.splwg.base.domain.common.characteristicType.AdhocDateAgeValidationAHPAlgComp

Sequence	Parameter	Required	Owner
1	From Date	☐	Customer Modification
2	To Date	☐	Customer Modification
3	Date Format1 (Stored Format)	✓	Customer Modification
4	Date Format2	☐	Customer Modification
5	Date Format3	☐	Customer Modification
6	Date Format4	☐	Customer Modification
7	Date Format5	☐	Customer Modification
8	From Age	☐	Customer Modification
9	To Age	☐	Customer Modification

Algorithm Type

Add Algorithm

Add a new algorithm as follows:

- Algorithm: **CM EXPDT-iii.**
- Description: **Date must be a future date**
- Algorithm Type: **CM ADHV-iiiJ**
- Effective Date: **1/1/2005**
- Parameters:
  - Sequence: **1**, Parameter: **blank**
  - Sequence: **2**, Parameter: **blank**
  - Sequence: **3**, Parameter: **YYYY-MM-DD**
  - Sequence: **4**, Parameter: **YYYY/MM/DD**
  - Sequence: **5**, Parameter: **MM-DD-YYYY**
  - Sequence: **6**, Parameter: **MM/DD/YYYY**
  - Sequence: **7**, Parameter: **MM.DD.YYYY**
  - Sequence: **8**, Parameter: **0.001**
  - Sequence: **9**, Parameter: **0**

Create References to New Algorithm

Create an ad hoc characteristic type and reference the previously created algorithm on it.

- Char type: **CM J-iii**
- Description**: iii's Adhoc validation test / Expiration Date**
- Type of Characteristic Value: **Ad hoc Value**
- Validation rule: **CM EXPDT-iii**
- Allow Search by Char Val: **Not Allowed**

Characteristic entity: choose **Notification Upload Staging**.

Maintaining Portals and Zones

The system uses portals and zones to display information throughout the system.

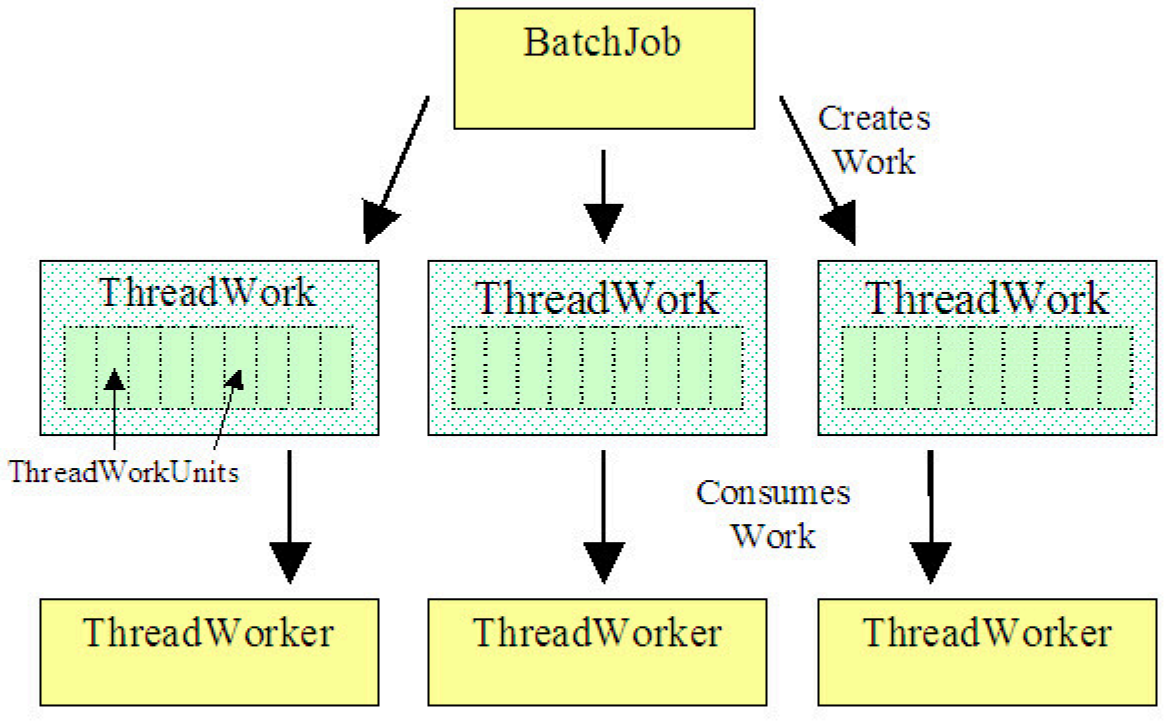
Note:
For more information on this topic, please refer to user documents Framework Administration, User Interface Tools, The Big Picture of Portals and Zones and Setting Up Portals and Zones.

Maintaining Background Processes

Maintaining Background Processes Overview

Each new background processes require the creation of two new classes: a BatchJob and a ThreadWorker. These classes fit into the "master-worker" pattern used by the background process runtime infrastructure to overcome the throughput limitations encountered by single-threaded processes. By splitting work among many concurrent threads often on multiple physical nodes background processes can achieve excellent scalability and react to changing work demands without additional programming. In this pattern:

- A BatchJob is responsible for determining the work to be processed for a batch run and then splitting that work into pieces that each ThreadWorker will process. When running a single process, a single BatchJob object is instantiated by the framework. The framework then makes calls to the BatchJob instance at the appropriate time. One such set of calls to the BatchJob instance is to return to the framework a collection of ThreadWork instances that will be distributed for execution.
- A ThreadWorker is responsible for processing a single ThreadWork instance for a run. Within the ThreadWork there are many WorkUnits representing the fine-grained units of work to be processed. In many cases the WorkUnits represent a complete database transaction, for example, a bill being created for an account. Whether or not the ThreadWorker executes on the same computer as other ThreadWorkers or the BatchJob that created its work is left as a configuration choice to be made at runtime. Within a single process, there may be many ThreadWorker objects. In general, each ThreadWorker instantiated in a batch run has a corresponding row in the Batch Instance table. The Batch Instance rows provide persistent state that is needed for the ThreadWorkers to operate correctly in failure/restart situations.



Creating a BatchJob

A BatchJob class is responsible for determining what work needs to be done within the run and splitting the work among ThreadWorkers.

The BatchJob Annotation

Each BatchJob class must declare a BatchJob annotation that specifies important attributes of the job. An example is shown below:

```
@BatchJob (rerunnable = false,

multiThreaded = true,

modules={todo},

softParameters = { @BatchJobSoftParameter

(name=OUTPUT-DIR, type=string) },

toDoCreation = @ToDoCreation (drillKeyEntity = user,

sortKeys = {lastName, firstName},

messageParameters = {firstName, lastName}

)

)
```

The annotation declares if the job can be rerun, supports more than one thread of operation, the modules that the job belongs to, its nonstandard runtime parameters and the details of how "ToDo" entries should be created in the case of errors. When not specified in the annotation, default values will be used.

Creating JobWork

The most important goal of a BatchJob class is to return an instance of JobWork describing what work should be done (ThreadWorkUnits) and have that work split into manageable chunks (ThreadWork) that can be processed by a single ThreadWorker.

Most commonly, ThreadWorkUnits contain only the ID values of the entities to be processed. For example, one can envision a process that performs an operation on a set of accounts. In general, one would expect that each ThreadWorkUnit would contain a single AccountId. The ThreadWorker objects would then be constructed in such a way that when asked to execute for a ThreadWorkUnit it would pull out the embedded AccountId and then perform the required business function.

There are convenience methods available from the AbstractBatchJob that make it easier to create JobWork instances. For example, the createJobWorkForEntityQuery(Query) method will accept a query returning BusinessEntity instances and create a JobWork instance containing the appropriate number of ThreadWork instances each containing (notwithstanding rounding) the same number of ThreadWorkUnits.

Declaring a ThreadWorker Class

It is the responsibility of the BatchJob to declare what class defines the ThreadWorkers that should perform the work. By returning a Class instance rather than ThreadWorker instances, the framework controls ThreadWorker instantiation which may occur on a different JVM than the one that the BatchJob instance resides.

Creating a ThreadWorker

The ThreadWorker performs the "heavy lifting" of a batch process. For a given run, there will ThreadWorkers created equal in number to the "thread count" parameter provided when a process is requested.

Initializing ThreadWork

Each ThreadWorker instance can expect to have its initializeThreadWork() method called once by the framework before any actual work is to be performed. This method may be implemented to do any setup necessary for that thread's execution, most commonly output files opened or variables initialized.

Warning:
It is very important that any setup necessary to execute a WorkUnit is done here and not in the creation of JobWork, this includes accessing batch parameters. There is no guarantee that static variables set at the time of JobWork creation will be available at this time. The framework may be calling ThreadWork in a different process from the creation of JobWork.

Executing a WorkUnit

The ThreadWorker can expect that its executeWorkUnit method will be called once for each ThreadWorkUnit that that ThreadWorker will process. For example, if the batch process will act upon 10,000 accounts and the process is submitted with a ThreadCount=10, we can expect that there are 10 ThreadWorkers created by the framework and each worker will have its executeWorkUnit method called by the framework for each of the 1,000 ThreadWorkUnits allocated to that thread.

Finalizing ThreadWork

Each ThreadWorker instance can expect to have its finalizeThreadWork() method called once after all ThreadWorkUnits have been processed. This gives the opportunity to close any open files or to do any other "tear down" processing for the ThreadWorker.

Choosing a ThreadExecutionStrategy

ThreadWork instances need to provide a strategy defining the execution policies for its work. That is, how the work for a thread will be processed. The interface that is implemented is ThreadExecutionStrategy. The most important aspect of this is how exceptions will be treated with respect to transactions.

- Should all the ThreadWorkUnits be wrapped in a single transaction with a single rollback on an exception?
- Should each ThreadWorkUnit be in its own transaction?
- Should the framework attempt to process many ThreadWorkUnits within a single transaction?
- If an exception occurs should the framework "back up" and reprocess the successful units?
- In general, new background processes are expected to chose from existing instances of ThreadExecutionStrategy, not create new ones. Please scan for existing implementations of ThreadExecutionStrategy.

Creating Javadocs for CM Source Code

Javadocs can be created for CM source code. They are designed to be integrated into the product's Javadocs.

The product's Javadocs are only delivered for objects or supporting objects that are intended to be referenced by CM code. For instance, only the domain and api packages are included, and some of the files created by the artifact generator are not delivered since they have no practical relevance to CM code. These files have been deliberately and explicitly omitted when creating the product's Javadocs.

Note that the process that generates Javadocs on CM source code is not selective, and running Javadocs on CM source code may include more object types than what is delivered with the product's Javadocs.

Refer to Creating Javadocs for CM Source Code *(on page 85)* for more information.

Upgrade JSP to XSLT

Trees and subpanels should be upgraded to use the application's XSLTs instead of the JSPs used in v1.5.x. This section describes the upgrade process.

Note that all other JSPs (tab pages, list grids, etc) must have been upgraded to XSLTs in v1.5.x. Thus, there is no tool to upgrade such code in V2.

Create User Exit Files

The user exits in the JSP-based system were directly placed within the JSP as code snippets within specially located markers. In the XSLT system, the meta-data is separated from the user exit, and resides in an .xjs file with the same name as the JSP file, with only user exits and each user exit function explicitly defined in the file.

Going from JSP to XSLT user exits is thus not trivial. However, a set of supplied scripts will create a new user exit .xjs file for each of the different JSP template files in your system. The following table lists the scripts to run for each template file:

Sub Panel	convertSubPanel.pl
Tree Page	convertTreePage.pl

Tree User Exit Changes

Since the XSLT user exits are now callout functions, five tree user exits need to be coded differently. The main purpose of these five user exits is to change a variable's value. The analogous new XSLT user exits return the desired value instead.

The user exits have been renamed to more accurately reflect their new function. Below is a comparison of names and purpose of JSP-based user exits versus XSLT-based user exits.

JSP name	Main purpose	XSLT name	Main purpose
setServiceIndex	Sets the desired serviceIndex variable.	overrideServiceIndex	Returns the desired index of the service.
setNavKey	Sets the desired newNavKey variable.	overrideNavKey	Returns the desired nav key.
setNavKeyIndex	Sets the desired navKeyIndex variable.	overrideNavKeyIndex	Returns the desired index of the nav key.
setImageOpenIndex	Sets the desired imageIndex variable.	overrideImageOpenIndex	Returns the desired index of the open image.
setImageClosedIndex	Sets the desired imageIndex variable.	overrideImageClosedIndex	Returns the desired index of the closed index.

Each user exit is passed the variable's original value. If the user exit does not return a value, the original variable's value will be used.

Below is an example of a JSP user exit and a converted XSLT user exit inside an .xjs file.

Here is the JSP user exit.

```
// $#BSES SETSERVICE

if (nodeName == 'newtype') {

   var myLetter = pageKeys.FT_TYPE.substr(0,1);

   if (myLetter == 'A') {

     serviceIndex = 1;

   }

   if (myLetter == 'B') {

     serviceIndex = 2;
   }
   if (myLetter == 'C') {
     serviceIndex = 3;
   }
   if (myLetter == 'P') {
     serviceIndex = 4;
   }
 }
// $#BSEE SETSERVICE
```

Here is the same user exit coded in an .xjs file:

```
function overrideServiceIndex(nodeName, services, pageKeys, serviceIndex) {
   var overrideIndex;
   if (nodeName == 'newtype') {
      var myLetter = pageKeys.FT_TYPE.substr(0,1);
      if (myLetter == 'A') {
        overrideIndex = 1;
      }
      if (myLetter == 'B') {
        overrideIndex = 2;
      }
      if (myLetter == 'C') {
        overrideIndex = 3;
      }
      if (myLetter == 'P') {
        overrideIndex = 4;
      }
   }
return overrideIndex;
}
```

Change Template Code in Program Components

The meta-data on the database for the CM JSP program component pages needs to point to the new XSLT template codes. There is a set of SQL scripts that update all the CM tree and sub panel program components with the correct template.

Run the SQLs in the script changeTemplateCodesTTRAndPN.sql against your database to perform this change.

Create XML File with UI Meta-data

The XSLT framework uses the meta-data at run-time to drive the transform. However, it does not query the database for this, and instead relies upon an intermediate representation in the form of an XML file stored with the same name (but with an .xjs extension) and location as the original JSP file.

The XML file is automatically created by the framework the first time the page is viewed if an existing XML file does not exist. Delete the existing XML file, if one exists, located in the same directory as the original JSP file. The XML file is named after the program component's name. To generate the XML file, view the program component from within the application.

Delete the JSP Files

Once the meta-data is changed and the new files are properly placed, there is no longer a need for the JSP files for the converted program components, and it would be a good idea to delete them to avoid confusion.

Find the JSP files in the file system and delete them.

Log into the Application and Test

For the new XSLT pages to be used by the system, instead of the system looking for the old JSPs, some server and browser caches need to be flushed. The easiest thing to do is restart the app server and start a new browser session.

Login and visit the converted pages to test functionality.

Utilities

Environment Batch Programs

displayEnvironment.bat

Property	Detail
Purpose	Displays the current configuration.
Description	Displays a set of environment variables and settings that may be needed to diagnose compile issues.
Usage	displayEnvironment.bat
Parameters	None.

switchEnvironments.bat

Property	Detail
Purpose	Sets the current development environment (project) for the Software Development Kit.
Description	Displays a list of development environments on the development client, allows the user to select one, and sets it as the current development environment for the Software Development Kit.
Usage	switchEnvironments.bat
Parameters	None.

createNewEnv.bat

Property	Detail
Purpose	Creates a new development environment (project) or configures a development environment to use the version of the Software Development Kit used for the current development environment.
Description	Configures a new app server to be a development environment. Also, executing this for an existing development environment configures that development environment to use the version of the Software Development Kit used by the current development environment.
Usage	createNewEnv.bat -a <appServerDir>
Parameters	• **-a <appServerDir>**. Specify the base directory of the app server to configure.

Services

Batch Program setupSvcXMLPrompted.bat

Property	Detail
Purpose	Setup service XML.
Description	After prompting the user for the program name of the service, this script sets up a service by creating the service XML file.

Property	Detail
Usage	setupSvcXmlPrompted.bat
Parameters	None.

Batch Program updateXMLMetaInfo.bat

Property	Detail
Purpose	Updates the XML Metainfo directory with the latest service XMLs.
Description	Updates the XML Metainfo directory of the current development environment with the latest service XMLs. This is needed, for example, for creating schemas for XAI.
Usage	updateXMLMetainfo.bat
Parameters	None.

Eclipse Tools/Wizards

There are a few wizards and tools available for developing against the framework within Eclipse plugins.

Batch Program startEclipse.cmd

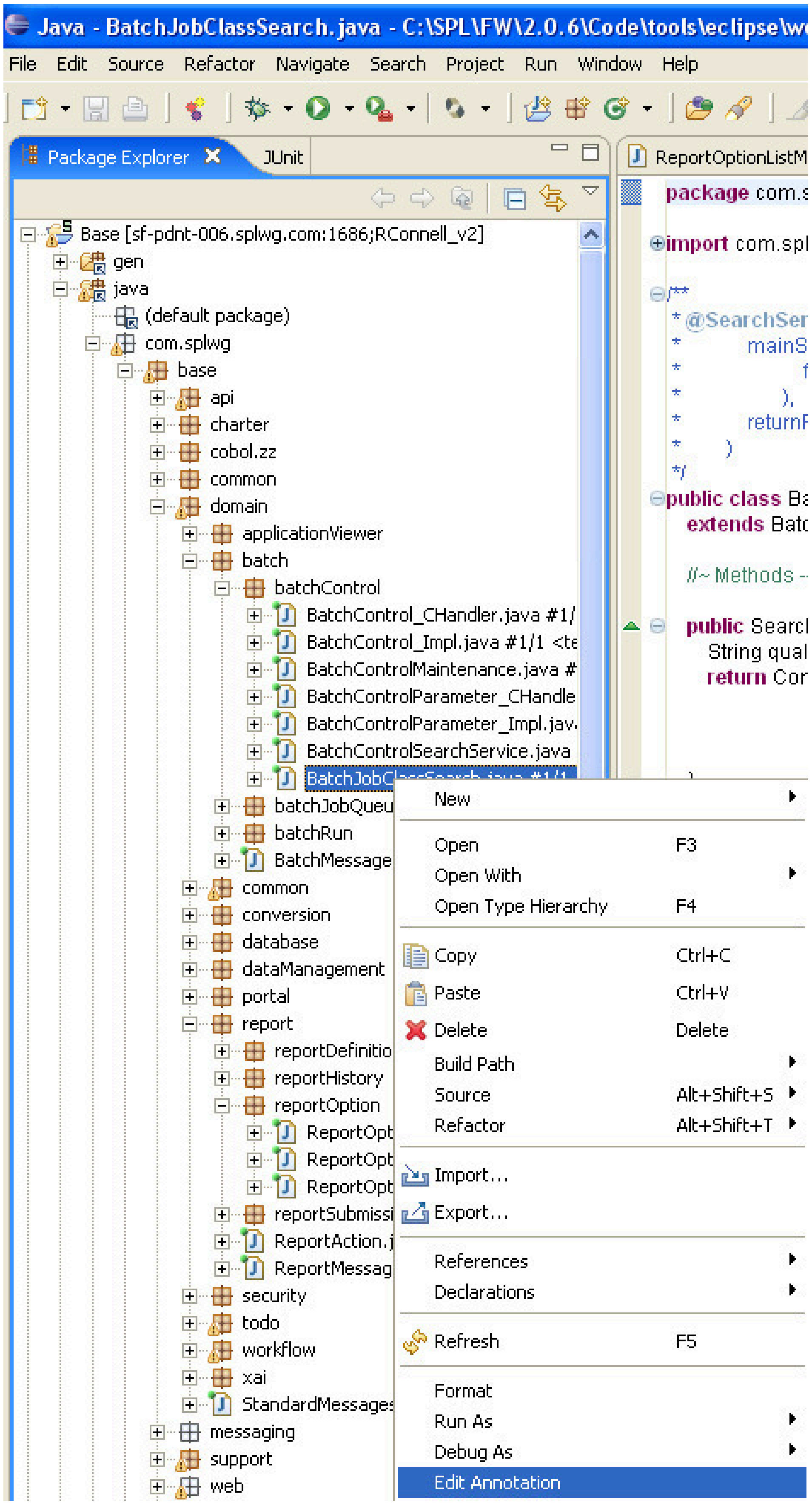
Property	Detail
Purpose	Launch the Ecliipse SDK for the current development environment (project).

Property	Detail
Description	Launches the Eclipse SDK for the current development environment.
Usage	startEclipse.cmd.
Parameters	None.

Annotation Editor

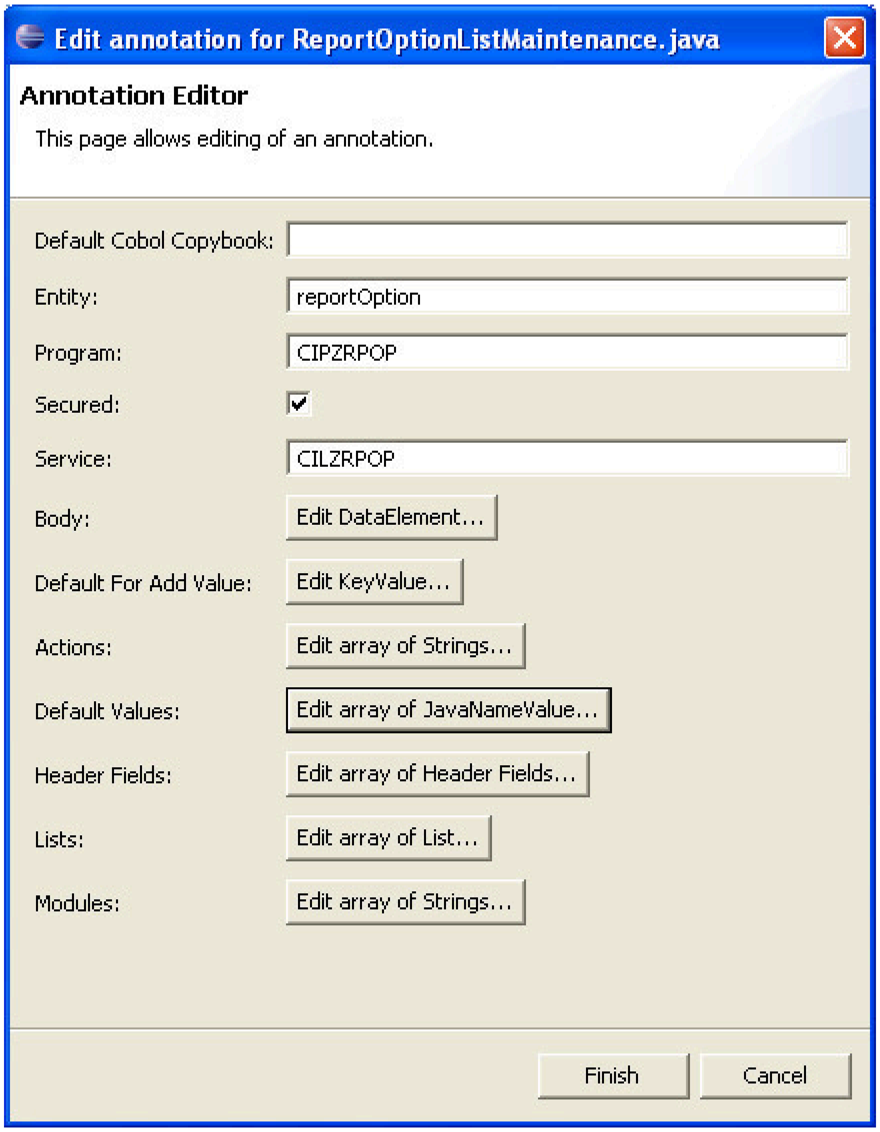
A lot of the Java classes that will be created to add behavior to the application require Annotations to provide meta-data about the implementation (see Java Annotations chapter in the Developer Guides).

The annotation editor plugin provides a convenient way to edit the annotations on these classes. It is available on any class that has an existing annotation, under the Package Explorer panel in Eclipse. Right click on the file in the Package Explorer, and there will be a menu item "Edit Annotation".



```
package com.s
import com.spl
/**
 * @SearchSer
 *         mainS
 *             f
 *           ),
 *         returnF
 *     )
 */
public class Ba
  extends Batc

  //~ Methods -

  public Searc
    String qual
    return Cor
```

Choosing this menu item will cause a new dialog window to appear, and the file to open into an editor if it is not already open. The dialog that appears will allow maintenance of the file's current annotation contents.



Edit annotation for ReportOptionListMaintenance.java

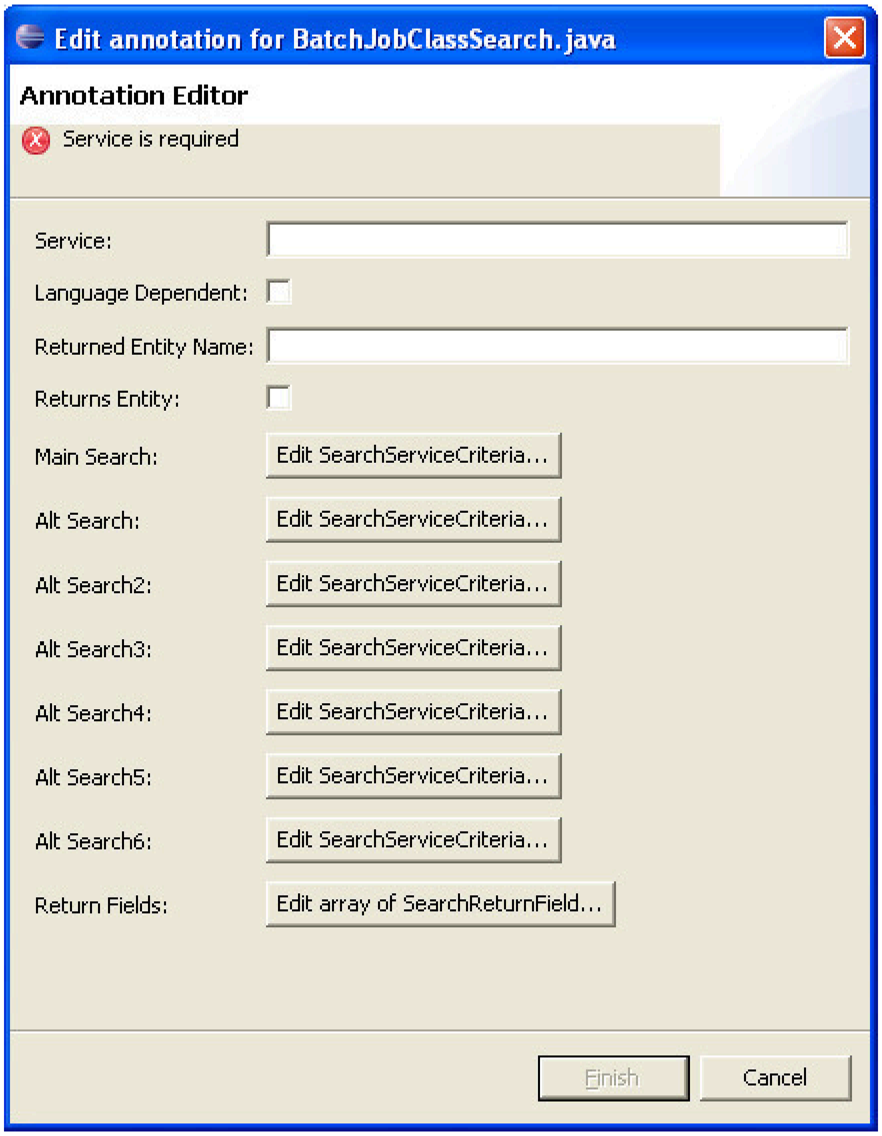
Annotation Editor

This page allows editing of an annotation.

Default Cobol Copybook:	
Entity:	reportOption
Program:	CIPZRPOP
Secured:	✓
Service:	CILZRPOP
Body:	Edit DataElement...
Default For Add Value:	Edit KeyValue...
Actions:	Edit array of Strings...
Default Values:	Edit array of JavaNameValue...
Header Fields:	Edit array of Header Fields...
Lists:	Edit array of List...
Modules:	Edit array of Strings...

The appearance of the dialog is dependent upon the particular annotation, but the standard dialog will present a layout of two columns, a label and an input for each annotation property. The bottom of the dialog will always present the Finish and Cancel buttons. The Cancel button is always available, and will throw away any changes made, leaving the file with the annotation unchanged.

The Finish button will only be enabled when the annotation has no errors. The annotation is validated after any change, and errors will be displayed near the top of the dialog and the Finish button disabled.



Edit annotation for BatchJobClassSearch.java

Annotation Editor

Service is required

Service:

Language Dependent:

Returned Entity Name:

Returns Entity:

Main Search: Edit SearchServiceCriteria...

Alt Search: Edit SearchServiceCriteria...

Alt Search2: Edit SearchServiceCriteria...

Alt Search3: Edit SearchServiceCriteria...

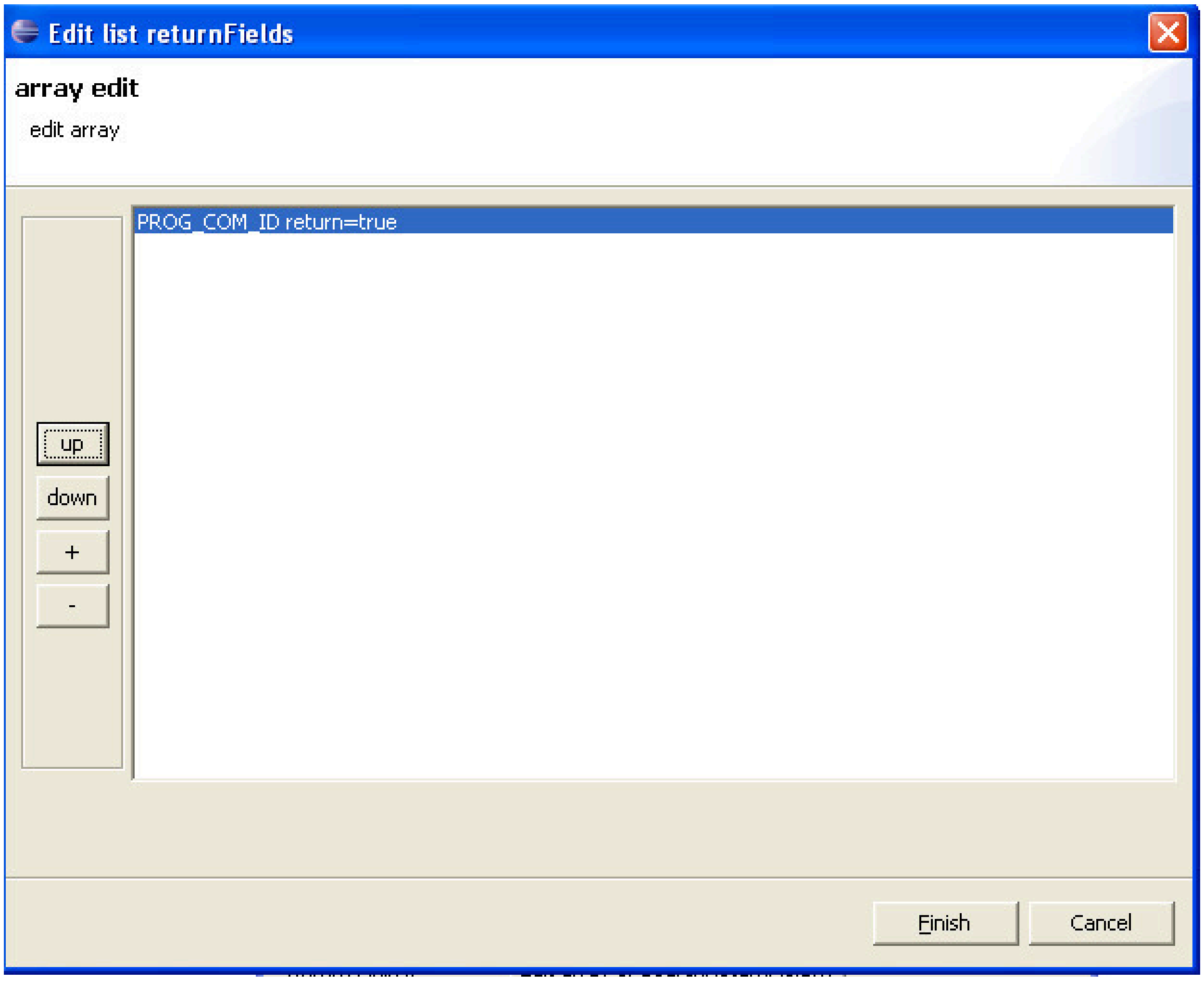
Alt Search4: Edit SearchServiceCriteria...

Alt Search5: Edit SearchServiceCriteria...

Alt Search6: Edit SearchServiceCriteria...

Return Fields: Edit array of SearchReturnField...

