Oracle® Configurator

Extensions and Interface Object Developer's Guide

Release 11*i*

Part No. B13607-03

May 2005

This document describes Configurator Extensions, which augment the functionality of a runtime Oracle Configurator, and the Oracle Configuration Interface Object (CIO), which is used by Configurator Extensions to access the runtime Oracle Configurator.

ORACLE®

Oracle Configurator Extensions and Interface Object Developer's Guide, Release 11*i*

Part No. B13607-03

Primary Author: Mark Sawtelle

Contributors: Raju Addala, Brent Benson, Ivan Lazarov, David Lee, Anupam Miharia, Janet Page, Marty Plotkin, Brian Ross

Contents

List of Examples

List of Figures

List of Tables

Send Us Your Comments

Oracle Configurator Extensions and Interface Object Developer's Guide, Release 11*i*

Part No. B13607-03

Oracle welcomes your comments and suggestions on the quality and usefulness of this publication. Your input is an important part of the information used for revision.

- Did you find any errors?
- Is the information clearly presented?
- Do you need more information? If so, where?
- Are the examples correct? Do you need more examples?
- What features did you like most about this manual?

If you find any errors or have any other suggestions for improvement, please indicate the title and part number of the documentation and the chapter, section, and page number (if available). You can send comments to us in the following ways:

- Electronic mail: czdoc_us@oracle.com
- FAX: 781-238-9898. Attn: Oracle Configurator Documentation
- Postal service:

  Oracle Corporation
  Oracle Configurator Documentation
  10 Van de Graaff Drive
  Burlington, MA 01803-5146
  USA

If you would like a reply, please give your name, address, telephone number, and electronic mail address (optional).

If you have problems with the software, please contact your local Oracle Support Services.

Preface

You can use Configurator Extensions to augment the functionality of your runtime Oracle Configurator beyond what is provided by Oracle Configurator Developer. You create Configurator Extension classes, which use the Configuration Interface Object (CIO) to perform various tasks, including accessing the Model, setting and getting logic states, and adding instantiable components. You can also use the CIO in your own applications, to interact with the Model.

Intended Audience

This manual is intended primarily for software developers writing Configurator Extensions. The language required for developing Configurator Extensions is Java.

This manual assumes that you are an experienced Java programmer.

Note: Be sure to check Section 1.2, "Prerequisite Skills for Developing Configurator Extensions" on page 1-2, which describes the Java development skills required for success with Configurator Extensions.

This manual also provides background on the CIO. This information is needed by developers of applications that have customized user interfaces that access the runtime Oracle Configurator.

Documentation Accessibility

Our goal is to make Oracle products, services, and supporting documentation accessible, with good usability, to the disabled community. To that end, our documentation includes features that make information available to users of assistive technology. This documentation is available in HTML format, and contains markup to facilitate access by the disabled community. Accessibility standards will continue to evolve over time, and Oracle is actively engaged with other market-leading technology vendors to address technical obstacles so that our documentation can be accessible to all of our customers. For more information, visit the Oracle Accessibility Program Web site at

`http://www.oracle.com/accessibility/`

Accessibility of Code Examples in Documentation Screen readers may not always correctly read the code examples in this document. The conventions for writing code require that closing braces should appear on an otherwise empty line; however, some screen readers may not always read a line of text that consists solely of a bracket or brace.

Accessibility of Links to External Web Sites in Documentation This documentation may contain links to Web sites of other companies or organizations that Oracle does not own or control. Oracle neither evaluates nor makes any representations regarding the accessibility of these Web sites.

TTY Access to Oracle Support Services Oracle provides dedicated Text Telephone (TTY) access to Oracle Support Services within the United States of America 24 hours a day, seven days a week. For TTY support, call 800.446.2398.

Structure

This book contains a table of contents, lists of examples, tables, and figures, a reader comment form, a preface, the chapters listed below, appendixes, a glossary, and an index.

Configurator Extension Basics

Provides essential information about implementing Configurator Extensions. Explains what Configurator Extensions are, and the different types available. Explains the relationship of Configurator Extensions and the CIO.

Building Configurator Extensions

Describes how to code Configurator Extensions, including suggestions for effective development practices and avoiding common mistakes.

Uses for Configurator Extensions

Collects instructions on how to use Configurator Extensions for specific tasks, such as generating custom output and filtering for connectivity.

CIO Basics

Explains the basics of the Oracle Configuration Interface Object (CIO) and how to use it.

Working with Configurations

Explains how to work with runtime configuration instances.

Working with Model Entities

Explains how to work with nodes of the runtime Model. such as Components and Features.

Using Logic Transactions

Explains how to use logic transactions to safely structure a configuration session.

Contradictions, Exceptions, and Validation

Explains how to validate configurations and handle contradictions.

Using Requests

Describes requests, which are programmatic attempts to modify a configuration

Configuration Session Change Tracking

Describes the Configuration Delta API, which enables you to track changes made to regions of your user interface during a configuration session.

Logging Through the CIO

Describes how you can use the Oracle Applications Logging Framework with Oracle Configurator and the Oracle Configuration Interface Object to provide a convenient and uniform interface for logging their activity.

Reference Documentation for the CIO

Explains how to access the reference documentation for the CIO, which is generated in Javadoc format.

Code Examples

Collects complete code examples related to topics covered elsewhere in this document.

Java Parameter Types for Configurator Extensions

Lists the Java classes that you can use for Configurator Extension method parameters when creating event bindings.

Glossary

Contains a glossary of terms and acronyms used throughout the Oracle Configurator documentation set.

Related Documents

The following documents are also included in the Oracle Configurator documentation set on the Oracle Configurator Developer compact disc:

- *About Oracle Configurator* documentation
- *Oracle Configurator Implementation Guide*
- *Oracle Configurator Installation Guide*
- *Oracle Configurator Developer User's Guide*
- *Oracle Configurator Methodologies*
- *Oracle Configurator Performance Guide*
- *Oracle Configurator Modeling Guide*

The following documents may also be useful:

- *Oracle8i JDBC Developer's Guide and Reference*

Conventions

In examples, an implied carriage return occurs at the end of each line, unless otherwise noted. You must press the Return key at the end of a line of input.

The following conventions are also used in this manual:

Convention	Meaning
. . .	Vertical ellipsis points in an example mean that information not directly related to the example has been omitted.
. . .	Horizontal ellipsis points in statements or commands mean that parts of the statement or command not directly related to the example have been omitted
boldface text	Boldface type in text indicates a new term, a term defined in the glossary, specific keys, and labels of user interface objects. Boldface type also indicates a menu, command, or option, especially within procedures
italics	Italic type in text, tables, or code examples indicates user-supplied text. Replace these placeholders with a specific value or string.
[ ]	Brackets enclose optional clauses from which you can choose one or none.
>	The left bracket alone represents the MS DOS prompt.
$	The dollar sign represents the DIGITAL Command Language prompt in Windows and the Bourne shell prompt in Digital UNIX.
%	The percent sign alone represents the UNIX prompt.
name()	In text other than code examples, the names of programming language methods and functions are shown with trailing parentheses. The parentheses are always shown as empty. For the actual argument or parameter list, see the reference documentation. This convention is not used in code examples.

Product Support

The mission of the Oracle Support Services organization is to help you resolve any issues or questions that you have regarding Oracle Configurator Developer and Oracle Configurator.

To report issues that are not mission-critical, submit a Technical Assistance Request (TAR) using Metalink, Oracle’s technical support Web site at:

`http://www.oracle.com/support/metalink/`

Log in to your Metalink account and navigate to the Configurator TAR template:

1. Choose the **TARs** link in the left menu.
2. Click on **Create a TAR**.
3. Fill in or choose a profile.
4. In the same form:
   a. Choose **Product**: Oracle Configurator or Oracle Configurator Developer
   b. Choose **Type of Problem**: Oracle Configurator Generic Issue template
5. Provide the information requested in the iTAR template.

You can also find product-specific documentation and other useful information using Metalink.

For a complete listing of available Oracle Support Services and phone numbers, see:

`www.oracle.com/support/`

Troubleshooting

Oracle Configurator Developer and Oracle Configurator use the standard Oracle Applications methods of logging to analyze and debug both development and runtime issues. These methods include setting various profile options and Java system properties to enable logging and specify the desired level of detail you want to record.

For general information about the logging options available when working in Configurator Developer, see the *Oracle Configurator Developer User's Guide*.

For details about the logging methods available in Configurator Developer and a runtime Oracle Configurator, see:

- The *Oracle Applications System Administrator's Guide* for descriptions of the Oracle Applications Manager UI screens that allow System Administrators to set up logging profiles, review Java system properties, search for log messages, and so on.
- The *Oracle Applications Supportability Guide*, which includes logging guidelines for both System Administrators and developers, and related topics.
- The Oracle Applications Framework Release 11i Documentation Road Map (Metalink Note # 275880.1).

Part I

Configurator Extensions

Part I contains the following chapters:

1 Configurator Extension Basics

Configurator Extensions extend the behavior of the runtime Oracle Configurator. A Configurator Extension is a custom-coded Java class that uses an established interface to access a configuration at runtime. The interface is called the Oracle Configuration Interface Object (CIO); it is described in Part II, "The Configuration Interface Object (CIO)".

This chapter contains an overview of how Configurator Extensions work and how to implement them. It also provides important facts about Configurator Extensions and prerequisites for developing them.

The sections of this chapter are:

- What are Configurator Extensions?
- Prerequisite Skills for Developing Configurator Extensions
- Important Facts About Configurator Extensions
- Requirements and Restrictions for Configurator Extensions
- Configurator Extensions and the CIO
- Installation Requirements for Configurator Extensions

See the Section "Structure" on page xiv in the Preface to verify the audience for whom this chapter is intended.

Note: Be sure to check Section 1.2, "Prerequisite Skills for Developing Configurator Extensions" on page 1-2, which describes the Java development skills required for success with Configurator Extensions.

Note: Review the *Oracle Configurator Performance Guide* for information on the performance impacts of Configurator Extensions.

1.1 What are Configurator Extensions?

Configurator Extensions extend your runtime Oracle Configurator by attaching custom code through established interfaces.

The term *Configurator Extension* includes the following:

- A Configurator Extension *class* is the Java class containing the methods that implement desired behavior
- A Configurator Extension *instance* is the event-driven execution (the Java object) of the Java class at runtime
- A Configurator Extension *Rule* is the set of arrangements that you make in Oracle Configurator Developer to associate the CX class to a Model

For additional information, see the chapter on Configurator Extensions in the *Oracle Configurator Developer User's Guide*, which explains the following essential topics related to incorporating Configurator Extensions into your configuration model:

- Configurator Extension Rules
- Configurator Extension Archives and the Archive Path
- Events and Event Binding
- Arguments and Argument Binding

1.2 Prerequisite Skills for Developing Configurator Extensions

To effectively develop a Configurator Extension, an appropriate level of Java development proficiency is required. The specific level of Java proficiency required depends on the specific functionality required by the desired Configurator Extension.

In general, the Configurator Extension developer should have the following knowledge:

- A basic understanding of these structures:
  - Oracle Applications Bills of Material (BOMs), which consist of Models, Option Classes, and Standard Items
  - Oracle Configurator Models, which consist of Components, Features, and Options
  - The relationship of these BOM and Model structures to the CIO
- Java programming experience that should include solid familiarity with:
  - The Collections class and its subclasses
  - Concurrency issues
  - CIO transaction handling (see Chapter 7)
  - Exception handling
  - Using Java Interfaces
  - HTML and the Java class `HttpServletResponse` (for writing Configurator Extensions that generate custom output)
- A working understanding of Oracle databases, including the principles of JDBC.
- A familiarity with the Oracle Configurator documentation, including the CIO reference documentation (see Appendix A).

The skills listed above are fundamental. Other specific expertise may be required for developing Configurator Extensions to the specific requirements for your project.

1.3 Important Facts About Configurator Extensions

Keep these facts in mind when working with Configurator Extensions and the CIO.

- Configurator Extension Rules have many of the same attributes as other Rules, and the procedure for defining them is similar. For example, Configurator Extensions have effectivity, can be disabled, and can participate in rule sequences. For more details about defining configuration rules, see the *Oracle Configurator Developer User's Guide*.
- When the runtime Oracle Configurator starts up, it creates an instance of the CIO. During the resulting configuration session, the CIO creates a `Configuration` object. Then Oracle Configurator creates runtime instances of all mandatory model structure, and, for each instance of each instantiated base node associated with a Configurator Extension, an instance of the class that you defined for your Configurator Extension. Oracle Configurator then attaches the Configurator Extension instance to the associated node.
- You can associate more than one Configurator Extension with a particular node; the CIO will create instances of all of the Configurator Extensions at runtime.
- In order to communicate with your application's Model, a Configurator Extension uses Oracle's CIO API. The CIO can also be used to develop a custom user interface that allows the runtime Oracle Configurator to access the Model. See Section 1.5, "Configurator Extensions and the CIO" on page 1-4, and all of Part II, "The Configuration Interface Object (CIO)".

Note: As a point of information, the user interfaces generated with Oracle Configurator Developer for the runtime Oracle Configurator communicate in this way with the configuration model.

1.4 Requirements and Restrictions for Configurator Extensions

You must observe certain requirements and restrictions when working with Configurator Extensions and the CIO.

1.4.1 Requirements for Configurator Extensions

Keep these requirements in mind when working with Configurator Extensions and the CIO.

- To build a Configurator Extension, you implement an object class in Java. Oracle requires that Configurator Extensions be implemented only in Java. Configurator Extensions can run on any Oracle platform that supports Java.
- The runtime Oracle Configurator automatically sets up a JDBC database connection for use by the CIO. Custom applications that take the place of the runtime Oracle Configurator must perform this task. See Section 4.3, "Initializing the CIO" on page 4-4 for details.
- If your host application uses a custom user interface in an MLS deployment, you may need to create **ICX** session tickets in order to correctly set the current language.
- If you have written Configurator Extensions that use custom messages, then those messages must be stored into and retrieved from the FND_NEW_MESSAGES table. You are responsible for translating these messages. See the information on MLS in the *Oracle Configurator Implementation Guide*.

1.4.2 Restrictions for Configurator Extensions

Keep these restrictions in mind when working with Configurator Extensions and the CIO.

- Configurator Extensions cannot be used to customize Oracle Configurator Developer.
- CIO interfaces are not thread-safe. A single configuration session should only be accessed by a single thread at a time. Whenever a custom application interacts directly with the CIO, you must ensure that it accesses a configuration session by only a single thread at a time. Multithreading problems can occur, for instance, when end users click multiple times in a child window. You can prevent multithreading problems by locking your User Interface or synchronizing on your servlet. See Section 5.10, "Sharing a Configuration Session" on page 5-10 for an example of when this is a consideration.
- If any Configurator Extensions cannot be loaded when you create a new configuration (for instance, due to internal errors or an incorrect class path or Archive Path), the configuration will fail to open.

1.5 Configurator Extensions and the CIO

Your Configurator Extension is a client of the CIO. When you program against the CIO, the CIO creates instances of a set of public interface objects that you work with. These interfaces are defined in the package `oracle.apps.cz.cio`. Your code should refer only to these public interface objects. See Section 4.2, "The CIO’s Runtime Node Interfaces" on page 4-2.

Configurator Extensions are invoked by the CIO through the runtime Oracle Configurator, and Configurator Extensions call the CIO to get information from the runtime configuration model. The CIO is like a broker for the runtime configuration model, in that it passes information both into and out of the model. Programmers writing Configurator Extensions need to know how to use the CIO.

1.6 Installation Requirements for Configurator Extensions

This section describes the elements that need to be installed to develop, compile, and test Configurator Extensions. See the *Oracle Configurator Installation Guide* and *Oracle Configurator Release Notes* for more detail.

1.6.1 Installation Requirements for Developing Configurator Extensions

In order to develop Java Configurator Extensions, you must install a Java development environment that enables you to compile Java classes, such as:

- The latest version of Oracle JDeveloper
- The latest certified patch release of the Java Development Kit (JDK) for your platform. For the JDK release number, see the *About Oracle Configurator* documentation for this release on Metalink, Oracle's technical support Web site.

You do not need JDBC drivers or database access to compile a Configurator Extension, although these are required to run one. The required driver classes are contained in the Oracle Applications environment.

Note: If you use a class from the collections library, such as `List`, then for compatibility with the CIO's package structure you must import the class using this syntax:

```
import com.sun.java.util.collections.List;
```

1.6.2 Installation Requirements for Compiling Configurator Extensions

In order to compile Configurator Extensions:

- Your class path should be the same as the class path for Oracle *i*AS (Internet Application Server.
- You should compile using the latest certified patch release of the Java Development Kit (JDK) for your platform. For the JDK release number, see the *About Oracle Configurator* documentation for this release on Metalink, Oracle's technical support Web site.
- The shared object files described in Table 1–1 must be installed and recognized by your operating system environment in the appropriate locations.

