**Oracle® Communications
Network Integrity**

Developer's Guide

Release 7.3.2

E66034-01

May 2016

ORACLE®

Oracle Communications Network Integrity Developer's Guide, Release 7.3.2

E66034-01

Contents

2   Working with Actions

4 Working with Discrepancies

5 Working with the POMS SDK

6 Working with the Extensibility SDK

7 Working with Automatic Discrepancy Resolution

8 Working with Incremental TMF814 Discovery

9 Working with CPU Utilization-enabled Discovery

10 Working with the Network Integrity Web Service

11 Working with Scan Run Complete Notifications

12 Working with JCA Resource Adapters

13 Working with Reports Extensibility

14 Working with SOA Extensibility

15 Localizing Network Integrity

A Network Integrity Plug-in Validation Error Messages

Preface

This guide explains how to extend Oracle Communications Network Integrity through standard Java practices using Oracle Communications Design Studio, which is an Eclipse-based integrated development environment. This guide includes references to both applications, and often directs the reader to see the Design Studio Help and the Network Integrity Help for instructions on how to perform specific tasks.

This guide should be read after reading *Oracle Communications Network Integrity Concepts*, because this guide assumes that the reader has a conceptual understanding of Network Integrity. This guide should be read from start to finish because the information presented in a chapter often builds upon information presented in a preceding chapter.

This guide includes examples of typical development code used in given situations. The guidelines and examples may not be applicable in every situation.

Audience

This guide is intended for developers who implement code to extend Network Integrity. The developers should have a good working knowledge of XML and Java development and, in particular, JDO, standard Java practices, and J2EE principles.

You should read *Oracle Communications Network Integrity Concepts* before reading this guide.

You should have a good working knowledge of Design Studio.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at
http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support

Oracle customers have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Document Revision History

The following table lists the revision history for this guide:

Version	Date	Description
E66034-01	May 2016	Initial release.

1

Using Design Studio to Extend Network Integrity

This chapter provides information on Oracle Communications Design Studio, an Eclipse-based integration development environment. Design Studio comes with features specific to Oracle Communications Network Integrity that enable you to extend Network Integrity.

This chapter contains the following sections:

- Installing Design Studio
- About Design Studio Perspectives
- About Design Studio Views
- About Projects
- Working with Projects
- About Specifications
- Working with Specifications
- About Source Control
- Working with Source Control for Network Integrity
- Tips and Tricks

Installing Design Studio

Use Design Studio to extend Oracle products. Different features are available for the different Oracle features, and each feature provides JAR files that are unique to the product.

See *Design Studio Installation Guide* for information about installing Design Studio and the Design Studio for Network Integrity feature.

Configuring Design Studio for Network Integrity

Configuring Design Studio for Network Integrity requires:

- Configuring Network Integrity Preferences
- Network Integrity Project Dependencies
- Configuring Data Dictionary Preference Settings

Configuring Network Integrity Preferences

Configuring Network Integrity preferences in Design Studio includes specifying a default cartridge package name for all created cartridge projects and specifying the default MIB directory.

To configure Network Integrity preferences, see the Design Studio Modeling Network Integrity Help.

Network Integrity Project Dependencies

All Network Integrity cartridge projects have dependencies on several other Network Integrity cartridge projects. Before creating a new Network Integrity cartridge project or importing productized Network Integrity cartridge projects, import the following projects into Design Studio:

- ora_uim_model

- ora_uim_mds

- ora_ni_uim_ocim

- NetworkIntegritySDK: this cartridge project contains common software components and libraries required for creating and extending Network Integrity projects.

These projects are available in the Oracle Communications Network Integrity 7.3.2 Software Developer Kit (included with the Oracle Communications Network Integrity 7.3.2 software) on the Oracle software delivery website:

https://edelivery.oracle.com

See the Design Studio Help for information about importing projects into Design Studio.

Configuring Data Dictionary Preference Settings

You configure data dictionary preference settings to specify the horizontal depth to which any data dictionary tree can expand.

To configure data dictionary preferences, see the Design Studio Help.

About Design Studio Perspectives

Perspectives define your Workbench layout and provide different functionality for working with different types of resources. Several perspectives are available within Design Studio. The Java, Studio Design, and Studio Environment perspectives are commonly used when extending Network Integrity.

For instructions on how to open a perspective, see the Design Studio Help.

About Design Studio Views

Within a given perspective, views further define the Workbench layout and provide different presentations of resources. Several views are available within Design Studio, and the available views are dependent upon the perspective.

For instructions on how to open a view in Design Studio, see the Design Studio Help.

Studio Design Perspective Views

When extending Network Integrity in the Studio Design perspective, you commonly use the Studio Projects view, Solutions view, and the Package Explorer view.

See the Design Studio Help for more information about perspective views.

Java Perspective Views

When extending Network Integrity in the Java perspective, you commonly use the Navigator view, Package Explorer view, and Error Log view.

About Projects

Projects contain Network Integrity artifacts that you create and define in Design Studio, such as custom actions and processors.

Everything you create in Design Studio resides in a project. The name you choose for the project becomes the name of the integrity archive (IAR) file, and everything you create within that project is automatically placed in the IAR file.

When extending Network Integrity, you can create one or many projects, depending on how you choose to organize the extensions.

Network Integrity projects are packaged extensions to the core application. They represent the necessary components needed for the following:

- Discovering network elements, either from a Network Management System (NMS) or through direct contact with the Network Element (NE)
- Importing network elements from an inventory system
- Assimilating network data using business logic
- Detecting discrepancies between the network and the inventory system
- Resolving discrepancies, either within the network, or in the inventory system

Network Integrity projects provide the ability to support new functionality as business cases arise, such as:

- New protocols, such as Command Line Interface (CLI) and Transport Layer Security (TLS)
- New standards, such as a new RFC
- New vendor devices, such as Juniper, Huawei
- New operational or business support systems

See the Design Studio Help for more information about creating projects.

About the Project Architecture

A Network Integrity cartridge project typically contains the following entities:

- Zero or more actions:
  - Zero or more discovery actions
    - * At least one discovery, file transfer, or file parser processor
  - Zero or more assimilation actions
    - * At least one assimilation processor

- – Zero or more import actions

    * At least one import, file transfer, or file parser processor

- – Zero or more discrepancy detection actions

    * At least one discrepancy detection processor

- – Zero or more discrepancy resolution actions

    * At least one discrepancy resolution processor

- Zero or more model collections

- Zero or more specifications

- Zero or more scan parameter groups

Alternatively, your project can contain address handler entities. A project containing address handler entities cannot contain any other entity types. This allows for a clear segregation of responsibility. So, for example, you create a project called Address Handlers where different address handler types exist (for example: IP Address, URL, and so on) and simply reference those from within their discovery and import cartridge projects.

Projects can also reuse actions from other projects to extend behavior. For example, a Juniper-specific SNMP cartridge project (that is, containing Juniper MIBs) could extend a generic SNMP cartridge project (MIB II only).

After all components are defined, projects are packaged into an IAR file and can be deployed to a running Network Integrity system as a cartridge.

See "Building and Packaging Projects" and "Deploying and Undeploying Cartridges" for more information.

After a cartridge is deployed, it is available to Network Integrity.

To determine whether a cartridge is deployed in Network Integrity:

1. From the Network Integrity main menu, click **Help**, and then select **About**.

    The Network Integrity components dialog appears.

2. Select the **Components** tab.

    The Network Integrity product version is displayed with the versions of all cartridges deployed in Network Integrity.

Working with Projects

When working with projects, see the following:

- Building and Packaging Projects

- Deploying and Undeploying Cartridges

- Debugging and Testing Cartridges

- Sealing and Unsealing Projects

- Exporting and Importing Cartridges

Building and Packaging Projects

Design Studio packages project information into cartridges that can be deployed into Network Integrity.

Projects can be developed by customers, systems integrators, Professional Services staff, and third-party vendors.

About the Project Build Order

When Design Studio builds a Network Integrity project, the build process takes place in the following order:

- Generation of Java source code: Generators are invoked to generate Java source codes from Network Integrity models, EJB descriptor files, XML schemas for the SNMP processor, and the Meta Model XML file.

- Java Source Compilation: Eclipse compiles the Java source (including generated Java source and implemented Java source) into classes.

- Building: Builders are invoked to build UI hints, the Data Dictionary, and specifications.

- Validation: Validators are invoked to validate Network Integrity model entities. Validation errors are raised and an error marker displayed on the related entities in Design Studio. If any validation errors are raised, the packaging stage does not take place.

- Packaging: Packagers are invoked to package the cartridge deployment model XML file, the UI hints Metadata Archive (MAR) file, specification Data Access Object (DAO) files, dependent JAR files, the manifest file for JAR files library for EJB, and the final IAR file for the Network Integrity cartridge.

About Build Artifacts

Design Studio generates various build artifacts for a Network Integrity project after a successful build. The generated directories are listed in the following order in the directory structure:

- **Out**: This directory contains all the compiled Java classes.

- **Generated**: Contains the following build artifacts:

  - Generated Java sources for actions and processors. If the project is sealed without Java source, the JAR file is displayed instead.

  - SNMP schema artifacts for the SNMP processor.

- **cartridgeBuild**: contains various build artifacts for the Network Integrity cartridge.

- **cartridgeBin**: contains the final packaged Network Integrity cartridge as an IAR file which can be deployed to the Network Integrity server through the cartridge management web service (CMWS).

The following directories comprise the normal directory structure for a Network Integrity project. Do not modify these directories:

- **dataDictionary**: contains the Data dictionary

- **doc**: contains documents

- **lib**: Copy any third party JAR library into this directory.

  Switch to the **Packager Explorer** view, and modify the Java class path to include any JAR files that have been added to this directory.

  Select the project, and click **F5** to refresh the project in Design Studio to get the modified Java class path affected.

- **model**: contains all Network Integrity models
- **out**: output directory for compiled Java classes
- **resources**: contains resources related to Network Integrity. This directory is empty by default
- **src**: the Java source directory.

Packaging Projects

Packaging a project is the last stage in building a cartridge. The cartridge is packaged as an IAR file, which can be deployed to the Network Integrity server through the CMWS.

The IAR file contains the following build artifacts:

```
IAR root/
                <cartridge-ejb-jar>.jar - This jar contains manifest.mf file to
refer to the jars under cartridgeLib/<cartridgeName>.
                oracle.communications.platform.entity.impl.SpecificationDAO

oracle.communications.platform.entity.impl.CharacteristicSpecUsageDAO

oracle.communications.platform.entity.impl.CharacteristicSpecificationDAO
                <cartridgeName_A>.mar
                <cartridgeName_B>.mar
                …
                <cartridgeName_N>.mar - Multiple MAR files if this cartridge is
reusing Actions from other cartridges.
                <Action_Name_A>_MetaModel.xml
                <Action_Name_B>_MetaModel.xml
                …
                <Action_Name_N>_MetaModel.xml - Meta Model XML file per Action.
                /META-INF/
                                cartridge.xml
                                manifest.xml
                /cartridgeLib/<cartridgeName>/*.jar (any dependent jar files used
by this cartridge, if available)
```

If a project contains only abstract entities, no IAR file is generated.

Deploying and Undeploying Cartridges

Network Integrity cartridges can be directly deployed or undeployed from Design Studio.

Use the Oracle Cartridge Deployer to deploy or undeploy any productized Network Integrity cartridge into a production system.

> **Note:** Before deploying or undeploying cartridges, ensure that:
>
> - You are logged out of the WebLogic Server Administration Console.
> - No one else is deploying or undeploying cartridges on the same server.
> - Network Integrity is not running a scan that makes use of the cartridge.

Creating a Design Studio Environment Project

Design Studio projects are collections of folders and files that represent the content you are working on. They are used for builds, version management, sharing, and resource organization. Projects map to directories in the file system. When you create a project, you specify a location for it in the file system. Design Studio uses the files and folders in a project to build a cartridge that you can import into Network Integrity. See "Building and Packaging Projects" for more information. To deploy or undeploy a cartridge from Design Studio, you must first create a Studio Environment Project. When you create a project, you specify its name and location for its corresponding file structure.

See the Design Studio Help for more information on creating an environment project.

Creating a Design Studio Environment For Network Integrity

Having created a Studio Environment Project, you then create the environment. An environment represents a connection to a particular server.

See the Design Studio Modeling Network Integrity Help for more information about creating Design Studio environments.

When creating and working with your environment, consider the following:

- When specifying the name of your environment, incorporate the name of the server.

- If you are using SSL, the CMWS URL must be specified with **https**. Also, you must configure the Environment editor **SSL** tab with the location of the keystore file.

- Configure the Environment editor **Properties** tab for the following properties:

  - **wladmin.host.name**: The host name or IP address where the Oracle WebLogic Administration Server is running.

  - **wladmin.host.port**: The port number on which the Oracle WebLogic Administration Server is running.

  - **wladmin.server.name**: The Oracle WebLogic Administration Server name.

Deploying a Cartridge

The Design Studio Network Integrity feature provides the ability to deploy a cartridge into Network Integrity. For instructions on how to deploy a cartridge, see the Design Studio Help.

Undeploying a Cartridge

The Design Studio Network Integrity feature provides the ability to undeploy a cartridge into Network Integrity. For instructions on how to undeploy a cartridge, see the Design Studio Help.

When a cartridge is undeployed, Network Integrity removes all the scan configurations and scan results associated with the cartridge and all the specifications associated with the cartridge (except those specifications still in use by other cartridges).

If a cartridge has a dependency on other deployed cartridges, the cartridge cannot be undeployed. For example, you cannot undeploy the Address_Handlers cartridge if the cartridges using Address_Handlers are still deployed in Network Integrity. You must undeploy all dependent cartridges from Network Integrity before Address_Handlers can be undeployed.

The Network Integrity CMWS Adapter automatically performs dependency checks at deployment or undeployment time and returns error messages if deployment or undeployment cannot be performed.

Redeploying a Cartridge

The Design Studio Network Integrity feature provides the ability to deploy a cartridge into Network Integrity, including previously deployed cartridges. For instructions on how to deploy a cartridge, see the Design Studio Help.

You can redeploy a Network Integrity cartridge using Design Studio only if the version of the redeployed cartridge (build number) is equal to, or greater than, the version of the deployed cartridge. For example, *my_cartridge* is already deployed with a build number of 28 (b28). If *my_cartridge* is up-versioned to b30, you can deploy it without undeploying my_cartridge (b28) and deploying it again.

Redeployment removes the deployed cartridge and deploys the new cartridge instead. Network Integrity does not allow more than one version of the same cartridge to be deployed at the same time.

Debugging and Testing Cartridges

This section provides information about debugging and testing cartridges in Network Integrity.

Starting the WebLogic Server in Test Mode

To debug a deployed Network Integrity cartridge, start the WebLogic Managed Server in debug mode (not the Administration Server).

Use the following procedure to start the WebLogic Managed Server in debug mode:

1. Stop both the Administration Server and Managed Server if they are still running.

2. Go to directory **<WEBLOGIC_HOME>/user_projects/domains/<DOMAIN>/bin**.

3. Copy the existing **startWebLogic.sh** script to a new script file, **startWebLogic_Debug.sh**.

4. Use a text editor to open **startWebLogic_Debug.sh**.

5. After the line `${JAVA_HOME}/bin/java ${JAVA_VM} -version`, add the following two lines:

   ```
   echo "Launching Java with debug port: 10171"

   JAVA_OPTIONS="-Xdebug -Djava.compiler=NONE -Xnoagent -Xrunjdwp:transport=dt_
   socket,server=y,address=10171,suspend=n $JAVA_OPTIONS"
   ```

   The debug port does not have to be 10171 if the port specified is available.

6. Save this change.

7. Copy the existing **startManagedWebLogic.sh** script to a new script file, **startManagedWebLogic_Debug.sh**.

8. Use a text editor to open **startManagedWebLogic_Debug.sh**.

9. Find the two lines that are referring to **startWebLogic.sh**.

10. Replace **startWebLogic.sh** with **startWeblogic_Debug.sh**. This change is to start the WebLogic Managed Server in debug mode by invoking the **startWebLogic_Debug.sh** script.

11. Save this change.

12. Start the Administration Server by running the usual start-up script, **startWebLogic.sh**.

13. Start the Managed Server in debug mode by running the new script, **startManagedWebLogic_Debug.sh**.

Configuring Remote Debugger in Design Studio

The Managed Server is now in debug mode. The next step is to configure the debugger in Eclipse to start remote-debugging the Network Integrity cartridges.

1. From the **Design Studio** main menu, select **Run** then **Debug Configurations**, then open the **Debug Configurations** dialog to switch Design Studio to the **Java** perspective.

2. From the left panel, select **Remote Java Application**.

3. Click **New** to create a remote Java application debug configuration.

4. Enter a name for this new debug configuration.

5. In the **Connect** tab, click the **Browse**.

6. Select an available project that contains the cartridge that to debug.

7. Ensure that the default setting for **Connection Type** is **Standard (Socket Attach)**.

8. Enter the **host IP address** where the Network Integrity system (WebLogic Managed Server) is running.

9. Enter the **debug port**, which should match the debug port entered in "Starting the WebLogic Server in Test Mode".

10. Keep the default settings for the rest of the tab.

11. Click **Apply** to save this new remote Java application debug configuration.

Now the developer can start to debug the Network Integrity cartridge (which should be already deployed on the Network Integrity system) from Design Studio by picking up the debug configuration just created. There is no difference from debugging a normal local Java application in Eclipse. We can put a break point in the cartridge Java source and start debugging from there. For instructions on how to debug a Java program in Eclipse, see the Eclipse Help topics **Java development user guide**, **Getting Started**, **Basic tutorial**, and **Debugging your programs**.

Sealing and Unsealing Projects

Some Network Integrity production cartridges are distributed as sealed projects. Unsealing Network Integrity production cartridges violates the license, support, and maintenance agreements with Oracle.

You may encounter build problems if you unseal a sealed cartridge in your workspace. The error logs may indicate that some dependent JAR files are missing from the workspace. The main cause for this is that the sealed cartridge may not have included any source code, and that a Clean operation may delete the JAR file, and then is not able to recreate it. The solution is to delete the unsealed cartridge, and re-import the sealed cartridge.

See the Design Studio Help for more information about sealing and unsealing cartridges.

Exporting and Importing Cartridges

This section provides an overview of exporting and importing Network Integrity cartridges.

Cartridge projects can be exported to archive files. This allows the cartridge projects to be distributed as a single or a set of archive files, rather than as the many files of a cartridge project. Once a project is exported to an archive file, the archive file can be distributed and then imported into a different Design Studio or Eclipse workspace.

Before exporting a cartridge project, you should decide whether you want to include your source code in the archive file. Cartridges can be extended without distributing source code. However, if you want to allow the user to modify the actual distributed cartridge, then you must distribute the source code.

Cartridges can also be exported in both sealed and unsealed states. If you are distributing a cartridge without source code, Oracle recommends you seal the cartridge before exporting it. This prevents the user from changing the cartridge model and therefore breaking the cartridge.

See the Design Studio Help for more information about sealing and unsealing cartridges.

Network Integrity production cartridges are distributed as sealed cartridges. Unsealing Network Integrity production cartridges violates the license, support, and maintenance agreements with Oracle.

See the following:

- Exporting a Cartridge with Source Code
- Exporting a Cartridge Without Source Code

Exporting a Cartridge with Source Code

To export a cartridge project containing source code:

1. From the Design Studio **File** menu, select **Export**.

   The Export Select dialog appears.

2. From the list of export destinations, expand the **General** node and select **Archive File**.

3. Click **Next**.

   The Export Archive file dialog appears.

4. Enter a destination archive file:

   a. Select the projects that you want to include in the archive.

   b. Specify the name and location of the archive file.

   c. In the **Options** section, accept the defaults.

   d. Click **Finish** to create an archive file containing the exported projects at the specified location.

Exporting a Cartridge Without Source Code

Before exporting a cartridge project without source code, the project's classpath must be modified.

See the following:

- Modifying the Classpath

- Exporting the Cartridge

Modifying the Classpath

To modify the classpath:

1. Open the **Navigator** view.

2. Use the Navigator view to rename the projects output directory *out*, to *classes*.

3. From the Design Studio **Window** menu, select **Show View**, and select the **Package Explorer** view.

4. Right-click the project and select **Properties**.

   The Properties dialog appears.

5. From the list of properties, select **Java Build Path**.

   The Properties dialog box displays the Java Build Path information.

6. Select the **Source** tab.

   The **Source** tab displays the folders on the build path for the selected project.

7. Remove the source directories that are part of the classpath:

   a. Select the source folders on the build path.

   b. Click **Remove**.

8. Select the **Libraries** tab, and click **Add Class Folder** to add the class folder *classes* to the classpath.

   Figure 1–1 shows how the class folder is added to the classpath.

Figure 1–1   Adding the Class Folder
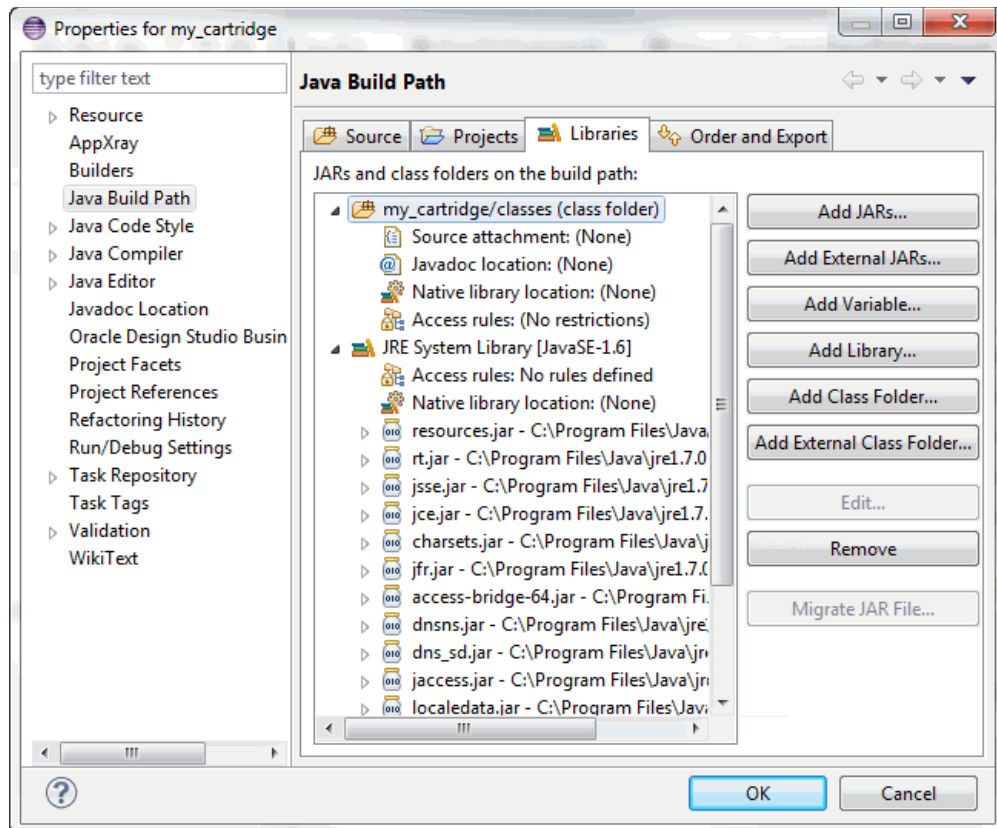
9. Select the **Order and Export** tab, and check the box corresponding to the *classes* class folder.

10. Click **OK** to complete the modification of the project classpath.

    After changing the classpath, if you wish to continue development on the cartridge, you should restore the classpath to its original configuration.

Exporting the Cartridge

To export the cartridge project:

1. From the Design Studio **File** menu, select **Export**.

    The Export Select dialog appears.

2. From the list of export destinations, expand the **General** node and select **Archive File**.

3. Click **Next**.

    The Export Archive file dialog appears.

4. Enter a destination archive file:

    a. Select the projects that you want to include in the archive.

    b. For the projects for which you are not including source code, expand the project tree and deselect the source directories which you previously removed from the classpath.

    c. Specify the name and location of the archive file.

d. In the **Options** section, accept the defaults.

e. Click **Finish** to create an archive file containing the exported projects at the specified location.

About Specifications

Network Integrity cartridges persist their results to persistent object modeling service (POMS) in the Oracle Communications Information Model. The Information Model defines a base set of entities and their relationships. Use specifications to extend the Information Model. Most cartridges must extend the Information Model entities and therefore must make use of specifications.

Scan parameter groups are a special type of specification. A specification used for model extension is associated with a single Information Model entity type. Multiple specification types can be defined for each Information Model entity type. The elements that comprise the specification are called characteristics.

Specifications can be shared between cartridge projects. Specifications created in a cartridge project are automatically related to all actions in the same cartridge project. You cannot add specifications to a model collection in the same cartridge project, but you can make the cartridge project containing the model collection dependent on another project that contains the specifications you want to add. Network Integrity ensures that when multiple cartridges are deployed together, their shared specifications are compatible.

When cartridge code persist information to POMS, it creates Information Model entities and usually a specific type of specification is attached to each Information Model entity to hold additional attributes. Within the Network Integrity UI, an Information Model entity and its specification are represented as a single object.

All action types must define which specification types (and by extension, which Information Model entities) they use by creating specifications in the cartridge project or adding specifications to the model collection. The **Model** tab defines the list of model collections on the action. Design Studio generates special classes for specifications, called specification helpers.

Characteristics on specifications appear in the Network Integrity UI as displayed information. Specification characteristics are always read-only in the Network Integrity UI. By configuring characteristics on specification, the following read-only fields can appear in the Network Integrity UI:

- **Label**: Specifies the label that displays in the UI

- **Tool Tip**: Specifies a short message when the pointer hovers over the field

Working with Specifications

Working with specifications requires the following high-level steps:

1. Add specifications to your cartridge project:

   a. Create or copy specifications and configure them to collect the information you want.

   b. Add existing specifications from dependent cartridge projects to the model collection.

2. Configure characteristics on new and copied specifications to appear in the Network Integrity UI.

To stop using a specification, remove it from the model collection or delete it from the cartridge project.

See the Design Studio Modeling Network Integrity Help for more information about specifications.

About Model Collections

Use model collections to add specifications that exist in other cartridge projects. Specifications from other cartridge projects inherit any changes and configurations you make to them in their original cartridge project.

See the Design Studio Modeling Network Integrity Help for more information about creating and using model collections.

About Specification Helpers

Design Studio generates specification helper classes to the following package:

- *Cartridge Default Package*.**Model Collection Name**.*Model Collection Name*

The names of the specification helpers are based on the names of the specifications. For example if the name of the specification is **deviceGeneric**, then the name of the specification helper is **DeviceGeneric**.

Specification helpers have getter and setter methods for each element in the specification. The specification helper also has a constructor which takes a POMS entity interface object. A code sample which illustrates the use of a specification helper is shown below. In the example, the DeviceGeneric class is the specification helper.

```
// create a Logical Device entity which uses
// the Device Generic specification.
LogicalDevice logicalDevice = PersistenceHelper.makeEntity(LogicalDevice.class);
DeviceGeneric logicalDeviceExt = new DeviceGeneric(logicalDevice);

// Set static attribute values to LogicalDevice.
logicalDevice.setId(makeLDevID(scanResponse));
logicalDevice.setName(rfc1213Mib.getSysName());
logicalDevice.setDescription(rfc1213Mib.getSysDescr());

// Set dynamic attributes/characteristics.
logicalDeviceExt.setMgmtIPAddress(scanResponse.getManagementIP());
logicalDeviceExt.setSysObjectId(rfc1213Mib.getSysObjectID());
```

About Source Control

See *Design Studio Developer's Guide* for information about source control.

Working with Source Control for Network Integrity

When developing cartridge projects for Network Integrity, you may store your work in various source control systems. The eclipse platform, upon which Design Studio is based, provides support for integrating with source control systems. Plug-ins are available for most common source control systems. The exact behavior of Design Studio when used in an environment where the files are backed by a source control system depends on the source control system and the source control Team plug-in that the developer is using.

This section describes which files must be source controlled and which files must be writable to continue working.

Table 1–1 describes the structure of the directories and the files in a Design Studio for Network Integrity project and recommends how they should be handled with respect to a source control system.

Table 1–1   Source Control Handling for Various Files and Directories

Directory or File	Description	Source Control Handling
ProjectDir/	Project's top level directory.	Under source control. All files directly under this directory must be source controlled.
*ProjectDir/***cartridgeBin/**	Cartridge bin directory is where the deployable IAR files are located.	This directory should be source controlled but the contents should not.
*ProjectDir/***cartridgeBuild/**	Cartridge build directory contains files which are outputs of the cartridge build process.	This directory should be source controlled but the contents should not.
*ProjectDir/***dataDictionary/**	This directory contains the files where the data dictionary information is stored.	This directory and its contents should be source controlled.
*ProjectDir/***doc/**	This directory contains documentation files.	This directory and its contents should be source controlled.
*ProjectDir/***generated/**	This directory contains generated artifacts of the build process.	This directory should be source controlled. Except for the src sub-directory, the contents of this directory should not be source controlled.
*ProjectDir/***generated/src/**	This directory contains generated artifacts of the build process.	This directory should be source controlled, but it contents should not.
*ProjectDir/***integrityLib/**	This directory contains jars that are part of the Network Integrity server Enterprise Archive (EAR). These jars are in the project's classpath.	This directory should be source controlled. The files in this directory should not be source controlled.
*ProjectDir/***integrityLib/ packaged**	This directory contains jars that are created by Design Studio for Network Integrity and which are packaged into the cartridge IAR file. The jars are added to the Network Integrity EAR when the cartridge is deployed. These jars are in the project's classpath.	This directory should be source controlled. The files in this directory should not be source controlled.
*ProjectDir/***lib/**	This directory contains jars and other files that are not part of the Network Integrity server EAR. Some of these files are part of the project classpath.	This directory should be source controlled. The mds.mar file is output to this directory. The mds.mar file should not be source controlled. The user may also want to source control other files in this directory.
*ProjectDir/***mdsArtifacts/**	This directory contains files that are both input and outputs of the UI Hints infrastructure.	This directory should be source controlled. The following files under this directory should also be source controlled: ■ **MDSAvailablePagePanels.xml** ■ **MDSAvailablePagePanels.xsd** ■ **MDSMetaData.xml** The remaining files in this directory should not be source controlled.

Table 1–1   (Cont.) Source Control Handling for Various Files and Directories

Directory or File	Description	Source Control Handling
ProjectDir/**model/**	This directory contains files that are used to persist the information about cartridges, actions, processors, model collections and address handlers.	This directory and its contents should be source controlled.
ProjectDir/**out/**	This directory contains output classes.	This directory should not be source controlled.
ProjectDir/**resources/**	This directory is not used.	This directory does not need to be source controlled.
ProjectDir/**src/**	This directory contains the user supplied code for the cartridge.	This directory and its contents should be source controlled.

Design Studio for Network Integrity assumes that all files and directories of a cartridge project are writable. Some source control systems and team plug-ins automatically manage the files and directories to make them writable as the software needs to write to them. If this is not the case for your chosen source control/Team plug-in combination, then you should manually ensure that this is the case before working with a source controlled project.

Tips and Tricks

This section provides tips and tricks for working with processors in Design Studio and compiling and building Network Integrity cartridges.

About Java Errors in the Generated Controller Class

Compile errors in the generated Controller class of an action usually mean that there are errors in the configuration of the processor table of that action. Look for a Design Studio Error on an action or processor involved in the processor chain. Correct the error, then save all files and perform a clean operation to regenerate all generated files.

Renaming or Deleting Actions and Processors

When renaming an action or a processor, Design Studio only renames and refactors the generated Java source code. Likewise, when deleting an action or a processor, Design Studio only deletes the generated Java source code. These changes result in errors remaining in the processor implementation code and they must be corrected manually.

Adding External Libraries to a Java Build Path

To add an external library to the project for use by a processor, you must first copy the JAR file into the lib directory of the cartridge project. Then, you must add an entry for this library into the project's Java Build Path. This can only be done in the Package Explorer or the Navigator view.

From either view, right-click the project and select **Properties**. In the Properties dialog, select **Java Build Path** in the left side, and select the **Libraries** tab. Now you can select **Add External Jars** to add your libraries.

About "Missing Required Library" Errors for External Libraries

You have copied the required library JAR files into the lib directory of your cartridge project, and you have added these libraries into your project's Java Build Path. If you

are still getting **missing required library** errors, refresh your cartridge project to cause Design Studio to notice the added library.

To refresh your project, go to the menu **Windows**, then **Show View,** then open **Package Explorer**, then right-click your project, and select **Refresh**. Follow this by cleaning and building the project.

Error Marker on Cartridge but not on any Entities

If there is an error marker on the cartridge itself, but there are no error marker on any cartridge entities (actions, processors, Model Collections, and so on), then try checking the cartridge project using the Package Explorer view or the Navigator view. Sometimes the error markers are on some generated artifacts instead.

If there are no error markers on anything else, then try a **Refresh** and **Rebuild** operation. Go into Package Explorer or Navigator view, right-click the top-level project, and select **Refresh**. Then, choose the menu **Project**, then **Clean**, and choose to **clean** and **rebuild** all projects.

2

Working with Actions

This chapter provides information about Oracle Communications Network Integrity actions, result categories, and discrepancies.

This chapter contains the following sections:

- About Actions
- About Import Actions
- About Discovery Actions
- About Assimilation Actions
- About Discrepancy Detection Actions
- About Discrepancy Resolution Actions

About Actions

Actions are entities that represent a particular software function that a deployed cartridge performs at run time. A cartridge project usually contains multiple actions.

At run time, when an action is deployed to Network Integrity (by deploying a Network Integrity cartridge from Oracle Communications Design Studio, or by using the Oracle Cartridge Deployer), an action is implemented as a J2EE Message Driven Bean (MDB).

Actions are of different types:

- **Import action**: Used for importing data, typically from an inventory system, and persisting the inventory data in the Results Model using POMS entity managers.

- **Discovery action**: Used for discovering data, typically from a network, and persisting the discovered data in the Results Model using POMS entity managers.

- **Assimilation action**: Used for post-processing previously discovered data, and persisting the data in the Results Model using POMS entity managers. The assimilation action cannot produce import results.

- **Discrepancy detection action**: Used for finding discrepancies between discovered entities and imported entities.

- **Discrepancy resolution action**: Used for fixing discrepancies in an external system, or a network.

See the Design Studio Modeling Network Integrity Help for more information about creating actions.

About Actions and Processors

An action performs a certain function that is supported by a Network Integrity project. To implement this function, a processor is introduced to implement an atomic sub-function, which is part of the functions performed by the action. For example, an SNMP discovery action has at least one processor that performs SNMP polling on network devices and another processor that models the discovered raw SNMP data into the Results Model and persists it using POMS entity managers.

An action contains one or more processors. Each processor is responsible for an atomic function. By chaining the processors inside an action, the action can perform a complex function, such as discovering a network, importing an inventory system, assimilating discovered data, or detecting and resolving discrepancies.

When an action is invoked, the processors are executed in the sequence they were placed inside the action. The code-generated action controller controls execution.

See "Working with Processors" for more information about processors.

About Action within Actions

You can add an entire action as a processor in an action. If the action you want to add belongs to another cartridge project, you must make your project dependent on the one containing the action you want to add.

You cannot modify the order in which the processors from an imported action are run, but you can place new processors in between its processors.

For example, Table 2–1 shows two actions.

Table 2–1   Example Action Used as a Processor in Another Action

Action A	Action B
Action A consists of the following processors:	Action B consists of the following processors:
1. Processor A1	1. Processor B1
2. Processor A2	2. Action A
3. Processor A3	3. Processor B2

The full representation of Action B in Table 2–1 is:

1. Processor B1

2. Action A:

    a. Processor A1

    b. Processor A2

    c. Processor A3

3. Processor B2

In this example, action B actually contains five processors. The sequence of the processors from action A cannot be changed in action B. However, new processors can be inserted between the processors from action A.

For example, the Cisco SNMP cartridge contains a discovery action, which extends the discovery action from the MIB-II SNMP cartridge.

Figure 2–1 shows the processors contained inside the Discover Generic Cisco SNMP action (from the Cisco SNMP Cartridge).

Figure 2–1   Discover Generic Cisco Action Processors

Name	Provider	Conditional	Imported Action	Owner Action
MIB II Properties Initializer	Oracle Communications		Discover MIB II SNMP	Discover MIB II SNMP
Cisco SNMP Properties Initializer	Oracle Communications			
MIB II SNMP Collector	Oracle Communications		Discover MIB II SNMP	Discover MIB II SNMP
MIB II SNMP Modeler	Oracle Communications		Discover MIB II SNMP	Discover MIB II SNMP
Cisco SNMP Logical Collector	Oracle Communications			
Cisco SNMP Physical Collector	Oracle Communications			
Cisco SNMP Logical Modeler	Oracle Communications			
Cisco SNMP Physical Modeler	Oracle Communications			

This discovery action contains Discover MIB II SNMP as the imported action. By importing the Discover MIB II SNMP action, the Discover Generic Cisco action automatically gets the MIB II discovery functions (logical device discovery) provided by the productized MIB-II SNMP cartridge.

In addition, the Discover Generic Cisco action discovers physical devices (through Cisco SNMP Physical Collector processor and Cisco SNMP Physical Modeler processor), modeling the logical side (through the Cisco SNMP Logical Collector processor and Cisco SNMP Logical Modeler processor).

About the Generated Action MDB and Controller

Every action becomes a J2EE Message Driven Bean (MDB) at run time. The controller controls the execution sequence of the processors inside an action.

Both the Action MDB and controller classes are code-generated. No further Java coding is necessary for either the MDB or the controller class. These two classes are transparent to a Network Integrity cartridge developer using Design Studio. At design time, the cartridge developer should not have to implement any Java code for an action because all required Java implementations for actions are code-generated.

The generated Action MDB and controller classes can be found at the following directory:

*Studio_Workspace\NI_Project_Root***generated\src***\Project_Default_Package\Action_Type\Action_Implementation_Prefix*

where the elements on the path are defined as follows:

- *Studio_Workspace*: Eclipse Workspace root

- *NI_Project_Root*: Network Integrity project root

- *Project_Default_Package*: The default package configured in the Project editor

- *Action_Type*: Select from the available action types:

    - assimilationactions

    - detectionactions

    - discoveryactions

    - importactions

    - resolutionactions

- Action_Implementation_Prefix: action implementation prefix in lowercase.

The generated MDB class is named: *ActionName*MessageDrivenBean.java.

The generated controller class is named *ActionName*MessageDrivenBeanController.java.

During design time, compilation errors or warnings against this Java class might occur. These errors and warnings are cleared after properly implementing and configuring the action (and its processors).

Figure 2–2 shows the directory that contains the generated MDB and controller classes.

Figure 2–2   Generated MDB and Controller Class Directory
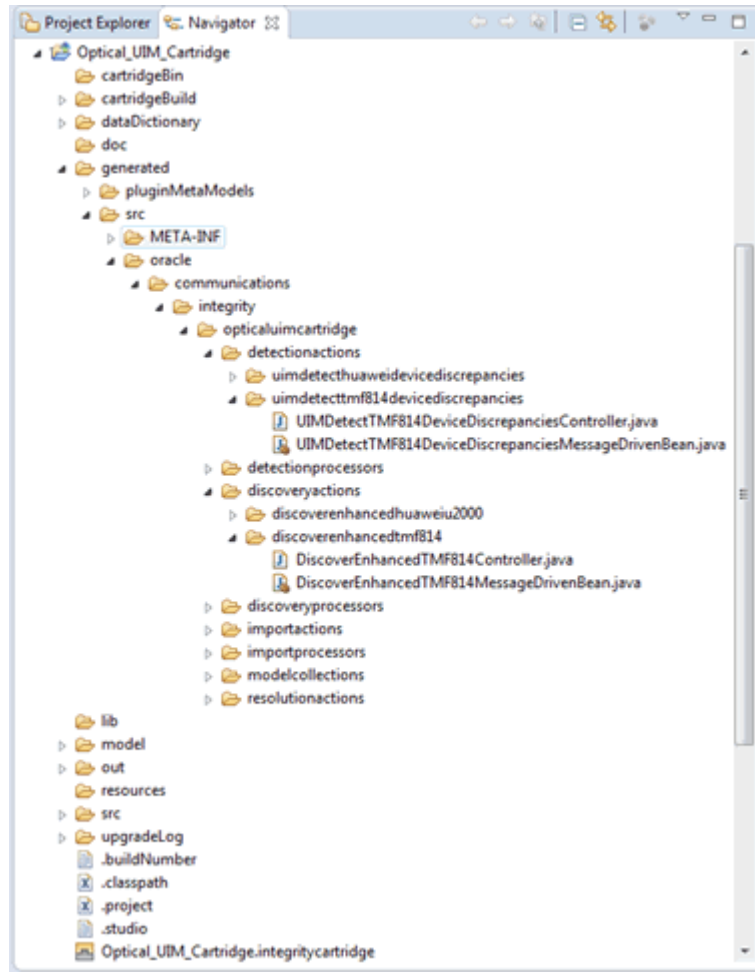
About Scan Parameter Groups

Scan parameter groups are a special type of specification that adds fields to the Network Integrity UI. You can add fields to the Create Scan page, allowing the Network Integrity user to pass scan parameter values to run-time scans. You can add fields to the Scan Details page, displaying the configured scan parameter values on configured scans.

Add and configure characteristics on scan parameter groups to create input fields for scan parameters in the Network Integrity UI.

You can add scan parameter groups to the following types of actions:

■   Assimilation actions

■   Discovery actions

■   Import actions

See the Design Studio Modeling Network Integrity Help for more information about creating and configuring scan parameter groups.

Extending the Create Scan Page

In Design Studio, you can configure characteristics on scan parameter groups to appear as input fields on the Create Scan page of the Network Integrity UI. These input fields allow the Network Integrity user to pass scan parameters to run-time scans.

For example, if a network device requires a login and password for Network Integrity to establish a connection, you can add input fields for the user name and password to the Create Scan page. Network Integrity users can enter the user name and password and save the values to the scan. Each scan run passes the user name and password parameter values to the network device to establish a connection.

See the Design Studio Modeling Network Integrity Help for more information about adding and configuring characteristics on scan parameter groups.

Figure 2–3 shows the Create Scans page. The Scan Action Parameters section lists all the input fields defined by characteristics on scan parameter groups in Design Studio.

Figure 2–3   The Create Scans Page

Extending the Scan Details Page

In Design Studio, you can configure characteristics on scan parameter groups to appear as read-only fields on the Scan Details page of the Network Integrity UI. These fields display the saved scan parameter values on the scan.

See the Design Studio Modeling Network Integrity Help for more information about adding and configuring characteristics on scan parameter groups.

Figure 2–4 shows the Scan Details page.

Figure 2–4   The Scan Details Page
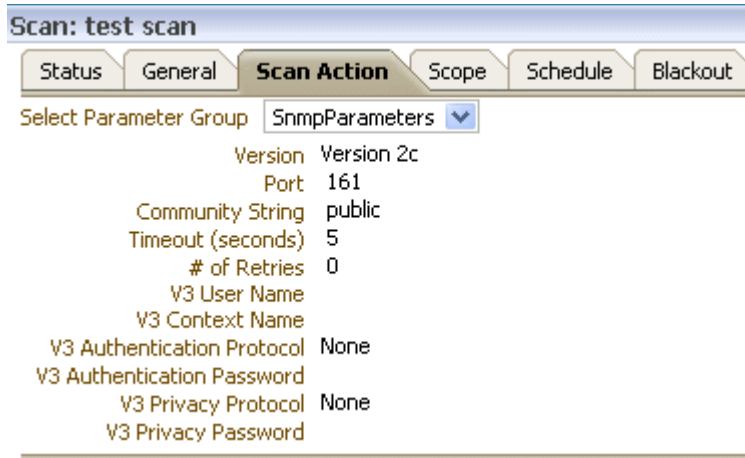
About Conditions

Design Studio sets conditions for processors used in action executions in Network Integrity.

An action can contain conditions. By creating and applying conditions to processors, at run time you can dynamically control which processors should be executed inside an action based on the condition (whether true or false).

Conditions are implemented as a Java class that implements the condition interface. Design Studio generates the code for the condition interface. You then implements the condition interface.

Conditions can be applied to one or more processors. Conditions can be set to be either true or false. One processor can also have multiple conditions applied. In this case, the processor are executed if all the conditions are true

See the Design Studio Modeling Network Integrity Help for more information about creating conditions and applying them to processors.

About Generated Classes and the Implementation Class

When a condition is configured for an action, Design Studio generates two classes:

- Condition interface, which takes the name *ConditionName_Implementation_ Prefix***Condition.java**

- Request, which takes the name *ConditionName_Implementation_Prefix***Request.java**

The generated classes are available at:

Studio_Workspace*NI_Project_Root***generated****src***Project_Default_Package**Action_ Type**Action_Implementation_Prefix*

> **Note:** This directory also contains generated action MDB and
> controller classes.

The following is a sample generated condition interface which defines one method,
`checkCondition`. In this sample, `ValidDeviceRequest` is the generated request class for
the condition:

```
public interface ValidDeviceCondition {

   /**
    * @param context
    * @param request
    * @return @see boolean
    * @throws ProcessorException
    */
  public boolean checkCondition(DiscoveryProcessorContext context,
             ValidDeviceRequest request) throws ProcessorException;
}
```

Design Studio also generates the skeleton implementation class for this condition
interface. To open the Java editor and start the Java implementation, click the
Implementation Class link.

Adding Dependent Actions with Conditions as Processors

When you add an action from a dependent cartridge project, the action comes with its
conditions. The conditions cannot be removed from any processors to which they are
applied in the dependent cartridge project.

You can add and remove additional conditions to processors belonging to actions from
dependent cartridge projects.

By adding new conditions to dependent action processors, you can change whether an
imported processor is executed.

Creating Condition Examples

See the following for examples of setting conditions in Network Integrity:

- Multiple Vendor SNMP Discovery

- Multiple Protocol Discoveries

About Model Collections in Actions

Use model collections to gather specifications from other, dependent cartridges and
make them available to actions in the current cartridge project.

Adding a model collection to an action enables the generation of the Specification
Helper classes for specifications from other cartridge projects. These classes are by the
action for modeling the discovered data into the Oracle Communications Information
Model and persisting it using POMS entity managers.

If an action is imported into another action in a different cartridge project, the Network
Integrity packager uses the model collections to determine how to build the
specification DAO files so that all specifications (from both the imported action and
the current action) are included.

See "About Model Collections" for more information about model collections.

About For Each Processors

An action can contain a For Each processor. The action controller sets the execution sequence of the processors based on the order in which the processors are configured. Usually a processor is invoked only once, and when it has executed, the controller invokes the next processor, until all processors in an action are invoked.

However, one or more processors may be executed repeatedly. For example, when importing an inventory system, it is typical to first get a list of devices from the inventory system, then go through the list of devices and import each device singly into Network Integrity. In this example, the processor importing a single device is repeatedly executed for all the devices in the returned device list. You can use For Each processors to create a loop, containing one or more processors, to repeatedly execute the processors. Design Studio for Network Integrity supports nested For Each processors.

A For Each processor expects a collection as the input parameter so that it can iterate through the collection and, for each object in the collection, invoke the processors inside the loop. There must be a preceding processor that outputs an array or a Java object that implements **java.lang.***Iterable* (for example, **java.util.List**) as an output parameters to create a For Each processor.

See the Design Studio Modeling Network Integrity Help for more information about creating For Each processors.

About Result Categories

Result category is a mandatory field for the following action types:
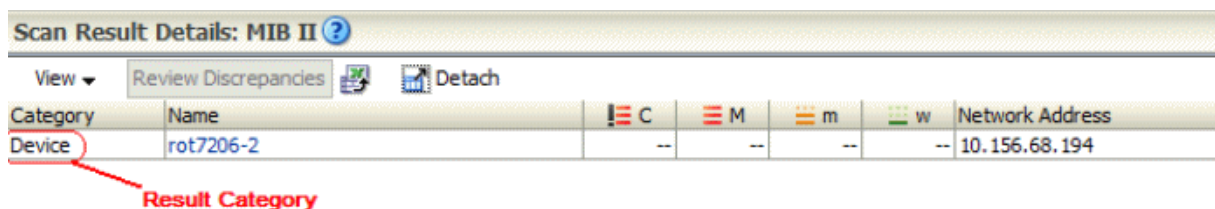
- Discovery action

- Import action

- Assimilation action

Result category is the identifier for a result group. An action configured with a result category persists the results to the corresponding result group after being deployed and executed in Network Integrity. The result category is visible in the Network Integrity UI when displaying the scan results.

Figure 2–5 shows the result category in the Network Integrity UI. The discovered device is stored under the result category, *Device*.

> **Tip:** Provide an appropriate result category when configuring an action, because this value is displayed in the Network Integrity UI.

Figure 2–5   Result Category in Network Integrity UI

Result categories identify a result group that an action adds the results to the result group. The result category value configured for the action must match the result group name in the Java implementation (the `addToResult` method) for the discovered data. See *Network Integrity Information Model Reference* for information about using result categories in modeling results.