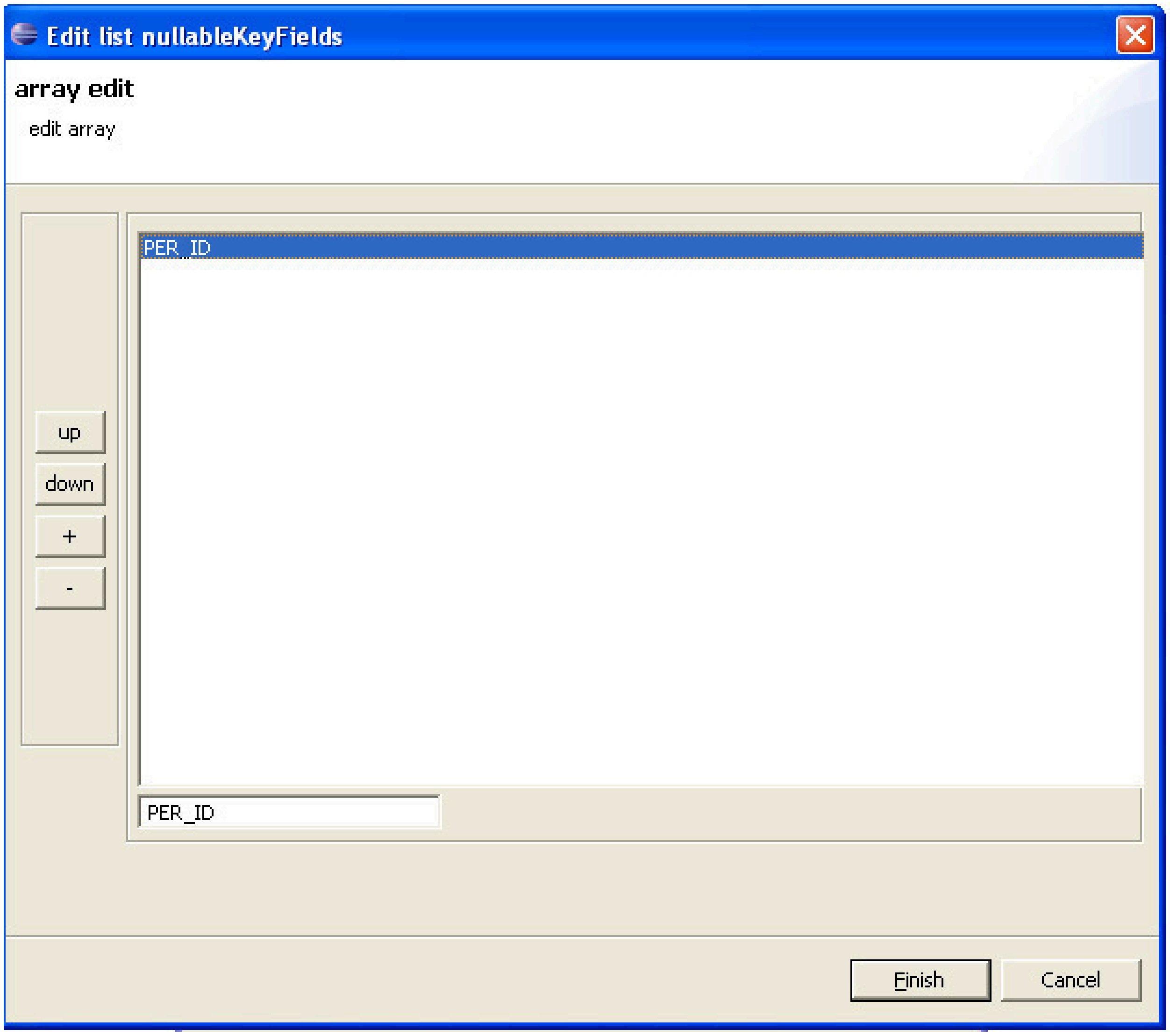
When the property value is itself a list of values or another annotation, there will a button instead of an input text box. Clicking the button will bring up another dialog to edit its information. In the case of lists, there is a standard dialog where elements can be added, deleted, or reordered.



PROG_COM_ID return=true

To add a new element, click the '+' button. This will popup a new dialog for the annotation being added (or sometimes a choice of the new annotation's type might need to be chosen first). Clicking the '-' button will delete the highlighted element. The 'up' and 'down' buttons can be used to move the highlighted element up or down within the list. To modify an existing element, double click its row in the list dialog, and a new dialog will open to edit its values.

Finally, there is a special list dialog for lists of strings. Instead of editing the elements in the list in a new dialog, a single input field near the bottom of the dialog is used to edit the highlighted entry.



PER_ID

PER_ID

When the finish button is finally pressed, the annotation (and only the annotation) in the file will change to contain the new values entered into the annotation dialogs. The changed file's annotation may be slightly reformatted. The changed file will also remain unsaved, pending user review of the annotation's changes.

Project database information

In order to use the Maintenance Object Wizard described in the next section, some information will have to be provided in order for Eclipse to connect to the database to retrieve the Maintenance Object metadata.

There are two ways to specify the database connection information.

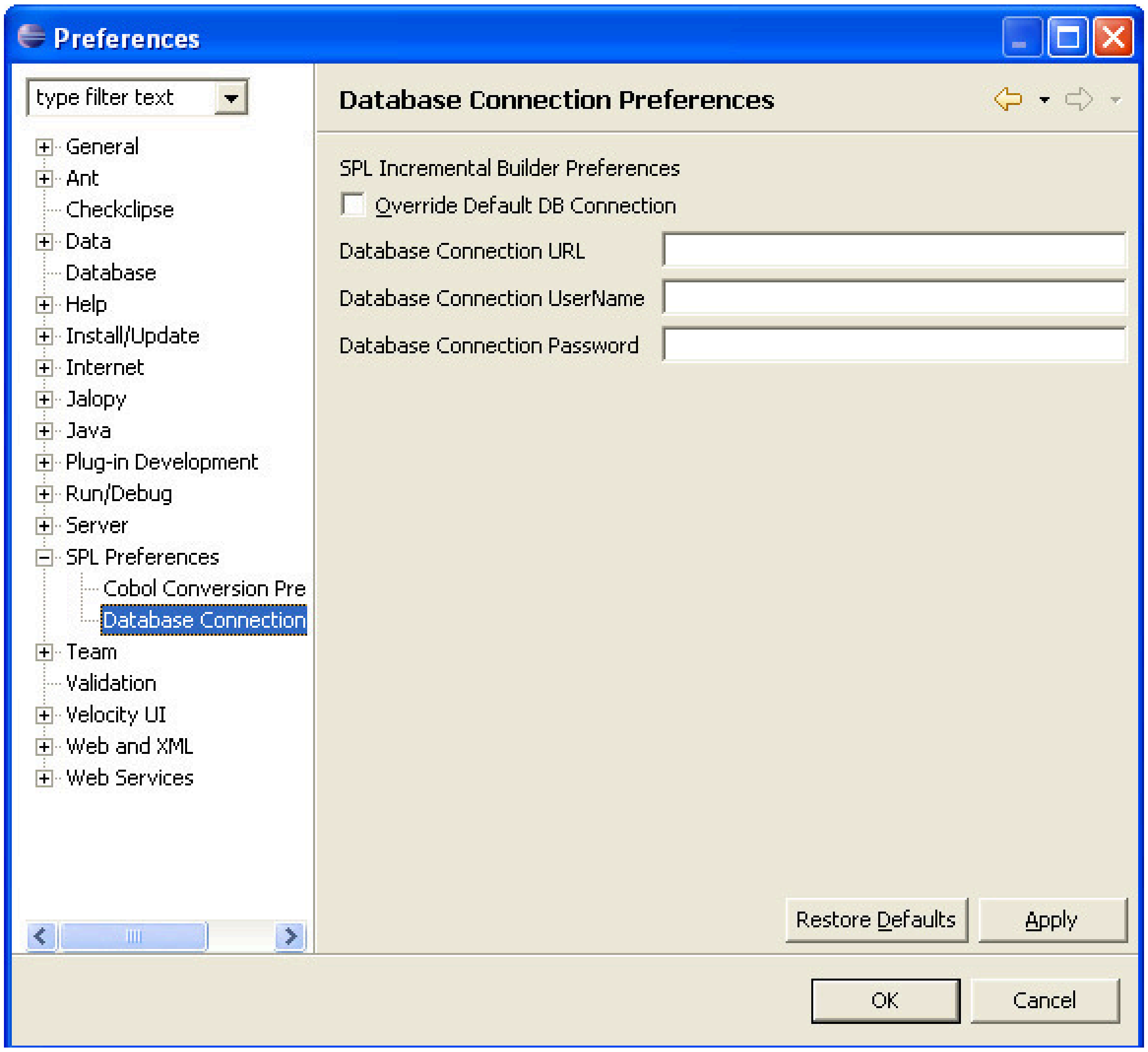
The first is a way for each project to possibly specify different connection information. This is done in the `.project` file stored in the project's directory. This is an XML file that describes the project. The database information can be supplied in a **buildCommand** node under **buildSpec** under the **projectDescription** root node:

```
<buildSpec>
<buildCommand>
<name>com.splwg.tools.dbConnection</name>
<arguments>
<dictionary>
<key>url</key>
<value><URL></value>
</dictionary>
<dictionary>
<key>username</key>
<value><USERNAME></value>
</dictionary>
<dictionary>
<key>password</key>
<value><PASSWORD></value>
</dictionary>
</arguments>
</buildCommand>
...
</buildSpec>
```

The values <URL>, <USERNAME>, <PASSWORD> should be replaced (including the surrounding '<' and '>') with the appropriate values for the database for the project.

This file will need to be hand edited, and Eclipse should be restarted after the edit is complete.

The second way is to provide a workspace-wide database connection. This is available in an Eclipse preference- go to "Window | Preferences...". Then in the tree pane on the left of the Preferences dialog, choose "SPL Preferences". Under OUAF preferences, choose "Database Connection". The preference pane on the right will now show inputs for the database connection information.



Database Connection Preferences

SPL Incremental Builder Preferences

Override Default DB Connection

Database Connection URL

Database Connection UserName

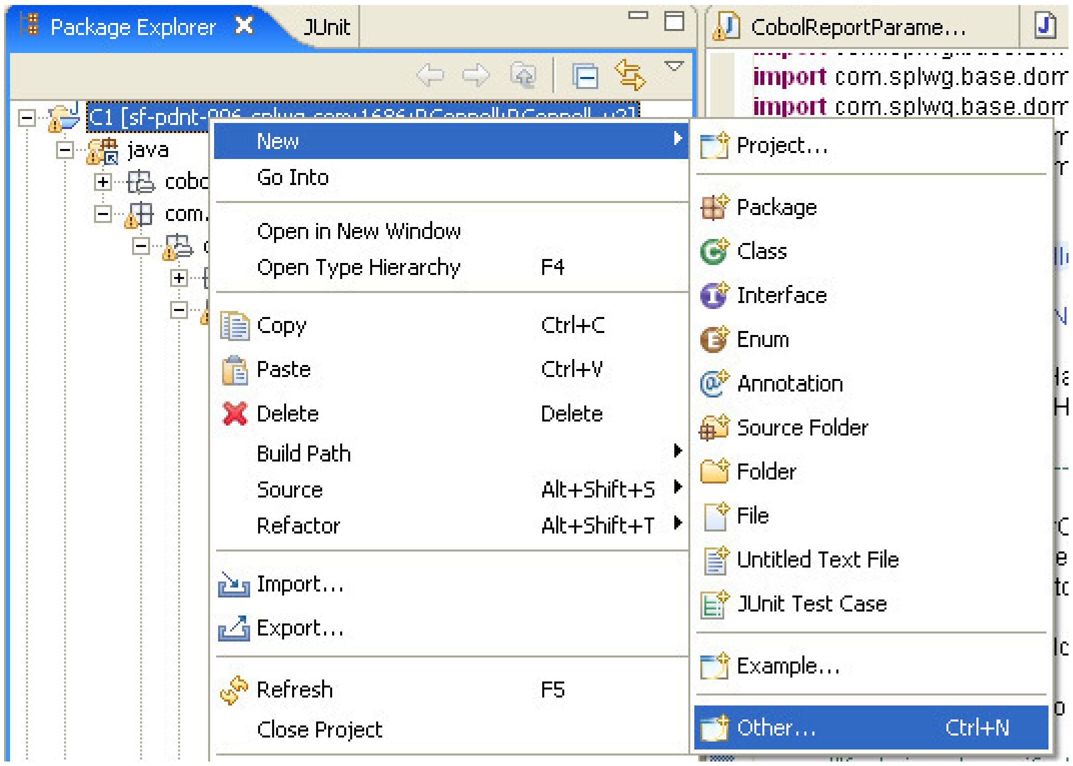
Database Connection Password

Click the override default DB connection if the database contains materialized views to the true development database for performance reasons. Enter the information into the appropriate text boxes and click OK. This will take effect immediately, without need of restarting Eclipse.

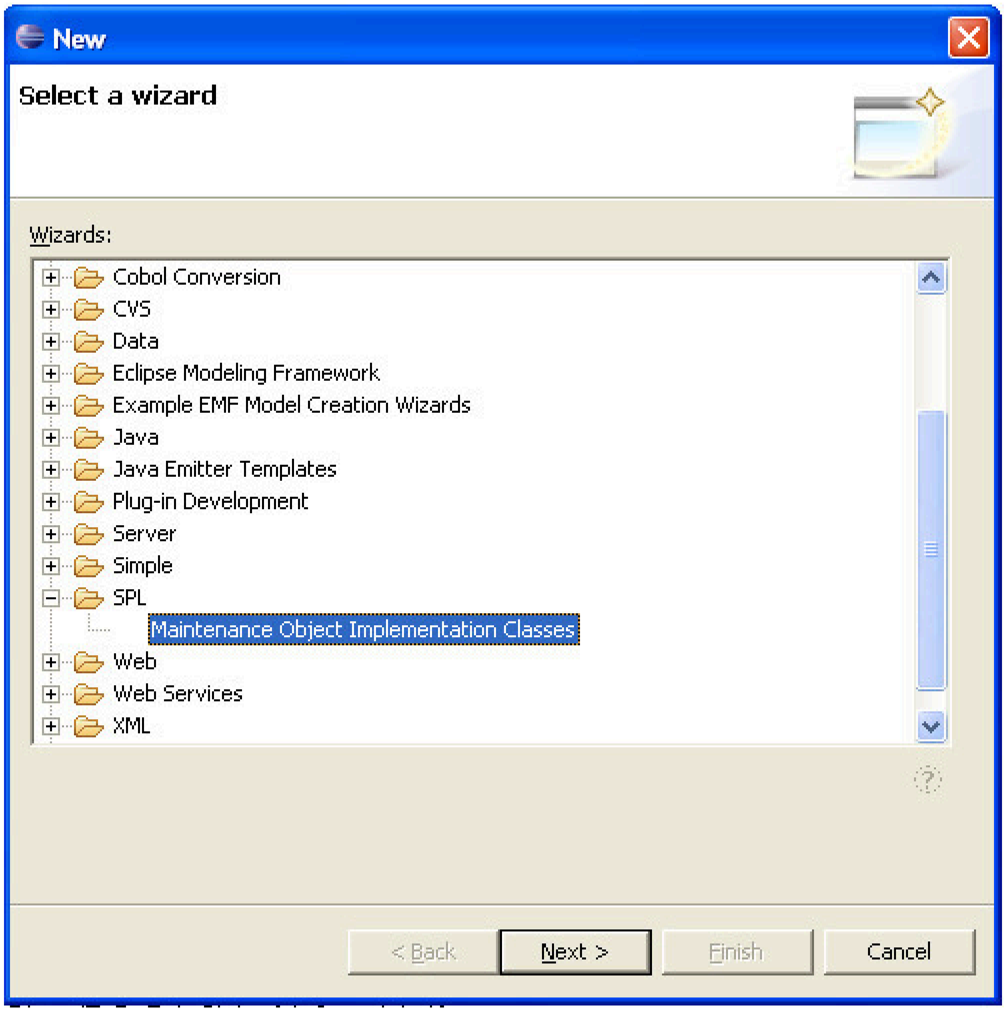
Maintenance Object wizard

In cases where a whole new "Maintenance object" is being added to the application, and the data is first entered onto the CI_MD_MO and related tables, there is a wizard that will use that meta data as a starting point and with some developer input, create all of the manually coded Java Entity_Impl classes with their proper tree structure, and also optionally create a Java Maintenance class "starting point".

The "Maintenance Object Wizard" is available under the "New .." menu item, either under the file menu, or by right clicking a node in the package explorer (the package explorer option is recommended, as it will default the project and source directory selected). From the list of new wizards available, choose "Other...".

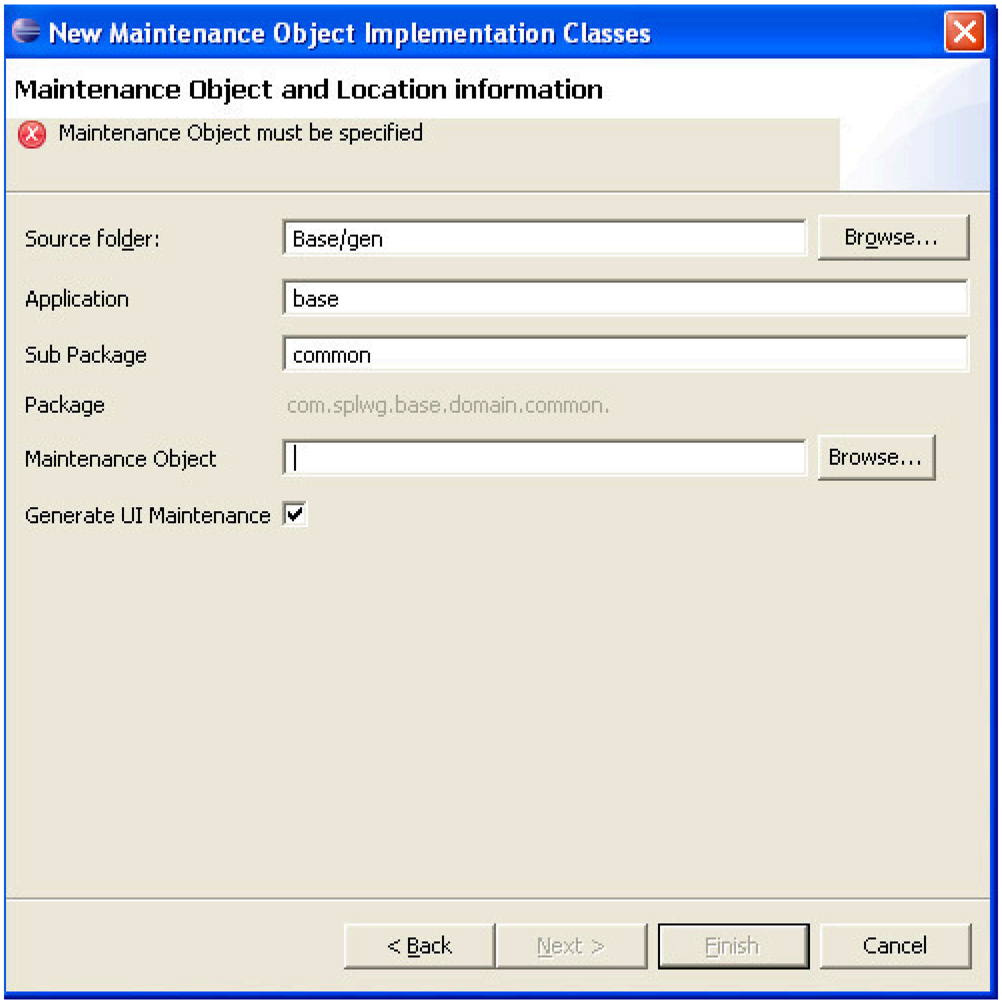


This will open a new dialog, where the maintenance object wizard is under SPL:



(Note that you can configure Eclipse so that in the future the "Maintenance Object implementation classes" wizard appears directly under the first "New..." menu.)

The first page of the wizard asks for the project and source path to place the new files. Then it asks for the some information it uses to construct the package name for the new files. The standard is that the new classes go under the application's domain path, with a possible extra sub package (e.g., a subsystem, like 'common' or 'customerInformation'), followed by the top level entity's name (e.g., account). The top level entity's name, along with lots of data used by the next page, comes from the maintenance object itself.


New Maintenance Object Implementation Classes

Maintenance Object and Location information

Maintenance Object must be specified

Source folder: Base/gen

Application: base

Sub Package: common

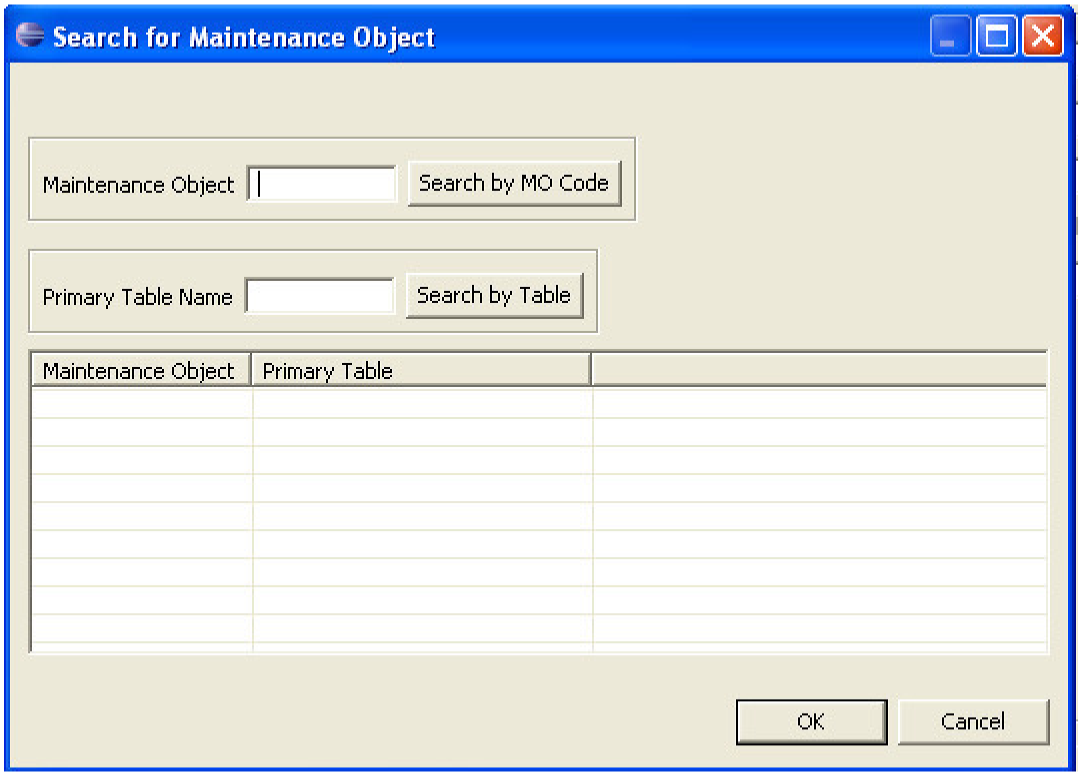
Package: com.splwg.base.domain.common.

Maintenance Object:

Generate UI Maintenance: ✓

Finally, you can optionally choose to Generate the UI Maintenance (the default is to generate it).

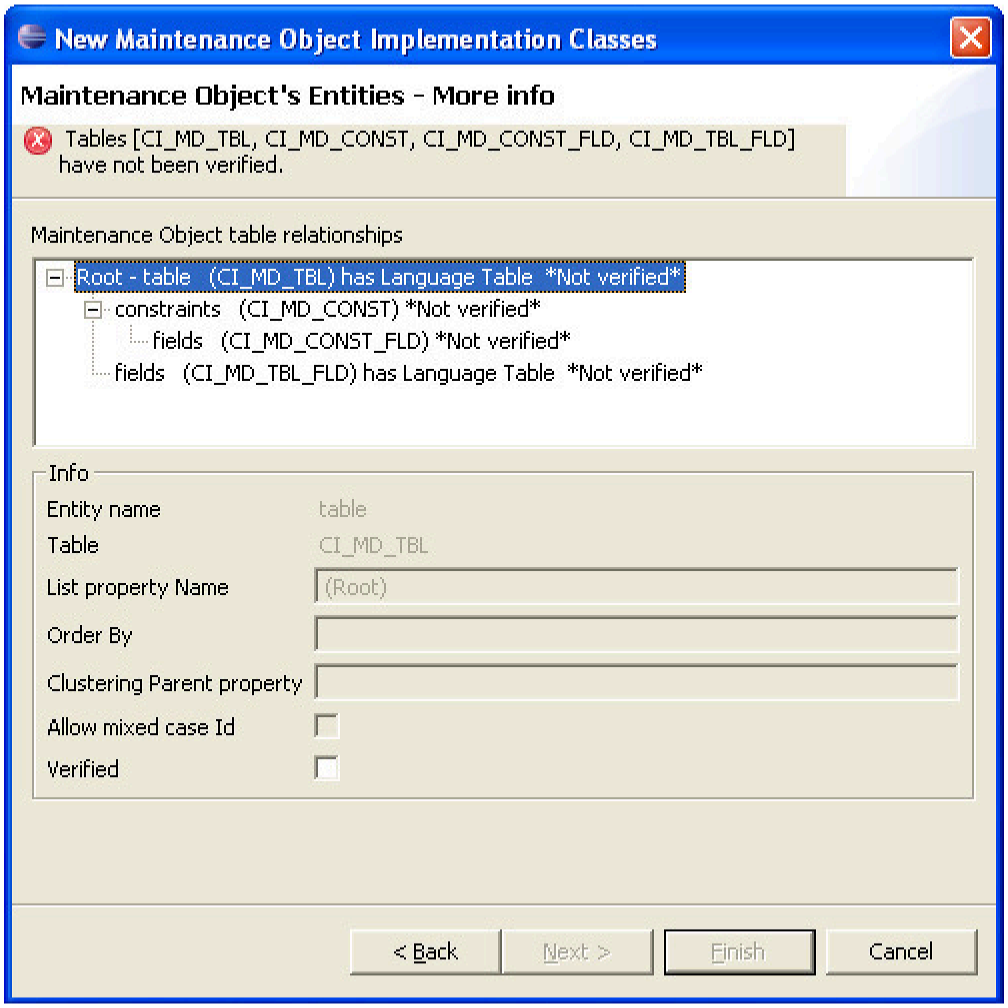
The maintenance object input has a "Browse..." button associated with it that will launch a search dialog where the maintenance object can be searched for by either the Maintenance Object's code, or by the primary table for the maintenance object.



Both of these searches are "likable" in that partial matches starting with the input will be shown.

Pressing OK on the search or double clicking a row will bring the selected maintenance object back into the Maintenance Object input on the main wizard.

Once the main wizard's inputs are specified, the next button can be pressed. This will display the second detailed wizard page. A tree view is displayed with the tree representation of the Maintenance object selected, with its child tables.



Maintenance Object's Entities - More info

Tables [CI_MD_TBL, CI_MD_CONST, CI_MD_CONST_FLD, CI_MD_TBL_FLD] have not been verified.

Maintenance Object table relationships

```
⊟ Root - table   (CI_MD_TBL) has Language Table  *Not verified*
    ⊟ constraints   (CI_MD_CONST) *Not verified*
        fields   (CI_MD_CONST_FLD) *Not verified*
    fields   (CI_MD_TBL_FLD) has Language Table  *Not verified*
```

Info

Entity name	table
Table	CI_MD_TBL
List property Name	(Root)
Order By	
Clustering Parent property	
Allow mixed case Id	☐
Verified	☐

Each node in the tree must be visited in order to enter some information or at least verify that the default data is correct. The nodes themselves show the list property name, the table, whether the table has a language table, and whether the node has been verified.

The selected node's data is shown in the "Info" box below the tree. Only editable information is available to be changed- other values may be disabled. The values that can be changed for each node include the list property name, the order by fields, the clustering parent property, and whether the Id can contain mixed case.

The list property name is the name on the parent that the child collection will be accessed by. For example, in the above Maintenance Object for Table, the table's child collection of rows on the table CI_MD_CONST will be accessed via the property 'constraints'. And likewise, each constraint will then have a child collection called fields.

The order by is an optional property. It is a comma-separated list of columns that specify the order in which the list will be retrieved when the collection is read from the database. An example for the constraints collection would be "CONST_ID, OWNER_FLG".

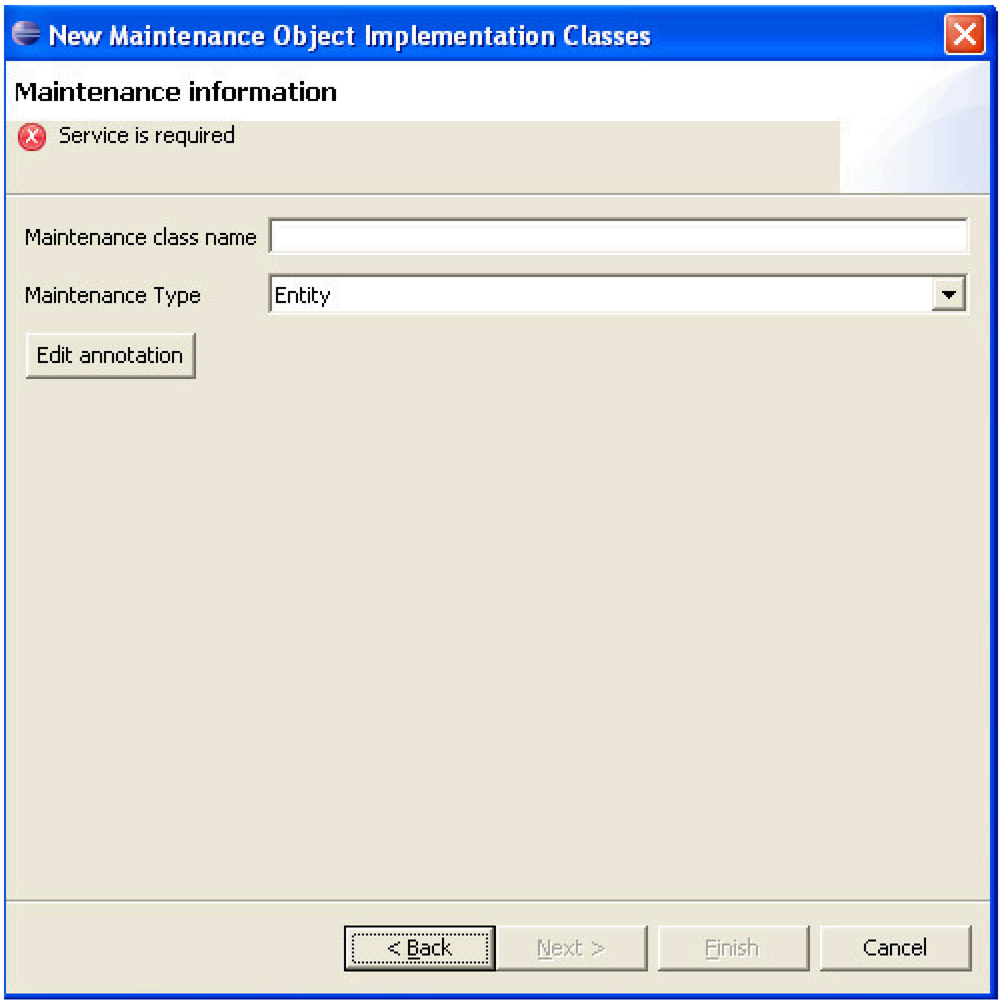
The clustering parent property comes into play for generated IDs. In some cases, it is beneficial to cluster the generated keys for related objects so that batch threading is more efficient. An example is every Service Agreement can have its ID generated with some portion of its account ID. The account will be accessed off of a serviceAgreement via the property account. Thus the Service Agreement root node in the above dialog would probably have a value of 'account' for the clustering parent property.

In case of user defined string keys, most of the time the application only uses uppercase keys. However, in some cases, mixed case keys are allowed. The "Allow mixed case Id" check box should be checked in this event.

Finally, to ensure that the developer reviews each node's values, the Verified check box must be checked for each node, prior to proceeding to the next page or finishing.

If the option to generate the Maintenance was not chosen, the Finish button will be enabled when the tree nodes' data is complete and valid. Clicking finish will cause all of the entity classes to be created in the specified package, and will open an editor window on each new class.

If the option to generate the Maintenance was chosen on the first page, the "Next>" button should be enabled after all the tree nodes data is finished and valid. Clicking "Next" will then present the final wizard page, where information about the maintenance class can be entered.


New Maintenance Object Implementation Classes

Maintenance information

Service is required

Maintenance class name

Maintenance Type: Entity

On this page, the maintenance class name and maintenance type can be chosen. The maintenance class name is something like the root entity name followed by 'Maintenance" by convention, although it can differ. The maintenance type choices are 'Entity', which is a standard maintenance for a single instance of the maintenance object at a time, with nested child lists, etc. The other choice is 'List", which is a simplified maintenance where many instances of the maintenance object are edited at once in a grid. This is usually limited to simple objects with a code and description, and maybe one or two other fields. Anything more complex would be difficult to present in the single grid.

(Note that changing the Maintenance Type will clear out any existing information on the annotation.)

After the class name and maintenance type is chosen, there is more information required to be edited on the Annotation. See Java Annotation in the Developer Guide for details about annotations. Clicking on the "Edit Annotation" button will launch a new dialog window for editing the annotation. The most important information that every maintenance must specify on the annotation is the service name. This field is immediately visible on the main dialog for the annotation, and must have a value entered. Most everything else will have been defaulted with appropriate values from the Maintenance Object meta data. See the developer guide mentioned above for more information on using the annotation editor.

After the maintenance information and annotation is complete and valid, pressing finish will cause the entity files and an empty maintenance class to be created, and editor windows opened on each of them.

Upgrade JSP to XSLT

Note:
JSPs other than trees and subpanels must have been upgraded to XSLTs in v1.5.x. Thus, there is no tool to upgrade such code in V2.

Batch Program convertTreePageExits.pl

convertTreePageExits Purpose

Creates user exit files from tree JSP files.

convertTreePageExits Description

This program creates user exit .xjs files for all tree JSP files under the current and child directories. The .xjs file will be created in the same directory with the same name as the JSP.

This should be run from a command prompt in the directory or parent directory of the JSP files.

convertTreePageExits Usage

Perl convertTreePageExits.pl

Batch Program convertSubPanelExits.pl

convertSubPanelExits Purpose

Creates user exit files from subpanel JSP files.

convertSubPanelExits Description

This program creates .xjs files for all subpanel JSP files under the current and child directories. The .xjs file will be created in the same directory with the same name as the JSP.

This should be run from a command prompt in the directory or parent directory of the JSP files.

convertSubPanelExits Usage

Perl convertSubPanelExits.pl

SQL Script changeTemplateCodesTTRAndPN.pl

changeTemplateCodesTTRAndPN Purpose

Changes the tree and subpanel template codes to the XSLT template codes.

changeTemplateCodesTTRAndPN Description

This changes the template codes of from JSP to XSLT template codes. This template code instructs the application to use the XSLT engine instead of the referring to a JSP.

These SQL commands should be run against the database.

Javadocs

Batch Program generateJavadoc.bat

Property	Detail
Purpose	Create javadocs from custom source code.
Description	This script runs the javadoc tool bundled with the jdk against CM source code in the standard directory and targets the javadocs directory. To integrate the javadocs with the product's javadocs the reindex tool needs to be run.
Usage	generateJavadoc.bat
Parameters	None.

Batch Program reindexJavadoc.bat

Property	Detail
Purpose	Recreate the Javadoc indices.
Description	This script recreates the Javadoc indices so that it shows all of the Javadocs in the Javadoc folder. If Javadocs have been generated for CM code, this will update the indices to include both the CM and the product's classes and packages.
Usage	reindexJavadoc.bat
Parameters	None.