Table 1–1 Required Software for Configurator Extensions

File	For Platform	Comment
`czlce.dll`	Windows NT	Must be in the PATH system environment variable on the host machine on which the Oracle Configurator Servlet is installed.
`libczlce.so` (or `.sh`)	UNIX family	Must be in the LD_LIBRARY_PATH environment variable for the Oracle Configurator Servlet.

See the *Oracle Configurator Installation Guide* and the *Oracle Configurator Implementation Guide* for complete details on installation and environment. For background on JDBC drivers, see the *Oracle8i JDBC Developer's Guide and Reference*.

1.6.3 Installation Requirements for Testing Configurator Extensions

If you have installed and set up Oracle Configurator Developer so that the **Test Model** button runs the Model Debugger successfully, then this setup should also be correct for testing Configurator Extensions.

The classes that implement your Configurator Extensions should be contained in Configurator Extension Archives, as described in the *Oracle Configurator Developer User's Guide*.

It is also possible to install your classes in the class path for *i*AS, which takes precedence over the Configurator Extension Archive Path. However, if you do so you will not obtain important advantages provided by using Archives. See the *Oracle Configurator Developer User's Guide* for details.

2 Building Configurator Extensions

This chapter describes the process for building Configurator Extensions.

The sections of this chapter are:

- Overview of Building Configurator Extensions
- Developing Java Classes and Archives
- Example of Configurator Extension Development
- Suggested Development Practices

See the Section "Structure" on page xiv in the Preface to verify the audience for whom this chapter is intended.

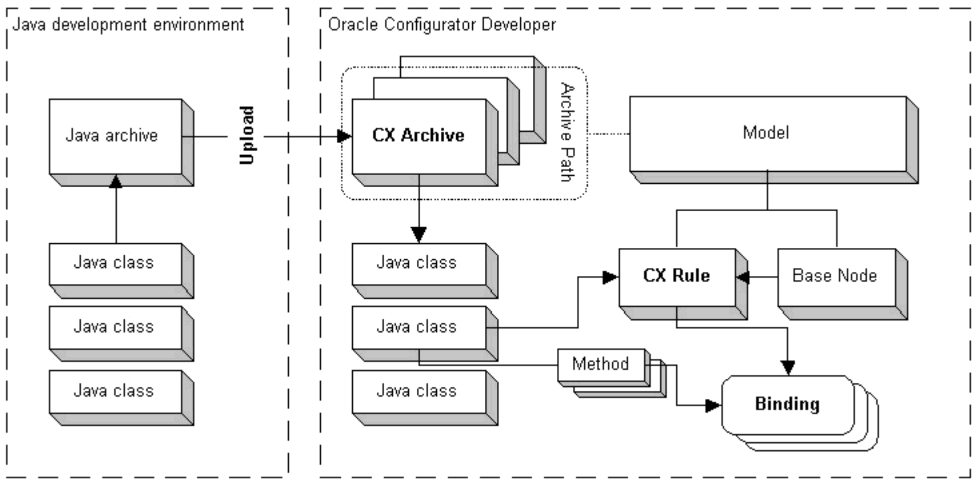
To understand the terms and concepts used in this section, see Chapter 1, "Configurator Extension Basics" and the chapter on Configurator Extensions in the *Oracle Configurator Developer User's Guide*.

2.1 Overview of Building Configurator Extensions

Figure 2–1, "Overview of Configurator Extension Development" on page 2-2 shows the relationship of a Java development environment to the Oracle Configurator Developer environment when creating Configurator Extensions. In the Java development environment, you compile Java classes and add them to Java archive files. In Oracle Configurator Developer, you upload Java archive files into Configurator Extension Archives.

In your Model, you specify the Archives that form the Model's Archive Path, which is an ordered list of one or more Configurator Extension Archives. Then you create Configurator Extension Rules, which associate Java classes from Archives with Model nodes. In each Rule, you create bindings, which bind together a configuration event, the parameters of a method in the Java class, and arguments related to the Model.

Figure 2–1 Overview of Configurator Extension Development



Java Development Tasks

The following tasks are normally performed by the programmer who is developing the Java code for Configurator Extensions. See Section 2.1.1, "Implementing Behavior with Java Classes" on page 2-3 for more details.

1. Develop Java classes and archives.

   See Section 2.2, "Developing Java Classes and Archives" on page 2-4.

2. Create Configurator Extension Archives and upload Java archives.

   See the *Oracle Configurator Developer User's Guide* for details on this and the following tasks.

3. Inspect the classes in an Archive.
4. Add archives to a Model's Archive Path.
5. Optionally, modify the Archive Path for a Model.

Configuration Modeling Tasks

The following tasks are normally performed by the model designer who is developing the configuration model and rules. See Section 2.1.2, "Incorporating Behavior into Configuration Models" on page 2-3 for more details.

6. Create a Configurator Extension Rule.

   See the *Oracle Configurator Developer User's Guide* for details on this and the following tasks.

7. Choose the Java class for a Rule.
8. Create event bindings for a Rule.
9. Bind arguments from the Model to parameters of Java methods.

   If you change the type or number of the parameters of a method used in a Configurator Extension Rule, then you must create a new binding that reflects those changes.

10. Test Configurator Extensions.

2.1.1 Implementing Behavior with Java Classes

Implement the behavior of your Configurator Extension by creating one or more Java classes and methods that use the Oracle Configuration Interface Object (CIO) to access a runtime configuration object. For details on using the CIO, see Part II, "The Configuration Interface Object (CIO)".

You can create your Configurator Extension class in any Java development environment. Then you store the compiled Java class in an archive file, using either the JAR or Zip format for your archive. You complete the coding stage of Configurator Extension development by uploading your archive to the Configurator Developer Repository as a Configurator Extension Archive.

Section 2.2, "Developing Java Classes and Archives" on page 2-4 provides the detailed procedure for the coding stage of Configurator Extension development.

For an example, see Section 2.3.1, "Example of Configurator Extension Coding" on page 2-5.

2.1.2 Incorporating Behavior into Configuration Models

The detailed procedure for the modeling stage of Configurator Extension development is provided in the *Oracle Configurator Developer User's Guide*. This section provides a simple overview.

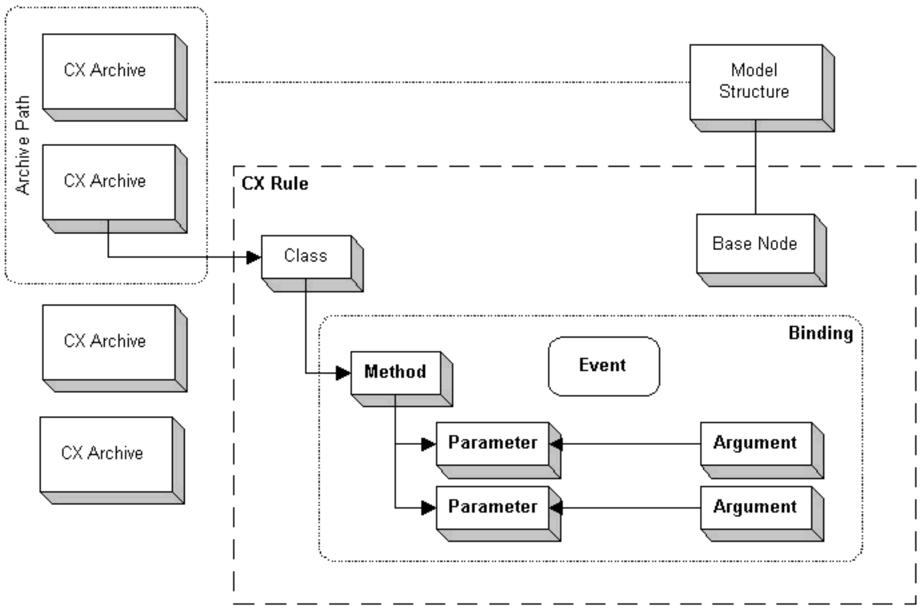
In Oracle Configurator Developer, you create a connection between your Java class and your configuration model. To create this connection, you create a Configurator Extension Rule that binds specific parameters of a Java method to specific nodes or Properties of a Model.

Figure 2–2, "Configurator Extension Binding" illustrates the relationship of bindings to Configurator Extension Rules. In this relationship:

- Each Model can include an Archive Path.
- A Configurator Extension Rule for the Model specifies:
  - A base node in the Model's structure
  - A Java class from one of the Archives in the Archive Path
  - One or more bindings
- A binding specifies:
  - A method from the specified Java class
  - An event
  - A mapping between each parameter of the method and an argument related to the Model

    The Java types of the parameters of your method must agree with the types of Model entities that are eligible for event binding. For a list of the Java classes that you can use in event bindings, see Appendix C, "Java Parameter Types for Configurator Extensions".

Figure 2–2 Configurator Extension Binding



For an example of the modeling stage of Configurator Extension development, see Section 2.3.2, "Example of Configurator Extension Modeling" on page 2-6.

2.2 Developing Java Classes and Archives

This section describes the basic process for coding Configurator Extensions.

Configurator Extensions depend on the CIO for access to your configuration model. For more background, see Part II, "The Configuration Interface Object (CIO)".

1. Use a Java development environment or text editor to create a `.java` file in which to define a Java class. See Example 2–1, "Sample Java Code for Configurator Extension (InstanceNameChange.java)" on page 2-6 for an example of a very basic Java class that can be used for a Configurator Extension.
2. Define your class path to include the package `oracle.apps.cz.cio`.

   See Section 1.6, "Installation Requirements for Configurator Extensions" on page 1-4.
3. Import the classes from the CIO that your Configurator Extension requires to do its work. See Chapter 4, "CIO Basics" on page 4-1 for background. The following example is typical:

   ```
   import oracle.apps.cz.cio.Component;
   ```

   If you use a class from the collections library, such as `List`, then for compatibility with the CIO's package structure you must import the class using this syntax:

   ```
   import com.sun.java.util.collections.List;
   ```
4. Define a class in which to determine the behavior of your Configurator Extension.

   ```
   public class InstanceNameChange {
     // implement methods here
   }
   ```

5. Create methods that implement the desired behavior for your Configurator Extension. Any methods that you intend to use in a binding in a Configurator Extension Rule must be declared as `public`.

   Call methods from the CIO that perform required interaction with your configuration model (see Section 4.2, "The CIO's Runtime Node Interfaces" on page 4-2).

   ```
   public void setDefaultName(Component comp, TextFeature tf) {
     // implement CX behavior here
   }
   ```

   Names of methods used for Configurator Extensions cannot be longer than 30 characters.

   The Java types of the parameters of your method must agree with the types of Model entities that are eligible for event binding. For a list of the Java classes that you can use in event bindings, see Appendix C, "Java Parameter Types for Configurator Extensions".

6. Compile the `.java` file into a `.class` file.

   Use the correct version of the Sun JDK for your platform. See Section 1.6.1, "Installation Requirements for Developing Configurator Extensions" on page 1-4.

7. Put the resulting `.class` file into a Java archive file.

   You can use either the JAR or Zip format for the Java archive. The archive must be valid. This means that the directory structure of the archive must correspond to the package structure of the Java packages in the archive. For example, the following examples refer to the same class in consistent ways. The first line shows an `import` statement using a package reference to the class, and the second line shows the directory path to the class as stored in an archive file:

   ```
   import oracle.apps.cz.cio.Component;

   oracle/apps/cz/cio/Component.class
   ```

8. Now the Java archive file can be incorporated into a Configurator Extension Archive in Configurator Developer. See Section 2.1.2, "Incorporating Behavior into Configuration Models" on page 2-3.

2.3 Example of Configurator Extension Development

This section provides a basic example of the development of a Configurator Extension, which consists of:

- Example of Configurator Extension Coding
- Example of Configurator Extension Modeling

2.3.1 Example of Configurator Extension Coding

Example 2–1 on page 2-6 shows the Java source code for a very simple Configurator Extension.

See Section 2.2, "Developing Java Classes and Archives" on page 2-4 for details on how to create this code and prepare it for use in a configuration model. See Section 2.3.2, "Example of Configurator Extension Modeling" on page 2-6 for how this code is used in a Configurator Extension Rule.

Example 2–1 Sample Java Code for Configurator Extension (InstanceNameChange.java)

```
// When bound to the event for addition of a component instance,
// takes input from the value of a bound Text Feature
// and changes the instance name to that corresponding text.

import oracle.apps.cz.cio.Component;
import oracle.apps.cz.cio.TextFeature;

public class InstanceNameChange {

    public void setDefaultName(Component comp, TextFeature tf) {

        String name = tf.getCurrentValue();
        comp.setInstanceName(name);
    }
}
```

2.3.2 Example of Configurator Extension Modeling

See the *Oracle Configurator Developer User's Guide* for details on how to incorporate a Configurator Extension in a configuration model. See Section 2.3.1, "Example of Configurator Extension Coding" on page 2-5 for how the behavior of this example is coded in Java.

Section 2.1.2, "Incorporating Behavior into Configuration Models" on page 2-3 provides a summary of the tasks for the modeling stage of Configurator Extension development.

The following list summarizes the options specific to this example:

- Use the Java source code in Example 2–1, "Sample Java Code for Configurator Extension (InstanceNameChange.java)" on page 2-6 to create your Java archive file and Configurator Extension Archive.
- When you define model structure, include a Component that can be instantiated multiple times and a Text Feature with some Initial Value of your choice.
- When you define a Configurator Extension rule, use the options listed in the following table:

Option	Choose ...
Model Node	The node of your Model on which you want Oracle Configurator Developer to place a button that adds additional instances of your Component.
Java Class	`InstanceNameChange`, from your Configurator Extension Archive
Java Class Instantiation	With Model Node Instance

- When you define an event binding, use the options listed in the following table:

Option	Choose ...
Event	`postInstanceAdd`
Command Name	A string that you choose as a command. For example: `ShowStructure`

Option	Choose ...
Event Scope	Your choice of scope. Try repeating the example with different scopes to see the effect when you test it.
Method Name	`showModelStructure`

- When you define your argument bindings, use the options listed in the following tables:

Option	Choose ...
Argument Type	`oracle.apps.cz.cio.Component`
Argument Specification	Event Parameter
Binding	`instance`

Option	Choose ...
Argument Type	`oracle.apps.cz.cio.TextFeature`
Argument Specification	Model Node or Property
Binding	The Text Feature whose value is used to name new instances of the Component.

- When you test the Model, try this procedure:
  1. Generate logic for the Model and refresh its User Interface.
  2. Click **Test Model** and select a User Interface. When it appears, the UI contains a field for the value of the Text Feature and a button (whose default caption is **Add Another**) for adding new instances of the instantiable Component.
  3. Click the button to add a new instance of the Component. This action is handled by the runtime Oracle Configurator as a `postInstanceAdd` event, which triggers the Configurator Extension, which is bound to that event.
  4. The runtime Oracle Configurator changes the name of the new instance of the Component to the value of the Text Feature.
  5. Change the value of the Text Feature, then add another instance of the Component. The new text value is used to name the new instance.

2.4 Suggested Development Practices

This section contains an assortment of suggestions for developing Configurator Extensions more efficiently and conveniently.

2.4.1 Observing Project Requirements

Using Configurator Extensions and the CIO allows you to build very powerful applications with Oracle Configurator. There are important requirements that you should fulfill if you want to maximize your success with Configurator Extensions.

- The programmers developing the Java code must possess the requisite skills. See Section 1.2, "Prerequisite Skills for Developing Configurator Extensions" on page 1-2 for a description.

- You must develop a test plan for your Configurator Extensions, including a way to isolate problems caused by them. You need to test your Configurator Extensions early and often.

  If you contact Oracle Support Services (as described in Product Support on page xvi), you will be asked to reproduce the problem without the Configurator Extensions. If it is impossible to reproduce the problem without Configurator Extensions, you will need to explain why you believe your code is not the cause of the problem. See Section 2.4.7, "Disabling Configurator Extensions" on page 2-10 for information on features that enable you to isolate the effects of your Configurator Extensions.

2.4.2 Avoiding Common Errors

Observe the following guidelines to avoid common coding errors:

- Ensure that any static variables and methods are thread-safe. Be aware that CIO interfaces are not thread-safe. See Section 1.4.2, "Restrictions for Configurator Extensions" on page 1-4 for details.
- Use one transaction per CIO operation. See Chapter 7, "Using Logic Transactions" for details.
- Handle exceptions properly and avoid empty catch blocks. See Section 2.4.3, "Handling Exceptions Properly" on page 2-8.

2.4.3 Handling Exceptions Properly

Improper handling of exceptions is the source of many problems that are difficult to diagnose. See Section 8.3, "Handling Exceptions" on page 8-5 for more information.

Do not ignore or swallow exceptions raised by your code. Ignoring exceptions makes it very difficult to determine the cause of some problems. Handling exceptions properly is sound Java coding practice.