For more information about this Java implementation, see "Working with Processors".

Design Studio does not explicitly validate this result category name against the actual result group name specified in the Java implementation.

The result category and action define a result source for the following action types:

- Discrepancy detection action
- Discrepancy resolution action

Both actions work on results (to perform discrepancy detection or resolution, respectively) based on the result source.

For example, a discovery action persists discovered data in two result categories:

- *Device*
- *Workstation*

A discrepancy detection action works on discovered data stored in the result categories that match the result groups in the Java implementation. If the result category configured for the discovery action does not match the actual result group name in the Java implementation, but the discovery detection action is configured with the result source based on the result category configured in Design Studio, the discrepancy detection action is not able to find the results to perform discrepancy detection at run time. In other words the result group name does not match the result category defined in result source.

About Import Actions

Import actions are used to import data from an inventory system into Network Integrity. The data is stored in the Oracle Communications Information Model representation and is flagged as having come from the inventory system. The Network Integrity GUI displays and reports on the data discovered by an import action. The data can also subsequently be processed by discrepancy actions that compare network-discovered data to inventory-discovered data, and reports differences between them.

Import actions are edited in Design Studio. As a result of the editing, Design Studio generates most of the required deployment artifacts. However, you must supply some Java implementation. After this is done, and all error problems are cleared, and if the import action is not abstract, Design Studio automatically packages the action into a cartridge Integrity ARtifact (IAR) file that can be easily deployed into the Network Integrity server. Then, on the Network Integrity server, an import scan can be created and executed, and the scan results viewed or reported on.

See the Design Studio Modeling Network Integrity Help for more information about creating and configuring import actions and processors.

See "Implementing an Import Processor" for more information.

About Discovery Actions

The discovery action discovers data, typically from the network, and persists it to the Oracle Communications Information Model. The discovery action accesses the network using a variety of technologies and protocols, such as simple network management protocol (SNMP).

Because SNMP is such an important protocol for network discovery, Network Integrity provides specific features to allow streamlined development of SNMP

network discovery cartridges within Design Studio for Network Integrity. See "Implementing the SNMP Processor" for more information.

See "Implementing a Discovery Processor" for more information.

See the Design Studio Modeling Network Integrity Help for more information about creating and configuring discovery actions and processors.

About Discovery Action Address Handlers

Discovery scans are often used to scan multiple devices in the network. A discovery scan can use a variety of protocols to perform a scan. To facilitate scan processing, Network Integrity supports an address expansion and validation software component called an address handler. Address handlers perform two functions:

- They validate that a user-supplied address string is syntactically correct for a protocol.

- They expand address strings which represent multiple addresses, into a collection of individual addresses.

  This allows the user to configure a scan of multiple addresses using a compact, efficient notation; for example: the notation 10.156.67.1-254 expresses the range of addresses from 10.156.67.1 to 10.156.67.254, which is 254 addresses.

Discovery actions can optionally specify an address handler to use. It is best practice to create an address handler whenever address validation is desired. Addresses are validated when a scan configuration for the discovery action is saved, and also when the scan is run.

In addition, address strings representing multiple addresses are expanded into a collection of addresses when the scan runs. When an address string is expanded into multiple addresses, Network Integrity calls into the discovery action multiple times until each individual address has been scanned. The scanning of multiple addresses is done in parallel.

Address handlers are created in Design Studio for Network Integrity. Design Studio for Network Integrity generates some artifacts for the address handlers. However, you should supply implementation code to complete the address handler.

Address handlers become stateless session beans in the run-time environment. Cartridge projects containing address handlers must be deployed before any cartridge project that uses the address handlers are deployed.

> **Note:** Address handlers cannot be created in the same cartridge project as actions. To add address handlers to actions, you must make the cartridge project that contains the actions dependent on the project that contains the address handlers.

You can download the and import the Address_Handler cartridge project which contains several basic address handlers. See "About the Address_Handlers Cartridge" for more information.

See the Design Studio Modeling Network Integrity Help for information about creating address handlers.

About the Address_Handlers Cartridge

Network Integrity provides the *Address_Handlers* cartridge which implements the following address handlers:

- *IPAddressHandler* validates and expands both IPv4 and IPv6 address.

  It validates and expands the following IP address formats:

  – Single IP addresses; for example: 10.156.67.123

  – IP address ranges using "-"; for example: 10.156.67.10-125

  – IP address ranges using "*"; for example: 10.156.67.*, equal to 10.156.67.0-255

  – IP addresses using Classless Inter-Domain Routing (CIDR); for example: 10.156.67.0/24

- *URLAddressHandler* validates URL syntax addresses.

- *File TransferAddressHandler* validates addresses and paths used by the file transfer processor, as follows.

  – Allows the field to contain one or two tokens delimited by "/"

  – Using a single token identifies:

    * The absolute path to files that are local to the Network Integrity server, for example: /tmp

  – Using two tokens identifies:

    * The remote location and absolute path

    * Host_name/path, for example: someserver.us.com/tmp/test

    * IPV4Address/path, for example: 10.156.58.63/tmp/test

    * IPV6Address/path

  – Validates the proper format of IPV4 and IPV6Address

  > **Note:** The file transfer processor does not support address expansion and relative paths.

- *Corba URLAddressHandler* validates that the address entered in Network Integrity is a properly formatted IPv4 or IPv6 CorbaLoc URL. For more information, see *Network Integrity CORBA Cartridge Guide*.

Implementing Address Handlers

You must specify the implementation class for an address handler. See the Design Studio Modeling Network Integrity Help for more information.

About the AddressHandler Interface

Address handlers must implement the AddressHandler interface which is shown and described in the following section:

```
package oracle.communications.integrity.api;

import java.util.List;
import oracle.communications.integrity.common.AddressHandlerException;
import oracle.communications.integrity.common.AddressesStatus;
```

```
/**
 * AddressHandler is common interface which should be implemented by the
 * class implementing the Address expansion and validation of addresses.
 */
public interface AddressHandler {

    /**
     * This method expands the list of address or addressRange provided.
     * @param addressRangeList - a list of String representing either an address
or an address range
     * @return List - a list of Strings each of which represents an individual
address
     * @throws AddressHandlerException
     */
    public List<String> expandAddressRange(List<String> addressRangeList) throws
AddressHandlerException;

    /**
     * This method validates the list of address provided.
     * @param address
     * @return AddressesStatus
     * @throws AddressHandlerException
     */
    public AddressesStatus validate(List<String> address) throws
AddressHandlerException;

    /**
     * This method validates the single address provided.
     * @param address
     * @return boolean
     * @throws AddressHandlerException
     */
    public boolean validate(String address) throws AddressHandlerException;

    /**
     * This method counts the number of addresses after expansion of address
parameter passed.
     * Here maxCountLimit can be NULL. If maxCountLimit is NULL, method return the
total count of expanded address.
     * If maxCountLimit is specified, method does not count the expanded address
     * beyound that limit and returns the maxCountLimit + 1.
     * @param addressRangeList
     * @param maxCountLimit
     * @return int
     * @throws AddressHandlerException
     */
    public int countExpandedAddresses(List<String> addressRangeList, Integer
maxCountLimit) throws AddressHandlerException;
}
```

About Dynamic Address Handlers

When you configure a Network Integrity discovery scan, you specify one or more addresses as the scope for the discovery scan.

The discovery scan scope can point to one or more addresses.

When the network changes, you likely need to modify the discovery scope to add or remove addresses.

You can create an address handler that references a file at run time, dynamically populating the discovery scan scope.

See the Design Studio Modeling Network Integrity Help for information about creating an address handler.

The following sections explain how to implement a dynamic address handler.

Validating the Address Handler

Validation methods are invoked to validate user-entered addresses. In this sample, an address is expected to be a path to a file (absolute, or relative to the WebLogic Server Network Integrity domain). This validation method checks each address, and the result indicates which addresses (if any) are not valid:

```
@Override
public boolean validate(String address) throws AddressHandlerException {
  File file = new File(address);
  if (!file.exists() || !file.isFile()) {
    return false;
  }
  return true;
}
```

You must also implement a list variant of the validation method without additional validation logic. The following sample shows the method for implementing a list variant.

```
@Override
public AddressesStatus validate(List<String> addresses)
    throws AddressHandlerException {
  AddressesStatus result = new AddressesStatus();
  for (String address : addresses) {
    if (!validate(address)) {
      result.getInvalidAddressList().add(address);
    }
  }
  result.setAllAddressValid(result.getInvalidAddressList().isEmpty());
    return result;
}
```

Expanding Address Handlers

When you run a scan, the address handler invokes address expansion methods to derive individual address from ranges of addresses.

The expandAddressRange method takes the addresses (as entered on the **Scope** tab) and returns a list of expanded addresses.

The file is read line by line and the following logic is applied:

■ Remove leading and trailing white space

■ Ignore empty lines

■ Ignore comments (starting with #)

■ When a line starts with $, it indicates a malformed address and the address expansion fails.

The explicit validate method is not invoked for expanded addresses.

The use of a LinkedHashSet avoids issues with duplicate addresses in the file, while still preserving the order. In this sample, each input address references a file.

```
@Override
public List<String> expandAddressRange(List<String> addresses)
    throws AddressHandlerException {
  Set<String> expandedAddresses = new LinkedHashSet<String>();
  for (String address : addresses) {
    expandedAddresses.addAll(readAddressesFromFile(address));
  }
  return new ArrayList(expandedAddresses);
}
public List<String> readAddressesFromFile(String path)
    throws AddressHandlerException {
  try {
    BufferedReader reader = new BufferedReader(new FileReader(path));
    try {
      List<String> addresses = new ArrayList<String>();
      String address = null;
      while ((address = reader.readLine()) != null) {
        // ignore blank lines, and comment lines (starting with #)
        address = address.trim();
        if (! address.isEmpty() && ! address.startsWith("#")) {
          // Address validation applies only to addresses entered as Scope for
scan. In this example, further validation may be of interest in case file content
is malformed. This illustrates how to reject an illegal dynamic address
          if (address.startsWith("$")) {
            throw new AddressHandlerException("Illegal address \"" + address + "\"
found in file \"" + path + "\"");
          }
          addresses.add(address);
        }
      }
      return addresses;
    } finally {
  reader.close();
  }
} catch (IOException ex) {
  throw new AddressHandlerException("Unable to read addresses from file \"" + path
+ "\"", ex);
  }
}
```

The following sample shows a method that returns the count of the expanded
addresses. For certain types of address handlers, counting is more efficient than
expansion. For example, a /24 IP address range is 256 addresses. In this sample,
addresses are expanded and counted.

```
@Override
public int countExpandedAddresses(List<String> addresses, Integer maxCount)
    throws AddressHandlerException {
  return expandAddressRange(addresses).size();
}
```

Testing the Dynamic Address Handler

To test a dynamic address handler, create a discovery action in Design Studio that uses
the dynamic address handler you implemented. See the Design Studio Modeling
Network Integrity Help for information about creating actions.

To test the dynamic address handler:

1. Deploy the cartridge containing the dynamic address handler and the discovery
   action.

2. Create an **address.txt** address file that is accessible to the application server. The file is created in the Weblogic domain home directory with the following content:

```
Some Address
Another Address
Address 3
```

3. In Network Integrity UI, create a scan and select the discovery action you created.

4. In the **Scope** tab, specify the **addresses.txt** file.

5. Run the scan.

6. On the **Scan Results** page, click **Display Addresses** to see the expanded addresses that were read from the file.

7. Edit the **addresses.txt** file and change the last address:

```
Some Address
Another Address
New Address
```

8. Run the scan again and view the addresses to see the new addresses that were read from the file.

About Discovery Action Result Categories

A discovery action must be configured with a valid result category. For example, a discovery action that discovers devices should be configured with the Device result category.

See "About Result Categories" for more information.

See the Design Studio Modeling Network Integrity Help for more information about adding a result category to a discovery action.

About the Discovery Action in the Network Integrity UI

After successfully building a discovery action in Design Studio (see "Building and Packaging Projects"), deploy the cartridge to Network Integrity (see "Deploying and Undeploying Cartridges").

When the cartridge containing the discovery action is successfully deployed to Network Integrity, log on to the Network Integrity UI and configure a scan using the deployed discovery action.

The recently deployed discovery action is available in the **Scan Action** list when creating a scan configuration. See the Network Integrity Help for more information about creating a scan.

Figure 2–6 displays a discovery action called *Discover Sample Device*.

Figure 2–6   Creating a New Scan Configuration

About Discovery Action Scan Parameter Groups

You can configure scan parameter groups for a discovery action. Add characteristics to scan parameter groups to appear in the Network Integrity UI as scan parameters. For example, consider the following scan parameters:

- **Port**: The port number that a discovery command is sent to

- **Username**: The user name to make the connection

- **Password**: The password to make the connection

When a scan is created using *Discover Sample Device* (see "About the Discovery Action in the Network Integrity UI"), the **Scan Action Parameters** section on the **Create Scan** page is filled with SNMP scan parameters.

Figure 2–7 displays the Scan Action Parameters area with SNMP scan parameters configured.

Figure 2–7   Configured SNMP Scan Parameters

To make configuration items available in the Network Integrity UI, add and configure characteristics on scan parameter groups. See the Design Studio Modeling Network Integrity Help for more information.

See "About Scan Parameter Groups" for more information.

About Assimilation Actions

Assimilation actions perform additional processing on existing Network Integrity network data to derive additional, often higher level, information from the data. For example, an assimilation action might be used to derive connectivity relationships between endpoints discovered by previous scans. Assimilation actions cannot manipulate or edit scan results.

Assimilation scans are different from other types of scans in that they do not retrieve their data from external sources. Instead, assimilation scans work on the scan results of other discovery, import, or assimilation scans. When you run an assimilation scan, the scan selects other scans as inputs to the assimilation scan in the Scope page of the Network Integrity GUI. You can select discovery, import, or other assimilation scans as input.

As with other scan types, the data from assimilation actions is stored in the Oracle Communications Information Model representation. The data from assimilation scans is flagged as having come from the network. The Network Integrity GUI displays and reports on the data discovered by an assimilation action. The data can also subsequently be processed by discrepancy actions, which compare network discovered data to inventory discovered data and report where differences are found.

Assimilation actions are edited in Design Studio. As a result of the editing, Design Studio generates most of the required deployment artifacts. However, you must supply some Java implementation. After this is done, and all error problems are cleared, and if the assimilation action is not abstract, Design Studio automatically packages the action into a cartridge Integrity ARtifact (IAR) file, which can be easily deployed into the Network Integrity server. Then, on the Network Integrity server, an assimilation scan can be created and executed, and the scan results viewed or reported on.

See "Implementing an Assimilation Processor" for more information.

See the Design Studio Help for more information on creating assimilation actions and processors.

About Discrepancy Detection Actions

The discrepancy detection action is a Network Integrity operation that compares discovery and import scan results, and reports on their differences by generating discrepancies.

A discrepancy detection action can be run immediately following a discovery, import, or assimilation scan. (Select the **Detect Discrepancy** check box in the scan configuration to set the trigger.) The entity results from the triggering scan become the Compare entities for the detection action. The action then uses a matching algorithm to find from the other side, and precedes with the comparisons.

See "About the Compare and Reference Sides" for a fuller description of the two sides of entities of discrepancy detection.

See "About the Base Detection Project and the Default Comparison Algorithm" for a description of the comparison algorithm.

Create a discrepancy detection action whenever new discovery, import, or assimilation actions are created, because every detection action is configured to receive results from specific actions only. See "About Result Sources" for more information.

See "About Discrepancy Detection Processors" for more information.

See the Design Studio Modeling Network Integrity Help for more information about creating discrepancy detection actions and processors.

About Discrepancy Detection

Discrepancy detection triggers immediately after a scan is finished. A scan is configured to use a single type of action, and therefore only generates Discovery results (representing network entities) or Import results (representing inventory system entities). Therefore, when the discrepancy detection action triggers, it has immediate access to one side of results: the compare entities.

For the other side of the results, the detection action searches the Network Integrity database for results with the following criteria:

- The results must come from the opposite system from the triggered scan. For example, if the detection action triggers from a discovery scan, then the detection action searches the database for Import result.

- The results have a matching name and result category (as configured by result source).

- The results must come from the most recent scan result.

If no matching results are found, then EXTRA_ENTITY discrepancies are generated for each root entity on that result.

About Result Sources

A result source specifies a list of scan actions that can trigger a discrepancy detection action. The triggering action must be a discovery, import, or assimilation action. By default, results from all categories are included in the discrepancy detection. It is possible to choose a subset of the categories to apply the discrepancy detection.

For example, Figure 2–8 illustrates a Cisco router discovery action that produces results in 2 categories: **Device** and **VPN**. Two separate detection actions are written to compare the results. Each detection action specifies a result source with the same action, but different result category. For example, the device discrepancy detection action receives results of Device category only.

Figure 2–8 Discrepancy Detection Actions (Example 1)
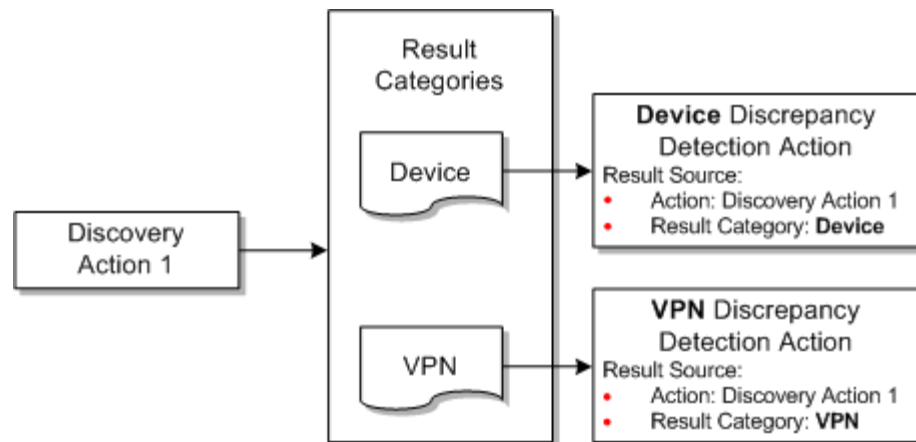
A result source that does not specify a result category matches every result category generated by the scan action. Figure 2–9 illustrates a Cisco discrepancy detection action that receives both device and VPN categories of results.

Figure 2–9 Discrepancy Detection Action (Example 2)
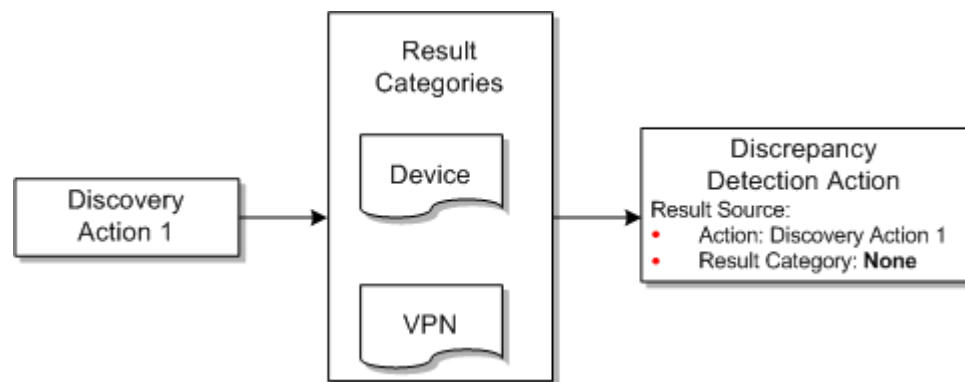
The result source is a mandatory field; there must be at least one entry in the table. Design Studio marks the discrepancy detection action with an error during a project build if the table has no entries.

> **Note:**   No two discrepancy detection actions can have the same result source.

About Result Source and Scan Types

Typically a result source configuration detection action has a single action as the result source: usually the discovery action. This detection action triggers when a scan is configured using that exact discovery action, and the **Detect Discrepancy** option is checked. This detection action does not trigger by scans configured with any other discovery or import action. Do not set the **Detect Discrepancy** option on the Import scan, because this might not trigger a detection action at all.

Generated Action MDB and Controller

The detection action is implemented as an MDB. See "About the Generated Action MDB and Controller" for more information.

About Discrepancy Resolution Actions

A discrepancy resolution action is an extendable Network Integrity operation which acts on an external system to resolve a discrepancy. For example, a resolution action updates a mismatch in an inventory system using information gathered from the network or generates a trouble ticket to kick off a network configuration change process.

A discrepancy resolution action operation is initiated by the Network Integrity user on the Manage Discrepancy page, using the following steps:

1. The user identifies the desired resolution action on selected discrepancies. Each discrepancy can have only one resolution action set.

2. The user submits the discrepancies with identified resolution actions to the system.

On receiving the submitted discrepancies, Network Integrity groups them based on their scan origin, result category, and resolution label, and then invokes the appropriate discrepancy resolution action.

The action then examines each discrepancy in detail, using the contained information to figure out the appropriate steps to resolve the problem.

As with other types of actions, a discrepancy resolution action is made up of a sequence of discrepancy resolution processors. The processors are shown in the Processor table in Design Studio. At the beginning of an action operation, these processors are invoked serially from top of the table to bottom. The first processor is given the list of submitted discrepancies marked. This processor determines a subset of these discrepancies to handle (which can range from none to all), performs the resolution operation, and sets their status to *Processed* or *Failed*. Then, the next processor is given the remaining discrepancies for processing, and so on.

The action is complete when all the processors are invoked. If there are any discrepancies which remain unhandled at the end, their status is automatically set to *Not Implemented*.

The following sections in this chapter describe general information about implementing a resolution action. For a detailed discussion of a working sample, see the following documents included with the cartridges:

- *Network Integrity Cisco Router and Switch UIM Integration Cartridge Guide*

- *Network Integrity MIB-II UIM Integration Cartridge Guide*

See "About Discrepancy Resolution Processors" for more information.

See the Design Studio Modeling Network Integrity Help for more information about creating discrepancy resolution actions and processors.

About the Resolution Action Label

The Resolution Action Label identifies the discrepancy resolution action in the Network Integrity UI. It is displayed as a command in the **Actions** menu of the Discrepancy Search Results table of the Review Discrepancies page.

Figure 2–10 displays the label corresponding to the command.

Figure 2–10   Resolution Action Label in Actions Menu of Network Integrity UI
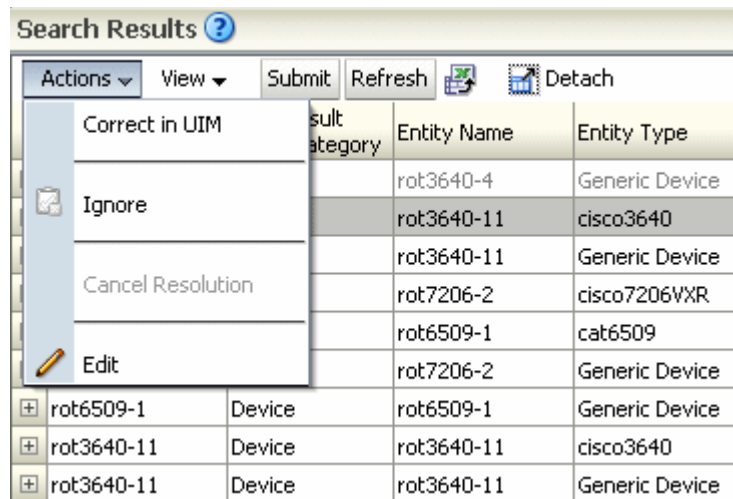
This label is a mandatory field. Design Studio reports an error if this label has no value. The use of a command phrase as the label string is recommended. Some example labels are:

- **Correct in Inventory System**

- **Open a Trouble Ticket**

The label input field allows you to choose either a label from another discrepancy resolution action defined within your workspace, or to type in a new label. A label can be shared by multiple actions; this implies that multiple actions are sharing a single menu item in the Actions menu of the Discrepancies page.

Network Integrity determines the correct action to invoke based on a combination of the label and the result source.

> **Note:**   No two discrepancy resolution actions can have the same label and the same result source.

About Result Sources

The result source is a list of discrepancy filtering criteria. Each criterion represents a single source of discrepancy, and is specified by a combination of the originating scan action and a result category. A resolution action only receives discrepancies from the specified result categories which were created by scans using the specified actions.

Figure 2–11 shows an example of result sources being applied in Network Integrity.

Figure 2–11   Result Source Example 1
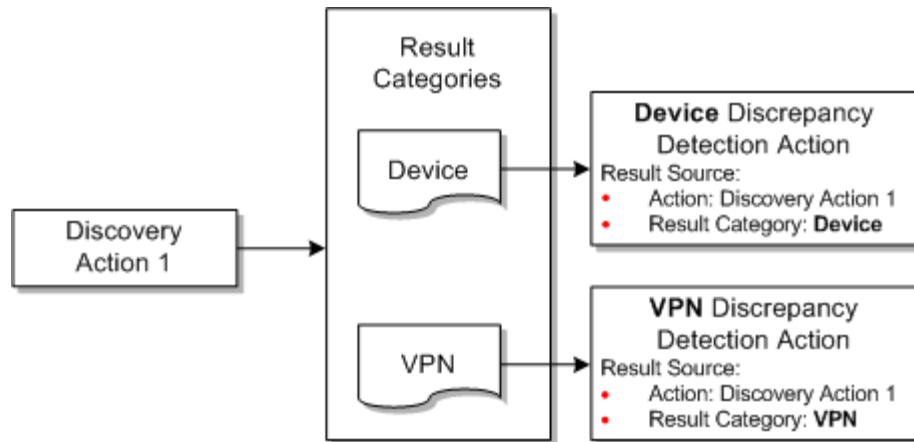
A criterion that does not specify any result category matches all result categories generated by the scan action in the criterion.

Figure 2–12 shows a representation of the discrepancy types.

Figure 2–12   Result Category Example
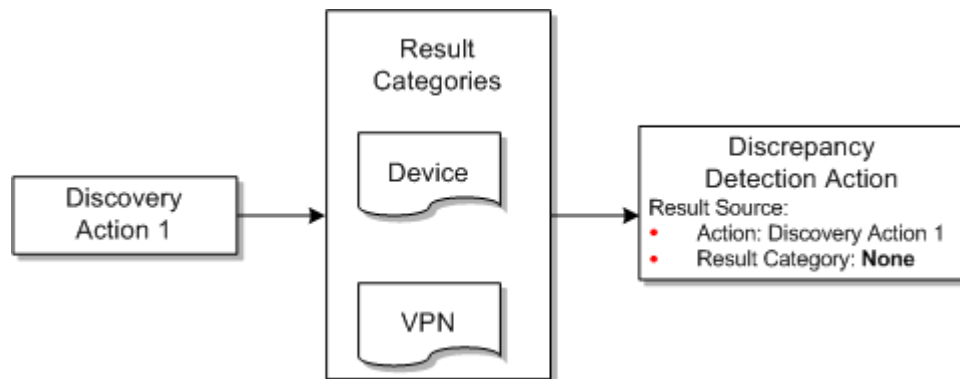
The result source is a mandatory field; there must be at least one entry in the table. Design Studio marks a discrepancy resolution action with an error during a project build if this table has no entries.

> **Note:**   No two discrepancy resolution actions can have the same label and the same result source.

Generated Action and MDB Controller

The discrepancy resolution action is implemented as an MDB, just like any other Network Integrity action.

See "About the Generated Action MDB and Controller" for more information.

3

Working with Processors

This chapter provides information about Oracle Communications Network Integrity processors. This chapter contains the following sections:

This chapter contains the following sections:

- About Processors

- Implementing a Processor

- Implementing an Import Processor

- Implementing a Discovery Processor

- Implementing the SNMP Processor

- Implementing an Assimilation Processor

- About Discrepancy Detection Processors

- About Discrepancy Resolution Processors

About Processors

In Network Integrity, processor entities are the building-blocks for actions, as they implement atomic sub-functions for actions.

For example, an SNMP processor is included in an action to poll network devices; a modeler processor is included in an action to model raw SNMP data from a network device and add it to a database. Combined, these two processors comprise a discovery action that polls SNMP-enabled network devices and persists the modeled SNMP data.

By adding multiple processors to an action, the action performs several complex function by executing the processors according to the sequence in which they were added to the action.

Processors are of different types:

- **Import processor**: Part of an import action.

- **Discovery processor**: Part of a discovery action that can discover anything.

- **SNMP processor**: Part of a discovery action that is prebuilt to discover only SNMP-enabled devices.

- **Assimilation processor**: Part of an assimilation action.

- **File transfer processor**: Used to retrieve files from local or remote directories. For more information, see *Network Integrity File Transfer and Parsing Guide*.

- **File parsing processor**: Used to parse data retrieved by the File Transfer processor so that the data is available to other processors. For more information, see *Network Integrity File Transfer and Parsing Guide*.

- **Discrepancy detection processor**: Part of a discrepancy detection processor action.

- **Discrepancy resolution processor**: Part of a discrepancy resolution action.

Unlike actions, processors are not visible in Network Integrity.

About Context Parameters

Configure input and output parameters for processors.

Input and output parameters are optional for a processor.

After adding input and output parameters for the processor, Oracle Communications Design Studio generates the request and response Java classes based on the input and output parameters.

Specifying Context Parameters before Creating Implementation Class

When creating a processor, it is a good practice to properly configure the context parameters before saving the processor. This way Design Studio properly generates the skeleton implementation Java class for the processor with the correct input and output parameters. If the input and output context parameters are modified later, the generated Interface changes, but Design Studio does not automatically update the implementation class. The user must manually update the implementation class to comply with the changed interface.

About Properties and Property Groups

A property group is a logical container configured on a processor. A property group can be added to multiple processors. Property group names must be unique within a processor.

Properties are added to property groups and are assigned property values to pass to the processor.

Property groups do not inherently pass any values to the processor other than the values belonging to its properties.

Property groups and properties are configured on processors on the **Properties** tab of the Processor editor.

Property groups can be configured as Managed groups, where the values for the properties it contains can be set at run time using the MBean interface. See *Network Integrity System Administrator's Guide* for more information. Only managed groups can contain sensitive properties.

Property groups can be configured as Map groups, where the property group produces a simplified API for properties that are used as maps.

Design Studio generates a Java class for the property group so that you can extend a cartridge to access the property values it contains using getter and setter methods.

A property consists of a name-value pair that is passed to the processor through the property group. Property names must be unique within the property group.

The property value can be set in the following ways:

- At design time, by setting the property with a static value.

- At deployment time, by setting the property with a cartridge model variable.

- At run time, using the MBean interface, by configuring its property group as a managed group.

You can configure properties as sensitive. To be configured as sensitive, the properties must be contained in managed property groups and their values must be encrypted. See *Network Integrity System Administrator's Guide* for information about how to encrypt property values.

You can set the encrypted value of a sensitive property with a model variable at deployment time, or you can set it at run time using the MBean interface.

For more information about setting sensitive properties, see the Design Studio Modeling Network Integrity Help.

For more information on adding property groups to a processor, adding properties to a property group, and setting cartridge model variables, see the Design Studio Help.

About Generated Code

This section describes code generation for processors in Network Integrity:

- About the Location for Generated Code
- About the Processor Interface
- About the PropertyGroup and Properties Classes

About the Location for Generated Code

Design Studio code-generates the relevant Java classes for the processor. The generated code is located at:

Studio_Workspace*NI_Project_Root***generated****src***Project_Default_Package**Processor_Type**Processor_Implementation_Prefix*

where:

- *Studio_Workspace* is the Eclipse Workspace root

- *NI_Project_Root* is the Network Integrity project root

- *Project_Default_Package* is the default package configured in the Project editor

- *Processor_Type* is run time following action types:

  – discoveryprocessors

  – importprocessors

  – assimilationprocessors

  – detectionprocessors

  – resolutionprocessors

- *Processor_Implementation_Prefix* is the action implementation prefix in lowercase.

About the Processor Interface

Every processor has a generated interface. The generated processor interface class is named *Processor_Name***ProcessorInterface.java**.

In general, the generated processor interface has the **invoke** method defined. The interface has two forms of **invoke** methods, depending on whether there is an output parameter defined for the processor.

```
// Signature for processor which does not have output parameters
public void invoke(<Processor_Specific_Context> context,
              ExampleProcessorRequest request) throws ProcessorException {
      // TODO Auto-generated method stub

// Signature for processor which has output parameters
public ExampleProcessorResponse invoke(<Processor_Specific_Context> context,
              ExampleProcessorRequest request) throws ProcessorException {
      // TODO Auto-generated method stub
      return null;
}
```

The generated processor interface has a slightly different signature, depending on the type of processor: for example, *Processor_Specific_Context* differs between processor types. See individual chapters on specific processors for more information.

About the PropertyGroup and Properties Classes

A properties class is always code-generated for the processor, whether the processor has property groups and properties configured or not. The properties class is used as an input parameter for the constructor of the generated request class.

The generated properties class is named *Processor_Name***ProcessorProperties.java**.

The generated properties class has a public method, **String[] getValidProperties()**. This method returns a string array that contains a list of valid property group names configured for this processor. If the processor has no property groups configured, this method returns an empty array.

If the processor has property groups and properties configured, for each property group a PropertyGroup class is code-generated.

The generated PropertyGroup class is named *PropertyGroup_Name***PropertyGroup.java**.

The generated PropertyGroup class represents the configured property group and all of its properties. The generated properties class has the getter methods to get each PropertyGroup directly, and has all the setter methods to modify the property values.

The generated PropertyGroup class has a public method, **String[] getValidProperties()**. This method returns a string array that contains a list of valid properties names configured for this property group. If the property group has no property configured, this method returns an empty array.

If the property group is not configured as a Map group, the generated PropertyGroup class provides getter methods for all the properties configured in this property group.

If the property group is configured as a Map group, the generated PropertyGroup class does not provide getter methods for all the properties configured in this property group. Instead, the API for the property group resembles a Java Map, where the property values are retrieved and set using the property name passed as a value.

Implementing a Processor

Implementing a processor is done in the Processor editor **Details** tab. See the Design Studio Help for specific configuration details.

You can click the **Implementation Class** link to open the Java editor for this implementation Java class. Design Studio auto-generates the skeleton Java implementation class, which implements the processor interface with an empty implementation method.

You must decide whether to complete implementing the method. If you modify the processor (for example, by adding output parameters or removing parameters), the implementation class displays a compiling error. This is expected because the skeleton implementation class is regenerated. You must modify the implementation class to match the changed processor interface.

When you delete a processor, you must manually delete the implementation class of the processor. Design Studio does not automatically delete an implementation class when you delete a processor.

For information about how to implement a processor, see the individual processor section.

About the Processor Finalizer

When a processor deals with resources (for example, sockets and files), it is necessary to clean up the resources used or created while the processor executes. Using a finalizer on the processor ensures that the used or created resources get cleaned up, whether the action fails or is successful. When implemented, the finalizer cleans up the resources used or created by the processor. It is not mandatory to implement the finalizer if the processor does not deal with a resource, or if the resource is used only within the processor (in which case the processor implementation should make sure the local resource is closed properly). The processor must implement the finalizer if the processor allocates a resource that is to be output for use by other processors.

Finalizers that are not inside a For Each loop are called by the action controller class (code-generated) before it completes. Finalizers that are inside a For Each loop are called by the action controller class at the end of the For Each loop. In all cases, finalizers are called in the reverse order to which they are registered (finalizers registered first are called last; finalizers registered last are called first).

About the ProcessorFinalizer Interface

The processor implementation class must implement the interface **oracle.communications.sce.integrity.sdk.processor.ProcessorFinalizer** to have the action controller clean up the resources that are used or created by the processor. If a processor does not use or create a resource, it does not implement the ProcessorFinalizer interface.

The processor defines only one method:

```
public void close(boolean failed);
```

The processor that implements the ProcessorFinalizer interface must implement this method to close all the resources used or created during the execution of this processor. This method takes an input parameter as Boolean. If there is an exception during the execution of the processors, the action controller calls the finalizer by passing **True** to this method; otherwise the action controller calls the finalizer by passing **False** to the method, in the successful case. The processor might implement the close logic differently for both successful and failed scenarios: for example, if it is a failed scenario, the close method might log an error message before closing the resources.

The following code shows how to implement the ProcessorFinalizer for a sample processor:

```
public class SampleProcessorImpl implements SampleProcessorInterface,
ProcessorFinalizer {
        public SampleProcessorResponse invoke(SampleProcessorRequest request)
                throws ProcessorException {
```

```
                          // Implement the Processor here…
               }

          public void close(boolean failed) {
                    if(failed) {
                              // something is failed, log extra error message here.
                    }
                    // close the InputStream here.
                    try {
                              myInputStream.close()
                    } catch(IOException ioe) {
                              // log the IOException here…
                    }
          }
     }
}
```

About Memory Considerations

The action controller class calls the finalizers for both successful and failed scenarios. The finalizers that are not inside a For Each loop do not begin until the end of the action. The finalizers that are inside a For Each loop do not begin until the end of the loop. When a processor that implements the ProcessorFinalizer completes the execution, it is still in the scope of the action. The processor does not get purged by the garbage collector to release the memory.

If a processor implements the ProcessorFinalizer, it is a good practice to limit the number of member variables for that processor and ensure that the processor is not using a large amount of memory. If the processor uses a lot of memory, it is a good practice to release the memory as soon as it is no longer required. For example, if a processor is using a large HashMap, and it also implements the ProcessorFinalizer, the processor should clear the contents of the HashMap when it is done using it and assign the null pointer to this HashMap.

Implementing an Import Processor

Many deployment artifacts for the import action and its processors are generated automatically while editing. However, you must supply implementations for the import processors using the **invoke** method.

Two forms of this method are shown in the following code fragments:

```
// Signature for processor which does not have output parameters
public void invoke(DiscoveryProcessorContext context,
               ExampleProcessorRequest request) throws ProcessorException {
      // TODO Auto-generated method stub

}
// Signature for processor which has output parameters
public ExampleProcessorResponse invoke(DiscoveryProcessorContext context,
               ExampleProcessorRequest request) throws ProcessorException {
      // TODO Auto-generated method stub
      return null;
}
```

The parameters and return type of the **invoke** method are:

- *Processor_Name*ProcessorResponse: This is the return type, for processors that have output parameters. For processors that do not have output parameters, the return type is void. This class is generated by Design Studio. It is a value object

containing values for each of the processor's output parameters. For processors that have output parameters, the **invoke** method must create a ProcessorResponse object, set its values and return the ProcessorResponse object.

- *Processor_Name*ProcessorRequest: This is a value object that has the following getters:

    – If scan parameter groups are specified for the import action, there is a getter that returns a scan parameter groups value object.

    – If properties are defined for the import processor, there is a getter that returns a *Processor_Name*ProcessorProperties value object.

    – There is a getter for each input parameter that is defined for the processor.

    – There is a getter method called **getScopeAddress**. This method is not useful for import processor implementation. Instead, the inventory system address and authentication information should be retrieved using the POMS API.

        See "Working with the POMS SDK" for more information.

    This class is generated by Design Studio.

- DiscoveryProcessorContext context: This is an SDK type that has the following methods:

    – **getActionName**: Returns the name of the action that the processor is executing under.

    – **getProcessorName**: Returns the name of the processor.

    – **persistResults**: Causes POMS objects to be flushed to the database. This helps to reduce memory consumption. See "About Persist Results" for more information.

    – **addToResult**: Adds a graph of POMS objects to the database under a result group. This method takes three parameters:

        * String resultGroupName: this is the name of a result group under which the results are persisted.

        * String resultGroupType: this is the type of the result group under which the results are persisted. This should match a category defined on the action.

        * DiscrepancyEnabled result: this is the root of result object graph to be persisted.

    – **getResultGroup**: Used to get an existing result group from your current scan if you must access the graph of POMS objects previously added to a result group. This method takes two parameters:

        * String resultGroupName: This is the name of a result group under which the results are persisted.

        * String resultGroupType: This is the type of result group under which the results are persisted. This should match a category defined on the action.

Implementing a Discovery Processor

Configuration of the discovery action and its discovery processors results in the generation of many deployment artifacts. However, you must supply implementations for the discovery processors.

The implementation needs to implement the **invoke** method. Two forms of this method are shown:

```
// Signature for processor which does not have output parameters
public void invoke(DiscoveryProcessorContext context,
                ExampleProcessorRequest request) throws ProcessorException
{
 // TODO Auto-generated method stub
}
// Signature for processor which has output parameters
public ExampleProcessorResponse invoke(DiscoveryProcessorContext context,
                ExampleProcessorRequest request) throws ProcessorException
{
 // TODO Auto-generated method stub
 return null;
}
```

The parameters and return type of the **invoke** method are:

■ *Processor_Name*ProcessorResponse: This is the return type, for processors that have output parameters. For processors that do not have output parameters, the return type is void. This class is generated by Design Studio. It is a value object containing values for each of the processor's output parameters. For processors that have output parameters, the **invoke** method must create a ProcessorResponse object, set it values and return the ProcessorResponse object.

■ *Processor_Name*ProcessorRequest: This is a value object that has the following getters:

  – If scan parameter groups are specified for the discovery action, there is a getter that returns a scan parameter groups value object.

  – If properties have been defined for the discovery processor, there is a getter that returns a *Processor_Name*ProcessorProperties value object.

  – There is a getter method for each input parameter that is defined for the processor.

  – There is a getter method named **getScopeAddress()**. This method returns the scope address configured for this discovery action.

  This class is generated by Design Studio.

■ DiscoveryProcessorContext context: This is an SDK type, which has the following methods:

  – getActionName: Returns the name of the action that the processor is executing under.

  – getProcessorName: Returns the name of the processor.

  – persistResults: Causes POMS objects to be flushed to the database. This helps to reduce memory consumption. See "About Persist Results" for more information.

  – addToResult: Adds a graph of POMS objects to the database under a result group. This method takes three parameters:

    * String resultGroupName: This is the name of a result group under which the results are persisted.

    * String resultGroupType: This is the type of the result group under which the results are persisted. This should match a category defined on the action.

           *   DiscrepancyEnabled result: This is the root of result object graph to be persisted.

     –   getResultGroup: Used to get an existing result group from your current scan if you must access the graph of POMS objects previously added to a result group. This method takes two parameters:

           *   String resultGroupName: This is the name of a result group under which the results are persisted.

           *   String resultGroupType: This is the type of result group under which the results are persisted. This should match a category defined on the action.

Implementation Code Example

The following Java code snippet demonstrates how to implement the **invoke** method for a discovery processor, and how to add results to the result group using the **addToResult()** method.

```
public SampleProcessorResponse invoke(
         DiscoveryProcessorContext context,
         SampleProcessorRequest request) throws ProcessorException {
    SampleProcessorResponse modelerResponse = new SampleProcessorResponse();
    SampleDevice device;

    // Get the input Sample Response Document from the Request.
    // This input response document models the sample device.
    SampleResponseType response = request.getSampleResponseDocument();

    try {
            // Make the Sample Device
            device = makeSampleDevice(response);
            // Add the device to the result group "Device", which matches
            // the result category configured in the Discovery Action.
            context.addToResult(device.getName(), "Device", device);
            modelerResponse.setSampleDevice(device);

    } catch (Exception e) {
            // Handle exception here…
    }
    return modelerResponse;
}
```

Implementing the SNMP Processor

There is no coding required for the SNMP processor. The Processor Interface, Request/Response, Properties, and the relevant helper classes of an SNMP processor are all code -generated and fully implemented.

The only configuration required for the SNMP processor is to configure the list of polled object IDs (OIDs). Before configuring the OIDs for the SNMP processor, the MIB directory must be properly specified for the Network Integrity preference. If the MIB directory is not properly specified in the preference, you cannot configure the SNMP processor.

See the Design Studio Modeling Network Integrity Help for more information about configuring SNMP processors.

About the Generated Implementation and XML Beans

The SNMP processor is a completely code-generated discovery processor. Along with the usual discovery processor implementations (see "Implementing a Discovery Processor"), Design Studio also generates the strongly-typed SNMP XML response document schema based on the OIDs configured for the SNMP processor.

The generated SNMP XML response document schemas are available at the following directory:

Project_Root**generated***SNMP_Processor_Name_*snmpdiscoveryprocessor.

Under this directory, the following sub-directories exist:

- lib: Contains the compiled XML Beans JAR file for the strongly-typed SNMP XML response document schemas

- snmpClasses: Contains the XML Beans Java classes for the strongly-typed SNMP XML response document schemas

- snmpSchemas: Contains the generated strongly-typed SNMP XML response document schemas

- xmlSrc: Contains the compiled XML Beans Java source for the generated strongly-typed SNMP XML response document schemas.

It is recommended to first look at the schemas generated in this directory to understand how to access the compiled XML Beans object for the SNMP response document.

The remaining implementations for the SNMP processor are at the following directory:

Studio_Workspace*NI_Project_Root***generated****src***Project_Default_Package*\snmpdiscoveryprocessors*SNMP_Processor_Implementation_Prefix*

The SNMP processor always has an output parameter, which is the SNMP XML response document (XML Beans object). This is available in the Response class for the SNMP processor.

Supporting New MIBs

When the productized Network Integrity cartridges are imported into Design Studio (see "Exporting and Importing Cartridges"), Network Integrity cartridges are bundled with a set of MIB files, which is the same set of MIB files bundled with the SNMP Resource Adapter (see "Working with JCA Resource Adapters").

If you must create a Network Integrity cartridge to poll certain MIB OIDs for certain specific devices, which are not part of the bundled MIB files, you must get the MIB file (or set of MIB files) that has the definitions of those MIB OIDs required to implement the new cartridge.

The new MIB files must be manually copied to the MIB directory configured in the Design Studio preference (see the Design Studio Modeling Network Integrity Help). After the new MIB files are copied to the MIB directory, the new MIB files are available to be loaded in Design Studio. There is no need to restart Design Studio.

> **Note:** The MIB files in Design Studio and on the SNMP resource adapter must match. See "Working with JCA Resource Adapters" for information about supporting new MIBs for the SNMP resource adapter.

Implementing an Assimilation Processor

Many deployment artifacts for the assimilation action and its processors are generated automatically while editing. However, you must supply implementations for the assimilation processors using the **invoke** method.

Two forms of this method are shown in the following code fragments:

```
// Signature for processor which does not have output parameters
public void invoke(AssimilationProcessorContext context,
                ExampleProcessorRequest request) throws ProcessorException {
      // TODO Auto-generated method stub

}
// Signature for processor which has output parameters
public ExampleProcessorResponse invoke(AssimilationProcessorContext context,
                ExampleProcessorRequest request) throws ProcessorException {
      // TODO Auto-generated method stub
      return null;
}
```

The parameters and return type of the **invoke** method are:

- *Processor_Name*ProcessorResponse: This is the return type, for processors that have output parameters. For processors that do not have output parameters, the return type is void. This class is generated by Design Studio. It is a value object containing values for each of the processor's output parameters. For processors that have output parameters, the **invoke** method must create a ProcessorResponse object, set its values and return the ProcessorResponse object.

- *Processor_Name*ProcessorRequest: This is a value object, which has the following getters:

  - If scan parameter groups are specified for the assimilation action, there is a getter that returns a scan parameter groups value object.

  - If properties are defined for the assimilation processor, there is a getter that returns a *Processor_Name*ProcessorProperties value object.

  - There is a getter for each input parameter that is defined for the processor.

  This class is generated by Design Studio.

- AssimilationProcessorContext context: this is an SDK type, which has the following methods:

  - **getActionName**: Returns the name of the action under which the processor is executing

  - **getProcessorName**: Returns the name of the processor

  - **persistResults**: Causes POMS objects to be flushed to the database. This helps to reduce memory consumption. See "About Persist Results" for more information.

  - **addToResult**: Adds a graph of POMS objects to the database under a result group. This method takes three parameters:

    * String resultGroupName: This is the name of a result group under which the results are persisted.

    * String resultGroupType: This is the type of the result group under which the results are persisted. This should match a category defined on the action.

* DiscrepancyEnabled result: This is the root of result object graph to be persisted.

- **getLatestReultGroupsInScope**: Returns an Iterator*DisResultGroup,* which is the latest results in scope. This is essentially the discovery or assimilation scan inputs to the assimilation action.

- **getLatestScanRunsInScope**: Returns an Iterator*DisScanRun,* which is the latest scan runs in scope.

  This is also essentially the discovery or assimilation scan inputs to the assimilation action but includes several other objects from the Network Integrity model.

  These additional Network Integrity model objects might be useful in performing out assimilation processing in some cases.

- **getPreviousAssimilationScanRun**: Returns the latest completed scan run for the current assimilation scan. Use this to look at previous results, comparing current scope with previous scope.

- **haveAllLatestScansInScopeChanged**: Returns *true* if any of the following conditions are met; *false* otherwise:

  * This is the first scan run for the assimilation scan.

  * The latest scan run of every scan that is in the scope of both the previous assimilation run and the current assimilation run is more recent than the previous assimilation run.

- **haveLatestScanInScopeChanged**: Returns *true* if any of the following conditions are met; *false* otherwise:

  * This is the first scan run for the assimilation scan.

  * At least one scan run in scope is more recent than latest assimilation scan run.