Developer Guide

Overview

The Oracle Utilities Application Framework provides a rich environment for developing applications. This document provides a reference for various topics that will help developers make the most of this application development framework. The sections in this document include:

- The **Java Annotations** section describes the meta-data that can be embedded in Java code for various purposes.
- The **Public API** section describes available methods, interfaces, etc., in the various Java classes like entities, maintenance classes, etc.
- The **Application Logs** section describes how logs are set up and used.
- The **Java Programming Standards** section describes Java coding practices that promote efficient development and maintenance as well as upgradeability.
- The **HQL Programming Standards** section describes HQL coding practices that promote efficient development and maintenance as well as upgradeability.
- The **SQL Programming Standards** section describes SQL coding practices that promote efficient development and maintenance as well as upgradeability.

- The **Database Design Standards** section describes database design practices that promote an efficient database, maintenance as well as upgradeability.
- The **System Table Guide** section describes the set of database tables that contain crucial information for the configuration and operation of the application. It also describes standards to be followed to ensure upgradeability.
- The **Key Generation** section describes the automatic generation of random and sequential primary keys.

Java Annotations

In order to direct the application how to deal with the code in certain classes, annotations are employed. These annotations can direct the generator how to generate the superclass, how to register the class, and at runtime can effect the behavior of the class. The annotations are potent metadata used at several levels in the application.

Technically, the annotations are structures described inside a JavaDoc comment prior to the start of classes or methods. They are structured via starting with a '@' sign, followed by the annotation name, and the body of the annotation inside parenthesis. The body can be either comma separated key=value pairs or a single value which specifies a value for a unique default key. The values can be any of strings (needing to be bound by quotes if there are special characters inside the string itself), lists (of either annotations or strings) bound by curly braces {} and separated by commas, or other annotations.

Each managed class (entity, change handler, business component, maintenance, etc.) typically has its own annotation. Each of these annotations has an underlying Java class in the com.splwg.shared.annotations package, where the name of the class is the name of the annotation suffixed by Annotation. The JavaDoc comments of these annotation classes should give more detail for each specific annotation.

An example will help illuminate:

Here is the entity annotation for batch control:

```
/**
 * @BusinessEntity (tableName = CI_BATCH_CTRL,
     oneToManyCollections = { @Child (collectionName = parameters, childTableName = CI_BATCH_CTRL_P,
                 orderByColumnNames = { "SEQ_NUM"})})
*/
public class BatchControl_Impl
```

The name of the annotation is BusinessEntity. It has specified properties tableName, and oneToManyCollections (there are others available, but they need not all be specified). The property tableName specifies the CI_BATCH_CTRL table as the table that this entity maintains. It also contains some oneToMany child collections, specified by the list of Child annotations. In this case, there is a single child, with a collection name of parameters, pointing to the child table CI_BATCH_CTRL_P, with a native order given by the column name SEQ_NUM.

Once an annotation exists, the annotation wizard (in the Eclipse editors plugin) can be used to maintain the annotation, showing all of the available annotation properties, and with some validation of the values entered. Thus, one way to create an annotation from scratch is to create a purely empty annotation with the correct name at the start of the class, and then use the annotation editor to fill in the details, and assure against typographical errors and not have to hunt down the allowed properties.

Here is a list of top-level annotations and their corresponding purpose or managed class type, and a pointer to an example class in the FW code where available.

BatchJobAnnotation for batch jobs, defining such properties as whether the batch is multithreaded and what soft parameters it uses. An example batch job in Java is defined in the class com.splwg.base.domain.todo.batch .BatchErrorToDoCreation.

BusinessComponentAnnotation for business components. This will register the business component either as a new one (and define whether it can be replaced or not), or a replacement of an existing one. An example business component is com.splwg.base.domain.todo.toDoEntry.ToDoEntryAssigner_Impl.

AlgorithmComponentAnnotation for defining algorithm implementations. This is used to create a new algorithm implementation, defining which algorithm spot it is for, and what soft parameters it uses. An example algorithm component is com.splwg.base.domain.common.characteristicType.AdhocNumericValidationAlgComp_Impl.

EntityChangeAuditorAnnotation for implementing audit behavior when an entity is modified. An example auditing component is com.splwg.base.domain.common.audit.DefaultTableAuditor_Impl.

BusinessEntityAnnotation for defining or extending business entities, with properties defining the table maintained and any one-to-many child tables, etc. An example entity is com.splwg.base.domain.batch.batchControl.BatchControl_Impl.

ChangeHandlerAnnotation for extending entity persistence behavior- adding validations, or adding extra code to execute on add/change/delete actions. An example change handler is com.splwg.base.domain.common.characteristicType.CharacteristicType_CHandler.

CodeDescriptionQueryAnnotation for adding services to handle drop down lists for the UI. There are no examples of this general component- the Oracle Utilities Application Framework implements only entity code descriptions.

EntityCodeDescriptionQueryAnnotation for adding services to handle drop down lists for the UI, that are directly related to entities. An example of an entity code description component is com.splwg.base.domain.common.country.CountryCodeDescriptionQuery.

MaintenanceExtensionAnnotation for extending a maintenance. There are no examples of maintenance extensions in the framework. It is purely an implementer component. Please see Maintenance Extensions (User Guide: Cookbook: , Hooking into User exits: Hooking into Maintenance Class User Exits).

QueryPageAnnotation for creating a new query page service. An example is com.splwg.base.domain.todo.toDoQueryByCriteria.ToDoQueryByCriteriaMaintenance.

PageMaintenanceAnnotation for creating a new generic page maintenance. An example is com.splwg.base.domain.security.user.SwitchUserLanguageMaintenance.

EntityListPageMaintenanceAnnotation for creating a new maintenance for an entity-type, with a list based front end. An example is com.splwg.base.domain.common.phoneType.PhoneTypeListMaintenance.

EntityPageMaintenanceAnnotation for creating a new entity maintenance, that maintains a single instance at a time. An example is com.splwg.base.domain.batch.batchControl.BatchControlMaintenance.

ListServiceAnnotation for creating a list service (read only), meant for trees for example. An example is com.splwg.base.domain.security.user.UserAccessGroupCountListInquiry.

Public API

SQL Return Codes

The framework generally returns the database-specific return codes from SQL execution. However, the framework returns SPL-specific return codes for commonly-used SQL execution result messages. These SPL-specific return codes are the same regardless of the database. This allows programs to be portable across different databases.

The following lists the SPL-specific return codes:

SQL Execution Result	SPL Return Code
OK	0
* Unnumbered SQL Error	999999990
Warning	999999991
End / no (more) row retrieved	999999992
Duplicate / unique index violation	999999993
More / multiple rows retrieved in single-row select	999999994
Deadlock	999999995
No connection	999999996
* Application Error	999999997
* Hibernate Error	999999998
* Programmatic Error	999999999

Note:
The SQL return codes marked with an asterisk ("*") are for errors peripheral to the actual execution of the SQL and do not have equivalent database return codes.

Standard Business Methods

In general, classes that are created to implement business logic, including change handlers, business entities, maintenance classes, and business components have access to standard methods intended to give application code access to framework functionality. Commonly, these classes extend the GenericBusinessObject class within their inheritance hierarchy. Below are some general descriptions of the provided methods. Please refer to the JavaDocs for more detail.

- createQuery(String)-Create an HQL query.
- createPreparedStatement(String)-Create a "raw" SQL statement. It is preferable to use the createQuery method.
- getActiveContextLanguage()/getActiveContextUser()-Get the language and user associated with the current request.
- createDTO(Class)-Create a new DataTransferObject instance for the entity corresponding to the provided business interface class.
- getDynamicComponent(various)-Get a Business Component instance corresponding to the input business interface for the component.
- getSystemDateTime()-Get the current DateTime instance appropriate for business logic.
- IsNull(Object)/notNull(Object)-Methods that answer the question if an object is null or is equivalent to null.
- isNullOrBlank(String)/notBlank(String)-Methods that answer the question if a String reference is null, zero length, or all blank.
- startChanges()/saveChanges()-Used to defer validation when making complex changes to entities. It may be the case where a valid entity can only be constructed by passing through one or more invalid states. By calling startChanges() at the beginning of the set of changes and saveChanges() at the end, some validations may be deferred until the entire coherent change is complete.

Business Entity Public Methods

BusinessEntity classes implement a combination of methods inherited from their generated superclasses as well as the framework classes that those generated superclasses extend. The generated methods are typically "convenience" methods based on the specific features of the entity. The framework methods are ones implemented by many or all entities. Similarly, some methods are expected to be invoked from other objects (public methods) and others are to facilitate business logic coded into the entities' business methods themselves.

Public Methods

These methods are exposed via the generated "business interface" of the entity.

- registerChange(Change) - Allows for another entity to register the fact that that entity has changed so that any dependant change handler logic in *this* entity may fire. This is most useful in situations where the changed object and the dependant object (the one needing to know about the change) are not directly related by parent-child relationships.
- getDTO() - Get a DataTransferObject representing the current state of the entity.
- setDTO(DataTransferObject) - Update the state of the entity based on the passed values in the DTO.
- getId() - Each entity has a method by this name with retrieves and Id instance of the appropriate class for the entity.
- getFoo() - Get the value of the persistent property "foo".
- fetchBar() - Convenience method that will fetch the value of "bar" where "bar" is a parent entity referenced by an optional foreign key refernce. The word "fetch" is used to denote that navigation to that entity is not provided from within HQL.
- getBazzes() - Get the EntityList containing members of the entity "baz". For example, a getPersonNames() method on the "person" entity might return an instance of an EntityList containing PersonName instances.

Protected Methods

These methods are exposed via the extended generated superclass of an entity (the "_Gen" class) for the use of business methods implemented on the entity. With few exceptions, the methods exposed as public methods on business entities are also exposed "within" the entity as protected methods for the convenience of business logic. Additionally, the following methods are added:

- thisEntity()-Returns the instance of the current entity. Generally, this is used when an entity needs to pass itself as an argument in a method call.
- addError(ServerMessage)-Add an error relating to the current entity.
- addError(ServerMessage, Property)-Add an error relating to the passed property on the current entity.
- addWarning(ServierMessage)-Add a warning to the current warning list.

Data Transfer Object Methods

DataTransferObjects (or DTOs) are transient objects meaning that changes to their state are not directly persisted. They provide a mechanism where the set of properties of an entity can be passed around in business logic without the implication that changes to their values will be transparently persisted to the database.

- getFoo()/setFoo(Bar)-Get or set the value of the property "foo".
- newEntity()-Create a new persistent entity based on the values currently held in the DTO.

Id Methods

Entities generally have an Id class created for them by the artifact generator. This provides clarity in the application code as to what "kind" of Id is being held or passed. Likewise, there are useful methods on these Id classes. Id instances are immutable.

- getEntity()-Get the business entity that this Id refers to or null if no such entity instance exists.
- getFoo()-In the case where the Id contains a persistent entity "foo", return that entity.
- getBarId()-Get the contained Id referring to the entity "bar".
- newDTO()-Create a new DTO instance with the Id property already set to this Id's value.

Maintenance Class Public Methods

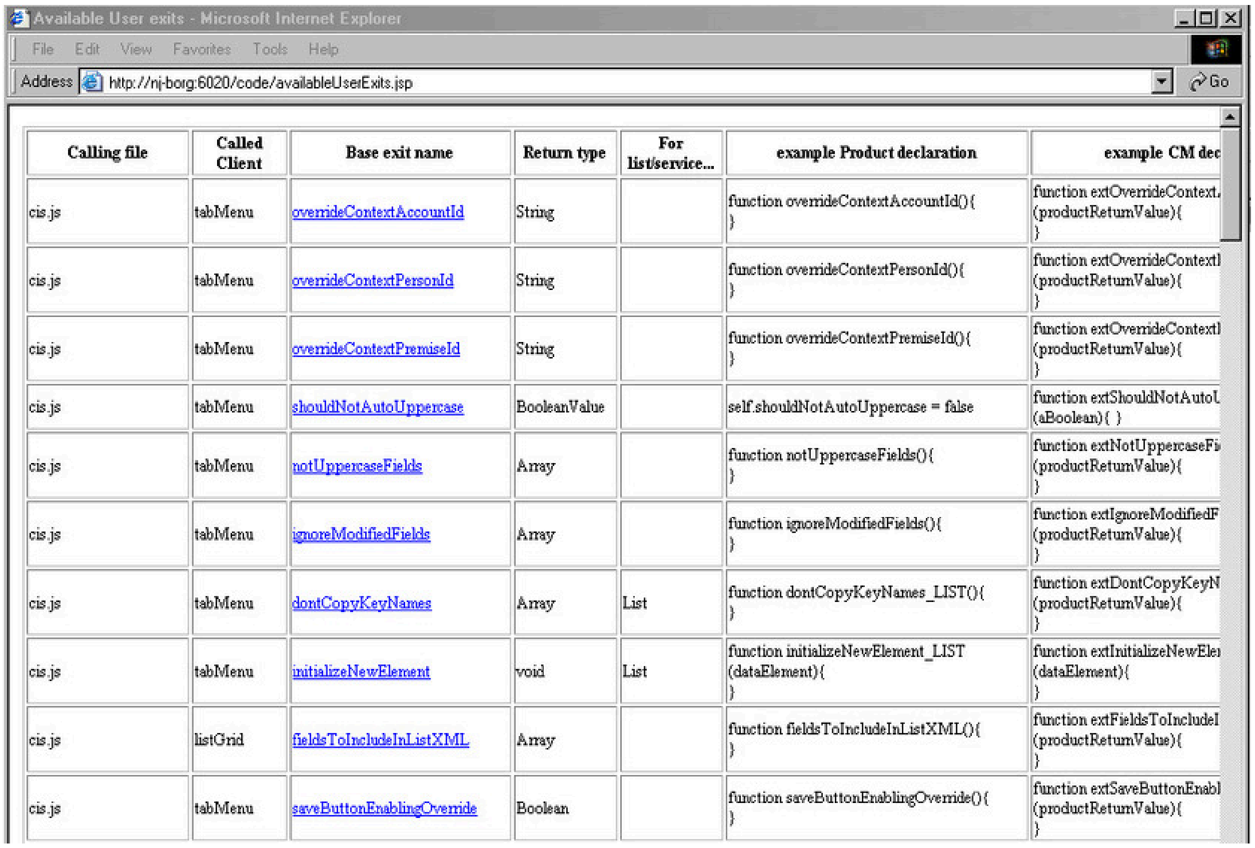
Please refer to the Javadocs for the public API.

UI Javascript User Exits

The client-side external user exits are designed to give implementers flexibility and power to extend the user interface of a OUAF products. Implementers have the ability to add additional business logic without changing OUAF product HTML files. These user exits were developed such that developers can create an include-like file based on external user exit templates.

There are two types of client user exits available. There are process-based user exits that wrap the similar product user exit code with pre- and post- external user exit calls, and there are also data-based user exits that simply allow the implementer to add/delete data from the product returned data.

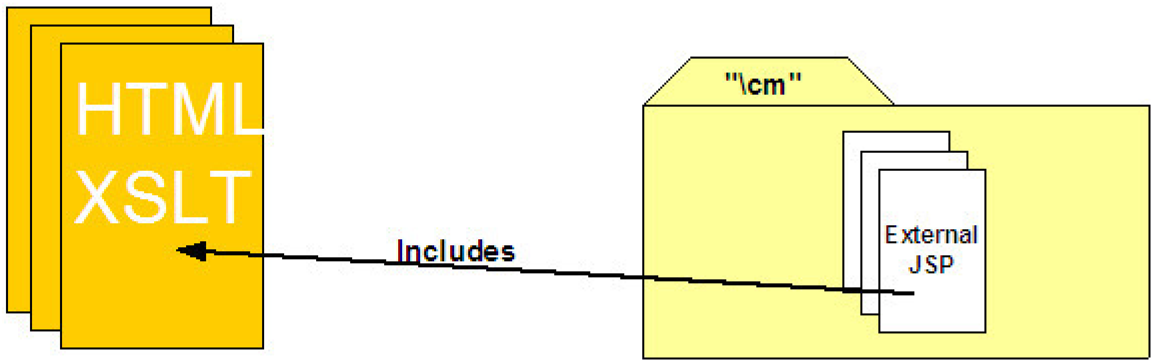
Both types of external user exit are only called if the function exists in the implementers external include JSP file. All available user exits are listed online in the system through the relative URL: /code/availableUserExits.jsp, with definition examples and links to the framework code that executes the call.


Available User exits - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://nj-borg:6020/code/availableUserExits.jsp
Go

Calling file	Called Client	Base exit name	Return type	For list/service...	example Product declaration	example CM dec
cis.js	tabMenu	overrideContextAccountId	String		function overrideContextAccountId(){ }	function extOverrideContext (productReturnValue){ }
cis.js	tabMenu	overrideContextPersonId	String		function overrideContextPersonId(){ }	function extOverrideContextI (productReturnValue){ }
cis.js	tabMenu	overrideContextPremiseId	String		function overrideContextPremiseId(){ }	function extOverrideContextI (productReturnValue){ }
cis.js	tabMenu	shouldNotAutoUppercase	BooleanValue		self.shouldNotAutoUppercase = false	function extShouldNotAutoU (aBoolean){ }
cis.js	tabMenu	notUppercaseFields	Array		function notUppercaseFields(){ }	function extNotUppercaseFi (productReturnValue){ }
cis.js	tabMenu	ignoreModifiedFields	Array		function ignoreModifiedFields(){ }	function extIgnoreModifiedF (productReturnValue){ }
cis.js	tabMenu	dontCopyKeyNames	Array	List	function dontCopyKeyNames_LIST(){ }	function extDontCopyKeyN (productReturnValue){ }
cis.js	tabMenu	initializeNewElement	void	List	function initializeNewElement_LIST (dataElement){ }	function extInitializeNewEle (dataElement){ }
cis.js	listGrid	fieldsToIncludeInListXML	Array		function fieldsToIncludeInListXML(){ }	function extFieldsToIncludeI (productReturnValue){ }
cis.js	tabMenu	saveButtonEnablingOverride	Boolean		function saveButtonEnablingOverride(){ }	function extSaveButtonEnabl (productReturnValue){ }

UI Available User Exits - Online Reference

The location of the external JSP file is the \cm directory under the web application root directory:

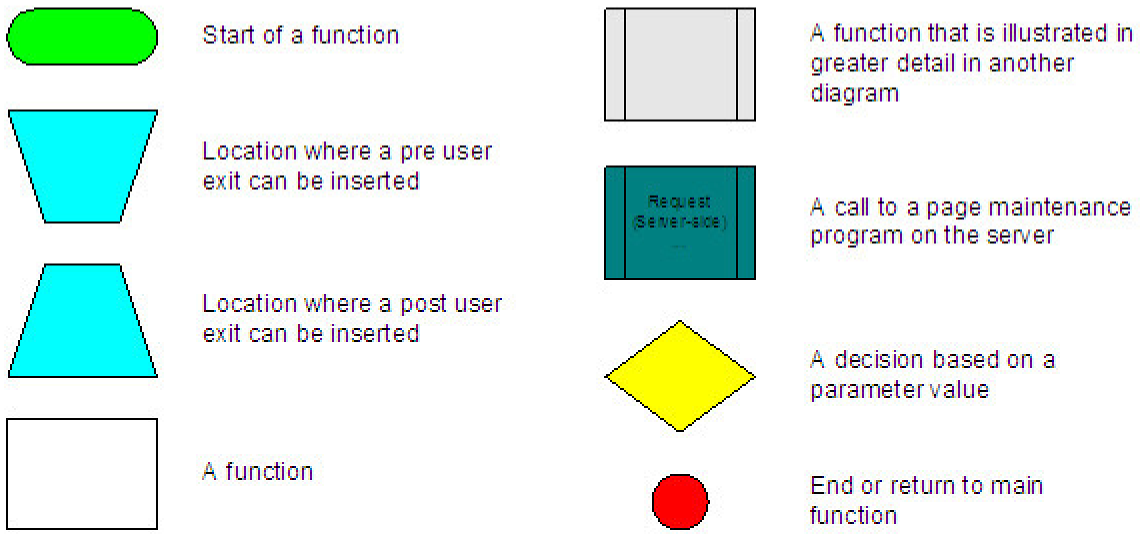


This document assumes that you are familiar with the framework architecture and its UI program component templates (XSLT) and that you know how the base exits work. It also assumes that you are proficient in JavaScript and HTML.

Client User Exit Flow

The following flowcharts illustrate the most common user exit functions used to modify the user interface. The flowcharts are designed to help you see the coordination of processing between the client and the server as well as where the pre and post external user exits can be used.

The following diagram explains the shapes used on subsequent flowcharts:

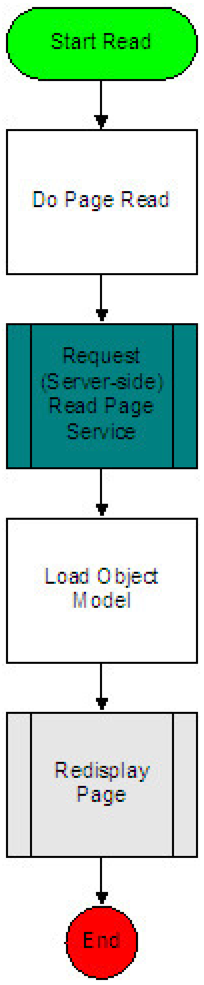

Start of a function
Location where a pre user exit can be inserted
Location where a post user exit can be inserted
A function
A function that is illustrated in greater detail in another diagram
Request (Server-side)
A call to a page maintenance program on the server
A decision based on a parameter value
End or return to main function

Flowchart Legend

Whenever you see a request for a server-side page service, you can refer to the Page Maintenance Program flowchart to see the server-side processing. You can determine the Page Action based on the service being requested: Read, Add, Change, Delete, or Default.

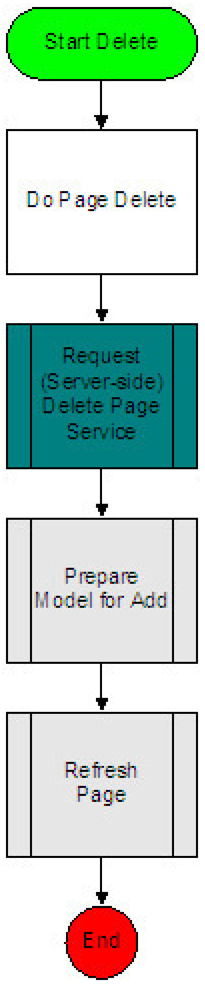
Read Page

The Read Page function is executed whenever data needs to be presented from the database to the user. It is called after a root item is selected from a search page or when navigating to another transaction via a Go To button or a Context menu.



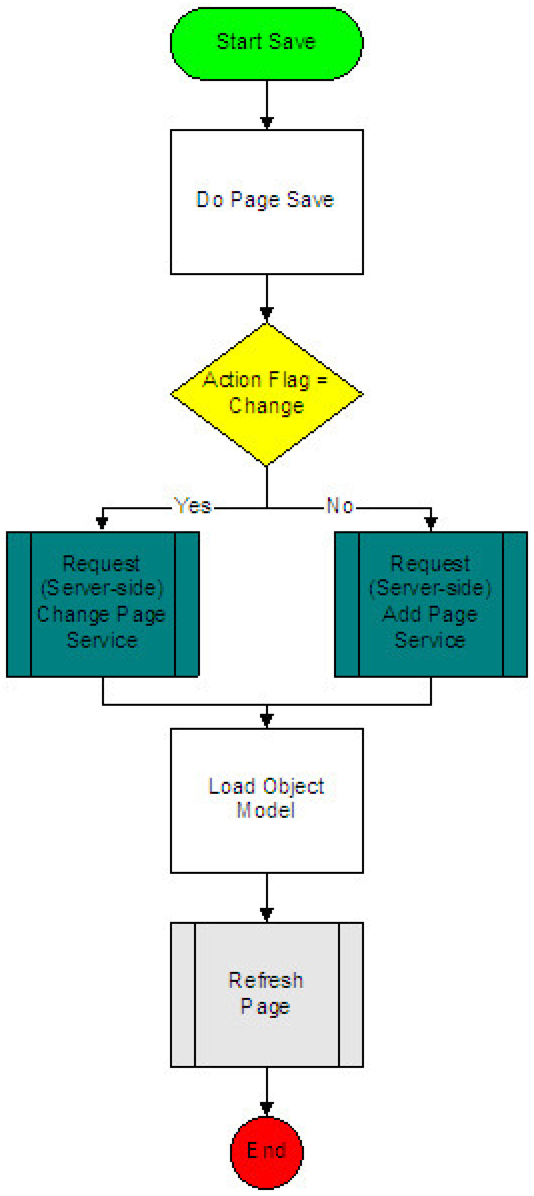
Delete Page

The Delete Page function is executed when the user clicks the Delete icon.



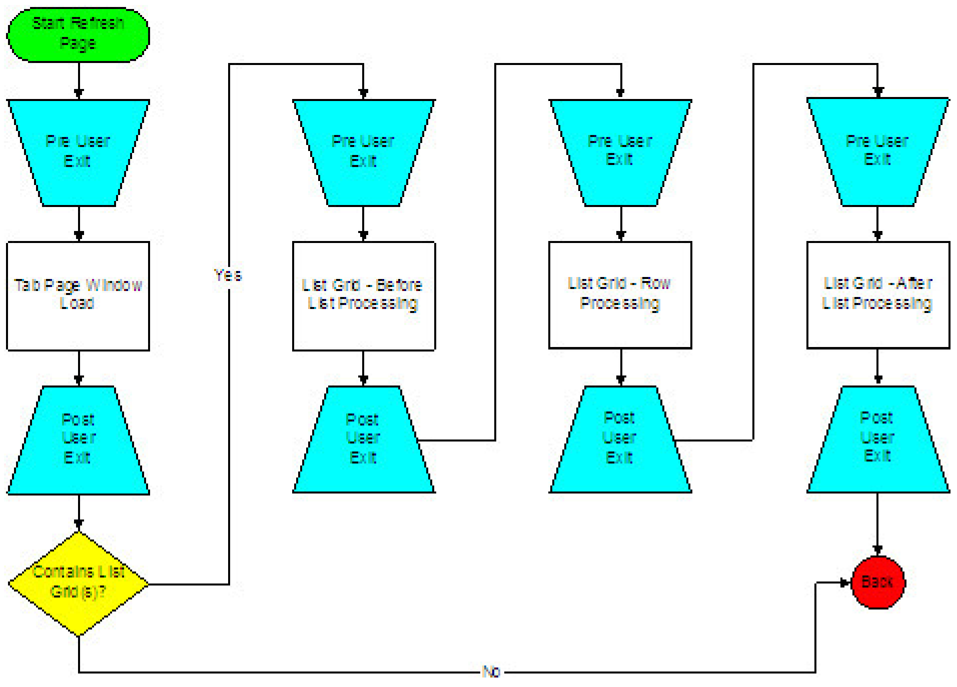
Save Page

The Save Page function is called whenever a user clicks the save icon (or the associated accelerator key). If the user has displayed an existing object on a maintenance page, the Action Flag and therefore the Change Page Service is requested. If an existing object is not displayed on a maintenance page and the user presses save (e.g., they are adding a new object to the database), the Action Flag does not equal change and therefore the Add Page Service is called.


Start Save
Do Page Save
Action Flag = Change
Yes
No
Request (Server-side) Change Page Service
Request (Server-side) Add Page Service
Load Object Model
Refresh Page
End

Refresh Page

The Refresh Page function is called from the Read, Delete, and Save page processes. It is also called when the user clicks the Refresh Page icon (or the related accelerator key) or when the user navigates to a different tab page.

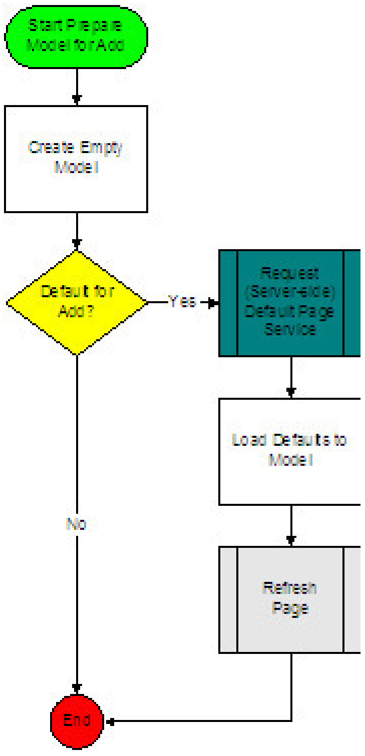


The pre/post Tab Page Window Load user exit is a good place to implement Field Level Security logic. By using the getFieldLevelSecurityInfo() function found on the "top" object (please refer to the Free Functions section found within the Technical Background chapter of the Development Tools documentation), an implementer can extend the behavior or look of the window. For example, a field can be made "read-only" if the user's Field Level security is lower than the required security level. This prevents the user from changing the value of the field.

You can use pre/post List Grid Row Processing exit to manipulate fields within the grid. For example, you can calculate the default value of a column depending on the values of other columns.

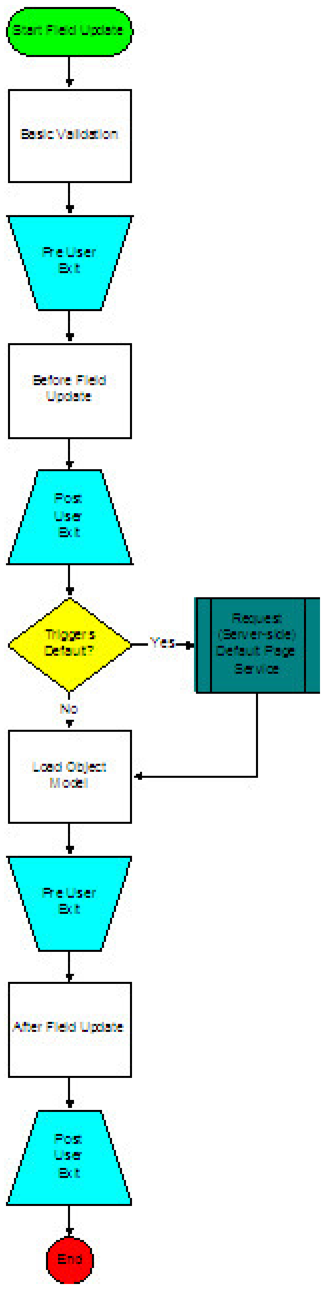
Prepare Model for Add

Prepare Model for Add is called when a user enters a page in Add mode (e.g., they click the + button next to a menu item). It is also called by Delete Page to load an empty model, which displays page with empty fields.



Update Field

The Update Field function is called when a user changes the focus from one field on the page to another (e.g., when a user tabs out of a field or clicks on another field).



The pre/post After Field Update user exit is a good place to manipulate HTML elements (e.g., hiding/unhiding or enabling/disabling) depending on the value entered by a user.

External User Exit Templates

Below is the list of all available external template files. They are located under the "\cm_templates" folder of the application root directory.

Note:
The flowcharts above illustrate user exits in the Tab Page and List Grid templates only; these are the templates in which most of your customizations will occur.

Accordion: accordionPage.jsp

Graph Panel: graphPanelExit.jsp

List Grid: listGridExit.jsp

Search Data: newSearchDataExit.jsp

Search Page: newSearchPageExit.jsp

Sub Panel: subPanelExit.jsp

Tab Page: tabPageExit.jsp

Tree Page: treePageExit.jsp

Template Structure

Each template has three main sections into which you insert your code.

- **User Variable Declaration** contains global variable declarations. (Do not declare any global variables unless it is absolutely necessary.)
- **User Function Declarations** contains your own functions. Your own functions are not called from the corresponding JSP file. Take coupling and cohesion into consideration when you design your functions.
- **Functions Called from the Corresponding HTML page** contains functions that are called from the HTML page. Uncomment the functions you need to use and add your code. You can find more technical information about the behavior of these functions in the external template files.

Design Approach

Examine the partial template below to see how the external include file looks. As you might notice everything is commented out. If you want to call a certain function, you have to uncomment the functions and/or sections. Please note that only declared functions in the external files can be called from the HTML Page. To see the entire external file templates or available functions, examine the "\cm_templates" folder under the application root directory.

```
<%@page contentType="text/html;charset=UTF-8"%>

<%@ taglib uri="spl.tld" prefix="spl" %>

<spl:initializeLocale/>


<!--

        ******************************************************************

        *                                                                *

        * Copyright (c) 2000, 2007, Oracle. All rights reserved.         *

        *                                                                *

        ******************************************************************

        *                                                                *

$#BSES* REVISION-INFO Start Exit, Do not modify - Dev. Only.

      *  $DateTime$

      *  $File$

      *  $Revision$

$#BSEE* REVISION-INFO End Exit, Do not modify - Dev. Only.

        ******************************************************************

-->


<script type="text/javascript">


//                            User Variable Declarations

//-----------------------------------------------------------------------------


/*

Replace With Your Code

*/


//                            User Function Declarations

//-----------------------------------------------------------------------------


/*

Replace With Your Code

*/


//             User Functions That Are CALLED From According JSP File
```

```
//---------------------------------------------------------------------------


/*

function extPreOnWindowLoadNoListBefore(){

//This should be used to set values/attributes when the page loads.

//This includes actions after a default.

//

//This function is called BEFORE SPL's internal functions are called


// Your Code

}

*/
```

The following discussion explains how the external file is included. The external file is a JSP file. This JSP is executed with appropriate HTTP request header data from within the XSLT engine that creates the HTML from the UI meta-data. The XSLT engine will output the rendered JSP code textually into the final HTML. If the file does not exist the server will not include the external file, otherwise every defined function (uncommented) in the file will be included and called at the appropriate times.

Using the External User Exit Templates

All the external user exit templates are located in the \cm_templates directory. Once the UI Program to be extended is known, the appropriate user exit template can be selected from the templates directory.