Never leave a catch block empty, as shown in Example 2–2. The empty catch block causes your code to silently ignore the exception. The program may then fail at some later point that is quite unrelated to the source of the problem, making it very hard to analyze.

Example 2–2 Empty Catch Block

```
...
    try {
        opt1.setState(IState.TRUE);
    }
    catch (LogicalException le) {
   // an empty catch block ignores exceptions
    }
...
```

This advice applies to both checked exceptions (such as predictable user errors) and unchecked exceptions (unpredictable program failures). Checked exceptions should always be handled, as shown in Example 2–3. Leaving a catch block empty is worse than not catching an unchecked exception at all, since an unhandled unchecked exception (with no catch block at all) causes the program to fail and preserves some failure information for debugging.

Example 2–3 Catch Block That Handles an Exception

```
...
    try {
        opt1.setState(IState.TRUE);
    }
    catch (LogicalException le) {
    // the exception is handled
        throw new RuntimeException("Error");
    }
...
```

2.4.4 Avoiding Circularity and Recursion

Avoid coding that results in circularity or recursion. Scenarios that might cause this are described in:

- Example of Circularity
- Example of Recursion

Example of Circularity

You might unintentionally define Configurator Extensions that call each other in a circular chain.

For example, you might bind the `postValueChange` event to a method that increments the value of a node, and also to some other method that increments the value of the same node. At runtime, the change to the node made by one method triggers the other method, which changes the node again, and triggers the first method. The resulting endless loop of value changes results in a stack overflow. You can determine whether this occurred by checking the stack trace.

This kind of scenario can also occur with the `onConfigValidate` event, which is dispatched during the validation performed after every CIO transaction.

Example of Recursion

You might unintentionally invoke a method that calls itself recursively in an endless loop.

For example, you might bind the method `setIntegerValue()` in Example 2–4 on page 2-9 to the `postValueChange` event. (You would also bind its `node` parameter to an Integer Feature, and its `config` parameter to the system parameter `Configuration`, with an event scope of `Base Node`.)

Example 2–4 Inadvertent Recursion (RecursionExample.java)

```
import oracle.apps.cz.cio.IInteger;
import oracle.apps.cz.cio.Configuration;
import oracle.apps.cz.cio.ConfigTransaction;
import oracle.apps.cz.cio.LogicalException;

public class RecursionExample {

    public void setIntegerValue(IInteger node, Configuration config) {
        ConfigTransaction tr = config.beginConfigTransaction();
        try {
            int val = node.getIntValue();
            node.setIntValue(val + 1 ); // no limit to setting values
            config.commitConfigTransaction(tr);
        } catch(LogicalException le) {
            le.getCause();
        }
    }
}
```

The `setIntegerValue()` method changes the value of the specified node inside a transaction (which is sound practice). However, every time a transaction is committed, the CIO traverses the list of changes to the configuration (as described in Section 8.1, "Validating Configurations" on page 8-1) and detects the change to the node, and this change triggers the `postValueChange` event, which calls the `setIntegerValue()` method again, in a loop.

To avoid this recursion, you must place a limit on the `setIntegerValue()` method, such as the following:

```
if (val < 100) { node.setIntValue(val + 1 ); } // limit to setting values
```

At runtime, this method increments the value of the Integer Feature until it reaches 100, and then stop.

2.4.5 Taking Advantage of Argument Binding

Try to make your code simple and reusable by taking advantage of the power of argument binding.

- When you want to get a node for processing, do not use *`node`*`.getChildByName()`. Instead, you can simply bind the desired node to a method parameter in Oracle Configurator Developer.
- When you only need one Property of a node, do not bind the node. Instead, bind the Property. For example, if you need the name of the node *`node`*, then bind to the System Property *`node`*`.Name()` instead of binding *`node`* itself and calling *`node`*`.getName()` in your code.

2.4.6 Sharing Class Instances

All the bindings on a single Configurator Extension Rule share an instance of a class. This means that any member variable can be shared.

You can group bindings based on their intended functionality or based on their class usage, and incur less overhead in the creation of objects.

If your Configurator Extension class uses static member variables to communicate between different instances of the class, the variables cannot be shared across configurations of different models. For example, a Configurator Extension Rule whose base node is in Model `M1` will not be able to share static member variables with a Configurator Extension Rule whose base node is in the Model `M2` even if both Configurator Extensions are bound to the same configurator extension class, `MyClass`.

2.4.7 Disabling Configurator Extensions

When debugging problems with Oracle Configurator, it is sometimes very helpful to disable some or all of your Configurator Extensions. Disabling Configurator Extensions shows whether the likely source of a problem is in your Configurator Extensions or in the Model that they are associated with. If the problem disappears when you disable Configurator Extensions, then the problem is likely to be in your code. If the problem persists, then the problem is likely to be in your model structure or configuration rules.

- To disable one or more individual Configurator Extensions, navigate to the Rules area of the Workbench in Oracle Configurator Developer. Then edit the Configurator Extension Rule and select its **Disable** check box, which disables only that Rule. See the *Oracle Configurator Developer User's Guide* for details.
- To disable many or all Configurator Extension Rules for a Model, navigate to the Rules area of the Workbench in Oracle Configurator Developer. Then select the rules, or a folder of rules, and select **Disable** from the **Actions** list. See the *Oracle Configurator Developer User's Guide* for details.
- To disable all Configurator Extensions in your runtime Oracle Configurator, set the `nofc` switch of the Oracle Configurator Servlet property `cz.activemodel`. See the *Oracle Configurator Installation Guide* for details on setting this switch.

  This setting overrides the settings in Oracle Configurator Developer.

  This setting also disables Functional Companions for Models that have already been published.
- To programmatically disable all Configurator Extensions and Functional Companions in your runtime Oracle Configurator, use `CIO.setActiveModelPath()`:

  ```
  // set on your CIO object
  cio.setActiveModelPath("/nofc");
  ```

  See Section 4.3, "Initializing the CIO" on page 4-4 for background on the `CIO` object.

2.4.8 Testing for a Null User Interface

If a Configurator Extension might be used with both DHTML UIs (created with the previous release of Oracle Configurator Developer) and generated UIs (created with the HTML-based version of Oracle Configurator Developer), then you should always test for the existence of a DHTML UI. This can also be a way to check which type of UI is in use.

To test for the existence of a DHTML user interface, call `Configuration.getUserInterface()`, as shown in Example 2–5. If the test occurs when the runtime configuration is rendered in a generated UI, then it always returns null.

Example 2–5 Testing for a Null User Interface

```
...
mUi = this.getRuntimeNode().getConfiguration().getUserInterface();
if (mUi != null) { // the UI is DHTML }
else { // the UI is generated }
...
```

2.4.9 Using Logging to Examine Problems

When debugging problems with Oracle Configurator, it is very helpful to examine the log file entries created by the CIO during a runtime configuration session. You can insert statements in your code to specify how the entries are written. See Chapter 11, "Logging Through the CIO" for details.

2.4.10 Checking for Deleted or Discontinued Nodes

When working with a runtime node that might have been deleted during a configuration session, always call `IRuntimeNode.isDeleted()` to test whether that node is actually deleted. Attempting to access or set some attribute of a deleted node generates a `NodeDeletedException` at runtime. Some methods commonly used to work with nodes are `getState()`, `setState()`, and so on.

If a configured instance of your Model might contain discontinued nodes, then you should also call `IRuntimeNode.isDiscontinued()` as a condition of working with a node. A discontinued node is one that exists in an installed configuration of a component (as recorded in Oracle Install Base), but has been removed from the instance of the component being reconfigured, either by deletion or by deselection. If a node has been discontinued by deselection, but not by deletion, then calling a method on it will not raise a `NodeDeletedException`.

For examples of situations in which you might need to test for deleted or discontinued nodes, see the following sections:

- Section 6.4, "Getting and Setting Logic States" on page 6-5
- Section 6.5, "Getting and Setting Numeric Values" on page 6-7
- Section 6.7, "Access to Options" on page 6-9, which includes a code example, Example 6–7, "Testing Whether an Option Is Selected" on page 6-10

3 Uses for Configurator Extensions

This chapter describes some possible uses for Configurator Extensions.

The sections of this chapter are:

- Types of Configuration Events
- Generating Custom Output
- Filtering for Connectivity

3.1 Types of Configuration Events

Every Configurator Extension must be bound to some configuration event. Therefore, you should review the available events to help determine the situations in which you can employ a Configurator Extension.

While there are no formal types for Configurator Extensions themselves, it is possible to categorize the configuration events to which you can bind Configurator Extensions. Table 3–1, " Types of Configuration Events" on page 3-1 lists the available types of configuration events and an example event for each type. For a list of events that you can use for processing configurations, see Table 5–3, " Events for Processing Configurations" on page 5-9. For more details, and a full list of the available events, see the chapter on Configurator Extensions in the *Oracle Configurator Developer User's Guide*.

Table 3–1 Types of Configuration Events

Event Type	Possible Use	Example Event
Configurator Extension	Triggering actions that are required when the base node for a Configurator Extension Rule is instantiated.	`postCXInit`
Connection	Filtering valid targets for a Connector.	`onValidateEligibleTarget`
Custom Command	Processing custom command strings that you define. Required when generating custom output.	`onCommand`
Session	Triggering actions that are required at some specified point in a configuration session.	`postConfigInit`
Value-Related	Validating selections or values.	`onConfigValidate`

3.2 Generating Custom Output

You can generate custom output that is displayed when the end user clicks a button in the UI of the runtime Oracle Configurator.

The Configurator Extension for this task must be bound to the `onCommand` event with a custom command string that you define. This custom command is handled by the UI layer for the runtime Oracle Configurator. The other requirement is that your Java method must take an argument of type `HttpServletResponse`.

For the detailed procedure for creating a Configurator Extension Rule, see Chapter 2, "Building Configurator Extensions" and the related sections of the *Oracle Configurator Developer User's Guide*. A summary of the required tasks is provided here, with additional explanation where necessary.

1. The Java method for your Configurator Extension class must take an argument of the type `javax.servlet.http.HttpServletResponse`. You must use this data type because it is the location where your Configurator Extension generates custom output.

   An example of a very simple custom output class is shown in Example 3–1, "Generating Custom Output (HelloWorldCX.java)" on page 3-3. The example prints a simple message in an HTML page.

2. Compile the Java class for your Configurator Extension and place it in a Java class archive file.

3. Create a Configurator Extension Archive for the class, and add it to the Archive Path for your Model.

4. Define a Configurator Extension Rule with the options listed in the following table:

Option	**Choose ...**
Model Node	The node of your Model on which you want the button for the command event to be placed by Oracle Configurator. This node is independent of the node to which you might bind an argument whose Argument Specification is **Model Node or Property.**
Java Class	`HelloWorldCX`, selected from your Configurator Extension Archive.
Java Class Instantiation	With Model Node Instance

5. Create an event binding for the Configurator Extension Rule with the options listed in the following table:

Option	**Choose ...**
Event	`onCommand`
Command Name	A string that you choose as a command. For example: `Say Hello`. Do not enclose the string in quotation marks. The string can contain spaces.
Event Scope	Your choice of scope. Try repeating the example with different scopes to observe the effect when you test it each time.
Method	`helloWorld`

6. Create an argument binding for the event binding with the options listed in the following table:

Option	Choose ...
Argument Type	`javax.servlet.http.HttpServletResponse`
Argument Specification	Event Parameter
Binding	`HttpServletResponse`

7. Generate logic for your Model, to reflect the addition of the Configurator Extension Rule.

8. Create or refresh a User Interface for your Model. This creates a button in the User Interface that by default is captioned with the Command Name that you specified in the binding for the `onCommand` event. The button is placed on the page for the Model Node that you associated with the Configurator Extension (the base node).

   To change the default caption of the button, edit the **Text Expression** field in the **Caption Source** for the button.

   The **Button Action** for the button is automatically set by Oracle Configurator Developer to use an **Action Type** of **Raise Command Event** in which the Command is the Command Name string in your event binding. The fact that these command strings are the same is what causes the button to invoke the Java class for your Configurator Extension. If you change the Command Name string in your event binding, you must also change it for the **Raise Command Event**.

9. Test the Configurator Extension from Configurator Developer by choosing the **Test Model** button, then choosing the Model Debugger, or the User Interface that you generated. When you click the button that triggers the Configurator Extension, it produces a secondary window and writes the specified message in it.

   You can modify the characteristics of the secondary window in Configurator Developer. The Action Parameters for the Button element include an Output Window Options field, into which you can enter HTML attributes for the window. See the *Oracle Configurator Developer User's Guide* for information on editing User Interface elements.

10. For another example of generating output, see Example B–1 in Section B.1, "Generating Output Related to Model Structure" on page B-1.

Keep the following in mind when working with custom output:

- If you bind multiple Configurator Extensions to the same command event, they share the same Button element in the User Interface. When you click that button in the runtime Oracle Configurator, it triggers all those bound Configurator Extensions.

- If you use the limited edition of Oracle Configurator Developer to create a DHTML UI for a Model that already contains multiple Configurator Extension command bindings, then it generates a Button for each command binding. However, when you click a button in the runtime Oracle Configurator, only the first Configurator Extension runs.

Example 3–1 Generating Custom Output (HelloWorldCX.java)

```
import java.io.IOException;
import javax.servlet.http.HttpServletResponse;
```

```
// This CX does not use the CIO, so no need to import CIO classes

public class HelloWorldCX {

    public HelloWorldCX() {
    }

    public void helloWorld(HttpServletResponse resp) {
        StringBuffer sb = new StringBuffer(511);

        sb.append("<html>");
        sb.append("<head>");
        sb.append("<title>Simple CX Test</title>");
        sb.append("</head>");
        sb.append("<body bgcolor='#FFFFFF' text='#000000'>");
        sb.append("HELLO WORLD. This is output from a Configurator Extension.");
        sb.append("</body>");
        sb.append("</html>");
        resp.setContentType("text/html");
        resp.setHeader ("Expires", "-1");  // required for MSIE
        try {
            resp.getWriter().println(sb.toString());
        }
        catch (IOException ioe) {
            throw new RuntimeException();
        }
    }
}
```

3.3 Filtering for Connectivity

You can define a Connection Filter Configurator Extension that filters the instances of a target Model that are displayed when an end user of the runtime Oracle Configurator clicks a **Choose Connection** button.

3.3.1 Defining a Connection Filter Configurator Extension

To define a Connection Filter Configurator Extension:

1. Define a Java class for your Configurator Extension.

   See Section 2.2, "Developing Java Classes and Archives" on page 2-4 for the basic procedure. See Section 3.3.3, "Example of a Connection Filter Configurator Extension" on page 3-5 for example code.

2. Define a method that determines the criteria for filtering a list of valid targets for a Connector.

   Example 3–2 on page 3-5 defines such a test in the body of `validateEligibleTarget()`.

3. In Oracle Configurator Developer, define a Configurator Extension Rule, and create a binding for the `onValidateEligibleTarget` event.

   Bind the Event Parameter named `target` as the argument to the parameter of your `validateEligibleTarget()` method named `target`.

   Bind the Event Parameter named `connector` to the Connector node whose target instances you want to filter.

See the *Oracle Configurator Developer User's Guide* for information about connectivity and creating Connectors.

3.3.2 Behavior of Connection Filter Configurator Extensions

In the runtime Oracle Configurator, when the end user clicks a **Choose Connection** button, Oracle Configurator gets the list of all target instances of the Connector, then invokes any Configurator Extension bindings that are listening for the `onValidateEligibleTarget` event on this Connector. If any of these bindings return `false`, then that instance is removed from the list of potential targets, and is not displayed in the Connection Chooser.

- If there are no target instances that satisfy the filter, then Oracle Configurator displays a notification of that fact to the end user.
- The same Connection Filter Configurator Extension can be associated with more than one Connector. The same filtering test is performed, but because the potential targets of the Connectors may be different, the resulting set of eligible instances may also be different.
- Different Connection Filter Configurator Extensions can be associated with the same Connector, for example:
  - Model_A includes Connector_A
  - In Model_A, Configurator Extension CX_1 is associated with Connector_A
  - Model_A is referenced in Model_B (and so Connector_A is accessible through the reference)
  - In Model_B, Configurator Extension CX_2 is associated with Connector_A

  In the runtime Oracle Configurator, when the end user clicks the **Choose Connection** button for Connector_A, Oracle Configurator displays a Connection Chooser containing all of the target instances that satisfy both CX_1 and CX_2.

3.3.3 Example of a Connection Filter Configurator Extension

For an example of a Connection Filter Configurator Extension, see Example 3–2 on page 3-5. This Configurator Extension searches the target Model for a Resource named `Resource1`, and returns False if the value of that Resource is less than 10; otherwise it returns True.

In the runtime Oracle Configurator, this Configurator Extension filters out any potential target instances in which the value of the Resource named `Resource1` is less than 10. (If the potential target instance does not even contain a Resource named `Resource1`, then a `NoSuchChildException` is raised.)