  * The scope of the assimilation scan has changed between this run and the previous run.

  This function avoids unnecessary assimilation processing.

- **getResultGroup**: Used to get an existing result group from your current scan if you need to access the graph of POMS objects previously added to a result group. This method takes two parameters:

  * String resultGroupName: This is the name of a result group under which the results are persisted.

  * String resultGroupType: This is the type of result group under which the results are persisted. This should match a category defined on the action.

About Discrepancy Detection Processors

The discrepancy detection processor is the atomic sub-function of a discrepancy detection action. The typical tasks of a detection processor are different than the scan-related processors (discovery, import, and assimilation) and include the following:

- Create and add filters to alter the default behavior of the base discrepancy detection action.

- Perform post-processing on the set of discrepancies produced by the base discrepancy detection action.

See "Discrepancy Detection Processor Patterns" for more information about the various patterns for detection action-processor implementation.

Discrepancy Detection Processor Patterns

There are several patterns of processor used inside a discrepancy detection action. Each successive pattern introduces a new level of flexibility, power, and complexity. The patterns are listed below, in order from the simplest to the most complex:

1. Reusing the base detect discrepancy action.

2. Adding new filters and handlers.

3. Adding post-processors.

Reusing the Base Detect Discrepancy Action

This usage pattern provides a baseline comparison algorithm between the compare and the reference sides. A discrepancy detection action using this pattern has the ability to compare exact entity attributes and associations, and can generate five of the seven types of discrepancy. (Ordering Errors and Association Ordering Errors are not detected by the baseline comparison algorithm, because it assumes that there are no ordered relationships.)

To use this pattern, use following steps:

1. Create a discrepancy detection action.

2. Add the Detect Discrepancies action as a processor. The Detect Discrepancies action belongs to the NetworkIntegritySDK project, which all Network Integrity cartridge project are dependent on by default.

3. Set the result source.

See the Design Studio Modeling Network Integrity Help for information about the tasks above.

About the Base Detection Project and the Default Comparison Algorithm

The Base Detection project contains a reusable discrepancy detection action called Detect Discrepancies. This discrepancy detection action is abstract and cannot be deployed by itself. It is intended to be imported by virtually all other discrepancy detection actions. The Detect Discrepancies action implements a general comparison algorithm that can work with all entity types and specifications, and can detect and report all seven types of discrepancy.

This ability enables a cartridge developer to build a working discrepancy detection cartridge for arbitrary discovered data without writing code. Its behavior is customizable, by using the techniques described in the following processor patterns.

The default comparison algorithm is outlined below.

1. The detector loops over the compare root entities.

2. The detector checks if each compare root entity should be considered for discrepancy detection. If it should not, the root entity is ignored, and the detector begins processing the next compare.

3. A **rootEntityHandler** finds the matching reference root entity for the compare root entity. The default **rootEntityLoader** uses the **Name** field to find the matching

reference root entity. If no reference root entity is found, an EXTRA_ENTITY discrepancy is generated.

4. The attributes of the matching entities are compared, and an ATTRIBUTE_VALUE_MISMATCH discrepancy is generated for each attribute with different values. If an attribute contains an ordered list of values, an ORDERING_ERROR discrepancy is generated if the order of the values does not match.

5. The associations of the matching entities are compared, and an EXTRA_ASSOCIATION or MISSING_ASSOCIATION discrepancy is generated for unmatched target entities of an association. The default relationship handler uses the **Name** field to match related entities of the compare and reference sides. If an association is an ordered association, an ASSOCIATION_ORDERING_ERROR discrepancy is generated if the order of the matching associated entities is different.

6. The child relationship of the matching entities is compared, and an EXTRA_ENTITY or MISSING_ENTITY discrepancy is generated for unmatched child entities. The default relationship handler uses the **Name** field to match child entities of the compare and reference sides. If a child relationship is an ordered association, then an ORDERING_ERROR discrepancy is generated if the order of the matching child entities is different.

7. The comparison continues by applying the above algorithm to all children entities recursively, until all entities have been checked. The comparison also stops at a given entity if one of the following is true: the entity is a compare root entity, or the entity is flagged as a shadow entity.

The Detect Discrepancy action creates discrepancies with a default severity of CRITICAL for EXTRA_ENTITY and MISSING_ENTITY, and WARNING for the other types.

Adding New Filters and Handlers

This usage pattern builds on the Reuse pattern by adding filters and handlers to customize the general comparison algorithm. The following changes can be achieved:

■ Which root discovery entities are of interest.

■ How to match discovery entities to import entities.

■ Which attributes are not significant for a particular entity type.

■ How to compare a particular attribute.

■ Which relationships to consider for a particular entity type.

■ What severity to apply to a discrepancy.

■ Define a relationship as ordered (to automatically add ORDERING checks).

■ Set a default/suggested resolution action (such as Ignore or Correct in UIM).

To use this pattern, follow the Reuse pattern to create your detection action, and then create one new detection processor, and move it above the discrepancy detector processor in the table. This new processor becomes the filter initializer processor for the detection action. (For example, in Figure 3–1, a new action follows this pattern by having its own Sample Filter Initializer processor placed above the imported discrepancy detector processor.)

Figure 3–1   Sample Filter Initializer
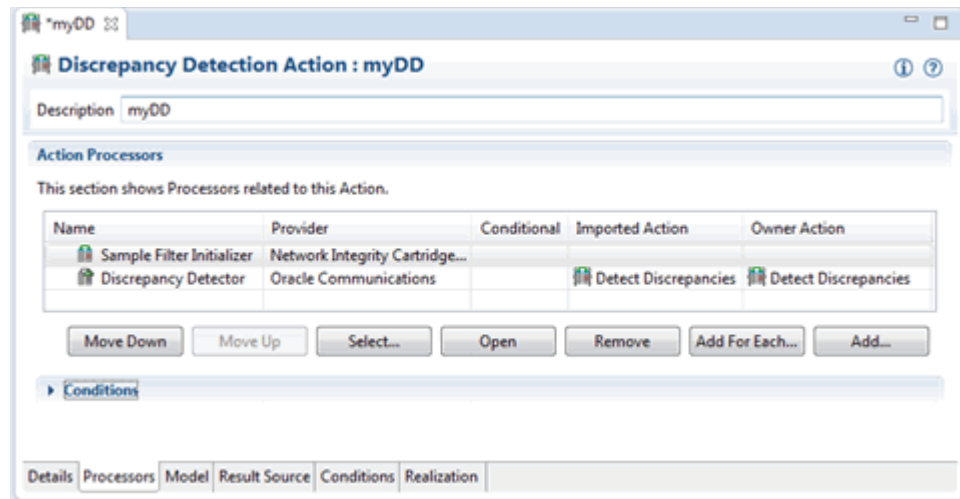
The main task of a filter initializer Processor is to register filters and handlers for use by the subsequent discrepancy detector processor. Handlers are code that implements various behaviors used during discrepancy detection. Filters are code that manipulates the handlers to be used by discrepancy detection.

About Filters

There are four different types of filters that can be added by the processor:

- **AttributeFilter**: This filter is called during the assignment of attribute handlers for the given entity type. This filter can add, modify and remove handlers from the given **attributeHandlers**.

- **RelationshipFilter**: This filter is called during the assignment of relationship handlers for a given entity type. This filter can add, modify and remove handlers from the given **relationshipHandlers**.

- **DiscrepancyFilter**: This filter is called during assignment of discrepancy handlers for a given entity type. This filter can modify or remove the default **discrepancyHandler**.

- **RootEntityFilter**: This filter is called during the assignment of the root entity handler for a given entity type. This filter can replace the default **rootEntityHandler** with another one.

About Handlers

There are four types of handlers that can be manipulated by their associated filters:

- **AttributeHandler**: This handler can change the mapping of attributes, or change the behavior of the comparison operation. For example, a string comparison is normally case-sensitive. An **attributeHandler** can be added to cause a case-insensitive comparison to be used instead.

  Network Integrity provides a **DefaultAttributeHandler** class which implements the necessary **AttributeHandler** interface and the default case-sensitive string comparison behavior. To override this behavior, create a class which subclasses **DefaultAttributeHandler**, and then override the following method:

  ```
  protected boolean equalsNonNull(Object a1, Object a2);
  ```

- **RelationshipHandler**: This handler can change the mapping of relationships. For example, a relationship comparison would normally check the identically-named relationship on the reference entity. A relationshipHandler can be added which causes a differently-named relationship to be used instead.

  Network Integrity provides a **DefaultRelationshipHandler** class that implements the necessary **RelationshipHandler** interface, and has knowledge of all relationships for each supported Oracle Communications Information Model entity type. The following method can be overridden by a new subclass to alter the default behavior.

  ```
  protected Object getKey(DiscrepancyEnabled entity)
  ```

  This method gets a key value that distinguishes a single entity from a set of entities within a single relationship. The **DefaultRelationshipHandler** implementation returns the value of the Name attribute for the input entity.

- **DiscrepancyHandler**: This handler can change the fields of a discrepancy immediately after it is generated. It can also completely remove the discrepancy. An example of its use is to adjust the severity value of a discrepancy of a DeviceInterface entity based on its Speed value.

  Network Integrity provides a non-accessible default **DiscrepancyHandler** implementation which does nothing. To override this behavior, create a class which implements the **DefaultHandler** interface, and implement the following method.

  ```
  DisDiscrepancy processDiscrepancy(DiscrepancyEnabled currentEntity,
                                    DisDiscrepancy generatedDiscrepancy)
  ```

  The overridden method should alter the input **generatedDiscrepancy**, and then return it.

- **RootEntityHandler**: This handler changes the algorithm for finding a matching reference entity for an input compare entity. An example of its use is to change the default comparison criteria to using the **ID** field to find the match, instead of the default of using **Name** field.

  See "Using Root Entity Filter and Handler" for a full example of the proper setup and usage of a root entity handler.

Filters and CimType

Filters register against one or more types of Information Model entities produced by a Discovery, Import, or Assimilation scan. Filters can also register against one of more specifications of an entity type, for more fine-grained control.

In Java code, the entity type and specification are designated by using the class CimType. To register a filter against an entity type (for example, Equipment), use the single parameter constructor for CimType:

```
CimType eqType = new CimType(Equipment.class);
```

To register a filter against a particular specification (for example, **cevSensorClock**, an Equipment specification defined in the Cisco UIM cartridge), use the two-parameter constructor for CimType:

```
CimType clockEqType = new CimType(Equipment.class, "cevSensorClock");
```

It is possible to take advantage of the inheritance model of the Information Model entity classes to register quickly against several classes with one call. For example, all

Information Model entities that support discrepancy detection inherit from the class **DiscrepancyEnabled**. Therefore, the following code CimType can register a filter against everything:

```
CimType allType = new CimType(DiscrepancyEnabled.class);
```

Filter and Handler Examples

The following examples demonstrate the types of filters and handlers. The prerequisite tasks for all examples are to:

1. Create a discrepancy detection action.

2. Set the result source.

3. Add the detect discrepancy action as a processor.

4. Create a filter initializer processor.

5. Move the new processor above the discrepancy detector processor.

Using Attribute Filter and Handler (Static Attribute)

The following code fragments shows how to add an attribute filter to ignore the static attribute **description** on LogicalDevices. The result of this code is that the new detection action does not generate any description Attribute Value Change discrepancies on LogicalDevices.

1. Define the filter class and remove the handler for the attribute **description**.

```
private class LogicalDeviceAttributeFilter implements AttributeFilter {
  public void filterAttributes(CimType cimType, Map<String, AttributeHandler>
attributeHandlers) {
     attributeHandlers.remove("description");
  }
}
```

2. In the processor **invoke** method, get the generic discrepancy detector from the context.

```
GenericDiscrepancyDetector detector = context.getDiscrepancyDetector();
```

3. In the **invoke** method, create the CIMType object to name the entity type, and add the custom filter.

```
CimType ldType = new CimType(LogicalDevice.class);
detector.addFilter(ldType, new LogicalDeviceAttributeFilter());
```

Using Attribute Filter and Handler (Characteristic)

The following code fragments show how to add an attribute filter to ignore the characteristic **systemObjectId** on LogicalDevice entities with the specification **DemoLogicalDevice**. The main difference between this example and the previous example is step 3, where the specification name must be included in the CimType constructor.

1. Define the filter class and remove the handler for the attribute **systemObjectId**.

```
private class DemoLogicalDeviceAttributeFilter implements AttributeFilter {
    public void filterAttributes(
                   CimType cimType,
                   Map<String, AttributeHandler> attributeHandlers) {
        attributeHandlers.remove("systemObjectId");
    }
}
```

2. In the processor **invoke** method, get the generic discrepancy detector from the context.

```
GenericDiscrepancyDetector detector = context.getDiscrepancyDetector();
```

3. In the **invoke** method, create the CIMType object to name the entity type and the specification, and add the custom filter.

```
CimType ldType = new CimType(LogicalDevice.class, "DemoLogicalDevice");
detector.addFilter(ldType, new DemoLogicalDeviceAttributeFilter());
```

Using Relationship Filter and Handler

In this example, the discrepancy detection action skips the **physicalPorts** relationship of all Equipment entities. By using the following code fragment, the new detection action no longer examines any children ports of equipment.

1. Define the filter class and remove the relationship handler for the relationship **physicalPorts**.

```
private class EquipmentRelationshipFilter implements RelationshipFilter {
    public void filterRelationships(
                    CimType cimType,
                    Map<String, RelationshipHandler> relationshipHandlers) {
        relationshipHandlers.remove("physicalPorts");
    }
}
```

2. In the processor **invoke** method, get the generic discrepancy detector from the context.

```
GenericDiscrepancyDetector detector = context.getDiscrepancyDetector();
```

3. In the **invoke** method, create the CIMType object to name the entity type, and add the custom filter.

```
CimType eqType = new CimType(Equipment.class);
detector.addFilter(eqType, new EquipmentRelationshipFilter());
```

Using Discrepancy Filter and Handler

This example sets the severity to Minor on every Missing Entity and Extra Entity discrepancy generated by the new detection action. Use the following code fragment for this task:

1. Define the filter class and add a new discrepancy handler. This handler performs a discrepancy type check, and sets the severity accordingly.

```
private class CustomDiscrepancyFilter implements DiscrepancyFilter {
    public DiscrepancyHandler filterDiscrepancies(
                    CimType cimType,
                    DiscrepancyHandler handler) {
        return new DiscrepancyHandler() {
            public DisDiscrepancy processDiscrepancy(
                DiscrepancyEnabled cimBase,
                DisDiscrepancy disDiscrepancy) {
                    if (DisDiscrepancyType.EXTRA_ENTITY ==
                        disDiscrepancy.getType()
                        ||
                        DisDiscrepancyType.MISSING_ENTITY ==
                        disDiscrepancy.getType()) {
                        disDiscrepancy.setSeverity(DisDiscrepancySeverity.MINOR);
```

```
            }
            return disDiscrepancy;
        }
    }; // end return new()
    }
}
```

2. In the processor **invoke** method, get the generic discrepancy detector from the context.

    ```
    GenericDiscrepancyDetector detector = context.getDiscrepancyDetector();
    ```

3. In the same **invoke** method, create the CIMType object to name the entity type, and add the custom filter.

    ```
    CimType allType = new CimType(DiscrepancyEnabled.class);
    detector.addFilter(allType, new CustomDiscrepancyFilter());
    ```

Using Root Entity Filter and Handler

This advanced technique in this example changes the matching algorithm that finds the matching reference entity for any compare entity. The default algorithm finds matches based on a comparison of the value of the **name** attribute. This example changes the comparison to use the **nativeEmsName** attribute instead.

> **Note:** This feature is used in the MIB II UIM cartridge.

The example is in two parts. The first part alters the root entity handler to match compare root entities with reference root entities using the **nativeEmsName** attribute. The second part use relationship handlers to make the discrepancy detector use **nativeEmsName** attribute to distinguish the children.

First, the root entity filter and handler code fragments are as follows:

1. Define a method in the new processor to create the root entity filter. This filter creates a new root entity handler and returns it.

    ```
    private RootEntityFilter getRootEntityFilter() {
        return new RootEntityFilter() {
            @Override
            public RootEntityHandler filterRootEntities(
                    CimType arg0, RootEntityHandler arg1) {
                return new MatchRootEntityByNativeEmsNameInsteadOfName();
            }
        };
    }
    ```

2. Define a private class that extends from **DefaultRootEntityHandler**. This class is the one created in step 1. Override the **getReferenceRootEntity()** method as follows. Notice the use of a string array containing the string **nativeEmsName** to specify the use of this attribute. Also notice the use of a **RuntimeException** to report problems.

    ```
    private class MatchRootEntityByNativeEmsNameInsteadOfName
            extends DefaultRootEntityHandler {
        @Override
        public DiscrepancyEnabled getReferenceRootEntity(DiscrepancyEnabled
    compareRoot) {
            try {
                PomsManagerFactory factory = new PomsManagerFactory();
    ```

```
                    DisResultGroupManager DisResultGroupManager =
                        factory.getDisResultGroupManager();
                    DisResultGroup g = DisResultGroupManager.getDisResultGroup(
                        (Persistent) compareRoot);
                    return new ReferenceRootFinder(g).
                            findReferenceRoot((Persistent) compareRoot,
                                            new String[] { "nativeEmsName" });
            } catch (Exception e) {
                logger.log(Level.SEVERE,
                        "Error while getting reference root, compareRoot " +
                        compareRoot, e);
                throw new RuntimeException(
                    "Error while getting reference root, Aborting discrepancy
generation",
                        e);
            }
        }
    }
```

3. In the **invoke** method of the processor, create the CIMType object to cover all entity types, and add the root entity filter defined in step 1.

```
CimType allType = new CimType(DiscrepancyEnabled.class);
context.getRootEntityLoader().addFilter(allType, getRootEntityFilter());
```

Part two adds a relationship filter to each entity type that the detection processor expects to encounter. This code fragment example shows a change to a single entity type. It changes the LogicalDevice to DeviceInterface child relationship to match using **nativeEmsName** instead of **name**. Normally, this code pattern needs to be repeated once for each entity type. (See the MIB II UIM and Cisco UIM cartridge packs for a full example.)

1. Define the relationship handler as a class inside the processor's class. This class should inherit from **DefaultRelationshipHandler**, and override the **getKey()** method to return

```
public class MatchDevIntfByNativeEmsName extends DefaultRelationshipHandler {
    @Override
    protected Object getKey(DiscrepancyEnabled entity) {
        return ((DeviceInterface) entity).getNativeEmsName();
    }
}
```

2. In the processor **invoke** method, get the generic discrepancy detector from the context.

```
GenericDiscrepancyDetector detector = context.getDiscrepancyDetector();
```

3. In the same **invoke** method, create the CIMType object to name the entity type, and add the custom filter.

```
CimType ldType = new CimType(LogicalDevice.class);
detector.addFilter(ldType, new RelationshipFilter() {
        @Override
        public void filterRelationships(
                CimType cimType,
                Map<String, RelationshipHandler> relationshipHandlers) {
            relationshipHandlers.put("deviceInterface",
                                    new MatchDevIntfByNativeEmsName());
        } // end filterRelationships
    } // end new RelationshipFilter
```

```
); // end addFilter
```

Adding Post-Processors

This usage pattern builds on the Reuse pattern and adds processors after the discrepancy detector processor. These **post-processors** access the full set of detected discrepancies using the **getDiscrepancies()** method of the **DiscrepancyDetectionProcessorContext** object (context). Because they are not persisted until all processors in the action have executed, the discrepancies can be manipulated completely by the post-processors. They can be modified or removed. Also, new discrepancies can be added.

Although all fields of a discrepancy can be modified by using setters, there are many fields that should not be altered. The following discrepancy fields can be safely changed by post-processors:

- **priority**, **notes**, **discrepancyOwner**

- **severity**, **compareValue**, **referenceValue**

- **operation** + **operationIdentifiedBy** + **status** (status set to **OPERATION_ IDENTIFIED**) (Must be set together.)

Any other discrepancy fields should not be altered; otherwise, discrepancy resolution actions may suffer errors and failures.

An example of the use of post-processors is to automatically assign all **CRITICAL** severity discrepancies to a specific department (using the **discrepancyOwner** field). The following code snippet from a post-processor shows how this is done.

```
@Override
public void invoke(DiscrepancyDetectionProcessorContext context,
        DiscrepancyPostProcessorProcessorRequest request)
        throws ProcessorException {

    for (DisDiscrepancy discrepancy : context.getDiscrepancies()) {
        if (discrepancy.getSeverity().equals(
                DisDiscrepancySeverity.CRITICAL)) {
            discrepancy.setDiscrepancyOwner("Sherlock Holmes");
        }
    }
}
```

About Discrepancy Resolution Processors

The only type of processor available to the discrepancy resolution action is the discrepancy resolution processor.

As with other types of actions, the list of processors are invoked serially from top of the table to bottom. The first processor is given the list of submitted discrepancies. This processor determines a subset of these discrepancies to handle (which can range from none to all), perform the resolution operation, and set their status to either *Processed* or *Failed*.

Then, the next processor is given the remaining discrepancies for processing, and so on. The action is complete when all the processors are invoked. If there are any discrepancies which remain at the end, their status is set to *Not Implemented*.

The discrepancy resolution processor is the Java implementation of a discrepancy resolution action. The processor performs the following tasks:

- Filter through its input list of discrepancies to process only those discrepancies it can handle
- Communicate with the discovery or import system to correct a discrepancy
- Report the status of a correction operation

See "Implementing a Processor" for more information.

Creating a Discrepancy Resolution Processor

See the Design Studio Modeling Network Integrity Help for information about creating a discrepancy resolution processor.

Implementing a Discrepancy Resolution Processor

This section provides details about the discrepancy resolution processor implementation.

About the Implementation Interface

The processor implementation class derives from a Design Studio-generated interface class. There is a single abstract method that the implementation class must implement. The abstract method has the following interface:

```
public <Processor_Name>Response invoke(
            DiscoveryResolutionProcessorContext context,
            <Processor_Name>Request request)
        throws ProcessorException
{
}
```

About Input Parameters for the Invoke Method

Table 3–1 describes the methods provided to the developer by the first parameter, *context*, outlined in "About the Implementation Interface".

Table 3–1   Methods from the context Parameter

Context method	Return Object Class	Description
getActionName()	String	Getter for the name of the action.
getProcessorName()	String	Getter for the name of this processor.
getUnhandledDiscrepancies()	Collection *DisDiscrepancy*	Getter for a list of unprocessed discrepancies for this invocation.
getAllDiscrepancies()	Collection *DisDiscrepancy*	Getter for a list of processed and unprocessed discrepancies for this invocation.
discrepancyProcessed(DisDiscrepancy disc)	void	Sets the status of the input discrepancy to OPERATION_PROCESSED.
discrepancyFailed(DisDiscrepancy disc, String failureMessage)	void	Sets the status of the input discrepancy to OPERATION_FAILED, and also sets the failure message.
discrepancyReceived(DisDiscrepancy disc)	void	Sets the status of the input discrepancy to OPERATION_RECEIVED.

The second parameter, *request*, contains getters for each item in the Input Parameters table. It also contains a getter to retrieve the groups and items listed in the Properties tabbed page.

Return Type of Invoke Method

The return type of the **invoke** method varies, depending on the output parameters setting in the Context Parameters tabbed page.

If there is no output parameter, then the return type is void.

If there are one or more output parameters, then the return type is a generated class with the name *Processor_Name*Response. This Response class has getters and setters for each item in the Output Parameters table.

About the General Flow of the Discrepancy Resolution Processor

The usual pattern for implementing a discrepancy resolution processor is as follows:

1. Fetch the list of unhandled discrepancies using **context.getUnhandledDiscrepancies()**

2. Allocate discrepancies based on logical groupings; for example: all discrepancies on a single card and on its children port.

   Keep discrepancies that can be handled by this processor, and ignore or remove other discrepancies.

3. For each group, perform operations to fix the discrepancies, Then, based on operation results, set their status to *Processed* or *Failed*.

   An error message can be saved in the Failure Reason field of the discrepancy, which is displayed in the Network Integrity UI.

4. Set output parameters.

Fetching Discrepancies

The discrepancy resolution processor can use the context input parameter to fetch the list of discrepancies to process. In the general flow, the processor uses the method **getUnhandledDiscrepancies()** on *context* to retrieve a list of discrepancies that are not yet handled by any previous processors.

It is also possible to retrieve the original full list of discrepancies by using the method **getAllDiscrepancies()**, but this list includes discrepancies that are already handled by a prior resolution processor.

It is possible to make updates to already handled discrepancies, such as updating the **Notes** field to add more text.

See "About Discrepancies" for more information about the attributes of a Discrepancy object.

Grouping Discrepancies

Usually, a single resolution processor is responsible for handling the discrepancies of a single entity type; for example: logical device or device interface only, or more frequently an explicit set of specifications of an entity type.

Sometimes, a processor specializes in handling discrepancies of a very specific nature. Therefore, the next logical task is to examine each unhandled discrepancy, to determine how it should be handled by this processor.

A processor frequently uses one or more of the following discrepancy attributes as criteria for handling. Of course, it may use all other attributes as criteria for determining special handling, if necessary.

See "About Discrepancies" for a detailed explanation of these attributes:

- Type: Indicates the error being reported; for example: attribute mismatch, missing entity, and so on.

- externalEntityType, staticEntityType: Indicates the type and specification of the target entity.

- attributeOrRelationshipName: Indicates the attribute or the association that has the discrepancy.

- compareValue, referenceValue: Each attribute indicates the value of an attribute on one side of the comparison.

- compareEntity, referenceEntity: Each attribute is a reference to one entity being compared; see "About the Compare and Reference Sides" and "About Discrepancy Types" for important information on what entity each attribute is actually referencing.

- childTargetEntity: This is an additional entity reference used only for Association or Entity discrepancy types; see "About Discrepancy Types" for more information.

Handling Discrepancies

Now that the target has been identified and grouped, the processor can decide whether to proceed with the handling. If the processor can resolve this discrepancy, then the processor can make appropriate API calls necessary to make the desired resolution on the system, and report the result.

See "Reporting the Resolution Result".

Alternatively, the processor can decide to skip the discrepancy, and begin processing the next one. The skipped discrepancy subsequently appears in the unhandled list of discrepancies for the next processor.

Reporting the Resolution Result

When a discrepancy has resolved successfully, simply pass this discrepancy into the *context* using the method **discrepancyProcessed**. This sets the discrepancy status to *Processed*.

```
context.discrepancyProcessed( discrepancy );
```

If the processor fails to resolve a discrepancy, it should set the discrepancy status to *Failed* using the method **discrepancyFailed** in the *context*.

This method takes an additional String argument, which the processor can set a short message to be displayed in the UI. The string is stored in the reasonForFailure attribute of the discrepancy.

> **Note:** This error message is limited to a maximum of 255 characters.

```
context.discrepancyFailed( discrepancy, "Sample error message.");
```

If the processor needs to make a series of asynchronous invocations to handle a discrepancy, it can set the discrepancy status to *Received* at the end of the first invocation.

This indicates to Network Integrity and to Network Integrity users that the discrepancy resolution is in progress. This is done using the method **discrepancyReceived** in the *context*.

```
context.discrepancyReceived( discrepancy );
```

See "About Discrepancy Status" for an explanation of the transition rules for status values.

Handling Discrepancies Asynchronously

There are situations in which a discrepancy resolution operation cannot be completed within a single invocation. For example, the CORBA interface for an external system to create a trouble ticket requires the caller to supply a callback object for the notification of the final operation result and ticket ID.

In this example, the resolution processor code can prepare the callback object and make the initial CORBA call to submit the trouble ticket, and then it must return from the **invoke** method. The subsequent resolution handling code must reside in the callback object, and receives the notification, updating the discrepancy status accordingly.

In such cases, the processor should set the status of the discrepancy to *RECEIVED* using **context.discrepancyReceived()** at the end of the handling code in the processor's **invoke** method. This indicates to Network Integrity and to Network Integrity users that resolution processing is in progress, and that additional status updates arrive later.

You must also save the entityID of the discrepancy (using **discrepancy.getEntityId()**) during the processor's **invoke** method. When the subsequent resolution handing operation reaches its conclusion, the status of the original discrepancy must be updated to *PROCESSED* or *FAILED*. This is done through the Network Integrity web service by first retrieving the discrepancy using the entityID, and then updating the status of the discrepancy.

The topic of how to save the entityID and how to create the subsequent code invocation is beyond the scope of this guide. You may use any techniques available in J2EE to perform these tasks.

4

Working with Discrepancies

This chapter provides an overview of discrepancies in Oracle Communications Network Integrity.

About Discrepancies

When Network Integrity detects a difference while comparing import and discovery data, it generates a discrepancy. The discrepancy captures all vital information about the difference, such as the entity and the name of the attribute or relationship containing the difference, the type of difference, and the values on both sides (that is to say, on the *Compare*, and the *Reference* sides).

These topics are further explored in:

- About the Compare and Reference Sides
- About Discrepancy Types
- About Discrepancy Status
- About Discrepancy Detail

About the Compare and Reference Sides

When dealing with discrepancies, the data from the two sides are named *Compare* and *Reference*. The significance is that the *Compare* side is the side of the scan that triggered the discrepancy comparison.

If a scan using a discovery action was also configured to detect discrepancies, the discrepancies created by that scan have discovery data on the *Compared* side, and import data on the *Reference* side.

On the other hand, if a scan uses an import action with detect discrepancies configured, the *Compared* fields of a discrepancy contain import data, and the *Reference* fields contain discovery data.

The discrepancy field *CompareSource* holds a value that indicates the origin of the compare-side data. The value is NETWORK for a discovery or an Assimilation scan, or INVENTORY for an import scan.

Table 4–1 shows CompareSource values for different discrepancy origins.

Table 4–1 Listing CompareSource Values for Different Discrepancy Origins

Discrepancy Origin	Compared Side	CompareSource	Reference Side	ReferenceSource
Discovery Scan	Discovery Data	NETWORK	Import Data	INVENTORY

Table 4–1 (Cont.) Listing CompareSource Values for Different Discrepancy Origins

Discrepancy Origin	Compared Side	CompareSource	Reference Side	ReferenceSource
Import Scan	Import Data	INVENTORY	Discovery Data	NETWORK
Assimilation Scan	Discovery Data	NETWORK	Import Data	INVENTORY

About Discrepancy Types

There are seven types of discrepancy; they can be divided into four groups of related issues.

- Attribute Value Mismatch. See "Attribute Value Mismatch".

- Extra Entity, Missing Entity. See "Extra Entity and Missing Entity".

- Extra Association, Missing Association. See "Extra Association and Missing Association".

- Ordering Error, Association Ordering Error. See "Ordering Error and Association Ordering Error".

Network Integrity does not allow new discrepancy types to be defined.

Attribute Value Mismatch

This discrepancy indicates that an entity exists in both the *Compare* and *Reference* results, but an attribute was found not to have the same value on both sides.

Each discrepancy reports a mismatch problem on a single attribute. An entity can have multiple Attribute Value Mismatch discrepancies reported, if it has several mismatched attributes on both sides.

Table 4–2 shows discrepancy attributes and descriptions.

Table 4–2 Attribute Value Mismatch: List of Discrepancy Attributes

DisDiscrepancy Attribute	Description
compareEntity	This is the target entity whose attribute has a mismatched value.
referenceEntity	This is the matching entity on the other side of the discrepancy detection.
childTargetEntity	Not used. This has no value.
attributeOrRelationshipName	This holds the name of the attribute containing the mismatch.
compareValue	The value of the attribute on the target entity.
referenceValue	The value of the attribute on the matching entity on the other side.

Extra Entity and Missing Entity

This discrepancy indicates that an entity (and any dependent children) is present on one side of the comparison, but is absent from the other side.

An Extra Entity discrepancy indicates that the entity is present in the *Compared* side, but not in the *Reference* side.

In Figure 4–1, the example for the Extra Entity discrepancy shows an FDDI card in slot 7 present on the Compared side that is missing on the Reference side.

A Missing Entity discrepancy indicates the reverse: the entity is absent is the *Compared* side, but present in the *Reference* side.

In Figure 4–1, the example for the Missing Entity discrepancy shows that slot 7 is missing an FDDI card on the Compared side that is present on the Reference side.

Figure 4–1   Examples of Extra Entity and Missing Entity
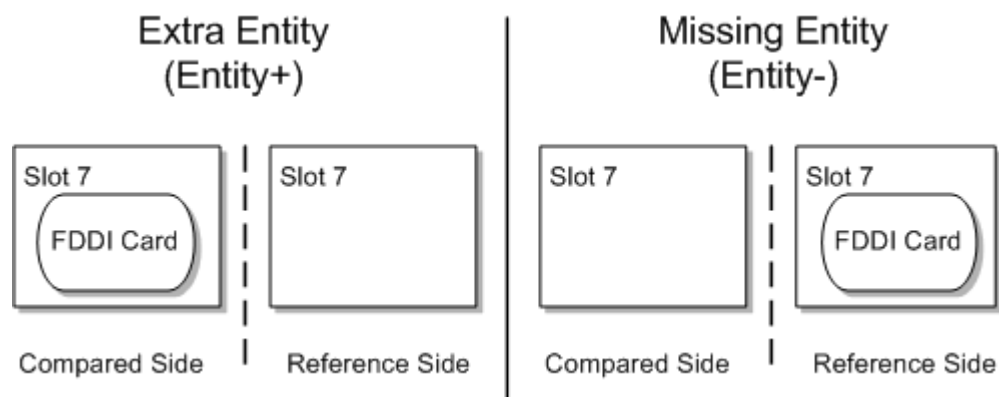
Table 4–3 shows discrepancy attributes and descriptions.

Table 4–3   Discrepancy Attributes and Descriptions

DisDiscrepancy Attribute	Description
compareEntity	This is the parent entity on one side of the comparison.
referenceEntity	This is the parent entity on the other side of the comparison.
childTargetEntity	This is the extra child entity on one side. The entity exists on the *Compared* entity tree when the discrepancy type is Extra Entity. The entity exists on the *Reference* entity tree when the discrepancy type is Missing Entity.
attributeOrRelationshipName	This holds the name of the association on the parent entity, which references the childTargetEntity.
compareValue	Not used. This has no value.
referenceValue	Not used. This has no value.

When resolving an Extra/Missing Entity discrepancy, the processor is tasked with either adding or removing an object from its target system. The processor must consider the system that it is managing (Import/Inventory or Discovery/Network), and examine the following discrepancy fields to determine the appropriate action:

- DiscrepancyType
- CompareSource

For example: A discrepancy resolution processor is created to make corrections to an inventory system. When this processor receives an Extra Entity discrepancy, it must check the value of CompareSource. If this value is *NETWORK*, the extra entity occurs in the network, and therefore it must be missing from the inventory system. The processor takes the corrective action of creating this entity in the inventory system.

However, if the discrepancy type is still Extra Entity, and CompareSource value is *INVENTORY*, the extra entity occurs in inventory.

Table 4–4 shows the resolution operations for the example processor, given the actual factors to be considered. The *Present in* columns indicate the system has the extra

entity. The *Resolution Operation* column lists the appropriate inventory operation to resolve this discrepancy.

Table 4–4   Appropriate Resolution Operations for Sample Processor

Discrepancy Type	Compare Source	Referece Source	Present in Network	Present in Inventory	Resolution Operation
Extra Entity	Network	Inventory	Yes	No	Add the network entity into Inventory.
Missing Entity	Network	Inventory	No	Yes	Remove the inventory entity.

> **Note:** Table 4–4 assumes that the discrepancy detection action was triggered from a Discovery scan.

If the discrepancies are generated by a discrepancy detection action that listens for results from Import scans, the compare source and reference source are reversed, and subsequently, the appropriate inventory operations are reversed as well. (This situation is not usual, but is certainly possible.) See Table 4–5 for this example.

Table 4–5   Appropriate Resolution Operations for Sample Processor (Import Scan)

Discrepancy Type	Compare Source	Referece Source	Present in Network	Present in Inventory	Resolution Operation
Extra Entity	Inventory	Network	No	Yes	Remove the inventory entity.
Missing Entity	Inventory	Network	Yes	No	Add the network entity into Inventory.

Network Integrity does not report Missing Entity discrepancies on the circuit of a root entity when the root entity is absent from either the *Compared* side or the *Reference* side.

For example, if a discovery scan finds Device1 with circuits A and B in the network, and the same device exists in inventory, but with circuits A, B, and C, Network Integrity reports a Missing Entity discrepancy on circuit C in the network.

In the above example, Network Integrity can fully compare the results for Device1 from the *Compared* side and the *Reference* side.

However, by default, when Device1 is not listed in the discovery results, Network Integrity does not report Missing Entity discrepancies on the device.

You can build a discrepancy detection action or extend the base discrepancy detection action to report missing Entity discrepancies on root entities. See "About Discrepancy Detection Actions" for more information.

Extra Association and Missing Association

This discrepancy indicates that an association in one entity (source) referencing another entity (target) is present on one side of the comparison, but is absent from the other side.

An Extra Association discrepancy indicates that the association is present in the *Compared* side, but not in the *Reference* side.

In Figure 4–2, the example for the Extra Association discrepancy shows a Mapped Device Interface association from Port 1 to Interface 2 present on the Compared side that is missing on the Reference side.

A Missing Association discrepancy indicates the reverse: the association is absent in the *Compared* side, but is present in the *Reference* side.

In Figure 4–2, the example for the Missing Association discrepancy shows that the Mapped Device Interface association from Port 1 to Interface 2 is missing on the Compared side but is present on the Reference side.

Each discrepancy indicates a problem with a single direction of association. If two entities have a bidirectional association with each other, and this bidirectional association is completely missing on one side, two discrepancies are generated by Network Integrity.

Figure 4–2   Examples of Extra Association and Missing Association
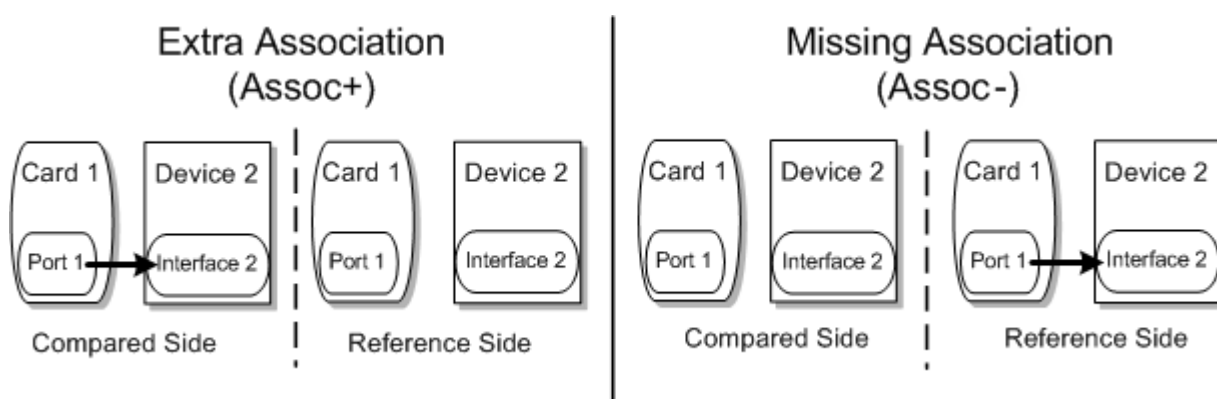
Table 4–6 shows discrepancy attributes and descriptions.

Table 4–6   Extra Association and Missing Association: List of Discrepancy Attributes

DisDiscrepancy Attribute	Description
compareEntity	This is the source entity on one side of the comparison.
referenceEntity	This is the source entity on the other side of the comparison.
childTargetEntity	This is the target entity of the association. The entity exists on the Compared side when the discrepancy type is Extra Association. It exists on the Reference side when the discrepancy type is Missing Entity.
attributeOrRelationshipName	This holds the name of the association on the source entity which references the childTargetEntity.
compareValue	Not used. This has no value.
referenceValue	Not used. This has no value.

The processor must examine the discrepancy to determine whether the appropriate resolution operation is to add the association, or to remove it.

Table 4–7 shows the appropriate operation, given the values of discrepancy type, compare source, and reference source within the discrepancy.

Table 4–7   Appropriate Resolution Operations for Sample Processor

Discrepancy Type	Compare Source	Reference Source	Present in Network	Present in Inventory	Resolution Operation
Extra Association	Network	Inventory	Yes	No	Add the association into the inventory entity.
Missing Association	Network	Inventory	No	Yes	Remove the association from the inventory entity.

If the discrepancies are generated by a discrepancy detection action that listens for results from Import scans, the compare source and reference source are reversed, and subsequently, the appropriate inventory operation are reversed as well. (This situation is not usual, but is certainly possible.)

Table 4–8 shows the appropriate operation for this particular situation.

Table 4–8   Appropriate Resolution Operations for Sample Processor (Import Scan)

Discrepancy Type	Compare Source	Reference Source	Present in Network	Present in Inventory	Resolution Operation
Extra Association	Inventory	Network	No	Yes	Remove the association from the inventory entity.
Missing Association	Inventory	Network	Yes	No	Add the association into the inventory entity.

Ordering Error and Association Ordering Error

In some cases, the ordering of child or associated entities is significant. This discrepancy indicates that matched entities appear in different orders between the two sides. The only difference between the two types of discrepancy is that an Ordering Error indicates a problem with a parent/child association, while an Association Ordering Error indicates a problem with some other association.

Table 4–9 shows discrepancy attributes and descriptions.

Table 4–9   Ordering Error and Association Ordering Error: List of Discrepancy Attributes

DisDiscrepancy Attribute	Description
compareEntity	This is the source/parent entity on one side of the comparison.
referenceEntity	This is the source/parent entity on the other side of the comparison.
childTargetEntity	Not used. This has no value.
attributeOrRelationshipName	This holds the name of the association having the ordering problem.
compareValue	Not used. This has no value.
referenceValue	Not used. This has no value.

About Discrepancy Status

The discrepancy status field identifies the state of a discrepancy within its life cycle. Table 4–10 lists the possible discrepancy statuses.

Table 4–10   Discrepancy Statuses

Status	Status Change Trigger	Valid Follow-On Statuses
Opened	NA	Ignored, Identified
Ignored	UI command	Opened, Identified
Identified	UI command	Submitted, Ignored, Opened
Submitted	Programmatic operation	Received, Failed, Processed, Not Implemented
Received	Programmatic operation	Failed, Processed
Failed	UI command	Ignored, Identified
Not Implemented	NA	NA
Processed	NA	NA

Figure 4–3 shows the discrepancy status lifecycle diagram.

Figure 4–3   Discrepancy Status Life Cycle
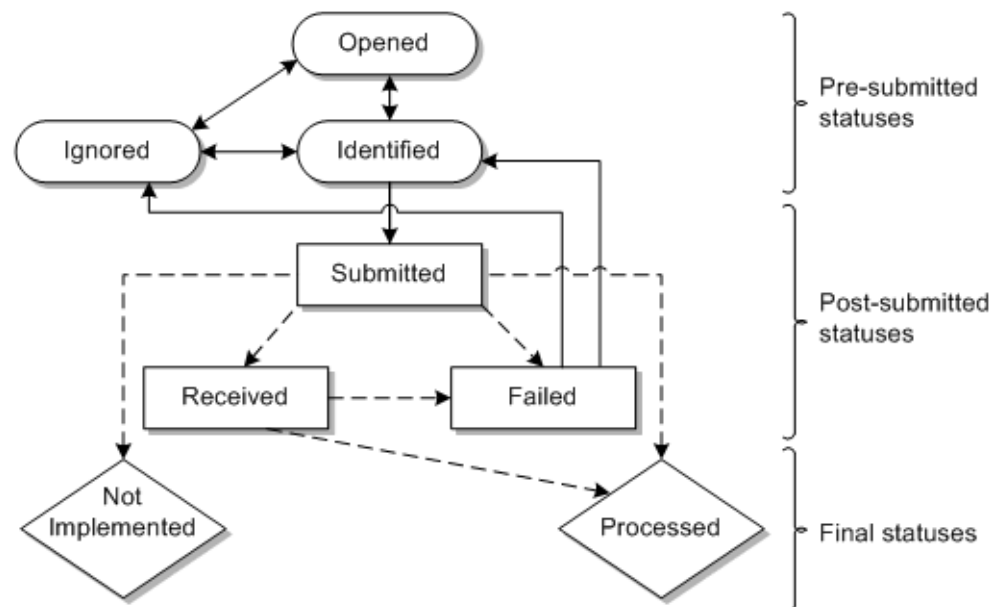
Every discrepancy begins with a status of *OPENED* when it is first detected. It can then be moved to one of two states by a user using a web UI operation:

- *IDENTIFIED*, by using a resolution action menu item

- *IGNORED*, by using the **Ignore** menu item

When a discrepancy is in the *IDENTIFIED* state, a user can use the **Submit** operation to move it to the *SUBMITTED* state. At this point, the discrepancy has moved out of user control, and into the control of a resolution action.

The resolution action processes the submitted discrepancy, and reports the outcome by setting the status to:

- *PROCESSED*, or

- *FAILED*

If the status is *PROCESSED*, the operation has succeeded, and the discrepancy can no longer be acted upon. If the status is *FAILED*, it becomes available for the user to specify an operation again, just like when it was first opened.

A resolution action may set a discrepancy status to *RECEIVED* immediately after the submit operation. This status indicates that the resolution operation is in progress, and reports its final operation status later.

About Discrepancy Detail

Table 4–11 lists all the attributes of a discrepancy. The Java type of a discrepancy is **DisDiscrepancy**. Use Java getter and setter patterns to retrieve and set the attribute's value. For example use the **getPriority()** method to get the value of priority, and **setPriority(***String***)** method to change its value.

Although the setters for all attributes are public, most fields should not be directly set by the processors. The following fields are safe to be used by processor Java implementations:

- *priority*

- *notes*

- *discrepancyOwner*

The *status* and *failureReason* fields should be set using the context methods when inside a processor **invoke** method. Otherwise, they can also be set using setters.

Table 4–11   Discrepancy Attributes

DisDiscrepancy Attribute	Type	Description
type	DisDiscrepancyType (Enum)	The discrepancy type. Valid values are: ■ ATTRIBUTE_VALUE_MISMATCH ■ EXTRA_ENTITY ■ MISSING_ENTITY ■ EXTRA_ASSOCIATION ■ MISSING_ASSOCIATION ■ ORDERING_ERROR ■ ASSOCIATION_ORDERING_ERROR
severity	DisDiscrepancySeverity (Enum)	The severity of the discrepancy. The values are (from most severe to least): ■ CRITICAL ■ MAJOR ■ MINOR ■ WARNING
entityName	String	The name of the entity for which this discrepancy is raised.
externalEntityType	String	The name of the specification, if the entity has a specification. Otherwise, the same value as staticEntityType.
staticEntityType	String	The name of the base entity type of the entity.

Table 4–11   (Cont.) Discrepancy Attributes

DisDiscrepancy Attribute	Type	Description
attributeOrRelationshipName	String	This holds the name of the attribute or relationship having the discrepancy.
compareEntity	long (Weak Reference)	This is the entityID of the entity for which this discrepancy is raised.
compareSystem	DisSource (Enum)	Indicates whether the compare data comes from Network (Discovery) or Inventory (Import) system. Valid values are NETWORK and INVENTORY.
compareValue	String	This is used by attribute value mismatch discrepancies to hold the value of the attribute on the compare side.
compareSource	String	The source value of the compareEntity. This value is copied from the Source field of the Scan configuration used to discover/import this entity into Network Integrity.
referenceEntity	long (Weak Reference)	This is the entityID of the entity of the discrepancy on the opposite side to the compareEntity.
referenceSystem	DisSource (Enum)	This indicates whether the reference data comes from Network (Discovery) or Inventory (Import) system. Valid values are NETWORK and INVENTORY.
referenceValue	String	This is used by attribute value mismatch discrepancies to hold the value of the attribute on the reference side.
referenceSource	String	This is the source value of the referenceEntity. This value is copied from the Source field of the Scan configuration used to discover/import this entity into Network Integrity.
childTargetEntity	long (Weak Reference)	Used by Extra/Missing discrepancies to indicate the child/target entityID of the entity of an association.
ancestorEntityName	String	This is the name of the ancestor (parent) entity for the discrepancy.
ancestorEntityType	String	This is the name of the specification, if the ancestor entity has a specification. Otherwise, it takes the same value as ancestorStaticEntityType.
ancestorStaticEntityType	String	This is the name of the base entity type of the ancestor entity.
parentResultGroup	DisResultGroup	This is a reference of the parent scan result detail (that is, the result group) of the compareEntity.
path	String	This is the path to the entity for this discrepancy. It is a comma-delimited list of entity IDs that describes the path from the root entity. For Missing Entity and Missing Association discrepancies, it is the path to the compareEntity followed by the entityID of the referenceEntity. For other discrepancy types, it is the path to the compareEntity.
priority	String	This is a user-editable field used to indicate the priority of this discrepancy. This would typically be used for customer-specific categorization, enabling a finer control than using severity alone.
notes	String	This is a user-editable field for comments.