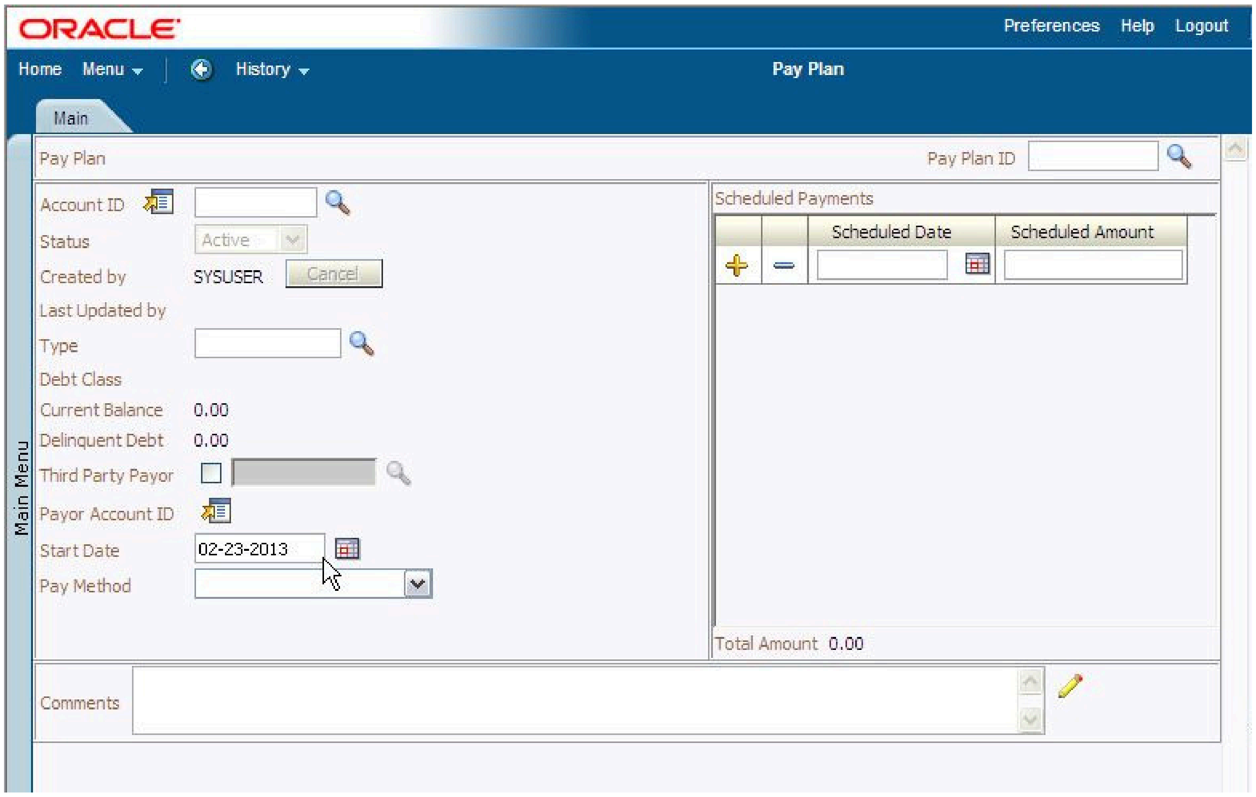
- Use any editor that supports the JSP file editing and open the approprite user exit.
- Determine the base user exit around which to insert your external user exit.
- Uncomment the necessary functions, and add your code.
- Save the external user exit file as ext_<JSPfilename>.jsp under the \cm directory. Where JSPfilename is the JSP file you want to extend.
- Test.

Create an External User Exit

The following example shows the process of creating an external user exit. In this example, we would like to disable the **Start Date** on the **Pay Plan** page and default it to "today's" date.

Find the Name of the JSP File

In Utilities **CC&B**, navigate to the **Pay Plan** Maintenance page (Main Menu -> Credit & Collection -> Pay Plan) and find the section where the **Start Date** field is displayed. From the screenshot below we can see that **Start Date** is under the main section of the page.

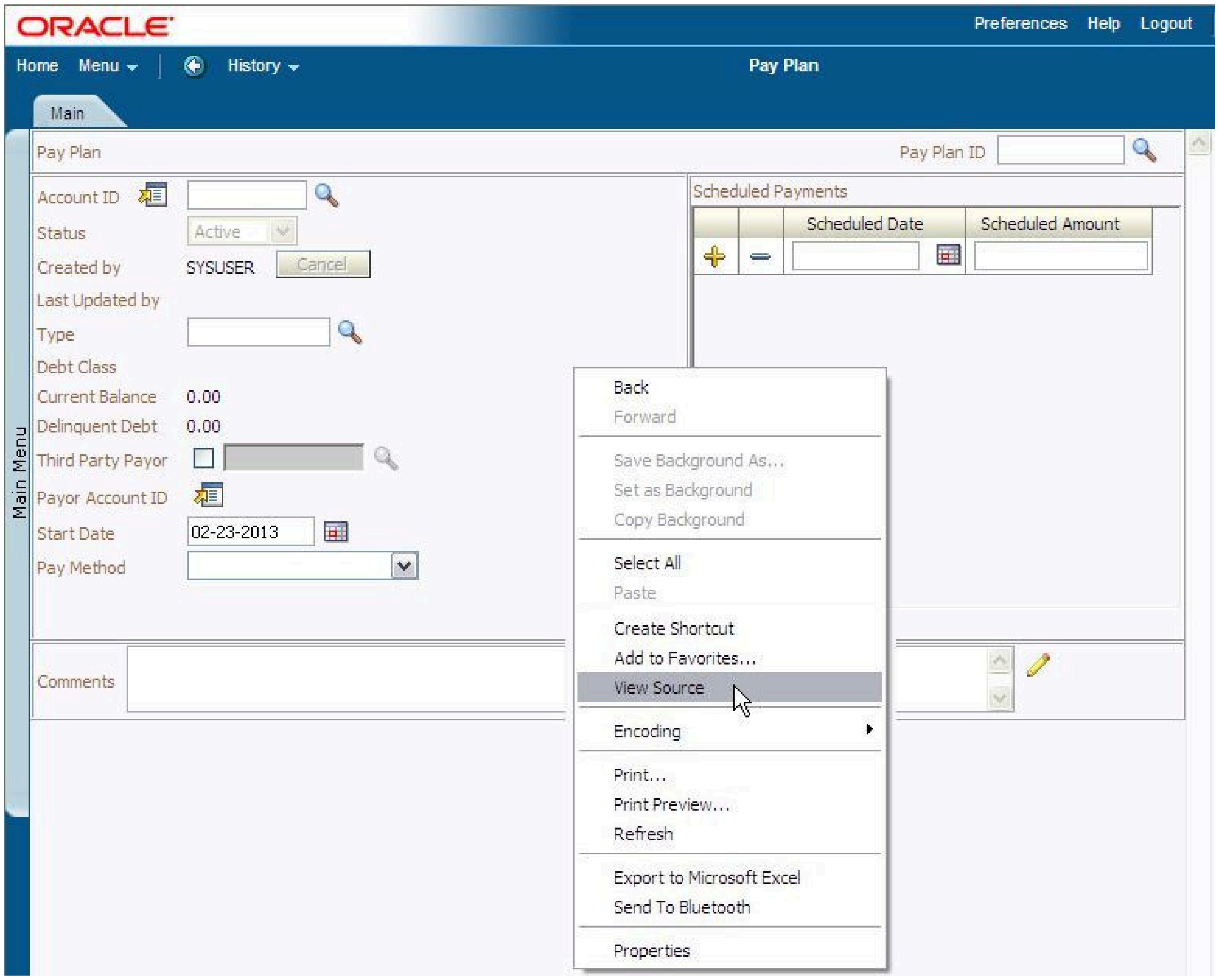


Account ID	
Status	Active
Created by	SYSUSER
Last Updated by	
Type	
Debt Class	
Current Balance	0.00
Delinquent Debt	0.00
Third Party Payor	
Payor Account ID	
Start Date	02-23-2013
Pay Method	

Total Amount 0.00

Pay Plan Maintenance Page

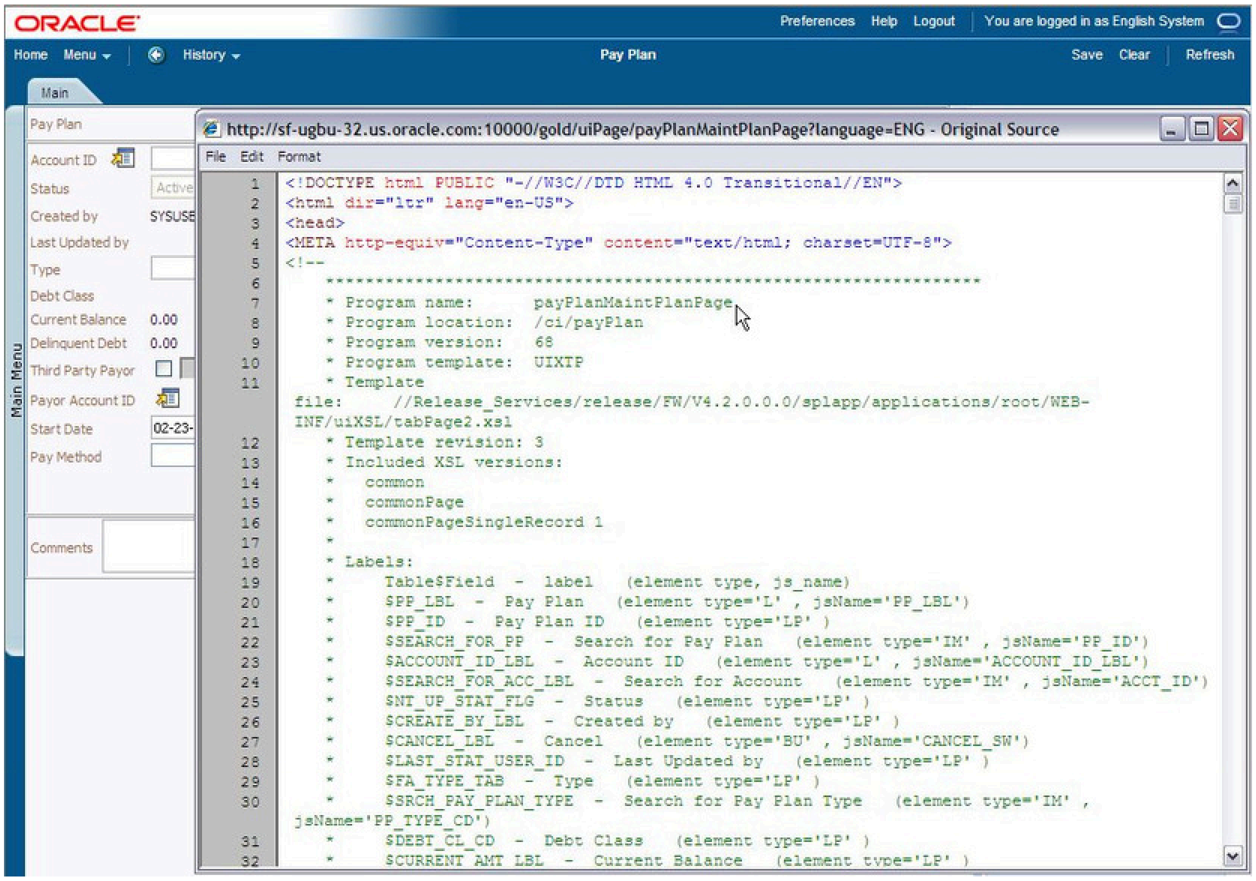
Determine the name of the program component we need to extend. Right click on the page and select the menu option **View Source**.



Account ID	
Status	Active
Created by	SYSUSER
Last Updated by	
Type	
Debt Class	
Current Balance	0.00
Delinquent Debt	0.00
Third Party Payor	
Payor Account ID	
Start Date	02-23-2013
Pay Method	

Pay Plan Maintenance Page - View Source

View Source would open the page source in a text editor.



```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html dir="ltr" lang="en-US">
<head>
<META http-equiv="Content-Type" content="text/html; charset=UTF-8">
<!--
    *******************************************************************
    * Program name:      payPlanMaintPlanPage
    * Program location:  /ci/payPlan
    * Program version:   68
    * Program template:  UIXTP
    * Template
 file:     //Release_Services/release/FW/V4.2.0.0.0/splapp/applications/root/WEB-
 INF/uiXSL/tabPage2.xsl
    * Template revision: 3
    * Included XSL versions:
    *   common
    *   commonPage
    *   commonPageSingleRecord 1
    *
    * Labels:
    *     Table$Field  -  label   (element type, js_name)
    *     $PP_LBL  -  Pay Plan   (element type='L' , jsName='PP_LBL')
    *     $PP_ID  -  Pay Plan ID   (element type='LP' )
    *     $SEARCH_FOR_PP  -  Search for Pay Plan   (element type='IM' , jsName='PP_ID')
    *     $ACCOUNT_ID_LBL  -  Account ID   (element type='L' , jsName='ACCOUNT_ID_LBL')
    *     $SEARCH_FOR_ACC_LBL  -  Search for Account   (element type='IM' , jsName='ACCT_ID')
    *     $NT_UP_STAT_FLG  -  Status   (element type='LP' )
    *     $CREATE_BY_LBL  -  Created by   (element type='LP' )
    *     $CANCEL_LBL  -  Cancel   (element type='BU' , jsName='CANCEL_SW')
    *     $LAST_STAT_USER_ID  -  Last Updated by   (element type='LP' )
    *     $FA_TYPE_TAB  -  Type   (element type='LP' )
    *     $SRCH_PAY_PLAN_TYPE  -  Search for Pay Plan Type   (element type='IM' ,
 jsName='PP_TYPE_CD')
    *     $DEBT_CL_CD  -  Debt Class   (element type='LP' )
    *     $CURRENT_AMT_LBL  -  Current Balance   (element type='LP' )
```

JSP Source Code View - payPlanMaintPlanPage

From the **menu bar** or the program file information section you can identify the program name as **payPlanMaintPlanPage** (look for the **Program name** in the source code comments).

Determine the Base User Exit

For this example, we want to disable the input element corresponding to the **start date** and display a message that the start date is disabled. This means we want to disable the field when the page loads; and therefore we want to insert our code inside the onWindowLoad() function. The external user exit function that allows us to do this is the extPostOnWindowLoad() function.

You can check the field names under the payPlanMaintPlanPage's Labels section.

```
* Program name:      payPlanMaintPlanPage

* Program location:  /ci/payPlan

* Program version:   68

* Program template:  UIXTP
```

```
    * Template file:     //FW/2.2.0/Code/modules/web/source/root/WEB-INF/uiXSL/tabPage2.xsl

    * Template revision: 4

    * Included XSL versions:

    *   common

    *   commonPage

    *   commonPageSingleRecord 3

    *

    * Labels:

    *     Table$Field  -  label   (element type, js_name)

    *     $PP_LBL  -  Pay Plan   (element type='L' , jsName='PP_LBL')

    *     ...

    * Widget Info:

    *     Widget_ID , Element Type - label info - label

    *     ...

    *     START_DT, IT - $START_DT -  Start Date


    *     PAY_METH_CD, IS - $PAY_METH_CD -  Pay Method

    *     ...
```

The two important pieces of information in this source view are:

The **Program name** definition - *payPlanMainPlanPage* in this example;

The **Template file** definition - *tabPage2.xsl* in this example.

Uncomment the Function and Add Code

Once the program name to be extended (e.g. *payPlanMainPlanPage*) and the template (e.g. *tabPage2.xsl*) to use are known, the associated template jsp file can be copied from the web application source's **/cm_templates** directory to the **/cm** directory and renamed to have the form *ext_XXXX.jsp*, where "*XXXX*" is the name of the program to be extended.

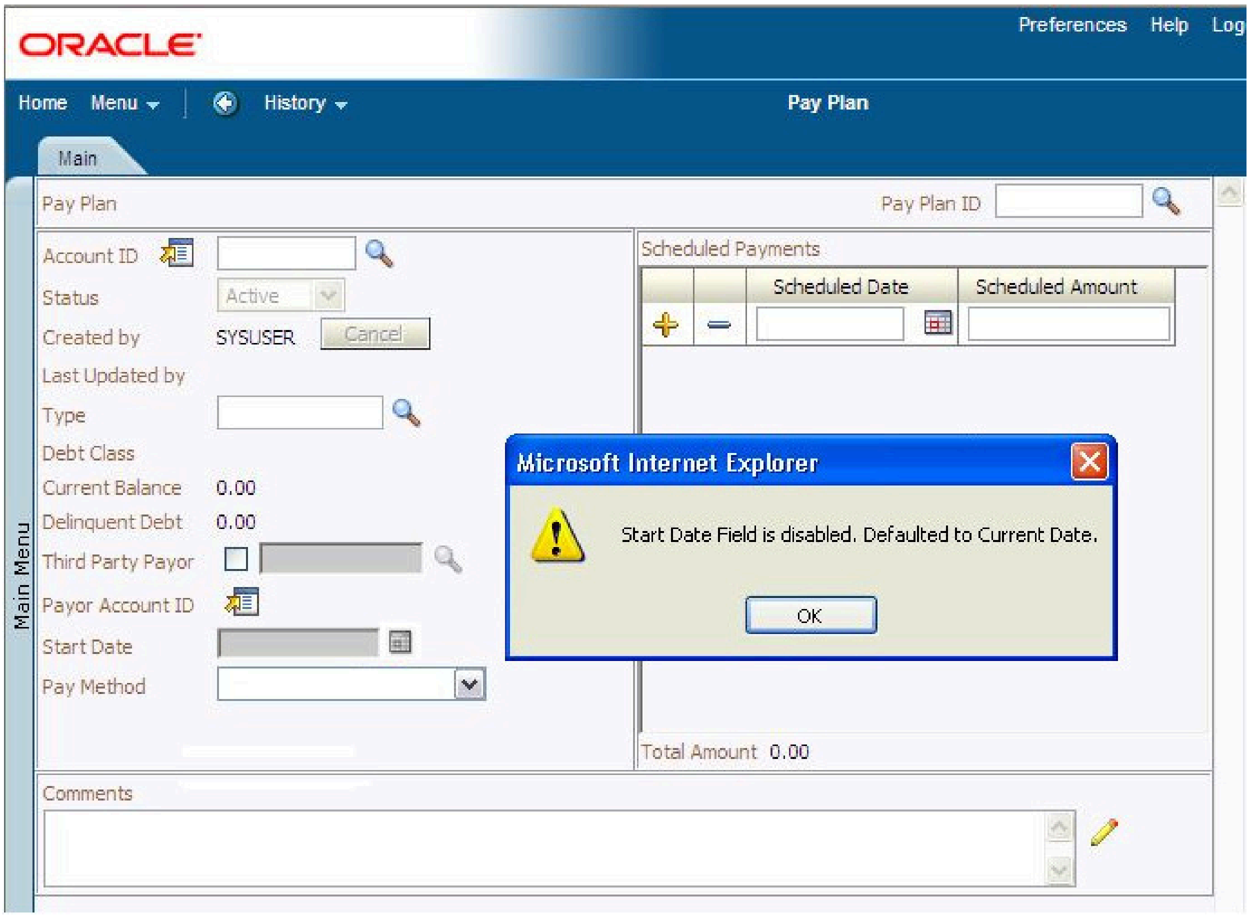
For example, in this case the jsp user exit template **./cm_examples/tabPageExit.jsp** would be copied and renamed to **./cm/ext_payPlanMaintPlanPage.jsp**. The following coding change inside the extPostOnWindowLoad() function would then be made for the modified behavior.

```

function extPostOnWindowLoad(){

//This should be used to set values/attributes when the page loads.
//This includes actions after a default.

//This function is called AFTER SPL's internal functions are called



protectField("START_DT");

alert ("Start Date Field is disabled. Defaulted to Current Date.");



}
```

Test Your Code

Now let's see if it works. First make sure that the user exit file is copied under the \cm folder of the application root directory. Reload the page by right-clicking the page and choosing Refresh from the context menu.

Note:
You may need to delete the browser cache before refreshing the page.



Status: Active

Created by: SYSUSER

Current Balance: 0.00

Delinquent Debt: 0.00

Total Amount: 0.00

Microsoft Internet Explorer

Start Date Field is disabled. Defaulted to Current Date.

Pay Plan Main Page after implementing External User Exit

Field-level Security Client-Side User Exit Example

Field level security information is exposed on the browser side.

Use the following function to retrieve a user's field level security for a given service or Navigation Key:

top.getFieldLevelSecurityInfo(serviceNameOrNavigationKey) ---> returns an Object keyed by security type.

The following example illustrates how to implement security for adjustment amount on the client. In the example, User Group 1 is authorized to freeze adjustments less than $10,000, and User Group 2 can authorize any adjustment. We want to disable the Freeze button, if the user's security doesn't meet the condition. There is a security type **ADJAMT** defined for the Adjustment Maintenance.

Fastpath:
Refer to *Field Level Security* in the Administration Guide, Defining General Options chapter for information about the data setup.

The following example code would be added to the extPreOnWindowLoad user exit:

```
var secInfo = top.getFieldLevelSecurityInfo("adjustmentMaint");

var adjAmtSecInfo = secInfo["ADJAMT"];

if (adjAmtSecInfo < "2" && parseFloat(model.getValue("ADJ_AMT")) > 10000 ) {

//disable the field

   protectField("FREEZE_SW")

}
```

How-To

The following are some how-to examples of typical behavior utilizing some of the standard user exits.

The examples are written for cases of modifying new CM transaction pages, where the function definitions are put into "extended JavaScipt" files (.xjs) that are meant to contain JavaScript user exits directly for a page.

If, on the other hand, an implementer wishes to modify the behavior of a shipped product page, each of the function below have a corresponding "ext" function that can defined in a /cm/extXXX.jsp file corresponding to the desired page that will fire after any product function call (see above example of hiding the Sequence column in the algorithm maintenance page).

How Do I Control the Initial Focus Within Tab Pages/Grids/Search Pages?

The system automatically places the initial focus on an appropriate widget (generally input fields) within a Tab Page/Search Page/Grid.

By default it will place focus on the first enabled field with a data class defined on it. (If input fields do not have the Field Name / Table Name defined within Meta Data they will have no data class)

If there are no fields satisfying this criteria within the tab page it will continue to look (recursively) into all the contained frames (e.g. list grids etc.)

If no field is found then no element receives focus.

You can override the default behavior at each level via the provision of a focusWidgetOverride() function within the user exit file which will return the Name of the Field to receive the focus or null.

If null is returned it will ignore all fields within this document and continue to search in lower level documents.

E.G.

From within a Tab Page (If you want focus to go on to a sub document)

```
function focusWidgetOverride() {

  return null;

}
```

From within a List Grid

```
function focusWidgetOverride() {

  return "TD_TYPE_DRLKY:0$TBL_NAME";

}
```

from within a Search Page

```
function focusWidgetOverride() {

return "LAST_NAME";

}
```

Note:
These functions can be as simple or complicated as you want. You could conditionally return a field name or null and this code will run each time the window loads. Also, if a tab page has a popup window or a search window open as it is loading then the initial focus will not be set to the tab page but stay with the popup window

How Do I Mark Fields that Won't Make the Model Dirty?

In certain windows, we have a concept of a "locator" field which typically acts as a filter on some lists of the object you're looking at. Examples are user group's filter on description, and several IB windows filter by date.

With the warning about loss of data when throwing away a dirty model, this results in the use of locator fields giving this warning, which wouldn't be expected. In order to avoid this warning on locator fields, you can add a function like the one that follows that enumerates the locator fields:

```
function ignoreModifiedFields(){

    return ['START_DTTM']

}


You can include any number of fields in the array, e.g.

return ['FIELD_1', 'FIELD_2', 'FIELD_3']
```

How Do I Control the Triggering of Defaults After a Search?

If a search returns multiple fields and more than one of these fields can trigger default, then it might be more efficient to only have one of these fields trigger the defaulting.

This is accomplished by creating a new function called **overrideDefaultTriggersFor_SEARCHGROUP** within the tab page that contains the search. Where **SEARCHGROUP** is the name of the searchGroup you want to override.

The function must return an object with the triggering field(s) are attributes with a **true** value.

For example

```
function overrideDefaultTriggersFor_SRCH1() {

var triggers = {};


triggers["ACCT_ID"] = true;

            triggers["SA_ID"]=true;


return triggers;

}
```

How Do I Avoid Automatically Setting Fields to Uppercase?

Model attributes that are also key fields are automatically coerced to be in uppercase. You can block this behavior on a field-by-field basis by defining the notUppercaseFields() function in your TabMenu's user exit file to return an array of field names that should not be converted.

Example:

```
function notUppercaseFields() {

   return ['ELEM_ATT$AT_NAME']

}
```

You can also provide a "global" override for an entire TabMenu by setting the shouldNotAutoUppercase variable to true:

```
var shouldNotAutoUppercase = true;
```

How Can I Force the Save Button to be Enabled?

The save button usually synchronizes itself to the state of the model such that if it hasn't been "dirtied" the button is disabled. You may wish to control the state of the save button e.g. because a save should always/never be allowed.

Simply define the function saveButtonEnablingOverride() on your TabMenu user exit file to return a boolean indicating whether the save button should be enabled. You can simply return a literal boolean, or perform any desired processing to determine the return value.

Example:

```
function saveButtonEnablingOverride() {

  return false;

}
```

How Can I Override the Processing After a Change/Add?

If you need to intervene in the processing after the system successfully completes a Change or Add operation, define the function privatePostChangeSucceeded() or privatePostAddSucceeded() in your TabMenu user exit file. The function should return a boolean to indicate whether the system should refresh the UI with the newly returned server data. You'd want to return false if e.g. you navigate to a different TabMenu.

Example :

```
function privatePostAddSucceeded() {

    var model = parent.model;

    var modeFlag = model.getValue('COMPL_NAV_MODE_FLG');

    var navKey = model.getValue('COMPL_NAV_KEY');

    var complSw = model.getValue('CMPLT_CLICKED_SW');

    if (complSw && model.getValue('ENRL_STATUS_FLG') == '30') {
        if (modeFlg && navKey){

             if (modeFlag == 'G') {

                parent.tabPage.gotoContext(navKey);

                return false;

             } else if(modeFlag == 'A') {

                parent.tabPage.addContext(navKey);

                return false;

             }

        }

   }

   return true;

}
```

How Do I Prevent the System from Setting Focus to a Widget After an Error?

When a service receives an error and shows a message after calling a back-end service, the browser attempts to set focus to the relevant widget in error. If you don't need this behavior, you can define the TabMenu variable dontSetFocusOnError to boolean "true.

Example:

```
var dontSetFocusOnError = true;
```

How Do I Prevent Attributes from Being Copied into New List Elements?

Key fields are automatically copied (based on name matching) from a parent list element into new child elements (e.g. created by using the scroll '+' button), in order to keep their prime keys consistent. If you want to inhibit this operation for certain fields, define the TabMenu function dontCopyKeyNames_<LIST NAME> to return an array of fields that should not be copied into new elements of the list named LIST_ NAME

Example:

```
function dontCopyKeyNames_ENRL_FLD() {

   return ['SEQ_NUM']

}
```

How Do I Customize New List Elements?

When you use '+' button on a grid or scroll you get a new, empty list element. If you want to customize the object, define a function in the TabMenu's user exit file named initializeNewElement_<LIST_NAME>(newElement).

Example:

```
function initializeNewElement_ENRL_LOG(newElement) {

    newElement.set('ENRL_LOG_TYPE_FLG', 'USER');

    newElement.set('USER_INFO', parent.model.getValue('CURRENT_USER_INFO'));

}
```

How Can I Get My Sequence Numbers to Default Properly on My List Grid?

If you are working with a List Grid that uses some type of sequence field (e.g. SEQNO, LINE_SEQ, SORT_SEQ) , there is a handy bit of technology that you can use that will cause the UI to do this job for you.

Just follow the steps below and you'll have the problem solved in no time. The sequence field will be populated in your "empty line" and any elements that are added from then on will have an appropriate value in the sequence field. If the user edits the sequence field at any point, the next element added to the list will incorporate the change without any problems.

Note:
The default Sequence Number functionality will default the next nearest tens value from the highest sequence. The defaulting will do nothing after the sequence reaches the highest number it can hold.

- In the user exit file of the **Tab Menu** - **not** the main Page or the List Grid - copy this JavaScript code:

```
function initializeNewElement_LIST_NAME(newElement) {

      var myListName = "LIST_NAME";

      var myListSeqName = "FIELD_NAME";

      var myListMaxSeq = 999;

      defaultSequenceNumber(myListName,myListSeqName,myListMaxSeq,newElement)

}
```

```
</SCRIPT>

<SCRIPT src="/zz/defaultSequenceNumber/defaultSequenceNumber.js"></SCRIPT>

<SCRIPT>
```

- For **LIST_NAME**, substitute your List Grid's list name. Be careful not to lose that underscore [_] just in front of **LIST_NAME** in the first line! Remember that JavaScript is case-sensitive and make sure that you use all UPPERCASE letters as shown here.
- For **FIELD_NAME**, substitute the name of your sequence field, whatever that might be in your List. Don't lose the quotes ["] ! Again, use all UPPERCASE letters.

How Do I Override the Tab Page Shown After an Error in a List (Grid/Scroll)?

When an error is received (e.g. after a Save) it attempts to set focus on the relevant widget, which might require flipping to a different tab page. If the error relates to a list (grid or scroll) the system might not choose the tab page you'd prefer. In that event you can control the tab page that should be opened by defining the TabMenu function overrideErrorTabPage_<LIST_NAME>().

Example:

```
function overrideErrorTabPage_BPA() {

    return 'bussProcessAssistantStepPage';

}
```

How Do I Disregard Unwanted Criteria from a Search Triggered by a Search Button?

When a search button (currently implemented as an IMG) is pushed, the system launches a search and "pulls" all applicable criteria values from the current model. It might be that certain criteria fields should be ignored in a particular case. You can define the function addIgnoreFieldsFor_<triggerFieldName>() on a tab or search page's user exit file to specify fields to ignore whenever the IMG button named triggerFieldName is pushed on that page.

The function takes a single argument, fields, and you should add key/value pairs where the key is a field name to ignore, and the value is true.

Example:

```
addIgnoreFieldsFor_ADDRESS1_SRCH = function(fields) {

  fields['CITY_SRCH'] = true

}


addIgnoreFieldsFor_PER_ID = function(fields) {

  fields['ENTITY_NAME_SRCH'] = true

}
```

How Do I Disregard Unwanted Search Result Columns?

When you accept the result of a NOLOAD search the system tries to populate the selected search result row into the current model. Sometimes this doesn't make sense e.g. because there is no corresponding attribute for a display-only column. You can exclude a column from being returned as part of a search result by defining the search client's (Tab Page or Search window) function ignoreResultColumns() in the corresponding page's user exit file. Return an object with keys specifying attributes and values all set to boolean "true".

Example:

```
function ignoreResultColumns() {

    return { ADDRESS1: true, CITY: true, POSTAL: true };

}
```

Since Searches can be shared by many search clients, it is possible that some clients want to get a specific column, but others don't. In that case, define the TabMenu function ignoreResultColumnsFor_<service> as above.

Example:

```
function ignoreResultColumnsFor_CILCCOPS() {

    return {CONT_OPT_TYPE_CD: true}

}
```

How Do I Format a Value Based on a Given Format?

If you need to format a value based on a given format, for example, on Person ID Number, if you select ID Type as SSN (999-99-9999), you can always format the Person ID Number before committing it to the server.

To do so, you can call the formatValue javascript function.

- In the user exit file of the tab page include the following lines:

```
</SCRIPT>

<SCRIPT src="/zz/formatValue/formatValue.js"></SCRIPT>

<SCRIPT>
```

- Now, you can start using the function to format a value. To use this function, you need to pass in both the **value** and the **format** into the function.

```
var phFormat = myData.getValue(pureListName + 'PHONE_TYPE_FORMAT');

if (pureFieldName == 'PHONE') {

  updateField(pureListName + 'PHONE' ,

        formatValue(myData.getValue(pureListName + 'PHONE'), phFormat));

}
```

Java User Exits (interceptors) Interfaces and Classes

The following are the interfaces used for Java User Exits (interceptors).

IAddInterceptor Interface

This interface defines the processing plug-in spots before or after invoking a service in add mode.

Interface com.splwg.base.api.serviceinterception.IAddInterceptor

Methods

PageBody aboutToAdd(RequestContext, PageBody)

This method is called before the service is invoked.

Input

- RequestContext - contains session parameters, such as, language cd, user id and etc.

Input/Output

- PageBody - The input page body to be added.

Return value

- PageBody or null -- If a page body is returned, this is considered the result of the service and the underlying service will not be executed. If null is returned, the service will run normally.

Throws

InterceptorError - throw this exception when an error occurs.

InterceptorWarning - throws this exception to signal an application warning

void afterAdd(RequestContext, PageBody)

This method is called after the service invoked in add mode.

Input

- RequestContext - contains session parameters, such as, language cd, user id and etc.

Input/Output

- PageBody - This contains the information that was added by the underlying service.

Return value

- Void.

Throws

InterceptorError - throw this exception when an error occurs

InterceptorWarning - throws this exception to signal an application warning

IChangeInterceptor Interface

This interface defines the processing plug-in spots before or after invoking a page service in change mode.

Interface com.splwg.base.api.serviceinterception.IChangeInterceptor

Methods

PageBody aboutToChange(RequestContext, PageBody)

This method is called before the service invoked in change mode.

Input

- RequestContext - contains session parameters, such as, language cd, user id and etc.

Input/Output

- PageBody - this object contains the information that is to be submitted to the underlying service.

Return value

PageBody or null - if a page body is returned, this is considered the result of the invocation and the underlying service will not be called. If null is returned, the underlying service will be invoked normally.

Throws

InteceptorError - throw this exception when an error occurs.

InterceptorWarning - throws this exception to signal an application warning

void afterChange(RequestContext, PageBody)

This method is called after change action is invoked in change mode.

Input

- RequestContext - contains session parameters, such as, language cd, user id and etc.

Input/Output

- PageBody - This holds the result of the underlying change service.

Return value

- Void.

Throws

InterceptorError - throw this exception when an error occurs.

InterceptorWarning - throws this exception to signal an application warning

IDeleteInterceptor Interface

This interface defines the processing plug-in spots before or after invoking a service in delete mode.

Interface com.splwg.base.api.serviceinterception.IDeleteInterceptor

Methods

boolean aboutToDelete(RequestContext, PageBody)

This method is called before the service with a delete action.

Input

- RequestContext - contains session parameters, such as, language cd, user id and etc.

Input/Output

- PageBody - the data to be deleted.

Return value

- Boolean - indicates whether or not to continue processing of the service. If true, continue with the normal underlying invocation. If false, do not continue (but the service returns "success" to the client invoker).

Throws

InterceptorError - throw this exception when an error occurs.

InterceptorWarning - throws this exception to signal an application warning

void afterDelete(RequestContext, PageBody)

This method is called after the service invoked in delete mode.

Input

- RequestContext - contains session parameters, such as, language cd, user id and etc.

Input/Output

- PageBody - the data that was deleted by the underlying service

Return value

- Void.

Throws

InterceptorError - throw this exception when an error occurs.

InterceptorWarning - throws this exception to signal an application warning

IReadInterceptor Interface

This interface defines the processing plug-in spots before or after a service retrieves information.

Interface com.splwg.base.api.serviceinterception.IReadInterceptor

Methods

PageBody aboutToRead(RequestContext, PageHeader)

This method is called before a service retrieves information.

Input

- RequestContext - contains session parameters, such as, language cd, user id and etc.