Example 3–2 Filtering for Connectivity (TargetFilter.java)

```
import oracle.apps.cz.cio.Resource;
import oracle.apps.cz.cio.Component;
import oracle.apps.cz.cio.NoSuchChildException;

public class TargetFilter {

  public boolean validateEligibleTarget(Component target){
    Resource resource = null;
    try {
      resource = (Resource)target.getChildByName("Resource1");
    } catch (NoSuchChildException nsce) {
      nsce.printStackTrace();
      return true;
    }
    if (resource.getValue() < 10) {
      return false;
    } else {
      return true;
    }
  }
}
```

Part II

The Configuration Interface Object (CIO)

Part II contains the following chapters:

4 CIO Basics

This chapter provides basic information about the Oracle Configuration Interface Object (CIO). For details about how to use the CIO for specific purposes, see the list of chapters in Part II, "The Configuration Interface Object (CIO)".

The sections of this chapter are:

- Background
- The CIO's Runtime Node Interfaces
- Initializing the CIO

4.1 Background

This section describes the CIO and its relationship to Configurator Extensions.

4.1.1 What is the CIO?

The Configuration Interface Object (CIO) is an API (application programming interface) that provides programs access to the Model used by a runtime Oracle Configurator, which you construct with Oracle Configurator Developer. The CIO is designed to enable you to programmatically perform any interaction with a configuration model that can be interactively performed by an end user during a configuration session.

The CIO is a top-level configuration server. The CIO is responsible for creating, saving and destroying objects representing configurations, which themselves contain objects representing Models, Components, Features, Options, Totals and Resources. The runtime configuration model can be completely controlled and manipulated through these interfaces, using methods for getting and setting logical, numeric and string values, and creating optional subcomponents.

Client Applications

The CIO is the only API supported by Oracle for programmatic interaction with the runtime Oracle Configurator. Consequently, any custom applications must use the CIO. Custom applications are those that integrate Oracle Configurator with a custom user interface (a UI not generated by Oracle Configurator Developer).

The CIO is also used by Configurator Extensions, as described in Section 1.5, "Configurator Extensions and the CIO" on page 1-4. Be sure to review the *Oracle Configurator Performance Guide* for information on the performance impacts of Configurator Extensions.

Most of the techniques for using the CIO apply equally to custom applications and Configurator Extensions. This document points out selected cases where there is a distinction between these two applications.

Implementation Language

The Oracle Configuration Interface Object is written in Java, and implemented as the Java package `oracle.apps.cz.cio`. To use the functionality of the CIO you must import classes from this package.

Note: Unless stated otherwise, references in this document to classes, methods, and properties refer to the package `oracle.apps.cz.cio`, and all code examples are in Java.

4.1.2 The CIO and Configurator Extensions

A Configurator Extension is Java code that calls the CIO.

Configurator Extensions are invoked by the CIO through the runtime Oracle Configurator, and Configurator Extensions call the CIO to get information from the running Model. The CIO is like a broker for the runtime Oracle Configurator, in that it passes information both ways. Programmers writing Configurator Extensions need to know how to use the CIO.

Each Configurator Extension is an object class. For every component instance in your Model that is associated with a Configurator Extension, the CIO creates an instance of this class.

4.2 The CIO's Runtime Node Interfaces

When you program against the CIO, you create one instance of the class `CIO` (see Section 4.3, "Initializing the CIO" on page 4-4) and one or more instance of the classes `Configuration` and `ConfigParameters` (see Chapter 5, "Working with Configurations"). You then use the public interfaces of the CIO, such as those listed in Table 4–1 on page 4-3, to access fields in the runtime node objects created by your instances of `CIO` and `Configuration`. Apart from `CIO` and `Configuration`, your code should refer only to these public runtime node interface objects. You should not implement any of the runtime node interfaces, but only use them as references to runtime node objects.

In Java, an interface is a special type that allows programmers more flexibility in the way that they implement the internal details of classes. In Java terms, an interface is a named collection of method definitions, without implementations of those methods. For example, in the CIO, the interface `IRuntimeNode` specifies methods that are implemented in the class `RuntimeNode`.

Note: In normal circumstances, the only CIO classes that you should create (with the Java keyword `new`) are:

- `CIO`
- `Configuration`
- `ConfigParameters`

You only need to create these objects when working with a custom application. Configurator Extensions do not need to create them, because that task is performed by the runtime Oracle Configurator when it starts a configuration session.

Table 4–1 on page 4-3 lists some of the interfaces defined in the Java package `oracle.apps.cz.cio` that you are most likely to use in working with the CIO. For more detail about these and the other CIO interfaces, see Appendix A, "Reference Documentation for the CIO".

Table 4–1 Important Runtime Node Interfaces for the CIO

Interface	Role of implementing classes
`Component`	Interface for components.
`IBomItem`	Implemented by all selectable BOM items.
`ICount`	Implemented by objects that have an associated integer count.
`IDecimal`	Implemented by objects that can both get and set a decimal value.
`IInteger`	Implemented by objects that have an integer value.
`IOption`	Implemented by objects that act as options. The defining characteristic of an option is that it can be selected and deselected.
`IOptionFeature`	Implemented by objects that contain selectable options. This interface provides a mechanism for selecting and deselecting options, and for determining which options are currently selected.
`IRuntimeNode`	Implemented by all objects in the runtime configuration tree. This interface implements behavior common to all nodes in the runtime configuration tree, including Components, Features, Options, Totals, and Resources.
`IState`	Implemented by objects that have logic state. This interface contains a set of input states, used to specify a new state for an object, a set of output states, returned when querying an object for its state, and a set of methods for getting and setting the object's state.
`IText`	Implemented by objects that have a textual value.

The functionality underlying the CIO interfaces is implemented by other classes in `oracle.apps.cz.cio`, which are subject to revision by Oracle. This interface/implementer architecture protects your code from the effects of such revisions, since the interfaces remain constant.

4.3 Initializing the CIO

In order to use any of the features of the CIO, an application must initialize it, using a JDBC driver to make a connection to the Oracle Configurator schema. This connection enables the CIO to obtain and store data about Model structure, Configuration Rules, and User Interface.

Note:

- This use of the CIO is intended for custom applications. If you are using the CIO in a custom application, you must initialize the CIO.
- When you run Configurator Extensions through the runtime Oracle Configurator or through the testing facilities of Oracle Configurator Developer, this initialization and connection work is automatically handled for you; you do not have to write your own code to initialize the CIO.

Use the following practice to initialize the CIO:

1. Import the necessary packages.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

import oracle.apps.cz.cio.*;
import oracle.apps.cz.common.*;
```

2. Load the database driver that you have installed. For instance:

```
Class.forName("oracle.jdbc.driver.OracleDriver");
```

3. Create a context object and pass to it the information needed to make a database connection: the host name and port number of the Web server, and the name of the DBC file. The context object manages the database connection; you should *not* create a separate connection object (for instance, with `java.sql.DriverManager.getConnection`).

```
contextObject = new CZWebAppsContext ("hostName", "portNumber", "dbcFileName");
```

4. Create a single global CIO object. This object is shared by any `Configuration` objects that are created during the configuration session.

```
CIO cioObject = new CIO();
```

Example 5–1 on page 5-3 shows how some of these steps are employed.

5 Working with Configurations

This chapter describes how to interact with configuration objects.

The sections of this chapter are:

5.1 Overview

The Configuration object, `oracle.apps.cz.cio.Configuration`, represents a complete configuration. You can use the CIO to work with multiple configurations within the same configuration session.

For essential background information about Configuration objects, see the chapter on managing configurations in the *Oracle Configurator Implementation Guide*.

You communicate with a runtime configuration through the Configuration object, using methods such as those listed in Table 5–1:

Table 5–1 Typical Methods of the Configuration Object

To do this ...	Use this method of `Configuration` ...
Access the CIO object that contains the Configuration object	`getCIO()`
Access the component object for which the Configuration object represents a configuration	`getRootComponent()`
Obtain a collection of current validation failures	`getValidationFailures()`

Obtain an indication of whether the complete configuration is satisfied	`isUnsatisfied()`
	`getUnsatisfiedItems()`

Obtain a collection of the selected nodes in the configuration	`getSelectedItems()`

Table 5–1 (Cont.) Typical Methods of the Configuration Object

To do this ...	Use this method of `Configuration` ...

Save the current configuration	`saveNew()`
	`saveNewRev()`

Close the current configuration	`closeConfiguration()`

The Configuration object also provides methods for starting, ending, and rolling back logic transactions performed on a configuration. Logic transactions maintain logic consistency; they are not database transactions. See Chapter 7, "Using Logic Transactions" on page 7-1.

5.2 Creating Configurations

Note: This use of the CIO is intended for custom applications. Configurator Extensions do not need to create a `Configuration` object, because that task is performed by the runtime Oracle Configurator when it starts a configuration session.

To create a Configuration object, which is the top-level entry point to a configuration, use `CIO.startConfiguration()`.

Note: The use of `CIO.startConfiguration()` completely replaces the use of all versions of `CIO.createConfiguration()`, which is now deprecated. Existing code that uses the deprecated method is still compatible with the CIO, but cannot use any new functionality.

This method takes as arguments a `ConfigParameters` object and a context object.

The context object provides the application context for the connection to the database. See Section 4.3, "Initializing the CIO" on page 4-4 for information on creating a context object.

The `ConfigParameters` object encapsulates all the information needed to create a configuration. To create a `ConfigParameters` object, invoke one of the constructors for `ConfigParameters`, depending on the type of configuration you need to create:

- To create an entirely new configuration, provide a Model ID:

  ```
  public ConfigParameters(int modelId)
  ```

  This is the constructor shown in Example 5–1 on page 5-3.

- To restore a saved configuration, provide its Configuration Header ID and Configuration Revision Number.

  ```
  public ConfigParameters(long headerId, long revisionNumber)
  ```

- To create a configuration for a BOM without a configuration model (sometimes known as a "native BOM" configuration), provide the Inventory Item ID, Organization ID, and effective date of the BOM to be exploded and configured:

  ```
  public ConfigParameters(int inventoryItemId, int organizationId, Date
  explosionDate)
  ```

To control the initialization of the new configuration, use the methods in the `ConfigParameters` class to set the configuration parameters. For details on these methods, see the reference for the CIO (described in Chapter A, "Reference Documentation for the CIO").

Use the methods in the following list to set the effective date for the configuration and the model's publication lookup date.

- `setEffectiveDate(java.util.Calendar effectiveDate)`
- `setModelLookupDate(java.util.Calendar modelLookupDate)`

If you do not set these dates, they default to the date when Oracle Configurator considers the configuration to have been created.

All other parameters to the `ConfigParameters` object are optional, and are defaulted.

Once a configuration has been created, changing a configuration parameter does not affect the configuration in any way.

To obtain access to the CIO object that created the configuration, use `Configuration.getCIO()`.

Most of the constructor and method arguments to `ConfigParameters` correspond to one of the initialization parameters for the runtime Oracle Configurator. The correspondences are shown in Table 5–2 on page 5-3. See the *Oracle Configurator Implementation Guide* for more information on the initialization parameters.

Table 5–2 Correspondence of Configuration Parameters to Initialization Parameters

Configuration Parameter	Argument	Initialization Parameter
Model ID	`modelId`	`model_id`
Configuration Header ID	`headerId`	`config_header_id`
Configuration Revision Number	`revisionNumber`	`config_rev_nbr`
Inventory Item ID	`inventoryItemId`	`inventory_item_id`
Organization ID	`organizationId`	`organization_id`
Configuration Effective Date	`effectiveDate`	`config_effective_date`
Model Lookup Date	`modelLookupDate`	`config_model_lookup_date`

Example 5–1 shows a technique for creating a Configuration object. For clarity, it omits some important tasks, such as using transactions and fully handling exceptions.

Example 5–1 Creating a Configuration Object (MyConfigCreator.java)

```
import oracle.apps.cz.cio.CIO;
import oracle.apps.cz.cio.ConfigParameters;
import oracle.apps.cz.cio.Configuration;
import oracle.apps.cz.cio.IRuntimeNode;
import oracle.apps.cz.cio.IState;
import oracle.apps.cz.cio.IOption;
import oracle.apps.fnd.common.Context;
import oracle.apps.cz.common.CZWebAppsContext;
import java.util.Calendar;

public class MyConfigCreator {
```

```
// Create the context object for this instance
    private static String hostName = "myhost";
    private static String portNumber = "1521";
    private static String dbcFileName = "myhost_mysid";
    private static CIO cio;
    private static Context context;

    public static void main(String [] args) {

        context = new CZWebAppsContext( hostName, portNumber, dbcFileName );
        CIO cio = new CIO(); // Create shared global CIO
        MyConfigCreator work = new MyConfigCreator();

        // Create a configuration object, using the shared CIO
        work.config1();

        // Use the same shared CIO to create more configurations
        // work.config2();
        // work.config3();
        // and so on ...
    }

    // Create and use a new Configuration object
    public void config1() {

        // Create the ConfigParameters object and set non-default parameters
        int modelId = 5005; // Hypothetical model ID
        ConfigParameters cp = new ConfigParameters(modelId);
        java.util.Calendar modelLookupDate =  Calendar.getInstance(); // current date and time
        cp.setModelLookupDate(modelLookupDate);

        try {

        // Create the Configuration object
        Configuration config = cio.startConfiguration(cp, context);

        // Perform an assertion against the configuration ...
        // 1. Get the root component of the configuration
        IRuntimeNode rootComp = (IRuntimeNode)config.getRootComponent();
        // 2. Get an Option Feature
        IRuntimeNode of1 = rootComp.getChildByName("option_feature_1");
        // 3. Select an Option of the Feature
        ((IOption)of1.getChildByName("option_1")).select();

        // Perform other assertions ...