Table 4–11   (Cont.) Discrepancy Attributes

DisDiscrepancy Attribute	Type	Description
discrepancyOwner	String	This is a user-editable field used to indicate an external owner of the discrepancy. It may be used for other purposes if desired.
operation	String	This holds the name of the resolution action being invoked.
operationIdentifiedBy	String	This is the ID of the user who identified the resolution action (the UI action to set the resolution operation, before the submit operation).
operationSubmittedBy	String	This is the ID of the user who submitted the resolution action.
submittedTime	Date	This is the timestamp when the status changed to OPERATION_SUBMITTED.
status	DisDiscrepancyStatus (Enum)	This is the current status of this discrepancy. Valid values are: ■ DISCREPANCY_OPENED ■ DISCREPANCY_IGNORED ■ OPERATION_IDENTIFIED ■ OPERATION_SUBMITTED ■ OPERATION_RECEIVED ■ OPERATION_NOT_IMPLEMENTED ■ OPERATION_PROCESSED ■ OPERATION_FAILED
lastStatusChangeTime	Date	This is the timestamp when the status attribute was last updated.
reasonForFailure	String	This holds the error message set by the processor using **context.discrepancyFailed()** method.
entityID	long	This is an Internal identifier.

5

Working with the POMS SDK

This chapter provides information about how the persistent object modeling service (POMS) manages persistent data in Oracle Communications Network Integrity.

This chapter contains the following sections:

- About POMS
- Working with POMS Entities
- Working with POMS Relationships
- Working with Specifications and Characteristics
- Working with the POMS Finder
- About Persist Results

About POMS

POMS manages all persisted data for Network Integrity. You use POMS for most cartridge development, but you rarely need to deal explicitly with persistence details.

POMS includes the Java definition of the entities and relationships described in *Oracle Communications Information Model Reference*.

While POMS includes both interface and implementation classes for the entities, you work only with interfaces. These interfaces provide getters and setters for attributes and relationships. Use the PersistenceHelper POMS SDK class to instantiate entities.

You can use the POMS SDK Finder class to find and retrieve existing persisted entities.

POMS is built on the EclipseLink Java persistence API (JPA) platform. You do not usually need to know EclipseLink or JPA to use the POMS SDK. The exception is **find** operations where you may have to know Java Persistence Query Language (JPQL). See "Working with the POMS Finder" for more information about the **find** operations.

Table 5–1 describes the POMS SDK APIs.

Table 5–1   POMS SDK API Description

POMS SDK APIs	Description
Entities	The POMS SDK represents modelled entities as Java interfaces with getters and setters for attributes and relationships. See "Working with POMS Relationships".
Specifications and characteristics	The POMS SDK includes APIs that allow you to operate on specifications and characteristics. See "Working with Specifications and Characteristics".

Table 5–1 (Cont.) POMS SDK API Description

POMS SDK APIs	Description
PersistenceHelper	The POMS SDK provides methods to instantiate a POMS entity or POMS Finder. See "Working with POMS Entities" and "POMS SDK Interfaces".
Finder	The POMS SDK provides various methods to define a query and retrieve matching persisted entities. See "Working with the POMS Finder" and "POMS SDK Interfaces".

Working with POMS Entities

The POMS Java interface for an entity has the same name as the entity described in the model document. For example, entity **Equipment** becomes:

```
public interface Equipment
```

Attributes are accessed with familiar Java getters and setters. For example. The Equipment name attribute is defined by:

```
public java.lang.String getName();
public void setName( java.lang.String name );
```

An entity may contain enumerated values for certain attributes. POMS implements these as Java enumerations. For example, the **EMSServiceState** from **LogicalDevice** has the following:

```
public enum EMSServiceState {
     UNKNOWN( "UNKNOWN" ),
     IN_SERVICE( "IN_SERVICE" ),
     OUT_OF_SERVICE( "OUT_OF_SERVICE" ),
     TESTING( "TESTING" ),
     IN_MAINTENANCE( "IN_MAINTENANCE" );

public oracle.communications.inventory.api.entity.EMSServiceState
getNativeEmsServiceState();
public void setNativeEmsServiceState(
oracle.communications.inventory.api.entity.EMSServiceState nativeEmsServiceState
);
```

When creating results, for example in a discovery processor, you must instantiate POMS entities. Use the **PersistenceHelper** class, passing the desired entity class to the **makeEntity** method:

```
Equipment equipment = PersistenceHelper.makeEntity(Equipment.class);
```

Working with POMS Relationships

Related entities are also accessed with getters and setters.

One-to-one Relationships

When a relationship refers to a single entity, the entity is accessed directly. For example, the mapped physical and logical devices:

```
public oracle.communications.inventory.api.entity.LogicalDevice
getMappedLogicalDevice();
public void setMappedLogicalDevice(
oracle.communications.inventory.api.entity.LogicalDevice mappedLogicalDevice );
```

One-to-Many or Many-to-Many Relationships

When a relationship refers to multiple entities, the entities are accessed through a collection. For example, the equipment to physical port relationship:

```
public java.util.List<oracle.communications.inventory.api.entity.PhysicalPort>
getPhysicalPorts();
public void setPhysicalPorts(
java.util.List<oracle.communications.inventory.api.entity.PhysicalPort>
physicalPorts );
```

A getter never returns **null** for the collection. If there are no related entities, an empty collection is returned. That means the developer can safely add entities without creating a collection. For example:

```
equipment.getPhysicalPorts().add(physicalPort);
```

Ordered and Unordered Relationships

POMS uses a **List** for the collection because the Oracle Communications Information Model defines an ordered relationship for physical ports on equipment. In other cases, order does not matter and so POMS uses a **Set** for the collection. For example, the parent relationship from **Equipment** to **EquipmentHolder**:

```
public
java.util.Set<oracle.communications.inventory.api.entity.EquipmentHolderEquipmentR
el> getParentEquipmentHolders();
public void
setParentEquipmentHolders(java.util.Set<oracle.communications.inventory.api.entity
.EquipmentHolderEquipmentRel> equipmentHolders);
```

Bi-directional Relationships

Certain relationships in the model are bi-directional. POMS includes accessors on entities on both sides of a bi-directional relationship, and the relationship can be set from either side. The physical device to logical device relationship described in the "One-to-one Relationships" example is bi-directional. The other side of this relationship, on the logical device, is defined as:

```
public java.util.List<oracle.communications.inventory.api.entity.PhysicalDevice>
getMappedPhysicalDevices();
public void setMappedPhysicalDevices(
java.util.List<oracle.communications.inventory.api.entity.PhysicalDevice>
mappedPhysicalDevices );
```

This is a many-to-one relationship, so there is a collection on the logical device side and single entity on the physical device side. To relate a physical and logical device, you can either set from the physical device:

```
physicalDevice.setMappedLogicalDevice(logicalDevice);
```

or set from the logical device:

```
logicalDevice.getMappedPhysicalDevices ().add(physicalDevice);
```

Relationship Entities

In some cases, the model defines an intermediate relationship entity instead of relating two entities directly. For example, the Information Model defines **EquipmentEquipmentRel** to relate two pieces of equipment. To create this type of

relationship, instantiate the relationship entity and set the related entities. For the equipment to equipment example:

```
EquipmentEquipmentRel parentEquipmentRel =
PersistenceHelper.makeEntity(EquipmentEquipmentRel.class);
parentEquipmentRel.setChildEquipment(equipment);
parentEquipmentRel.setParentEquipment(parentEquipment);
```

Working with Specifications and Characteristics

You can use the generated specification helper classes to avoid directly dealing with specifications and characteristics. See "About Specifications" and "Working with Specifications" for a description of the underlying API and for more information on when to directly manipulate specifications.

You can determine if an entity supports characteristics and specification by referencing the model documentation, or by checking the POMS interface. Entities that support characteristics and specifications extend the CharacteristicExtensible interface. For example:

```
oracle.communications.inventory.api.CharacteristicExtensible
<oracle.communications.inventory.api.entity.EquipmentCharacteristic>;
```

The specification and characteristics are related entities like any other, characteristics being multi-valued:

```
public oracle.communications.inventory.api.entity.EquipmentSpecification
getSpecification();
public void setSpecification(
oracle.communications.inventory.api.entity.EquipmentSpecification specification );

public
java.util.Set<oracle.communications.inventory.api.entity.EquipmentCharacteristic>
getCharacteristics();
public void setCharacteristics(
java.util.Set<oracle.communications.inventory.api.entity.EquipmentCharacteristic>
characteristics );
```

As a convenience, POMS also lets you access a characteristic by name through the map returned by **getCharacteristicMap**:

```
public java.util.Map<String,
oracle.communications.inventory.api.entity.EquipmentCharacteristic>
getCharacteristicMap();
```

Working with the POMS Finder

You can use the POMS Finder to retrieve previously persisted data, however, you do not typically need to use the Finder.

The most basic use of the Finders is "Find by Entity". More powerful and flexible queries are possible with the Java Persistence Query Language (JPQL). You can also control whether entities are returned completely or a with a subset of attributes. You can also use paging to return data in manageable chunks where queries might return a large volume of data.

Find by Entity

To find entities matching an example entity, instantiate an entity of the appropriate type and set one or more attributes. Use the **findByEntity** method to return a collection of matching entities. Here is an example that looks for the specification for a Cisco 3640 physical device:

```
Finder finder = PersistenceHelper.makeFinder();
PhysicalDeviceSpecification example =
    PersistenceHelper.makeEntity(PhysicalDeviceSpecification.class);
example.setName("Cisco3640");

Collection<PhysicalDeviceSpecification> specifications =
    finder.findByEntity(example, "name");
if (specifications.size() == 1) {
    System.out.println("found specification");
}
```

Find by JPQL

Java Persistence Query Language (JPQL) is a powerful way to express queries. The following examples can be understood without knowing JPQL, especially if the developer is familiar with SQL; however, you must learn JPQL to build their own queries.

For an introduction to JPQL, use the following link:

http://download.oracle.com/javaee/6/tutorial/doc/bnbtg.html.

To perform a JPQL query use the following workflow:

1. Instantiate a Finder.

2. Initialize any parameters (these parameters are bound to variables in the JPQL expression).

3. Specify the desired result type.

4. Use the **findByJPQL** method to return matching results.

In following example queries, the first is equivalent to the example in the "Find by Entity" section and returns a particular specification. The second uses a join in the JPQL expression to return all physical devices that use this specification.

```
Finder finder = PersistenceHelper.makeFinder();
finder.addParameter("name", "Cisco3640");
finder.setRsultClass(PhysicalDeviceSpecification.class);
Collection< PhysicalDeviceSpecification> specifications = finder.findByJPQL(
                "SELECT o FROM PhysicalDeviceSpecification o " +
                "WHERE o.name = :name");

finder.setRsultClass(PhysicalDevice.class);
Collection< PhysicalDevice> cisco3640Devices = finder.findByJPQL(
                "SELECT o FROM PhysicalDevice o JOIN o.specification s " +
                "WHERE s.name = :name");
```

A JPQL query does not need to return complete entities. It can return one or more attributes from matched entities. To return only name and ID from a physical device, the developer would modify the previous example as follows:

```
Collection cisco3640Devices = finder.findByJPQL(
                "SELECT o.name,o.id FROM PhysicalDevice o JOIN o.specification s
WHERE s.name = :name");
```

```
for (Oject device : cisco3640Devices) {
    Object[] attributes = (cisco3640DevicesObject[]) device;
    System.out.println("Found Cisco 3640 named " + attributes[0] + " with id " +
    attributes[1]);
}
```

The code snippet also shows how to iterate over the results. Since the returned type is not a POMS entity, the attribute values are available as **Object** arrays. You would not set the result class in this case.

While JPQL and the Finder support operations that modify persisted data (update, delete, and so on), you should never modify POMS data with JPQL. The Finder is intended only for read operations.

Find with Paged Results

When working with a large number of entities, process them in smaller batches to reduce memory usage. The Finder supports paged results. Initialize the Finder normally, then specify the range of value to retrieve. This modifies the original physical device example to page through devices 20 at a time:

```
int pageSize = 20;
int start = 0;
while (true) {
    finder.setRange(start, start + pageSize - 1);
    Collection<Physicaldevice> cisco3640Devices = finder.findByJPQL(
        "SELECT o FROM PhysicalDevice o JOIN o.specification s WHERE s.name =
:name");
    for (PhysicalDevice device : cisco3640Devices) {
        System.out.println(device.getName());
        if (cisco3640Devices.size()) < pageSize) {
            break;
        }
        start += pageSize;
    }
}
```

POMS SDK Interfaces

The following are the **PersistenceHelper** API methods:

```
public static < E extends Object > E makeEntity( Class< E > entity );
public static oracle.communications.platform.persistence.Finder makeFinder( ) ;
```

The following are the **Finder** API methods:

```
/**
    * Set the result Class to query.
    *
    * @param resultClass
    *           the interface of each result in the result set
    */
   public void setResultClass(Class resultClass);


/**
    * Set the range of the result set to return, starting of the zero-based
    * start index and ending at the end index, exclusive. For example,
    * setRange(0,5) returns 5 results indexed at 0 thru 4.
    *
    * <p>
    * Setting the range is meaningless if the order of the results is not
```

```
     * consistent. setOrdering is assumed.
     *
     * @param start
     *             zero-based start index
     * @param end
     *             ending index, exclusive
     * @see javax.jdo.Query#setRange
     */
    public void setRange(long start, long end);

    /**
     * Add the parameter name and value that are used to define the filter.
     *
     * <p>
     * Parameter names beginning with an underscore ('_') are illegal. They may
     * conflict with additional parameters used internally by this Finder.
     *
     * @param names
     *             the parameter name to be declared
     * @param param
     *             the parameter value to be bound to the query
     * @throws java.lang.IllegalArgumentException
     *              if illegal parameters are passed
     */
    public void addParameter(String name, Object param);
/**
     * Find entities by example.
     * Any non-null attributes in the example entity is used as the search
criteria, however
     * the attribute names in the mustUseAttributes argument are used as criteria
if the
     * attribute is null.
     *
     * <p>
     * This is a convenience method that performs a simple query in one call.
     * Incremental query construction is not over-written by calling this
     * method.
     *
     * @param entity
     *             the example entity which non-null attributes are used as the
search criteria.
     * @param attributes
     *             list of attribute names which must be used as search criteria
even if their values
     *             in the example entity are null.
     * @param <E>
     *             a oracle.communications.platform.persistence.Persistent type
     * @return Collection of results of the matching entities
     */
    public < E extends Persistent > Collection< E > findByEntity( E entity, String
... mustUseAttributes );
/**
     * This method returns the result of executing a JPQL search using the passed
expression.
     * The caller can pass the query parameter with {@link #addParameter(Integer,
Object) addParameter} or
     * {@link #addParameter(String, Object) addParameter}.
     *
     * @param jpql The JPQL
     * @return Collection of search results
```

```
 */
public Collection findByJPQL(String jpql);
```

About Persist Results

The persistResults method is available in the context of discovery, import and assimilation scan action types. This method persists in-memory result entities to the database and invalidates the entities. You may or may not need to explicitly call this method, depending on the sort of results that your action produces for a given invocation.

If the result set is small (for example, one result group for a particular device), then there is no need to call this method. Your result entities are automatically persisted when the action completes.

If the result set is large (for example multiple devices imported from an inventory system), call persistResults to write the information to the database, reducing memory consumption. In the context of an import action, you would likely want to call the persistResults after results for each device are modeled.

Since persistResults invalidates any in-memory entities, you should not hold a reference to any result entity across a call to persist results.

6

Working with the Extensibility SDK

This chapter provides information about the extensibility SDK for Oracle Communications Network Integrity.

This chapter contains the following sections:

- About Extensibility Scenarios
- Extending MIB II SNMP Discovery for Updated Vendor and Interface Type
- Extending an Existing Cartridge to Discover and Reconcile New Characteristics
- Extending the MIB II SNMP Discovery to Change Interface Name Value
- Multiple Vendor SNMP Discovery
- Multiple Protocol Discoveries

About Extensibility Scenarios

Cartridge projects and actions in Network Integrity are extensible using Oracle Communications Design Studio for Network Integrity. The productized and sample cartridges provided by Network Integrity are designed to be completely extensible and re-usable.

When you make a cartridge project dependent on another, you allow the dependent cartridge project access to the extensible elements from the base cartridge project.

The following sections are examples of some common extensibility scenarios.

Each of the scenarios follows a detailed example but is meant to demonstrate the many extensibility features and methods within Network Integrity cartridge development. The following concepts are demonstrated in the scenarios:

- Re-using existing actions
- Conditional execution using conditions
- The use of specifications and characteristics to extend the model
- The use of input and output parameters
- The use of scan parameter groups and characteristics to extend the Network Integrity UI
- Using filters to modify default discrepancy detection behavior
- What extension points are available in productized cartridges

The scenarios are made up of high-level steps. For more detailed steps, see the Design Studio Help or the Design Studio Modeling Network Integrity Help.

See the following extensibility scenarios:

- Extending MIB II SNMP Discovery for Updated Vendor and Interface Type

   Describes how to update the vendor number and interface type mapping tables in the MIB II SNMP Discovery cartridge.

- Extending an Existing Cartridge to Discover and Reconcile New Characteristics

   Describes how to extend an existing cartridge to discover new data from a device and reconcile this data with an Inventory system.

- Extending the MIB II SNMP Discovery to Change Interface Name Value

   Describes how to extend the MIB II SNMP Discovery action to map the SNMP variable ifName to the interface entity name rather than the entity interface description.

- Multiple Vendor SNMP Discovery

   Describes how to extend an existing cartridge to discover device data from multiple vendors.

- Multiple Protocol Discoveries

   Describes how to extend an existing cartridge to discover data using multiple protocols.

Extending MIB II SNMP Discovery for Updated Vendor and Interface Type

This scenario describes the steps required to update the vendor number and interface type mapping tables in the MIB II SNMP Discovery cartridge. The vendor number table translates an enterprise object identifier number to a vendor name. The interface type table translates an ifType value into a human readable name. These mapping tables are created and output by the MIB II Properties Initializer processor.

The following tasks are performed in this example:

- Adds a new interface type (#333, "tachyonEther")

- Adds a new vendor number (#90210, "West Beverly Hills School District")

- Changes an existing vendor name (#34416, from "Ottawa Area Intermediate School District" to "Ottawa Area Middle School District")

The following cartridges must be loaded in the Design Studio and not have any errors:

- Address_Handlers

- MIB_II_Model

- MIB_II_SNMP_Cartridge

This scenario is made up of high-level steps that are explained in greater detail in the Design Studio Modeling Network Integrity Help.

To extend the MIB II SNMP Discovery cartridge project for updated vendor and interface type information:

1. Create a Network Integrity cartridge project called **Vendor_Type_Update**. Make your cartridge project dependent on the MIB_II_SNMP_Cartridge cartridge project.

2. Create a discovery action called **Discover Updated MIB II SNMP.**

3. In Discover Updated MIB II SNMP, add the Discover MIB II SNMP action as a processor.

4. Create a discovery processor called **MIB II Properties Updater** and place it after the MIB II Properties Initializer processor. This processor will be used to update the two mapping tables.

5. Open the Processor editor **Context Parameters** tab for MIB II Properties Updater and add **snmpIfTypeMap** and **snmpVendorNameMap** as input parameters. These parameters are the output from the MIB II Property Initializer processor.

6. Create the implementation class for this discovery processor. See "Implementing a Processor" for instructions on how to add an implementation class to a processor.

7. Add the implementation code into the body of the **invoke** method of the discovery processor implementation class, similar to the following:

```
// Rename 34416 from "Ottawa Area Intermediate School District"
//   to "Ottawa Area Middle School District"
// Add a new vendor ID 90210 = West Beverly Hills School District
//
Map<String, String> vendorNameMap = request.getSnmpVendorNameMap();
vendorNameMap.put("34416", "Ottawa Area Middle School District");
vendorNameMap.put("90210", "West Beverly Hills School District");

// Add a new interface type 333 as tachyonEther.
//
Map<String, String> ifTypeMap = request.getSnmpIfTypeMap();
ifTypeMap.put("333", "tachyonEther");
```

8. Build, deploy, and test your cartridge.

Figure 6–1 shows the processor workflow of the Discover Updated MIB II SNMP action and the placement of the MIB II Properties Updater processor.

This discovery action inherits all the processors from the Discover MIB II SNMP action. See *MIB-II SNMP Cartridge Guide* for more information.

Figure 6–1   Discover Updated MIB II SNMP Action

Extending an Existing Cartridge to Discover and Reconcile New Characteristics

This scenario describes the steps required to extend an existing cartridge project to discover new data from a device and reconcile this data with an inventory system.

For this scenario the following data is discovered and stored on the physical device:

- Running Configuration Last Saved Date
- Running Configuration Last Modified Date
- Startup Configuration Last Modified Date

The following cartridges must be loaded in the Design Studio and not have any errors:

- Address_Handlers
- ora_ni_uim_cisco_device_sample
- ora_ni_uim_devices
- Cisco_Model
- Cisco_SNMP_Cartridge
- Cisco_UIM_Cartridge
- Cisco_UIM_Model
- MIB_II_Model
- MIB_II_SNMP_Cartridge
- MIB_II_UIM_Cartridge

This scenario is made up of high-level steps that are explained in greater detail in the Design Studio Modeling Network Integrity Help.

To extend a cartridge project to discover and reconcile new characteristics:

1. Create a new UIM cartridge project called Disco_Recon_Specs.

2. From the ora_ni_uim_cisco_device_sample cartridge project, copy and rename the following specifications to Disco_Recon_Specs:

   - cisco3640, rename it to **cisco3640Custom**
   - cat6509, rename it to **cat6509Custom**
   - cisco7206, rename it to **cisco7206Custom**

3. Open the Data Schema editor for Disco_Recon_Specs and create the following characteristics:

   - runningConfigLastSavedDate
   - runningConfigLastChangedDate
   - startupConfigLastChangedDate

4. Add the new characteristics to the renamed specifications.

5. Create a new Network Integrity cartridge project called **Disco_Recon_Char**. Make Disco_Recon_Char dependent on Disco_Recon_Specs and Cisco_UIM_Cartridge.

6. Add the renamed specifications to the model collection for Disco_Recon_Char.

7. Create a discovery action in Disco_Recon_Char and name it **Discover Extended Cisco**.

8. In Discover Extended Cisco, add the Discover Enhanced Cisco SNMP action as a processor.

9. In Discover Extended Cisco, create an SNMP processor called **Custom Cisco Collector**.

10. Perform a web search and download a MIB file called CISCO-CONFIG-MAN-MIB. Do the following:

    a. Copy the MIB file to the MIB directory.

        To find the MIB directory, in Design Studio, on the **Windows** menu, select **Preferences**. In the Preferences dialog box, expand **Oracle Communications Design Studio**, then select **Network Integrity**. The MIB directory is displayed in the dialog box.

    b. Copy the MIB file to the SNMP Adapter on the Network Integrity server. See "Extending the SNMP JCA Resource Adapter" for more information.

    c. On the Processor editor **SNMP** tab, load the CISCO-CONFIG-MAN-MIB file and add the following MIB object IDs to Custom Cisco Collector from CISCO-CONFIG-MAN-MIB.private.enterprises.cisco.ciscoMgmt.ciscoConfig ManMIB.ciscoConfigManMIBObjects.ccmHistory:

        – ccmHistoryRunningLastChanged

        – ccmHistoryRunningLastSaved

        – ccmHistoryStartupLastChanged

11. Create a discovery processor named **Custom Cisco Modeler** to map the new fields to the specifications and characteristics:

    a. On the Processor editor **Context Parameters** tab, add the following input parameters:

        – physicalDevice: output by the Cisco SNMP Physical Modeler processor

        – the document output by Custom Cisco Collector processor

12. Create the implementation class for Custom Cisco Modeler. See "Implementing a Processor" for more information.

13. Add the following implementation code into the **invoke** method that was auto-generated:

    Note: Import statements are required to successfully compile the following code, the imports should all be resolvable by Eclipse with the existing classpath. No classpath changes are necessary.

```
// Get the running config and startup config values from the SNMP response
document
// Keep the values in local variables
CiscoConfigManMibMib configMib =
request.getCustomCiscoCollectorResponseDocument().getDiscoveryResult().getCisco
ConfigManMibResults();
String runningConfigChanged =
Long.toString(configMib.getCcmHistoryRunningLastChanged());
String runningConfigSaved =
Long.toString(configMib.getCcmHistoryRunningLastSaved());
String startupConfigChanged =
Long.toString(configMib.getCcmHistoryStartupLastChanged());
```

```
if (request.getPhysicalDevice() != null) {
   // Get the physical device.
   PhysicalDevice physicalDevice = request.getPhysicalDevice();

   // Get the specification name on the physical device
   String specName = physicalDevice.getSpecification().getName();
   if (specName != null) {

      // Change the specification to the custom specification type
      // and set the new fields
      if (specName.equals(Cisco3640.SPEC_NAME)) {
         Cisco3640Custom custom = new Cisco3640Custom(physicalDevice);
         custom.setRunningConfigLastChangedDate(runningConfigChanged);
         custom.setRunningConfigLastSavedDate(runningConfigSaved);
         custom.setStartupConfigLastChangedDate(startupConfigChanged);
      } else if (specName.equals(Cat6509.SPEC_NAME)) {
         Cat6509Custom custom = new Cat6509Custom(physicalDevice);
         custom.setRunningConfigLastChangedDate(runningConfigChanged);
         custom.setRunningConfigLastSavedDate(runningConfigSaved);
         custom.setStartupConfigLastChangedDate(startupConfigChanged);
      } else if (specName.equals(Cisco7206VXR.SPEC_NAME)) {
         Cisco7206VXRCustom custom = new Cisco7206VXRCustom(physicalDevice);
         custom.setRunningConfigLastChangedDate(runningConfigChanged);
         custom.setRunningConfigLastSavedDate(runningConfigSaved);
         custom.setStartupConfigLastChangedDate(startupConfigChanged);
      }
   }
}
```

Figure 6–2 shows the processor workflow of the Discover Extended Cisco action and the placement of the Custom Cisco Collector and Custom Cisco Modeler processors.

This discovery action inherits all the processors from the Discover Enhanced Cisco SNMP action. See *Cisco Router and Switch UIM Cartridge Guide* for more information.

Figure 6–2   Discover Extended Cisco Action

MIB II Properties Initializer

Cisco SNMP Properties Initializer

MIB II SNMP Collector

MIB II SNMP Modeler

Cisco SNMP Logical Collector

Cisco SNMP Physical Collector

Cisco SNMP Logical Modeler

Cisco SNMP Physical Modeler

Cisco Enhanced Modeler

Custom Cisco Collector

Custom Cisco Modeler

14. Create a discrepancy detection action named **Detect Extended Cisco**:

   a. On the Action editor **Result Source** tab, add the Discover Extended Cisco action as the result source. This indicates that the extended discrepancy detection action applies to extended discovery results.

   b. On the Action editor **Processors** tab, add the Detect Enhanced Cisco Discrepancies action as a processor.

     See *Cisco Router and Switch UIM Cartridge Guide* for more information about the Detect Enhanced Cisco Discrepancies action.

15. Create a discrepancy resolution action named **Resolve Extended Cisco in UIM**:

   a. On the Action editor **Details** tab, enter **Correct in UIM** in the **Resolution Action Label** field.

   b. On the Action editor **Result Source** tab, add the Discover Extended Cisco action as the result source.

   c. On the Action editor **Processor** tab, add the Resolve Cisco in UIM action as a processor.

     See *Cisco Router and Switch UIM Cartridge Guide* for more information about the Resolve Cisco in UIM action.

16. (Optional) Create an import action.

The existing Import Cisco from UIM action available in the Cisco UIM cartridge imports the extended devices types with new characteristics. Create this import action if you want to deploy the Extensibility cartridge without also deploying the Cisco UIM cartridge.

   a. Create an import action called **Import Extended Cisco from UIM**. See the Design Studio Modeling Network Integrity Help for information about how to create an import action, and how to extend existing import actions.

   b. On the Action editor **Processors** tab add the Import Cisco from UIM action as
   a processor.

   See *Cisco Router and Switch UIM Cartridge Guide* for more information about
   the Import Cisco from UIM action.

Extending the MIB II SNMP Discovery to Change Interface Name Value

This scenario describes the steps required to extend the MIB II SNMP discovery action
to map the ifName to the interface name rather than the interface description. In
addition, this scenario exposes a scan parameter that the end-user can use to control
the behavior of the interface name mapping.

Note: Changing how the name field is mapped affects how generic
discrepancy detection looks up import entities because the lookup is
done using name field (this can be modified using discrepancy
detection filters, see "About Filters" for details). If the interface name
field is modified for discovery, but is not modified on the import data,
many 'extra entity' discrepancies are produced because discrepancy
detection is unable to find the interface of the import side.

Avoid this issue by ensuring that the name field for discovery and
import are identical, or by using a different field than name to look up
the interface on the import side. An example of using a different field
is in the Detect MIB II UIM Discrepancies action in the MIB_II_UIM_
Cartridge. This discrepancy detection action overrides the default
lookup to use the NativeEMSName instead of the name field.

The following high-level steps are involved in this scenario:

- Create new Network Integrity cartridge project

- Create new discovery action that re-uses an existing discovery action

- Create new scan parameter groups with new characteristics

- Add new processor to change mapping of interface name

The following cartridges must be loaded in the Design Studio and not have any errors:

- Address_Handlers

- MIB_II_Model

- MIB_II_SNMP_Cartridge

This scenario is made up of high-level steps that are explained in greater detail in the
Design Studio Modeling Network Integrity Help.

To extend the MIB II SNMP cartridge to change the interface name value:

1. Create a Network Integrity cartridge project called **InterfaceName**. Make your
   cartridge project dependent on the MIB_II_SNMP_Cartridge cartridge project.

2. Create a discovery action called **Discover Custom MIB II SNMP.**

3. In Discover Custom MIB II SNMP, add the Discover MIB II SNMP action as a
   processor.

4. Create a scan parameter group called **MIBIICustomParameters**. Add the scan
   parameter group to the Discover Custom MIB II SNMP action.

5. For MIBIICustomParameters, create a characteristic called
   mapIfDescToInterfaceName.

6. Add two enumeration values to mapIfDescToInterfaceName:

   a. Open the Data Schema editor for mapIfDescToInterfaceName.

   b. Click the **Enumerations** subtab.

   c. Add an enumeration called **True** and another called **False**.

   d. In the **Default** column, set **True** to be the default value.

7. On the Scan Parameter Group editor Layouts tab for MIBIICustomParameters, do
   the following:

   a. Select mapIfDescToInterfaceName.

      The UI Settings area displays the scan parameter values for
      mapIfDescToInterfaceName.

   b. In the **Display Name** field, enter **Map Description to Interface Name**.

      This is the name that will appear in the Network Integrity UI for the scan
      parameter.

8. Save all changes.

9. In the Discover Custom MIB II SNMP action, create a discovery processor called
   Custom Interface Name Modeler.

10. Open the Processor editor **Context Parameters** tab for Custom Interface Name
    Modeler and add **logicalDevice** as an input parameter. This parameter is the
    output from the MIB II SNMP Modeler processor.

11. On the Processor editor **Details** tab, create the implementation class for the
    discovery processor.

12. Add the implementation code similar to the following:

> **Note:** Import statements are required to successfully compile the
> following code, the imports should all be resolvable by Eclipse with
> the existing classpath, no classpath changes are necessary.

```
@Override
public void invoke(DiscoveryProcessorContext context,
  CustomInterfaceNameModelerProcessorRequest request)
  throws ProcessorException {

  // if the user specified they do not want the ifDesc as the name of the
interface then use the ifName instead

  if
("false".equalsIgnoreCase(request.getMibiiCustomParameters().getMapIfDescToInte
rfaceName())) {
    List<DeviceInterface> deviceInterfaces =
request.getLogicalDevice().getDeviceInterfaces();
    changeInterfaceNameToIFName(deviceInterfaces);
    }
  }

private void changeInterfaceNameToIFName(List<DeviceInterface>
deviceInterfaces) {
```

```
// loop through every interface and change the mapping.
  for (DeviceInterface deviceInterface : deviceInterfaces) {
// the Discover MIB II SNMP Discovery Action is inserting the ifName into the
VendorInterfaceNumber so the following code copies that to the name field
    deviceInterface.setName(deviceInterface.getVendorInterfaceNumber());
  // Change interface name on any sub-interfaces as well
    changeInterfaceNameToIFName(deviceInterface.getSubInterfaces());
  }
}
```

13. To register discrepancy detection and discrepancy resolution on the new Discover Custom MIB II SNMP discovery action, add new result sources to the Detect MIB II UIM Discrepancies and Resolve MIB II in UIM in the MIB_II_UIM_Cartridge that register for results from the Discover Custom MIB II SNMP discovery action. See "About Discrepancy Detection Actions" and "About Discrepancy Detection Processors" for details. (Alternatively, the Detect MIB II UIM Discrepancies and Resolve MIB II in UIM actions could be extended in the InterfaceName_Cartridge. See "Extending an Existing Cartridge to Discover and Reconcile New Characteristics" Extensibility Scenario for details on doing this).

Figure 6–3 shows the processor workflow of the Discover Custom MIB II SNMP action and the placement of the Custom Interface Name Modeler processor.

This discovery action inherits all the processors from the Discover MIB II SNMP action. See *MIB-II SNMP Cartridge Guide* for more information.

Figure 6–3   Discover Custom MIB II SNMP Action

Multiple Vendor SNMP Discovery

This scenario describes the steps required to extend an existing cartridge to discover data from devices from multiple vendors.

The following cartridges must be loaded in the Design Studio and not have any errors:

- Address_Handlers
- ora_ni_uim_devices
- MIB_II_Model
- MIB_II_SNMP_Cartridge
- MIB_II_UIM_Cartridge
- Cisco_Model
- Cisco_SNMP_Cartridge
- Cisco_UIM_Cartridge
- Cisco_UIM_Model

There are multiple scenarios, depending on your objectives.

One way is you want to discover devices from a single vendor. You should then extend the MIBII SNMP cartridge by reusing the Discover MIB II SNMP action and adding an SNMP Collector and an SNMP Modeler for the vendor. The SNMP Collector polls vendor-specific MIBs and the SNMP Modeler models the devices based on the collected SNMP OIDs.

Another way is you want to discover multiple vendor devices, for example, Cisco and Juniper devices. You should extend the Enhanced Cisco SNMP action in the Cisco UIM cartridge.

Use the sysObjectId from RFC1213MIB to determine a device vendor. For example, Cisco devices have the sysObjectId value that starts with **1.3.6.1.4.1.9**, and Juniper device have the sysObjectId value starting with **1.3.6.1.4.1.2636**. Set up a range of IP addresses and scan those IP addresses by polling the sysObjectId. Based on the sysObjectValue returned, configure two conditions: one returns true if the sysObjectId value starting with **1.3.6.1.4.1.9** (meaning it is a Cisco device), or return false if otherwise; the other return true if the sysObjectId value starting with **1.3.6.1.4.1.2636** (meaning it is a Juniper device), or return false if otherwise.

The Cisco UIM cartridge contains the Discover Enhanced Cisco SNMP action. Create a discovery action by reusing this Discover Enhanced Cisco SNMP action, which gives this new discovery action all the functions to discover the enhanced Cisco devices (including the MIB II SNMP discovery). Extend this discovery action to support Juniper devices by creating a Juniper SNMP collector and a Juniper modeler to this discovery action. The two conditions determine when to execute the Cisco related collectors and modelers and when to execute the Juniper collector and modeler based on the device type.

This scenario is made up of high-level steps that are explained in greater detail in the Design Studio Modeling Network Integrity Help.

To extend a cartridge to discover devices from multiple vendors:

1. Create a Network Integrity cartridge project called **Multi-Vendor**. Make your cartridge project dependent on the Cisco_UIM_Cartridge cartridge project.

2. Create a discovery action called **Discover Multi-Vendor.**

3. In Discover Multi-Vendor, add the Discover Enhanced Cisco SNMP action as a processor.

4. Manually copy the JUNIPER-MIB to the MIB directory and to the SNMP adapter on the Network Integrity server. See "Supporting New MIBs" and "Extending the SNMP JCA Resource Adapter" for more information.

5. Create an SMMP processor called **Juniper SNMP Collector** and add it to Discover Multi-Vendor as the last processor.

6. In Juniper SNMP Collector, add the OID **jnxBoxDescr** (from JUNIPER-MIB).

In a production environment, you would add more OIDs to poll and model more information.In this example, only the description field is polled.

7. Create an SNMP processor called **Juniper SNMP Modeler** and add it to Discover Multi-Vendor as the last processor. This processor takes the SNMP output parameter from Juniper SNMP Collector as its input parameter. Implement this processor by implementing the **invoke** method. In this example, only the description field for the Juniper device is logged. In a realistic scenario, the complete model of the Juniper device would exist in this **invoke** method.

The following is the Java snippet for the **invoke** method.

```
@Override
public void invoke(DiscoveryProcessorContext context,
JuniperProcessorProcessorRequest request) throws ProcessorException {
  logger.log(Level.INFO, "Processing Juniper device " +
  request.getScopeAddress());
  JuniperSNMPCollectorResponseType responseDoc =
request.getJuniperSNMPCollectorResponseDocument();
  DiscoveryResultType result = responseDoc.getDiscoveryResult();
  JuniperMibMib juniperMibResults = result.getJuniperMibResults();
  if(juniperMibResults != null) {
    logger.log(Level.INFO, "Juniper Device Description: " +
juniperMibResults.getJnxBoxDescr());
  }
}
```

8. Create a Cisco condition that checks the sysObjectId to determine whether a device is a Cisco device or not. This condition takes the **mibiisnmpCollectorResponseDocument** (an output parameter from MIB II SNMP Collector) as the input parameter. The following is a Java snippet for this Cisco condition:

```
public class CiscoConditionImpl implements CiscoCondition {
  private static final String CISCO_PREFIX = "1.3.6.1.4.1.9.";
@Override
  public boolean checkCondition(DiscoveryProcessorContext context,
  CiscoRequest request) throws ProcessorException {
    MIBIISNMPCollectorResponseType snmpResponse = request
    .getMibiisnmpCollectorResponseDocument();
    logger.log(Level.INFO, "CiscoConditionImpl"
    + context.getProcessorName());
    if (snmpResponse != null
    && snmpResponse.getDiscoveryStatus() == DiscoveryStatus.SUCCESS) {
      logger.log(Level.INFO, "CiscoConditionImpl discovery succeeded");
      if (snmpResponse.getDiscoveryResult().getRfc1213MibResults() != null) {
        String sysObjectId = snmpResponse.getDiscoveryResult()
        .getRfc1213MibResults().getSysObjectID();
        logger.log(Level.INFO, "CiscoConditionImpl raw sys object id:" +
        sysObjectId);
        if (sysObjectId != null) {
          if (sysObjectId.startsWith(".")) {
            sysObjectId = sysObjectId.substring(1);
          }
          return sysObjectId.startsWith(CISCO_PREFIX);
        }
```

```
        }
      }
      return false;
    }
  }
```

9. Create a Juniper condition, that checks the sysObjectId to determine whether a device is a Juniper device. This condition takes the **mibiisnmpCollectorResponseDocument** (an output parameter from MIB II SNMP Collector) as the input parameter. This Juniper condition is similar to the Cisco condition. The difference is that the sysObjectId for Juniper device starts with **1.3.6.1.4.1.2636.**.

10. Apply the Cisco condition to the following processors and set **Equals** to **true**:

   ■ Cisco SNMP Logical Collector

   ■ Cisco SNMP Physical Collector

   ■ Cisco SNMP Logical Modeler

   ■ Cisco SNMP Physical Modeler

   ■ Cisco Enhanced Modeler

   By applying the Cisco condition, the processors are invoked only if the device is a Cisco device.

11. Apply the Juniper condition to the following Juniper processors and set **Equals** to **true**:

   a. Juniper SNMP Collector

   b. Juniper SNMP Modeler

> **Note:** In this example, only a single Juniper OID is collected and the value of the collected Juniper OID in the Juniper SNMP Modeler is logged. In a realistic scenario, several Juniper OIDs are collected to model a Juniper device. See "Extending an Existing Cartridge to Discover and Reconcile New Characteristics" on how to map new SNMP OIDs to new Characteristics and how to update UIM related actions for importing, discrepancy detection and resolution with the new Characteristics.

Figure 6–4 shows the processor workflow of the Discover Multi-Vendor action and the placement of the Juniper SNMP Collector and Juniper Modeler processors.

This discovery action inherits all the processors from the Discover Enhanced Cisco SNMP action. See *Cisco Router and Switch UIM Integration Cartridge Guide* for more information.

Figure 6–4   Discover Multi-Vendor Action

Multiple Protocol Discoveries

This scenario describes the steps required to extend an existing cartridge to discover data using multiple protocols.

The following cartridges must be imported into the Design Studio and build without errors:

- Address_Handlers
- Cisco_Model
- Cisco_SNMP_Cartridge
- Cisco_UIM_Cartridge
- Cisco_UIM_Model
- MIB_II_Model
- MIB_II_SNMP_Cartridge
- MIB_II_UIM_Cartridge

In this scenario, a range of devices can be discovered. Some devices are SNMP-enabled; some devices support an alternate protocol (for example, TL1). With a list of IP addresses for each of these devices, the discovery action can dynamically discover a device using either SNMP or the alternate protocol.

The Cisco UIM Sample cartridge contains the sample Discover Enhanced Cisco SNMP action. Create a discovery action that reuses the Discover Enhanced Cisco SNMP action. This discovery action can be extended to support the alternate protocol by creating a discovery processor that implements the alternate protocol to this discovery action. To use a JCA resource adapter for this alternate protocol, see "Working with JCA Resource Adapters".

Create a condition that checks whether the SNMP polling to a device is successful or not. If a device supports SNMP, this condition returns true; otherwise if the device supports the alternate protocol, this condition returns false. By applying this condition to the processors, the discovery action can dynamically discover a device using either SNMP or the alternate protocol.

This scenario is made up of high-level steps that are explained in greater detail in the Design Studio Modeling Network Integrity Help.

To extend a cartridge to discover devices using multiple protocols:

1. Create a discovery action called **Discover MultiProtocol** and make it dependent on the Cisco_UIM_Cartridge cartridge project.

2. Create a discovery action called **Discover Multi-Protocol.**

3. In Discover Multi-Protocol, add the Discover Enhanced Cisco SNMP action as a processor.

4. Create a discovery processor called **Alternate Protocol Collector** to implement the alternate protocol to discover a device and add it to Discover Multi-Protocol as the last processor.

5. Implement Alternate Protocol Collector by implementing the **invoke** method. In this example, one line is logged indicating that this processor implements an alternate protocol. In a realistic scenario, implement the alternate protocol to discover a device in this **invoke** method. The following is the Java snippet for the **invoke** method:

```
@Override
  public void invoke(DiscoveryProcessorContext context,
  AlternateProtocolCollectorProcessorRequest request)
  throws ProcessorException {
    logger.log(Level.INFO, "SNMP Failed - using alternate protocol to discover
device " + request.getScopeAddress());
  }
```

6. Create a condition called **SnmpSucceeds** that checks the SNMP results from MIB II Collector to determine whether the SNMP discovery on a device is successful or not. This condition takes mibiisnmpCollectorResponseDocument (an output parameter from MIB II SNMP Collector) as the input parameter. The following is a Java snippet for this SnmpSucceeds condition:

```
public class SnmpSucceedsConditionImpl implements SnmpSucceedsCondition {
  @Override
  public boolean checkCondition(DiscoveryProcessorContext context,
  SnmpSucceedsRequest request) throws ProcessorException {
    MIBIISNMPCollectorResponseType snmpResponse =
request.getMibiisnmpCollectorResponseDocument();
    return snmpResponse != null && snmpResponse.getDiscoveryStatus() ==
DiscoveryStatus.SUCCESS;
  }
}
```

7. Apply the SnmpSucceeds condition to the following processors and set the **Equals** to be **true**:

   ■ MIB II SNMP Modeler

   ■ Cisco SNMP Logical Collector

   ■ Cisco SNMP Physical Collector

- Cisco SNMP Logical Modeler

- Cisco SNMP Physical Modeler

- Cisco Enhanced Modeler

By applying the SnmpSucceeds condition, these processors are invoked only if the SnmpSucceeds condition returns **true**.

8. Apply the SnmpSucceeds condition to the Alternate Protocol Collector processor and set the **Equals** to be **false**.

> **Note:** In this example, only the message is logged to indicate that an alternate protocol is used in the Alternate Protocol Collector processor. In a realistic scenario, the alternate protocol would be implemented and the network data collected using this protocol and model the collected network data. See "Extending an Existing Cartridge to Discover and Reconcile New Characteristics" section on how to map the collected network data (in that section, the network data is SNMP OID) to new Characteristics and how to update UIM related actions for importing, discrepancy detection and resolution with the new Characteristics.

Figure 6–5 shows the processor workflow of the Discover MultiProtocol action and the placement of the Alternate Protocol Collector processor.

This discovery action inherits all the processors from the Discover Enhanced Cisco SNMP action. See *Cisco Router and Switch UIM Integration Cartridge Guide* for more information.

Figure 6–5   Discover MultiProtocol Action

7
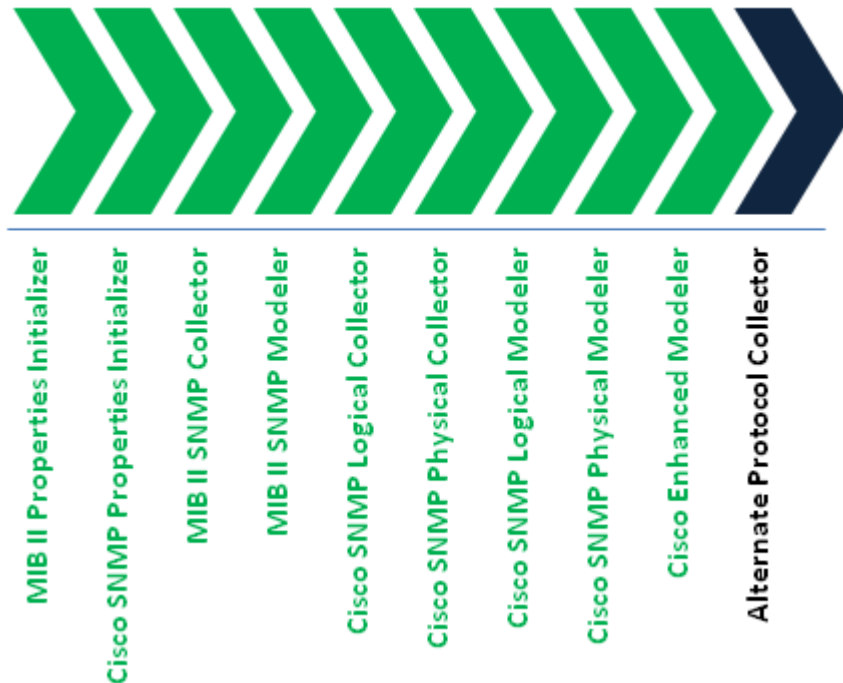
Working with Automatic Discrepancy Resolution

This chapter explains how to design a discrepancy detection action that allows Oracle Communications Network Integrity to automatically resolve specific types of discrepancies.

About Automatic Discrepancy Resolution

Automatic discrepancy resolution enables Network Integrity to automatically resolve specific discrepancies without the user having to interact with the UI. Discrepancies are resolved as part of the discrepancy detection scan.

Network Integrity identifies automatically resolved discrepancies. In the scan results, automatically resolved discrepancies have the value **autoResolve** in the Submitted By and Resolved By columns.

The NetworkIntegritySDK cartridge project contains an abstract action that makes up the framework for automatic discrepancy resolution.

Using the Design Studio for Integrity feature, extend cartridges that detect discrepancies with the abstract automatic discrepancy resolution action. Oracle Communications Design Studio creates the framework implementation for you to complete.

You can complete the implementation by creating either a custom processor or with a properties file.

After you deploy your cartridges with the new implementation into the run-time application, users of Network Integrity can configure scans that automatically resolve all the discrepancies matching the implementation you created.

About the Automatic Discrepancy Resolution Solution

This section describes the components that make up the automatic discrepancy resolution framework. Also, this section identifies reference implementations that you can use as examples to help create your own solution.

Action and Processors

The NetworkIntegritySDK cartridge project contains an abstract discrepancy detection action called Auto Resolve Discrepancies. This abstract action contains the framework for automatic discrepancy resolution.

The Auto Resolve Discrepancies action has the following processors:

- Check Auto Resolution Selected: this processor verifies whether a scan is configured with the **Auto Resolve Discrepancies** option enabled. If enabled, this processor sets a flag to run the next processors.

- Identify Auto Resolving Discrepancies: this processor identifies the discrepancies that match the customized implementation.

- Prepare Resolving Discrepancies: this processor puts all the identified discrepancies in the DISCREPANCY_SUBMITTED state.

The automatic discrepancy resolution implementation can be completed with either a custom processor or with a properties file. If you complete the implementation with a custom processor, you must create a new discrepancy detection processor in the action that extends the Auto Resolve Discrepancies action.

Figure 7–1 illustrates the processor workflow of the automatic discrepancy resolution solution. The DD Processor for Java Implementation processor is not required for an implementation that uses a properties file.