Input/Output

- PageHeader - The data describing the information that should be read.
- Return value

  PageBody or null - If a page body is returned, this is considered the result of the service and underlying will not be invoked. If null is returned, the underlying service will be invoked normally.

Throws

InterceptorError - throw this exception when an error occurs.

InterceptorWarning - throws this exception to signal an application warning

void afterRead(RequestContext, PageBody)

This method is called after the service retrieved the information.

Input

- RequestContext - contains session parameters, such as, language cd, user id and etc.

Input/Output

- PageBody - result of read service

Return value

- Void.

Throws

InterceptorError - throw this exception when an error occurs.

InterceptorWarning - throws this exception to signal an application warning

InterceptorError class

The class com.splwg.base.api.serviceinterception.InterceptorError subclasses the java.lang.Exception class. This class contains information regarding an error condition that occurred during the pre/post processing plug-in. This exception is caught by the framework and is used to build the appropriate application error object.

Attributes

- Message Category
- Message Number
- List of Parameters (Strings) and types

Methods

void setMessageNumber(BigInteger messageNumber)

Set the message number (required)

void setMessageCategory(BigInteger messageCategory)

Set the message category (required)

void setMessageParameters(List messageParameters)

Set the message parameters list

void setMessageParameterTypeFlags(List messageParameterTypeFlags)

Set the message parameter type flags list. The size should match the message parameters list.

InterceptorWarning class

The class com.splwg.base.api.serviceinterception.InterceptorWarning subclasses the java.lang.Exception class. This class contains information regarding one or more warning conditions that occurred during the pre/post processing plug-in. This exception is caught by the framework and is used to build the appropriate application warning object(s).

Attributes

- List of warning server messages

Constructors

InterceptorWarning(ServerMessage warningMessage)

Create a new InterceptorWarning with the given warning message as its sole message

InterceptorWarning(List warningMessages)

Create a new InterceptorWarning with the given List of warning messages

Methods

void addWarningMessage(ServerMessage message)

Add the given server message to the list of warning messages

RequestContext Methods

Class com.splwg.base.api.service.RequestContext includes the following accessor methods:

String getLanguageCode()

Returns the current user's language code

String getUserId()

Return the user id

Data Objects

Both PageHeader and PageBody are "wrappers" on underlying Maps that hold datatypes of various types, keyed by field names (Strings). The valid field names for a service are described in the service meta info file (an xml document). Null values are not allowed; use empty strings to represent missing values (e.g. for null date).

Note that most system datatypes are represented in these Java objects as simple Strings. Note the following:

- Booleans are represented by the Java Boolean class
- Date values are represented as Strings in the format YYYY-MM-DD
- Date/Time values are represented as Strings in the format YYYY-MM-DD-HH:MM:SS
- Time values are represented as Strings in the format HH:MM:SS
- BigInteger values are represented as Java BigInteger values
- BigDecimal and Money values are represented as Java BigDecimal values, with the appropriate scale.

PageHeader and PageBody Methods

Both PageHeader and PageBody implement the methods described in the following topics.

Object get(String fieldName)

Returns the Object value of the field named fieldName (may need to cast the result to the appropriate datatype)

String getString(String fieldName)

Convenience method that returns the String value of the field named fieldName.

boolean getBoolean(String fieldName)

Convenience method that returns the Boolean value of the field named fieldName.

BigInteger getBigInteger(String fieldName)

Convenience method that returns the BigInteger value of the field name fieldName.

void put(String fieldName, Object value)

Set the value at the given fieldName to the given value.

PageHeader

The methods for class com.splwg.base.api.service.PageHeader are described above.

PageBody

Class com.splwg.base.api.service.PageBody implements the methods described above. In addition, it supports the following methods:

ItemList getList(String name)

Return the ItemList with the given name

ItemList

Class com.splwg.base.api.service.ItemList is the Java representation of a list header and list children objects. The methods are as follows:

ListHeader getHeader()

Return the list header object.

String getName()

Return the ItemList's name

List getList()

Return the java.util.List of ListBody child objects.

void setList(List list)

Set the underlying list to the provided list of ListBody instances.

ListHeader

The class com.splwg.base.api.service.ListHeader is functionally equivalent to the class PageHeader, above.

ListBody

The class com.splwg.base.api.service.ListBody is functionally identical to the class PageBody, above. In addition, it has this useful method:

String getActionFlag()

Return the flag describing the pending action for this ListBody (e.g. add, change, delete).

CMServiceConfig.xml structure

The ServiceConfig.xml and CMServiceConfig.xml will look similar to the following:

```
<ServiceInterceptors>

    <Service name="CMLPXXXX">

        <Interceptor action="add">

            com.splwg.interceptor.CMLPXXXXAddInterceptor

        </Interceptor>

        <Interceptor action="change">

            com.splwg.interceptor.CMLPXXXXChangeInterceptor

        </Interceptor>

    </Service>

</ServiceInterceptors>
```

The above example illustrates how interceptors are defined for the service CMLPXXXX. You can define one or more interceptors, depending on the action, for each service. The valid actions are "add", "change", "delete", and "read".

Note:
It is valid to have the same interceptor class for more than one action as long as the class implements the corresponding interceptor interface.

Application Logs

Logging has many purposes. Notably, it allows tracing of what is happening when something goes wrong. However, a user/developer does not always want to see EVERY log entry-besides clutter, it may slow down the application. In this light, the framework has wrapped the powerful and flexible log4j logging framework as an API. There are two important aspects:

- Placing logging statements within application code so that logging entries may be created at runtime.
- Configuring logging at runtime so that the appropriate logging entries are created and directed to the appropriate log destination.

Logging within Business Logic

The following describes how to implement logging when adding a class that implements business logic:

- Add a constant referencing the logger for the class. By convention logger should be named "logger" and should pass the declaring class as its argument. For example, a logger in the Adjustment_CHandler class would be declared as follows:

```
private static final Logger logger

= LoggerFactory.getLogger(Adjustment_CHandler.class);
```

- Add entries with the appropriate logging level. The levels are: "debug", "info", "warn", "error" and "fatal". The following will log a warning entry to the log:

```
logger.warn("Unexpected status for frozen adjustment: " + status);
```

- In general, we expect entries of level "info" or more severe to be rare and therefore not to impose a substantial performance penalty. However, "debug" entries we can expect to be very fine grained and that they usually will not find their way to actual logs but will be filtered out via runtime configuration. To lessen the performance impact of debug logging, the logging statement should be wrapped as follows:

```
if (logger.isDebugEnabled()) {

   logger.debug("Processing adjustment " + adjustment.getId());

}
```

- There are times when you want to know how long code block takes to execute. In general, the logging provides the time each log statement is issued. However, it is clearer to see an actual elapsed time of some process being investigated. In this case, there are some additional methods on the logger:

```
debugStart(message) or infoStart(message)

debugTime(message, start) or infoTime(message, start)
```

  - These should be used in the pairs given, as follows:

```
long start = debugStart("Starting process");

//... code for process

debugTime("End process", start);
```

  - This will cause each statement to be logged, plus the final "End Process" statement will give the elapsed time since debugStart was called.

Please refer to the JavaDocs on the com.splwg.shared.logging.Logger class for more detail.

Configuring Logging at Runtime

Having instrumented the code to create logging entries, the question remains, how to cause the various logger level messages to actually trigger at runtime? A very detailed description of this can be found at http://logging.apache.org/log4j/docs/manual.html.

Property Configuration

Control of log4j occurs based on properties typically set in the log4j.properties file in the application classpath. You can change the log level of a given logger in this file. Note, however, that values may be overridden on the command line by specifying system properties (e.g. via "-Dlog4j..."). Note that "inheritance" of logger levels works such that (in our standard of qualified class name as the logger name) you can change a whole package's log level by specifying only a portion of the logger name. Note that you may commonly desire to enable ("global") debug logging on your local environment. To do this, you can simply change the line

```
log4j.logger.com.splwg=info
```

to

```
log4j.logger.com.splwg=debug
```

Trace Flags

Trace flags allow for specialized logging that cuts across many classes. They can be set for user requests by entering the online system in "debug" mode and setting the "trace" flags appropriately. Likewise, they can be set in batch either by interactive prompts for the trace flag values when a job starts or by setting system property values. See the JobSubmitterConfiguration class for specific system property names.

- **traceSQL** - Causes special detail of the submitted SQL. This can be useful when troubleshooting performance problems.
- **traceTime** - This can only be enabled for online requests or JUnit tests by setting traceTime(true) on the request context. Enabling time tracing will cause special profiling entries to be placed in the application log for the purpose of attributing request latency to the various layers of the application or to specific SQL statements. These entries are queued in memory until after profiling entries are no longer being generated and then spooled to the logs so as not to corrupt the performance instrumentation with logging overhead. The ProfilingReport standalone Java program can be run to post-process these logs, or a portion of them and generate a report.

Java Programming Standards

Rationale

In order to make it easier for programmers working on the same codebase to easily read each other's (and their own!) code, we need to enforce certain standard coding conventions. These conventions will also be helpful when comparing code revisions under version control, as the code should be formatted consistently and no irrelevant formatting-related differences will appear in the diff.

Guidelines

First, there are basic code standards documented for Java here: https://www.oracle.com/technetwork/java/codeconventions-150003.pdf. Like most coding guidelines, these are quite reasonable and differ only in minor details from other guidelines.

The web page http://geosoft.no/development/javastyle.html also has some very nice tips. Note that we won't prefix instance variable names with underscores--instead we use Eclipse syntax coloring to make ivars easily visible.

We use the prefix *fetch* in method names in entity implementation classes, in order to perform object navigations that aren't already defined by Hibernate mappings.

Here are some additional notes:

Not surprisingly, a lot can be learned from good Smalltalk style. The books "Smalltalk With Style" (Klimas, Skublics, Thomas) and "Smalltalk Best Practices Patterns" (Kent Beck) provide a lot of good ideas for code organization and naming that are applicable to Java as well as Smalltalk.

All code should be:

Written with tabs equal to 4 spaces, not "hard" tabs. Each level of indentation should be one "tab".

Generally free of hard-coded "magic" strings or numbers (e.g. max number of items in some list). If you need such a string or number value, you should use (or create) a constant or property.

Classes should use specific, not package-based imports, where practical. I.e. import com.foo.UsefulClass, not com.foo.*.

Variables should generally be private. Only create accessor (e.g. get/set) methods when absolutely needed ("Dont reveal your private parts").

Prefix "getter" methods with "get", e.g. "getFoo()", setters with "set", e.g. "setBar(aBar)". Don't use "Flag" or "Switch", or abbreviations thereof, e.g. "getAllowedSw()" should be "getIsAllowed(), and "setAllowedSw(aBoolean)" should be "setIsAllowed(aBool)".

Use camel-case instance and parameter variable names, without underscore prefixes or suffixes (do use uppercase for constants, as suggested in the guidelines reference above). Instance variables start with lower-case letters.

Methods should generally be public or private (again, to allow future subclassing). Use of interfaces is encouraged to declare useful sets of public methods.

Don't abbreviate except for standard industry abbreviations (e.g. HTML, HTTP). Use long, meaningful class, method, and variable names.

Methods should be short and clear. Instead of placing comments before a section of code in a method, rather create another method that describes what is being done by the method name.

When using Java API collections, reference them through generic interfaces, not specific implementation classes, e.g.

```
List someList = new ArrayList();

...

Map someMap = new HashMap();

...
```

This lets you change your mind about implementation (e.g. ArrayList to LinkedList) without breaking any code.

Naming Standards

General guidelines

Don't use reserved java words

Don't use spaces

Don't abbreviate

Don't use punctuation

Don't start the name with a number

Here are our project guidelines for naming properties:

Generally, don't abbreviate. The exceptions are SA, SP when the name would get too long if written as e.g. *ServiceAgreement* as part of a much longer field name

In line with the above, spell out **amount** and **total**

Boolean values (SW) are prefixed with **is**, **has**, **can**, **are**, or **should**, according to what is grammatically correct.

Date fields end with **Date**

Time fields end with **Time**

Datetime fields end with **DateTime**

Id is spelled **Id**

Don't include a final *Flag* (FLG) or *Code* (CD)

Use **min** instead of minimum, and **max** instead of **maximum**

Can be generic- that is, for the field BILL_STATUS, you can just name it **status**

Entity Naming Guidelines

Be specific- the name MUST be unique

Language tables (_L) don't need to be named

Don't append "View" to a view.

Don't abbreviate

Don't use plural names (e.g. BillMessages)

Collection Naming Guidelines

Class Name

The class name for a collection includes the owning entity name and the collection name in singular form.

<owning_entity><collection_name_in_singular_form>

Examples:

AdjustmentTypeAlgorithm

AdjustmentTypeCharacteristic

BillableChargeTemplateLine

Collection Name

For collections, the one-off generation created a large number of collection names. Many of these are overly verbose, and should be shortened. Simply modify the collectionName in the entity annotation. Here are guidelines:

Shorten **adjustmentTypeAlgorithms** to **algorithms**

Shorten **adjustmentTypeCharacteristics** to **characteristics** (in rare cases you may have more than one kind of characteristic, in which case you need more specific names)

Remove the owning entity name from the front of the collection name, e.g. **billableChargeTemplateLines** becomes **lines**

Lookup Naming Guidelines

Here are guidelines for naming Lookups (on the Lookup Field maintenance):

Be specific- the name MUST be unique across all lookups

Don't include a final standard suffix *Flag* or *Lookup* (The suffix Lookup is automatically added by the generator to the classes generated for each Lookup field.)

Examples:

WO_STATUS_FLG -> writeOffStatus

STM_RTG_METH_FLG -> statementRoutingMethod

Here are guidelines for naming Lookups Value properties (on Lookup Value maintenance):

Try to word the name in a way that makes sense when prepended by *is*, and is also valid when standing alone as a constant. (eg {isComplete, COMPLETE}, {isFrozen, FROZEN})

The name might match the english description of the lookup value.

Examples:

HOW_TO_USE_FLG : - -> subtractive

ITEM_STATUS_FLG : A -> active

DGRP_PRIO_FLG : `10` -> highest10

DGRP_PRIO_FLG : `20` -> priority20

Special Cases

'Type' Entity Controlling Characteristics for 'Instance' Entities - Characteristic Controls

There are 'type' entities that control the characteristics for their 'instance' entities. These are tables typically named **CI_CHTY_<type_entity>**, e.g., CI_CHTY_CCTY. These type entities specify a list for its instances the valid characteristic types, default characteristic types, required characteristic types, etc. This list is the type entity's **Characteristic Controls**.

The following are the naming conventions for the characteristic controls:

Characteristic control class	**<type_entity>CharacteristicControl**
Characteristic control collection	**characteristicControls**

For example, the class name for characteristic control of Customer Contact Type is **CustomerContactTypeCharacteristicControl**.

And the collection is defined as follows:

```
/**
  * @version $Revision: #1 $
  * @BusinessEntity (tableName = CI_CC_TYPE,
      oneToManyCollections = { @Child (collectionName = characteristicControls, childTableName = CI_CHTY_CCTY)})
*/
```

HQL Programming Standards

The applications use an object relational mapping library called Hibernate (information available at http://www.hibernate.org/). This library handles persistence operations against the database for changed entities, and also provides a querying language.

The Hibernate Query Language (http://www.hibernate.org/hib_docs/reference/en/html/queryhql.html) provides a more object oriented approach to querying against the database. Joins can more clearly be indicated via "navigation" to the related foreign key, letting hibernate fill in the join when it constructs the SQL.

Note that in most situations only a subset of the hibernate query language is used. For instance, when constructing a query whose order is important, the query must programmatically specify the order by, as opposed to placing the order by clause into the HQL itself. This allows the application to perform additional operations upon the HQL that may be required for different databases, and also to apply validations to the HQL.

Here are some examples of creating and using queries. The convenience methods to create the query are available on any "context managed object"- that is, entities, change handlers, business components, maintenance classes, and the implementer extensions of any of them.

To select all algorithms with a given algorithm type:

```
AlgorithmType algorithmType = ... ;

Query query = createQuery("from Algorithm algorithm where " +

"algorithm.algorithmType = :algorithmType");

query.bindEntity("algorithmType", algorithmType);

List algorithms = query.list();
```

The above algorithms list will contain as elements the algorithms for that algorithm type.

To sort the above query by the algorithm's code/id:

```
AlgorithmType algorithmType = ... ;

Query query = createQuery("from Algorithm algorithm where " +

"algorithm.algorithmType = :algorithmType");

query.bindEntity("algorithmType", algorithmType);

query.addResult("algorithm", "algorithm");

query.addResult("algorithmId", "algorithm.id");

query.orderBy("algorithmId", Query.ASCENDING);

List queryResults = query.list();
```

The above queryResults list will contain as elements instances of the interface QueryResultRow. Each query result row will have two values, keyed by "algorithm" and "algorithmId". The list will be ordered (on the database) ascending by the algorithm's IDs.

Since HQL works with the entity's properties instead of the tables' column names, there may be extra research required when writing queries. The source of the property information is in the hibernate mapping document for each entity class- they are documents that exist in the same package as the entity, have the same root file name as the entity's interface, and end with .hbm.xml. These files will give the list of properties available for each entity that can be referenced when writing HQL.

More information can be found in the JavaDocs associated with the Query interface.

Examples

Even with all of the above, there are a few cases that stand out with possibly needing examples in order to help. Notably, dealing with language entries and lookups may be confusing.

Here is an example of selecting all algorithm types where the description is like some input:

```
String likeDescription = ...;
Query query = createQuery("from AlgorithmType_Language algTypeLang join algTypeLang.id.parent algType where algTypeLang
.description like :likeDescription and algTypeLang.id.language = :language");
query.bindEntity("language", getActiveContextLanguage());
query.addResult("algType", "algType);
query.bindLikableStringProperty("likeDescription", AlgorithmType.properties.languageDescription, likeDescription);
List algorithmTypes = query.list();
```

The algorithmTypes list will contain as elements the algorithm types whose description is like likeDescription. Note that the string likeDescription will have a trailing '%' appended when it is bound to the query.

Here is an example of selecting particular lookup values, with descriptions like an input value:

```
        String description = header.getString(STRUCTURE.HEADER.DESCR);
        Query query = createQuery("from LookupValue_Language lookupValLang "
                 + "where upper(lookupValLang.description) like upper(:description) and lookupValLang.id.language = :la
nguage and "
                 + "lookupValLang.id.parent.id.fieldName = 'RPT_OPT_FLG');
        query.bindLikableStringProperty("description", LookupValue.properties.languageDescription, description);
        query.bindEntity("language", getActiveContextLanguage());
        query.addResult("lookupValue", "lookupValLang.id.parent");
        query.addResult("description", "lookupValLang.description");
        query.orderBy("description");
        List results = query.list();
```

The list results will contain QueryResultRows, with values keyed by "lookupValue" and "description".

Union queries

You may note that hibernate's HQL does not allow unions, as this does not reconcile with the object oriented approach of HQL. However, as this can be a common technique to apply, a programmatic union has been provided in the Oracle Utilities Application Framework. The application will actually open two cursors and flip back and forth between rows from each cursor when each would be the next one, based upon the order by clause. This should at most read one extra row from each cursor opened than may be needed (in the case of limited maximum rows).

In order to union two queries, they must have identical result columns, order by clauses, and max rows setting. Note that some of the properties of the union query be modified directly, leaving the individual queries to omit those properties.

Creating a union query is simple. Given two queries that need to be unioned together, simply issue:

UnionQuery union = query.unionWith(query2);

If a third (or later) query needs to be unioned, add it to the union directly:

union.addQueryToUnion(query3);

Performance

In order to evaluate the performance of HQL queries, it is necessary to first run the HQL through the hibernate engine at run-time in order to produce the equivalent SQL. First, code the initial HQL into the application or a unit test or standalone executable program. Start the application or test program with SQL tracing turned on. When the HQL under construction executes, grab the SQL from the log/console. Then follow the directions in ??? 07 SQL Programming Standards to check the performance of the SQL.

In general, most of the advice under the SQL programming standards applies equally for coding HQL when applicable at all.

Raw SQL

In rare cases, it may be necessary to forgo the use of HQL and instead use raw SQL. This is not a preferred approach, as the data returned will not be Java entities, but columns of primitive data types.

However, for possible performance reasons (no db hints are allowed in HQL) or if a table is not mapped into a Java entity, this approach exists.

There are parallel methods available on subclasses of GenericBusinessObject that create PreparedStatements, instead of Query objects. So, instead of createQuery, the method createPreparedStatement should be called on a Raw SQL statement.

The PreparedStatement is similar to the regular jdbc PreparedStatement, but has some extra functionality, and a slightly different interface so that it is similar to the regular HQL Query interface (they are interchangeable in some cases).

The main difference is that the prepared statement is created with raw SQL. Use the actual table and column names instead of the Java entity names and property names. Also, the select clause must exist as in normal SQL but not HQL.

Additionally, this break-out into raw SQL allows SQL statements that update table data. Again, this is normally frowned upon, and instead should be done by entity manipulation. However, in cases where a set-based SQL could update many rows at once, this option is available, whereas HQL is ONLY meant for querying without any updates.

For more help on constructing raw SQL queries please see SQL Programming Standards *(on page 232)*.

SQL Programming Standards

This document describes the SQL programming standards to be used in any database query. These standards will ensure that all database queries across the system have been structured properly and thus have less chance to cause performance issues. All developers must adhere to these standards.

Composing SQL Statements

Prerequisite

This document assumes that you have a basic knowledge of SQL syntax and database functions.

Composing a SELECT Statement

General SELECT Statement Considerations

- Before composing an SQL statement, you should have in front of you the ERD of the tables involved in that SQL. You should make sure you fully understand the relationships between the tables.
- As you may know, an SQL may return a single record or a set of records as its result set. When a set is to be returned, it is managed by a cursor that loops through that set and issues a separate database call for each record in the set.
- Therefore, when you design your SQL, think carefully if the task can be easily achieved in a single SQL or rather that the nature of task is such that a row-by-row processing would make more sense. Examples for the latter could be a list processing or simply because the calculation per row is too complicated to be handled by the database.

Selection List

- If a list of fields is to be returned, specify them prefixed by their table's alias name as specified in the From Clause.
- Use the DISTINCT option when the result list of records may contain duplicate rows in respect to the specified list of fields AND only one copy of the duplicated rows is needed.
- For top-level batch programs, always specify the WITH HOLD keyword on the main SQL of a cursor based processing. This is to keep the cursor open after a commit or rollback. Without this, main cursor will be closed and fetch of the next record or restart processing will fail (specific to DB2) with SQL error 501.

Database-specific Features

Oracle

- Oracle7 and later provides new approach for optimization: cost-based optimization (CBO). CBO evaluates the cost to, or impact on, your system of the execution path for each specific query and then select the lowest-cost path. The CBO was designed to save you the trouble of fiddling with your queries. Occasionally, it is not giving you the results you want and you have exhausted all other possible problem areas, you can specify hints to direct the CBO as it evaluates a query and creates an execution plan. If you have used hints before, you know that a hint starts with /*+ and ends with */. A hint applies only to the statement in which it resides; nested statements consider as separate statement and require their own hints. Furthermore, a hint currently has a 255-character limit. Since the use of hint is database-specific, we should make use of Database Functions to accomplish it.
- The most effective hints for use with the CBO are:

  - *FULL* - tells the optimizer to perform a full table scan on the table specified in the hint
  - SELECT /*+FULL(table_name)*/ COLUMN1,COLUMN2.....
  - *INDEX* - tells the optimizer to use one or more indexes for executing a given query.
  - Note: If you just want to ensure the optimizer doesn't perform a table scan, use INDEX hint without specifying an index name and the optimizer will use the most restrictive index. A specific index should not be used as the actual index name may differ on the client's site.
  - SELECT /*+INDEX(table_name index_name1 indexname2...) */
  - COLUMN1, COLUM2
  - *ORDERED* - tells the optimizer to access tables in particular order, based on the order in the query's FROM clause (often referred to as the driving order for a query)
  - SELECT /*+ORDERED*/ COLUMN1, COLUMN2
  - FROM TABLE1, TABLE2
  - *ALL_ROWS* - tells the optimizer to choose the fastest path for retrieving all the rows of a query, at the cost of retrieving a single row more slowly.
  - SELECT /*+ALL_ROWS*/ COLUMN1, COLUMN2...
  - *FIRST_ROWS* - tells the optimizer to choose the approach that returns the first row as quickly as possible.
  - Note: the optimizer will ignore the first rows hint in DELETE and UPDATE statements and in SELECT statements that contain any of the following: set operators, group by clauses, for update clause, group functions, and the distinct operators.
  - SELECT /*+FIRST_ROWS*/ COLUMN1, COLUMN2...
  - *USE_NL* - tells the optimizer to use nested loops by using the tables listed in the hint as the inner (non-driving) table of the nested loop. Note: if you use an alias for a table in the statement, the alias name, not the table name, must appear in the hint, or the hint will be ignored.
  - SELECT /*+USE_NL(tableA table B) */ COLUMN1, COLUMN2...
- Hints are an Oracle specific feature and are not supported by the DB2 SQL syntax.
- If you need to add a hint to your SQL make sure that a different SQL version is used for DB2 where the hint is not used.
- Base product developers should not duplicate their SQL in this case but rather use the special database functions file "dbregex.txt". In this file you should add a new hint-code that in Oracle translates into the specific hint whereas in DB2 it translates into an empty string.

FROM Clause

- Any table that has least one of its fields specified in the Selection List and/or any table that is directly referred to in the Where Clause (excluding sub-selects if any) must be specified in this section.
- Label each table with a meaningful short alias and use this alias to reference the table anywhere in the SQL.

WHERE Clause

General WHERE Clause Considerations

- All tables specified in the From Clause must participate in a join statement with another table. Table left not joined, would cause a Cartesian join to be applied for this table and any other table on the list, resulting in an incorrect result list let alone very poor performance.
- Note that there is no such thing as "conditional" join where the only join statement for a table is under a condition. In cases where the condition is not met and thus the join is not performed, one would end up with the same problem described previously.
- The final result set is built up by taking the full population of the tables involved and applying the restricting criteria to it one after another where the intermediate result population of one step is the input for the next step. Therefore, it is recommended to specify the most restrictive criteria first so that at the end of one step, lesser records are processed in the next step.
- This is of course a very schematic and simplified way to describe the internal process. This is not necessarily how the database is actually processing the statements. However, setting up the criteria as described would direct the database to use the right path.

Use of Sub-Selects

- When you need to further test each processed record in the Where clause for meeting an additional condition, AND that condition can NOT be checked directly on the Where clause level, you probably need a sub-select.
- As it is performed once for each outer level record it is considered as quite an expensive tool. Therefore if the criteria checked in a sub-select can be moved to the outer where clause level, it is preferable. If you still need to use a sub-select, it is very important to restrict the outer where clause population to the very minimum possible so that lesser records would need to be further checked for the sub-select condition.
- When no value needs to be returned from the sub-select query but rather simply use it to check if a certain condition is true or false, use the EXISTS function as follows:

    - Select ...
    - From ...
    - Where ... AND EXISTS (<sub-select>)
- A sub-select query may refer to any value of the outer level record as its input parameters. Notice that if your sub-select does NOT refer to any of the processed record fields, it means that the result set of the sub-select would be the same for ALL the processed records.
- Note that this could, but not necessarily, be an indication that your sub-select is set up wrong. One case where it is definitely wrong is when the sub-select result is input to an EXISTS function.

Use of in Function

- Whenever a field needs to be tested against a list of valid values it is recommended to use the IN function and not compare the field against each and every value.
- Wrong way:
    - Select ...
    - From ...
    - Where ... (A = '10' or A='20' or A='30')
- Right way:
    - Select ...
    - From ...
    - Where ... A IN ('10','20','30')

Use of Database Functions

- Not all database functions available for one database are valid for others. Make sure that when you do use a database function the SQL works properly on every database supported by the product.
- Avoid using LIKE as this will cause table scans. To achieve the 'LIKE' function where the first part of the string is specified, e.g., "CM%", BETWEEN may be used with the input criteria padded with high and low values.

Other

- Depending on the data distribution, search on optional index column will likely to cause time out. See example -
- Select BSEG_ID
- From CI_BSEG
- Where MASTER_BSEG_ID = &IN.MASTER-BSEG-ID
- For such cases, consider additional restrictions or re-create the index to become composite - MASTER_BSEG_ID + BSEG_ID.

Sort Order

- When a result list should be displayed in a specific order, sorting should take place on the database level and NOT on the client. This is especially important in cases when the list cannot be returned in full but rather in batches of records. Sorting each batch of records separately would not guarantee the sort order between records of different batches.
- Columns in the sort order list must be specified in the selection list.
- Prefix each field used in this clause with its table's alias name.
- Explicitly specify whether sorting should be ascending or descending and do not rely on database defaults.

Grouping

- When a set of records needs to be grouped together by a simple and straightforward condition, it is recommended to use the database Group By Clause. In this case only the final summarized records are to be returned to the client resulting in a lesser number of database calls as opposed to processing the full list let alone a simpler program without any special grouping logic.

Existence Checks

- The common technique used to check whether a certain condition is met or not, obviously when no data needs to be returned, is simply COUNT how many records match that condition. A zero number indicates that no record has met that condition.
- Notice that this is not very efficient as we are asking the database to scan the records for an accurate number that we don't really care about. All we really want to know if there is at least one such record and NOT how many they are.
- When the tables involved are of low volume there should be no problem using this technique. It is very simple and uses common SQL syntax to all databases.
- However, when that condition is checked against a high volume table that many of its records meet that condition, scanning all the matching records to get a count we don't need should be avoided.
- In this case use the EXISTS function as follows:
- Select 'x'
- From <The main table of the searched field, where it is defined as the PK of that table>
- Where <search field> = <search value> and
- EXISTS
- (<sub-select with the desired condition. This is the high volume table>);
- For example :
- Select 'x'

- From CI_UOM
- Where UOM_CD = input UOM_CD and
- EXISTS (select 'x'
- From CI_BSEG_CALC_LN
- Where UOM_CD = input UOM_CD);
- If this does not work for your special case, use the following option :
- Select 'x'
- From CI_INSTALLATION
- Where EXISTS
- (<sub-select with the desired condition>) ;
- Remember : This type of existence check using the Installation Options record should only be used in rare cases and should be consulted with the DBA first before implementation.
- Note that we use CI_INSTALLATION as this table has only one row.

SQL statements to avoid

Decimal Delimiter

In Europe the decimal delimiter is often set to be comma character. DB2 database configured this way will return SQL syntax error in the following cases:

- select ....,1,
- insert ....values(...1,2,3...)
- insert ....values(...1 ,2,...)
- order by 1,2,3
- order by 1 ,2
- update...set abc=1,def='XX'
- case (? as varchar(50),12

To avoid this problem, surround the comma with spaces.

Testing SQL Statements

Result Data

Once your SQL is ready, it is essential to test that it actually returns the expected result.

Create sample data for each condition checked by your SQL. Then execute the SQL and make sure it returns the expected result for each case.

Performance Testing - Oracle Only

Overview

An SQL may perform reasonably well even if not efficiently written in cases where the volume of processed data is low, like in a development environment. However, the same SQL may perform very poorly when executed in a real high volume environment. Therefore, any SQL should be carefully checked to make sure it would provide reasonable performance at execution time.

Obviously there could be many reasons for an SQL to perform poorly and not all of them are easy to predict or track.

In general, these could be subcategorized into two main groups:

- Basic issues related to the SQL code. These may be missing JOIN statements, inefficient path to the desired data, inefficient use of database functions, etc.
- More complicated issues having to do with lack of indexes, database tuning and handling of high volume of data, efficiency of I/O system etc.

The latter group of issues may only be truly tested on a designated environment simulating a real production configuration. These performance tests are typically conducted by a team of database and operating system experts as part of a thorough performance testing of a predefined set of process.

It is the first group of issues that can and should be tested by the programmer at this stage. This is done by analysis of the SQL's **Explain Plan** result.

What is an Explain Plan?

An explain plan is a representation of the access path that is taken when an SQL is executed within Oracle.

The optimal access path for an SQL is determined by the database optimizer component. With the **Rule Based Optimizer** (RBO) it uses a set of heuristics to determine access path. With the **Cost Based Optimizer** (CBO) we use statistics to analyze the relative costs of accessing objects.

Since the Cost Based optimizer relies on actual data volume statistics to determine the access path, to generate an accurate Explain Plan using the cost based optimizer requires a database set up with the proper statistics of a real high volume data environment.

Note:
A cost based optimizer Explain Plan generated on an inadequate database, would be totally inaccurate and misleading!

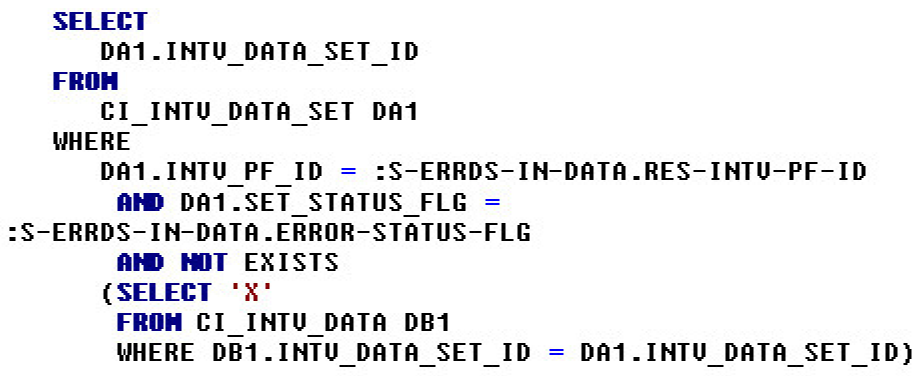
Obviously, our development database does not qualify as an optimal environment of cost based optimizations. Since the Rule Based optimizer is not data dependant it would provide a more reliable Explain Plan for this database.

Note:
An efficient rule based Explain Plan does not guarantee an efficient cost based one when the SQL is finally executed on the real target database. However, a poor rule based Explain Plan would most probably remain such on a database with a higher volume of data.

Note:
When the SQL is complicated and mainly designed to process high volume tables it is recommended to also analyze its Explain Plan on an appropriate high volume database.