        } catch (Exception e) {
          // Perform exception handling here
        }
    }
}
```

5.3 Removing Runtime Configurations

To remove all runtime structure and memory associated with a configuration, use `CIO.closeConfiguration()`. Oracle recommends that you invoke this method when ending a configuration session and before exiting the runtime Oracle Configurator.

5.4 Saving Configurations

You save a runtime configuration so that you can operate on it later, after it has been closed at the end of a configuration session.

When you save a configuration, it is stored in the CZ schema of the Oracle Applications database. To later operate on a saved configuration, you must first restore it, as described in Section 5.6 on page 5-6.

There are several methods for saving configurations. Choose the one that suits your requirements, as described in the following list.

- Use `Configuration.saveNew()` to save an entirely new Configuration object.

  The saved Configuration object has a new Configuration Header ID and a Configuration Revision Number of 1.

- Use `Configuration.saveNewRev()` to save a new revision of a previously saved Configuration object.

  The saved Configuration object has the same Configuration Header ID as the previously created Configuration object, but the Configuration Revision Number uses the next available Revision Number.

- Use `Configuration.save()` to save subsequent changes to a previously saved Configuration object, overwriting the existing configuration data.

  The saved Configuration object has the same Configuration Header ID *and* the same Configuration Revision Number as the previously created Configuration object.

- For more information on saving configurations, see the *Oracle Configurator Implementation Guide*.

Note: Do not save a Configuration object during a logic transaction (see Chapter 7, "Using Logic Transactions"). You may miss some validation messages that are not available until the transaction is committed.

5.5 Monitoring Changes to Configurations

When changes are made to a configuration, the CIO monitors whether the configuration needs to be saved. You can access the flag that tracks this status.

5.5.1 How the CIO Monitors Changes to Configurations

During a runtime configuration session, the CIO monitors whether changes have been made to the current configuration, and whether those changes need to be saved. Changes can result either from end user actions in the user interface of the runtime Oracle Configurator, or from assertions made through the CIO by your Configurator Extensions or custom application code.

To keep track of whether a configuration needs to be saved, the CIO maintains a Boolean changed-state flag, whose values are interpreted as "clean" or "dirty". At the beginning of a configuration session, the flag is set according to the following rules:

- Any new configuration having no assertions against it is marked as clean.
- Any restored configuration having no assertions against it is marked as clean, regardless of whether it produces validation failures when restored.

- Any new or restored configuration with assertions against it is marked as dirty.

During the configuration session, if there are unsaved changes, then the changed-state flag is set to dirty by the CIO.

When the configuration is saved, the changed-state flag is set to clean. It does not matter how the saving is performed: by a Configurator Extension or by a custom user interface.

When the Cancel button is clicked in the user interface of the runtime Oracle Configurator, the UI Server checks the changed-state flag; if it is dirty, the UI Server produces a dialog asking the user whether to continue exiting the session without saving the changes. If you write a custom user interface, it should do the same, using the technique described in Section 5.5.2, "How You Can Monitor Changes to Configurations" on page 5-6.

5.5.2 How You Can Monitor Changes to Configurations

You can get or set the value of the changed-state flag of a configuration.

- To get the value of the changed-state flag, use the method `Configuration.areAllChangesSaved()`.

  This method returns TRUE the configuration is clean (that is, if all the changes that have been made to this configuration during the configuration session have been saved). This method returns FALSE if the configuration is dirty (that is, if there are changes that have been made to this configuration that have not been saved).

  You can use this method when you want to determine whether a configuration needs to be saved.

- To set the value of the changed-state flag, use the method `Configuration.setAllChangesSaved()`, which takes the boolean argument `clean`.

  If you pass TRUE as the value of `clean`, then the changed-state flag is set to "clean". Any further changes to the configuration make it dirty again. If you pass FALSE as the value of `clean`, then the changed-state flag is set to "dirty".

  You can use this method when you want to change the configuration through the CIO without interfering with the end user’s sense of what has changed during a configuration session. For example, if you use a Configurator Extension to create and rename of an instance of an instantiable component when the configuration is created, the changed-state flag is set to dirty. You can then use `setAllChangesSaved()` to set the flag to clean, so that if the end user clicks the Cancel button before making any changes, the UI Server does not produce the dialog asking whether to continue exiting the session without saving changes.

5.6 Restoring Configurations

You restore a configuration in order to operate on if it has been saved and closed (as described in Section 5.4, "Saving Configurations" on page 5-5).

- To restore a Configuration object from the Oracle Configurator schema, use `CIO.startConfiguration()`. For details about that method, see Section 5.2 on page 5-2 and Example 5–1 on page 5-3.

Note: The use of `CIO.startConfiguration()` completely replaces the use of all versions of `CIO.restoreConfiguration()`, which is now deprecated. Existing code that uses the deprecated method is still compatible with the CIO, but cannot use any new functionality.

- When you restore a configuration, any user requests (see Section 9.3, "User Requests" on page 9-2) that cannot be applied are reported as validation failures. See Section 9.5, "Failed Requests" on page 9-4.
- You may be able to improve performance by restarting the current configuration, instead of restoring it. See Section 5.7, "Restarting Configurations" on page 5-8.
- You must be aware of the possible effects of changing the model structure or configuration rules in Oracle Configurator Developer between the time you save a configuration and the time you restore it.
- If you change the Instantiability settings for a Model or Component to decrease or increase the Initial Minimum, this might change the number of previously saved instances that exist when restore a saved configuration. Unmodified initial instances are restored in the order they were initially created, until they possibly exceed the Initial Minimum. However, no instances that you modify or add will be lost.

  Here is an example of the preceding point:

  1. Define the Initial Minimum of an instantiable component as 5.
  2. Create a configuration. The Initial Minimum of 5 is enforced, instantiating that number of components.
  3. Modify 2 of the initially instantiated components. For instance, make them targets of Connectors, or select options of their children.
  4. Add 1 new component instance, and delete 1 initial instance.

     There are now 5 instances: 2 modified initial instances, 2 unmodified initial instances, and 1 added instance.
  5. Save the configuration. All 5 instances are saved.
  6. Change the Initial Minimum of the instantiable component to 3.
  7. Restore the saved configuration.

     The following 4 instances are restored:

     – The 1 added instance (because added instances are always restored). Added instances are not counted against changes in the Initial Minimum.

     – The 2 modified initial instances (because modified instances are always restored).

     – Only the first 1 of the unmodified initial instances (because the other 1 unmodified initial instance exceeds the new Initial Minimum of 3, and is not restored).

     Only unmodified instances can be lost when a configuration is restored. Any modified or added instances are restored, regardless of the Initial Minimum.

     If the Initial Minimum is increased, then the configuration might be restored with more instances than were saved.

- Remember that it is only the User True configuration inputs to the model that are saved, not all the Logic True effects that those inputs may have when reapplied later. When you restore a configuration, any user requests that cannot be applied are reported as validation failures. Consequently, you should notify end users of changes to your configuration model or rules.

  Here is an example of the preceding point:

  1. Define a Logic Rule stating that Option1 Requires Option2.
  2. In a configuration session, the end user selects Option1, which then has an input state of TRUE.

     See Section 6.4, "Getting and Setting Logic States" on page 6-5 for an explanation of input and output states.
  3. Your configuration rule causes the selection of Option2, which then has an output state of LTRUE. The end user observes the effect of this change to Option2. This effect might include the calculation of a price, or the inclusion of a certain item in the order.
  4. The configuration is saved. Only the input state of TRUE for Option1 is saved.
  5. The configuration rule "Option1 Requires Option2" is deleted or disabled.
  6. The configuration is restored. Only the state of UTRUE for Option1 is restored. Because your configuration rule is no longer affecting Option2, its input state remains UNKNOWN. The end user observes, with confusion, that the previous selection of Option2 no longer occurs. The effect of this situation might be that a previously observed price or item no longer appears in the order.

- For more information on restoring configurations, see the *Oracle Configurator Implementation Guide*.

5.7 Restarting Configurations

Use `Configuration.restartConfiguration()` to restart the current configuration. You restart a configuration when you want to remove the effects of a configuration session without removing the components that you are configuring from the session. When you restart a configuration, the CIO:

- Rolls back logic transactions
- Removes requests
- Reverses the assertions that had set logic states and values
- Removes component instances added during the session, and restores component instances deleted during the session

You must be using the CIO with a custom user interface to use `restartConfiguration()`; this method cannot be used with a user interface generated by Oracle Configurator Developer.

5.8 Automatic Behavior for Configurations

You can define behavior that is executed whenever a configuration is processed in certain ways, by defining Configurator Extensions bound to certain events. Table 5–3 on page 5-9 describes some of these events, and the circumstances under which you should use them. For a list of types of events, see Table 3–1, " Types of Configuration

Events" on page 3-1. For more details, and a full list of the available events, see the chapter on Configurator Extensions in the *Oracle Configurator Developer User's Guide*.

Table 5–3 Events for Processing Configurations

Event	Triggered ...	Comments
postConfigNew	When a newly-created configuration is activated	See Section 5.2, "Creating Configurations" on page 5-2 for background on creating configurations.
preConfigSave	Before a configuration is saved	You can save a configuration using the Model Debugger in Oracle Configurator Developer.
postConfigSave	After a configuration is saved	Clicking the **Finish** button in the runtime Oracle Configurator terminates the configuration session and saves the configuration, if it is valid.
postConfigRestore	After a configuration is restored	You can restore a saved configuration using the Model Debugger in Oracle Configurator Developer.
preConfigSummary	Immediately before the Summary screen is displayed	Clicking the **Summary** button in the runtime Oracle Configurator displays the Summary screen.

See the *Oracle Configurator Developer User's Guide* for details on how to create Configurator Extensions that are bound to events.

In the runtime Oracle Configurator, the Configurator Extension runs when one of the events listed in Table 5–3 on page 5-9 is executed (such as after a configuration is saved).

5.9 Access to Configuration Parameters

If you are using Oracle Configurator in a Web deployment, you can use a Configurator Extension to obtain a list of the initialization parameters that are passed from the host application to your configuration Model.

To access initialization parameters, create a Configurator Extension that calls `Configuration.getUserParameters()`, which returns a `NameValuePairSet` object. This object contains all the parameter names and values stored by the runtime Oracle Configurator when it processes the initialization message sent by the host application to the Oracle Applications Framework.

As a security measure, the initialization parameter `pwd`, which contains a password, is not returned by `getUserParameters()`.

To add your own user-defined configuration parameters to those contained in the initialization message, making them a part of the configuration, use `ConfigParameters.addUserParam()`, which takes the name of the parameter (a string) and the value (an object). To obtain the value of one of these configuration parameters, call `ConfigParameters.getUserParam()`.

See the *Oracle Configurator Implementation Guide* for more information about the initialization message.

5.10 Sharing a Configuration Session

During a configuration session, your application may require the ability to launch a custom user interface in a child window of the runtime Oracle Configurator window. This child UI might interact with the user and perform updates to the state of the configuration model. When these interactions are finished, the child UI returns control to the parent window containing the runtime Oracle Configurator UI.

If your application opens such a child window, that window needs shared access to the configuration model, through the `Configuration` object.

You can get the Configuration object from the HTTP session by using the key `configurationObject`. You can obtain a URL for returning to the parent window by requesting the session object `czReturnToConfiguratorUrl`. The example in Section B.3, "Sharing a Configuration Session in a Child Window" on page B-6 illustrates the use of these objects. You can obtain these objects by using one of the following methods from the Java servlet/JSP API:

- `javax.servlet.http.HttpSession.getValue ("czReturnToConfiguratorUrl")`
- `javax.servlet.jsp.PageContext.getAttribute("czReturnToConfiguratorUrl", PageContext.SESSION_SCOPE)`

During the period of user interaction with the child UI window, you should prevent any use of the parent window, since that might interfere with the changes to the state of the application or configuration model being made in the child window.

> **Caution:** The custom UI in the child window must be running in the same HTTP session as the parent window containing the runtime Oracle Configurator. You must also ensure thread safety, as noted under Section 1.4.2, "Restrictions for Configurator Extensions" on page 1-4.

You can create the kind of child window that you need in the HTML-based version of Oracle Configurator Developer, by creating a UI element (such as a Custom Button) that supports the **Open URL** action in a generated Configurator UI, using the specifications provided in Table 5–4. For background, see the *Oracle Configurator Developer User's Guide*.

Table 5–4 UI Specifications for Invoking Child Window

Option	Choice
Caption Source	**Text Expression**, indicating to the end user the action that the UI element performs
Action Type	**Open URL**
Target URL Source	**Text Expression**, pointing to your custom child UI (such as a JSP), which must be located in the OA_HTML directory. The specific expression for Example B–4 is: `/OA_HTML/TestChildWin.jsp`
Target Window	**Child Window** Select the option to **Lock Main Window while Displaying Child**

These specifications are used for Example B–4, "Sharing a Configuration Session in a Child Window (TestChildWin.jsp)" on page B-7.

6 Working with Model Entities

This chapter describes how to interact with entities in model structure.

The sections of this chapter are:

The root component, and every other node in the underlying runtime Model tree, implements the `IRuntimeNode` interface. This interface exposes several attributes of the configuration model, such as the type of the node (based on a set of node type constants), its name, the node ID, a runtime ID that is unique to this node across all nodes created by this particular CIO, the parent node (which is null for the root component), a (possibly empty) collection of children, and information about whether this part of the runtime tree has been satisfied. See Section 6.8, "Introspection through IRuntimeNode" on page 6-10.

6.1 Opportunities for Modifying the Configuration

During a configuration session, there are certain optimal points for modifying the configuration.

Note: This use of the CIO is intended for Configurator Extensions.

Use `IRuntimeNode.getConfiguration()` to get the configuration to which a node belongs. The code fragment in Example 6–1 shows how to get the `Configuration` object associated with the current node in the runtime Oracle Configurator.

Example 6–1 Getting the Configuration from a Runtime Node

```
IRuntimeNode node = getRuntimeNode();
Configuration config = node.getConfiguration();
```

You can modify a configuration by using a Configurator Extension bound to one of the configuration events described in Table 5–3, " Events for Processing Configurations" on page 5-9, Table 3–1, " Types of Configuration Events" on page 3-1, and the chapter on Configurator Extensions in the *Oracle Configurator Developer User's Guide*.

For instance, if you want to modify the configuration immediately after a new configuration session has been initialized, then bind your Configurator Extension to the `postConfigNew` event.

Modifying the configuration through a Configurator Extension is sometimes referred to as side-effecting it.

Caution: Be careful of recursion when using the events `postValueChange` and `onConfigValidate`, which are triggered when a change to the configuration is detected by Oracle Configurator. It is possible to enter an infinite loop in which changes that you make in your Configurator Extension trigger an event that makes the Configurator Extension run again. See Section 2.4.4, "Avoiding Circularity and Recursion" on page 2-9 for more details.

Be careful when binding a Configurator Extension to the `postCXInit` event, since that event always occurs when a configuration session begins.

6.2 Accessing Components

The CIO represents instantiable components with two structures that are used together: `Component` and `ComponentSet`. An individual instance of a component is represented by the interface `Component`. A set of these instances of a given component is represented by an instance of the class `ComponentSet`. Both structures inherit from the interface `IRuntimeNode`.

In Oracle Configurator Developer, there is no element that corresponds to a `ComponentSet`, but you can control the Instantiability settings for a node. The Instantiability settings for initial minimum and initial maximum determine the minimum and maximum number of instances that can be added at runtime. Components that have a minimum number of instances of 1 and a maximum number of instances of 1 are called required components. Components that have a minimum number of instances of 0 and a maximum number of instances of 1 or more are called instantiable components. See the *Oracle Configurator Developer User's Guide* for details about required and instantiable components.

6.2.1 Adding and Deleting Instantiable Components

Note: This use of the CIO is intended for both custom applications and Configurator Extensions.

It is most likely that you would add or delete instantiable components in a Configurator Extension.

Use `ComponentSet.add()` to add an instantiable component. The result is a new object that uses the `Component` interface.

The `add()` method can throw a `LogicalException` if adding the component causes a logical contradiction.

Use `ComponentSet.delete()` to delete an instantiable component.

In the user interface for the runtime Oracle Configurator, a configurable component is normally represented by a single screen. The screen that represents the parent node of this component contains a button that adds instances of the component, producing a new component screen and a new `Component` object. This is equivalent to adding instances through `ComponentSet.add()`. The screen representing the configurable component itself contains a button that deletes that instance of the component. This is equivalent to deleting the instance through `ComponentSet.delete()`.

In a user interface generated by Oracle Configurator Developer, when the end user adds an instance of an instantiable component that is a BOM Model (which is represented by a `BomInstance` object), that instance is automatically selected. If the addition causes any contradictions, the appropriate messages are displayed. However, if you use a Configurator Extension to add an instance of a BOM Model, that instance is *not* automatically selected. If you want your Configurator Extension to select the instance, you must do it explicitly, as shown in Example 6–2 on page 6-3. Instantiable components that do not represent BOM Models cannot be selected.

Example 6–2 Adding and Selecting an Instance of a BOM Model

```
...
ComponentSet compSet = (ComponentSet)comp1.getChildByName("My Model");
Component comp = compSet.add();
if (comp instanceof BomModel) {
   (BomInstance(comp)).select();
}
...
```

See Section 5.6, "Restoring Configurations" on page 5-6 for information on the effects of changes to Instantiability settings in Oracle Configurator Developer when restoring configurations in which instances have been added, deleted, or modified.

Note: There are some performance problems that can arise when adding and deleting several instantiable components. See the *Oracle Configurator Modeling Guide* for details.

6.2.2 Renaming Instances of Components

During a configuration session, when the end user of the runtime Oracle Configurator creates a new instance of a configurable component, the user interface displays a distinctive name for the instance.

For more information on controlling the display of instance names in the runtime Oracle Configurator, see the *Oracle Configurator Implementation Guide*.

You can access the default name that is displayed in the runtime user interface, by using the methods `setInstanceName()`, `getInstanceName()`, and `hasInstanceName()` in the interface `Component`.

You can use `setInstanceName()` to set the name of an instance of an instantiable component. The component to be renamed cannot be a required component. The name that you set persists when you restore the configuration that contains the instance.

You can use `hasInstanceName()`, and `getInstanceName()` to test whether the name of an instance has been set, and to return the name.

For a fragmentary example of how to change the name of an instance, see Example 6–3.

Example 6–3 Renaming an Instance of a Component

```
...
String inputText = "My Instance Name";