Figure 7–1   Auto Resolve Discrepancies Processor Workflow

Scan Parameter Groups and the Network Integrity UI

NetworkIntegritySDK contains a scan parameter group called AutoResolutionParameter. This scan parameter group adds the **Auto Resolve Discrepancies** check box to the Network Integrity UI Scan Configuration screen.

Reference Implementations

The Network Integrity MSS Integration cartridge demonstrates a complete reference implementation of automatic discrepancy resolution using a custom processor and a properties file.

The Network Integrity Optical UIM Integration cartridge demonstrates a complete reference implementation of automatic discrepancy resolution using a custom processor.

The Network Integrity Optical TMF814 CORBA cartridge includes the AutoResolutionParameter scan parameter group.

Implementing Automatic Discrepancy Resolution

This section assumes that you already have valid, deployable cartridges that perform discovery, import, and discrepancy detection, to which you are adding automatic discrepancy resolution.

If your existing cartridge solution is made up of unsealed cartridges, see "Implementing Automatic Discrepancy Resolution in an Unsealed Cartridge Solution".

If your existing cartridge solution contains one or more sealed cartridges, see "Implementing Automatic Discrepancy Resolution in a Sealed Cartridge Solution".

Implementing Automatic Discrepancy Resolution in an Unsealed Cartridge Solution

See the Design Studio for Network Integrity Help for information about any of the steps in this section.

To implement automatic discrepancy resolution when working with unsealed cartridges:

1. In your cartridge with a fully implemented discrepancy detection action:

   - Add the abstract Auto Resolve Discrepancies action from NetworkIntegritySDK as a processor to a discrepancy detection action.

   - Move the processors belonging to Auto Resolve Discrepancies to the end of the **Action Processors** list.

2. In your cartridge with a fully implemented import or discovery action, add the AutoResolutionParameter scan parameter group.

   Add the scan parameter group to the action that is the result source for discrepancy detection action.

3. Complete and customize the implementation for automatic discrepancy resolution. See "Completing the Automatic Discrepancy Resolution Implementation" for more information.

4. Save and close all files.

5. Build, deploy, and test your cartridge.

Implementing Automatic Discrepancy Resolution in a Sealed Cartridge Solution

See the Design Studio for Network Integrity Help for information about any of the steps in this section.

To implement automatic discrepancy resolution when working with a sealed cartridge:

1. Create a new cartridge.

2. Add the following dependencies to the new cartridge:

   - All sealed and unsealed cartridges being extended by the new cartridge

> **Note:** The new cartridge needs to extend a discovery action and a discrepancy detection action. These actions may belong to one or more cartridges. At least one of these cartridges is sealed.

- ora_ni_uim_device
- NetworkIntegritySDK
- Address Handler

3. Create a new discovery action in the new cartridge.

4. For the new discovery action:

   - Specify **IPAddressHandler** as the address handler.
   - Specify **Device** as the result category.
   - Add *discoveryAction* as a processor.

     Where *discoveryAction* is a discovery action from another cartridge.

   - Add the AutoResolutionParameter scan parameter group.
   - Add or create any additional scan parameter groups required to configure the new discovery action.

5. Create a new discrepancy detection action in the new cartridge.

6. For the new discrepancy detection action:

   - Add *ddAction* as a processor.

     Where *ddAction* is a discrepancy detection action from another cartridge that uses the result source from *discoveryAction*.

   - Add the abstract Auto Resolve Discrepancies action as a processor.
   - Move the processors belonging to Auto Resolve Discrepancies to the end of the **Action Processors** list.
   - Specify the new discovery action as the result source.

7. For the discrepancy resolution action whose result source is *ddAction*, add the new discrepancy detection action as a result source.

8. Complete and customize the automatic discrepancy resolution implementation for the new cartridge. See "Completing the Automatic Discrepancy Resolution Implementation" for more information.

9. Save and close all files.

10. Build, deploy, and test your cartridge.

Completing the Automatic Discrepancy Resolution Implementation

You can complete the automatic discrepancy resolution implementation in the following ways:

- Completing Automatic Discrepancy Resolution Using a Properties File
- Completing Automatic Discrepancy Resolution with a Custom Processor

Completing Automatic Discrepancy Resolution Using a Properties File

Create a file called **autoResolve.properties** in the **/src** directory in the cartridge with the automatic discrepancy resolution action. Use this properties file to configure the discrepancies that can be automatically resolved.

The **autoResolve.properties** file is a list of property/value pairs. The accepted properties are:

- extraEntities, for resolving extra entity discrepancies.

- missingEntities, for resolving missing entity discrepancies.

- mismatches, for resolving attribute value mismatch discrepancies.

- extraAssociation, for resolving extra association discrepancies.

- missingAssociation, for resolving missing association discrepancies.

See "About Discrepancy Types" for more information about discrepancies.

The properties file uses the following syntax:

*property***=***res_label1***:***entity_type1***[***spec_name1***:***attrib_list1***|***spec_name2***:***attrib_list2***]{}***...

where:

- *property* is one of the accepted properties. Each property can appear once in the properties file. Each property can specify multiple resolution labels, entity types, specification names, and attribute lists.

- *res_label* is the resolution label you want Network Integrity to use to resolve the discrepancy. You can specify multiple resolution labels to resolve discrepancies to different inventory systems.

- *entity_type* is a type of entity (for example, an Equipment or a Physical Device entity).

- *spec_name* is the specification for the entity type. You can omit *spec_name* if the same resolution label applies to all specifications for the entity type.

- *attrib_list* is a comma-separated list of attributes on the entity or specification to be resolved.

Example 7–1 demonstrates an automatic discrepancy resolution implementation completed using a properties:

Example 7–1   Sample autoResolve.properties File

```
extraEntities=Correct in MSS:Equipment[tmf814EquipmentGeneric]{}Correct in
UIM:PhysicalDevice[cisco3640|cisco7206VXR]
missingEntities=Correct in MSS:Equipment{}Correct in UIM:PhysicalDevice[cisco3640]
mismatches=Correct in MSS:Equipment[:serialNumber]{}Correct in
UIM:PhysicalDevice[cisco3640:softwareVer,serialNumber|cisco7206VXR:hardwareRev]
extraAssociations=Correct in UIM:LogicalDevice
```

Example 7–1 demonstrates a properties file that does all of the following:

- The line starting with **extraEntities** resolves in MSS all extra entity discrepancies on equipment entities with the tmf814EquipmentGeneric specification, and resolves in UIM all extra entity discrepancies on physical device entities with the cisco3640 or cisco7206VXR specifications.

- The line starting with **missingEntities** resolves in MSS all missing entity discrepancies on equipment, and resolves in UIM all missing entity discrepancies on physical device entities with the cisco3640 specification.

- The line starting with **mismatches** resolves in MSS all serial number attribute value mismatch discrepancies on equipment entities, and resolves in UIM all software version and serial number attribute value mismatch discrepancies on physical device entities with the cisco3640 specification, and all hardware revision attribute value mismatch discrepancies on physical device entities with the cisco7206VXR specification.

- The line starting with **extraAssociation** resolves in UIM all extra association discrepancies on logical device entities.

See the reference implementation properties file in the MSS Integration cartridge to use as a starting point. The reference properties file includes comments, examples, syntax, and tips to help you complete your implementation.

Completing Automatic Discrepancy Resolution with a Custom Processor

See the Design Studio for Network Integrity Help for information about any of the steps in this section.

To implement automatic discrepancy resolution with a custom processor:

1. In the action that contains the Auto Resolve Discrepancies action, create a new discrepancy detection processor.

2. Move the new processor after the Check Auto Resolution Selected processor.

3. Add **autoResolutionManager** as an input parameter for the new discrepancy detection processor.

4. Create and complete the implementation class for the new discrepancy detection processor.

   See the reference implementation class from the Optical UIM Integration cartridge to use as a starting point. The reference implementation class includes comments, examples, syntax, and tips to help you complete your own implementation.

8

Working with Incremental TMF814 Discovery

This chapter explains how to design a discovery action that allows Oracle Communications Network Integrity to discover only the network elements that have changed in the network since the last discovery scan.

About Incremental TMF814 Discovery

Incremental TMF814 discovery enables Network Integrity to discover only the network elements, such as managed elements (MEs), topological kinks (TLs), and subnetwork connections (SNCs), that have changed in the network since the last discovery scan; thus, avoiding the need to discover the entire network.

The NetworkIntegritySDK cartridge project contains an abstract action that makes up the framework for incremental TMF814 discovery.

Using the Design Studio for Integrity feature, extend cartridges to run incremental discovery using the abstract incremental discovery action.

After you deploy your cartridges with the new implementation into the run-time application, users of Network Integrity can configure scans that discover only the network elements that have changed in the network since the previous discovery scan.

About the Incremental TMF814 Discovery Solution

This section describes the components and the framework that make up the incremental TMF814 discovery. Also, this section identifies reference implementations that you can use as examples to help create your own solution.

Action and Processors

The NetworkIntegritySDK cartridge project contains an abstract discovery action called Abstract Incremental Discovery, which contains the framework for incremental TMF814 discovery.

The Abstract Incremental Discovery action has the following processor:

- **Incremental Discovery Initializer**: This processor validates the scan parameters and verifies whether the scan is enabled for incremental scan.

Copying Information From Previous Scan Results

The NetworkIntegritySDK cartridge project contains the framework that copies the information from previous scan results. The EntityCopier object provides the extensible interface that the copier entities extend. The EntityCopierFactory object generates the copier instances.

Table 8–1 lists the copier objects used to copy information from the results of the previous scan.

Table 8–1   Copier Objects From Previous Scan Results

Entity Name	Entity Copier
DeviceInterfaceConfigurationItem	DeviceInterfaceConfigItemCopier
DeviceInterface	DeviceInterfaceCopier
Equipment	EquipmentCopier
EquipmentHolder	EquipmentHolderCopier
InventoryGroup	InventoryGroup
LogicalDeviceConfigurationItem	LogicalDeviceConfigItemCopier
LogicalDevice	LogicalDeviceCopier
MediaInterface	MediaInterfaceCopier
PhysicalDevice	PhysicalDeviceCopier
PhysicalPort	PhysicalPortCopier
Pipe	PipeCopier
PipeTerminationPoint	PipeTerminationPointCopier
TrailPath	TrailPathCopier

You must create your own copier objects for any custom entities in your custom cartridge projects. Then you must modify the EntityCopier object to implement custom copier objects. Then you must extend the EntityCopierFactory object to add the new copiers.

The EntityCopier interface has the following methods:

- *copyEntity(T entity)* - create a new T entity and copy all its attributes & specification characteristics from T entity.

- *deepCopyEntity(T entity)* - create a new T entity and copy all its attributes, calls the deepCopyEntity() for its child entities to be created.

    For example, *deepCopyEntity()* of LogicalDeviceCopier, return new LogicalDevice with its complete hierarchy which includes MediaInterfaces, Device Interfaces, LogicalDeviceConfigurationItems, children of MediaInterfaces, DeviceInterfaces as similar to the givenLogicalDevice entity. *CopyEntity()* of LogicalDevice returns the new LogicalDevice entity by copying all attributes and specification characteristic from the given LogicalDevice entity.

Scan Parameter Groups and the Network Integrity UI

NetworkIntegritySDK contains a scan parameter group called IncrementalScanParameter. This scan parameter group adds the **Incremental Scan** check box to the Network Integrity UI Scan Configuration screen.

Reference Implementations

The Network Integrity Incremental TMF814 Discovery cartridge demonstrates a complete reference implementation of incremental TMF814 discovery. The Incremental TMF814 Discovery cartridge includes the IncrementalScanParameter scan parameter group.

Implementing Incremental TMF814 Discovery

This section assumes that you already have valid, deployable cartridges that perform discovery, import, and discrepancy detection, to which you are adding incremental TMF814 discovery. You can implement incremental TMF814 discovery in a sealed cartridge solution.

Implementing Incremental TMF814 Discovery in a Sealed Cartridge Solution

See the Design Studio for Network Integrity Help for information about any of the steps in this section.

To implement incremental TMF814 discovery when working with a sealed cartridge:

1. Create a new cartridge.

2. Add the following dependencies to the new cartridge:

   ■ All sealed and unsealed cartridges being extended by the new cartridge

   > **Note:** The new cartridge needs to extend a discovery action and a discrepancy detection action. These actions may belong to one or more cartridges. At least one of these cartridges is sealed.

   ■ ora_ni_uim_device

   ■ NetworkIntegritySDK

   ■ Address Handler

3. Create a new discovery action in the new cartridge.

4. For the new discovery action:

   ■ Specify **IPAddressHandler** as the address handler.

   ■ Specify **Device** as the result category.

   ■ Add *discoveryAction* as a processor.

     Where *discoveryAction* is a discovery action from another cartridge.

   ■ Add Abstract Incremental Discovery action as a processor.

   ■ Add the IncrementalScanParameter scan parameter group.

   ■ Add or create any additional scan parameter groups required to configure the new discovery action.

5. Complete and customize the incremental discovery implementation for the new cartridge.

6. Save and close all files.

7. Build, deploy, and test your cartridge.

9

Working with CPU Utilization-enabled Discovery

This chapter explains how to design a discovery action that allows Oracle Communications Network Integrity to discover devices based on their CPU utilization.

About CPU Utilization-enabled Discovery

CPU utilization-enabled discovery provides the mechanism to manage the discovery of devices based on their CPU utilization. This is an optional feature that enables you to configure the CPU utilization threshold value in cartridges, which enables the scan to skip the devices that are running above the specified CPU utilization threshold value.

The NetworkIntegritySDK cartridge project contains an abstract action with two processors and one scan parameter group that constitute the framework for CPU utilization-enabled discovery. See "Action and Processors" and "Scan Parameter Groups and the Network Integrity UI" for more information.

Using Design Studio for this feature, extend cartridges to run discovery using the Abstract CPU Utilization Discovery action.

After you deploy your cartridges with the new implementation into the run-time application, users of Network Integrity can configure scans that discover only those devices that are running below the user-specified CPU utilization threshold value.

About CPU Utilization-enabled Discovery Solution

This section describes the components and the framework that make up the incremental TMF814 discovery. Also, this section identifies reference implementations that you can use as examples to help create your own solution.

Action and Processors

The NetworkIntegritySDK cartridge project contains an abstract discovery action called Abstract CPU Utilization Discovery, which contains the framework for CPU utilization-enabled discovery.

The Abstract CPU Utilization Discovery action has the following processors:

- **CPU Property Initializer**: This processor initializes the **cpuProperties** file that contains the deviceCPUValue variable, which is required to store the CPU utilization value of the device obtained by the network.

■ **CPU Utilization Compare Processor**: This processor is responsible for comparing the user-specified threshold value with the CPU utilization value of the device obtained by the network.

About the Mechanism of Comparing CPU Usage Values

The NetworkIntegritySDK cartridge project contains the framework that compares the CPU utilization threshold value specified by the user and the CPU utilization value obtained from the network device. The user-specified CPU utilization threshold value is obtained by the **CPU Utilization Parameters** scan parameter group. The Device CPU Set Processor uses the **cpuProperties** file to set the CPU value of the device in the deviceCPUValue variable. This value is used as an input for the CPU Utilization Compare Processor to compare the CPU utilization value specified by the user and that of the device.

Scan Parameter Groups and the Network Integrity UI

A new scan parameter group, **CPU Utilization Parameters**, has been added in the NetworkIntegritySDK cartridge. The **CPU Utilization Parameters** scan parameter group is available for selection in the **Select Parameter Group** list under the **Scan Action Parameters** area.

In the Network Integrity UI Scan Configuration screen, selecting the **CPU Utilization Parameters** scan parameter group displays the **CPU Utilization %** field, which enables you to specify the CPU utilization threshold value between 1 to 99.

Reference Implementations

The Network Integrity Cisco Router and Switch SNMP cartridge demonstrates a complete reference implementation of discovery based on CPU utilization. The Cisco Router and Switch SNMP cartridge includes the **CPU Utilization Parameters** scan parameter group from the NetworkIntegritySDK cartridge. See *Network Integrity Cisco Router and Switch SNMP Cartridge Guide* for more information.

Implementing CPU Utilization-enabled Discovery

This section assumes that you already have valid, deployable cartridges that perform discovery, import, and discrepancy detection, to which you are adding CPU utilization-enabled discovery. You can implement CPU utilization-enabled discovery in a sealed cartridge solution.

Implementing CPU Utilization-enabled Discovery in a Sealed Cartridge Solution

You can implement CPU utilization-enabled discovery for any device that supports polling for CPU utilization.

See the Design Studio for Network Integrity Help for information about any of the steps in this section.

To implement CPU utilization-enabled discovery when working with a sealed cartridge:

1. Create a new cartridge.

2. Add the following dependencies to the new cartridge:

    ■ All sealed and unsealed cartridges being extended by the new cartridge

> **Note:** The new cartridge needs to extend a discovery action and a discrepancy detection action. These actions may belong to one or more cartridges. At least one of these cartridges is sealed.

- ora_ni_uim_device
- NetworkIntegritySDK
- Address Handler

3. Create a new discovery action in the new cartridge.

4. For the new discovery action:

   - Specify **IPAddressHandler** as the address handler.
   - Specify **Device** as the result category.
   - Add *discoveryAction* as a processor.

     Where *discoveryAction* is a discovery action from another cartridge.

   - Add Abstract CPU Utilization Discovery action as a processor.
   - Add the CPU Utilization Parameters scan parameter group.
   - Add or create any additional scan parameter groups required to configure the new discovery action.
   - Add new processors to obtain the CPU usage value of the device from the network. Set the deviceCPUValue variable in **cpuProperties** file for the Device CPU Set Processor as follows:

     ```
     request.getCpuProperties().setDeviceCPU(deviceCPUValue);
     ```

     This sets the value of the deviceCPUValue variable, which is used by the CPU Utilization Compare Processor (from NetworkIntegritySDK cartiridge) to compare the user-specified CPU utilization threshold value with CPU utilization value (set in deviceCPUValue variable) of the device.

5. Complete and customize the incremental discovery implementation for the new cartridge.

6. Save and close all files.

7. Build, deploy, and test your cartridge.

10

Working with the Network Integrity Web Service

This chapter provides information about the Oracle Communications Network Integrity Web service.

This chapter contains the following sections:

- About the Network Integrity Web Service
- Network Integrity Web Service Operations
- Network Integrity Web Service Special Function Operations
- Network Integrity Web Service Scenarios
- Network Integrity Web Service Samples

About the Network Integrity Web Service

The Network Integrity Web service enables Oracle Communications products and third party applications to interact with Network Integrity and reduces integration complexity by providing a standards-based interface. With the API, clients can externally manage Network Integrity through Web services.

At a high-level the Network Integrity Web service supports:

- Configuring all types of scans
- Running discovery and reconciliation scans
- Retrieving scan results including any found discrepancies
- Initiating corrective actions such as reconciling discrepancies in Inventory systems.

Most operations that can be done in the Network Integrity UI can be done through the Web service. One operation that is currently not possible is to create or update the Import System configured in the Network Integrity UI. This is a one-time setup that must be done in the Network Integrity UI and cannot be done through the Web service.

The Network Integrity Web service is standards based using JAX-WS over HTTP.

Security

The Network Integrity Web service uses the same security as the Network Integrity UI. Any user who is able to login into the Web UI can also use the Web service. This is assigned using NetworkIntegrityRole.

> **Note:** All Network Integrity Web service requests (Soap UI requests and automated Web service requests) must include a time stamp to access Network Integrity Web service.

Model Based

The Network Integrity Web service operates on the Network Integrity Model. Knowledge of the entities, attributes and relationships in the Network Integrity model is essential for using the Web service.

For Network Integrity entity, attribute and relationship names, see *Network Integrity Information Model Reference*.

For cartridge entity, parameter, and relationship names and descriptions, see your cartridge documentation.

Concurrency with UI and other Web Service Clients

Web service operations take immediate effect in the system and therefore there is scope for collisions with users working in the Network Integrity UI. If the Web service operation collides with an update that another user has done in the Network Integrity UI or another Web service client, then an error is returned to a client. For example, if a Web service client deletes a DisConfig (scan) while a user is editing the same scan in the Network Integrity UI, the user receives an error when that user attempt to save changes. If two clients (Web service client or Network Integrity UI user) are trying to update/delete the same entity, the last client to commit changes receives the error.

Listing of Network Integrity Web Service Operations

All Network Integrity Web service operations must include a time stamp to satisfy the Web service security policy. See "Security" for more information.

Table 10–1 describes the DisConfig operations. See *Network Integrity Information Model Reference* for more information on the DisConfig entity.

Table 10–1   DisConfig Operations

Operation	Description
createDisConfig	This operation creates a new scan in the system (DisConfig is equivalent to scan in the Network Integrity UI).
deleteDisConfig	This operation deletes a scan from the system. All results and discrepancies produced by this scan are deleted as well. A fault is returned if the delete fails. This delete operation returns a fault if the scan to be deleted has discrepancies in the Received or the Submitted state. Add **<v1:forceDelete>YES</v1:forceDelete>** to the delete request to force the scan to delete and bypass this particular fault.
findDisConfig	This operation finds scans in the system based on search criteria provided in the request. Full scan data is returned but client applications can limit the amount of data returned, or support paging, by providing a fromRange and toRange in the request. A fault with a faultstring is returned if the find fails.
getDisConfig	This operation gets the details about a scan. It requires the DisConfig entity ID to be passed in the request, and returns the full details of the scan including scan parameters, Scope Addresses, and Schedule information in the response, if found. If not found, a fault is the response.
updateDisConfig	This operation updates a scan in the system. All the values for the scan are required in the request. The client application should perform a **get** operation and update the required values for the update operation. A fault with a faultstring is returned if the update fails.

Table 10–2 describes the DisScanRun operations. See *Network Integrity Information Model Reference* for more information on the DisScanRun entity.

Table 10–2   DisScanRun Operations

Operation	Description
findDisScanRun	This operation finds scan results in the system based on search criteria provided in the request (DisScanRun is equivalent to scan results in the Network Integrity UI). Full scan result data is returned but client applications can limit the amount of data returned, or support paging, by providing a fromRange and toRange in the request. A fault with a faultstring is returned if the find fails.
deleteDisScanRun	This operation deletes scan results from the system. All results and discrepancies attached to the scan results are deleted as well. A fault with a faultstring is returned if the delete fails.
getDisScanRun	This operation gets all the details about an instance of scan results. The operation requires the Discrepancy entity id to be passed in the request, and returns the full details of the Discrepancy including references to the compare and reference Oracle Communications Information Model entities which the discrepancy was found on. If not found, a fault is the response.

Table 10–3 describes the DisBlackoutSchedule operations. See *Network Integrity Information Model Reference* for more information on the DisBlackoutSchedule entity.

Table 10–3   DisBlackoutSchedule Operations

Operation	Description
createDisBlackoutSchedule	This operation creates a new blackout schedule in the system. A recurrence rule, duration, and start time are required in the request. The blackout schedule can be assigned to scan configurations on creation, or they can be associated later with an update operation.
deleteDisBlackoutSchedule	This operation deletes a blackout schedule in the system. If any scans are associated with the blackout schedule then the associations are removed as well. A fault with a faultstring is returned if the delete fails.
getAllDisBlackoutSchedule	This operation returns the full details of all the blackout schedules in the system. An empty response is returned if no blackout schedules exist in the system.
getDisBlackoutSchedule	This operation requires the blackout schedule entity id to be passed in the request, and returns the full details of the blackout schedule in the response if found. If not found, a fault is the response.
updateDisBlackoutSchedule	This operation updates a blackout schedule in the system. All the values for the blackout schedule are required in the request, not just the values changing. A fault with a faultstring is returned if the update fails.

Table 10–4 describes the DisTag operations. See *Network Integrity Information Model Reference* for more information on the DisTag entity.

Table 10–4   DisTag Operations

Operation	Description
createDisTag	This operation creates a new tag, a name for the tag is required. The parent tag entity id can be provided in the creation or can be add after in an update request. A fault with a faultstring is returned if the delete fails.
deleteDisTag	This operation deletes the specified tag and all child tags. The entity id of the tag to be deleted is required. If any scans are associated with the tag then the associations are removed as well. A fault with a faultstring is returned if the delete fails.

Table 10–4   (Cont.) DisTag Operations

Operation	Description
getDisTag	This operation requires the tag entity id to be passed in the request, and returns the full details of the tag including all child tags in the response, if found. If not found, a fault is the response.
getAllRootDisTags	This operation returns the full details of all the tags configured in the system. The root tags returned also include the details of children tag entities. A fault with a faultstring is returned if an error occurs.
updateDisTag	This operation updates a tag, an entity id and name for the tag is required. All the values for the blackout schedule are required in the request, not just the values that are changing. Modifications to the hierarchy must be performed on the child tag, for example, to make a child tag a root tag call the update operation with no parent tags specified. A fault with a faultstring is returned if the update fails.

Table 10–5 describes the DisDiscrepancy operations. See *Network Integrity Information Model Reference* for more information on the DisDiscrepancy entity.

Table 10–5   DisDiscrepancy Operations

Operation	Description
findDisDiscrepancy	This operation finds Discrepancies in the system based on search criteria provided in the request. The search criteria available in the Web service operation is the same as the criteria available in the Network Integrity UI (DisScanRun is equivalent to scan results in the Network Integrity UI). Full Discrepancy data is returned but client applications can limit the amount of data returned, or support paging, by providing a fromRange and toRange in the request. A fault with a faultstring is returned if the find fails.
getDisDiscrepancy	This operation gets all the details about a Discrepancy. The operation requires the Discrepancy entity id to be passed in the request, and returns the full details of the Discrepancy including references to the compare and reference Information Model entities which the discrepancy was found on. If not found, a fault is the response.
updateDisDiscrepancy	This operation updates a discrepancy in the system. All the values for DisDiscrepancy are required in the request, not just the values changing. The valid values of status are: ■ DISCREPANCY_OPENED ■ DISCREPANCY_IGNORED ■ OPERATION_IDENTIFIED ■ OPERATION_SUBMITTED ■ OPERATION_RECEIVED ■ OPERATION_NOT_IMPLEMENTED ■ OPERATION_PROCESSED ■ OPERATION_FAILED The operation value is equivalent to resolution action value in the Network Integrity UI and the valid values are dependent on what discrepancy resolution are currently installed in the system. A fault with a faultstring is returned if the update fails.

Table 10–6 describes the DisPlugin operations. See *Network Integrity Information Model Reference* for more information on the DisPlugin entity.

Table 10–6   DisPlugin Operation

Operation	Description
getAllDisAssimilationPlugin	This operation returns details about all assimilation plugins deployed in the system (AssimilationPlugin is equivalent to assimilation/scan action in the Network Integrity UI).
getAllDisInventoryImportPlugin	This operation returns details about all import plugins deployed in the system (InventoryImportPlugin is equivalent to import/scan action in the Network Integrity UI).
getAllDisNetworkDiscoveryPlugin	This operation returns details about all discovery plugins deployed in the system (NetworkDiscoveryPlugin is equivalent to discovery/scan action in the Network Integrity UI).
getAllDisDiscrepancyDetectionPlugin	This operation returns details about all discrepancy detection plugins deployed in the system (Discrepancy Detection Plugin is equivalent to a discrepancy detection action)
getAllDisDiscrepancyResolutionPlugin	This operation returns details about all discrepancy resolution plugins deployed in the system (Discrepancy Resolution Plugin is equivalent to a discrepancy resolution action)
getDisAssimilationPlugin	This operation returns details about an assimilation plugin deployed in the system (AssimilationPlugin is equivalent to assimilation/scan action in the Network Integrity UI). The request requires an Assimilation Plugin entity id to be passed, and returns the full details of the Assimilation Plugin in the response if found. If not found, a fault is the response.
getDisInventoryImportPlugin	This operation returns details about an import plugin deployed in the system (InventoryImportPlugin is equivalent to import/scan action in the Network Integrity UI). The request requires an Import Plugin entity id to be passed, and returns the full details of the Import Plugin in the response if found. If not found, a fault is the response.
getDisNetworkDiscoveryPlugin	This operation returns details about a discovery plugin deployed in the system (NetworkDiscoveryPlugin is equivalent to discovery/scan action in the Network Integrity UI). The request requires a Discovery Plugin entity id to be passed, and returns the full details of the Discovery Plugin in the response if found. If not found, a fault is the response.
getDisDiscrepancyDetectionPlugin	This operation returns details about an discrepancy detection plugin deployed in the system (Discrepancy Detection Plugin is equivalent to a discrepancy detection action). The request requires an Discrepancy Detection Plugin entity id to be passed, and returns the full details of the Discrepancy Detection Plugin in the response if found. If not found, a fault is the response.
getDisDiscrepancyResolutionPlugin	This operation returns details about an discrepancy resolution plugin deployed in the system (Discrepancy Resolution Plugin is equivalent to a discrepancy resolution action). The request requires an Discrepancy Resolution Plugin entity id to be passed, and returns the full details of the Discrepancy Resolution Plugin in the response if found. If not found, a fault is the response.

Table 10–7 describes the DefaultDisInvetoryConfig operations.

Table 10–7   DefaultDisInventoryConfig

Operation	Description
getDefaultDisInventoryConfig	This operation returns the inventory system configured in the Network Integrity system. This is the inventory system configuration that is entered in the "Manage Import System" task of the Network Integrity UI. The Import System cannot be created or updated through the Web service; it must be done using the Network Integrity UI.

Table 10–8 describes the Special operations.

Table 10–8   Special Operations

Operation	Description
startScan	This operation starts a scan. The response returns a reference to the scan result entity so that the client application can monitor the progress of the scan. (DisScanRun is equivalent to the scan results in the Network Integrity UI). If the scan is already running or in the process of stopping then the startScan operation fails. If the scan could not be started, a fault with a reason is the response.
stopScan	This operation stops a scan that is running. The scan is set to a STOPPING state immediately and then transition to STOPPED when actually ended. If the scan is not currently running, this call is a no-op. If the scan could not be set to Stopping, a fault with a reason is the response.
submitDisDiscrepancy ResolutionOperations	This operation submits the list of discrepancies provided in the request for resolution processing. The status of the discrepancies must be 'OPERATION_IDENTIFIED' to submit them, otherwise a fault is returned. A fault with a faultstring is returned if the operation fails.
getLatestScanStatus	This operation returns the scan status for the most recent execution of a scan. This operation is more efficient than getDisScanRun and therefore is more appropriate for client applications that are monitoring the status of a scan (DisConfig is equivalent to scan in the Network Integrity UI). A fault with a faultstring is returned if the operation fails.

Table 10–9 describes the Information Model entity operations. Information Model entities are described in *Oracle Communications Information Model Reference* and *Network Integrity Information Model Reference*.

Table 10–9   Information Model Entity Operations

Operation	Description
getRootEntity	This operation gets all the details about a discovered, imported, or assimilated root Information Model entity. The root entity id for the request is obtained from either a getDisScanRun operation response or findDisScanRun operation response. The id is found in the 'rootEntityRefsRef' element in the result groups. Multiple ids can be passed in the request. The response entity can be many different types depending on what the cartridge persisted in the result group. An example root entity type is Physical Device or Logical Device, but other Information Model types are possible. If not found, a fault is the response.
getResultEntity	A generic operation to get any type of Information Model entity given an entityId and the entity type. Multiple entities can be retrieved in a single request. If not found, a fault is the response.
getSpecification	This operation gets all the details about specification deployed in the system. Most Information Model entities support specifications which is blueprint for what characteristics are supported, among other things. All the characteristics defined in this specification are returned. Specifications are deployed to the system when cartridges containing them are deployed. If not found, a fault is the response.
getLogicalDevice	This operation requires the LogicalDevice entity id to be passed in the request, and returns the full details of the LogicalDevice if found. If not found, a fault is the response.
getDeviceInterface	This operation requires the DeviceInterface entity id to be passed in the request, and returns the full details of the DeviceInterface if found. If not found, a fault is the response.
getMediaInterface	This operation requires the MediaInterface entity id to be passed in the request, and returns the full details of the MediaInterface if found. If not found, a fault is the response.

Table 10–9   (Cont.) Information Model Entity Operations

Operation	Description
getLogicalDeviceAccount	This operation requires the LogicalDeviceAccount entity id to be passed in the request, and returns the full details of the LogicalDeviceAccount if found. If not found, a fault is the response.
getPhysicalDevice	This operation requires the PhysicalDevice entity id to be passed in the request, and returns the full details of the PhysicalDevice if found. If not found, a fault is the response.
getEquipment	This operation requires the Equipment entity id to be passed in the request, and returns the full details of the Equipment if found. If not found, a fault is the response.
getEquipmentHolder	This operation requires the EquipmentHolder entity id to be passed in the request, and returns the full details of the EquipmentHolder if found. If not found, a fault is the response.
getPhysicalPort	This operation requires the PhysicalPort entity id to be passed in the request, and returns the full details of the PhysicalPort if found. If not found, a fault is the response.
getPhysicalConnector	This operation requires the PhysicalConnector entity id to be passed in the request, and returns the full details of the PhysicalConnector if found. If not found, a fault is the response.
getCustomObject	This operation requires the CustomObject entity id to be passed in the request, and returns the full details of the CustomObject if found. If not found, a fault is the response.
getCustomNetworkAddress	This operation requires the CustomNetworkAddress entity id to be passed in the request, and returns the full details of the CustomNetworkAddress if found. If not found, a fault is the response.
getTelephoneNumber	This operation requires the TelephoneNumber entity id to be passed in the request, and returns the full details of the TelephoneNumber if found. If not found, a fault is the response.
getInventoryGroup	This operation requires the InventoryGroup entity id to be passed in the request, and returns the full details of the InventoryGroup if found. If not found, a fault is the response.
getService	This operation requires the Service entity id to be passed in the request, and returns the full details of the Service if found. If not found, a fault is the response.
getNetwork	This operation requires the Network entity id to be passed in the request, and returns the full details of the Network if found. If not found, a fault is the response.
getNetworkNode	This operation requires the NetworkNode entity id to be passed in the request, and returns the full details of the NetworkNode if found. If not found, a fault is the response.
getNetworkEdge	This operation requires the NetworkEdge entity id to be passed in the request, and returns the full details of the NetworkEdge if found. If not found, a fault is the response.
getPipe	This operation requires the Pipe entity id to be passed in the request, and returns the full details of the Pipe if found. If not found, a fault is the response.
getPipeTerminationPoint	This operation requires the PipeTerminationPoint entity id to be passed in the request, and returns the full details of the PipeTerminationPoint if found. If not found, a fault is the response.
getPipeDirectionality	This operation requires the PipeDirectionality entity id to be passed in the request, and returns the full details of the PipeDirectionality if found. If not found, a fault is the response.
getTrailPath	This operation requires the TrailPath entity id to be passed in the request, and returns the full details of the TrailPath if found. If not found, a fault is the response.

Table 10–9   (Cont.) Information Model Entity Operations

Operation	Description
getGeographicPlace	This operation requires the GeographicPlace entity id to be passed in the request, and returns the full details of the GeographicPlace if found. If not found, a fault is the response.
getGeographicAddress	This operation requires the GeographicAddress entity id to be passed in the request, and returns the full details of the GeographicAddress if found. If not found, a fault is the response.
getGeographicAddressRange	This operation requires the GeographicAddressRange entity id to be passed in the request, and returns the full details of the GeographicAddressRange if found. If not found, a fault is the response.
getGeographicLocation	This operation requires the GeographicLocation entity id to be passed in the request, and returns the full details of the GeographicLocation if found. If not found, a fault is the response.
getGeographicSite	This operation requires the GeographicSite entity id to be passed in the request, and returns the full details of the GeographicSite if found. If not found, a fault is the response.
getNetworkNodeRole	This operation requires the NetworkNodeRole entity id to be passed in the request, and returns the full details of the NetworkNodeRole if found. If not found, a fault is the response.
getPhysicalConnectorRole	This operation requires the PhysicalConnectorRole entity id to be passed in the request, and returns the full details of the PhysicalConnectorRole if found. If not found, a fault is the response.
getPipeRole	This operation requires the PipeRole entity id to be passed in the request, and returns the full details of the PipeRole if found. If not found, a fault is the response.
getPhysicalPortRole	This operation requires the PhysicalPortRole entity id to be passed in the request, and returns the full details of the PhysicalPortRole if found. If not found, a fault is the response.
getDeviceInterfaceRole	This operation requires the DeviceInterfaceRole entity id to be passed in the request, and returns the full details of the DeviceInterfaceRole if found. If not found, a fault is the response.
getLogicalDeviceRole	This operation requires the LogicalDeviceRole entity id to be passed in the request, and returns the full details of the LogicalDeviceRole if found. If not found, a fault is the response.
getCustomObjectRole	This operation requires the CustomObjectRole entity id to be passed in the request, and returns the full details of the CustomObjectRole if found. If not found, a fault is the response.
getPhysicalDeviceRole	This operation requires the PhysicalDeviceRole entity id to be passed in the request, and returns the full details of the PhysicalDeviceRole if found. If not found, a fault is the response.
getEquipmentRole	This operation requires the EquipmentRole entity id to be passed in the request, and returns the full details of the EquipmentRole if found. If not found, a fault is the response.
getNetworkEdgeRole	This operation requires the NetworkEdgeRole entity id to be passed in the request, and returns the full details of the DeviNetworkEdgeRole ceInterface if found. If not found, a fault is the response.
getPlaceRole	This operation requires the PlaceRole entity id to be passed in the request, and returns the full details of the PlaceRole if found. If not found, a fault is the response.
getNetworkRole	This operation requires the NetworkRole entity id to be passed in the request, and returns the full details of the NetworkRole if found. If not found, a fault is the response.

Network Integrity Web Service Operations

Most of the operations defined in the Network Integrity Web service follow the naming pattern of:

- Create
- Get
- Get All
- Delete
- Update
- Find

However, a few of the Web service operations do not follow this pattern. See "Network Integrity Web Service Special Function Operations" for more information.

Create

Each **create** operation inserts a new entity into the system. For example, the createDisBlackoutSchedule operation creates a new blackout schedule in the system.

If successful, the changes are immediately available in the system and can be viewed in the Network Integrity UI.

The request for each **create** operation is named **create<EntityType>Request**. The request contains the full details of the new entity to be created. Multiple entities cannot be created in a single request, only a single entity is supported.

The following fields should not be supplied in the create request as they are populated automatically by the system.

- entityId
- entityVersion
- lastModifiedDate
- lastModifiedUser
- createdDate
- createdUser

The response from each **create** operation is named **create<EntityType>Response** and contains the entityId of the created entity if the operation was successful. The entityId returned is used in subsequent **get** and **delete** operations.

If a **create** operation fails, the response contains a fault with a faultCode, faultString, and extra CrudFault details.

Example 10–1   Create Request

```
<v1:createDisTagRequest>
  <v1:disTag>
    <v13:name>Sample Tag</v13:name>
    <v13:description>Created through Web Service</v13:description>
  </v1:disTag>
</v1:createDisTagRequest>
```

Example 10–2   Create Response

```
<ns118:createDisTagResponse>
```

```
        <ns118:disTagRef>
           <ns2:entityId>9584</ns2:entityId>
        </ns118:disTagRef>
    </ns118:createDisTagResponse>
```

Example 10–3   Create Failure (a name was not specified for the tag)

```
<ns2:Fault>
   <faultcode>ns2:Server</faultcode>
   <faultstring>ILLEGAL_NAME</faultstring>
   <detail>
      <ns158:crudFault>
         <ns152:rootStackTrace/>
      </ns158:crudFault>
   </detail>
</ns2:Fault>
```

Entity Type Support

Each **create** operation supports the following entity types:

- DisBlackoutSchedule

- DisTag

- DisConfig

Get

Each **get** operation retrieves an entity from the system. The **get** request requires a unique entity id and the entity details are returned in the response. For example, the getDisBlackoutSchedule operation returns all the details of a specific blackout schedule in the system.

The request for each **get** operation is named **get<EntityType>Request**. The request contains a single entityId of the entity to be retrieved. Only one entityId can be specified in the request, multiples are ignored. The exception to this is the getRootEntity and getResultEntity operations; these operations accept multiple entity id values.

If the entityId provided is not found in the system a fault is returned, not an empty response.

The response from each **get** operation is named **get<EntityType>Response** and contains the details of the entity retrieved from the system.

If a **get** operation fails, the response contains a fault with a faultCode, faultString, and extra CrudFault details.

Example 10–4   Get Request

```
<v1:getDisTagRequest>
   <v1:disTagRef>
      <v11:entityId>9586</v11:entityId>
   </v1:disTagRef>
</v1:getDisTagRequest>
```

Example 10–5   Get Response

```
<ns118:getDisTagResponse>
 <ns118:disTag>
```

```
    <ns2:entityId>9586</ns2:entityId>
    <ns2:entityVersion>1</ns2:entityVersion>
    <ns12:parentRef>
        <ns2:entityId>9584</ns2:entityId>
    </ns12:parentRef>
    <ns12:name>Sample Child Tag</ns12:name>
    <ns12:description>Child Created through WS</ns12:description>
 </ns118:disTag>
</ns118:getDisTagResponse>
```

Example 10–6   Get Failure (entity id was not found)

```
<ns2:Fault>
   <faultcode>ns2:Server</faultcode>
   <faultstring>Cannot find Tag with entity Id 9586</faultstring>
   <detail>
      <ns158:crudFault>
         <ns151:rootStackTrace/>
      </ns158:crudFault>
   </detail>
</ns2:Fault>
```

Entity Type Support

Each **get** operation supports the following entity types:

- DisBlackoutSchedule

- DisTag

- DisConfig

- DisDiscrepancy

- DisInventoryImportPlugin

- DisNetworkDiscoveryPlugin

- DisAssimilationPlugin

- DisDiscrepancyResolutionPlugin

- DisDiscrepancyDetectionPlugin

- DisScanRun

- RootEntity

- ResultEntity

- Specification

- DefaultDisInventoryConfig

- DeviceInterface

- PhysicalDevice

- EquipmentHolder

- MediaInterface

- Equipment

- LogicalDevice

- PhysicalPort

- PhysicalConnector

■ CustomObject

Get All

Each **get all** operation retrieves all entities of a certain type from the system. For example, the getAllDisBlackoutSchedule operation returns all the details of all the blackout schedules currently in the system.

These operations are only available for entities that would not typically have many entries in the system and that do not support a **find** operation.

The request for each **get all** operation is named **getAll<EntityType>Request**. The request does not support any request parameters.

The response from each **get all** operation is named **getAll<EntityType>Response** and contains the details of all the entities retrieved from the system.

If a **get all** operation fails, the response contains a fault with a faultCode, faultString, and extra CrudFault details. Since the **get all** operations do not take any input parameters, they should only fail due to environment or authentication issues.

Example 10–7   Get All Request

```
<v1:getAllRootDisTagsRequest/>
```

Example 10–8   Get All Response

```
<ns118:getAllRootDisTagsResponse>
  <ns118:rootDisTags>
     <ns2:entityId>9584</ns2:entityId>
     <ns2:entityVersion>3</ns2:entityVersion>
     …etc
  </ns118:rootDisTags>
  <ns118:rootDisTags>
     <ns2:entityId>9585</ns2:entityId>
     <ns2:entityVersion>3</ns2:entityVersion>
     …etc
  </ns118:rootDisTags>
</ns118:getAllRootDisTagsResponse>
```

Entity Type Support

Each **get all** operation supports the following entity types:

■ DisBlackoutSchedule

■ RootDisTag

■ DisInventoryImportPlugin

■ DisNetworkDiscoveryPlugin

■ DisAssimilationPlugin

■ DisDiscrepancyResolutionPlugin

■ DisDiscrepancyDetectionPlugin

Delete

Each **delete** operation removes an entity from the system. For example, the deleteDisBlackoutSchedule operation removes a particular blackout schedule from the system.

If successful, the result of a **delete** operation is immediately viewable in the Network Integrity UI.

The request for each **delete** operation is named **delete<EntityType>Request**. The request contains a single entityId of the entity to be deleted. Only one entityId can be specified in the request, multiples are ignored.

If the entityId provided is not found in the system, or if the entity cannot be deleted, a fault is returned.

> **Note:** The deleteDisConfig operation has an additional optional parameter you can enter in the delete request to force a scan to be deleted, even if it has associated discrepancies in the Running or Submitted state. See Table 10–1, " DisConfig Operations" for more information.

The response from each **delete** operation is named **delete<EntityType>Response** and contains the entityId of the entity deleted, which matches the id in the request.

If a **delete** operation fails, the response contains a fault with a faultCode, faultString, and extra CrudFault details.

Example 10–9   Delete Request

```
<v1:deleteDisTagRequest>
    <v1:disTagRef>
        <v11:entityId>9579</v11:entityId>
    </v1:disTagRef>
</v1:deleteDisTagRequest>
```

Example 10–10   Delete Response

```
<ns118:deleteDisTagResponse>
    <ns118:disTagRef>
        <ns2:entityId>9579</ns2:entityId>
    </ns118:disTagRef>
</ns118:deleteDisTagResponse>
```

Example 10–11   Delete Failure (entity id was not found)

```
<ns2:Fault>
   <faultcode>ns2:Server</faultcode>
   <faultstring>Cannot find Tag with Entity Id9579</faultstring>
   <detail>
      <ns158:crudFault>
         <ns151:rootStackTrace/>
      </ns158:crudFault>
   </detail>
</ns2:Fault>
```

Entity Type Support

Each **delete** operation supports the following entity types:

- DisBlackoutSchedule

- DisTag

- DisConfig

- DisScanRun

Update

Each **update** operation modifies an existing entity in the system. For example, the updateDisBlackoutSchedule operation updates a blackout schedule currently in the system.

If successful, the update is immediately available in the system and can be viewed in the Network Integrity UI.

The request for each **update** operation is named **update<EntityType>Request**. The request must contain the full details of the new entity to be created, not just the fields that have changed. Multiple entities cannot be updated in a single request, only a single entity is supported. Unlike the create operation, the entityId must be supplied in the **update** operation to uniquely identity which entity to modify.

The entity version passed in the request must match the version that is held on the server. The entity version is incremented by the system every time the entity is modified. The entity version ensures that the entity has not been changed by some other user between when the entity was last retrieved and when updated. If the entity has been changed by some other user a fault is returned as follows: Entity Version Mismatch: **Input Version=1::Latest Version=2**

Because the full details of the entity are required in the **update** request, the recommended steps are to do a **get**, **get all**, or **find** operation to get the details of the entity, and then copy these details into the update request, and modify the desired fields.

The following fields should not be supplied in the update request as they are populated automatically by the system or are not currently used.

- lastModifiedDate

- lastModifiedUser

- createdDate

- createdUser

Each response from the **update** operation is named **update<EntityType>Response** and contains the entityId of the updated entity if the operation was successful.

If the **update** operation fails, the response contains a fault with a faultCode, faultString, and extra CrudFault details.

Example 10–12   Update Request

```
<v1:updateDisTagRequest>
  <v1:disTag>
    <v11:entityId>9586</v11:entityId>
    <v11:entityVersion>1</v11:entityVersion>
    <v12:parentRef>
       <v2:entityId>9584</v2:entityId>
    </v12:parentRef>
    <v11:name>Sample Child Tag</v11:name>
    <v11:description>Modified through WS</v11:description>
```

```
    </v1:disTag>
</v1:updateDisTagRequest>
```

Example 10–13   Update Response

```
<ns118:updateDisTagResponse>
    <ns118:disTagRef>
        <ns2:entityId>9586</ns2:entityId>
    </ns118:disTagRef>
</ns118:updateDisTagResponse>
```

Example 10–14   Update Failure (wrong entity version supplied)

```
<ns2:Fault>
    <faultcode>ns2:Server</faultcode>
    <faultstring>Entity Version Mismatch: Input Version=2::Latest
Version=3</faultstring>
    <detail>
        <ns158:crudFault>
            <ns151:rootStackTrace/>
        </ns158:crudFault>
    </detail>
</ns2:Fault>
```

Entity Type Support

Each **update** operation supports the following entity types:

- DisBlackoutSchedule
- DisTag
- DisConfig
- DisDiscrepancy

Find

Each **find** operation retrieves a list of entities that match filter search criteria. For example, the findDisConfig operation retrieves a list DisConfig entities currently in the system that match a given set of search criteria.

Each **find** operation is equivalent in capability to the Search screens in the Network Integrity UI.

The request for each **find** operation is named **find<EntityType>Request**. The find request can contain:

- From and To Ranges
- Sorting Fields (Ascending and Descending)
- Attribute Criteria
- Extended Attribute Criteria
- Criteria Operator (Equals, Contains, etc.)
- Conjunction Criteria (AND/OR)

Entity Type Support

Each **find** operation supports the following entity types:

- DisConfig

- DisScanRun

- DisDiscrepancy

From and To Range

The fromRange and toRange are used to limit the number of rows returned to a client. These fields support paging in UIs through the Web service. It is also useful to improve performance and memory usage by retrieving many rows in smaller, more manageable chunks.