Generate the SQL's Explain Plan

- Let's assume this is the SQL to be checked

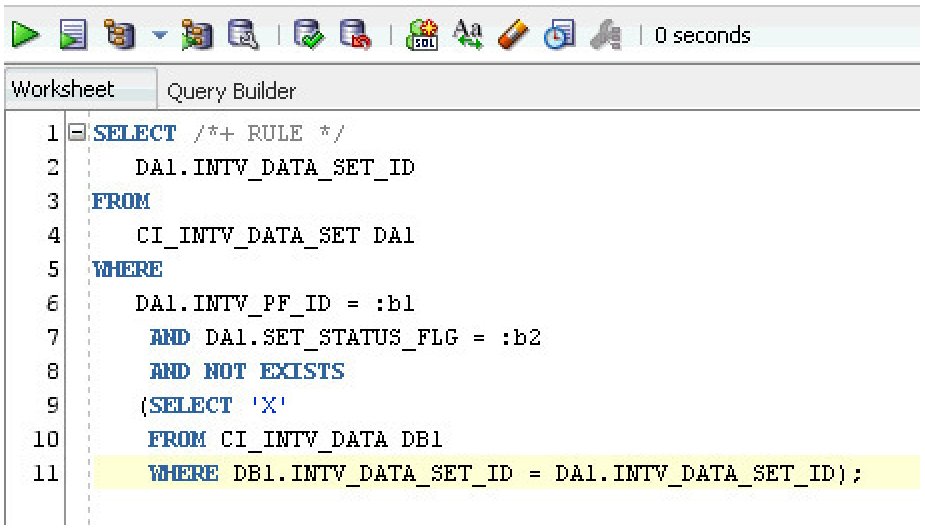


```
      SELECT
         DA1.INTV_DATA_SET_ID
      FROM
         CI_INTV_DATA_SET DA1
      WHERE
         DA1.INTV_PF_ID = :S-ERRDS-IN-DATA.RES-INTV-PF-ID
          AND DA1.SET_STATUS_FLG =
   :S-ERRDS-IN-DATA.ERROR-STATUS-FLG
          AND NOT EXISTS
         (SELECT 'X'
          FROM CI_INTV_DATA DB1
          WHERE DB1.INTV_DATA_SET_ID = DA1.INTV_DATA_SET_ID)
```

SQL To Check

- Adjust the SQL Statement:

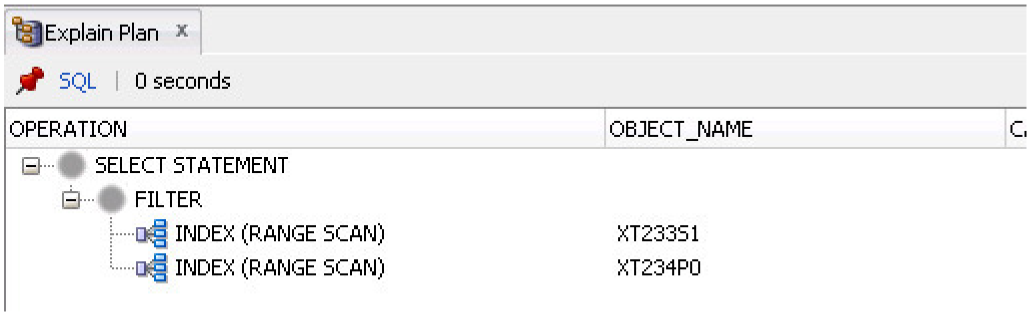
- Extract the tested SQL into the SQL Developer editor.
- Replace the COBOL name of each **Host Variable** with the equivalent database identifier **:b<n>** where n is a unique number identifying that host variable. If the same variable appears more than one in the SQL use the same database host variable id in all occurrences.
- Force the database to analyze the SQL in **Rule Base** mode by introducing the RULE database hint phrase.



```
SELECT /*+ RULE */
    DA1.INTV_DATA_SET_ID
FROM
    CI_INTV_DATA_SET DA1
WHERE
    DA1.INTV_PF_ID = :b1
     AND DA1.SET_STATUS_FLG = :b2
     AND NOT EXISTS
    (SELECT 'X'
     FROM CI_INTV_DATA DB1
     WHERE DB1.INTV_DATA_SET_ID = DA1.INTV_DATA_SET_ID);
```

Adjust the SQL Statement

- Generate the **Explain Plan:**
  - Position the cursor on the SQL statement.
  - Run the Explain Plan by hitting F10(or right-click anywhere on the SQL statement and click **Explain Plan...** from the context menu).
  - The generated plan will appear in the output tab.



```
OPERATION                          OBJECT_NAME
SELECT STATEMENT
  FILTER
    INDEX (RANGE SCAN)             XT233S1
    INDEX (RANGE SCAN)             XT234P0
```

Explain Plan

Analyzing the Explain Plan

Access Methods

Oracle finds the data to read by using the following methods:

- **Full Table Scan** (FTS). Using this method the whole table is read.
- **Index Lookup** (unique & non-unique). Using this method, data is accessed by looking up key values in an index and returning rowids where a rowid uniquely identifies an individual row in a particular data block.
- **Rowid.** This is the quickest access method available. Oracle simply retrieves the block specified and extracts the rows it is interested in. Most frequently seen in explain plans as Table access by Rowid.

Common Issues to Be Aware of

Cartesian Product

- A Join is a predicate that attempts to combine 2 row sources. Cartesian Product is created when there are no join conditions between 2 row sources and there is no alternative method of accessing the data. Typically this is caused by a coding mistake where a join has been left out. The CARTESIAN keyword in the Explain Plan indicates this situation.

Full Table Scan

- A Full Table Scan, e.g. TABLE ACCESS FULL phrase, found in the Explain Plan usually indicates an inefficient access path. This means that the only way the database found to get to the desired data is by reading every single row in the table.
- Notice that if the logic indeed requires reading all data, then this database decision is indeed correct. However, if you intended to get a small subset of rows from a large table and ended up reading all of it this is definitely not efficient and should be fixed. If this is the case, try and find a better SQL structure that would avoid a full table access. If you can't find such, please consult a DBA as this SQL may require an additional Index to be created for the table.
- Sometimes there would be a proper index on a particular table but still a full table scan would be chosen for the access path of that table. This may be as result of an inefficient Join Order. Please see details below.

Join Order

A Join is a predicate that attempts to combine 2 row sources. We only ever join 2 row sources together. Join steps are always performed serially even though underlying row sources may have been accessed in parallel. The join order makes a significant difference to the way in which the query is executed. By accessing particular row sources first, certain predicates may be satisfied that are not satisfied by with other join orders. This may prevent certain access paths from being taken.

- Make sure the join between 2 tables is done via indexed fields as much as possible.
- Also, if such an index exists, make sure you specify fields in the order they are defined by that index.

Nested Loops

This is a common type of processing a join between 2 row sources. First we return all the rows from row source 1, then we probe row source 2 once for each row returned from row source 1.

Row source 1

Row 1 ------------- -- Probe -> Row source 2

Row 2 ------------- -- Probe -> Row source 2

Row 3 ------------- -- Probe -> Row source 2

Row source 1 is known as the **outer table.** Row source 2 is known as the **inner table.** Accessing row source 2 is known a probing the inner table. For nested loops to be efficient it is important that the first row source returns as few rows as possible as this directly controls the number of probes of the second row source. Also it helps if the access method for row source 2 is efficient as this operation is being repeated once for every row returned by row source 1.

Sort

Sorts are expensive operations especially on large tables where the rows do not fit in memory and spill to disk.

There are a number of different operations that promote sorts:

- Order by clauses
- Group by
- Sort merge join

Note that if the row source is already appropriately sorted then no sorting is required. In other words, if the fields you sort by happen to be defined by an Index in that particular order then sort operation is avoided. Therefore, whenever you see that an explicit sort operation has taken place, check if it can be avoided by using an index or sometimes just by making sure your are using an index's fields in the right order.

If no such index exists and the number of rows to be sorted is of high volume, please consult a DBA as this may justify adding a new index.

More Extensive Performance Testing

Special attention should be paid to background processes that are designed to process high volume tables. A thorough performance testing exercise in a benchmark format may be called upon.

SQL Development and Tuning Best Practices

- Length of the DataType Matters.

  For example if you define a column with VARCHAR2(4000) (just the maximum limit) then you may outflow you array as given in the example below.

Varchar2(n) where *n* is "right sized"	**Varchar2(4000) for *everything* (just in case)**
Assume 10 columns, average width is 40 characters (some are 80, some are 10...).	Assume 10 columns, average, minimum, maximum width is 4000.
400 bytes per row on a fetch.	40,000 bytes per row on a fetch.
Assume array fetch of 100 rows, so array fetch buffer or 40,000 bytes.	Assume array fetch of 100 rows, so array fetch buffer or 4,000,000 bytes.
Assume 25 open cursors, so 1,000,000 bytes of array fetch buffers.	Assume 25 open cursors, so 100,000,000 bytes.
Assume connection pool with 30 connections — 30MB.	Assume connection pool with 30 connections — 3GB.

- NOT Null columns should be preferred over Null able columns. The reason is if you have an Index on a Null able column then it would not be used by the SQL as the optimizer thinks that it would not find any values in some of the columns so prefer a full scan.

  As a workaround for columns with NULL data types the Index create SQL should look like:

  ```
  Create INDEX ABC ON TAB1 (COLUMN1, 0);
  ```

This will make sure that in case the Column1 is null the optimizer will consider the value as 0 and leads to index scan as compared to Full scans.

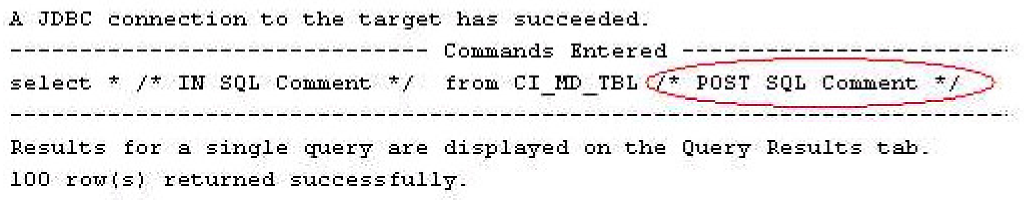
- Always try to substitute the Bind Variables in a SQL with the actual constant value if there is only one possible. Having too much Bind variables sometimes confuses the Optimizer to take the right access path. So this is good for the stability of the SQL plans.
- Fields which are foreign keys to other tables and are used in SQLs for the Join criterion are good candidates for creating Indexes on.
- Do not create any objects in the database of which the name may collide with any SQL reserved words.
- Views are generally used to show specific data to specific users based on their interest. Views are also used to restrict access to the base tables by granting permission only on views. Yet another significant use of views is that they simplify your queries. Incorporate your frequently required, complicated joins and calculations into a view so that you don't have to repeat those joins/calculations in all your queries. Instead, just select from the view.
- Avoid creating views within views as it affects the performance.
- Offload tasks, like string manipulations, concatenations, row numbering, case conversions, type conversions etc., to the front-end applications if these operations are going to consume more CPU cycles on the database server. Also try to do basic validations in the front-end itself during data entry. This saves unnecessary network roundtrips.
- Always be consistent with the usage of case in your code. On a case insensitive server, your code might work fine, but it will fail on a case-sensitive server if your code is not consistent in case.
- Make sure you normalize your data at least to the 3rd normal form. At the same time, do not compromise on query performance. A little bit of denormalization helps queries perform faster.
- Consider indexing those columns if they are frequently used in the ORDER clause of SQL statements.
- Use tools like Tkprofs and AWR Report for measuring the Performance of your SQLs.
- In the SQL Explain Plans, usage of Nested Loops are good when there are table joins involved.
- Always looks for Autotrace to measure the SQL plan as it is closer to the plan which the optimizer takes during the actual execution of the SQL. This can be get easily through SQL Developer and other database monitoring tools.
- While looking at the Autotrace Plans look for consistent gets and make sure they are low. The other thing reported by the Autotrace is COST. Do not worry too much about cost if the Consistent gets is low and you are getting a desirable Plan.
- Make sure that the Statistics are current and not stale while you are trying to Tune a SQL.
- Having Secondary Unique Indexes help in achieving Index Unique scans. This will eliminate the Table scans. It is worth trying and see if that makes a difference.

- Oracle Optimizer executes the explain plans of a SQL from Inner to the outer area and from bottom to the top. So make sure that the cardinality of the inner most Join criterion should be low.
- Always keep in mind the usage of the SQL in a real production scenario where the data in the tables can go exponentially. Make sure that the SQLs can handle it and the Explain plan should be accordingly tuned.
- Usage of <<, != make the Index NOT to be used. Instead of this use the greater than or less than statements.
- If you wrap a column a column with some functions like TO_DATE, TO_CHAR, SUBSTR and so on then the Index on the Column would not be used.
- Avoid using UNION and make sure you use UNION ALL if possible. This will boost performance.
- Using EXISTS , NOT EXISTS are better than using IN , NOT IN statements respectively.

- Usage of Leading Hints helps in choosing what Table should be the first table in the join order.

```
/*+ LEADING(Table1 Table2) */
```

- When writing comments within SQL statements make sure that the comments are not added at the beginning because DB2 will not be able to parse it. You can instead put the Comments at the end and it will work. For Oracle this is not an issue.



```
A JDBC connection to the target has succeeded.
------------------------------ Commands Entered ------------------------
select * /* IN SQL Comment */  from CI_MD_TBL /* POST SQL Comment */
------------------------------------------------------------------------
Results for a single query are displayed on the Query Results tab.
100 row(s) returned successfully.
```

Additional Resources

Additional information on optimizing SQL in your OUAF applications can be found in the *Oracle Utilities Application Framework - Technical Best Practices* whitepaper available on the My Oracle Support (MOS) Knowledge Base (article 560367.1).

Database Design

The objective of this document is to provide a standard for database objects (such as tables, columns, and indexes) for products using Oracle Utilities Application Framework. This standard is introduced to insure clean database design, to promote communications, and to reduce errors leading to smooth integration and upgrade processes. Just as Oracle Utilities Application Framework goes thorough innovation in every release of the software, it is also inevitable that the product will take advantage of various database vendors' new features in each release. The recommendations in the database installation section include only the ones that have been proved by vigorous QA processes, field tests and benchmarks.

Database Object Standard

This section discusses the rules applied to naming database objects and the attributes that are associated with these objects.

Naming Standards

The following naming standards must be applied to database objects.

Table

Table names are prefixed with the owner flag value of the product. For customer modification **CM** must prefix the table name. The length of the table names must be less than or equal to 30 characters. A language table should be named by suffixing **_L** to the main table. The key table name should be named by suffixing **_K** to the main table.

It is recommended to start a table name with the 2-3 letter acronym of the subsystem name that the table belongs to. For example, **MD** stands for meta-data subsystem and all meta data table names start with **CI_MD**.

Some examples are:

- CI_ADJ_TYPE
- CI_ADJ_TYPE_L

Note:
A language table stores language sensitive columns such as a description of a code. The primary key of a language table consists of the primary key of the code table plus language code (LANGAGUE_CD).

Note:
A key table accompanies a table with a surrogate key column. A key value is stored with the environment id that the key value resides in the key table.

Note:
The tables prior to V2.0.0 are prefixed with CI_ or SC_.

Columns

The length of a column name must be less than or equal to 30 characters. The following conventions apply when you define special types of columns in the database.

Use the suffix **FLG** to define a lookup table field. Flag columns must be CHAR(4). Choose lookup field names carefully as these column names are defined in the lookup table (CI_LOOKUP_FLD) and must be prefixed by the product owner flag value.

Use the suffix **CD** to define user-defined codes. User-defined codes are primarily found as the key column of the admin tables.

Use the suffix **ID** to define system assigned key columns.

Use the suffix **SW** to define Boolean columns. The valid values of the switches are 'Y' or 'N'. The switch columns must be CHAR(1)

Use the suffix **DT** to define Date columns.

Use the suffix **DTTM** to define Date Time columns.

Use the suffix **TM** to define Time columns.

Some examples are:

- ADJ_STATUS_FLG
- CAN_RSN_CD

Indexes

Index names are composed of the following parts:

[X][C/M/T]NNN[P/S]

X - letter **X** is used as a leading character of all base index names prior to Version 2.0.0. Now the first character of product owner flag value should be used instead of letter X. For client specific implementation index in **Oracle**, use **CM.**

C/M/T - The second character can be either **C** or **M** or **T**. **C** is used for control tables (Admin tables). **M** is for the master tables. **T** is reserved for the transaction tables.

NNN - A three-digit number that uniquely identifies the table on which the index is defined.

P/S/C - **P** indicates that this index is the primary key index. **S** is used for indexes other than primary keys. Use **C** to indicate a client specific implementation index in **DB2** implementation.

Some examples are:

- XC001P0
- XT206S1
- XT206C2
- CM206S2

Warning:
Do not use index names in the application as the names can change due to unforeseeable reasons.

Sequence

The base sequence name must be prefixed with the owner flag value of the product.

Trigger

The base trigger name must be prefixed with the owner flag value of the product.

Note:
When implementers add database objects, such as tables, triggers and sequences, the name of the objects should be prefixed by **CM.** For example, Index names in base product are prefixed by **X;** the Implementers' index name must not be prefixed with **X.**

Column Data Type and Constraints

User Define Code

User Defined Codes are defined as CHAR type. The length can vary by the business requirements but a minimum of eight characters is recommended. You will find columns defined in less than eight characters but with internationalization in mind new columns should be defined as CHAR(10) or CHAR(12). Also note that when the code is referenced in the application the descriptions are shown to users in most cases.

System Assigned Identifier

System assigned random numbers is defined as CHAR type. The length of the column varies to meet the business requirements. Number type key columns are used when a sequential key assignment is allowed or number type is required to interface with external software. For example, Notification Upload Staging ID is a Number type because most EDI software uses a sequential key assignment mechanism. For sequential key assignment implementation, the DBMS sequence generator is used in conjunction with Number Type ID columns.

Date/Time/Timestamp

Date, Time and Timestamp columns are defined physically as DATE in Oracle. In DB2 the DATE, TIME and TIMESTAMP column types, respectively, are used to implement them. Non-null constraints are implemented only for the required columns.

Number

Numeric columns are implemented as NUMBER type in Oracle and DECIMAL type in DB2. The precision of the number should always be defined. The scale of the number might be defined. Non-null constraints are implemented for all number columns.

Fixed Length/Variable Length Character Columns

When a character column is a part of the primary key of a table define the column in CHAR type. For the non-key character columns, the length should be the defining factor. If the column length should be greater than 10, use VARCHAR2 type in Oracle and VARCHAR type in DB2.

Null Constraints

The Non-null constraints are implemented for all columns except optional DATE, TIME or TIMESTAMP columns.

Default Value Setting

The rule to set the database default value is the following:

- When a predefined default value is not available, set the default value of Non-null CHAR or VARCHAR columns to blank except the primary key columns.
- When a predefined default value is not available, set the default value Non-null Number columns to 0 (zero) except the primary key columns.
- No database default values should be assigned to the Non Null Date, Time, and Timestamp columns.

Foreign Key Constraints

Referential Integrity is enforced by the application. In database, do not define FK constraints. Indexes are created on most of Foreign Key columns to increase performance.

Standard Columns

Owner Flag

Owner Flag (OWNER_FLG) columns exist on the system tables that are shared by multiple products. Oracle Utilities Application Framework limits the data modification of the tables that have owner flag to the data owned by the product.

Version

The **Version** column is used to for optimistic concurrency control in the application code. Add the **Version** column to all tables that are maintained by a Row Maintenance program.

System Table Guide

Key components of products built on the Oracle Application Framework are the system tables. The data in those tables controls many aspects of the application. There are standards required for these tables to support the installation, development, configuration and customization of the Oracle Utilities products. Implementations add their own records to the system tables. Adhering to the data standards is a prerequisite for seamless upgrade to the next release of the product(s). Please refer to the Oracle Utilities Application Framework System Table Guide section of your product's Database Administration Guide for details about the system tables and the standards to follow.

Key Generation

Key generation is performed for tables that have sequential or system generated prime key. This is performed automatically for Java instances via the OUAF enTegrity.

Tables with a system-generated key contain their own unique key that is replicated in a related 'key table' suffixed with '_K'. The purpose of the key table is to store the table identifier as well as the identifier of the environment in which the data row exists. An example is the Account table containing the Account identifier and the Account Key table containing the Account identifier and the Environment identifier.

These key tables support the Archiving and ConfigLab functionality by ensuring that a key will be unique across environments.

Note:
Oracle recommends that customers using Oracle Utilities Application Framework version 4.2 or later and currently using ConfigLab switch to Configuration Migration Assistant (CMA) for their configuration data migration needs and retain ConfigLab for migration of master and transaction data migration. Also note that CMA functionality is not available to every Framework-based product. For details, including tips and requirements for moving from ConfigLab to CMA for configuration data migrations, see the "Configuration Migration Assistant" section in the *Oracle Utilities Application Framework Aministration Guide*.

Metadata for Key Generation

The required table metadata that is used by the key generator indicates:

- The type of key, e.g. whether it is system-generated or sequential
- The key table in which key values are stored
- The length of the inherited portion of the key.

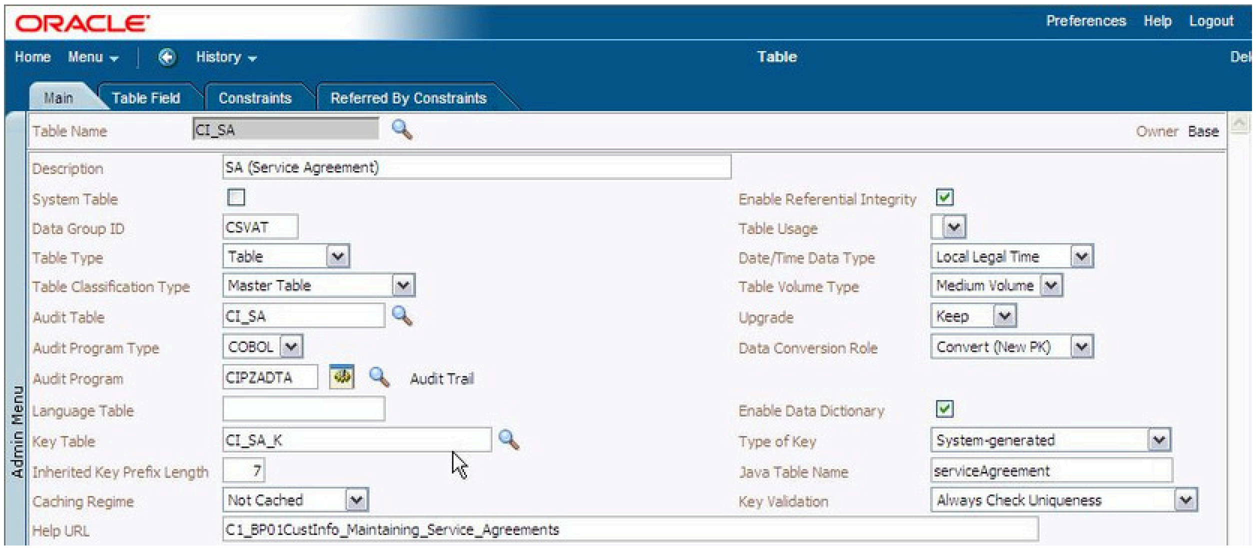


Table Name	CI_SA	Owner	Base
Description	SA (Service Agreement)		
System Table	☐	Enable Referential Integrity	✓
Data Group ID	CSVAT	Table Usage	
Table Type	Table	Date/Time Data Type	Local Legal Time
Table Classification Type	Master Table	Table Volume Type	Medium Volume
Audit Table	CI_SA	Upgrade	Keep
Audit Program Type	COBOL	Data Conversion Role	Convert (New PK)
Audit Program	CIPZADTA		
Language Table		Enable Data Dictionary	✓
Key Table	CI_SA_K	Type of Key	System-generated
Inherited Key Prefix Length	7	Java Table Name	serviceAgreement
Caching Regime	Not Cached	Key Validation	Always Check Uniqueness
Help URL	C1_BP01CustInfo_Maintaining_Service_Agreements		

Example Table Metadata Key Information

In the Service Agreement table metadata example above, the metadata key information is shown by the values in the fields Key Table, Type of Key and Inherited Key Prefix Length.

The primary key constraint is used to retrieve the name of the key field for the table from the field metadata.

The field metadata provides the field data type and length.

Note:
Key Types. Although there are more types of keys indicated in metadata drop-down list, the only types currently supported by the key generator in the Oracle Utilities Application Framework are system-generated and sequential.

Note:
Special Annotation. If a table's inherited key prefix length is non-zero, a special entry "clusteringParentProperty" must be in the business entity annotation for this table.

Development Performance Guidelines

This document includes information, guidelines, and strategies to help designers and developers understand performance impacts when developing a feature using the Oracle Utilities Application Framework.

Object-Relational Mapping: Background

OUAF uses an Object-Relational Mapping (ORM) engine, which maps tables to entities using the system's table, table/field, field, and constraint metadata to guide the creation of mapping definitions during artifact generation.

Entities represent database tables. They are created as Java objects during a database "session", which has the lifetime of a single DB transaction.

DONT: Entities are not safe to use for reference, calling methods, etc., after the session that created them has ended. For example, don't copy entities into application caches. DO: Instead, let the application cache do the data retrieval and return the data to the session. ID objects are safe to store across sessions. Note in the following example that the entity AlgorithmType is not stored:

```
public class AlgorithmTypeInfoCache implements ApplicationCache {
  private static final AlgorithmTypeInfoCache INSTANCE = new AlgorithmTypeInfoCache();
  private final ConcurrentMap<AlgorithmType_Id, AlgorithmTypeInfo> algorithmTypeInfoById = new ConcurrentHashMap<Algori
thmType_Id, AlgorithmTypeInfo>();
  protected AlgorithmTypeInfoCache() {  ContextHolder.getContext().registerCache(this);    }
  public String getName() {  return "AlgorithmTypeInfoCache";    }
  public void flush() {algorithmTypeInfoById.clear();    }
  public static AlgorithmTypeInfo getAlgorithmTypeInfo(AlgorithmType_Id algTypeId) {
    return INSTANCE.privateGetAlgorithmTypeInfo(algTypeId);
  }

  private AlgorithmTypeInfo privateGetAlgorithmTypeInfo(AlgorithmType_Id algTypeId) {
    AlgorithmTypeInfo algTypeInfo = algorithmTypeInfoById.get(algTypeId);
    if (algTypeInfo != null) return algTypeInfo;
    AlgorithmType type = algTypeId.getEntity();
    if (type == null) return null;
    AlgorithmTypeInfo info = new AlgorithmTypeInfo(type);
    AlgorithmTypeInfo prev = algorithmTypeInfoById.putIfAbsent(algTypeId, info);
    if (prev != null) return prev;
    return info;
  }
}
```

DO: it is safe to use XML documents (to be consumed by BOs, BSs, or SSs) for moving data between sessions.

Every entity has a unique corresponding "id" class, e.g. BatchControl has BachControlId. The ORM framework automatically generates correct SQL to perform the following essential tasks:

- Read, update, insert, delete one entity (row) from the database.
- Navigate between related entities as per their key/constraint relationships, for example from a parent entity to a collection of children.

The ORM defers database calls for performance

The ORM tries to be as "lazy" as possible; its basic stance is to avoid loading any data from the DB until the last possible moment. Let's use the following example to describe how the data is only loaded at last moment possible:

```
BatchControl someBatchControl = batchControlId.getEntity();

BatchControlParameters parms = someBatchControl.getParameters();

for (BatchControlParameter each : parms) {

String name = each.getBatchParameterName();

}
```

In the above example, the getEntity() call only retrieves the parent ID as a proxy. The "someBatchControl" is not fully "hydrated" until some other property is accessed. "Hydrating Entities" is the process of taking a row from the database and turning it into an entity.

The getParameters() call only retrives the child IDs, again as proxies.

Only when the getBatchParameterName() is called, is a row (the child row) actually retrieved.

ID Objects

- When you create an ID, the ID object will not be null. After you use an ID to retrieve an entity (using getEntity()), that is when you find out if the entity actually exists. Just because an ID exists, doesn't mean the entity itself exists! DO: So you must check for null before attempting to use the entity you retrieved. For example:

```
BatchControlId id = ...

BatchControl batchControl = id.getEntity();

if (batchControl == null) { /* oh oh */ }
```

Counting a collection

DO: If you want to count the number of batch control parameters that belong to a parent batch control, use the size() method as in the following example:

```
BatchControl someBatchControl = ...;

BatchControlParameters parms = someBatchControl.getParameters();

int count = parms.size();
```

The framework implementation code has an optimized implementation of the size() method, which either counts the existing in-memory collection elements, if they are already present, or issues a SQL count(*) query, if they aren't.

Avoid unnecessary work

DON'T: In the example, below, the call to listSize() is unnessary. In most cases, you shouldn't need to write something to loop over a collection:

```
if(query.listSize() > 0) {

while (iter.next()) { .... }

}
```

The call to listSize() will make an unnecessary call to "select count(*)". Let the iterator do the work. Avoid the extra call to the database.

ORM 'Navigation' is your friend

Don't be tempted to hand-write queries that are equivalent to navigations between entities:

```
BatchControlId batchControlId = ...

Query<BatchControlParameter> query = createQuery("from BatchControlParameter parm where parm.id.batchControlId = :paren
tId");

query.bindId("parentId", batchControlId);

List<BatchControlParameter> list = query.list();
```

DO: Use this instead - it'll use the cache and will almost certainly be faster:

```
BatchControl batchControl = id.getEntity();

if (batchControl == null) { /* oh oh */ }

BatchControlParameters list = batchControl.getParameters();
```

How to Pre-load Entities Using Fetch

This technique is for performance intensive jobs that are doing too many single-row SQL retrieves. The "fetch" command will pre-load the entities, resulting in one fewer database calls.

Write a query using "left join fetch" to select all data. The ORM will fetch the associated collection for every retrieved table into the session cache. Subsequent navigation to the underlying collection is then an in-memory operation with no database IO. Again, PREFER code that performs standard navigation.

- As a general strategy:
    - For most jobs, navigation is just fine.
    - Write code using navigation first, then ADD the fetch query later, only if it's needed.

This is a link to the Hibernate help on "fetch": http://docs.jboss.org/hibernate/stable/core/reference/en/html_single/#performance-fetching

Session Cache

If an entity is retrieved previously within a session it does not, in most cases, have to be retrieved again since it is stored in the session cache.

As a result, multiple BO reads against Java-backed MOs do not re-execute SQL.

Level 2 Cache Applicable for Batch

Hibernate's Level 2 cache is a second level of caching that allows sharing of data between sessions. This is useful for static, admin data like rates, type codes, etc., since objects that are added to this cache cannot be updated. The caching is enabled per entity on the Table transaction's Caching Regime Flag with values of "Not Cached" and "Cached for Batch." By default, the product is not configured to have Log and Language tables as *not* cached.

Flushing - COBOL and Save Points

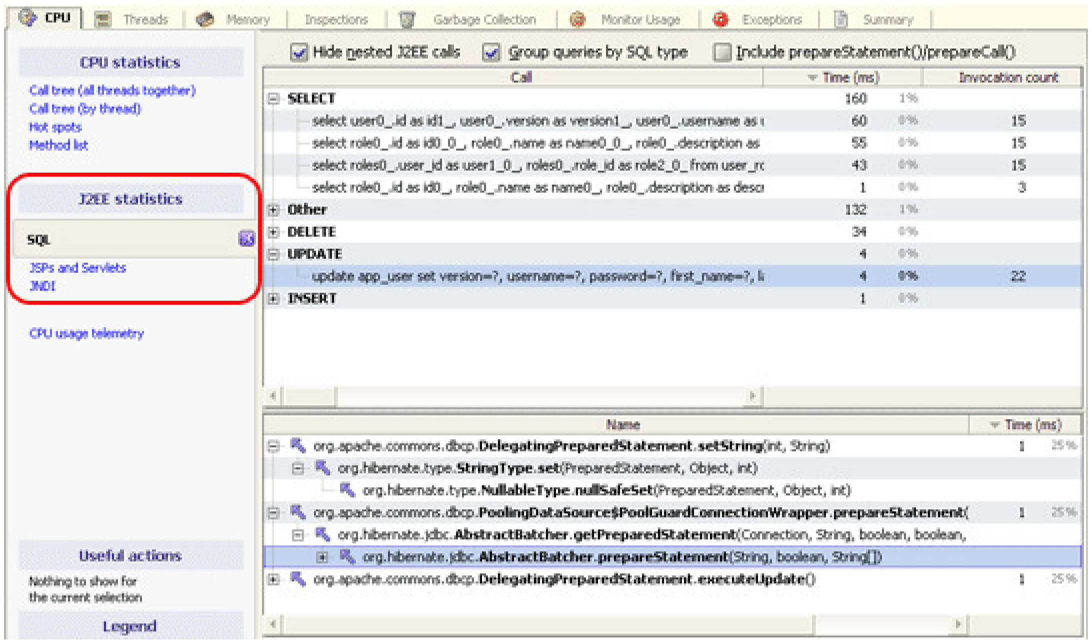
Flushing means writing the changes to the database. It syncs the database with the session cache. Flushing is expensive but necessary to maintain data integrity. The system flushes under the following conditions:

- Before commit
- Before raw SQL queries
- Before most HQL queries
- When specifically requested
- Savepoints

Avoid Extra SQL

Inspect generated SQL for extra calls. Tools like Oracle's tkprof, Yourkit java profiler, or debug application logs can help identify extra database calls. The screen capture below shows how Yourkit is able to reveal SQL statements behind PreparedStatement calls.

Yourkit Demo: http://www.yourkit.com/docs/demo/JavaEE/JavaEE.htm



✓ Hide nested J2EE calls ✓ Group queries by SQL type ☐ Include prepareStatement()/prepareCall()

Call	Time (ms)		Invocation count
SELECT	160	1%	
select user0_.id as id1_, user0_.version as version1_, user0_.username as u	60	0%	15
select role0_.id as id0_0_, role0_.name as name0_0_, role0_.description as	55	0%	15
select roles0_.user_id as user1_0_, roles0_.role_id as role2_0_ from user_rc	43	0%	15
select role0_.id as id0_, role0_.name as name0_, role0_.description as desc	1	0%	3
Other	132	1%	
DELETE	34	0%	
UPDATE	4	0%	
update app_user set version=?, username=?, password=?, first_name=?, l	4	0%	22
INSERT	1	0%	

Name	Time (ms)	
org.apache.commons.dbcp.**DelegatingPreparedStatement.setString**(int, String)	1	25%
org.hibernate.type.**StringType.set**(PreparedStatement, Object, int)		
org.hibernate.type.**NullableType.nullSafeSet**(PreparedStatement, Object, int)		
org.apache.commons.dbcp.**PoolingDataSource$PoolGuardConnectionWrapper.prepareStatement**(	1	25%
org.hibernate.jdbc.**AbstractBatcher.getPreparedStatement**(Connection, String, boolean, boolean,		
org.hibernate.jdbc.**AbstractBatcher.prepareStatement**(String, boolean, String[])		
org.apache.commons.dbcp.**DelegatingPreparedStatement.executeUpdate**()	1	25%

Prepared statement - use binding

DON'T: Never concatenate values - DO: use binding instead. Besides helping to reduce security concerns with SQL injection, concatenation results in reparsing of SQL statements by the database. You could also lose the benefits of any PreparedStatement caching by the jdbc drivers.