ComponentSet compSet = (ComponentSet)comp1.getChildByName("My Model");
Component comp = compSet.add();
comp.setInstanceName(inputText);
...
```

For a full example of how to change the name of an instance, see Example 2–1 on page 2-6.

6.3 Accessing Features

There are several specialized types of Features. Each Feature type implements the `IRuntimeNode` interface, enabling you to use its general methods for working with runtime nodes (see Section 6.8, "Introspection through IRuntimeNode" on page 6-10). Each type also implements its own interface with appropriately specialized methods.

Table 6–1, " Feature Types, Value Types, and Interfaces" on page 6-4 lists the types of Features that you can work with in the CIO, the types of their values, and the CIO interface for working with them.

Table 6–1 Feature Types, Value Types, and Interfaces

CIO Interface	Feature Type	Value Type
IState	Boolean	boolean state (true/false/unknown)
IDecimal	Decimal	floating point numeric
IInteger	Integer	integer numeric The value can be positive, negative, or zero.
IText	Text	string
ICount, IState	Count	boolean, with an associated integer-valued numeric count
`IOptionFeature`	Option	logic value for the Option Feature itself a set of Options for the Option Feature's children

Some of these types require special comment:

- Option Features are represented by `OptionFeature` objects. An `OptionFeature` has a logic value. If the Option Feature is satisfied, the value is `TRUE`. The values of an `OptionFeature` object are Options.

  You can use the methods `getMinSelected()` and `getMaxSelected()`, of `IOptionFeature`, to determine the minimum and maximum number of a Feature's child Options that can be selected. If you do, first use `hasMinSelected()` or `hasMaxSelected()` to determine whether there is a minimum or maximum number of Options. You can use `areOptionsCounted()` to determine whether the Feature has Counted Options.

  Keep in mind that an end user of the runtime Oracle Configurator can select an Option of an Option Feature, but not the Option Feature itself. However, in a Configurator Extension, it is possible to use `select()` to select an `OptionFeature` object itself. You should avoid selecting `OptionFeature` objects. If you do so and save the configuration, then this selection is not applied if you later restore the configuration.

See Section 6.7, "Access to Options" on page 6-9 for information about methods for working directly with Options.

- `CountFeature` objects have an associated integer-valued numeric count, and are a special case of `IntegerFeature` that has a count greater than or equal to zero. `CountFeature` objects behave like counted options in an `OptionFeature`.

> **Note:** In Oracle Configurator Developer, if you set the minimum count of an Integer Feature greater than or equal to zero, then at runtime the CIO treats this Feature as a `CountFeature` object. If you set the minimum count to less than zero, then the CIO treats this Feature as an `IntegerFeature` object. When working with runtime nodes, you must consider this distinction to ensure that you are working with the expected set of objects. For example, if you use `IRuntimeNode.getChildrenByType()` to collect Integer Feature objects, then you must make two calls, one with an `IRuntimeNode.COUNT_FEATURE argument`, and another with an `IRuntimeNode.INTEGER_FEATURE` argument.

6.4 Getting and Setting Logic States

To interact with objects that have a logic state, you use methods of the `IState` interface. This interface contains:

- A set of constants that represent input states, used to specify a new state for an object, listed in Table 6–2:

Table 6–2 Input Logic States

State	Description
`FALSE`	The input state used to set an object to false.
`TRUE`	The input state used to set an object to true.
`TOGGLE`	The input state used to turn an object state to true if it is false or unknown, and to make it unknown or false if it is true.

- A set of constants that represent output states, returned when querying an object for its state listed in Table 6–3:

Table 6–3 Output Logic States

State	Description
`LFALSE`	The Logic False output state, indicating that the state is false as a consequence of a rule.
`LTRUE`	The Logic True output state, indicating that the state is true as a consequence of a rule.
`UFALSE`	The User False output state, indicating that a user has set this object to false.
`UTRUE`	The User True output state, indicating that a user has set this object to true.
`UNKNOWN`	The Unknown output state, indicating that there is no current state.

- A set of methods for getting and setting the object's state listed in Table 6–4:

Table 6–4 Methods for Getting and Setting State

Method	Description
`getState()`	Gets the current logic state of this object.
`setState()`	Change the current logic state of this object.
`unset()`	Retracts any user selection made on this node.
`isFalse()`	Tells whether this feature is in false state.
`isTrue()`	Tells whether this feature is in true state.
`isUser()`	Tells whether this feature is in a user- specified state.
`isLogic()`	Tells whether this feature is in a logically specified state.
`isUnknown()`	Tells whether this feature is in unknown or known state.

Observe the following practices when you use methods of the `IState` interface:

- The code fragment in Example 6–4 uses `getState()` with `UTRUE` to test whether the state of an Option node is **user true**, meaning that the Option has been selected by the end user.

Example 6–4 Getting the State of a Node

```
// Get the necessary components from the configuration.
baseComponent = (Component)comp_node.getChildByName("Component-1");
of = (OptionFeature)baseComponent.getChildByName("Feature-1");
op = (Option)of.getChildByName("Option-1");
intFeat = (IntegerFeature)baseComponent.getChildByName("IF-1");
// Check if the option is set to UTRUE.
// If so, set the Integer value to 5.
if( op.getState() == IState.UTRUE )
  intFeat.setIntValue(5);
```

- When using `getState()`, Always check for deleted or discontinued nodes. See Section 2.4.10, "Checking for Deleted or Discontinued Nodes" on page 2-12.
- Using `isUnknown()`, which returns `TRUE` if the Feature is in an unknown state, is important when a node is cast to an integer or decimal class such as `IntegerNode` or `ReadOnlyDecimalNode`. When the numeric value of the node is zero, a zero value can mean either `UNKNOWN` (if no value has been set by the user) or `KNOWN` (if the value has been set to zero by the user).
- The code fragment in Example 6–5, which uses `setState()` with `TOGGLE`, toggles the state of the selected item in the Model tree.

Example 6–5 Setting the State of a Node

```
private void toggleSelectedItem() {
    IState node = (IState)getSelectedNode();
    node.setState(IState.TOGGLE);
    }
```

You should not use the `TOGGLE` state unless you are working with a user interface. If you do not need to render the result in the interface (for instance, if you are using batch validation) then it is much more efficient to set the state directly:

```
node.setState(IState.TRUE);
...
node.setState(IState.FALSE);
```

If you do need to use TOGGLE, do not turn off defaulting, because the CIO must turn defaulting on in order to determine the correct state to toggle to. This operation impairs performance.

- If you try to set the state of a `RuntimeNode` to `UNKNOWN` and this causes a contradiction, then the CIO throws a nonoverridable `LogicalException`. For example, assume the following Model structure:

  ```
  M
  |_A (Boolean, UNKNOWN)
  |_B (Boolean, UNKNOWN)
  ```

  And a logic rule:

  ```
  A Requires B
  ```

  When you select A, it makes B `LTRUE`. If you try setting B to `UNKNOWN`, you get a nonoverridable logical contradiction:

  ```
  A.setState(IState.UTRUE);
  ...
  try {
    B.setState(IState.UNKNOWN);
  } catch (LogicalException le) {
  //le is not overridable
  ```

- When you are not interested in the difference between `UTRUE` and `LTRUE`, the proper way to determine whether the state of a node is true is to call `IState.isTrue()`.

  By contrast, if you test the state of the node this way:

  ```
  (state == IState.TRUE)
  ```

  then the test only returns `TRUE` if the logic state is `UTRUE`, but not if it is `LTRUE`.

6.5 Getting and Setting Numeric Values

You can use the following methods to get and set the values of objects that have numeric values. Consult the CIO reference (see Chapter A, "Reference Documentation for the CIO") for the hierarchy of the classes you wish to use.

For decimal values, use:

```
IDecimal.setDecimalValue()
IReadOnlyDecimal.getDecimalValue()
```

For integer values, use:

```
IInteger.setIntValue()
IInteger.getIntValue()
```

The code fragment in Example 6–6 uses `setIntValue()` to change the value of an Integer Feature. Note that you can use the generalized `IRuntimeNode` interface for flexibility in getting a child node, and then cast the node object to a particular interface to perform the desired operation on it.

Example 6–6 Setting a Numeric Value

```
// select a node by name
```

```
IRuntimeNode limit = baseComp.getChildByName("Current Limit");

// use an interface cast to set the node’s value by the desired type
((IInteger)limit).setIntValue(5);
```

To determine whether a numeric value has violated its Minimum or Maximum range, you may need to iterate through the collection of validation failures returned by `Configuration.getValidationFailures()` after setting a value, for instance with `IInteger.setIntValue()`. See Section 8.1, "Validating Configurations" on page 8-1 for more background.

There is a subtlety that you should take note of. `IDecimal.setDecimalValue()` does not throw a `LogicalException` when setting the value of a decimal feature that exceeds the feature's minimum/maximum limits. The collection of validation failures returned by `Configuration.getValidationFailures()` does not include any failures that result from setting a numeric value until the logic transaction has been closed. Thus, there is no way to roll back a transaction once it is committed. You can only undo the setting of the value. Here is a suggested method for dealing with this situation:

1. Open a transaction.
2. Get the minimum or maximum for the Feature, with `getMin()` or `getMax()`.
3. Set the new value appropriately.
4. Close the transaction.
5. Get the collection of validation failures for the configuration, to find out about the status of *other* nodes.
6. If the last transaction caused a minimum/maximum violation, then call `Configuration.undo()`, which retracts the last action in the transaction.

This situation illustrates why it is a good practice to perform the setting of a single value inside a logic transaction. You can always undo the transaction if the result is unsatisfactory. Remember: inside a transaction, you can roll back an action; outside a transaction, you undo an action.

6.5.1 Working with Decimal Quantities

Quantities for imported BOM Standard Items can be either integers or decimals.

Table 6–5, " Methods for Integer and Decimal Nodes" on page 6-8 lists certain methods of CIO classes and interfaces that are relevant to decimal quantities. The table indicates the corresponding methods to be used for BOM nodes having Integer (indivisible) values or Decimal (divisible) values. Using the wrong type of method raises an `IncompatibleValueException`. For details on these methods, see Chapter A, "Reference Documentation for the CIO".

In the classes `IRuntimeNode` and `RuntimeNode`, the methods `hasIntegerValue()` and `hasDecimalValue()` should be used to find out if a run-time node belongs to a Decimal or an Integer BOM.

`StateCountNode.getDecimalCount()` is a general method for getting the count and works for both Integer and Decimal BOMs.

Table 6–5 Methods for Integer and Decimal Nodes

Class/Interface	Integer Method	Decimal Method
`BomNode`	`getDefaultQuantity()`	`getDecimalDefaultQuantity()`

Table 6–5 (Cont.) Methods for Integer and Decimal Nodes

Class/Interface	Integer Method	Decimal Method
`BomNode`	`getMaxQuantity()`	`getDecimalMaxQuantity()`
`BomNode`	`getMinQuantity()`	`getDecimalMinQuantity()`
`IBomItem`	`getMaxQuantity()`	`getDecimalMaxQuantity()`
`IBomItem`	`getMinQuantity()`	`getDecimalMinQuantity()`
`ICount`	`getCount()`	`getDecimalCount()`
`ICount`	`setCount()`	`setDecimalCount()`
`StateCountNode`	`getCount()`	`getDecimalCount()`
`StateCountNode`	`setCount()`	`setDecimalCount()`

When using one of the methods listed in Table 6–5, always check for deleted or discontinued nodes. See Section 2.4.10, "Checking for Deleted or Discontinued Nodes" on page 2-12.

6.6 Accessing Properties

You can determine which Properties belong to a runtime node, then use methods of the class `Property` to obtain information about the Properties.

Use `IRuntimeNode.getProperties()` to get a collection of the properties associated with a node.

Use `IRuntimeNode.getPropertyByName()` to get a particular property of a node, based on its name.

When you have the Property, use methods of the class `Property,` such as `getStringValue(),` to obtain specific information.

6.7 Access to Options

An Option is a child of an Option Feature which supports a boolean state (true, false, or unknown) and a count. Options implement the `IRuntimeNode` interface.

`OptionFeature` objects have special methods for selecting options and querying for selected options. See Section 6.3, "Accessing Features" on page 6-4 for information about methods for working directly with Features.

In a custom application, you can use `IOPtionFeature.select()` to select a specified Option. If a maximum number of selections has been defined for an `OptionFeature`, and that maximum has been reached, then this method implements mutual exclusion behavior by first deselecting the most recently selected Option that does not cause a contradiction when deselected, then selecting the newly specified option. The minimum number of selections defined for the `OptionFeature` does not affect this behavior.

You can find out which Option has been deselected, after a selection is committed, by using `IOPtionFeature.getSelectedOptions()` and examining the list of selected nodes.

The `getSelectedOption()` method throws the `SelectionNotMutexedException` if this feature does not support mutually exclusive (mutexed) selections.

You can use the interface `IOption` to select, deselect, and determine the selection state of Options. Table 6–6 on page 6-10 lists these methods.

Table 6–6 Methods of the Interface IOption

Method	Action
`deselect()`	Deselect this Option.
`isSelected()`	Returns true if this Option is selected, and false otherwise. When using `isSelected()`, always check for deleted or discontinued nodes. See Section 2.4.10, "Checking for Deleted or Discontinued Nodes" on page 2-12.
`select()`	Select this Option.

The code fragment in Example 6–7 displays a "check" icon if an Option of a runtime node is selected:

Example 6–7 Testing Whether an Option Is Selected

```
IRuntimeNode rtNode = (IRuntimeNode)value;
if (value instanceof IOption) {
  IOption optionNode = (IOption)value;
  if !(optionNode.isDeleted() || optionNode.isDiscontinued()) {
    if (optionNode.isSelected()) {
      setIcon(checkIcon);
    }
  }
}
```

In this example, assume that `checkIcon` points to an icon file, and that `setIcon()` is a custom method that displays it.

6.8 Introspection through IRuntimeNode

You can get information about a node in a Model at runtime by using methods of the interface `IRuntimeNode`. This helps you to write "generic" Configurator Extensions, which can interact with a Model tree dynamically, without having prior knowledge of its structure. Table 6–7 on page 6-10 lists some of the more important of these methods.

Table 6–7 on page 6-10 lists some of the methods defined in the interface `IRuntimeNode` that you are most likely to use in working with the CIO. For more detail about these and the other CIO interfaces, see Chapter A, "Reference Documentation for the CIO".

Table 6–7 Important Methods of the Interface IRuntimeNode

Method	Action
`getCaption()`	Get the Caption of this node to be displayed in messages.
`getChildByID()`	Gets a particular child identified by its ID. `ComponentSet.getChildByID()` could have duplicate children with same ID, so it returns only the first child. Instead, call `getChildByInstanceNumber()` or change the instance name.
`getChildByName()`	Gets a particular child identified by its name.

Table 6–7 (Cont.) Important Methods of the Interface IRuntimeNode

Method	Action
`getChildren()`	Gets the children of this runtime configuration node.
`getDescription()`	Returns the design-time description of the runtime node.
`getName()`	Gets the name of the node.
`getParent()`	Gets the parent of the node.
`getProperties()`	Returns a collection of the properties associated with this node. The collection contains items of the type Property.
`getRuntimeID()`	Gets the runtime ID of the node.
`getType()`	Gets the type of this node.
`isEffective()`	Returns true if this particular node is effective given the effectivity criteria of the model.
`isUiVisible()`	Returns true if this node is visible in the UI, meaning that a control for it appears in the UI for selection by end users. Returns false if this node is not visible in the UI (according to the value of CZ_PS_NODES.UI_OMIT). Note that a node reported as not visible by this method is nevertheless included in a UI created with the "show all nodes" option.
`isUnsatisfied()`	Returns true if this particular node, or any one of its children, has not been completely configured.

The code fragment in Example 6–8 creates a Configuration object `config`, sets `homeTheater` to the root component of the configuration, and sets `userType` to the child node with the user-visible name "User Type".

Example 6–8 Getting a Child Node by Name

```
Configuration config = m_cio.startConfiguration(params, context);
IRuntimeNode homeTheater = (IRuntimeNode) config.getRootComponent();

IRuntimeNode userType = homeTheater.getChildByName("User Type");
```

The code fragment in Example 6–9 uses a test for the value of the `TEXT_FEATURE` field of an `IRuntimeNode` object named `comp` to gather a list of all the children of that node that are `TextFeature` objects. It is assumed that `traverseTree()` is a custom method.

Example 6–9 Collecting All Child Nodes by Type

```
//get all the text features
List textFeatList = IRuntimeNode comp.getChildrenByType(IRuntimeNode.TEXT_
FEATURE);
traverseTree(comp.getChildComponentNodes(),
             IRuntimeNode.TEXT_FEATURE,
             textFeatList);
Iterator iter = textFeatList.iterator();
```

7 Using Logic Transactions

In order to help you maintain consistency in interactions with the Oracle Configurator logic engine, you must use *configuration-level logic transactions*. A logic transaction comprises all the logical assertions that constitute a user interaction. At the end of a transaction, you can obtain a list of all validation failures, by calling `Configuration.getValidationFailures()`. See Section 8.1, "Validating Configurations" on page 8-1.

The Configuration object, `oracle.apps.cz.cio.Configuration`, provides a set of methods for starting, ending, and rolling back configuration-level logic transactions. Note that logic transactions are not database transactions.

Inside a transaction, the normal course of action is to set the logical states and numeric values of runtime nodes (as described in Section 6.4, "Getting and Setting Logic States" on page 6-5 and Section 6.5, "Getting and Setting Numeric Values" on page 6-7).

- Use `Configuration.beginConfigTransaction()` to create a new transaction, returning a `ConfigTransaction` object. After performing the desired series of operations (for instance, setting states and values), you must end, commit, or roll back the transaction by passing the `ConfigTransaction` object to one of the mutually exclusive methods that finish the transaction:

  ```
  endConfigTransaction
  commitConfigTransaction
  rollbackConfigTransaction
  ```

- `Configuration.commitConfigTransaction()` commits the given transaction or series of nested transactions, propagates the effect of user selections throughout the configuration, and triggers validation checking (see Section 8.1, "Validating Configurations" on page 8-1).
- `Configuration.endConfigTransaction()` ends the transaction that was started with `beginConfigTransaction()`, without committing it (thus skipping validation checking).
- `Configuration.rollbackConfigTransaction()` rolls back the unfinished transaction, undoing the operations performed inside it.

You can nest intermediate transactions with `beginConfigTransaction()` and `endConfigTransaction`, delaying validation checking until you call `commitConfigTransaction()`. You should not perform any actions (such as setting states or counts, or selecting Options) before opening a nested transaction. If there are actions performed in an uncommitted parent transaction, these may produce erroneous results for `Configuration.getUnsatisfiedItems()`. You must end or commit inner transactions before ending or committing the outer ones that contain them. When rolling back unfinished transactions, with `rollbackConfigTransaction()`, you can roll back outer transactions, which automatically rolls back the inner transactions.

Transactions should also be used when you employ nonoverridable requests. See Section 9.4, "Nonoverridable Requests" on page 9-2.

There are situations in which you must take care to commit a transaction at the appropriate time. The fragmentary code in Example 7–1 on page 7-2 illustrates the need for wrapping a common operation inside a transaction to insure that the operation's effects are reflected in other parts of the program. Example B.2.1, "Setting Nonoverridable Requests" on page B-3 also illustrates the use of transactions.

Example 7–1   Using a Logic Transaction with a Deletion

```
...
Component comp;
ComponentSet compSet;
ConfigTransaction tr;
Configuration config;
IOption opt;

// ----------------------------------------------------------
// This sequence produces unintended results:
...