If the fromRange is not provided the default value is 0 which means the find returns the first row on. If the toRange is not provided in the request then the find operation is unbounded and returns all rows to the end.

```
<v1:findDisConfigRequest>
   <v1:disConfigSearchCriteria>
      <fromRange>0</fromRange>
      <toRange>20</toRange>
      <descending>name</descending>
      <disConfigConjunctionCriteriaItem>
         <nameAttributeCriteria>
            <value>Cisco</value>
            <operator> EQUALS</operator>
         </nameAttributeCriteria>
         <conjunction>AND</conjunction>
      </disConfigConjunctionCriteriaItem>
   </v1:disConfigSearchCriteria>
</v1:findDisConfigRequest>
```

Ascending and Descending

The ascending and descending fields control how the entity results are sorted in the response. The ascending and descending fields hold the name of the attribute to be sorted on. Multiple ascending and descending fields can be specified to add more than one level of sorting. If both an ascending and descending sort field are not provided in the request then the order of the entities returned is not sorted, and returned in the order they are persisted.

```
<v1:findDisConfigRequest>
   <v1:disConfigSearchCriteria>
      <fromRange>0</fromRange>
      <toRange>20</toRange>
      <descending>name</descending>
      <disConfigConjunctionCriteriaItem>
         <nameAttributeCriteria>
            <value>Cisco</value>
            <operator>EQUALS</operator>
         </nameAttributeCriteria>
         <conjunction>AND</conjunction>
      </disConfigConjunctionCriteriaItem>
   </v1:disConfigSearchCriteria>
</v1:findDisConfigRequest>
```

Attribute Criteria

The attribute criteria specifies the field and value to match when performing the find operation. In addition, an operator needs to be specified in the attribute criteria to determine how the match is done (for example, **EQUALS**, **NOT_EQUALS**, etc.).

Zero or more attribute criteria are contained within an entity's ConjunctionCriteriaItem.

The <EntityType>ConjunctionCriteriaItem element defines a list of valid <attributeName>AttributeCriteria child elements. For example, the disConfigConjunctionCriteriaItem has an attributeCriteria for every attribute that is searchable, namely the nameAttributeCriteria, descriptionAttributeCriteria, enabledAttributeCriteria, etc.

For each attribute criteria the value to match and the operator to use to perform the match. The operators that are valid depend on the attribute type. For a list of valid operators, see the operator section below.

You can use wildcards in the value field for attributes that are text types. The supported wildcard characters are "*'", "%", and "_". "*" and "%" both represent a match of zero or more characters. "_"represents a match of any single character. Wildcard characters can be escaped with a backslash "\". To insert a backslash in the query, insert two backslashes "\\".

```
<v1:findDisConfigRequest>
   <v1:disConfigSearchCriteria>
      <fromRange>0</fromRange>
      <toRange>20</toRange>
      <descending>name</descending>
      <disConfigConjunctionCriteriaItem>
         <nameAttributeCriteria>
            <value>Cisco</value>
            <operator>EQUALS</operator>
         </nameAttributeCriteria>
         <conjunction>AND</conjunction>
      </disConfigConjunctionCriteriaItem>
   </v1:disConfigSearchCriteria>
</v1:findDisConfigRequest>
```

Multiple Attribute Criteria

Multiple criteria for the same attribute can be passed in a single find operation. In the example below the find request is looking for scans that start with the name Cisco or Juniper. It is necessary to specify the 'OR' conjunction in this scenario or no rows is returned.

```
<v1:findDisConfigRequest>
   <v1:disConfigSearchCriteria>
      <fromRange>0</fromRange>
      <toRange>20</toRange>
      <descending>name</descending>
      <disConfigConjunctionCriteriaItem>
         <nameAttributeCriteria>
            <value>Cisco*</value>
            <operator>EQUALS</operator>
         </nameAttributeCriteria>
         <nameAttributeCriteria>
            <value>Juniper*</value>
            <operator> EQUALS </operator>
         </nameAttributeCriteria>
         <conjunction>OR</conjunction>
      </disConfigConjunctionCriteriaItem>
   </v1:disConfigSearchCriteria>
</v1:findDisConfigRequest>
```

Extended Attribute Criteria

Extended Attribute Criteria allow the client application to find entities based on the attribute values on related entities. For example, to find all scans with a certain Scope Address would not be possible without extended criteria because the scope address is not defined on the DisConfig entity. Multiple criteria for the same attribute can be passed in a single find operation.

In the example below, the scope relationship on the DisConfig entity is followed, and then the addresses relationship if followed on the DisScope, to specify the addresses to match against. This search finds DisConfig entities that have either the address 10.156.68.136 or 10.156.68.140 in the scope. The schemas for the Web service define all the relationships and attributes that can be specified in the find operation.

```
<v1:findDisConfigRequest>
   <v1:disConfigSearchCriteria>
      <fromRange>0</fromRange>
      <toRange>20</toRange>
      <disConfigConjunctionCriteriaItem>
         <disConfigExtendedCriteriaItem>
            <scope>
               <disScopeConjunctionCriteriaItem>
                  <disScopeExtendedCriteriaItem>
                     <addresses>
                        <disAddressConjunctionCriteriaItem>
                           <addressAttributeCriteria>
                              <value>10.156.68.136</value>
                              <operator>EQUALS</operator>
                           </addressAttributeCriteria>
                           <addressAttributeCriteria>
                              <value>10.156.68.140</value>
                              <operator>EQUALS</operator>
                           </addressAttributeCriteria>
                           <conjunction>OR</conjunction>
                        </disAddressConjunctionCriteriaItem>
                     </addresses>
                  </disScopeExtendedCriteriaItem>
               </disScopeConjunctionCriteriaItem>
            </scope>
         </disConfigExtendedCriteriaItem>
         <conjunction>OR</conjunction>
      </disConfigConjunctionCriteriaItem>
   </v1:disConfigSearchCriteria>
</v1:findDisConfigRequest>
```

Criteria Operators

The following are the allowed search operators for each entity and attribute. If the Web service clients sends the wrong operator for a search criteria the Web service search request fails and the client gets a message, which shows the allowed operators for that search criteria.

DisConfig

Table 10–10 shows the allowed search operators for DisConfig attributes.

Table 10–10   Allowed Search Operators for DisConfig Attributes

Attribute Name	EQUALS	NOT_EQUAL	STARTS_WITH	FALSE	TRUE
Tag	Y	Y	Y	N/A	N/A
Name	Y	Y	N/A	N/A	N/A
ScanAction	Y	Y	N/A	N/A	N/A
ScanType	Y	Y	N/A	N/A	N/A
Description	Y	Y	N/A	N/A	N/A
Source	Y	Y	N/A	N/A	N/A
NetworkAddress	Y	N/A	N/A	N/A	N/A
Enabled	N/A	N/A	N/A	Y	Y
Run Reconciliation	N/A	N/A	N/A	Y	Y

DisScanRun

Table 10–11, Table 10–12, and Table 10–13 show the allowed search operators for DisScanRun.

Table 10–11   Allowed Search Operators for DisScanRun Attributes

Attribute Name	EQUALS	NOT_EQUAL	STARTS_WITH
Tag	Y	Y	Y
Name	Y	Y	N/A
Status	Y	Y	N/A
ScanType	Y	Y	N/A
Source	Y	Y	N/A
ScanAction	Y	Y	N/A

Table 10–12   Allowed Search Operators for DisScanRun Attributes

Attribute Name	BEFORE	AFTER	ON_OR_ AFTER	ON_OR_ BEFORE	BETWEEN	NOT_ BETWEEN
ScanStartTime	Y	Y	Y	Y	Y	Y
ScanEndTime	Y	Y	Y	Y	Y	Y
DiscrepancyDetection StartTime	Y	Y	Y	Y	Y	Y
DiscrepancyDetection EndTime	Y	Y	Y	Y	Y	Y

Table 10–13   Allowed Search Operators for DisScanRun Attributes

Attribute Name	EQUALS	NOT_ EQUAL	GREATER_ THAN	LESS_ THAN	BETWEEN	NOT_ BETWEEN
MinorDiscrepancies	Y	Y	Y	Y	Y	Y
MajorDiscrepancies	Y	Y	Y	Y	Y	Y
CriticalDiscrepancies	Y	Y	Y	Y	Y	Y
WarningDiscrepancies	Y	Y	Y	Y	Y	Y

DisDiscrepancy
Table 10–14 and Table 10–15 show the allowed search operators for DisDiscrepancy.

Table 10–14   Allowed Search Operators for DisDiscrepancy Attributes

Attribute Name	EQUALS	NOT_ EQUAL	STARTS_ WITH	IS_BLANK	IS_NOT_ BLANK
Tag	Y	Y	Y	N/A	N/A
Severity	Y	Y	N/A	N/A	N/A
Status	Y	Y	N/A	N/A	N/A
ResolutionAction	Y	Y	N/A	Y	Y
Owner	Y	Y	N/A	Y	Y
Priority	Y	Y	N/A	Y	Y
EntityName	Y	Y	N/A	N/A	N/A
ScanResultDetailName	Y	Y	N/A	N/A	N/A
ScanType	Y	Y	N/A	N/A	N/A
EntityName	Y	Y	N/A	N/A	N/A
ScanResultDetailName	Y	Y	N/A	N/A	N/A
ScanName	Y	Y	N/A	N/A	N/A
EntityType	Y	Y	N/A	N/A	N/A
CorrectedBy	Y	Y	N/A	N/A	N/A
SubmittedBy	Y	Y	N/A	N/A	N/A
ParentEntityNamw	Y	Y	N/A	N/A	N/A
ParentEntityType	Y	Y	N/A	N/A	N/A
Discovery/ImportValue	Y	Y	N/A	N/A	N/A
Discovery/ImportSource	Y	Y	N/A	N/A	N/A
ScanResultDetailCategory	Y	Y	N/A	N/A	N/A
Type	Y	Y	N/A	N/A	N/A
ScanType	Y	Y	N/A	N/A	N/A

Table 10–15   Allowed Search Operators for DisDiscrepancy Attributes

Attribute Name	BEFORE	AFTER	ON_OR_ AFTER	ON_OR_ BEFORE	BETWEEN	NOT_ BETWEEN
ScanStartTime	Y	Y	Y	Y	Y	Y
ScanEndTime	Y	Y	Y	Y	Y	Y
DiscrepancyDetection StartTime	Y	Y	Y	Y	Y	Y
DiscrepancyDetection EndTime	Y	Y	Y	Y	Y	Y
SubmittedTime	Y	Y	Y	Y	Y	Y
LastStatusChangeTime	Y	Y	Y	Y	Y	Y

Between/Not Between Operator

When specifying the **BETWEEN** and **NO_BETWEEN** operators, two attribute criteria must be supplied or a fault is returned. The error message returned is **Incorrect number of values or incorrect format specified for attribute criteria: numberWarning.**

The following example searches for scan results that found between 10 and 100 discrepancy warnings.

```
<v1:findDisScanRunRequest>
   <v1:disScanRunSearchCriteria>
      <v11:fromRange>0</v11:fromRange>
      <v11:toRange>20</v11:toRange>
      <v11:disScanRunConjunctionCriteriaItem>
         <v12:disScanRunExtendedCriteriaItem>
            <v14:counts>
               <v119:disDiscrepancyCountsConjunctionCriteriaItem>
                  <v120:warningAttributeCriteria>
                     <v121:value>10</v121:value>
                     <v121:value>100</v121:value>
                     <v121:operator>BETWEEN</v121:operator>
                  </v120:warningAttributeCriteria>
               </v119:disDiscrepancyCountsConjunctionCriteriaItem>
            </v14:counts>
         </v12:disScanRunExtendedCriteriaItem>
         <v12:conjunction>AND</v12:conjunction>
      </v11:disScanRunConjunctionCriteriaItem>
   </v1:disScanRunSearchCriteria>
</v1:findDisScanRunRequest>
```

Data Criteria

Date fields must be in the format mm/dd/yyyy mm:dd:ss AM/PM. The server time is always used for dates in Network Integrity. The following example searches for scan runs that started after the August 11th, 2010 10:00 am. Because the **AFTER** operator is used, scans that match this start time exactly are not included in the response. If operator **ON_OR_AFTER** was used then exact match start time scans are included in the response.

```
<v1:findDisScanRunRequest>
   <v1:disScanRunSearchCriteria>
      <v11:fromRange>0</v11:fromRange>
      <v11:toRange>20</v11:toRange>
      <v11:disScanRunConjunctionCriteriaItem>
         <v12:discoveryBeginTimeAttributeCriteria>
            <v13:value>08/11/2010 10:00:00 AM</v13:value>
            <v13:operator>AFTER</v13:operator>
         </v12:discoveryBeginTimeAttributeCriteria>
         <v12:conjunction>AND</v12:conjunction>
      </v11:disScanRunConjunctionCriteriaItem>
   </v1:disScanRunSearchCriteria>
</v1:findDisScanRunRequest>
```

Conjunction Criteria

The conjunction must be either **AND** or **OR**. Only the top level conjunction is used, conjunctions on lower level elements are ignored.

```
<v1:findDisConfigRequest>
```

```
                    <v1:disConfigSearchCriteria>
                        <fromRange>0</fromRange>
                        <toRange>20</toRange>
                        <descending>name</descending>
                        <disConfigConjunctionCriteriaItem>
                            <nameAttributeCriteria>
                                <value>Cisco*</value>
                                <operator>EQUALS</operator>
                            </nameAttributeCriteria>
                            <nameAttributeCriteria>
                                <value>Juniper*</value>
                                <operator>EQUALS</operator>
                            </nameAttributeCriteria>
                            <conjunction>OR</conjunction>
                        </disConfigConjunctionCriteriaItem>
                    </v1:disConfigSearchCriteria>
                </v1:findDisConfigRequest>
```

The conjunction appears at many levels in the find hierarchy. The conjunction at lower levels controls how the criteria at lower levels are evaluated logically.

In the following example the inner conjunction is OR because this request is designed to find any ScanRun that has discrepancy, regardless of severity. Notice the outer conjunction that has the value AND, this has no effect on the extended attribute criteria.

To change this find so it only finds scans that have a discrepancy of every severity, the inner conjunction on the disDiscrepancyCountsConjunctionCriteriaItem element would be changed to AND.

```
<v1:findDisScanRunRequest>
    <v1:disScanRunSearchCriteria>
        <v11:disScanRunConjunctionCriteriaItem>
            <v12:disScanRunExtendedCriteriaItem>
                <v14:counts>
                    <v119:disDiscrepancyCountsConjunctionCriteriaItem>
                        <v120:criticalAttributeCriteria>
                            <v121:value>0</v121:value>
                            <v121:operator>GREATER_THAN</v121:operator>
                        </v120:criticalAttributeCriteria>
                        <v120:majorAttributeCriteria>
                            <v121:value>0</v121:value>
                            <v121:operator>GREATER_THAN</v121:operator>
                        </v120:majorAttributeCriteria>
                        <v120:minorAttributeCriteria>
                            <v121:value>0</v121:value>
                            <v121:operator>GREATER_THAN</v121:operator>
                        </v120:minorAttributeCriteria>
                        <v120:warningAttributeCriteria>
                            <v121:value>0</v121:value>
                            <v121:operator>GREATER_THAN</v121:operator>
                        </v120:warningAttributeCriteria>
                        <v120:conjunction>OR</v120:conjunction>
                    </v119:disDiscrepancyCountsConjunctionCriteriaItem>
                </v14:counts>
            </v12:disScanRunExtendedCriteriaItem>
            <v12:conjunction>AND</v12:conjunction>
        </v11:disScanRunConjunctionCriteriaItem>
    </v1:disScanRunSearchCriteria>
```

Find Response

Each **find** response contains all the details of the entities that matched the attribute criteria. The response only contains the number of entities defined by the from an to range. Subsequent **find** operations may be called to get all the entities depending on the number of rows matching the search criteria and the from and to range specified.

```
<ns118:findDisConfigResponse>
   <ns118:disConfigs>
       <ns2:entityId>9612</ns2:entityId>
       <ns2:entityVersion>1</ns2:entityVersion>
       <ns4:tagsRef>
          <ns2:entityId>9584</ns2:entityId>
       </ns4:tagsRef>
       <ns4:tagsRef>
          <ns2:entityId>9586</ns2:entityId>
       </ns4:tagsRef>
       <ns7:parameterGroups>
          <ns2:entityId>9606</ns2:entityId>
       .
       .
       .
       <ns7:enabled>YES</ns7:enabled>
       <ns7:dataSource>TRUE</ns7:dataSource>
       <ns7:startScanReady>true</ns7:startScanReady>
   </ns118:disConfigs>
</ns118:findDisConfigResponse>
```

Network Integrity Web Service Special Function Operations

There are a few Network Integrity Web service operations that do not follow the standard pattern and are designed for a special purpose.

The Network Integrity Web service special function operations are:

- Start Scan

- Stop Scan

- Get Latest Scan Status

- Submit Discrepancies For Resolution Processing

Start Scan

The **startScan** operation starts a scan for a given DisConfig entityId. This operation is identical to the start scan operation in the Network Integrity UI. The request expects a DisConfig entityId and the response contains the entityId of the DisScanRun that was created for the scan.

Example 10–15   Request:

```
<v1:startScanRequest>
   <v1:disConfigRef>
      <v11:entityId>9612</v11:entityId>
   </v1:disConfigRef>
</v1:startScanRequest>
```

Example 10–16   Response:

```
<ns118:startScanResponse>
   <ns118:disScanRunRef>
```

```
      <ns2:entityId>14721</ns2:entityId>
   </ns118:disScanRunRef>
</ns118:startScanResponse>
```

Stop Scan

The **stopScan** operation stops a scan for a given DisConfig entityId. This operation is identical to the stop scan operation in the Network Integrity UI. The request expects a DisConfig entityId and the response contains the entityId of the DisScanRun that was created for the scan.

Example 10–17   Request:

```
<v1:stopScanRequest>
   <v1:disConfigRef>
      <v11:entityId>9612</v11:entityId>
   </v1:disConfigRef>
</v1:stopScanRequest>
```

Example 10–18   Response:

```
<ns118:stopScanResponse>
   <ns118:disScanRunRef>
      <ns2:entityId>13846</ns2:entityId>
   </ns118:disScanRunRef>
</ns118:stopScanResponse>
```

Get Latest Scan Status

The **getLatestScanStatus** returns the status of the latest run of a scan. The operation is equivalent to the information displayed in the Status section of the Manage Scans page of the Network Integrity UI. In addition to the status of the scan the operation returns information about the number of addresses being discovered, the number of discrepancies found, and the start time and duration of the scan.

This method is more efficient to call to monitor the running of a scan rather than call findDisScanRun many times.

Example 10–19   Request:

```
<v1:getLatestScanStatusRequest>
   <v1:disConfigRef>
      <v11:entityId>9612</v11:entityId>
   </v1:disConfigRef>
</v1:getLatestScanStatusRequest
```

Example 10–20   Response (Running Scan)

```
<ns118:getLatestScanStatusResponse>
   <ns118:scanStatus>
      <ns120:discrepancySeverityCounts>
         <ns2:entityId>0</ns2:entityId>
         <ns2:entityVersion>0</ns2:entityVersion>
         <ns56:numberWarning>0</ns56:numberWarning>
         <ns56:numberMinor>0</ns56:numberMinor>
         <ns56:numberMajor>0</ns56:numberMajor>
         <ns56:numberCritical>0</ns56:numberCritical>
      </ns120:discrepancySeverityCounts>
      <ns120:discoveryWorkCounts>
         <ns121:totalNoOfWorkItems>2</ns121:totalNoOfWorkItems>
```

```
        <ns121:noOfCompletedWorkItems>0</ns121:noOfCompletedWorkItems>
        <ns121:noOfFailedWorkItems>0</ns121:noOfFailedWorkItems>
        <ns121:noOfInProgressWorkItems>2</ns121:noOfInProgressWorkItems>
        <ns121:startTime>07/16/2010 11:17:05</ns121:startTime>
        <ns121:duration/>
    </ns120:discoveryWorkCounts>
    <ns120:discrepancyWorkCounts>
        <ns121:totalNoOfWorkItems>0</ns121:totalNoOfWorkItems>
        <ns121:noOfCompletedWorkItems>0</ns121:noOfCompletedWorkItems>
        <ns121:noOfFailedWorkItems>0</ns121:noOfFailedWorkItems>
        <ns121:noOfInProgressWorkItems>0</ns121:noOfInProgressWorkItems>
        <ns121:duration/>
    </ns120:discrepancyWorkCounts>
    <ns120:jobStateString>Running</ns120:jobStateString>
    <ns120:discrepancyDetectionEnabled>true</ns120:discrepancyDetectionEnabled>
  </ns118:scanStatus>
</ns118:getLatestScanStatusResponse>
```

Example 10–21   Response (Completed Scan)

```
<ns118:getLatestScanStatusResponse>
        <ns118:scanStatus>
          <ns120:discrepancySeverityCounts>
            <ns2:entityId>15456</ns2:entityId>
            <ns2:entityVersion>1</ns2:entityVersion>
            <ns55:numberWarning>1</ns55:numberWarning>
            <ns55:numberMinor>0</ns55:numberMinor>
            <ns55:numberMajor>0</ns55:numberMajor>
            <ns55:numberCritical>0</ns55:numberCritical>
          </ns120:discrepancySeverityCounts>
          <ns120:discoveryWorkCounts>
            <ns121:totalNoOfWorkItems>2</ns121:totalNoOfWorkItems>
            <ns121:noOfCompletedWorkItems>2</ns121:noOfCompletedWorkItems>
            <ns121:noOfFailedWorkItems>0</ns121:noOfFailedWorkItems>
            <ns121:noOfInProgressWorkItems>0</ns121:noOfInProgressWorkItems>
            <ns121:startTime>07/16/2010 11:59:26</ns121:startTime>
            <ns121:endTime>07/16/2010 11:59:52</ns121:endTime>
            <ns121:duration>26s</ns121:duration>
          </ns120:discoveryWorkCounts>
          <ns120:discrepancyWorkCounts>
            <ns121:totalNoOfWorkItems>2</ns121:totalNoOfWorkItems>
            <ns121:noOfCompletedWorkItems>2</ns121:noOfCompletedWorkItems>
            <ns121:noOfFailedWorkItems>0</ns121:noOfFailedWorkItems>
            <ns121:noOfInProgressWorkItems>0</ns121:noOfInProgressWorkItems>
            <ns121:startTime>07/16/2010 11:59:52</ns121:startTime>
            <ns121:endTime>07/16/2010 11:59:55</ns121:endTime>
            <ns121:duration>3s</ns121:duration>
          </ns120:discrepancyWorkCounts>
          <ns120:jobStateString>Completed</ns120:jobStateString>

<ns120:discrepancyDetectionEnabled>true</ns120:discrepancyDetectionEnabled>
        </ns118:scanStatus>
      </ns118:getLatestScanStatusResponse>
```

Submit Discrepancies For Resolution Processing

The **submitDisDiscrepancyResolutionProcessing** operation takes a list of discrepancy entityIds and submits these discrepancies to be processed by a resolution action. This is the same as the Submit discrepancies operation in the Network Integrity UI.

The discrepancies submitted must have a discrepancy status of IDENTIFIED and have an Operation populated or else a fault is returned. The status and operation of the discrepancy can be updated using the updateDisDiscrepancy operation.

This operation is a two step operation in the Network Integrity UI to first add discrepancies to the queue, and then submit them. In the Web service this is a single operation.

If the operation is successful, the entityIds of the discrepancies submitted is returned in the response.

After submitting the discrepancies the status of the discrepancies is set to SUBMITTED.

Example 10–22   Request

```
<v1:submitDisDiscrepancyResolutionOperationsRequest>
    <!--1 or more discrepancies: -->
    <v1:disDiscrepancyRef>
        <v11:entityId>15448</v11:entityId>
    </v1:disDiscrepancyRef>
</v1:submitDisDiscrepancyResolutionOperationsRequest>
```

Example 10–23   Response

```
<ns118:submitDisDiscrepancyResolutionOperationsResponse>
    <ns118:disDiscrepancyRef>
        <ns2:entityId>15448</ns2:entityId>
    </ns118:disDiscrepancyRef>
</ns118:submitDisDiscrepancyResolutionOperationsResponse>
```

Example 10–24   Failure (one or more discrepancies not in IDENTIFIED status)

```
<ns2:Fault>
    <faultcode>ns2:Server</faultcode>
    <faultstring>DISCREPANCY_RESOLUTION_INVALID_STATUS</faultstring>
    <detail>
        <ns127:crudFault>
            <ns119:rootStackTrace/>
        </ns127:crudFault>
    </detail>
</ns2:Fault>
```

Network Integrity Web Service Scenarios

The following sections describe how to use the Web service in an end-to-end fashion.

Creating a Scan

A scan is created using the createDisConfig operation, but there may be data and entities to be created or retrieved before calling the createDisConfig operation.

Prerequisites:

- A plugin entity id is required to create a scan. The list of discovery, import, and assimilation plugins that are deployed in the system can be determined by calling getAllDisInventoryImportPlugin, getAllDisNetworkDiscoveryPlugin, and getAllDisAssimilationPlugin.

- The plug-in entity may define one or more plug-in parameters (for example, SnmpParameters) that it expects to be passed. If it does then the plug-in returned

in the previous step has one or more specificationsRef elements in the response. The expected plug-in parameters can be determined by calling getSpecification to determine the available plug-in parameters. Some plug-in parameters are optional and some are mandatory.

For more information about the parameters returned by getSpecification, see your plug-in or cartridge documentation.

- If the scan is to be tagged on creation then the tag entity ids must be retrieved using one of getAllRootDisTags, getDisTag, createDisTag.

- If the scan is to have blackout schedules on creation then the blackout entity ids must be retrieved using one of getAllDisBlackoutSchedule, getDisBlackoutSchedule, createDisBlackoutSchedule.

The response from the createDisConfig operation, if successful, is an entity id for the scan. The entity id is used for deleting, retrieving, starting, and stopping the scan.

Starting, Stopping, and Monitoring a Scan

The scan can be started using startScan operation and the DisConfig entity id that was returned when it was created. (It is also possible to do a findDisConfig operation to get the entity id).

The start scan operation returns the scan run entity id from that you can use to monitor the status and results of the scan.

It is also possible to monitor the scan progress using the DisConfig entity id and the getLatestScanStatus. This operation is more efficient and reports the current status of the scan along with other details.

An in-progress scan can be stopped using the stopScan operation and the DisConfig entity id. When the operation returns the scan is transitioned to **STOPPING** state, and asynchronously transitions to **STOPPED** when all scan processes have ended.

Retrieving Scan Results

The starting point for retrieving scan results is the DisScanRun entity. The entity id of the DisScanRun is returned when the scan was started, or can be determined by performing the findDisScanRun operation.

If the scan successfully discovers data the DisScanRun has one or more resultGroups that contain one or more rootEntityRefsRef. These ids are used in the getRootEntity call to retrieve the root of the discovered data. The getRootEntity operation, unlike other **get** calls, accepts multiple entity ids for retrieving all root entities in a single call.

The getRootEntity operation does not retrieve the complete tree of results for performance reasons and to limit scope of entity traversal. The response from getRootEntity often contains references to other entities. These entities can be retrieved using the generic getResultEntity operation, or by type-specific **get** operations (getLogicalDevice, getEquipment, getPhysicalDevice, getLogicalDevice, getEquipmentHolder, and so on).

Most result data entities have specifications. To get details about the specification the entity is using, the getSpecification operation can be called using the specificationRef on the entity.

Working with Discrepancies

The starting point for working with discrepancies is the DisScanRun entity. The entity id of the DisScanRun is returned when the scan was started, or can be determined by performing the findDisScanRun operation.

The list of discrepancies created in discrepancy detection is in the DisScanRun entity as discrepanciesRef ids. The DisDiscrepancy entity can be retrieved using the getDisDiscrepancy operation passing the discrepanciesRef from the DisScanRun entity. The discrepancies can also be found using the findDisDiscrepancy operation with search criteria.

Several fields on the discrepancy, including the status, operation (resolution action), owner, priority, reasonForFailure, and notes can be updated using the updateDisDiscrepancy operation.

Discrepancies can be submitted for resolution by calling the submitDisDiscrepancyResolutionOperations operation. The operation takes a list of discrepancies to be submitted in the request. Discrepancies must be in the status of **IDENTIFIED** and have an operation populated to be submitted.

Network Integrity Web Service Samples

Network Integrity includes example requests and responses of calling the Web service. Find these examples in the Network Integrity Web Service Samples ZIP file.

Contents of the Network Integrity Web Service Samples ZIP File

Table 10–16 describes the directories, files, and file contents for the Network Integrity Web Service Samples ZIP file.

Table 10–16   Network Integrity Web Service Samples ZIP File Contents

Directory/File	Description
build.xml	An example ANT build script that shows how to run the client with an SSL keystore as a VM argument.
WSDL-Documentation.html	Generated WSDL documentation that shows all the available operations. A short description of each operation is provided. Full WSDL source is included for reference.
IntegrityWebserviceSoapUIProject.xml	SoapUI Project File
integrity-schema\wsdl\ NetworkIntegrityControlService.wsdl	Web Service Definition (WSDL)
integrity-schema\referenceSchema	Supporting XML Schema files
integrity-schema\schema	Supporting XML Schema files
integrity-ws-client.jar	Jar file containing Java Client type generated from the WSDL
jaxb-bindings.xml	JAXB Binding file to adjust generated package names when generating client classes from WSDL. These bindings are required if not using the provided integrity-ws-client.jar and generating client class files using a Web service client generation tool.
src\oracle\integrity\ws\client\NetworkIntegrityControlService.java	This is a client side proxy class to get port types. This is the class where policy files and other authentication details are set.
src\oracle\integrity\ws\test\SampleNIClient.java	An example client java class that makes a Web service call.

Sample Java Client

Included in the Web Service Samples ZIP file is a sample java client. The sample java code is included in the src directory and contains:

- a sample client side proxy for getting a port type and setting the required policies and authentication.

- a client class that calls the getAllDisNetworkDiscoveryPlugin operation and prints the result to standard out.

To compile the sample JAVA code, the following JAR files are necessary:

- **weblogic.jar**: available in *WL_Home***/server/lib/**

- **wseeclient.jar**: available in *WL_Home***/server/lib/**

- **jrf.jar**: available in *MW_Home*/oracle_common/modules/oracle.jrf_11.1.1/

- **integrity-ws-client.jar**: included the Network Integrity Web Service Samples ZIP file.

> **Note:** The required Web service policy, **Wssp1.2-2007-Https-UsernameToken-Plain.xml** is included in the **wseeclient.jar.**

To run the sample JAVA code, you must run it with a full installation of WebLogic Server and ADF, because the JAR files referenced during compile require other JAR files. Set your classpath to point to the above JAR files in their installed location on your system. This can be done by installing WebLogic and ADF on your development system or run the client on your Network Integrity server.

If you plan on running a Web service client to communicate with a Network Integrity server that does not have a valid SSL certificate, you must download your server certificate and save it to a file to be used by your client. Then use the following VM argument when running your client. In this example, a file called jssecacerts has the SSL key that was downloaded.

```
-Djavax.net.ssl.trustStore=jssecacerts
```

Sample Soap UI Project

A SoapUI project is provided in the Cartridge Developer package to give examples of all the Web service calls and examples of the responses. The SoapUI project tests various Web service call scenarios.

To install the Soap UI, use the following procedure:

1. Download and Install SoapUI 3.5.1 (newer versions of SoapUI may work with the bundled project file, but it has not been tested)

2. Start the SoapUI application.

3. From the **File** menu, select **Import Project**.

4. Select the **IntegrityWebserviceSoapUIProject.xml** file and click **Open**.

Also in the project is a NetworkIntegrityControlMockService that simulates the real Web service. For each operation there is one or more example responses provided in the mock service. The number of example requests in the binding does not always match the number of responses because the responses would be the same structure with a different id returned (for example, create blackout response).

You can use the provided example requests or create new requests right-clicking the operation and selecting "New Request". This creates a new request with all fields populated with a question mark. Many of the example requests in the project require modification to execute successfully because the entityIds in the example does not match other systems.

The NetworkIntegrityControlMockService views examples of Web service responses for different scenarios. The mock service can also be started to respond to Web service calls with mock responses. See the SoapUI documentation for more information.

Submitting Request to the Server

To submit a request to the server you must do the following:

1. Ensure the request is valid and all mandatory attributes are set.

2. Ensure the username and password are set in the request. See the next section on how to add the username and password to the request for how this is done.

3. Add a new endpoint by clicking on the drop down at the top of the request and select **add new endpoint**.

4. Add a new endpoint with the following format:

   ```
   https://Managed_
   Server:Port/NetworkIntegrityApp-NetworkIntegrityControlWebService-context-root/
   NetworkIntegrityControlServicePortType
   ```

5. Click **Play** to submit the request.

Specifying User Name and Password in Request

To add the user name and password to a request.

1. Click **Aut** tab at the bottom of the request.

2. Enter the user name and password that has access to login to the Network Integrity UI.

3. Right click the request and select **Add WSS Username Token**.

4. Accept the default **PasswordText** and select **OK**.

   The following structure is added to the request.

   ```
   <wsse:Security soapenv:mustUnderstand="1"
   xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-
   secext-1.0.xsd">
       <wsse:UsernameToken wsu:Id="UsernameToken-4"
   xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-u
   tility-1.0.xsd">
     <wsse:Username>niuser</wsse:Username>
         <wsse:Password
   Type="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-pr
   ofile-1.0#PasswordText">niuser123</wsse:Password>
         <wsse:Nonce
   EncodingType="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-mess
   age-security-1.0#Base64Binary">ZS2K4yCoqOoQg6KL9DetBw==</wsse:Nonce>
     <wsu:Created>2010-09-13T01:21:17.578Z</wsu:Created>
     </wsse:UsernameToken>
   </wsse:Security>
   ```

5. Delete the **Nonce** and **Created** elements in the above example (highlighted in bold) to reduce errors on future calls.

11

Working with Scan Run Complete Notifications

This chapter describes an Oracle Communications Network Integrity event notification, Scan Complete Notification, which allows external components to receive asynchronous event notification messages about the completion of scans.

You can develop a client to monitor event notifications, to and trigger follow-on actions.

About Clients for Monitoring Scan Run Complete Notification Messages

You can develop a message-driven bean (MDB) or Java messaging system (JMS) client that listens to the Network Integrity event notification JMS topic (**oracle/communications/integrity/EventNotificationTopic**) for scan-complete notification messages. For example, you can write post-processing logic that listens for messages that trigger other scans or send emails or SMS messages using the MDB/JMS client.

Develop the MDB/JMS client to listen to the Network Integrity application server for the JMS topic. The client must belong to the NetworkIntegrityRole group to access the JMS topic. See *Network Integrity System Administrator's Guide* for more information on the NetworkIntegrityRole group.

Table 11–1 lists the properties used by EventNotificationTopic for client filtering.

Table 11–1   EventNotificationTopic Properties for Client Filtering

Property	Description
Status	Indicates the final scan run state: ■ COMPLETED ■ STOPPED ■ FAILED
Scan Action Name	Indicates the name of the scan action.
Scan Action Type	Indicates the type of the scan action: ■ NETWORK_DISCOVERY ■ INVENTORY_IMPORT ■ ASSIMILATION
Discrepancy Detection	A Boolean that indicates whether discrepancy detection was enabled on the scan action: ■ 1: discrepancy detection enabled. ■ 0: discrepancy detection not disabled.

Notification messages also contain other properties which may be useful to you. For example, the ScanRunId can be obtained from the message body, which retrieves additional information about the scan run.

The following example is a sample MDB/JMS client implementation model:

```
package model;

import javax.annotation.Resource;
import javax.annotation.security.RunAs;

import javax.ejb.ActivationConfigProperty;
import javax.ejb.MessageDriven;

import javax.jms.JMSException;
import javax.jms.MessageListener;
import javax.jms.TextMessage;

import weblogic.javaee.MessageDestinationConfiguration;

@MessageDriven(activationConfig =
    { @ActivationConfigProperty(propertyName = "connectionFactoryJndiName",
        propertyValue = "oracle/communications/integrity/NIXATCF"),
        @ActivationConfigProperty(propertyName = "destinationName", propertyValue
= "oracle/communications/integrity/EventNotificationTopic"),
            @ActivationConfigProperty(propertyName = "destinationType",
propertyValue = "javax.jms.Topic")
    } , mappedName = "oracle/communications/integrity/EventNotificationTopic")

@MessageDestinationConfiguration(connectionFactoryJNDIName =
"oracle/communications/integrity/NIXATCF")
@RunAs("NetworkIntegrityRole")
public class MyEjbTestBean implements MessageListener {
    @Resource
    javax.ejb.MessageDrivenContext context;
    public void onMessage(javax.jms.Message message) {
        TextMessage text = (TextMessage)message;
        try {
            // write post-processing logic here
            // like trigger other scans, or send e-mails or SMS messages
                System.out.println("entered mdb.... ");
                System.out.println("received the following message: " );
                System.out.println("Status : "+text.getStringProperty("Status"));
                System.out.println("Scan_Action_Name :
"+text.getStringProperty("Scan_Action_Name"));
                System.out.println("Scan_Action_Type :
"+text.getStringProperty("Scan_Action_Type"));
                System.out.println("Discrepancy_Detection :
"+text.getBooleanProperty("Discrepancy_Detection"));
                System.out.println("scan txt : "+text.getText());

        } catch (JMSException e) {
            //Add log statements here
        }
    }
}
```

Implementing Custom Code to Stop a Scan

A Network Integrity discovery cartridge typically comprises actions that include processors, which run sequentially in an iterative manner based on conditions (**True** or **False**).

The action controller sets the running sequence of the processors based on the order in which the processors are configured. Usually a processor is invoked only once and after its completion, the controller invokes the next processor, until all processors in an action are invoked. However, one or more processors may be run repeatedly in an iterative manner.

For example, when importing an inventory system, it is typical to first get a list of devices from the inventory system, then go through the list of devices, and then import each device individually into Network Integrity. In this example, the processor importing a single device is repeatedly run for all the devices in the returned device list.

A running scan does not stop immediately when you click **Stop Scan**. If a processor in a scan had already started before you clicked **Stop Scan**, the processor continues to run until its completion; the next processor in the sequence looks for the value of the condition and the custom code in its invoke method to stop the processor; if the condition is **True**, the scan is stopped before the next processor starts and all the results of the scan are deleted.

You can add the custom code to any processor depending on its functionality and your requirements. The amount of time that a scan will take to stop depends on how you configure the processors and how you implement the custom code to stop the processors.

To stop a scan when you click **Stop Scan**, Oracle recommends that you add the following custom code to the beginning of the processor's invoke method and ensure that this code resides outside the try/catch block:

```
if(((BaseDiscoveryController)context).isScanStopped()){
logger.info("Scan is stopped, interrupting data collection");
// Add custom code here to close any open resources, such as connections, sockets,
// sessions, and so on.
throw new ProcessorException("Scan is interrupted");
}
```

12

Working with JCA Resource Adapters

This chapter provides overview information about the J2EE Connector Architecture (JCA) simple network management protocol (SNMP) resource adapter included with Oracle Communications Network Integrity and other third party or customized JCA resource adapters that may be used with Network Integrity.

This chapter contains the following sections:

- About Resource Adapters
- About Productized SNMP JCA Resource Adapter
- About Third Party or Customized JCA Resource Adapters

About Resource Adapters

A JCA resource adapter is a system-level software driver used by a Java application to connect to an Enterprise Information System (EIS). The resource adapter can be configured to use any protocol required by the EIS for connectivity. The resource adapter plugs into an application server (for example Oracle Fusion Middleware) and provides connectivity between an EIS (for example, a database system), the application server, and the enterprise application (see Figure 12–1, "JCA Functional Blocks").

JCA defined a standard architecture for connecting a J2EE platform to heterogeneous EISs. Examples of EISs include Enterprise Resource Planning (ERP) and mainframe transaction processing (TP). The connector architecture defines a Common Client Interface (CCI) for EIS access. The CCI defines a client API for interacting with heterogeneous EISs and enables an EIS vendor to provide a standard resource adapter for its EIS.

An application server that support JCA, like Fusion Middleware, can ensure seamless connectivity to multiple EISs. In the same way, any EIS with a JCA resource adapter can plug into an application server that supports JCA.

For details about the JCA 1.5 specification and additional JCA documentation, see:

http://java.sun.com/j2ee/connector/download.html

Figure 12–1   JCA Functional Blocks

Understanding JCA Resource Adapter Connectivity Options

A resource adapter provides the following types of connectivity between an application and an EIS.

- **Outbound communication**: The resource adapter allows an application to connect to an EIS system and perform work. The application initiates all communication. The resource adapter serves as a passive library for connecting to an EIS, and executes in the context of the application threads.

- **Inbound communication**: The resource adapter allows an EIS to call application components and perform work. The EIS initiates all communication. The resource adapter can request threads from the application server or create its own threads.

- **Bi-directional communication**: The resource adapter supports both outbound and inbound communication.

Understanding JCA Resource Adapters with Network Integrity

This chapter describes productized SNMP JCA resource adapter and 3rd party or customized JCA resource adapters, and their use within Network Integrity.

Network Integrity administrators can configure the productized SNMP JCA resource adapter included with the Network Integrity software. Network Integrity system integrators can extend this SNMP JCA resource adapter with additional MIB files at run time to poll additional SNMP object identifiers (OIDs).

In addition to the productized JCA resource adapter for use with SNMP, Network Integrity system integrators can also use any standard J2EE JCA resource adapters (3rd party or customized) in their customized Network Integrity cartridge. They can deploy these resource adapters wherever the Network Integrity application is deployed. These adapters can be standalone, or clustered within a Weblogic server.

Network Integrity cartridges can:

- use a deployed resource adapter

- communicate with various network devices

- send commands

- collect data through various protocols (for example, SNMP, TLI, or CORBA)

See *Oracle Communications Design Studio Developer's Guide* for details on creating a Network Integrity cartridge project. See *Network Integrity Installation Guide* for details on deploying an SNMP JCA resource adapter.

About Productized SNMP JCA Resource Adapter

The SNMP discovery processor uses the SNMP JCA resource adapter, contained in the Network Integrity software to poll the SNMP enabled network devices.

The SNMP JCA resource adapter implements the connector architecture to provide SNMP functions for Network Integrity. Oracle Fusion Middleware (the application server) is the container for the SNMP JCA resource adapter and provides connection pool management. The SNMP JCA resource adapter provides outbound communication only to Enterprise Information Systems (network devices) and transaction management is not required.

The SNMP JCA resource adapter supports all SNMP-enabled network devices provided a proper set of MIB files are installed.

SNMP JCA resource adapter has record and playback functions for user who want to collect and view raw SNMP data and later reuse the data for testing purposes. For details on how to configure the SNMP resource adapter to run in record and playback mode, see "Record and Playback Mode".

Installing the SNMP JCA Resource Adapter

The SNMP resource adapter installs as part of the Network Integrity Installer. See *Network Integrity Installation Guide* for more details.

Extending the SNMP JCA Resource Adapter

The SNMP resource adapter is installed with the following pre-bundled MIB files:

- ATM-MIB

- ATM-TC-MIB

- CISCO-CONFIG-MAN-MIB

- CISCO-ENTITY-VENDORTYPE-OID-MIB

- CISCO-FRAME-RELAY-MIB

- CISCO-PRODUCTS-MIB

- CISCO-SMI

- CISCO-TC

- CISCO-VLAN-IFTABLE-RELATIONSHIP-MIB

- CISCO-VTP-MIB

- ENTITY-MIB

- IANAifType-MIB

- IF-MIB

- INET-ADDRESS-MIB

- IP-MIB

- RFC1155-SMI

- RFC1213-MIB

- RFC1315-MIB

- RMON-MIB

- SNMP-FRAMEWORK-MIB

- SNMPv2-CONF

- SNMPv2-MIB

- SNMPv2-SMI

- SNMPv2-TC

- enterprise-numbers.txt

If a device is not supported by the MIB files included with the SNMP JCA resource adapter, then the user must install additional MIB file(s) that support such a device. These additional MIB files provide the corresponding MIB OIDs and definitions for the device that the user wants to poll. Ensure that the same MIB file(s) are available in Design Studio for the corresponding cartridge development. The MIB file(s) on both Design Studio and the SNMP JCA resource adapter must match. Manually copy these MIB files to the SNMP JCA resource adapter.

To copy new MIB files to the SNMP JCA resource adapter, use the following steps:

1. Log in to the server where Network Integrity is installed.

2. Go to directory *NI_HOME*/**integrity/snmpAdapter/mibs**, where *NI_HOME* is the location chosen using the NI installer during the Network Integrity installation.

3. Copy the new MIB files to this directory.

> **Tip:** There is no need to restart the server. The SNMP JCA resource adapter automatically loads the new MIB files when needed.

4. Perform an update operation of 'snmpadapter' application in Admin console.

Record and Playback Mode

SNMP JCA resource adapter supports record and playback mode.

When the SNMP JCA resource adapter is configured to run in record mode, the resource adapter polls a network device, and the device returns the polled data to the resource adapter. The SNMP JCA adapter then returns the SNMP data to the discovery cartridge and also writes the SNMP data to a file that it stores on a local hard drive.

When the SNMP JCA resource adapter is configured to run in playback mode, the resource adapter does not require a connection to the network device. Instead the resource adapter reads the SNMP data file (created in Record mode and stored on the local hard drive) and sends the SNMP data back to discovery cartridge.

To switch the mode of SNMP resource adapter, use the following steps to create a configuration file.:

1. Log in to the server where Network Integrity is installed.

2. Go to directory *NI_DOMAIN_HOME*/**config**.

3. Create a directory called **snmpAdapterConfig**.

4. Within the new directory, create a file called **snmpAdapter.properties**.

5. Add the following content to the file:

```
#MODE=normal
MODE=record
#MODE=playback
```

> **Tip:** Enable a mode by removing the comment symbol (#) from the beginning of the line. In the above example, record mode is enabled.

The SNMP JCA resource adapter creates the record files in *NI_Domain*/**snmpData**. The exact directory and filename depends on the IP address. For example, device 10.156.66.191 is stored at *NI_Domain*/**snmpData/10/156/66/191/10.156.66.191_** *XXXXX*.**rec**, where *XXXX* is the name of the request set by the scan element.

Playback mode loads recorded SNMP results and send them back to the Network Integrity cartridge without actually polling the network devices.

There is no need to restart the Weblogic server after changing the SNMP resource adapter properties file. SNMP JCA resource adapter dynamically switches the mode based on the current configuration in the properties file.

For clustered environment, the user manually creates and modifies the properties file for every SNMP JCA resource adapter installed on every node.

Invoking the SNMP JCA Resource Adapter in a Network Integrity Cartridge

Design Studio creates (code-generates) the complete implementation of the SNMP processor for discovery action. This SNMP processor can perform SNMP discoveries of SNMP enabled network devices.

After the SNMP processor discovers a device, the processor can use the SNMP JCA resource adapter to perform SNMP polling on the discovered network devices.

There is no coding effort to use the SNMP resource adapter in a Network Integrity cartridge.

About Third Party or Customized JCA Resource Adapters

The following sections provides information on building JCA resource adapters and on invoking third party or custom Resource adapters.

Building a JCA Resource Adapter in WebLogic

To create a JCA resource adapter for use in a customized Network Integrity cartridge, see:

http://download.oracle.com/docs/cd/E12839_01/web.1111/e13732/toc.htm

This Fusion Middleware document provides detailed instructions for creating a resource adapter in Weblogic.

Invoking a Third Party or Customized JCA Resource Adapter

The following workflow describes the steps required to implement third party or customized JCA resource adapters in Network Integrity.

1. Deploy third party or customized JCA resource adapters into the Network Integrity system.

2. Implement a Design Studio discovery processor to invoke the third party or customized JCA resource adapter.

   a. Locate the following code auto-generated from the discovery processor.

   ```
   @Override
   public SampleProcessorResponse invoke(DiscoveryProcessorContext context
                     SampleProcessorRequest request) throws ProcessorException {
         // TODO Auto-generated method stub
    return null;
   }
   ```

   b. Use the **SampleProcessorRequest** generated class to obtain the address scope, property group, and other attributes.

   > **Tip:** This class provides important elements used when invoking a resource adapter. For example, to use a TL1 resource adapter to make a TL1 request, the TL1 resource adapter needs to know which device it should communication with. This information is obtained from the **SampleProcessorRequest** in the following sources:
   >
   > - IP address: available from the address scope
   >
   > - port number: available from the property group
   >
   > - login information for the TL1 session including username and password: available from the property group

   c. Use the data provided by **SampleProcessorRequest** to implement the Java code to invoke the JCA resource adapter.

Depending on the resource adapter, the way to invoke a resource adapter can different. Typically the invoke process requires several JNDI name lookups to get some JCA Connection Factory and Interaction Specification classes. From the JCA Connection Factory, the user can create Interaction. Next is to do the execution from Interaction by passing the Interaction Specification.

If user is using an existing 3rd party resource adapter, it should come with a developer guide that provides the detailed instruction on how to implement the client code to invoke this resource adapter. If a user creates a customized resource adapter from scratch, the user should have all the knowledge on how to invoke this customized JCA resource adapter.