Service Script vs. Java Services

Service Scripts perform slower than java services. There is an overhead on scripting that comes from xml manipulation and xpath evaluation. Lots of moves, complicated XPath - proportional to amount of XPath. Here are some tips:

- One complicated XPath expression should be faster than several smaller ones - the overhead is in the setup.
- Smaller documents will process faster - think about that when designing script schemas - only send what you need.

Java Performance Patterns

- Loop over entryset of a hashmap, not the entities
- Concatenate strings using StringBuilder
- **Use Findbugs** - it will help expose patterns to be avoided.

Batch Performance

Commit Considerations

DON'T: Do not commit too frequently. For example, we do not commit ever record since each commit has overhead at the database; however, sessions with lots of objects in the cache should commit more frequently. Adjust your default value accordingly.

Clustered vs. Distributed Mode Performance: Clustered is Preferred

No coding changes required for clustered mode and no reason to use the distributed mode anymore.

Clustered mode was created for stability, not performance; however, clustered mode should have less overhead because "tspace" table is not continually accessed. This tspace table stored the batch job's instructions and information used by the distributed mode and accessed by the batch threads. There have been cases where this table is in high contention.

Light Business Objects

Sometimes it is possible to create a smaller schema that only accesses the objects that are needed for a particularly performance-instensive process. This can be particularly important if there are many child collections that are referenced by a particular business object (BO). When a child collection on the maintenance object (MO) is not mapped, the application does not issue a read. While a large BO with many collections might be acceptable for online transactions, a process used in a batch could benefit by a smaller subset of elements. These smaller BOs are referred to as Light or "Lite" BOs.

When any XML-mapped field is referenced, the application must parse the XML's columns data. Depending on how much data is mapped, there could be some minor savings gained by avoiding the XML data since it will not need to be parsed.

Here is an example of a very large BO:

```
<schema fwRel="2">

<mainSection type="group" mdField="C1_TAXFORM_MAIN_LBL"/>
```

```
<selectTaxFormLabel type="group" mdField="C1_SELECT_TAXFORM_LBL"/>

<taxFormId mapField="TAX_FORM_ID" isPrimeKey="true" fkRef="C1-TXFRM"/>

<formType mapField="FORM_TYPE_CD" fkRef="C1-FRMTY"/>

<bo mapField="BUS_OBJ_CD" fkRef="F1-BUSOB" suppress="true"/>

<boStatus mapField="BO_STATUS_CD"/>

<statusUpdateDateTime mapField="STATUS_UPD_DTTM" suppress="true"/>

<totalAmountDueOverpaid mapField="C1_TOTAL_OWED_OVERPAID"/>

<remittanceAmount mapField="C1_FORM_PYMNT_AMT"/>

<formSource mapField="C1_FORM_SRCE_CD" fkRef="C1-FRMSC"/>

<formDetailsInfoSection type="group" mdField="C1_ITF_FORM_DTLS_INFO"/>

<demographicInfoSection type="group" mdField="C1_TF_DEMO_INFO_LBL"/>

<taxpayerDemographicInformation type="group">

<includeDA name="C1-TaxpayerDemoInfo"/>

</taxpayerDemographicInformation>

<lineItemsInfoSection type="group" mdField="C1_TF_LINE_ITEMS_LBL"/>

<obligationId mapField="SA_ID" fkRef="SA"/>

<receiveDate mapField="RECV_DT" required="true"/>

<taxFormFilingType mapField="TAX_FORM_FILING_TYPE_FLG" default="C1OR" required="true"/>

<formBatchHeaderId mapField="FORM_BATCH_HDR_ID" fkRef="C1-FBHDR"/>

<documentLocator mapField="DOC_LOC_NBR"/>

<adjustReason dataType="lookup" mdField="C1_TXF_ADJRSN_FLG" mapXML="BO_DATA_AREA"/>

<transferReason dataType="lookup" mdField="C1_TXF_TFRRSN_FLG" mapXML="BO_DATA_AREA"/>

<reverseReason dataType="lookup" mdField="C1_TXF_RVSRSN_FLG" mapXML="BO_DATA_AREA"/>

<cancelReason dataType="lookup" mdField="C1_TXF_CANRSN_FLG" mapXML="BO_DATA_AREA"/>

<issuesSection type="group" mdField="C1_TF_ISSUES_LBL"/>

<suspenseIssueList type="group" mapXML="BO_DATA_AREA">

<includeDA name="C1-IssuesList"/>

</suspenseIssueList>

<waitingForInfoIssueList type="group" mapXML="BO_DATA_AREA">

<includeDA name="C1-IssuesList"/>

</waitingForInfoIssueList>

<taxpayerPersonID fkRef="PER" mdField="PER_ID" mapXML="BO_DATA_AREA"/>

<account fkRef="ACCT" mdField="ACCT_ID" mapXML="BO_DATA_AREA"/>

<taxRole fkRef="C1-TXRL" mdField="TAX_ROLE_ID" mapXML="BO_DATA_AREA"/>

<transferAdjustSection type="group" mdField="C1_TF_TRANSFER_ADJUST"/>

<adjustedFromForm fkRef="C1-TXFRM" mapXML="BO_DATA_AREA" mdField="C1_TF_ADJUSTED_FROM"/>

<transferredFromForm fkRef="C1-TXFRM" mapXML="BO_DATA_AREA" mdField="C1_TF_TRANSFERRED_FROM"/>
```

```
<adjustedToForm fkRef="C1-TXFRM" mapXML="BO_DATA_AREA" mdField="C1_TF_ADJUSTED_TO"/>

<transferredToForm fkRef="C1-TXFRM" mapXML="BO_DATA_AREA" mdField="C1_TF_TRANSFERRED_TO"/>

<version mapField="VERSION" suppress="true"/>

</schema>
```

This performance-sensitive process only required a subset of the BO, so the following light BO was created:

```


<schema fwRel="2">

<taxFormId mapField="TAX_FORM_ID" isPrimeKey="true" fkRef="C1-TXFRM"/>

<formType mapField="FORM_TYPE_CD" fkRef="C1-FRMTY"/>

<bo mapField="BUS_OBJ_CD" fkRef="F1-BUSOB" suppress="true" required="true"/>

<boStatus mapField="BO_STATUS_CD"/>

<receiveDate mapField="RECV_DT"/>

<formChangeReasons type="group" mapXML="BO_DATA_AREA">

<formChangeReasonsList type="list">

<formChangeReason mdField="FORM_CHG_RSN_CD" isPrimeKey="true" fkRef="C1-FRCHR"/>

</formChangeReasonsList>

</formChangeReasons>

<formChangeComments mdField="COMMENTS" mapXML="BO_DATA_AREA"/>

<version mapField="VERSION" suppress="true"/>

</schema>
```

Benefits of Light BOs

- Not mapping unneeded child collections removes the need to read the child table from the database.
- The "read" benefits extend to BO updates as well, though in a minor way. The light BO would be used for the read when finishing an update. However, this benefit may be reduced by the availability of the **updateWithoutRead** action, which would do the update with whatever BO was presented, and then *not* do a read at the end. This would achieve an optimization as well as the light BO update.
- XML document size reduction. There are performance penalties when transmitting and parsing large XML documents.

Disadvantage of Light BOs

While this is not really a disadvantage of the concept, creating additional BOs results in more objects to maintain.

Tips and Conventions

- The light BO Code ends with "Light" or "Lite" and description ends with "Light". The rest of the code and description should follow the normal BO naming standard.
- Instance Control is set to "Do not allow new instances".
- Each MO has one parent light BO. The parent light BO schema would contain just the first level elements and no collections.
- Each "child" light BO for the MO has the parent light BO as its parent. This is important for the BO Hierarchy dashboard zone.
- When to update an existing light BO versus creating a new light BO: Generally reuse and update an existing light BO if the level or "collection level" of the element to add is in an existing light BO. A new light BO is warranted if a new "collection level" is needed. If multiple levels and many elements are needed, you may consider reading the actual BO instead of the light BO.

Data Explorer

It is important to understand that Data Explorers process ALL records returned from the database, even if they are not displayed. For example, FK ref info strings, BS calls, SS calls, Inhibit Row in Explorers - all can cause per-row processing even if they are not displayed.

Data Explorers are rendered using JavaScript. They are not designed to display many records, and trying to do so will result in possibly unacceptable performance. DO: Consider limiting the results returned and asking Users to add additional filter criteria to narrow down the results. DON'T: Don't try to display hundreds of records.

Zone Configuration

- DO: Consider limiting the number of rows retrieved by the database limiting the query size. Specify this on the zone parameter and the query will use the "rownum" technique to restrict the number of rows returned.
- DO: As a rule-of-thumb: 10 columns (even if not visible) in a data explorer zone should be an alert to really think about performance implications.

- DO: Try to perform all processing in the SQL instead of fkInfo, BS, or SS calls in other columns. As described earlier, these would be additional processes run on a per-row basis. If a description exists, consider using the description and a Navigation Option instead of the Foreign Key. For example, replace the Person fkRef with Person Name and its Navigation Option.

Table Indices and SQL

Here are some more common patterns to look out for. (This is not meant to be a complete SQL tuning guide.)

- Put Indexes on the most commonly used criteria. If there is no proper index, the optimizer does a full table scan. Consider:
    - Primary keys, foreign keys, ORDER BY fields.
    - Secondary Unique Indexes
- DO: Use a JOIN instead of EXISTS. This is faster for unique scan indexes.
- DO: Use EXISTS instead of IN when working with ID fields, use '=' instead of LIKE. Using LIKE on a system-generated key isn't "reasonable"
- CONSIDER: Using functions like TO_DATE(), SUBSTR() etc. means indexes on those fields won't be used! Use only when necessary.
- DO: Use the power of optional filters - and not just in the WHERE clause.

```
FROM d1_tou_map tm, d1_tou_map_l tml

FROM d1_tou_map tm,  [(F1) d1_tou_map_l tml,]
```

- DO: Only include necessary tables:

```
SELECT A.usg_grp_cd, A.usg_rule_cd, A.exe_seq,A.referred_usg_grp_cd,A.usg_rule_cat_flg, B.crit_seq, C.descr100
 DESCR
FROM D1_USG_RULE A, d1_usg_rule_elig_crit B, d1_usg_rule_elig_crit_l C
WHERE A.usg_grp_cd= :H1
AND A.usg_grp_cd = B.usg_grp_cd
AND A.usg_rule_cd = B.usg_rule_cd
AND b.usg_grp_cd = C.usg_grp_cd
AND b.usg_rule_cd = C.usg_rule_cd
AND b.crit_seq = C.crit_seq
AND C.language_cd= :language
```

Note that Table B is not necessary; you could instead simply link directly from A to C.

- Offload tasks, like string manipulations, concatenations, row numbering, case conversions, type conversions etc., to the front-end applications
- Do basic validations in the front-end during data entry. This saves unnecessary network roundtrips.
- Avoid using UNION - use UNION ALL if it is possible.
- Operators <> and != will NOT use the index! Also the word "NOT" Use the Greater Than or Less Than operators.

```
select * from ci_scr_step where  (scr_cd <> 'ZZCW03') has cost 68

select * from ci_scr_step where (scr_cd > 'ZZCW03' or scr_cd < 'ZZCW03') has cost 1!!!
```

UI Maps and BPAs

UI maps will not be able to display many rows very quickly. DONT display hundreds of rows in a UI Map. Alternatively, the zone type "SERVICE" can display a large number of records faster.

DO: Ensure that the html code is proper. Malformed HTML in UI maps (for example, opening a <div> and not closing it) can cause significant performance degradations at the browser. It is possible to copy and paste HTML into Eclipse to check its validity. There are also various tools like html tidy that can help to identify bad html.

DO: Minimize browser-to-server calls. Namely, invokeBO/BS/SS will perform a call to a server to retrieve the data, which can be slow. Many of these such calls on load of the UI Map will result in slow performance.

- Use service script instead of BPA if multiple calls need to be made to BO, BS, SS.
- Create a "bulk" processing service script instead of calling the same one multiple times. Instead of multiple invokeBS calls on load of a UI map, write a pre-processing service script instead.

Diagnosing Performance Issues

Execution times can be obtained in a number of ways.

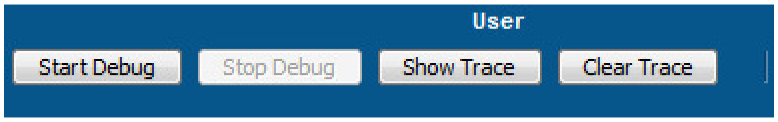
Fiddler

In a UI-oriented scenario, the first recommended analysis tool is to use an http logger like fiddler (http://www.fiddler2.com). This tool should make it apparent if there are excessive calls from the client browser to the server and the server response times as seen from the browser. The timings can then be categorized as server-side or client side calls. When using fidder be sure to enable the following:

- "Time-to-Last-Byte"
- "Response Timestamp"

OUAF 'Show Trace' button

Enable debug mode by adding debug=true to the url. Then use the "Start Debug", "Stop Debug" and "Show Trace" buttons



Log Service times in spl_service.log

In log4j2 properties, add the following, including adding "serviceDispatcher” to the comma-separated named loggers, to log service execution times:

```
logger.serviceDispatcher.name = com.splwg.base.api.service.ServiceDispatcher

logger.serviceDispatcher.appenderRef.userLog.ref = userLog

logger.serviceDispatcher.level = debug

logger.serviceDispatcher.additivity = false
```

Optimization and Performance Profiling

To squeeze every second of a given program for mission critical optimizations, it may be necessary to craft a repeatable unit test and profile the results using a profiling tool such as YourKit (www.yourkit.com). This section will include some code samples to log execution times. Attaching a profiler could give clues to optimization points. A common pattern to follow in testing code is to allow the System to "warm up," for example to load up the necessary application caches which are only done once and are not relevant to the code being optimized.

Basic Logging

The following code can be placed in a junit test to log execution times:

```
long start = logger.debugStart("Starting process");

//... code for process

logger.debugTime("End process", start);
```

Timing code ('shootout'):

The code below will run a BO Update 100 times and report the amount of time taken. Note the 5 "warmup" executions before the repeated 100 runs.

```
    public void testMultiplePluginScripts() throws Exception {
        String docString1 = "<DR_ShortCreateIntervalRecords><factId>219250542869</factId><longDescr>REEE</longDescr></D
R_ShortCreateIntervalRecords>";
        Document doc1 = DocumentHelper.parseText(docString1);

        String docString2 = "<DR_ShortCreateIntervalRecords2><factId>219250542869</factId><longDescr>REEE</longDescr
></DR_ShortCreateIntervalRecords2>";
        Document doc2 = DocumentHelper.parseText(docString2);

        // warmups
        for (int i = 0; i < 5; i++) {
            BusinessObjectDispatcher.execute(doc1, BusinessObjectActionLookup.constants.FAST_UPDATE);
            rollbackAndContinue();
            BusinessObjectDispatcher.execute(doc2, BusinessObjectActionLookup.constants.FAST_UPDATE);
            rollbackAndContinue();
        }

        long totalElapsed = 0;
        // speed
        for (int i = 0; i < 100; i++) {
            long start = System.nanoTime();
            BusinessObjectDispatcher.execute(doc1, BusinessObjectActionLookup.constants.FAST_UPDATE);
            flush();
            totalElapsed += System.nanoTime() - start;
            rollbackAndContinue();
        }
        System.out.println("Script (100): " + totalElapsed / 1000000 + "ms");

        totalElapsed = 0;
        for (int i = 0; i < 100; i++) {
            long start = System.nanoTime();
            BusinessObjectDispatcher.execute(doc2, BusinessObjectActionLookup.constants.FAST_UPDATE);
            flush();

            totalElapsed += System.nanoTime() - start;

            rollbackAndContinue();

        }

        System.out.println("Java (100): " + totalElapsed / 1000000 + "ms");

}
```

Using PerformanceTestResult helpers

A performance helper suite of classes was introduced, allowing "shoot-out"s like the above to be more simple:

```


          Callable<Void> exprCallable = new Callable<Void>() {

            @Override

            public Void call() throws Exception {

                expression.value(context);

                return null;

            }

        };

        Callable<Void> javaCallable = new Callable<Void>() {

            @Override

            public Void call() throws Exception {

                function(x);

                return null;

            }

        };

        PerformanceTestCallable exprPerfCallable = new PerformanceTestCallable("Expression "

                + expression.getExpressionString(), exprCallable);

        PerformanceTestCallable javaPerfCallable = new PerformanceTestCallable("Java", javaCallable);


        PerformanceTestResult compareResult = PerformanceTestHelper.compare(20, 200000, exprPerfCallable,

                javaPerfCallable);

        compareResult.printResults();
```

The API is com.splwg.base.api.testers.performance.PerformanceTestHelper:

```

    public static PerformanceTestResult compare(int warmups, int reps, PerformanceTestCallable... callables)
```

```
            throws Exception {
```

Each of the performance callables is treated the same. It gets a series of warmup executions, in order to populate caches, and allow hotspot JVM optimizations of any methods. Then the accurate system nano timing (e.g., System.nanoTime()) is called around the loop of the given number of reps.

Profiling

The code below uses YourKit's controll classes to create a snapshot.

```
    public void testProfilePluginScripts() throws Exception {
        String docString = "<DR_ShortCreateIntervalRecords><factId>219250542869</factId><longDescr>REEE</longDescr></DR
_ShortCreateIntervalRecords>";
        Document doc = DocumentHelper.parseText(docString);

        // warmups
        for (int i = 0; i < 5; i++) {
            BusinessObjectDispatcher.execute(doc, BusinessObjectActionLookup.constants.FAST_UPDATE);
            rollbackAndContinue();
        }

        Controller controller = new Controller();
        controller.forceGC();
        controller.startCPUProfiling(ProfilingModes.CPU_SAMPLING, Controller.DEFAULT_FILTERS);
        for (int i = 0; i < 500; i++) {
            BusinessObjectDispatcher.execute(doc, BusinessObjectActionLookup.constants.FAST_UPDATE);
            rollbackAndContinue();
        }
        controller.captureSnapshot(ProfilingModes.SNAPSHOT_WITHOUT_HEAP);
    }
```

PerformanceTestHelper API

As before, the PerformanceTestHelper helps by providing a seamless interface into the yourkit profiler, for various options of sampling, tracing, monitoring threads or timing in threads:

```
    public static PerformanceTestCallableResult profileSample(int warmups, int reps, PerformanceTestCallable callable)
```

```
        throws Exception {
public static PerformanceTestCallableResult profileTrace(int warmups, int reps, PerformanceTestCallable callable)
        throws Exception {

public static PerformanceTestCallableResult monitor(int warmups, int reps, int numThreads,
        PerformanceTestCallable callable) throws Exception {

public static PerformanceTestCallableResult timeInThreads(int warmups, int reps, int numThreads,
        PerformanceTestCallable callable) throws Exception {
```

The PerformanceTestHelper utility class uses reflection to know whether the yourkit library is available or not. If it is not available (such as on the build server), the behavior reverts to simple timing protocols of the corresponding callable iterations. If it is available (such as on a developer's workstation, and they want to profile a test), then the yourkit profiler is connected to. This would require actually running the test under a profile session, else an error is produced.

Profiling a callable is somewhat similar to the simple timing of a callable, except for some added steps:

1. Performs some warmups
2. Forces garbage collection via the yourkit API
3. Starts the given profile type (sample vs trace)- the test should be run without automatically starting the profiler
4. Wrap the repetition loop in a timer
5. Capture a snapshot

This design approach allows profile/performance tests to be checked into version control, for re-profiling at a later point, and for documentation examples of how to profile code, etc.

References and Additional Resources

Batch programs and strategies:

For details on writing batch programs, using threads for performance improvements, and other batch-related information, see the *Batch Best Practices* whitepaper available on the My Oracle Support (MOS) Knowledge Base (article 836362.1).

Hibernate fetching strategies:

http://docs.jboss.org/hibernate/stable/core/reference/en/html_single/#performance-fetching

Yourkit profiling demo:

http://www.yourkit.com/docs/demo/JavaEE/JavaEE.htm

Packaging Guide

CM Packaging Utilities Cookbook

This document describes the installation, configuration, and operation of the packaging utilities provided with the Software Development Kit. These utilities enable developers to prepare releases of their custom modifications, called CM releases, to the products. Releases prepared using these utilities may be installed on top of an existing base product environment.

Note:
CM releases will correspond to a specific base product version and can only be installed on base product environments of that version. Customers installing a CM release must first verify the corresponding base product version with the Implementation team.

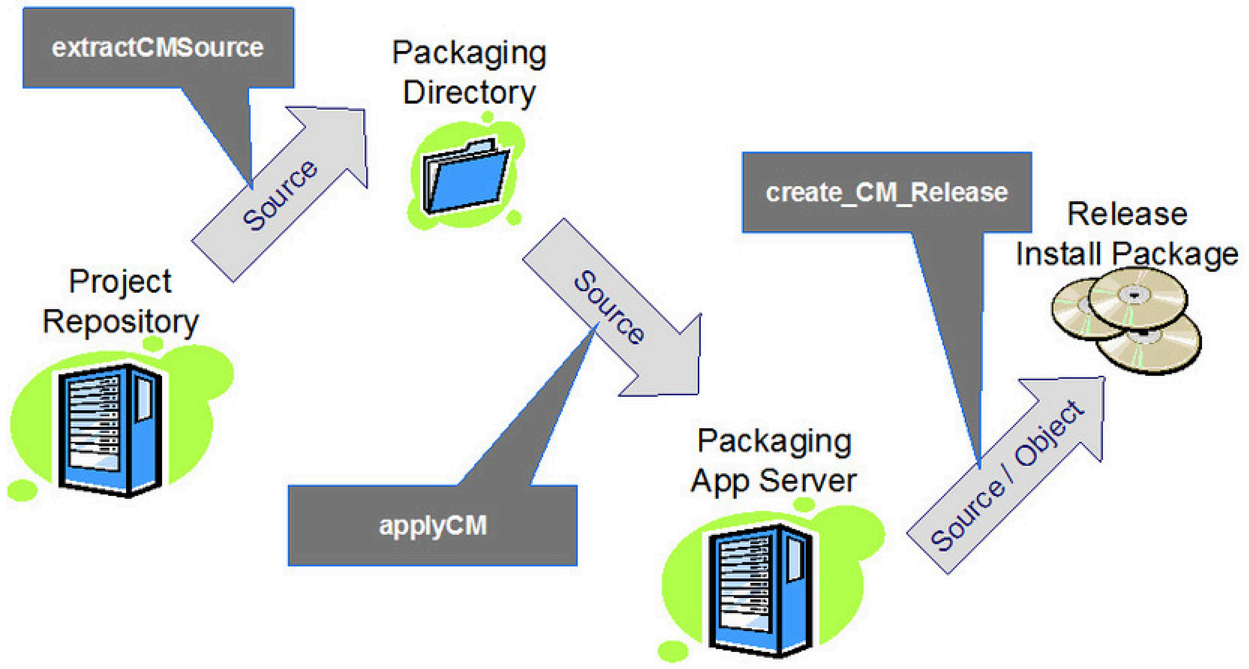
Note:
This document describes CM packaging utilities operation for Oracle database only. The application server can be Unix or Windows/DOS operating system. In Unix you must execute the script with **.sh** suffix, in window the script with **.cmd** suffix. They both will execute the same Perl script with **.plx** suffix. For instance:

- **applyCM.sh**: Unix driver script
- **applyCM.cmd**: Windows driver script
- **applyCM.plx**: Perl script

All the examples in this document are related to Unix. If you are in Windows/DOS simply execute the same scripts, but using **.cmd** extension instead **.sh**.

App Server CM Packaging Overview

The following diagrams describe the app server CM packaging procedure.

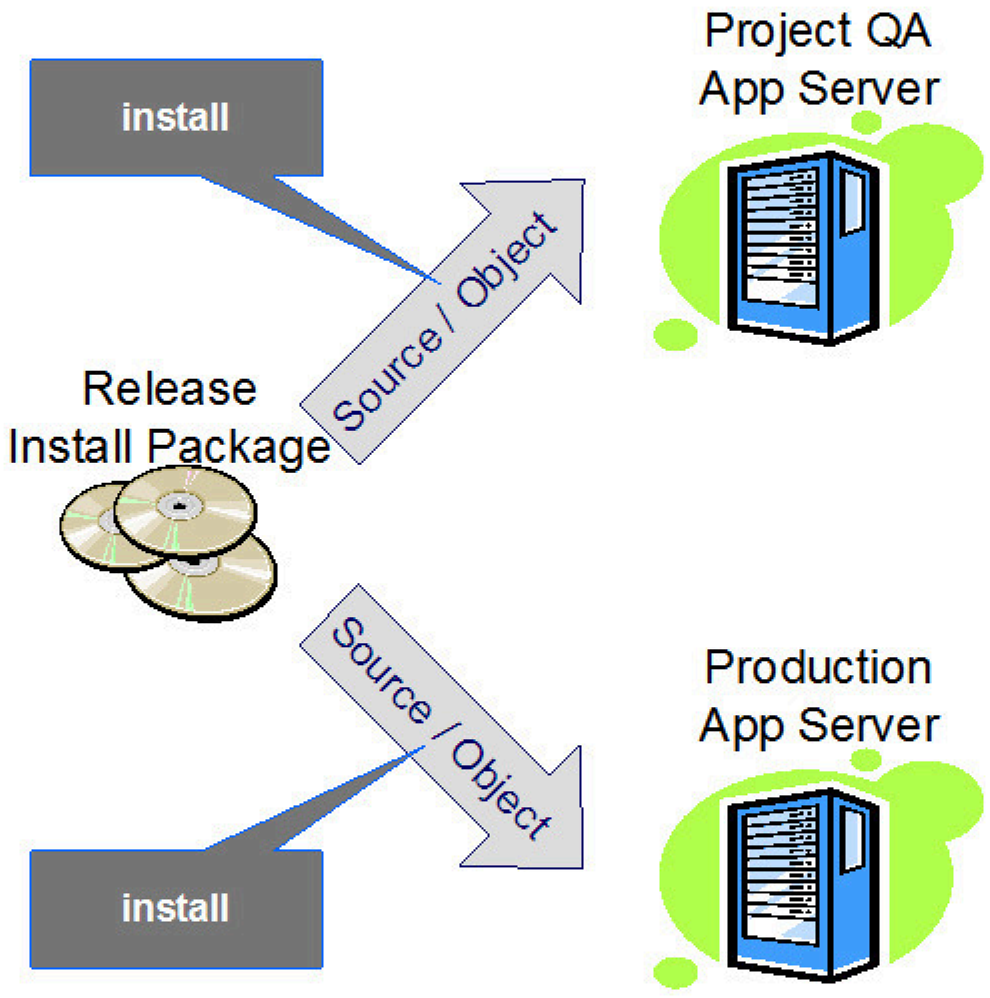


The starting point of packaging the app server component is the **project repository**. The tool **extractCMSource** is used to get the source from the project repository into the **packaging directory**.

Note:
The packaging directory must not be used for any other purpose except for storing the extracts. Mixing other files into the packaging directory will result in errors in succeeding processes.

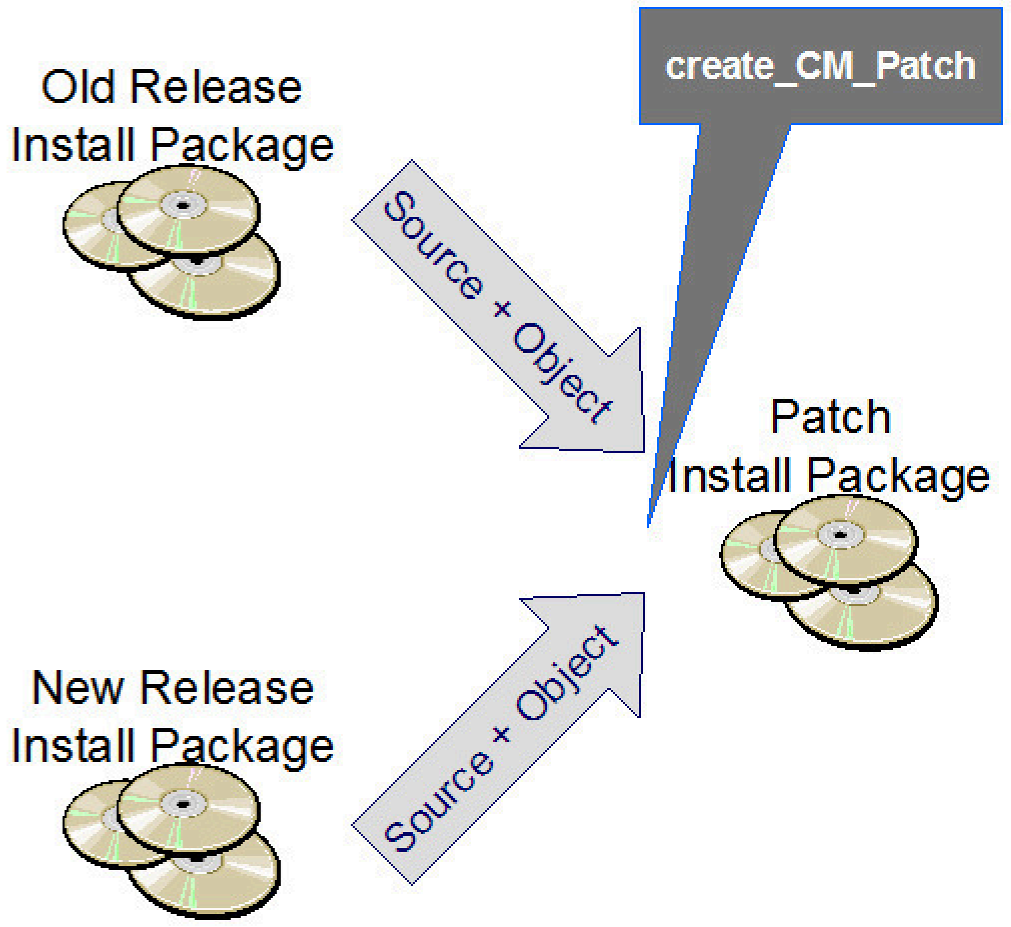
applyCM copies the extracted source to the **packaging app server**. It then does all the necessary steps like generate, compile, etc., to update the packaging app server runtime based on the extracted source.

create_CM_Release is then used to create CM **release install package** from the packaging app server. The CM release install package contains all CM code that has been applied to the packaging app server.



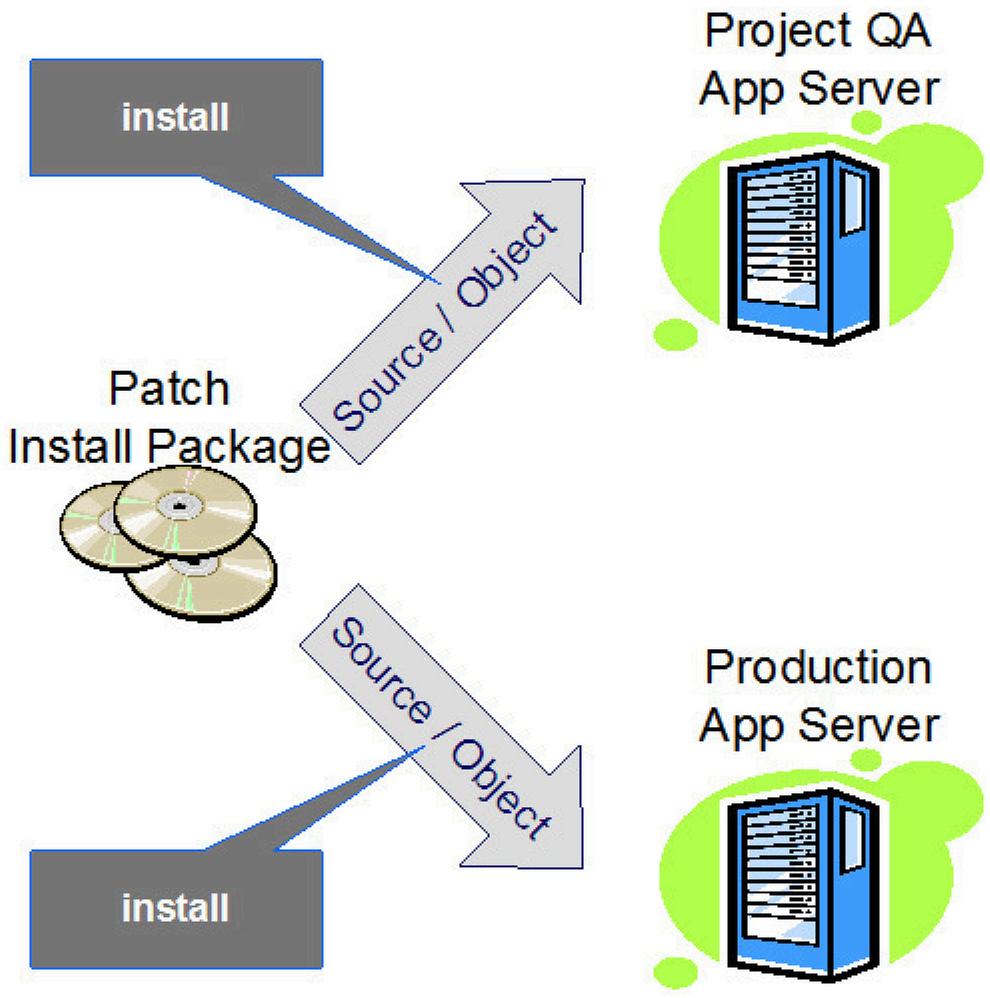
The **install** tool applies the CM release install files to either **QA** or **production app servers**.

Note:
Release install packages are usually applied only to fresh environments, e.g., to apply the first batch of CM code or when upgrading to a new version of the product. To install additional code to an existing environment, patch install packages (described next) are used.



CM patch install packages are used to update an existing installation with the changes since it was last updated. A patch install package is created by **create_CM_Patch** as the difference between two CM release install packages (a newer one and an older one), e.g., for a monthly update schedule, CM release install packages are created every month and every month, a cm patch install package is created using the release install packages from the previous and the current month.

To create a patch install package, a new release install package must be created first. Note that a release (not patch) install package must be available for the previous period. Executing create_CM_Patch with the two CM release install packages as input creates the patch install package.