...
// Select a child of compSet.
...
opt.select()
...
// User wants to see the list of all selected nodes:
collec = config.getSelectedItems();
// The returned collection includes children of the deleted component,
// because no transaction was commited.

// ----------------------------------------------------------
// This sequence produces the intended results:
...
// Add a component:
comp = compSet.add();
...
// User selects a child of compSet (interactively).
...
// Delete the component, inside a transaction:
tr = config.beginConfigTransaction();
compSet.delete(component);
config.commitConfigTransaction(tr);
...
// User wants to see the list of all selected nodes:
collec = config.getSelectedItems();
// The returned collection does NOT include children of the deleted component,
// because the deletion transaction was commited.
```

8 Contradictions, Exceptions, and Validation

This chapter describes how to handle:

- Validation, which is the act of checking that a configuration is valid and complete
- Logical exceptions, which are the representation in the CIO of contradictions, (violations of your configuration rules that are presented to the end user)
- Programming exceptions, which are raised by your code

The sections of this chapter are:

- Validating Configurations
- Handling Logical Contradictions
- Handling Exceptions

8.1 Validating Configurations

Validating a configuration means checking whether it is valid (that is, the selections in it do not violate any configuration rules) and whether it is complete (that is, all components in it are satisfied).

The CIO validates a configuration after a transaction is committed or rolled back. See Chapter 7, "Using Logic Transactions" for a description of what happens in a transaction.

Validation checking and reporting occur when a logical transaction is ended by using `Configuration.commitConfigTransaction()` or `Configuration.rollbackConfigTransaction()`.

After a committal or rollback, the CIO traverses the nodes of the Model, checking for validation failures, selected items and unsatisfied items. These are kept in a set of collections maintained on the `Configuration` object.

After the transaction is committed, you can call the methods of `oracle.apps.cz.cio.Configuration` listed in Table 8–1:

Table 8–1 Methods for Validating Configurations

Method	Description
`getValidationFailures()`	Returns a collection of `ValidationFailure` objects. Call this after committing or rolling back a transaction, in order to inspect the list of validation failures.

Table 8–1 (Cont.) Methods for Validating Configurations

Method	Description
`getSelectedItems()`	Returns a collection of selected items as `StatusInfo` objects indicating the set of selected (true) items in the Configuration.
`isUnsatisfied()`	Returns `TRUE` if the configuration is incomplete.
`getUnsatisfiedItems()`	Returns a collection of unsatisfied items as `StatusInfo` objects indicating the set of unsatisfied items in the Configuration.
`getInformationalMessages()`	Gets a collection of `StatusInfo` objects describing all the informational messages in the configuration. These messages are created explicitly by external callers or Configurator Extensions or by the CIO in response to an exception thrown by a Configurator Extension.
`getUnsatisfiedRuleMessages()`	Gets a list of messages for unsatisfied relations in the configuration.

To determine whether a configuration has validation failures, call `getValidationFailures()` and check whether the collection it returns is empty.

Validation failures are instances of the class `StatusInfo`. A `StatusInfo` object has a reference to the runtime node, which you obtain with its `getNode()` method. Use `StatusInfo.getStatus()` to return the current status of the node.

The status of a node has a life cycle. The stages in the life cycle are represented by the constants described in Table 8–2, " Life Cycle of StatusInfo Objects" on page 8-2. As nodes become selected, or unsatisfied, or have validation failures, they have a status reflected by `StatusInfo.STATUS_NEW`. If they continue to be selected since the last transaction their status is `StatusInfo.STATUS_EXISTING`. If they become deselected, their status becomes `StatusInfo.STATUS_DELETED` until the next transaction at which time they are removed from the collection.

Table 8–2 Life Cycle of StatusInfo Objects

StatusInfo Constant	Status Description
`STATUS_NEW`	The node has newly attained this status since the last check.
`STATUS_EXISTING`	The node already had this status during the last check, and it still does.
`STATUS_DELETED`	The node has newly lost this status since the last check.
`STATUS_REMOVED`	The node had the deleted status during the last check, so it is removed.

If you are writing a Configurator Extension that validates a configuration, the method that you bind to the `onConfigValidate` event should return a list of `CustomValidationFailure` objects in the event of a validation failure. This allows you to return more than one failure. Your validation method can include several tests. You can track which tests failed, and determine why the tests failed. If the validation fails, then information about the failure is gathered by the CIO in a List of `CustomValidationFailure` objects. The information in these objects is presented to the user in a message, and does not persist after the presentation.

In general, if a Configurator Extension needs to return a violation message about a particular runtime node, you have to create a `CustomValidationFailure` object and pass it the runtime node, the message, and boolean parameter indicating whether to persist the failure. The code fragment in Example 8–1 illustrates this point.

Example 8–1 Returning a List of Validation Failures

```
public List validateMin() {
...
IRuntimeNode node;
ArrayList failures = new ArrayList();
...
//check to see if the value in the config is not at least the min value
if( !
(val >= min) )
    failures.add( new CustomValidationFailure("Value less than minimum", node,
true) );
    if(failures.isEmpty())
        return null;
    else
        return failures;
...
}
```

If the violation persists after the next user action, the Configurator Extension should not need to create a new `CustomValidationFailure`, but should instead return a `StatusInfo` object with the same status (`STATUS_EXISTING`). This value prevents the CIO from returning the previously seen violation message as a new violation message (`STATUS_NEW`), which might be annoying for the user. However, if the user explicitly makes the same invalid selection again, then the message is presented again.

You should use the form of the constructor for `CustomValidationFailure` that sets the boolean parameter `willPersist` to `true`. This keeps the failure from disappearing once the message is displayed to the user, which can lead to a situation in which invalid configurations are displayed as valid.

Invalidating a configuration with a Configurator Extension (by creating `CustomValidationFailure` objects) can sometimes lead to performance issues, since the validation tests are run each time the enclosing transaction is committed. One way to avoid this is to place the validation tests outside the transaction, or bind the validating Configurator Extension to an event other than `onConfigValidate`.

Another way to alleviates this performance issue is to persist the validation failure, as shown in Example 8–1 on page 8-3, because if the boolean parameter `willPersist` is `true`, then the validation tests are not run each time the enclosing transaction is committed. However, if you are programmatically marking the configuration as invalid in this way, you must remove the persisted failure when configuration becomes valid again. To remove the persisted failure, you can remove the `CustomValidationFailure` in the following way:

```
CustomValidationFailure cvf = findPreviousCustomValidationFailure(node);
cvf.removeCustomValidationFailure();
```

Note that in this example `findPreviousCustomValidationFailure()` is a your custom method for finding the failure for a given node. One way of implementing this is by maintaining a `Map` object in your code in which the keys are nodes and the values are `CustomValidationFailure` objects. You should clear the map in when your terminates so that Java garbage collection will release the memory.

8.2 Handling Logical Contradictions

When you make a logic request to modify the state of a configuration, for instance by using `IState.setState()`, the result may be a failure of the request because of a logical contradiction. Such a failure creates and throws a *logical exception*, accessed through either of these objects:

- `LogicalException`, which cannot be overridden
- `LogicalOverridableException`, which can be overridden

See Section 8.2.2, "Overriding Contradictions" on page 8-4 for details on using `LogicalOverridableException` to override the contradiction.

- Use `LogicalException.isOverridable()` to determine whether the exception is an instance of `LogicalOverridableException`, which can be overridden with its `override()` method.
- Use `LogicalException.getCause()` to get the runtime node that caused the failure.
- Use `LogicalException.getReasons()` to get a list of Reason objects for the failure. See Section 8.2.1, "Generating Error Messages from Contradictions" on page 8-4.
- Use `LogicalException.getMessage()` to provide a message containing both the cause and the reasons.

  Use `LogicalException.getMessageHeader()` to provide a message containing only the causes. You can pass a `caption` argument to this method, which is the string to use as the node name. Use this caption as an alternative to the node caption provided by the CIO for the message.

8.2.1 Generating Error Messages from Contradictions

You can use the `Reason` object to wrap the information returned by a contradiction, in order to include error message information from the table FND_NEW_ MESSAGES.

- Use `Reason.translate()` to get the message associated with this reason.
- Use `Reason.getNode()` to get the node associated with this reason.
- Use `Reason.getType()` to get the type of reason held in this object.
- Use `Reason.toString()` to convert this object to a string.

8.2.2 Overriding Contradictions

Your runtime Oracle Configurator or Configurator Extension can provide a message to your user, and ask whether the contradiction should be overridden.

If a logical contraction can be overridden, then a `LogicalOverridableException` is signalled, instead of a `LogicalException`. `LogicalOverridableException` is a subclass of `LogicalException` that adds an `override()` method. Use `LogicalOverridableException.override()` to override the contradiction.

Both types of exceptions (`LogicalException` and `LogicalOverridableException`) may be thrown from any of the "set" methods (like `setState()`) or from `Configuration.commitConfigTransaction()`.

If you want to override the overridable exception you have to call its `override()` method, which can also throw a `LogicalException`. This means that even when you try to override the exception you still trigger a contradiction and cannot continue. If the override succeeds, then you still need to call `commitConfigTransaction()` to close the transaction. If you don't want to override or if you get a `LogicalException` you need to call `rollbackConfigTransaction()` to purge it. The Example 8–2 on page 8-5 is a fragment of pseudocode that illustrates this point.

Note that the operations represented with `[ASK "text"]` and `[SHOW "text"]` are not part of the CIO but suggest where your own custom application should try to handle the situation.

Example 8–2 Handling and Overriding Logical Exceptions

```
try {
      // begin a transaction
    ConfigTransaction tr = config.beginConfigTransaction();

      // call the "set" method
    opt1.setState(IState.TRUE);
      // commit the transaction
    config.commitConfigTransaction(tr);
  }
  catch(LogicalOverridableException loe) {
    proceed = [ASK "Do you want to override?"];
    if (! proceed) {
      config.rollbackConfigTransaction(tr);
    }
    else {
      try {
          // override the contradiction and ...
        loe.override();  // returns a list of failed requests
          // ... finish the transaction
        config.commitConfigTransaction(tr);
      }
      catch (LogicalException le) {
          // we cannot do anything
        [SHOW "Cannot be overriden"]
        config.rollbackConfigTransaction(tr);
      }
    }
  }
  catch (LogicalException le) {
      // we cannot do anything
    [SHOW "Cannot be overriden"]
    config.rollbackConfigTransaction(tr);
  }
```

In Example 8–2, the statement `loe.override();` returns a list of failed requests. See Section 9.5, "Failed Requests" on page 9-4.

8.3 Handling Exceptions

This section describes how to handle exceptions raised by the CIO.

Caution: Improper handling of exceptions is the source of many problems that are difficult to diagnose. See Section 2.4.3, "Handling Exceptions Properly" on page 2-8 for more information.

8.3.1 Handling Types of Exceptions

When a Configurator Extension is invoked, the runtime Oracle Configurator wraps a transaction around this invocation. This transaction enables the work of the Configurator Extension to be either committed or rolled back, as necessary. See Chapter 7, "Using Logic Transactions" for background.

If your Configurator Extension needs to handle an exception, you can choose the type of exception to throw. The runtime Oracle Configurator handles the exception as follows:

- If your throwable exception is one that extends `java.lang.Error` or `java.lang.RuntimeException`, it is fatal. The runtime Oracle Configurator does the following:
  - Drops any open transactions
  - Kills the configuration session, but allows the end user to start a new session

  Caution: Your code should not ignore or swallow such exceptions; doing so can lead to problems that are difficult to debug.

  In the case of a fatal exception, your code should throw an unchecked exception, as shown in Section 8.3.2, "Raising Fatal Exceptions" on page 8-6.

- If your throwable exception does not extend `Error` or `RuntimeException`, then it is nonfatal. The runtime Oracle Configurator does the following:
  - Rolls back the transaction, which undoes the work done by the Configurator Extension
  - Uses the message for exception to create an `InformationalMessage` object (described in Section 8.3.3, "Presenting Messages for Exceptions" on page 8-7)
  - Allows the user's configuration session to continue
  - Allows other Configurator Extensions bound to the same triggering event to run

8.3.2 Raising Fatal Exceptions

If your Configurator Extension code encounters an unexpected problem that you cannot handle, you should convert the exception that you caught into an unchecked exception. For this purpose, use the exception `oracle.apps.cz.utilities.CheckedToUncheckedException`, which extends `RuntimeException`.

`CheckedToUncheckedException` allows you to change a checked exception into an unchecked one, as shown in Example 8–3, "Raising a Fatal Exception" on page 8-6. The new unchecked exception contains the messages and stack traces from both the original checked exception and the new unchecked exception. However, extra properties of specialized checked exceptions that you throw as a `CheckedToUncheckedException` are not retained in the new unchecked exception.

Example 8–3 Raising a Fatal Exception

```
public void setBoolean (BooleanFeature bf)
{
  try {
    bf.setState(IState.TRUE);
  }
  catch (LogicalException le) {
    throw new CheckedToUncheckedException(le);
  }
}
```

8.3.3 Presenting Messages for Exceptions

If you want to present messages to the end user without rolling back the transaction, your Configurator Extension should add a new `InformationalMessage`, by calling `Configuration.addInformationalMessage()` on the `Configuration` object for the session, as shown in Example 8–4 on page 8-7. In Example 8–4, the `desc` parameter could be bound to anything in the Model that returns the string that supplies the text for the message (such as the value of a TextFeature node, a literal, or a certain System Parameters). The `node` parameter could be bound to the node on which the exception occurs.

Example 8–4 Presenting an Informational Message

```
public void nodeMessage(String desc, IRuntimeNode node) throws LogicalException
   {
     try
     {
       Configuration config = node.getConfiguration();
       ConfigTransaction tr = config.beginConfigTransaction();
       InformationalMessage iMsg = new InformationalMessage("The node is: " +
desc, node);
       config.addInformationalMessage(iMsg);
       config.commitConfigTransaction(tr);
     }catch (LogicalException le){
       throw le;
     }
   }
```

You can call `Configuration.getInformationalMessages()` to get a collection of `StatusInfo` objects that describe all the `InformationalMessages` in the configuration. For information on the `StatusInfo` object, see Section 8.1, "Validating Configurations" on page 8-1.

Note: You can only use `addInformationalMessage()` to present a message from a Configurator Extension to the end user. After the message is dismissed by the user it disappears, without passing any information back to the runtime Oracle Configurator. You cannot use an `InformationalMessage` object to get a response from the end user in reaction to a message.

8.3.4 Compatibility of Certain Deprecated Exceptions

The exceptions `FuncCompMessageException` and `FuncCompErrorException` were introduced in a previous version of the CIO, but are now deprecated, and are retained only for backward compatibility with existing code. Even though these two exceptions extend `RuntimeException`, they are not fatal in the CIO. They are treated as non-fatal exceptions, as described in Section 8.3.1, "Handling Types of Exceptions" on page 8-5.

Caution: The classes `FuncCompMessageException` and `FuncCompErrorException` are now deprecated, but are retained for backward compatibility with existing code.

A `FuncCompErrorException` rolls back the open transaction, and allows the end user's configuration session to continue. In general, you should not throw a

`FuncCompErrorException` unless you have very good reasons to believe that the exception is benign and that the user should also be notified of it. You should document these reasons in your code.

A `FuncCompMessageException` allowed you to present a dialog box displaying a specified message, and the name of the Functional Companion that raised the exception. When the end user dismissed the dialog box, the runtime Oracle Configurator committed the open CIO transaction, and allowed the end user to proceed with the configuration session. It was possible that the Model could be left in an uncertain state. In the current version of the CIO, the transaction is rolled back, instead of committed.

9 Using Requests

This chapter describes requests, which are programmatic attempts to modify a configuration.

The sections of this chapter are:

- Getting Information about Requests
- User Requests
- Nonoverridable Requests
- Failed Requests

9.1 About Requests

A request is an attempt to modify a configuration by setting the logical state or numeric value of a node in the configuration Model (such as an Option or BOM Item). Table 9–1 on page 9-1 lists some methods of this type:

Table 9–1 Methods Typically Used to Make Requests

Method	Described In ...
`IState.setState()`	Getting and Setting Logic States on page 6-5
`ICount.setCount()`	Getting and Setting Numeric Values on page 6-7
`IOPtion.select()`	Access to Options on page 6-9

- Requests that set a state or value, such as those listed in Table 9–1, are called *user requests*. See Section 9.3, "User Requests" on page 9-2.
- You can code a set of user requests that are applied to a configuration at any time. These are called *nonoverridable requests*. These requests can be applied only programmatically, and have a higher priority than user requests. See Section 9.4, "Nonoverridable Requests" on page 9-2.
- When user requests fail, due to an override of a contradiction, the CIO generates a list of these *failed requests*. See Section 9.5, "Failed Requests" on page 9-4.
- You can get information about a request by interrogating an instance of the Request object. See Section 9.2, "Getting Information about Requests" on page 9-1.

9.2 Getting Information about Requests

The class `oracle.apps.cz.cio.Request` exposes logic requests. A `Request` object can be used to represent several kinds of requests.

The `Request` object provides a set of methods for determining the value of the request, and the runtime node on which the request has been made:

- `getNumericValue()`
- `getValue()`
- `getRuntimeNode()`

The `Request` object also provides a set of methods for determining the type of the request. These methods are listed in Table 9–2. (In the value column, the test for the value of the request is case-sensitive. The value strings must be lowercase.)

Table 9–2 Type Methods of the Class Request

This returns TRUE if ...	... the request made was for ...	The value of the request is ...
`isNumericRequest()`	changing the numeric value of a runtime node	a Number
`isStateRequest()`	changing the state of a runtime node	`true`, `false`, `toggle`, `unknown`
`isTrueStateRequest()`	changing the state of a runtime node to True	`true`
`isFalseStateRequest()`	changing the state of a runtime node to False	`false`
`isToggleStateRequest()`	toggling the state of a runtime node	`toggle`
`isUnknownStateRequest()`	unsetting the state of a runtime node	`unknown`

9.3 User Requests

You can obtain a list of the Request objects that represent all current user requests in the system, by using the method `Configuration.getUserRequests()` in your Configurator Extension.