The following code snippet demonstrates how to invoke a JCA resource adapter that implements Common Client Interface (CCI):

```
…
context = new InitialContext();
SampleAdapterConnectionSpecImpl cspec =
               (SampleAdapterConnectionSpecImpl)context.lookup(JNDI_SAMPLE_CONN_
SPEC);
cxFactory = (ConnectionFactory) context.lookup(JNDI_SAMPLE_CONN_FACTORY);
connection = cxFactory.getConnection(cspec);
ispec = (SampleAdapterInteractionSpec)context.lookup(JNDI_SAMPLE_INTER_SPEC);
```

```
interaction = connection.createInteraction();
RecordFactory recordFactory = cxFactory.getRecordFactory();
IndexedRecord input =
recordFactory.createIndexedRecord(SampleAdapterIndexedRecord.INPUT);
input.add(request);
IndexedRecord output =
recordFactory.createIndexedRecord(SampleAdapterIndexedRecord.OUTPUT);
interaction.execute(ispec, input, output);
out=(String)output.get(SampleAdapterIndexedRecord.MESSAGE_FIELD);
…
```

In this example, the "out" contains the collected results as an XML document as String. However, different resource adapter have different output. To detail all possible kinds of output is beyond the scope of this document.

The final output should be wrapped inside the **SampleProcessorResponse** class (code-generated) and return as the returned value of this **invoke** method.

13

Working with Reports Extensibility

This chapter provides overview information about the Oracle Business Intelligence (BI) Publisher and reports extensibility for Oracle Communications Network Integrity.

This chapter contains the following sections:

- About BI Publisher
- Reports Provided with Network Integrity
- Network Integrity Report Building Blocks
- Developing BI Publisher Reports
- Using BI Publisher Features
- Installing and Integrating BI Publisher with Network Integrity
- Uploading and Exporting Reports

About BI Publisher

Oracle BI Publisher (formerly XML Publisher) is an enterprise reporting solution to design, use, author, manage, and deliver report documents in various formats. This tool provides the following services:

- Creating New Report
- Uploading Existing Report
- Organizing reports in folders
- Viewing a report in various formats like RTF, HTTP, PDF, XLS
- Scheduling a report for delivery to destination like Email id
- Internationalization of report
- Configuring various data sources like Oracle Business Intelligence Enterprise Edition (OBIEE), Web service, and database

This chapter does not describe all Oracle BI Publisher features. Customized, or more advanced reports can be created using BI Publisher. For more information about Oracle BI Publisher, see the Oracle Business Intelligence Publisher Installation documentation:

http://download.oracle.com/docs/cd/E12844_01/doc/bip.1013/e12690/toc.htm

About BI Publisher Desktop

BI Publisher Desktop is a plug-in for Microsoft Word that enables a user to create report templates and represent data from data sources. This plug-in provides various wizards to create tables, grouped tables, graphs, charts, and so on. It internally uses Microsoft Word's native features and the eXtensible Stylesheet Language (XSL) to implement these wizards.

Reports Provided with Network Integrity

Network Integrity includes the following BI Publisher reports:

- Scan History Report
- Discovery Scan Summary Report
- Device Discrepancy Detection Summary Report
- Device Discrepancy Detection Detail Report
- Discrepancy Corrective Action Report

Scan History Report

The *Scan History Report* shows the discovery and discrepancy summaries for each scan for each scan configuration falling within the specified start and end dates. This report is accompanied by the following graphs:

- Discovery Scan History: a graph showing a history of the run discovery scans.
- Discrepancy Scan History: a graph showing a history of the run discrepancy scans.
- Discrepancy Severity History: a graph showing a history of the discrepancies by severity.

The following fields are used to generate this report:

- Start Time: the date stamp indicating when a scan started.
- End Time: the date stamp indicating when a scan finished.

Discovery Scan Summary Report

The *Discovery Scan Summary Report* shows the summary of the latest scan for each scan configuration, per vendor and per device type. This report generates a pie-chart, illustrating the summary findings, for each scan configuration.

The following fields are used to generate this report:

- Vendor: the name of the vendor for the discovered device.
- Device Type: the type of device discovered.

Device Discrepancy Detection Summary Report

The *Device Discrepancy Detection Summary Report* shows the summary of the latest scan for each scan configuration. This report generates a pie-chart that shows the accuracy of the latest scans for each scan configuration.

The following fields are used to generate this report:

- Vendor: the name of the vendor for the discovered device.
- Device Type: the type of device discovered.

Device Discrepancy Detection Detail Report

The *Device Discrepancy Detection Detail Report* lists details of all discrepancies for the latest scan for each scan configuration.

The following fields are used to generate this report:

- Vendor: the name of the vendor for the discovered device.

- Root device name: the name of the root device in the scan result tree.

- Root device type: the type of the root device in the scan result tree.

- Owner: the user name of the owner of the discrepancy.

- Parent entity type: the type of the parent entity on which discrepancy occurred.

- Parent entity name: the name of the parent entity on which discrepancy occurred.

- Entity type: the type of the entity on which discrepancy occurred.

- Inventory value: the value of the field on the inventory side on which discrepancy occurred.

- Network value: the value of the field on the network side on which discrepancy occurred.

- Severity: the severity of the discrepancy (for example, major, critical, minor, warning).

- Discrepancy type: the type discrepancy (for example, entity+, entity-, attribute).

- Description: the description of the discrepancy.

- Status: the status of the discrepancy (for example, processed, failed, ignored).

- Scan name: the name of the scan in which the discrepancy is found.

Discrepancy Corrective Action Report

The *Discrepancy Corrective Action Report* shows corrective actions against specified discrepancies for the latest scan for each scan configuration. Discrepancies that have not been actioned are not considered in this report.

The following fields are used to generate this report:

- Submitted By: the user who submitted the discrepancy for correction.

- Action: the action taken against the discrepancy.

- Discrepancy Status: Status of the discrepancy.

- Owner: the user name of the owner of the discrepancy.

- Priority: the priority of the discrepancy.

- Failure Reason: the reason for failure for the corrected discrepancy.

- Discrepancy Type: the type discrepancy (for example, entity+, entity-, attribute)

- Entity Type: the type of the entity on which discrepancy occurred.

- Inventory Value: the value of the field on the inventory side on which discrepancy occurred.

- Network Value: the value of the field on the network side on which discrepancy occurred.

Network Integrity Report Building Blocks

Network Integrity uses BI Publisher to generate reports. These reports use Rich Text Format (RTF) templates and a report Definition Extensible Markup Language (XML) file (.xdo) that tracks the various data sources, parameters, values, rendering options, and report properties that BI publisher uses to populate the RTF reports.

RTF Templates

The RTF template defines the layout and display for report data. The BI Publisher server uses the templates to format the data from various data sources. These formats include tables, charts, graphs, and so on.

A user can use general word processing features and BI Publisher's simplified tags for XSL expressions. The template parser inside the server processes the RTF document and converts it to XLS formatting objects (XSL-FO), useful for rendering the final formatted report in PDF, RTF, or HTML formats.

Report Definition Files

The reports Definition file (.xdo) contains all the configurations pertaining to a report and all the layout definitions that refer to the various templates used by the report.

Data Source Query Tools

The data source tools use various protocols such as SQL, Web service, and XML files that can get data from data sources for use in a report. Network Integrity typically uses SQL queries pertaining to Network Integrity schemas and retrieves query results from a Network Integrity database through Java database connectivity (JDBC).

See *Network Integrity Information Model Reference* for more information about how entities and parameters relate to the Network Integrity database.

Parameters

Parameters capture values at run-time and use these values to filter or manipulate data. Parameter definitions includes data type, default value, parameter type, display label, and list of values.

List of Values

The Menu parameter type uses the list of values definitions that BI Publisher render as a dropdown in a report. The dropdown menu is populated with values defined in the list of values. This definition can be a pre-defined hard coded list or it can be the result of an SQL query.

Rendering Options

You can render reports using formats like PDF, RTF, HTML, EXCEL, and so on. You can choose the render option for the reports they create.

Report Properties

The following list provides details on various report property functions:

- **Run report online**: Enable this property to view a report online. You can only schedule and view reports in history after BI Publisher runs the scheduled report.

- **Show Controls**: Enable this property to see the parameters and other View report screen controls like **View**, **Send**, **Schedule**, **Analyzer**, and so on.

- **Show Report Links**: Enable this property to see a link menu in the view report screen that contains different menu items that provide HTTP links for a report. You can bookmark these links.

- **Auto Run**: Enable this property to cause a report to run automatically when a user attempts to view it. If this property is not enabled, the user must click **View** in the view screen of the report.

- **Enable document cache**: Enable this property to provide a cache in the report document. This property is appropriate for reports that connect to live databases that are frequently updated.

Developing BI Publisher Reports

This section explains how to develop a custom report by creating an example report called *Discrepancy Severity History Report*.

Report Requirements

The *Discrepancy Severity History Report* requires the following

- Display the following fields:
  - **Scan Name**
  - **Discrepancy Scan Start & End time**
  - **Counts of Critical, Major, Minor, Warning discrepancy types**
- Filter on discrepancy start date.
- Group data by scan name.
- Create a bar chart for all discrepancy severity types.

Knowledge of the Network Integrity data model is necessary to write the SQL queries needed for the *Discrepancy Severity History Report* (see Network Integrity Models). This information is provided in the following sections.

Configuring a Data Source Tutorial

The following steps provide a tutorial for configuring a data source:

1. Log on to BI Publisher as an Administrator.

2. Click the **Admin** tab.

3. Click **JDBC Connection** under the **Data Sources** section

4. Click **demo** to edit the connection details for this data source.

5. In the **Update Data Source** screen, enter the following details:

   a. For **Connection String,** enter:

   jdbc:oracle:thin:@*Host_name***:***Port***:***SID*

   For example, a sample connection string may look like **jdbc:oracle:thin:@myhost:1521:orcl**

   b. For the **User Name**, enter:

   oe

   c. For the **Password,** enter:

oe

    d. For the **Database Driver Class** enter the default:

    oracle.jdbc.driver.OracleDriver

6. Click **Test Connection**.

7. Click **Apply** to save the connection details after confirmation that the connection is successfully established.

Creating a Report Tutorial

The following steps provide a tutorial for creating a report:

1. Log on to BI Publisher as an Administrator.

2. Click the **Reports** tab.

3. Click the **MyFolders** link.

4. Click **Create a new folder link** in the **Folder and Report Tasks** tab.

5. Enter the following folder name:

    Test

6. Click **Create** to create the folder.

7. Click the new **Test** folder.

8. Click **Create a new Report**.

9. Enter the following report name:

    Discrepancy Severity History Report

10. Click **Create** to create the report.

11. Click **Edit** link below the report name.

12. Select **Parameter**.

13. Click **New** button.

14. Enter the following values to create a parameter called **p_StartDate**.

    a. For **Identifier,** enter:

    p_StartDate@

    b. For the **Parameter Type**, select **Date**.

    c. For the **Display Label,** enter:

    Discrepancy Begin Date Range Start

    d. For the **Date Format String** enter:

    MM-dd-yyyy

15. Repeat steps 12 to 14 to create a parameter called **p_EndDate**. Label the **Identifier** as **p_EndDate** and the **Display Label** as **Discrepancy Start Date Range End**.

16. Select **Data Model**.

17. Click **New**.

18. Enter the following values to create a data set called **MyDataSet**.

    a. For **Name** enter:

MyDataSet

b. For **Type**, select **SQL Query**.

c. For **Data Source** select **demo** from the dropdown menu.

d. For the **SQL Query** text box enter the following SQL query as required by the Network Integrity model to display the field described in "Report Requirements":

```
select  dc1.NAME as SCANCONFIGNAME,
        case when dsr.DISCREPDETECTBEGINTIME is null then null else to_
char(dsr.DISCREPDETECTBEGINTIME,'YYYY-MM-DD HH12:MI:SS TZR AM') end
STARTTIME,
         nvl(to_char(dsr.DISCREPDETECTENDTIME, 'YYYY-MM-DD HH12:MI:SS TZR
AM'),' ') as DISCREPDETECTENDTIME,
CASE WHEN DISCREPDETECTBEGINTIME IS NULL THEN NULL ELSE
nvl(ddc.NUMBERCRITICAL,0) END C,
        CASE WHEN DISCREPDETECTBEGINTIME IS NULL THEN NULL ELSE
nvl(ddc.NUMBERMAJOR,0) END M,
        CASE WHEN DISCREPDETECTBEGINTIME IS NULL THEN NULL ELSE
nvl(ddc.NUMBERMINOR,0) END mi,
        CASE WHEN DISCREPDETECTBEGINTIME IS NULL THEN NULL ELSE
nvl(ddc.NUMBERWARNING,0) END w,

from
        DISCONFIG dc left outer join DISCONFIG dc1 on dc.scanconfig=
dc1.entityid
        left outer join DISSCANRUN dsr on dc.ENTITYID=dsr.CONFIG
        left outer join DISDISCREPANCYCOUNTS ddc on
dsr.DISCREPANCYCOUNTS=ddc.ENTITYID

where
dc.scanconfig is not null and
(:p_StartDate is null or trunc(dsr.DISCREPDETECTBEGINTIME, 'DDD') >= :p_
StartDate) and
(:p_EndDate is null or trunc(dsr.DISCREPDETECTBEGINTIME, 'DDD') <= :p_
EndDate)
```

19. Save the data set. The XML source data can now be viewed in a report.

Building an RTF Template Tutorial

The following steps provide a tutorial for building an RTF template:

1. Using Microsoft Word, create an RTFF file called *DiscrepancySeverityHistory.rtf*.

2. Open the Microsoft Word file.

3. From the **AddIns** menu, select **BI Publisher**.

4. Log on to BI Publisher as an Administrator. The Login Wizard immediately opens to a screen showing the *DiscrepancySeverityHistory.rtf* report selected.

5. Click the **Open Report** button.

6. Enter the following text as the title of the report (see Figure 13–1):

   Discrepancy Severity History Report

7. Enter the following text as a description of the report (see Figure 13–1):

   The report shows historical discrepancies severity summaries for scans.

8. Enter the following code in the field after the description (see Figure 13–1). This code is a declaration of the parameter strings already defined in "Creating a Report Tutorial":

```
<?param@begin:p_Start;'None'?>
<?param@begin:p_End;'None'?>
```

9. Below the declaration, insert a table with two columns (see Figure 13–1).

10. Enter the table column titles as **FilterName** and **FilterValue** and enter **StartDate** and **EndDate** in first column of both the rows (see Figure 13–1).

11. Insert the following code into the second and third rows of the second column respectively (see Figure 13–1).

```
<?$p_Start?>
<?$p_End?>
```

Figure 13–1   Report Example
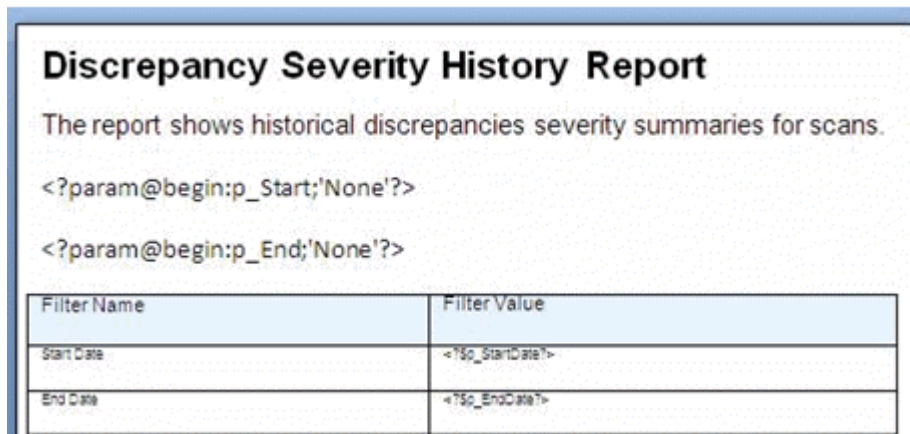
12. Select **Oracle BI Publisher** then **Insert** then **Table wizard**.

13. Select the **Table** option.

14. Click **Next**.

15. Select the data set that you defined in "Creating a Report Tutorial".

16. Select the field to display on the report and arrange them in an appropriate order.

17. Click **Next**.

18. Select **Scanconfigname** from the **Group By** dropdown menu and to be grouped by horizontally.

19. Click **Next**.

20. Select any field from the **Sort By** dropdown menus. This step is optional.

21. Label each field as required.

22. Click **Finish**.

23. Place the cursor below the table within the groupby clause.

24. Select **Oracle BI Publisher** then **Insert** then **Chart**.

25. From the **Data** explorer tab, drag the **STARTTIME** field into the text box adjacent to **Labels**.

26. Drag the **C**, **M**, **Mi**, **W** fields into the text boxes adjacent to **Values**.

27. From the **Type** dropdown menu, select **Line Graph**.

28. From the **Style** dropdown menu, select **Confetti**.

29. Select the **Group Data** check mark.

30. Select the **Chart is inside group** check mark.

31. Change the items as required in the **Properties** list. The values are **True** or **False**.

32. Click **OK** to insert the chart into the Microsoft Word document.

33. Select **Oracle BI Publisher** menu then **Upload Template As** to upload the template to the BI Publisher server.

34. Enter a name for the new template, for example, *SevHistoryReportTemplate*.

35. Click **OK**.

36. Navigate to the BI Publisher server where the template is now located.

37. Click **View** to see the reports.

Using Microsoft Word Native Features and XSL

RTF templates can use standard word processing features of Microsoft Word like tables, formatting, styles, and so on. The BI Publisher Desktop plug-in also provides additional wizards for use in Microsoft Word. These wizards internally generate XSL equivalent code which is interpreted by BI Publisher server while rendering the template.

Using BI Publisher Features

Oracle BI Publisher provides the following important features:

- Scheduling Reports
- Localizing Reports

Scheduling Reports

To schedule reports, use the following tasks:

- Enabling BI Publisher Scheduler Tutorial
- Using BI Publisher to Schedule Reports Tutorial

Enabling BI Publisher Scheduler Tutorial

BI Publisher requires a database to create a scheduling schema and a database user to use the scheduling schema.

To create an database user for scheduler schema and grant required permissions to that user, use the following steps:

1. Use the following SQL example to create a database user:

```
SQL> CREATE USER bipubsched
  2  IDENTIFIED BY welcome
  3  DEFAULT TABLESPACE USERS
  4  TEMPORARY TABLESPACE TEMP
  5  QUOTA 20G ON USERS
  6  QUOTA 1M ON TEMP;
```

```
User created.

 SQL> GRANT CREATE SESSION TO bipuser; -- or "GRANT CONNECT TO bipuser;"

 Grant succeeded.

 SQL> grant create table to bipublisher;

 Grant succeeded.
```

2. Log on to BI Publisher as an Administrator.

3. Select **Admin**.

4. Under **System Maintenance**, select **Scheduler Configuration**.

5. Enter the following values:

   a. For **Database Connection Type** dropdown menu, select **jdbc**.

   b. For **Database Type** dropdown menu, select **Oracle 10g**.

   c. For **Connection String**, enter **jdbc:oracle:thin:@10.177.219.0:1521:ORCL**.

   d. For **Username**, enter **bipudsched**.

   e. Enter a **Password**.

   f. For **Database Driver Class**, enter **oracle.jdbc.OracleDriver**.

6. Click **Text Connections** to verify if the connection is successful.

7. If the connection was successful, click **Install Schema** to install the schema in the database.

8. Restart BI Publisher to apply the change. You can schedule reports after the server restarts.

Using BI Publisher to Schedule Reports Tutorial

To schedule reports, use the follow procedure:

1. In BI Publisher, navigate to the folder you stored your reports.

2. Click **Schedule** below the report you want to schedule.

3. In the **Schedule Reports** screen, enter values as required.

   > **Tip:** Before you can configure Email destinations, ensure that the mail server has been configured. This can be done from the **Admin** tab on the main menu in the **Delivery** section.

4. Click **Submit** to schedule the report. When the scheduled report has been run, the report is stored in the **History**.

Localizing Reports

The localization of a report is achieved by translating the templates used for that report and adding those templates to the report Definition in BI Publisher. There are two options for adding translated templates to a report Definition.

Localized Template Option

In this option, there is one RTF template for one locale supported for the report. There are *n* RTF templates for *n* locales supported. Figure 13–2 illustrated that there is an RTF template with French locale and the data is in an XML file. BI Publisher uses both and renders the report in French text.

Figure 13–2   Localized Template
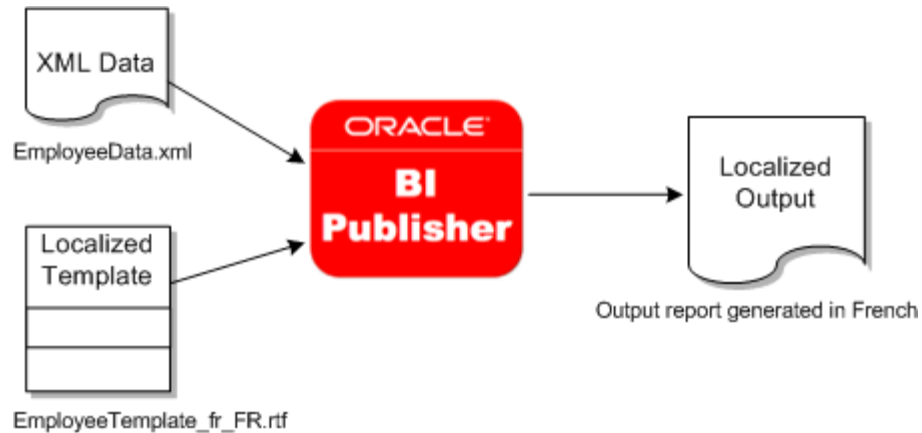
Naming convention for the report template is **TemplateName**_*<language code>*_*<TERRITORY CODE>***.rtf**.

This RTF template must be uploaded to BI Publisher to include this in the report Definition.

XLIFF File Option

In this option, there is one RTF template for one locale supported for the report. There are *n* RTF templates for *n* locales supported. illustrated that there is an RTF template with French locale and the data is in an XML file. BI Publisher uses both and renders the report in French text.

In this option, there one RTF template but *n* XML localization interchange file format (XLIFF) files for *n* locales supported. Figure 13–3 illustrates that BI Publisher receives data in XML, base template, and an XLF file with translated text strings to render the report in French text.

Figure 13–3   XLIFF Localized Template
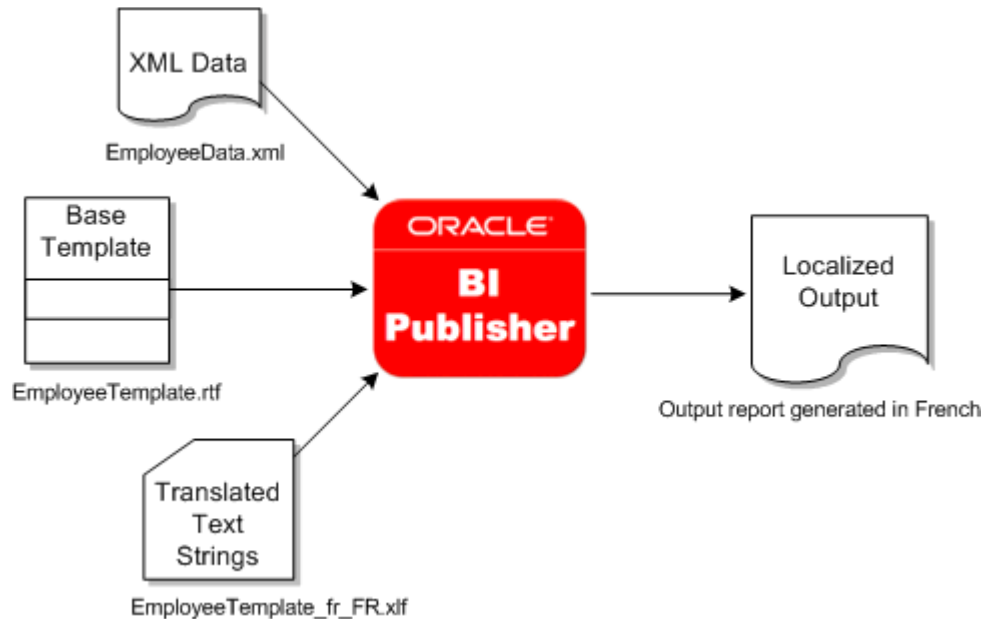
Naming convention for the report template is **TemplateName_**<*language code*>_
<*TERRITORY CODE*>**.rtf**.

Upload all supported *n* XLF files and corresponding *n* locales to the BI Publisher
report to include these into the report Definition.

Installing and Integrating BI Publisher with Network Integrity

BI Publisher can be installed as a standalone application or as a web application
deployed in the same application server where the Network Integrity is deployed.

BI Publisher is partially integrated with Network Integrity which supports various BI
Publisher application links in the Network Integrity **Links** tab. The BI Publisher
application can be launched from Network Integrity using the BI Publisher link.

Installing BI Publisher in Standalone Mode

The following link provides the steps to install BI Publisher in standalone mode:

http://download.oracle.com/docs/cd/E12844_
01/doc/bip.1013/e12690/T434820T487782.htm

Deploying BI Publisher

To deploy BI Publisher as an Enterprise ARchive (EAR) in a WebLogic application
server where Network Integrity is running, use the following steps:

1. Download the BI Publisher zip file.

2. Unzip the zip file.

3. Navigate to the **RH_Linux/Oracle_Business_Intelligence_Publisher_
    Standalone/manual/generic** directory where the **xmlpserver** war and ear files are
    located.

4. Explode the war file to create an **xmlpserver** directory with all the war files
    present within using the following command:

jar -xvf xmlpserver.war

5. Deploy the **xmlpserver** folder onto WebLogic as an application.

6. Copy all the files under **RH_Linux/Oracle_Business_Intelligence_Publisher_Standalone/manual/generic** directory into (for example) **~/BIP** or **/home/<username>/BIP**

7. Search the **xmlp-server-config.xml** file in the Network Integrity domain of WebLogic.

8. Insert the following in the file:

    path="/home/<username>/BI

9. Save the file.

10. Copy font files from **RH_Linux/Oracle_Business_Intelligence_Publisher_Standalone/manual//fonts** and put them in **JAVA_HOME/jre/lib/fonts**. This is the JDK that the Network Integrity installer uses.

11. Restart the application server.

12. Launch BI publisher. For example:

    ```
    http://localhost:7101/xmlpserver
    ```

13. Login as administrator.

Integrating BI Publisher with Network Integrity using WebLogic Enterprise Manager

Complete one of the following prerequisites before attempting this procedure:

- Installing BI Publisher in Standalone Mode

- Deploying BI Publisher

To integrate BI Publisher with Network Integrity using the WebLogic Enterprise Manager, use the following steps:

1. Launch WebLogic enterprise manager. For example

    ```
    http://10.177.219.11:7101/em
    ```

2. Log on to Enterprise Manager using the username and password used to log on to WebLogic Admin Console

3. Click the WebLogic Domain folder link in the left side navigation panel.

4. Expand the Weblogic Domain folder to display the domains.

5. Select the domain where the Network Integrity application is installed.

6. Select **System Mbean Browser** from the dropdown menu. The System Mbean Browser shows all the Mbeans deployed in WebLogic server.

7. Select the NIRegionLinksService Network Integrity Mbean to view its properties.

8. Enter BI Publisher link against any URL property (for example, URL06).

9. Enter the display string of the URL against URLName06. This link appears in the **Links** tab of the Network Integrity UI the next time the UI refreshes.

Integrating BI Publisher with Network Integrity using JConsole

Complete one of the following prerequisites before attempting this procedure:

- [Installing BI Publisher in Standalone Mode](#)

- [Deploying BI Publisher](#)

To integrate BI Publisher with Network Integrity using the JConsole:

1. Use the following command to launch JConsole in Windows.

    ```
    %JAVA_HOME%/bin/jconsole -J-Djava.class.path=%JAVA_
    HOME%\lib\jconsole.jar;%JAVA_HOME%\lib\tools.jar;%WLS_
    HOME%\server\lib\wljmxclient.jar
    -J-Djmx.remote.protocol.provider.pkgs=wlserver.management.remote
    ```

2. Use the following command to launch JConsole in Linux.

    ```
    jconsole -J-Djava.class.path=$JAVA_HOME/lib/jconsole.jar:$JAVA_
    HOME/lib/tools.jar:$WLS_HOME/server/lib/wlserver.jar
    -J-Djmx.remote.protocol.provider.pkgs=wlserver.management.remote
    ```

3. In the JConsole **New Connection** screen, select the **Remote Process** option.

4. In the Remote Process text box, enter **service:jmx:t3://10.147.240.137:7007/jndi/wlserver.management.mbeanservers.runtime**

    The host name corresponds to the IP address of the host where Network Integrity is installed.

    The port number is the port on which the managed server is running. For example, 7003.

5. Enter the same **username** and **password** used for accessing the WebLogic admin console and click **connect**.

6. Navigate to **oracle.communications.integrity** then **NIRegionalLinksService** then **oracle.comminications.integrity** then **Attributes**. **NIRegionalLinkService** is the Mbean placeholder for BI Publisher link.

7. Set URL06 with the BI PublisherWebLogic URL.

8. Set URLName06 with the corresponding display title. Now the Network Integrity shows BI Publisher link that points to the BI Publisher URL in the **Links** tab after next page refresh.

Integrating BI Publisher with the Network Integrity Installer

Complete one of the following prerequisites before attempting this procedure:

- [Installing BI Publisher in Standalone Mode](#)

- [Deploying BI Publisher](#)

To integrate BI Publisher with Network Integrity using the Network Integrity Installer:

1. Log on to Network Integrity.

2. Navigate to the Network Integrity **Installer** screen.

3. Navigate to the **System Wide Shortcut links (Optional)** screen.

4. Enter a name for the link to BI Publisher in the **Name** text box (for example, **BI Publisher**).

5. Enter the URL for the link to BI Publisher in the **Link** text box.

6. Click **Next** and follow the Wizard instructions.

Uploading and Exporting Reports

Oracle BI Publisher provides the following methods to upload and export reports:

- Exporting a Report
- Uploading a Report

Exporting a Report

To export a BI Publisher report, use the following steps:

1. Log on to BI Publisher as administrator.
2. Click the **Admin** tab.
3. In the **System Maintenance** section, click **Report Repository**.
4. Change the **Path** if required.
5. Copy the path and paste it into the Explorer Address bar, or an equivalent software.
6. Navigate to User folders.
7. Select the user name associated to the report you want to export. There are *n* folders for *n* reports in that user folder. These reports can be zipped and stored in any location for future use.

Uploading a Report

To upload a BI Publisher report, use the following steps:

1. Log on to BI Publisher as administrator.
2. Click the **Reports** tab.
3. Click **My Folders**.
4. Click **Create a new folder**.
5. Click **Create**.
6. Click the newly created folder link.
7. Click **Upload a Report**.
8. Save the reports downloaded from secondary memory or from location where they were stored.
9. Unzip each of the zip files into individual folders.
10. Click browse and navigate to the unzipped folder.
11. Select the file with.xdo extension.
12. Click upload button. The report is created in you folder now.
13. Click the report – "Edit" link and click the "Layouts" link in the left side navigation.
14. Click browse and upload all the RTF files in the unzipped folder.
15. Click Save link in the left navigation.
16. Repeat steps 10 to 15 for all additional unzipped report folders.

14

Working with SOA Extensibility

This chapter provides overview information about Service-Oriented Architecture (SOA) extensibility for Oracle Communications Network Integrity.

This chapter contains the following sections:

- About SOA Extensibility
- Extensibility Tasks

About SOA Extensibility

SOA extensibility topics covered in this chapter include creating an SOA development environment, setup, development, and testing of the Network Integrity SOA application.

The Business Process Execution Language (BPEL) provides enterprises with an industry standard for business-process orchestration and execution. Using BPEL, you design a business process that integrates a series of discrete services into an end-to-end process flow.

The Oracle BPEL Process Manager is a tool for designing and running business processes. This product creates, deploys, and manages cross-application business processes with both automated and human workflow steps in a service-oriented architecture.

The Sample Network Integrity SOA application provides a BPEL process that contains two parallel sequences. These sequences automate search and update Network Integrity discrepancies.

The following shows how this automation occurs:

1. Search for Network Integrity discrepancies of type attribute mismatch for nativeEmsServiceState and update their resolution to **Correct in UIM**, if those discrepancies' network value is **In service** and import value is **Out of service**.

2. Search for Network Integrity discrepancies of type attribute mismatch for physicalAddress and update their priority to **High** and discrepancy owner to given input value.

Purpose of Documentation

The developer should learn to install SOA, setup SOA Development environment, and use it for Network Integrity SOA application extensibility.

Extensibility Tasks

The tasks involve setting up of developer environment to update and extend the Network Integrity SOA application for future requirements.

Required software includes:

1. Oracle WebLogic Server

2. Oracle JDeveloper

3. Oracle Application Development Framework

4. Oracle Application Runtime Framework

5. Oracle Fusion Middleware Repository Creation Utility

6. SOA suite

7. Oracle Database

Extensibility Tasks

To implement SOA extensibility, use the following tasks:

- Installing Oracle Weblogic Server

- Installing Oracle JDeveloper

- Installing Oracle Application Runtime

- Installing Oracle SOA Suite

- Creating SOA Metadata Service Schemas

- Updating JDeveloper for Latest SOA Composite Editor

- Creating WebLogic Domain with SOA Products

- Creating and Updating Sample SOA Application Using Network Integrity Web Service

- Starting and Stopping SOA Servers

- Building and Deploying the SOA Application

- Testing Sample SOA application

- Testing Network Integrity SOA Application Using EM

- Testing Network Integrity SOA Application Using soa-infra

- Testing Network Integrity SOA Application Using SOAP UI Tool

Installing Oracle Weblogic Server

To install Oracle Weblogic Server, use the following procedure:

1. Download Oracle WebLogic Server.

2. Run **./wls1036_linux32.bin**

3. Click **Next**.

4. Enter the *WL_Home* directory location to create a home directory for Oracle Fusion Middleware.

5. Click **Next**.

6. Select the **I wish to receive security updates via Oracle Support** check box and click **Next**. (Optional)

7. Select **Custom** for the installation type.

8. Click **Next**.

9. Select the **WebLogic Server** check box to install all WebLogic Server components.

10. Click **Next**.

11. Select the **Sun JDK** check box.

12. Click **Next**.

13. Review the installation directories.

14. Click **Next**.

15. Review the installation summary of the products and JDKs to be installed.

16. Click **Next**. This step begins the installation.

17. When the installation is complete, deselect **Run Quickstart**.

18. Click **Done**.

19. Setup **BEA_HOME**, **JAVA_HOME**, **WL_HOME** environment variables and update **PATH** with the Java executable location. For example,

```
export BEAHOME=/opt/beahome
export WL_HOME=$BEAHOME/wlserver_10.3
export JAVA_HOME=$BEAHOME/jdk160_33_R27.6.5-32

export PATH=$JAVA_HOME/bin:$PATH
```

Installing Oracle JDeveloper

To install Oracle JDeveloper, use the following procedure:

1. Download Oracle JDeveloper (**Oracle_JDeveloper_11g_and_Oracle_Application_ Development_Framework_11g.zip**) software from the Oracle software delivery website:

    https://edelivery.oracle.com/

2. Unzip the installer to any directory.

3. Open a console.

4. Change the console directory to the unzipped installer directory.

5. Run the installer using the following command:

    ```
    java -jar jdevstudio11116install.jar
    ```

    The Installer starts extracting the setup files and Installation wizard opens when it reaches to 100%.

6. Click **Next**.

7. Select **Use the existing Middleware Home** to select the Middleware home you created in "Installing Oracle Weblogic Server".

8. Select J**Developer Studio and ADF** too install all JDeveloper Studio and ADF components.

9. Click **Next**.

10. Select the existing **Sun SDK**.

11. Click **Next**.

12. Confirm **JDeveloper** and **WLS home** directories and click **Next**.

13. Review the Installation summary and click **Next**. This step begins the installation.

14. Click **Done** when the installation is complete.

Installing Oracle Application Runtime

To install Oracle Application Runtime, use the following procedure:

1. Download Oracle Application Development Runtime software from the Oracle software delivery website:

   https://edelivery.oracle.com/

2. Unzip the installer to any directory.

3. Open a console.

4. Change the console directory to the unzipped installer directory.

5. Run the installer using the following command:

   ```
   . Disk1/runInstaller
   ```

6. Enter the **JAVA HOME** location to launch installation wizard.

7. Click **Next**.

8. Click **Next** button after Prerequisite Checks are complete.

   > **Tip:** Install the required system package if a check fails.

9. Click **Browse** and navigate to *WL_Home*.

10. Click **Next**.

11. Click **Install**.

12. Click **Next** after the installation is complete.

13. Click **Finish**.

Installing Oracle SOA Suite

To install Oracle SOA Suite, use the following procedure:

1. Download Oracle SOA Suite software from the Oracle software delivery website:

   https://edelivery.oracle.com/

2. Unzip the installer to any directory.

3. Open the console and change to unzipped folder directory.

4. Run the installer using the following command:

   ```
   . Disk1/runInstaller
   ```

5. Enter the **JAVA HOME** location to launch installation wizard.

6. Click **Next**.

7. Click **Next** after Prerequisite Checks are complete.

> **Tip:** Install the required system package if a check fails.

8. Click **Browse** and navigate to *WL_Home*. Do not modify the **Oracle Home Directory** name.

9. Click **Next**.

10. Click **Install**.

11. Click **Next** after the installation is complete.

12. Click **Finish**.

Creating SOA Metadata Service Schemas

To creates a metadate service (MDS) schema for the Business Activity Monitoring (BAM) and SOA servers, use the following procedure:

1. Download Oracle Fusion Middleware Repository Creation Utility software from the Oracle software delivery website:

   https://edelivery.oracle.com/

2. Unzip the Repository Creation Utility (RCU) to any directory.

3. Open the console and change to unzipped folder directory.

4. Run the installer using the following command:

   ```
   ./rcuHome/bin/rcu
   ```

5. Click **Next**.

6. Select **Create** in the **Create Repository** screen

7. Click **Next**.

8. Enter database details as required.

9. Click **Next**.

10. Click **OK**.

11. Select **Create a new Prefix** in the **Select Components** screen and enter a prefix in the text box.

12. Select the following from the **Component** list:

   - Metadata Service
   - SOA Infrastructure
   - Business Activity Monitoring
   - User Messaging Service

   These components are required for the SOA and BAM servers.

   > **Tip:** Remember the **Schema Owners** for subsequent procedures.

13. Click **Next**.

14. Enter passwords for all components in the **Schema Password**s screen.

   > **Tip:** Remember the **Schema Passwords** for subsequent procedures.

15. Click **Next**.

16. Review the **Schema Owner**, **Tablespace Type**, and **Tablespace Name** for each **Component** in the **Summary** screen.

17. Click **Next** to accept the settings.

18. Click **OK** to create the tablespaces.

19. Click **OK** when the prerequisites are complete.

20. Click **Create** in the Summary screen to create the tablespaces. This step can take up to ten minutes.

21. Click **Close** after the tablespaces are created.

Updating JDeveloper for Latest SOA Composite Editor

SOA design time in JDeveloper requires a JDeveloper extension called SOA Composite editor. While this is normally updated over the network when using release-level software, you can also perform the update manually if you have the extension file. JDeveloper periodically prompts you to accept an automatic network update. Since this is released software, you have the option to click OK to update to a newer version.

To update JDeveloper for the latest SOA Composite editor, use the following procedure:

1. Start JDeveloper Studio.

2. Select Default Role.

3. Deselect **Show this dialog every time**.

4. Click **OK**.

5. Click **No** for **Migrate from previous release**. After starting JDeveloper, wait for the Integrated Weblogic Domain to be created. This domain is created the first time you run JDeveloper after installation. It is not used by SOA. Watch for the completion message for setting up the domain in the JDeveloper Messages log window at the bottom of the JDeveloper IDE:

   ```
   [12:37:11 PM] Creating Integrated Weblogic domain...
   [12:38:05 PM] Extending Integrated Weblogic domain...
   [12:38:14 PM] Integrated Weblogic domain processing
    completed successfully.
   ```

   Now you can update the SOA Composite editor extension. These instructions show you how to update the extension over the network.

6. Select **Help | Check For Updates**.

7. Click **Next**.

8. Select **Search Update Centers**.

9. Select **Oracle Fusion Middleware Products**.

10. Click **Next**. The system searches the update center for extensions.

11. From the list of extensions, select **Oracle SOA Composite Editor**.

12. Click **Next** to begin downloading. When the extension finishes downloading, it is listed with the version number detail.

13. Click **Finish**.

14. Restart JDeveloper when prompted.

15. Click **No** for **Migrate from previous release**.

16. When JDeveloper is running again, select **Help** then **About**.

17. Select the Version tab and review the version.

Creating WebLogic Domain with SOA Products

To creates an Oracle WebLogic domain with the required products for SOA applications, use the following procedure:

1. Open the console and change to unzipped folder directory.

2. Run the following command:

```
./<BEAHOME>/wlserver_10.3/common/bin/config.sh
```

3. When the Welcome screen appears, select **Create a new WebLogic domain**.

4. Click **Next**.

5. Select **Generate a domain**, **SOA Suite**, **Enterprise Manager**, and **Business Activity Monitoring**. **Dependent products** are selected automatically.

6. Click **Next**.

7. Enter **domain1** for the domain name.

8. Click **Next**.

9. Enter the user name **weblogic** and a password. The password **welcome1** is assumed in this document.

10. Click **Next**.

11. Select **Sun SDK 1.6_33** and leave **Development Mode** checked.

12. Click **Next**.

13. Select the check boxes for the components that you want to change.

14. Enter **Welcome1** for the **Schema Password**.

15. Change the **Service**, **Host Name**, and **Port** values as required.

16. Click **Next**.

17. Review the Schema Owners for the individual component schemas and confirm that the owners match those selected in the "Creating SOA Metadata Service Schemas" procedure.

> **Tip:** To change the Schema Owner field, use the following steps:
>
> **1.** Remove the check boxes for all **Component Schema** items.
>
> **2.** Select the check box for the **Component Schema** that you want to change.
>
> **3.** Change the **Schema Owner** field.
>
> **4.** Remove the check box for the component schema item you changed.

18. Click **Next** to begin a data source connection test.

19. Click **Next** if all connection tests are successful. If the connection tests are not successful, click **Previous** and correct any errors.

20. Click **Next**.

21. Click **Create** in the **Configuration Summary** screen.

22. Click **Done** when the domain has been created.

When a domain is created, the Configuration Wizard creates one admin server and two managed servers with the following details:

- Admin Server

   Name: admin_server

   Port: 7001

- SOA Server

   Name: soa_server1

   Port: 8001

- BAM Server

   Name: bam_server1

   Port: 9001

See the **startManagedServer_readme.txt** file in the domain folder to start the servers.

Creating and Updating Sample SOA Application Using Network Integrity Web Service

To update an SOA application using the Network Integrity SOA application, use the following procedure:

1. Download the Sample Network Integrity SOA application (**NetworkIntegrity-SOA_Sample_App-***version***.zip**) software from the Oracle software delivery website:

   https://edelivery.oracle.com/

2. Unzip the application to any directory.

3. Start **Oracle Jdeveloper**.

4. From the **Jdeveloper** main menu, choose **File** then **Open** then browse to **NISOAApplication** folder and select **NISOAApplication.jws**.

5. Click **Open**.

The **NISOAApplication.jws** contains the **NIDiscrepancyService** project. The main components for this project are:

- **NetworkIntegrityControlService.wsdl**: This is the Network Integrity Sample Web Services WSDL file.

- **xds**: This folder contains Network Integrity Sample Web Service schema files.

- **composite.xml**: This file describes the entire composite assembly of services, service components, references, and wires

   In the project, composite.xml file is automatically created when the SOA project was created. In this application only service components (including Network Integrity Sample Web Service) are used.

- **NIBPELDiscrepancyProcess.bhel**: This file contains a list of variables and the main sequences in which he Network Integrity Web Service calls to update the Network Integrity Discrepancies are defined. There are two parallel sequences named as **Sequence_1** and **Sequence_2** to update Attribute mismatch discrepancies for **nativeEMSServiceState** (go to step 6) and **physicalAddress** (go to step 8) respectively.

It is necessary that both client side artifacts (wsdl and schema) and server side artifacts are in sync and of same version.

6. To search for **natieEMSServiceState** attribute mismatch discrepancies (**Sequence_ 1**), search for the following discrepancies:

   ■ **TYPE = ATTRIBUTE_VALUE_MISMATCH**

   ■ **ATTRIBUTEORRELATIONSHIPNAME = nativeEmsServiceState**

   ■ **STATUS = DISCREPANCY_OPENED**

   ■ **COMPARESOURCE = INVENTORY**

   ■ **REFERENCESOURCE = NETWORK using findDiscrepancy webservice operation.**

7. Loop over each discrepancy and submit to **updateDiscrepancy** if **COMPAREVALUE = 'IN_SERVICE'** and **REFERENCEVALUE ='OUT_OF_ SERVICE'** to update **OPERATION** as **'Correct in UIM'** and **STATUS** as **'OPERATION_IDENTIFIED'**.

8. To search for physicalAddress attribute mismatch discrepancies, search for the following discrepancies:

   ■ **TYPE = ATTRIBUTE_VALUE_MISMATCH**

   ■ **ATTRIBUTEORRELATIONSHIPNAME = physicalAddress**

   ■ **STATUS = DISCREPANCY_OPENED using findDiscrepancy webservice operation.**

9. Loop over each discrepancy and submit to **updateDiscrepancy** by setting **PRIORITY** to **High** and **DISCREPANCYOWNER** to given value.

10. Right-click **composite.xml** and select **Configure WS Policies** to add appropriate security client policy to the Network Integrity Web Service component.

11. Update NetworkIntegrityControlService.wsdl's SOAP address location with Network Integrity Web Service URL. For example:

```
<soap:address location="https://<host_address>:<ssl_
port>/NetworkIntegrityApp-NetworkIntegrityControlWebService-context-root/Networ
kIntegrityControlServicePortType"/>
```

This should be done before building the SOA application or use deployment plan while deploying the SOA application to update the SOAP address location with the Network Integrity Web Service URL. This configuration is required for SOA application to communicate with Network Integrity Web Services.

Starting and Stopping SOA Servers

To start and stop SOA servers, use the following procedure:

1. To start the Administration Server run to following command: **<domain>/startWeblogic.sh**

2. To start the SOA managed server, run the following command (here soa_server1 is name of SOA managed server): **<domain>/bin/startManagedServer.sh soa_ server1**

3. To enter the WebLogic console, use:

```
http://Host_Address:7001/console
```

4. To enter the Enterprise Manager console, use:

   ```
   http://Host_Address:7001/em
   ```

5. To enter SOA Infra, use:

   ```
   http://Host_Address:8001/soa-infra
   ```

6. Press **Ctrl + C** to stop the servers.

Building and Deploying the SOA Application

To build and deploy the SOA application, use the following procedure:

1. In Jdeveloper, go to Application Navigator then right-click **NIDiscrepancyService** project.

2. Click **Make NIDiscrepancyService.jpr** in the menu to build the project. The project should build successfully without any compilation errors or warnings.

3. Start the Administration and SOA servers that are created as part SOA domain creation (see "Starting and Stopping SOA Servers" and "Creating WebLogic Domain with SOA Products").

4. Create a standalone server connection for the SOA server.

5. Right-click **NIDiscrepancyService** and select **'Deploy' to Application server**.

6. The SOA suite provides an ant script to deploy and undeploy the SOA archive (SAR) file (deployable SOA application jar) in the BEA HOME. Use the following to deploy and undeploy the SAR file:

   - To deploy, use the following:

     ```
     ant -f <BEAHOME>/Oracle_SOA1/bin/ant-sca-deploy.xml
     -DserverURL=<http://soa_server_host:soa_server_port>
     -DsarLocation=<SOA archive file path>
     ```

     For example,

     ```
     ant -f /home/beahome/Oracle_SOA1/bin/ant-sca-deploy.xml
     -DserverURL=http://<localhost>:8001
     -DsarLocation=/home/example/beahome/mywork/NISOAApplication/NIDiscrepancySe
     rvice/deploy/sca_NIDiscrepancyComposite_rev1.0.jar
     ```

   - To undeploy, use the following:

     ```
     ant -f <BEAHOME>/Oracle_SOA1/bin/ant-sca-deploy.xml undeploy
     -DserverURL= <http://soa_server_host:soa_server_port>
     -DcompositeName=<SOA composite name>
     -Drevision=<SOA composite version>
     ```

     For example,

     ```
     ant -f /home/beahome/Oracle_SOA1/bin/ant-sca-deploy.xml undeploy
     -DserverURL=http://<localhost>:8001
     -DcompositeName=NIDiscrepancyComposite
     -Drevision=1.0
     ```

Testing Sample SOA application

To test a sample SOA application, use the following three tools:

- [Testing Network Integrity SOA Application Using EM](#)
- [Testing Network Integrity SOA Application Using soa-infra](#)
- [Testing Network Integrity SOA Application Using SOAP UI Tool](#)

> **Note:** Oracle Enterprise Manager (EM) can also helpful in debugging and auditing of BPEL sequence exceptions.