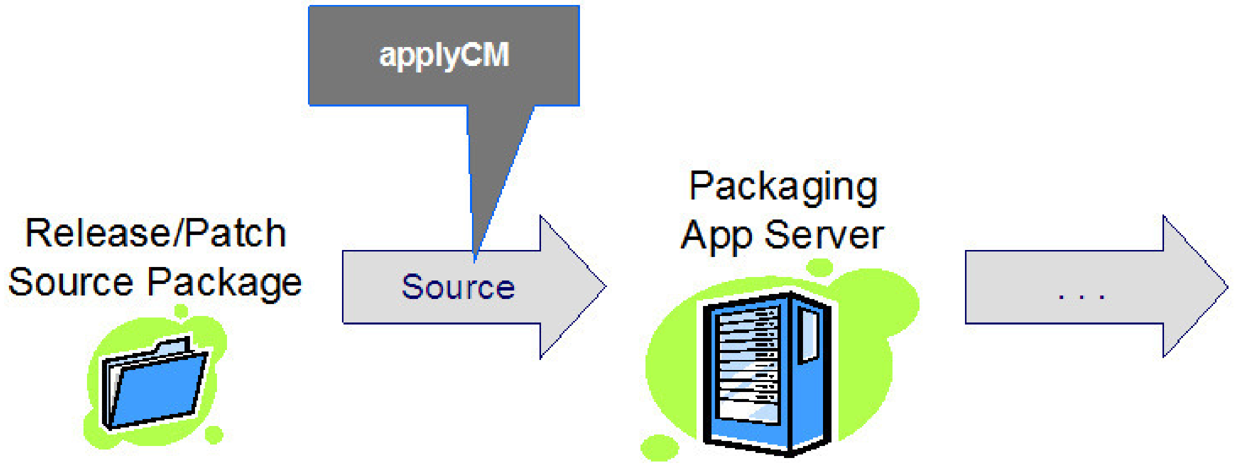
Developing Off-site

When developing off-site, there may be no available environments on the target platform. In this case, development and QA must be done off-site, but packaging and QA must be re-done on-site using environments on the target platform.

Off-site Process

Development and QA (and the packaging) is done using the same procedure as on-site except for the following:

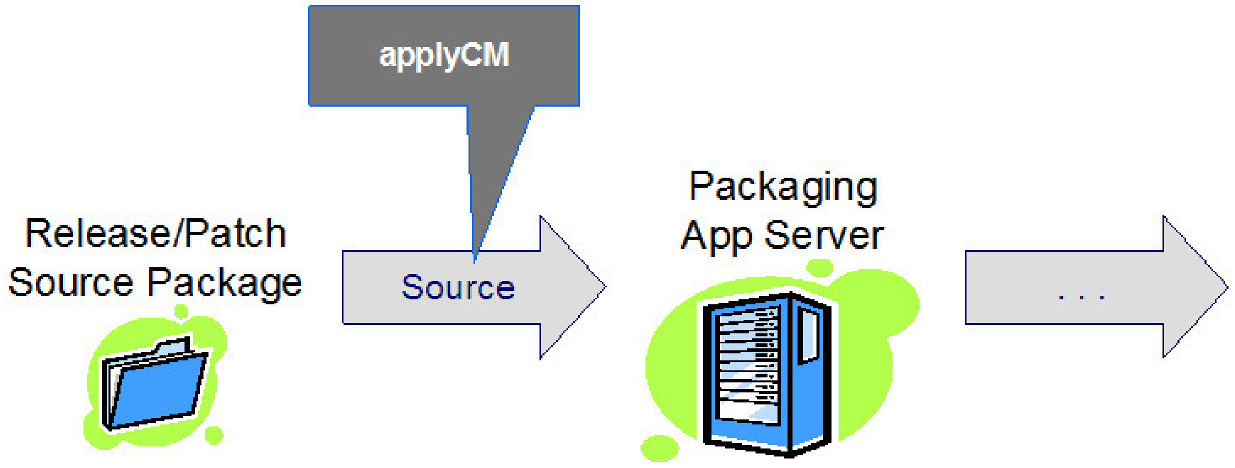
- Packaging and QA are done on environments that may not match the target platform.
- Instead of sending a release or patch install package to the implementation site, only the source from the install package is sent. This package is called the release/patch source package. It is created by executing extractCMSource with the **data** directory of the install package as the source directory.



Off-site Development Release

On-site Process

Upon receiving the release/patch source package, the on-site team proceeds with the regular packaging procedure starting from the applyCM step using the release/patch source package as the source directory.



On-site Packaging

Guidelines

Applying a CM patch install package to QA or production app servers is the same as for release install package, e.g., it is done simply by executing the install tool from the the package directory. By using these scripts, implementation developers can prepare an installation package containing the contents of their custom modifications. Developers need to build a packaging environment on a server of the same operating system platform as used by the target environment to create CM release packages. A version number must be used to identify each custom modifications (CM) release version. Once developers select a version number format, the version number must be stored on the environment in the file **$SPLEBASE/etc/CMVERSION.txt**, to achieve this place CMVERSION.txt file in etc subdirectory in your patch directory.

Implementers are strongly recommended to use CM packaging utilities for implementation delivery to customer site. It will ensure the correct installation complying with base product rules and will keep an environment upgradable. Please, note, that web files can be also packaged in archive war format (if **$isExpanded** environment variable is set to **false**), in that case it is not possible to just manually copy changes to the directories.

App Server CM Packaging Tools

The following utilities are provided in this package for maintaining the packaging environment and creating release versions of customer modifications (CM):

extractCMSource utility, used to extract source from an app server or from a release or patch install package.

applyCM utility, used to apply a CM patch to a packaging environment.

create_CM_Release utility, used to create a full CM release version.

create_CM_Patch utility, used to create CM patch release version utility.

Instructions for using these utilities are described in the following sections.

Post Install Setup

After the CM Packaging Tools installation, it is required to copy the proper **spl-tools-<VERSION>.jar** to the actual jar location, e.g.:

cp <CM script dir>/tools/spl-tools-4.5.0.jar <CM script dir>/lib/

Using the extractCMSource.plx Utility

This utility written in perl extracts source code from an app server, typically the project repository, or from a release or patch install package.

Note:

extractCMSource.plx is a Windows-only utility.

Display Usage

To display the usage information, execute the utility without any parameters from the Windows command prompt.

```
perl extractCMSource.plx

 extractCMSource.plx -s sourceDirectory -d destinationDirectory -v Version
         [ -n subDirectoryName ] [-m CmOwner ]
    -v Version
        The Version Number to attach to this release of
        the Customer Modifications.
    -s sourceDirectory
        sourceDirectory is the location to extract from.
    -d destinationDirectory
        destinationDirectory is the location in which the extracted files
        are placed. The program then creates a subdirectory under the
        destinationDirectory to hold the patch.
    -n subDirectoryName
        Subdirectory under "Directory" in which this patch will be placed.
        If this parameter is not provided, an automatic directory name
        is generated based on the environment name and date/time.
    -m CmOwner
        Used for a Multi-CM jar enhancement, this optional parameter
        specifies the CM owner that needs to be extracted from the
        project directory (e.g., "cm01"). If not specified, the script
        uses the default "cm".
```

Extract From an App Server

To extract the source from a development app server, specify the app server directory as the source directory. For example, the following invocation extracts source from an app server named CM_PROJ1 in the C:\SPL directory into C:\CMExtarct and marks the extract as version CM1.0:

```
perl extractCMSource.plx -s C:\SPL\CM_PROJ1 -d C:\CMExtract -v CM1.0
```

Extract From Release/Patch Install Package

To extract the source from a release or patch install package, specify the **data** directory in the install package as the source directory. For example, the following invocation extracts source from a patch install package named CM1.0_1 in the C:\CMInstall directory into C:\CMExtract and marks the extract as version CM1.0_1:

```
perl extractCMSource.plx -s C:\CMInstall\CM1.0_1\CMCCB\data

     -d C:\CMExtract -v CM1.0_1
```

FW Utility to extract CM sources from Unix environments

The following utility, delivered with the Oracle Utility Application Framework, is to extract CM sources from a Unix environment (note that **extractCMSource.plx** is the Window only utility):

$SPLEBASE/bin/extractCMSourceUNIX.sh

Usage:

-v <version>

The Version Number to attach to this release of the Customer Modifications. (Example: CM001)

-t <target directory>

Target Direcory is where the extracted files will be placed. This program will then create a subdirectory under that directory for the patch with a timestamp. with a timestamp.

Using the applyCM Utility

After an Implementation team has completed CM development on a Windows server or prepared a fix in a development environment, they'll need to copy and apply the CM modules to a packaging environment on the same platform as the target (e.g. production, testing). In other words, if the target system is a Unix platform, the packaging environment must be on Unix as well.

The **applyCM.sh** utility (**applyCM.cmd** for Windows installations) serves this need. It can be used to copy and apply all CM development modules to a packaging environment or any specific extract (patch) of CM development. The script needs to be executed using the full pathname (this is necessary because you need to be located in a different folder, see below). In addition, you need to be set to a target environment (e.g. packaging environment).

Script: **<CM scripts>/applyCM.sh**

Usage:

(no options)

Apply patch on top of the existing base product and possibly CM integration environment. This mode will add new CM files from patch to the environment and replace the changed ones. But it will not delete the previously existing in the environment CM modules that are not part of the patch.

It needs to be executed from the source root folder.

-d

Remove all previous CM modules from the environment and apply patch on clean base product environment. This option is useful when needed to create the CM integration environment from scratch.

It needs to be executed from the source root folder.

-b

Recompile the existing sources in current environemnt. Usually it is used to execute full recompile a development environment.

It needs to be executed from the application folder root, e.g. $SPLEBASE.

-n

It won't stop/start automatically the target environment.

The input for **applyCM.sh** utility is current folder (source root folder), which contains the following subfolders:

- java
- scripts
- etc
- services
- splapp

These subdirectories contain only CM modules created according to the rules of the document "Naming conventions for tailoring application implemetation" (see Installation Guide of the product). This directory structure should be prepared and filled with relevant CM modules on development Windows server, then copied over to the server that hosts a packaging environment by ftp utility. After that you can apply the patch to the packaging environment. **Modules that are not created using these conventions will be ignored by applyCM.sh utility.** You have to reside in the patch directory to apply the patch. ApplyCM utility will generate and compile java code, will create java jar file (cm.jar) required for customer implementation platform.

Using the create_CM_Release Utility

The **create_CM_Release.sh** utility is used to create a CM full release package that will contain only customer modification (CM) files. This is used to install a full set of customizations on top of the base product environment.

In order to build a CM release version that is compatible with the target platform, you need a packaging environment on the same operating system as the one on which the receiving product is installed. The target environment for installing the release version on a customer site can either be a pure base product environment, or an environment that already contains previous CM versions. In the second case, all previous CM modules will be removed by the **install** utility at the beginning of the installation process.

It is **mandatory** that every implementation version is identified by **its own release version number.** This number may be in any free standard and must be recorded in the **$SPLEBASE/etc/CMVERSION.txt** file on the environment.

Here are the detailed instructions for creating the full release version for CMs:

- Log in to the server with the administrator user id and initialize a packaging environment. You will use this environment to create the CM release version.
- Change the directory to the directory that contains the Developers Tool Suite utilities (**CM_ packaging**).
- Execute the utility using the following command:

```
./create_CM_Release.sh -e $SPLENVIRON -v $VERNO -d $RELEASEHOME,
```

where

$SPLENVIRON is the target packaging environment

$VERNO is the CM version number (the content of the file **CMVERSION.txt**)

$RELEASEHOME is the name of the directory on the server where you want to place the resulting CM release package.

For example:

./create_CM_Release.sh -e M4_Q1_SUNDB2 -v M.4.0.0 -d /versions

Tar and zip the resulting CM release directory for Unix platform or zip it for Windows platform and ship it to your customer.

The customer who wishes to install the delivered package onsite will follow the instructions:

- Decompress and untar the installation media to a temporary directory for the Unix server or unzip it for Windows server.
- Change directory to the target directory.
- Login and initialize the target environment.
- Change to the Installation directory using the following command

  ```
  cd CMCCB.$VERNO
  ```

  where

  $VERNO is the version number (the content of the file **CMVERSION.txt**)
- Run the following script

  ./install.sh - for Unix

  install.cmd - for Windows

Using the create_CM_Patch Utility

The utility **create_CM_Patch.sh** is used to create a patch release of CMs. A patch release version is created as a difference between a previous CM version and a new CM version. This type of release may be useful if the implementation team wants to ship only an update of the previously released version by preparing a smaller package that can be delivered easily by email or ftp to the customer.

Fastpath:
Before executing the utility, be sure that both packages are available in the same directory on the server. During the installation process at the customer site, the patch install utility will not remove the previous version of CM modules, and will only install the patch content on top of the previous CM version.

Here is the process for creating a patch release CM version:

- Log in to the server with the administrator user id.
- Change directory to the name of the directory that contains the SDK packaging utilities (**CM_packaging**).
- Execute the utility by entering the following command:

  ```
  ./create_CM_Patch.sh -d $RELEASEHOME
  ```

  where:

  $RELEASEHOME is the directory that currently holds your CM full release packages - and where you also want to put your new patch package.

Tar and zip the resulting CM patch directory for Unix platform or zip it for Windows platform and ship it to your customer. The customer who wishes to install the delivered package onsite will follow the instructions:

- Decompress and untar the installation media (on Unix) or unzip (on Windows) to a temporary directory.
- Change directory to that directory.
- Login and initialize the target environment.
- Change to the Installation directory by using the following command

  ```
  cd CMCCB.$VERNO
  ```

  where

  $VERNO is the version number (the content of the file **CMVERSION.txt**)

- Run the following script

  ./install.sh - on Unix

  install.cmd - on Windows

Multi-CM Application Functionality

The Multi-CM Jar functionality allows multiple teams to develop features of the application separately.

Each team is identified by a CM application owner. For example, a team in one geographic location may be identified by CM application owner **cm01**, whereas another team will be **cm02**.

Teams can have independent project repositories, packaging directories, packaging app servers, releases, and patches.

The different CM application releases/patches are then installed into a target QA or production app server so that they contain all cm applications, e.g., a target environment will contain *cm01*, *cm02*, *cm3*, etc.

The Multi-CM application is activated by using the "–m" option when running `extractCMSource.plx`. With this option, only the source code for the given CM application owner is extracted from the project repository. It creates a file (`etc/cm_owner.txt`) in the packaging directory identifying the CM application owner.

If the cm_owner.txt file exists, all subsequent utilities, namely, applyCM, create_CM_release, create_CM_patch, and install operate using the designated CM application owner. The installation of the final CM package removes the previously-installed CM application owner's files and modules, replaces them with the ones from the installation package, and, finally, appends the owner to the existing cm_owner.txt file in the target environment.

If the Multi-CM application is activated, the applyCM script applies to the packaging environment all the jar files extracted from development (excluding <CmOwner>.jar, which is compiled and created by applyCM). Such jar files are added to the Classpath for the Java compilation. The create_CM_release, in case the Multi-CM application is activated, copies only the <CmOwner>.jar to the installable package:

```
perl extractCMSource.plx -s C:\SPL\CM_PROJ1 -d C:\CMExtract -v CM1.0 –m cm01
```

Each CM owner development requires separate CM jar structures, e.g., `structures/cm01_jars_structure.xml`.

Each CM owner development can also contain customized user exits, and if there are additional templates, each must be defined in the relevant structure, e.g.:

```
templates/cm01_web.xml.exit_end.include

structures/cm01_template_structure.xml
```

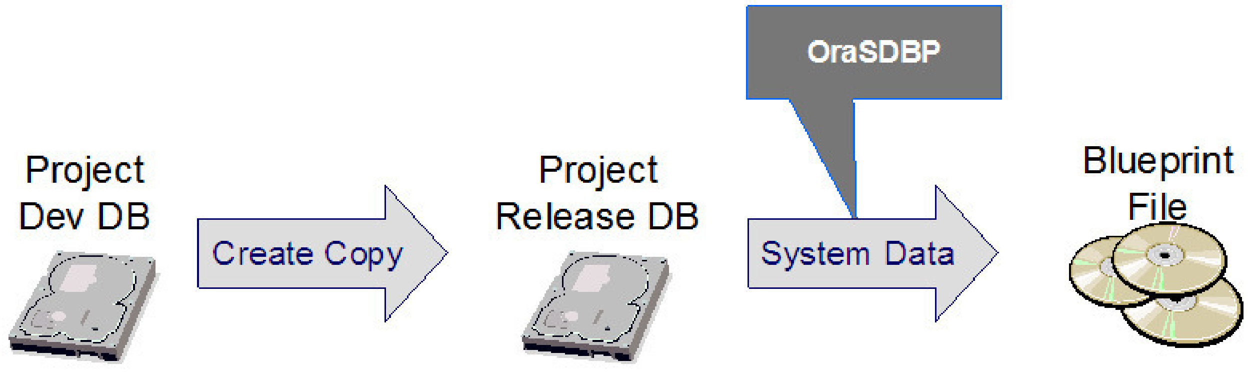
CM System Data Packaging Tools

CM System Data Packaging Overview

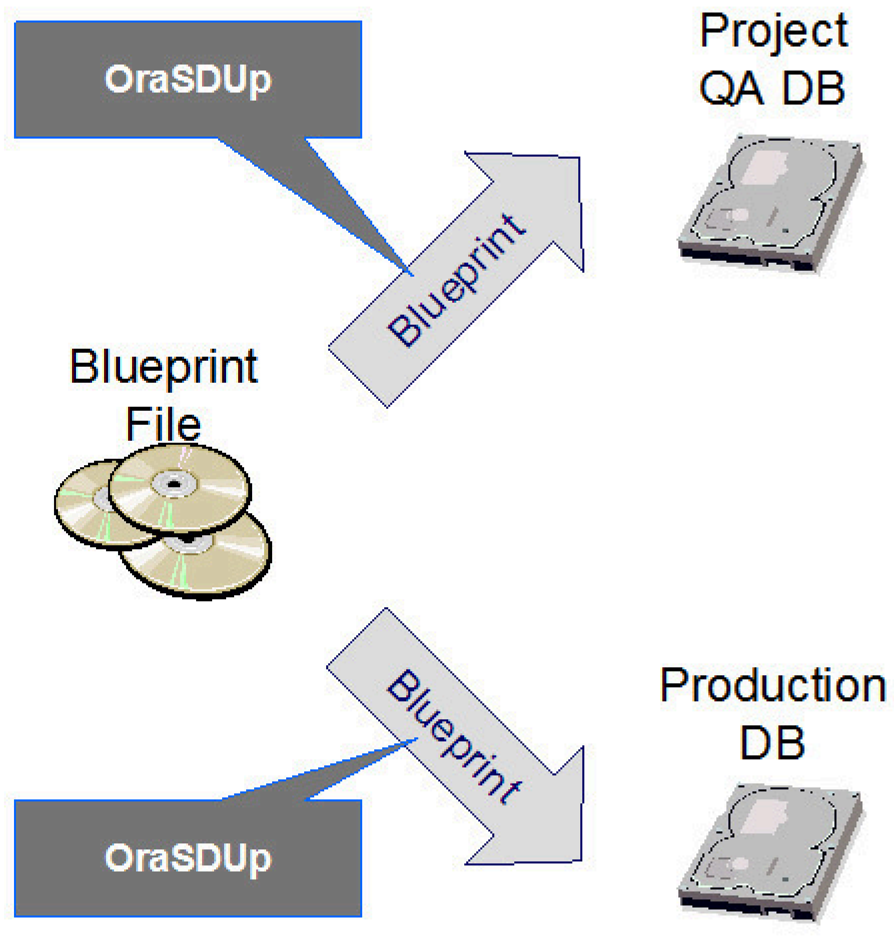
CM System Data Packaging Tools allow implementers to extract and package Customer Modification ('CM') system data from their databases and deliver it to their customers.

The following example uses the Oracle database platform to illustrate the extract and upload process.

As a standard release process, the implementers can add the CM system and Meta data records to the base product database or change base product system and Meta data records according to the specific rules (see "System Table Guide" document in the Installation Guide of the product). Implementers can then choose to migrate the CM data to a customer database as a full extract of CM data, or incremental differences between the current version of the system data on the customer site and the new version of the implementation development database.



Packaging CM system data starts by creating a copy of the **project dev database** into a **project release database**. A **blueprint file** of the system data is then created by running the **GenSysdataBP** java utility from OraGenBP.jar.



To apply the latest changes to a **QA** or **production database**, run **OraSDUpg** java utility in OraDBI.jar with the blueprint file as input and then specifying the target database.

Extract Process

Extract process involves extracting CM system data based on the rules defined in a parameter file and packaging it in a binary file - blueprint. This file can then be used as an input source by the data upload process.

The following example uses the Oracle database platform to illustrate the extract and upload process.

GenSysdataBP java utility in OraGenBP.jar, included in this package, is the extract utility that reads an input parameter file for the list of Oracle database tables, extracts data from these tables and compresses into a binary file (blueprint).

A sample extract parameter file extract_cmsystbls.lst is included in this package to provide the implementers a starting point. This parameter file, as can be seen, defines rules for the tool to extract

CM data based on their key definition. However, in some cases, CM data may be stored on 'CI' rows. The column user_exit_pgm on CI_MD_PRG_COM table is one such case. For cases like these, the implementers can choose to change the extraction rules in this file to match their requirements.

To extract your data, make a copy of extract_cmsystbls.lst file and edit it to match your requirements. Execute the extract process from a Command-window and provide it with the required parameter when prompted.

The data in input parameter must match the following format:

CI_ALG_TYPE_L;LANGUAGE_CD = 'ENG';VERSION

Where, the first field stores the table name, second field stores the selection criteria (where clause for selecting data) and the third field stores the list of column that should be ignored during extraction. The character semi-colon is used as the field separator. If there are multiple columns that need to be ignored (not included in the data being extracted), comma can be used in the third field as the separator.

GenSysdataBP accepts the following parameters:

- -d Connect String

  Where the Connect String contains:
- Schema owner name (say CISADM)
- Password for schema owner.
- Database name.

  This is a mandatory parameter. If not entered, the utility will prompt the user to build the connect string.

  Connect String should be entered in the following format:

  CISADM,CISADMPSWD,DBNAME

  (Comma-separated and no space).
- -i Input Parameter file name.

  Name of the input parameter file that the utility reads to get the list of tables and their selection criteria. This parameter is optional. The default input parameter file name is CDXSdBp.Inp.
- -o Output File Name.

  This is the name of the binary file that the utility creates. This parameter is optional. The default output file name is "OraSdBp" (without extension).

- -c NLS characterset of the target database

  The utility uses this parameter to set the NLS_LANG parameter on the client side. This parameter is then validated against the character set of the source database and is saved in the blueprint. This is mandatory parameter and is prompted for if not set by the user.
- -s

  Specify to generate object level SQL files. Optional.
- For help (to list all the accepted parameters with a brief description), execute the utility without any parameters.

Upload Process

Data upload process compares the data included in the blueprint file (generated by the extract process) and that extracted from a target database and generates output SQL to synchronize them.

The java utility OraSdUpg in OraDBI.jar, included in this package, is used by the process utility to compare and synchronize the data in the target database with that in the input blueprint file.

OraSdUpg reads an input parameter file for the list of the tables to be upgraded along with the selection criteria and upgrade rules for each table.

Each table has a corresponding record in the file with following 6 fields separated by semi-colon:

- Table Name
- The instance of the table. This number should be always set to 1. The cases where more than one instances of a table are processed are extremely rare and are not discussed here.
- Selection Criteria for the table.
- Insert allowed indicator (T/F): Whether records should be inserted into the target database table if they missing in the database but exist in the binary file.
- Update allowed indicator (T/F): Whether records should be updated in the table if they have different values than in the binary file.
- Delete allowed indicator (T/F): Whether the obsolete data in table in the target database. Obsolete records exist in target database but not in the binary file.
- Fresh Install Indicator (T/F): Whether the table should be seeded during the very first install. This indicator is only used when the utility is invoked with "-f" switch.
- List of columns not updated can be specified in the sixth field. Use a comma to separate the column names if multiple columns are to be ignored during updates. These columns will be inserted but will not be updated during the data synchronization process.

Following is the example of how these records should look like in the file:

CI_LOOKUP;1;LANGUAGE_CD = 'ENG';T;T;F;T;DESCR

A sample file upload_cmsystbls.lst has been included in this package. Implementer can make a copy of this file and edit it to match their requirements.

Before making connection to the target database, the utility reads the header from the blueprint and sets NLS_LANG environment variable on the client machine. It then validates this character set setting to the character set of the target db after making a connection and warns user if there is a mismatch.

The utility can be executed in verification and modification modes. In verification modes, the action SQL statements are simply written to the log file but in modification mode they are applied the target database.

It is very important to note that the primary requirement for OraSdUpg is definition (column and primary key) of tables being upgrade in the target database should be same as that in the database from which the binary file was extracted.

Be careful while selecting the table and the selection criteria because to compare the data, the utility, for each table, first loads the data from the binary file and the database in the memory. If a table has huge amount of data and selection criteria set causes the utility to work on large quantity of data, it may run out of memory.

To avert unique key constraint violation error that can be caused by improper sequence of data deletion and insertion on a table and also the foreign key issues, the utility first gathers all the generated action statements for all the tables before executing them. The execution of all the generated statements is done in multiple iterations. After each iteration, all the failed statements during that iteration are collected and executed again in the next iteration. The iterations are repeated till either all the statements in iteration are executed successfully or they fail.

The utility disables and enables all the triggers on the tables being upgraded before and after applying database changes. No triggers get executed during the system data upgrade.

OraSdUpg accepts the following parameters:

- -d Connect String

  Where the Connect String contains:

- Schema owner name (say CISADM)
- Password for schema owner.

- Database name.

  This is a mandatory parameter. If not entered, the utility will prompt the user to build the connect string.

  Connect String should be entered in the following format:

  CISADM,CISADMPSWD,DBNAME

  (Comma-separated and no space).
- -b Bypass the database character set validation.

  Before upgrading data in database, the utility validates character set stored in the blueprint by OraSDBp against that of target database. The user can bypass this validation step by setting this switch.
- -p Input Parameter file name.

  Name of the input parameter file that the utility reads to get the list of tables and their selection criteria. This parameter is mandatory.
- -i Input Binary File.

  This is the name of the binary file that the utility reads to extract the data that it then uses to upgrade the target database. This is a mandatory parameter.
- -f

  Treats the data synchronize process as New install. When set, the flag forces OraSdUpg to use "fresh install indicator" for the tables where INSERT indicator is set to false and compels it to insert missing records in all of them. Optional.
- -u

  Makes OraSdUpg run in the modification mode. Optional.
- - q

  Provide to run in silent mode (non-interactive mode). Optional.
- -m

  If it is provided the DB triggers are not disabled. Optional.
- -l Log File Name.

This is the name of the file that OraSdUpg creates, if it is missing and starts appending the information about the action it is performing.

- -h Help.

  This option will list all the accepted parameters with a brief description.

Note:
It is recommend that the implementers execute the upload process first in the verification mode and review the SQL before running the tool in the modification mode.

Tailoring Your Oracle Utilities Application Implementation

This document describes the naming conventions and processes that must be followed to ensure a successful upgrade of the Oracle Utilities application base product release-on-release. The implementation team responsible for tailoring the Oracle Utilities application to meet specific customer needs must follow this guide to preserve their changes and ensure successful upgrades. Only the changes described in this document are considered as permitted for the tailoring of the base product. Any changes that do not conform to these rules may be overridden by the install utility during a base product upgrade.

Some naming conventions used in this document:

- `$SPLEBASE` (for UNIX) and `%SPLEBASE%` (for Windows) is the generic Oracle Utilities environment directory name.
- `$SPLENVIRON` (for UNIX) and `%SPLEBASE%` (for Windows) is the generic Oracle Utilities environment name.
- `$SPLDB` (for UNIX) and `%SPLDB%` (for Windows) is the database type.

Preserving Customer Changes

For any kind of a customer modification, the file's directory structure and naming conventions are defined in this section. The implementation team must follow these conventions to preserve the results of their work during a subsequent base product upgrade.

- The configuration parameters of the environment being upgraded are displayed (as default parameters) during the configuration stage of the install process. These parameters may be changed if new settings are preferred.
- The base product is shipped with examples of different kinds of modules that may be used by implementation teams. The examples can be found in the following directories:
  - `$SPLEBASE/splapp/applications/root/cm_templates` contains Oracle Utilities Application Framework Web file examples.
  - `$SPLEBASE/splapp/applications/root/<application product code>/cm_templates`. This directory contains Oracle Utilities application product Web file examples. The <application product code> varies by product; for example, the Oracle Utilities Customer Care and Billing, the <application product code> is `c1`.
  - `$SPLEBASE/scripts/cm_examples`. For batch script examples, this directory has two subdirectories: `FW` for Oracle Utilities Application Framework examples, and <application product code> for Oracle Utilities application product examples (e.g., `CCB` for Oracle Utilities Customer Care and Billing, `TAX` for Oracle Public Sector Revenue Management).

Note:
For simplicity, this document generally uses UNIX platform naming conventions. To apply these names to the Windows platform, use the Windows naming conventions "%" sign instead of the "$" sign, and backslashes ("\") instead of forward slashes ("/") as directory separators (e.g., `%SPLEBASE%\splapp\applications\root\cm_templates`).

Tailoring Web Files

Base product UI files are located in the directory `$SPLEBASE/splapp/applications/root`. Implementers may develop their own UI files under the directory `$SPLEBASE/splapp/applications/root/cm`. No specific naming conventions are enforced under this directory.

The root directory may be deployed in war file format for runtime environment (`SPLApp.war`). Use provided utilities to incorporate your **cm** directory into the `SPLApp.war` file.

Tailoring the CM Java Application

Implementers may write their own Java classes to extend the Oracle Utilities application functionality. All Java files should belong to the `com.splwg.cm` package. The CM Java application should be compiled into a jar file named `cm.jar`. The SDK Customer Modification packaging utilities will help build this file. The `cm.jar` is typically deployed into the following directories:

```

```

```
$SPLEBASE/splapp/applications/root/WEB-INF/lib

$SPLEBASE/splapp/applications/XAIApp/WEB-INF/lib

$SPLEBASE/splapp/businessapp/lib

$SPLEBASE/splapp/standalone/lib
```

Additional third-party jar files can be deployed by following the `cm*.jar` naming standard. Customers may use this option to deploy any additional functionality, interfaces with other applications, and so on. These will not be built by the SDK Customer Modification packaging utilities, but will be deployed into the application once it is supplied in jar format.

The root directory may be deployed in a war file format for the runtime environment (`SPLApp.war`). Use the provided utilities to incorporate your `/cm` directory into `SPLApp.war` file.

Important:
All `cm*.jar` files that need to be applied must be defined in `$SPLEBASE/structures/cm_jars_structure.xml`. If the file does not exist in the target environment, the sample `cm_jars_structure.xml.example` file can be copied from the SDK packaging's `/etc` folder.

Manual cm.jar deployment

The `cm.jar` file is usually deployed as part of the CM packaging process (`extractCMsource`, `applyCM`, `create_CM_release`, etc.), but in some cases it may be desirable to manually deploy the `cm.jar` file to one or more target environments.

CAUTION:
This should be done with care and should only be considered if the `cm.jar` components are self-contained and have no external dependencies.

To manually deploy `cm.jar`:

1. The SPLEBASE/structures/cm_jars_structure.xml must exist and should have at least the following:

```
<?xml version="1.0" encoding="UTF-8"?>

<jar_structure>

   <cm.jar>

  <source_dir_jar>@SPLEBASE@/etc/lib</source_dir_jar>

  <dest_folders>
```

```
 <dest_folder_1>@SPLEBASE@/splapp/applications/XAIApp/WEB-INF/lib</dest_folder_1>

 <dest_folder_2>@SPLEBASE@/splapp/applications/root/WEB-INF/lib</dest_folder_2>

 <dest_folder_3>@SPLEBASE@/splapp/businessapp/lib</dest_folder_3>

 <dest_folder_4>@SPLEBASE@/splapp/standalone/lib</dest_folder_4>

  </dest_folders>

  <child_jvm_path>@SPLEBASE@/splapp/standalone/lib</child_jvm_path>

   </cm.jar>

</jar_structure>
```

The <cm.jar> element identifies the jar file name, usually `cm.jar`, as defined here.

Element <source_dir_jar> defines the source location of the abovementioned jar. The directory in the example above should work for most cases.

The `dest_folder_` *n* elements point to the target locations where the jar will be placed. The directories in this example should work for all.

2. Manually copy the `cm.jar` to the directory specified in the `<source_dir_jar>` element, typically `$SPLEBASE/etc/lib`.

3. Run initialSetup.sh (or .bat on Windows) to do the rest. This will copy the `cm.jar` to the specified target locations and rebuild the war and ear files.

Positioning Custom Scripts

Customers and implementers may put their scripts under the directory `$SPLEBASE/scripts/cm`.

Replacing the Oracle Utilities Logo

Customers may want to replace the Oracle Utilities logo image on the Main menu with another logo image. To do this, put the logo <customer_logo_file>.gif file into the directory `$SPLEBASE/etc/conf/root/cm` and create a new "External" Navigation Key called **CM_logoImage**.

To replace the logo, run the Oracle Utilities application from the browser with the parameters:

```
http://<hostname>:<port>/cis.jsp?utilities=true&tools=true
```

From the **Admin** menu, select **Navigation Key**. Add the above Navigation Key with its corresponding **URL Override** path.

The syntax for the URL path is:

For Windows:

```
http://<host name>:<port>/cm/<customer_logo_file>.gif
```

For UNIX:

```
http://<host name>:<port>/spl/cm/<customer_logo_file>.gif
```

The root directory may be deployed in war file format for the runtime environment (`SPLApp.war`). Use the provided utilities to incorporate your `cm` directory into the `SPLApp.war` file.

Using the Implementation Version File

Implementers may keep the implementation version number in the `CMVERSION.txt` file in the `$SPLEBASE/etc` directory. This file is preserved by the install utility.

Tailoring XML Schema

Note:
This implementation option is applicable for Oracle Enterprise Taxation Management application only.

Implementers may generate their own XML schemas and store them in the directory `$SPLEBASE/ splapp/ xmlMetaInfo`. The implementation schemas must use the naming convention `CML*.xml`.

Tailoring Templates and User Exits

The templates delivered under the folder `$SPLEBASE/templates` can be overridden by the Application by creating a copy of the template file with the same name but prefixed by "cm.". The cm copy will be customized.

Since the templates can contain user exits (special statements that allow to import external files during the template processing). Those user exits can be overridden by creating a copy of the user exit file with the same name but prefixed by "cm_". The cm copy will be customized.

Index