Testing Network Integrity SOA Application Using EM

To test a sample SOA application with EM, use the following procedure:

1. Log on to the Enterprise manager as admin.

2. Expand the SOA folder to the deployed composite (**NIDiscrepancyComposite**).

3. Click **Test** to test composite.

4. Enter any value for the input argument for SOA Web Service.

5. Click **Test Webservice**. Wait for a response.

6. Click **Launch Message Flow Trace** to see detailed output.

7. Click **NIBPELDiscrepancyProcess** to view the Audit Trail, Flow, and so on.

8. Expand the payloads to see detailed input and output of each Web Service invoked.

Testing Network Integrity SOA Application Using soa-infra

To test a sample SOA application with soa-infra, use the following procedure:

1. Log on to soa-infra using the following URL:

   ```
   http://Host_Address:8001/soa-infra
   ```

2. Enter any input required for the test.

3. Click **Invoke**.

Testing Network Integrity SOA Application Using SOAP UI Tool

To test a sample SOA application with the Simple Object Access Protocol (SOAP) UI tool, use the following procedure:

1. Create a SOAP UI project at the following URL:

   ```
   http://Host_
   Address:8001/soa-infra/services/default/NIDiscrepancyComposite/nibpeldiscrepanc
   yprocess_client_ep?WSDL
   ```

2. Enter any input required for the test.

3. Create a request run.

15

Localizing Network Integrity

This chapter provides information on localizing the Oracle Communications Network Integrity UI and Help. Localization is the process of translating a UI or Help system from the original language in which it was written into a different language for use in a specific country or region. For example, the Network Integrity UI and Network Integrity Help are written in English. If your company is based in France and you purchase Network Integrity, you may want to localize Network Integrity to display the UI and Help in French.

Localizing Network Integrity involves modifying a specific set of files that Network Integrity uses to display text in the UI and in the Help.

This chapter contains the following sections:

- Software Requirements
- Setting the Language Preference in Internet Explorer
- Determining the Locale ID
- Localizing Network Integrity
- Localizing Network Integrity Help

> **Note:** The procedures in this chapter use Windows syntax for directory paths and commands. If you are working on a Unix or Linux platform, adapt the syntax accordingly.

> **Note:** Before localizing your Network Integrity environment, you must identify a strategy for maintaining future localizations. Oracle does not provide a delta file in which you can readily see the details of what changed between releases.

Software Requirements

The following software is required to localize Network Integrity:

Design Studio

Localizing the Network Integrity UI involves working with the Network Integrity localization pack that you import into Oracle Communications Design Studio, modify, and deploy into Network Integrity. Design Studio also provides various editors, such as an XML editor and an HTML editor, that you can use to translate files for localization.

Help Indexer

Localizing the Network Integrity Help involves regenerating a search index file based on translated HTML files. This is accomplished using Oracle Help Indexer.

To access Help Indexer, download the **help-indexer.jar** file from the following website to a local directory such as *tempDir*:

http://www.oracle.com/technetwork/topics/utilsoft-085729.html

Java

Using Help Indexer requires that you have Java installed. The java command should be in your path.

Setting the Language Preference in Internet Explorer

For a localized version of Network Integrity to display correctly in Internet Explorer, users need to configure language preferences.

To configure language preferences in Internet Explorer:

1. From the **Tools** menu, select **Internet Options**.

   The Internet Options window appears.

2. Click **Languages**.

   The Language Preferences window appears.

3. The language you plan to use must display at the top of the list to have priority.

   If the language you plan to use is listed:

   a. Select the language.

   b. Click **Move Up** or **Move Down** to place the language you plan to use at the top of the list.

   If the language you plan to use is not listed:

   a. Click **Add**.

      The Add Language window appears.

   b. Select a language.

   c. Click **OK**.

      The Language Preference window returns.

   d. Select the language you have added, and click **Move Up** to move it to the top of the list.

4. Click **OK**.

Determining the Locale ID

A locale ID is a standardized ID that represents a language and region in which the language is spoken. For example, **fr_CA** is the locale ID for French spoken in Canada, and **es_MX** is the locale ID for Spanish spoken in Mexico.

Localizing Network Integrity involves copying and renaming existing files to include a locale ID. The renamed files that include a locale ID become the translated version of the original files.

To determine the locale ID:

1. From Internet Explorer, select **Tools**, then select **Internet Options**.

   The Internet Options window appears.

2. Click **Languages**.

   The Languages window appears.

3. Click **Add**.

   The Add Language window appears.

   Languages are listed alphabetically. Several languages are spoken in more than just one country, so the locale ID reflects the language and the country in which the language is spoken. For example, there multiple locale IDs for French:

   - fr-BE for French spoken in Belgium

   - fr-CA for French spoken in Canada

   - fr-FR for French spoken in France

   - fr-LU for French spoken in Luxembourg

   - fr-MC for French spoken in Monaco

   - fr-CH for French spoken in Switzerland

4. Locate the language to which you are localizing and determine the appropriate locale ID.

5. Close the Add Language, Languages, and Internet Option windows.

Localizing Network Integrity

The following sections describe localizing Network Integrity:

- About the Localization Pack

- Creating the Localization Pack

- Deploying the Cartridge Containing the Localized Files

- Testing the Network Integrity Localization

About the Localization Pack

The Network Integrity UI makes use of the full depth of i18n support provided by the Application Development Framework (ADF) stack. The application UI is fully internationalized by making use of XML Localization Interchange File Format (XLF) files to keep all display strings separate from other code artifacts. Various parts of the ADF stack (ADF Faces, ADF Model, and ADF Data Control) are also built with full i18n support. A localization pack is a collection of XLF files and other property files, that together localize the UI to another language. A localization pack can be built into a cartridge that can be deployed into Network Integrity.

The expected outcome is that the user can successfully create, build, and deploy a localization pack.

Creating the Localization Pack

Use the following procedure to create a localization pack:

1. Download **localization.iar** from the localization pack in the Oracle Communications Network Integrity 7.3.2 Software Developer Kit (included with

the Oracle Communications Network Integrity 7.3.2 software) on the Oracle
software delivery website:

https://edelivery.oracle.com

> **Note:** The localization pack also contains a partial sample traditional
> Chinese localization, for your reference, where parts of the Scan
> Configuration Creation page are translated into traditional Chinese.

2. Extract the **META-INF/MANIFEST.MF** file to a temporary location.

3. Open **MANIFEST.MF** and edit the value of **Bundle-Name: Localization** and
   Bundle-Description: Localization as follows:

```
Bundle-Name: Localization : localization_pack_name
Bundle-Description: Localization : localization pack description
```

Where *localization_pack_name* is the name of the localization pack you are creating,
and where *localization pack description* describes the localization pack you are
creating.

4. Save **MANIFEST.MF** and return it to **localization.iar/META-INF**.

5. Extract **META-INF/cartridge.xml** to a temporary location.

6. Open **cartridge.xml** and edit the values of the **name** and **languageCode** tags:

```
<localizations>
   <localization>
      <name>Locale_Name</name>
      <languageCode>Locale_ID</languageCode>
   </localization>
</localizations>
```

Where *Locale_Name* is the locale of the localization pack you are creating; for
example, **French**, and where *Locale_ID* is the standardized locale ID that represents
a language and region in which the language is spoken. For example, **fr-CA** is the
locale ID for French spoken in Canada, and **es-MX** is the locale ID for Spanish
spoken in Mexico. A locale ID can also represent a language without specifying
the region in which the language is spoken. For example:

```
<localizations>
   <localization>
      <name>French</name>
      <languageCode>fr</languageCode>
   </localization>
</localizations>
```

7. Save **cartridge.xml** and return it to **localization.iar/META-INF**.

8. Extract **localization.iar/localization.jar** to a temporary location.

9. Extract **localization.jar/oracle** to a temporary location.

10. Edit all the XLF files found in **localization.jar/oracle** or any of its nested folders:

   a. Edit the name of each XLF file to add an underscore and the locale ID before
      the file extension, as shown in the following example:

      DisAddressMsgBundle_**fr**.xlf

> **Note:** Compound locale IDs, such as fr-CA, should be added to the XLF file name with an underscore in the place of the hyphen, as in the following example:
>
> DisAddressMsgBundle_**fr_CA**.xlf

b. Open each XLF file and edit the **file** tag so that the **source-language** attribute is set to the locale ID, as in the following example:

```
<file source-language="fr"
original="oracle.communications.inventory.api.entity.PhysicalPortMsgBundle"
datatype="xml">
```

> **Note:** The **source-language** attribute for compound locale IDs, such as fr-CA, should be set to the first two characters only, as in the following example:
>
> ```
> <file source-language="fr"
> original="oracle.communications.inventory.api.entity.PhysicalPortMs
> gBundle" datatype="xml">
> ```

c. Open each XLF file, locate each **trans-unit** tag and edit its child **source** tag with the translated value for the desired localization.

11. Edit all the PROPERTIES files found in **localization.jar**/**oracle** or any of its nested folders:

a. Edit the name of each PROPERTIES file to add an underscore and the locale ID before the file extension, as shown in the following example:

IntegrityUIBundle_**fr**.properties

> **Note:** Compound locale IDs, such as fr-CA, should be added to the XLF file name with an underscore in the place of the hyphen, as in the following example:
>
> IntegrityUIBundle_**fr_CA**.properties

b. Open each PROPERTIES file and edit the value for each key with the translated value for the desired localization. For example, edit the INTEGRITY_MANAGE_SCAN_CONFIG key, as in the following example:

```
INTEGRITY_MANAGE_SCAN_CONFIG=new_value
```

Where *new_value* is the translated value for the key for the desired localization.

c. (Optional) To enter extended character values (such as Chinese characters), you must use Unicode Escapes (only one character is allowed per escape sequence). Save each PROPERTIES file with UTF-8 encoding, then convert each PROPERTIES file to Unicode Escapes using the native2ascii tool provided with your JDK by entering the following command:

```
native2ascii -encoding UTF-8 input_file_name output_file_name
```

Where *input_file_name* is the name of the PROPERTIES file being converted, and where *output_file_name* is the name of the converted file.

See the partial sample Chinese localization included in the localization pack for an example.

12. Return all XLF and PROPERTIES files to **localization.jar**.

13. Return **localization.jar** to **localization.iar**.

14. Deploy **localization.iar** using the cartridge deploy tool.

15. (Optional) To localize link names in the **Link** panel in the Network Integrity UI, you must edit the MBean with the translated values for the desired localization. See *Network Integrity System Administrator's Guide* for more information about viewing and editing the MBean.

16. (Optional) To localize cartridge-specific scan parameters, see the Design Studio Help. Cartridge-specific scan parameters can be localized within Design Studio, where you can set multiple language preferences and then assign a language preference to a scan parameter group.

Deploying the Cartridge Containing the Localized Files

After the translations are complete, build the localization pack to create a cartridge that can be deployed into Network Integrity. Every cartridge should be cleaned and rebuilt prior to deploying.

See the Design Studio Help and the *Network Integrity Installation Guide* for more information about deploying cartridges.

> **Note:** When a cartridge containing localizable XLF files is deployed into Network Integrity, the **NetworkIntegrity.ear** file automatically redeploys, resulting in the localization changes being applied to the UI.

Testing the Network Integrity Localization

When running the Network Integrity UI, the user chooses the appropriate language from the web browser. This is usually done using the Character or Text Encoding menu of the browser, or from a Language preference setting. The UI displays the selected language after the corresponding localization pack is deployed. Otherwise, the UI displays the default English language.

There may be parts of the UI that are supplied by third parties, which are not fully internationalized. Those parts always display in English.

Localizing Network Integrity Help

The following sections describe localizing Network Integrity Help:

- About Network Integrity Help
- Localizing the Network Integrity Help Files
- Deploying the Localized Help System
- Testing the Network Integrity Help Localization

About Network Integrity Help

Network Integrity Help uses Oracle Help for the Web. Oracle Help is a browser-based Help system that runs as a web application based on a Java servlet. You do not need

specialized knowledge of Oracle Help to localize Network Integrity Help; you can use the information in this chapter, supplemented by the Oracle Help documentation. See *Oracle Fusion Middleware Developer's Guide for Oracle Help* for more information.

Network Integrity Help consists of a set of files, as described in the following sections.

About the Help Files

This section provides information about the Help files, including their location, a brief description of their purpose, and whether or not they require configuring or translating for localization. For details about configuring or translating the content of the Help files, see "Localizing the Network Integrity Help Files".

Oracle Help File

An Oracle Help configuration file is located in the *NI_Home*/**integrity**/**NetworkIntegrity.ear**/**NetworkIntegrityApp_NetworkIntegrityUI_webapp1.war**/**helpsets** directory. The **ohwconfig.xml** configuration file contains references to each Help system deployed into an application. Upon installation, the **ohwconfig.xml** file references the default Network Integrity Help system (English) deployed into Network Integrity. This file requires configuration for localization.

Network Integrity Help Files

The Network Integrity Help files are located in the *NI_Home*/**integrity**/**NetworkIntegrity.ear**/**NetworkIntegrityApp_NetworkIntegrityUI_webapp1.war**/**WEB-INF/lib/Network_Integrity_Help.jar** file, which contains the following Help files:

- ***.htm files**: Each HTML file is a separate Help topic. The text in all of the HTML files requires translation.

- **Network_Integrity_Help.hs**: This file describes the Help system. When Network Integrity Help is initiated through the Network Integrity UI, **Network_Integrity_Help.hs** is the starting point. This file does not require translation.

- **toc.xml**: This file defines the Table of Contents (TOC) that appears in the left pane of the Oracle Help window. The text in this file requires translation.

- **map.xml**: This file associates Help IDs with the HTML file names. The TOC uses the IDs to link entries to Help topics. This file does not require translation.

- **search.idx**: This file is used when you perform a text search of the Help content. The file defines a search index that searches the Help content in the HTML files. After the HTML files are translated, the search index must be regenerated using the Java-based Help Indexer. For more information, see "Software Requirements" and "Regenerating the Search Index File".

- **target.db**: This file contains cross-reference information used for navigating between Help topic headings. This file does not require translation.

- **dcommon/html/cpyr.htm**: This file defines the Help copyright page, and requires translation. (The **dcommon** directory contains standard Oracle support files, including a CSS file, several graphics files, and the Help copyright page, but only the Help copyright page requires translation.)

Localizing the Network Integrity Help Files

To localize Network Integrity Help, perform the work described in the following sections:

- Extracting the Help Files

- [Translating the Help Files](#)
- [Regenerating the Search Index File](#)
- [Creating the Localized Help JAR File](#)
- [Configuring the Oracle Help File](#)

Extracting the Help Files

Use the default Help system installed with Network Integrity as the starting point for your localization.

To extract the Help files:

1. Copy the *NI_Home*/**integrity**/**NetworkIntegrity.ear**/**NetworkIntegrityApp_NetworkIntegrityUI_webapp1.war**/**WEB-INF**/**lib**/**Network_Integrity_Help.jar** file to a local directory, such as *tempDir*.

2. Open the *tempDir*/**Network_Integrity_Help.jar** file.

3. Select all objects in the **Network_Integrity_Help.jar** file and extract them into the same directory in which the **Network_Integrity_Help.jar** file resides (*tempDir*).

4. Click the **File** column heading in the *tempDir* directory to sort the objects by file type.

   The following objects are present:

   - **dcommmon** directory
   - **img** directory
   - **META-INF** directory
   - **target.db**
   - **Network_Integrity_Help.jar**
   - **Network_Integrity_Help.hs**
   - numerous *****.htm** files
   - **search.idx**
   - **map.xml**
   - **toc.xml**

You do not need to do anything with the **img** or **META-INF** directories, or with the **target.db**, **Network_Integrity_Help.hs**, or **map.xml** files.

Translating the Help Files

To translate the Help files, perform the translations described in the following sections:

- [Translating the Copyright Page](#)
- [Translating the Help Topics](#)
- [Translating the Table of Contents](#)

Translating the Copyright Page

The copyright page text is defined in the *tempDir*/**dcommon**/**html**/**cpyr.htm** file. Translate the content of the title, heading, and paragraph elements (<title>, <h1> - <h6>, <p>) to the local language.

For example, translate the bolded content in [Example 15–1](#):

Example 15–1   Excerpt from cpyr.htm

```
<title>Oracle Legal Notices</title>
<link rel="stylesheet" href="../css/blafdoc.css" type="text/css" />
</head>
<body>
<h1>Oracle Legal Notices</h1>

<h2>Copyright Notice</h2>
<p>Copyright &copy; 1994-2012, Oracle and/or its affiliates. All rights
reserved.</p>
```

Translating the Help Topics

The Help topics text is defined in the numerous *tempDir*/***.htm** files, and each file
requires translating.Translate the content of the title, heading, paragraph, and table
data elements (<title>, <h1> - <h6>, <p>, <td>) to the local language.

For example, translate the bolded content in Example 15–2. Elements that are not text,
such as the HTML tags themselves, should not be changed.

Example 15–2   Excerpt from olh_integ_scans002.htm

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
<meta name="OAC_IGNORE_SKIP_NAV" content="true" />
<meta http-equiv="Content-Type" content="text/html; charset=us-ascii" />
<meta http-equiv="Content-Style-Type" content="text/css" />
<meta http-equiv="Content-Script-Type" content="text/javascript" />
<title>Creating a Scan</title>
<meta name="generator" content="Oracle DARB XHTML Converter (Mode = ohj/ohw) -
Version 5.1.2 Build 040" />
<meta name="date" content="2011-12-20T20:51:30Z" />
<meta name="robots" content="noarchive" />
<meta name="doctitle" content="Creating a Scan" />
<meta name="relnum" content="Release 7.1" />
<meta name="partnum" content="E23703-01" />
<meta name="topic-id" content="CreateScansMain" />
<link rel="copyright" href="./dcommon/html/cpyr.htm" title="Copyright"
type="text/html" />
<link rel="stylesheet" href="./dcommon/css/blafdoc.css" title="Oracle BLAFDoc"
type="text/css" />
<link rel="contents" href="toc.htm" title="Contents" type="text/html" />
</head>
<body>
<p><a id="CJAJAIGJ" name="CJAJAIGJ"></a><a id="CreateScansMain"
name="CreateScansMain"></a></p>
<div class="sect2">
<h1>Creating a Scan</h1>
<p>To create a scan:</p>
<ol>
<li>
<p>From the Tasks panel, click <span class="bold">Manage Scans</span>.</p>
<p>The Manage Scans page appears.</p>
</li>
```

Translating the Table of Contents

The TOC text is defined in the *tempDir*/**toc.xml** file. Each item in the TOC is defined
by a <tocItem> element. Translate the content to the local language.

For example, translate the bolded content of the text attribute in Example 15–3. Do not change the content of the target attribute.

Example 15–3   Excerpt from toc.xml

```
<tocitem target="olh_integ_main001.htm-sthref3" text="Getting Started with Network
Integrity" />
```

> **Note:**   Oracle Help automatically translates the Help window menu options, field names, and informational, warning, and error messages. The translation is based on the locale defined in the **ohwconfig.xml** file.
>
> For example, if the only language preference specified is English, and the **ohwconfig.xml** file defines a single locale of French, Oracle Help translates the Help window menu options, field names, and messages to French.
>
> That said, Oracle recommends that the language preference with the highest priority be the same language defined as the locale in the **ohwconfig.xml** file.

Regenerating the Search Index File

After translating the Help files, regenerate the search index file to reflect the content of the translated files.

To regenerate the search index file:

1. Download the **help-indexer.jar** file to a local directory, such as *tempDir*, and install Java.

    See "Software Requirements" for more information.

2. Open a Windows command prompt.

3. Change the directory to the directory containing the translated Help files by entering the following command:

    ```
    cd tempDir
    ```

4. Enter the following command, which creates a new **search.idx** file, and overwrites the existing **search.idx** file.

    ```
    java -mx64m -classpath pathToJarFile/help-indexer.jar
    oracle.help.tools.index.Indexer -l=locale -e=charSet pathToHelpFiles search.idx
    ```

    For example:

    ```
    java -mx64m -classpath C:\tempDir\help-indexer.jar
    oracle.help.tools.index.Indexer -l=fr_CA -e=8859_1 C:\tempDir search.idx
    ```

    In the above command:

    -   **java** is the java command in your path. If it is not in your path, specify the full path to **java.exe**.

    -   **-mx64m** increases the default heap size. This is optional, but the Oracle Help documentation recommends that you specify it.

    -   **-classpath** specifies the location of the **help-indexer.jar** file.

- **oracle.help.tools.index.Indexer** is the Java class that is called to regenerate the search index. This Java class file resides in the **help-indexer.jar** file.

- **-l** specifies the *locale*, a standardized ID that represents a language and region in which the language is spoken. For example, **fr-CA** is the locale ID for French spoken in Canada, and **es-MX** is the locale ID for Spanish spoken in Mexico. If you do not specify a locale, the system's default locale is used.

    See "Determining the Locale ID" for more information.

- **-e** specifies the Java-supported character set encoding (*charset*). For example, 8859_1.

- *pathToHelpFiles* is the full path to the translated Help files.

- **search.idx** is the name of the search index file that Network Integrity Help uses.

See *Oracle Fusion Middleware Developer's Guide for Oracle Help* for more information about these parameters.

Creating the Localized Help JAR File

After translating the Help files and regenerating the search index, create a new JAR file containing the localized Help files.

To create the new JAR file:

1. In Windows Explorer, navigate to the *tempDir* directory. This is the directory containing the **Network_Integrity_Help.jar** file, the translated Help files, and the regenerated search index file.

2. Copy the **Network_Integrity_Help.jar** file, and paste it in the same directory (*tempDir*).

3. Select the copied version of the **Network_Integrity_Help.jar** file and rename it **Network_Integrity_Help.jar_*locale*.jar**, where *locale* is the standardized ID that represents a language and region in which the language is spoken. For example, **fr-CA** is the locale for French spoken in Canada, and **es-MX** is the locale for Spanish spoken in Mexico.

    For more information, see "Determining the Locale ID".

4. Open the **Network_Integrity_Help_*locale*.jar** file.

5. Select and delete all of the objects in the JAR file.

6. Add the localized Help files to the **Network_Integrity_Help_*locale*.jar** file. (This includes all of the directories and all of the files in *tempDir*, with the exception of **Network_Integrity_Help.jar** and **Network_Integrity_Help_*locale*.jar**.

7. Save and close the **Network_Integrity_Help_*locale*.jar** file.

You can verify that you included all of the directories and files by checking the number of objects in the **Network_Integrity_Help.jar** file and in the **Network_Integrity_Help_*locale*.jar** file; the two JAR files should contain the same number of objects. To determine the number of objects in each JAR file, select all of the objects in each JAR file; this provides a count of all objects selected.

Configuring the Oracle Help File

After translating the Help files, regenerating the search index, and creating a localized Help JAR file, configure the *NI_Home*/**integrity**/**NetworkIntegrity.ear**/

**NetworkIntegrityApp_NetworkIntegrityUI_webapp1.war/
helpsets/ohwconfig.xml** file to reflect the localized Help JAR file.

To configure the **ohwconfig.xml** file:

1. Open the **ohwconfig.xml** file.

   The file defines the default Help system (English):

```
<locales>
    <!-- English: -->
    <locale language="en">
        <books>
            <helpSet id="integrity"
                    jar="../WEB-INF/lib/Network_Integrity_Help.jar"
                    location="Network_Integrity_Help.hs"/>
        </books>
    </locale>
</locales>
```

2. Update the <locale> element to reflect the localized Help system:

```
<locales>
    <!-- French Canadian: -->
    <locale language="fr">
        <books>
            <helpSet id="integrity_fr_ca"
                    jar="../WEB-INF/lib/Network_Integrity_Help_fr_ca.jar"
                    location="Network_Integrity_Help.hs"/>
        </books>
    </locale>
</locales>
```

You do not need to change the location attribute value, which is the name of the file
that resides in the specified JAR file.

About Multiple Locales

Oracle Help can support multiple locales. For multiple locales, each localized Help
system is configured with a <locale> element in the **ohwconfig.xml** file. For example,
the following results in both French and Spanish Help systems being available in
Network Integrity upon redeployment:

```
<locales>
    <!-- French: -->
    <locale language="fr">
        <books>
            <helpSet id="integrity_fr_ca"
                    jar="../WEB-INF/lib/Network_Integrity_Help_fr_ca.jar"
                    location="Network_Integrity_Help.hs"/>
        </books>
    </locale>
</locales>
<locales>
    <!-- Spanish: -->
    <locale language="es">
        <books>
            <helpSet id="integrity_es_mx"
                    jar="../WEB-INF/lib/Network_Integrity_Help_es_mx.jar"
                    location="Network_Integrity_Help.hs"/>
        </books>
    </locale>
```

```
</locales>
<parameters>
    <combineBooks>false</combineBooks>
    <useLabelInfo>true</useLabelInfo>
    <cacheSize>3</cacheSize>
</parameters>
```

When multiple locales are defined, the language preference for all locales must be set. If not set, only the first locale defined in the **ohwconfig.xml** file displays in Network Integrity Help. See "Setting the Language Preference in Internet Explorer" for more information.

When multiple locales are defined, the <parameters> element configuration values are applied:

■ <combineBooks>

To merge Help systems, set <combineBooks> to **true**. The Help navigational views behave as a single, integrated Help system.

To use separate Help systems, set <combineBooks> to **false**. The separate Help navigational views are accessed based on the language preference with the higher priority.

Regardless of the <combineBooks> value, each locale that is defined in the **ohwconfig.xml** file must be specified as a language preference. See "Setting the Language Preference in Internet Explorer" for more information.

> **Note:** Oracle Help automatically translates the Help window menu options, field names, and informational, warning, and error messages. The translation is based on the first locale defined in the **ohwconfig.xml** file.
>
> For example, if the only language preference specified is English, and the **ohwconfig.xml** file defines the locales of French and Spanish, Oracle Help translates the Help window menu options, field names, and messages to French.
>
> However, when multiple locales are defined, the language preference for all locales must be specified. Otherwise, only the first locale defined in the **ohwconfig.xml** file displays in Network Integrity Help. So, when the language preferences are set, Oracle Help translates the Help window menu options, field names, and messages to the language preference with the highest priority.

■ <useLabelInfo>

If <useLabelInfo> is set to **true**, author-defined labels are used for the navigators of merged Help systems.

If <useLabelInfo> is set to **false**, default labels such as Contents, Index, and Search are used for the navigators of merged Help systems.

■ <cacheSize>

<cacheSize> indicates the number Help systems kept in memory at one time. The default value is 3.

See *Oracle Fusion Middleware Developer's Guide for Oracle Help* for more information.

Deploying the Localized Help System

The original Help system, located in the *NI_Home*/**integrity**/**NetworkIntegrity.ear**/ **NetworkIntegrityApp_NetworkIntegrityUI_webapp1.war**/**WEB-INF**/**lib**/ **Network_Integrity_Help.jar** file, is deployed when you deploy the **NetworkIntegrity.ear** file.

To deploy the localized Help system:

1. Repackage the *NI_Home*/**integrity**/**NetworkIntegrity.ear** file to include the localized Help files. To do this:

   a. Delete the *NI_Home*/**integrity**/**NetworkIntegrity.ear**/**NetworkIntegrityApp_ NetworkIntegrityUI_webapp1.war**/**WEB-INF**/**lib**/**Network_Integrity_ Help.jar** file.

   b. Copy the *tempDir*/**Network_Integrity_Help.jar**_locale_**.jar** file to the *NI_Home*/ **integrity**/**NetworkIntegrity.ear**/**NetworkIntegrityApp_ NetworkIntegrityUI_webapp1.war**/**WEB-INF**/**lib** directory.

   > **Note:**   If your Network Integrity Help is supporting multiple locales, each JAR file defined by each <locale> element in the **ohwconfig.xml** file must be present in the *NI_Home*/**integrity**/**NetworkIntegrity.ear**/ **NetworkIntegrityApp_NetworkIntegrityUI_webapp1.war**/ **WEB-INF**/**lib** directory.

2. Deploy the repackaged **NetworkIntegrity.ear** file.

   For instructions on how to deploy the **NetworkIntegrity.ear** file, see *Network Integrity System Administrator's Guide*.

Testing the Network Integrity Help Localization

After you deploy the localized Help system, test your Network Integrity environment to verify that the localized Help system is working correctly.

In Network Integrity, open the Help. Tests should include the following:

- Navigate to several topics from links in the Table of Contents to ensure that the correct topics appear and display correctly.

- Test several links within Help topics to ensure they are working.

- Search for several terms and verify that you get the expected results.

- If testing multiple locales that function as a single Help system, verify translations for all locales.

- If testing multiple locales that function as separate Help systems, change the language preference priority to verify translations for each locale.

A

Network Integrity Plug-in Validation Error Messages

This appendix provides information about the Oracle Communications Network Integrity plug-in validation error messages.

This appendix contains the following sections:

- Error Message Classifications and Conditions
- Design Studio Logging

Error Message Classifications and Conditions

Table A–1 lists the error messages, error classifications, and error conditions for the Network Integrity plug-in.

> **Note:** Text inside *{}* represents a variable that is replaced based on the current error condition.

Table A–1   Network Integrity Error Message, Classification, and Error Condition

Error Message	Classification	Error Condition
Action names must start with a letter.	Error	Occurs when you create an action without a letter as the first character in the name.
The character *{character}* is not valid in an implementation prefix.	Error	Occurs when the implementation prefix of an action or processor contains characters that cannot be part of a Java identifier.
Processor *{processor name}* already has more than one parent action assigned.	Error	Occurs if an attempt is made to associate a processor to a second action. A processor can have only one parent.
Processor Parameter: *{parameter name}* not found in Parameter list for Processor {processor name}.	Informational	Occurs if an attempt is made to rename an input or output parameter of a processor, which no longer exists in the Parameter list.
Processor property group: *{property group name}* not found in property group list for Processor *{processor name}*.	Informational	Occurs if an attempt is made to rename a property group of a processor, which no longer exists in the property group list.
Action condition *{condition name}* not found in condition list for action *{action name}*.	Informational	Occurs if an attempt is made to rename a condition of an action, which no longer exists in the condition list.

Table A–1 (Cont.) Network Integrity Error Message, Classification, and Error Condition

Error Message	Classification	Error Condition
The generated implementation prefix for this entity conflicts with the implementation prefix of entity \"*{entity name}*\". Choose a different name.	Error	Occurs when an implementation prefix of an action or a processor conflicts with an existing prefix.
Cannot get cartridge from action: *{action name}*.	Error	Occurs when Oracle Communications Design Studio is unable to determine the cartridge to which the current action belongs as part of dependency checks before building.
SNMP Parameters cannot be added to the discovery action because the project does not have a data dictionary.	Warning	Occurs when an SNMP processor is created and no data dictionary exists with the project. To correct this, create a Data Dictionary, and then create the SNMP processor.
SNMP Parameters cannot be added to the discovery action because a Data Dictionary Element matching the name SnmpParameters was found but it is not assigned the Scan Parameter Group Specification type.	Warning	Occurs when a SNMP processor is created and the project's Data Dictionary exists with an SnmpParameters structure that is not of Type scan parameter group. To correct, delete the conflicting SnmpParameters or change its Entity Type to scan parameter group.
SNMP Processor has not specified any OIDs	Error	Occurs if an SNMP processor has not specified any OIDs.
Processor implementation has not been specified	Error	Occurs if the processor's Implementation Class is not specified on the processor's **Details** tab.
Processor implementation is missing	Error	Occurs if the processors implementation class, which is specified on the processor's **Details** tab, is missing in Design Studio.
Processor implementation package does not match Processor interface package	Error	Occurs if the package defined in the processor's Implementation Class does not match the package of the processor's generated interface.
MIB Directory has not been specified. See Oracle Design Studio Network Integrity preferences.	Error	Occurs if the MIB Directory is not specified in the Oracle Design Studio Network Integrity Preferences (Window -> Preferences -> Oracle Design Studio -> Network Integrity).
MIB directory *mib directory* does not exist. See Oracle Design Studio Network Integrity preferences	Error	Occurs if the MIB Directory as specified in the Oracle Design Studio Network Integrity Preferences (Window -> Preferences -> Oracle Design Studio -> Network Integrity) does not exist.
MIB module *mib module name* does not exist	Error	Occurs if the MIBs specified as part of the SNMP processor are not available in the MIB Directory.
Processor is not used in an action	Warning	Occurs if the processor is not used by an action.
Action has not specified a result category. At least one result category must be specified	Error	Occurs if the action has not defined at least one result category.
Action has not specified a result source. At least one result source must be specified	Error	Occurs if the Discrepancy detection action does not contain at least one result source.

Table A–1   (Cont.) Network Integrity Error Message, Classification, and Error Condition

Error Message	Classification	Error Condition
Result source action *action name* cannot be found	Error	Occurs if the discrepancy detection action's result source action cannot be found. For example, the action has been deleted.
Result source *action name result source name* cannot be found	Error	Occurs if the discrepancy detection action's result source cannot be found. For example, it has been deleted from the action.
Scan Parameter Group *group_name* does not exist	Error	Occurs if the Data Dictionary Structure referenced by an action's scan parameter group has been deleted.
Data dictionary element for Scan Parameter Group *group_name* is invalid	Error	Occurs if the Data Dictionary Structure or its Elements are invalid. For example, the Entity Type is not a scan parameter group.
SNMP Processor requires "SnmpParameters" Scan Parameter Group	Error	Occurs if the SnmpParameters scan parameter group is not available in the workspace. To correct, ensure the MIB_II_SNMP_Cartridge is imported in the workspace. Next, remove and re-add the SNMP processor to the discovery action.
Address handler implementation has not been specified	Error	Occurs if the Implementation Class for an AddressHandler is not specified.
Address handler implementation is missing	Error	Occurs if the Implementation class itself is not in Design Studio.
Address handler implementation package does not match interface package	Error	Occurs if the package defined in the AddressHandler's Implementation Class does not match the package of the AddressHandler's generated interface.
Specification *specification name* does not exist	Error	Occurs if the Specification referenced by a processor's Model Collection does not exist. For example, it has been deleted.
Data dictionary element for specification *specification name* is invalid	Error	Occurs if the Data Dictionary Element is invalid. For example, POMS does not support it.
Stale imported Action *action name*. The imported Action's Processors have changed since they were imported.	Error	Occurs when imported action's processors have changed. For example, the ordering of the processors in the owning action has changed.
Action contains no Processors	Warning	Occurs when an action exists without any processors.
Cartridge contains neither Actions nor address handlers	Error	Occurs when a new Integrity Project contains no actions or address handlers.
Provider has not been specified	Warning	Occurs when the cartridge Provider has not be specified on the Network Integrity cartridge **Properties** tab.
Cartridge cannot contain both actions and address handlers	Error	Occurs when an Integrity project contains both address handlers and actions, which is invalid.
Condition implementation has not been specified for condition *condition name*	Error	Occurs when the Implementation Class has not been provided for a condition within an action.
Condition implementation is missing for condition *condition name*	Error	Occurs if the Implementation class itself is not in Design Studio.

Table A–1   (Cont.) Network Integrity Error Message, Classification, and Error Condition

Error Message	Classification	Error Condition
Model Collection is not associated with any Actions. A model collection must be associated with at least one Action.	Error	Occurs when the Model Collection is not associated to at least one action.
Resolution Action has not specified a Resolution Action Label.	Error	Occurs when the resolution action does not have a Resolution Action Label, which is used as the resolution string in the UI for resolving discrepancies.
Error Retrieving Cartridge Model	Error	Occurs when a given action's processors do not have a Provider.
Action *action name* is not a valid Action and cannot be added.	Error	Occurs when selecting an invalid action when adding processors to an action.
Action *action name* does not contain any Processors. Actions must contain at least one processor to be eligible for inclusion in another Action.	Error	Occurs when importing an action, which contains no processors.
The are no output parameters on any of the Processors that are of a type that can be iterated over.	Error	Occurs when adding a For Each to an action, which has processors that do not have an output parameter that allows iteration.
The order of Processors from Imported Actions can not be changed.	Error	Occurs when the order of processors from imported actions is changed.
Processor *processor name* uses parameter *parameter name* and Processor *processor name* outputs this parameter, continuing may make the Action invalid. Do you want to continue?	Confirmation	Occurs when changing the order (Moving Down) of processors within an action resulting in invalidating the flow of parameters thus making the action as a whole invalid.
Processor *processor name* has a condition that uses parameter *parameter name* and Processor *processor name* outputs this parameter, continuing may make the Action invalid. Do you want to continue?	Confirmation	Occurs when changing the order of processors (Moving Down) within an action resulting in invalidating one or more conditions.
Processor *processor name* outputs parameter *parameter name* and Processor *processor name* uses this parameter, continuing may make the Action invalid. Do you want to continue?	Confirmation	Occurs when changing the order (Moving Up) of processors within an action resulting in invalidating the flow of parameters thus making the action as a whole invalid.
Processor *processor name* outputs parameter *parameter name* and Processor *processor name* has a condition that uses this parameter, continuing may make the Action invalid. Do you want to continue?	Confirmation	Occurs when changing the order of processors (Moving Up) within an action resulting in invalidating one or more conditions.
Action should not be null	Error	Occurs when adding or removing elements (processors, For Each blocks, and so on) from an action, which is null.
The condition could not be added because the following *action name* are read only	Error	Occurs when attempting to add a condition to an action, which is read only.
The condition could not be removed because the following *action name* are read only	Error	Occurs when attempting to remove a condition from an action, which is read only.
The condition interface *condition interface name* has not been generated. It is recommended to save and build the Action before creating the implementation so that the interface is generated. Continue creating the implementation class anyway?	Confirmation	Occurs if the condition interface has not been generated before the implementation class being generated.

Table A–1   (Cont.) Network Integrity Error Message, Classification, and Error Condition

Error Message	Classification	Error Condition
Condition '*condition name}* has relations. Are you sure you want to delete it?	Warning	Occurs when the condition to be deleted has relationship to a processor.
A condition called *condition name* already exists on this plug-in, specify a different name.	Error	Occurs when attempting to create a condition with a name that already exists within the action.
The condition name must have a length greater than 0 but not exceeding 50 characters	Error	Occurs when the length of the target condition name is not within the valid range of 1 – 50 characters.
This output parameter type is used by a for each, therefore the parameter type must be an iterable type.	Error	Occurs when the output parameter type used as an input to a For Each is not iterable.
Processor *processor name* is not writable, so references to this output parameter is not updated.	Error	Occurs when trying to modify a processor, which is read only.
Input parameters are referencing this output parameter. Changing the name or type may generate compile errors. Do you want to continue?	Warning	Occurs when changing the name of an output parameter, which has referencing input parameters on processors whose java classes are already generated.
Input parameters are referencing this output parameter. Removing it generates compile and validation errors. Do you want to continue?	Warning	Occurs when removing an output parameter, which has referencing input parameters on processors whose java classes are already generated.
There are no output parameters available from preceding Processors to be selected	Informational	Occurs when selecting a processor's input parameters and no preceding processor has an output Parameters.
No uses of output parameter *parameter name* were found.	Informational	Occurs when viewing the usage of an output parameter, which is not used as an input parameter.
The provided name already exists. Enter a different name.	Error	Occurs when adding a condition using a name that already exists.
The name cannot exceed 50 characters	Error	Occurs when adding a condition with a name that exceeds 50 characters.
The name must start with a letter.	Error	Occurs when creating an Element (for example, processor, address handler) with an invalid name (i.e. starts with a digit) using the Design Studio Model Entity Wizard.
Action names must start with a letter.	Error	Occurs when creating an action with an invalid name.
A value for implementation prefix is required when the use default option is not selected.	Error	Occurs when creating an action and no implementation prefix is specified when the default option is not selected.
The implementation prefix must begin with a letter.	Error	Occurs when specifying an action's or processor's Implementation Prefix starting with a character other than a letter.
Error trying to lookup interface in project.	Error	Occurs when Design Studio is attempting to create a class that implements an interface, which does not exist in the Project.
An error occurred attempting to create a Java class. Details...	Error	Occurs when Design Studio is unable to create a Java class likely due to a Java Model problem or permissions.

Table A–1   (Cont.) Network Integrity Error Message, Classification, and Error Condition

Error Message	Classification	Error Condition
The generated interface *interface name* could not be found in your project. It is recommended to save and build before creating the implementation so that the interface is available. Continue creating the implementation class anyway?	Warning	Occurs when generating the implementation before the interface is available. For example, when creating a new processor, it is recommended to save and build before creating the implementation class.
The required interface, *interface name*, could not be found. Please clean and build the project.	Error	Occurs when selecting the implementation before the interface is available. For example, when creating a new processor, it is recommended to save and build before selecting the implementation class.
The package rename cannot be performed because the following entities are not writable:	Error	Occurs when modified the default package on the Project editor **Properties** tab and the underlying classes are read only.
Project name should not contain spaces.	Error	Occurs when attempting to create a Integrity Project with a name that contains spaces.
A Default Cartridge Package is required	Error	Occurs if there is no Default Cartridge Package specified under the Oracle Design Studio -> Network Integrity section in the Design Studio Preferences located under Window -> Preferences.
Spaces are not allowed in the package name	Error	Occurs if the Default Cartridge Package value contains spaces.
This removes all generated UI hints artifacts. Do you wish to continue?	Confirmation	Occurs when clicking the Clean UI Hints button located on the UI Hints tab of the Network Integrity cartridge element.
The UI Hints could not be cleaned, please ensure the mds.mar file is not read only	Error	Occurs when attempting to clean the UI Hints while the mds.mar file is read only. The mds.mar is located in the cartridge lib directory.
Spaces are not allowed in the package name	Error	Occurs when attempting to rename the Default Package property on the Properties tab of the Network Integrity cartridge element.
Please fix fields with errors.	Error	Occurs when creating an output parameter with an invalid Type.
The first character in a parameter name should be lowercase	Warning	Occurs when adding output parameters to a processor and the parameter name begins with an invalid character (i.e. uppercase).
A {field name} value must be entered.	Error	Occurs when adding output parameters, property groups and properties to a processor and no name value is specified.
Parameter *parameter name* could not be added because a parameter with the same name already exists. Remove the parameter with the same name and retry the operation.	Error	Occurs when adding an output or input parameter using a name that already exists in the Parameter list. Names must be unique in the parameter list since the name generates the getter methods.
The name *parameter name* already exists as a parameter, enter a different name	Error	Occurs when adding an output parameter using a name that already exists.
The name cannot contain spaces	Error	Occurs when adding an output parameter or property group to a processor and the name contains spaces.

Table A–1   (Cont.) Network Integrity Error Message, Classification, and Error Condition

Error Message	Classification	Error Condition
The name cannot start with a number	Error	Occurs when adding an output parameter or property group to a processor and the name starts with a number.
Parameter Type *parameter type* may produce warnings in generated code. Do you want to continue?	Confirmation	Occurs if the parameter type of an output or input parameter may cause compile warnings.
The parameter type *parameter type* produces the following warning in generated code. Do you want to continue?	Confirmation	Occurs if the generated code contains warnings based on the parameter type of an output or input parameter.
The name must be a valid java identifier that does not contain special characters	Error	Occurs when adding an output parameter or property group to a processor and the name contains a special character (for example, %).
The name *parameter name* is a reserved word in Java, enter a different name	Error	Occurs when adding an output parameter or property group to a processor and the name is equivalent to a reserved word in Java and therefore would cause compiling errors.
Type *parameter type* could not be found in the project	Error	Occurs if the parameter type of an output or input parameter could not be found in the Integrity Project.
A property group with the name *property group name* already exists on this input	Error	Occurs when adding a property group using a name that already exists.
A Property with the name *property name* and value *property value* already exists, please choose a different name/value combination	Error	Occurs when adding or modifying a Property using a name and value that already exists.
One or more errors exist with the fields	Error	Occurs when creating a property group with an invalid name.
A property group with the name *property group name* already exists, please choose a different name	Error	Occurs when modifying a property group changing its name to a name that already exists.
One or more errors exist with the fields	Error	Indicates a problem with result groups or result source.
A *field name* value must be entered.	Error	Occurs when creating a result category or condition with no name.
The result category name must have a length greater than 0 but not exceeding 255 characters	Error	Occurs when modifying a result category changing its name to have a length of 0 or greater than 255 characters.
Data dictionary named *data dictionary name* could not be found.	Error	Occurs when the data dictionary elements of a model collection cannot be found.
The MIB File *mib filename* could not be loaded because of the following error: Details...	Error	Occurs when a file other than a MIB File is selected when clicking the Load MIB button within an SNMP processor.
A valid MIB Module called *mib module name* could not be found in MIB File: *mib filename*	Error	Occurs when the target MIB File attempting to be loaded by a SNMP processor does not contain any MIB Modules.
The MIB directory *mib directory* either does not exist or is not accessible. Either create this directory or change the configured MIB Directory in the Network Integrity Preferences Page (Preferences then Oracle Design Studio then Network Integrity)	Error	Occurs when the configured MIB Directory as specified in the Network Integrity Preferences Page is not accessible.

Table A–1 (Cont.) Network Integrity Error Message, Classification, and Error Condition

Error Message	Classification	Error Condition
The following error occurred loading MIB *mib filename*: Details...	Error	Occurs when the target MIB File is corrupt.
Selected node: *oid*, is not readable, only readable nodes are supported.	Error	Occurs when attempting to load an OID, which is not readable.
Selected node: *oid*, is not supported (only scalar and table column are supported).	Error	Occurs when attempting to load an OID, which is not scalar or a table column.

Design Studio Logging

When developing cartridge projects within Design Studio for Network Integrity it is likely that the developer requires logging for traceability during normal cartridge operation and for debugging. This section outlines how to introduce logging into the developer's implementation. This section addresses logging that is visible inside the WebLogic log files. It does not discuss introducing Design Studio logging (for example, Design Studio Error Logs).

Network Integrity uses the java.util.logging package for logging messages. For an overview of the Java logging framework, visit Oracle's site on the subject at

http://download.oracle.com/javase/6/docs/api/index.html

To create an instance of the appropriate logger add a static variable to an implementation class passing in the name of the current class. For example,

```
private static final Logger logger = Logger
                    .getLogger(DiscrepancyDetectorImpl.class.getName());
```

When the above is defined, invoke logging according to the API specification. For example,

```
logger.log(Level.SEVERE, "Error while detecting discrepancies.", e);
```

To redirect the Network Integrity logs produced by the above into a WebLogic log file use the following procedure:

1. Insert the following 2 XML fragments into the file *<DOMAIN_HOME>*
   /config/fmwconfig/servers/<TargetServer>/logging.xml.<TargetServer>.
   <TargetServer> represents the name of the WebLogic Server where the Network Integrity application is running.

   a. The following fragment goes inside the **<log_handlers>** block and defines the log handler and log file location. If required, change the log handler; however, this value must match the value referenced in the fragment in step b. If necessary, change the location where the log file is generated.

   ```
       <log_handler name='ni-handler'
   class='oracle.core.ojdl.logging.ODLHandlerFactory'>
       <property name='path'
   value='${domain.home}/servers/${weblogic.Name}/logs/ni-weblogic.log'/>
       <property name='maxFileSize' value='10485760'/>
       <property name='maxLogSize' value='104857600'/>
     </log_handler>
   ```

   b. This fragment goes inside the **<loggers>** block (at the end) and defines the logger name. This name refers to the Java package of a customer's implementation code, the log level and the handler. The handler must match the value configured in step a (for example, **ni-handler**). If necessary, tailor the

log level. Consult Table A–2 that maps the Java log levels to the ODL log levels (for example, TRACE:32) used in the logging.xml file.

```
<logger name="oracle.communications.integrity" level="TRACE:32">
  <handler name="ni-handler"/>
</logger>
<logger name="oracle.communications.activation" level="TRACE:32">
  <handler name="ni-handler"/>
</logger>
<logger name="oracle.communications.inventory" level="TRACE:32">
  <handler name="ni-handler"/>
</logger>
```

2. Save **logging.xml**.

When determining what level to set in the **logging.xml** (step 1) use Table A–2 to map the Java Log Levels to ODL Log Levels.

Table A–2   Java Log Level to ODL Log Level Mapping

Java Log Level	ODL Message Type:Log Level	ODL Description
SEVERE.intValue()+100	INTERNAL_ERROR:1	The program has experienced an error for some internal or unexpected non-recoverable exception.
SEVERE	ERROR:1	A problem requiring attention from the system administrator has occurred.
WARNING	WARNING:1	An action occurred or a condition was discovered that should be reviewed and may require action before an error occurs.
INFO	NOTIFICATION:1	A report of a normal action or event. This could be a user operation, such as "login completed" or an automatic operation such as a log file rotation.
CONFIG	NOTIFICATION:16	A configuration-related message or problem.
FINE	TRACE:1	A trace or debug message used for debugging or performance monitoring. Typically contains detailed event data.
FINER	TRACE:16	A fairly detailed trace or debug message.
FINEST	TRACE:32	A highly detailed trace or debug message.

For more information on ODL visit

http://download.oracle.com/docs/cd/B31017_01/web.1013/b28952/logging.htm