```
...
IRuntimeNode node = getRuntimeNode();
Configuration config = node.getConfiguration();
List requests = config.getUserRequests();
Iterator it = requests.iterator();
while (it.hasNext()) {
  Request req = (Request)it.next();
  IRuntimeNode node = req.getRuntimeNode();
  String value = req.getValue();
}
...
```

9.4 Nonoverridable Requests

You can specify a set of logic requests to be applied to a configuration at any time that have a higher priority than user requests. Such requests are called nonoverridable requests.

You apply nonoverridable requests automatically on the creation of a configuration, following the practice illustrated in Example 9–1 on page 9-3 and in the following steps:

1. Begin a configuration transaction, using `Configuration.beginConfigTransaction()`.

   ```
   ConfigTransaction tr = config.beginConfigTransaction();
   ```

   See Section 7, "Using Logic Transactions" on page 7-1 for details about transactions.

2. Specify that the transaction contains nonoverridable requests, using `ConfigTransaction.useNonOverridableRequests()`.

   ```
   tr.useNonOverridableRequests();
   ```

3. Specify the desired user requests using the appropriate methods.

   ```
   BooleanFeature feat = (BooleanFeature)node.getChildByName("Feature_1234");
   feat.setState(IState.TRUE);
   ```

   See Section 9.3, "User Requests" on page 9-2 for details about setting logic requests.

4. When you have set all the desired nonoverridable requests, commit the logic transaction.

   ```
   config.commitConfigTransaction(tr);
   ```

These steps are combined in Example 9–1. For a fuller example of using nonoverridable requests, see Example B.2.1, "Setting Nonoverridable Requests" on page B-3.

Example 9–1 Using Nonoverridable Requests

```
...
      ConfigTransaction tr = config.beginConfigTransaction();
      tr.useNonOverridableRequests();
      BooleanFeature feat = (BooleanFeature)node.getChildByName("Feature_1234");
      feat.setState(IState.TRUE);
      config.commitConfigTransaction(tr);
...
```

9.4.1 Usage Notes on Nonoverridable Requests

- You can think of a transaction that includes `ConfigTransaction.useNonOverridableRequests()` (as illustrated in Step 2 on page 9-3) as putting the CIO in "nonoverridable request mode". You can nest any number of subtransactions within this transaction; the requests in these subtransactions all inherit this mode of being nonoverridable requests. You can perform overrides and rollbacks as you would with ordinary user requests. You must commit or roll back the nonoverridable-request transaction, as in step 4, to indicate the conclusion of the nonoverridable requests. You can then specify other user requests in your Configurator Extension.
- When you save a configuration that includes nonoverridable requests, the nonoverridable requests are saved as part of the configuration. When you restore such a configuration, with `CIO.restoreConfiguration()`, the nonoverridable requests are reapplied to the configuration.
- You can get a list of the list of nonoverridable requests present in a configuration by using `Configuration.getNonOverridableRequests()`.
- In a nonoverridable transaction, you can retract a nonoverridable request by calling `unset()` on the appropriate runtime node.

9.4.2 Limitations on Nonoverridable Requests

- After you apply nonoverridable requests to a configuration, you cannot override any of the nonoverridable requests with user requests. But you can override nonoverridable requests with other nonoverridable requests. An attempt to override a nonoverridable request with a user request throws a `NonOverridableRequestException`, which cannot be overridden.
- You cannot use nonoverridable requests to add or delete components, or create a connection.

9.5 Failed Requests

When you use `LogicalOverridableException.override()` to override a logical contradiction (see Section 8.2.2, "Overriding Contradictions" on page 8-4), the `override()` method returns a List of Request objects. These Request objects represent all the previously asserted user requests that failed due to the override that you are performing.

See Section B.2.2, "Getting a List of Failed Requests" on page B-5 for an example.

10 Configuration Session Change Tracking

This chapter describes the CIO's Configuration Delta API for tracking changes that have been made to a configuration.

The sections of this chapter are:

10.1 Overview

This section is divided as follows:

- For a general overview of the Configuration Delta API, see Section 10.2, "How It Works" on page 10-2.
- For examples of how the Configuration Delta API is used, see:
  - Section 10.3, "Starting a Session" on page 10-5
  - Section 10.4, "Tracking Session Changes" on page 10-7
  - Section 10.5, "Updating a Region" on page 10-8
  - Section 10.6, "Handling Screen Changes" on page 10-9
- For information on a specialized customization topic, see Section 10.7, "Creating a Custom DeltaValidator" on page 10-10.
- For detailed reference documentation that describes the classes of the Configuration Delta API, see Appendix A, "Reference Documentation for the CIO".

You can use the CIO's Configuration Delta API to query a Configuration object about changes (*deltas*) that have been made to the configuration during the current configuration session.

Note: Although the functionality described in this section uses the terms **delta** and **tracking**, this functionality is distinct from the tracking of deltas described in the *Oracle Telecommunications Service Ordering Process Guide*. In that document, the term **delta** refers to a change made to a configuration relative to an instance of that configuration residing in an installation repository.

The Configuration Delta API provides a unified interface that enables you to query for deltas only on the specific nodes in which you register interest. Contrast this to the set of methods listed Example 10–1, which provide change information only for the entire set of the nodes in a configuration.

Example 10–1 Change-Detection Methods for the Configuration Object

```
Configuration.getSelectedItems()
Configuration.getUnsatisfiedItems()
Configuration.getUnsatisfiedItems()
Configuration.getUnsatisfiedRuleMessages()
Configuration.getValidationFailures()
```

10.2 How It Works

Note: This use of the CIO is intended for both custom applications and Configurator Extensions.

Both custom applications and Configurator Extensions can be clients of the Configuration Delta API.

The Configuration Delta API consists of the classes and interfaces in the CIO listed in Table 10–1. The Instances column indicates how many instances of the class exist at runtime, during a configuration session.

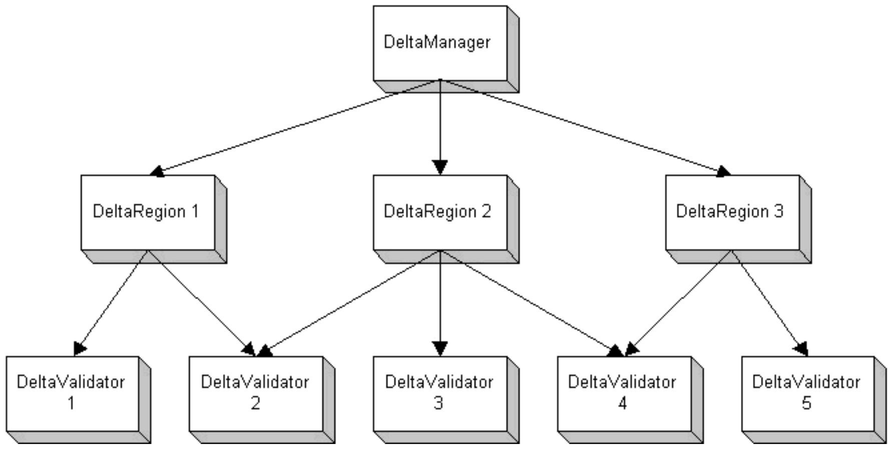
Table 10–1 Classes and Interfaces for the Configuration Delta API

Class or Interface	Role	Instances
`DeltaManager`	Manages all changes made by end user actions during a configuration session. See Section 10.2.2 on page 10-3.	One per client.
`DeltaRegion`	Maintains list of watched runtime nodes and changes to be tracked on those nodes. See Section 10.2.3 on page 10-4.	One per each region of interest in the user interface. Can register multiple DeltaValidators, one for each type of change to be tracked.
`DeltaValidator`	Manages all defined types of changes. Base class for all DeltaValidators. See Section 10.2.4 on page 10-4.	One per each type of change to be tracked. Can be registered with multiple DeltaRegions.
`IValidatorChange`	Represents any change type. See Section 10.2.5 on page 10-5.	Not instantiated. Implemented by all DeltaValidators.

10.2.1 Relationship of the Classes

The diagram in Figure 10–1 on page 10-3 shows the relationship of the classes in the Configuration Delta API, using a typical example of their use.

Figure 10–1 Example Class Relationships in the Configuration Delta API



In Figure 10–1, the DeltaManager is managing a UI containing three DeltaRegions (labeled 1, 2, and 3).

- Each DeltaRegion maintains a list of runtime nodes that are watched for changes (the watched-nodes list).
- Each DeltaRegion is registered with the DeltaManager and contains a list of DeltaValidators, which determine the types of changes that are watched in the region.
- In the example:
  - DeltaRegion 1 has registered DeltaValidators 1 and 2
  - DeltaRegion 2 has registered DeltaValidators 2, 3, and 4
  - DeltaRegion 3 has registered DeltaValidators 4 and 5
- Each DeltaValidator can be registered with multiple DeltaRegions. Each DeltaValidator watches for a particular change type in a combined list of all the runtime nodes in all the DeltaRegions that it is registered with.
- In the example:
  - Since DeltaValidator 1 is registered only with DeltaRegion 1, its watched-nodes list is the same as the watched-nodes list in DeltaRegion 1.
  - DeltaValidator 2 is registered with two DeltaRegions (1 and 2). Hence, its watched-nodes list is the union of the watched-nodes lists from both DeltaRegions 1 and 2.

10.2.2 Role of the DeltaManager

The `DeltaManager` object is instantiated once, at the beginning of a configuration session, and is cached on the Configuration object for the session. The `DeltaManager` manages all the changes made by end user actions during that session.

The `DeltaManager` is identified by an ID that is passed to the method that creates it, `Configuration.createDeltaManager()`.

You can register multiple DeltaRegions with the DeltaManager, to manage the regions of your client's user interface.

10.2.3 Role of DeltaRegions

A `DeltaRegion` object represents a distinct portion of your client's user interface. For example, your UI might have a navigation region, an update region, and a summary region; your client would create a `DeltaRegion` object for each of them.

Each `DeltaRegion` maintains a list of watched runtime nodes in that region. You determine which nodes are to be watched for changes by registering a `DeltaRegion` object with the `DeltaManager`, using the method `DeltaManager.registerRegion()`, which takes as arguments the list of nodes to watch, the list of DeltaValidators to watch them with, and an ID. See Example 10–5 on page 10-7 for an example of registering a region.

10.2.4 Role of DeltaValidators

A `DeltaValidator` object manages defined types of changes. A `DeltaValidator` can be thought of as a reusable software component that reports on a particular type of change.

Each particular change type is handled through a specialized subclass of the class `DeltaValidator`. The CIO provides a set of default change types that correspond to the types of changes that can be made through the CIO. Each subclass defines a change object (in the form of an inner class) that implements methods that provide information about the specified type of change.

Table 10–2 lists a sampling of the default change types, and the specialized DeltaValidators that represent them. For details on the methods of these change object classes, and the complete set of `DeltaValidator` subclasses, see the CIO reference documentation described in Appendix A.

You can write custom DeltaValidators for change types that are not already provided by the CIO. For details, see Section 10.7, "Creating a Custom DeltaValidator" on page 10-10.

Table 10–2 Default Change Types and Their Change Objects

Change Type	Class for Change Object
ATP (Availability to Promise)	`AtpDeltaValidator.AtpChange`
Availability (for selection)	`AvailabilityDeltaValidator.AvailabilityChange`
Connection	`ConnectionDeltaValidator.ConnectionChange`
Count (of runtime nodes)	`CountDeltaValidator.CountChange`
Deletion	`DeletionDeltaValidator.DeletionChange`
Price	`PriceDeltaValidator.PriceChange`
Selection status (change between selected and deselected)	`SelectionDeltaValidator.SelectionChange`
Logic state (of a node)	`StateDeltaValidator.StateChange`
Satisfaction (change between satisfied and unsatisfied)	`UnsatisfactionDeltaValidator.UnsatisfactionChange`

Table 10–2 (Cont.) Default Change Types and Their Change Objects

Change Type	Class for Change Object
ValidationFailure messages	`ValidationDeltaValidator.ValidationChange`

Each change object (inner class) implements the method `getType()` of the interface `IValidatorChange`. Each inner class must also implement any methods that are appropriate to their particular change type. See Example 10–7 on page 10-9 for examples of how you would use both the `IValidatorChange` methods and the type-specific methods.

10.2.5 Role of the IValidatorChange Interface

The `IValidatorChange` interface:

- Represents any kind of DeltaValidator change. It is implemented by all DeltaValidators to represent their specific change object.
- Is the interface for the class ValidatorChange, which is the base class for all the change-object inner classes described in Section 10.2.4, "Role of DeltaValidators" on page 10-4.
- Provides the method `getType()`, which returns one of the DeltaValidator type constants defined in the DeltaValidator object. See Example 10–7 on page 10-9 for an example of how you would use this method.

10.3 Starting a Session

Your client should perform the following steps once, at the beginning of a configuration session.

1. Create a Configuration object.

   See Example 10–2 in Section 10.3.1, "Creating a Configuration Object" on page 10-5.

2. Create a DeltaManager object and associate it with the Configuration object.

   See Example 10–3 in Section 10.3.2, "Associating a DeltaManager" on page 10-6.

3. Specify the DeltaValidators corresponding to the change types you want to track during the configuration session.

   See Example 10–4 in Section 10.3.3, "Specifying DeltaValidators" on page 10-6.

4. Get a list of the nodes in the region whose changes you are interested in querying and register that region.

   See Example 10–5 or Example 10–6 in Section 10.3.4, "Registering DeltaRegions" on page 10-6.

10.3.1 Creating a Configuration Object

If you are working with a custom application, create a Configuration object, as described in see Section 5.2, "Creating Configurations" on page 5-2 for required background information. See especially Example 5–1 on page 5-3.

Example 10–2 Creating a Configuration Object

```
. . .
```

```
// Create a new Configuration and DeltaManager
ConfigParameters params = new ConfigParameters(modelId);
Configuration config = cio.startConfiguration(params, context);
...
```

Note: The fragmentary code examples in this section are meant to be read together, as parts of a larger example. Identifiers are shared between examples; where the same identifier occurs in multiple examples, it refers to the same object. These fragmentary examples are assembled together in Example B–5 on page B-8.

10.3.2 Associating a DeltaManager

Associate a DeltaManager object with the Configuration object for the current configuration session.

Example 10–3 Associating a DeltaManager with a Configuration

```
...
DeltaManager deltaMgr = config.createDeltaManager("MyDeltaMgr");
...
```

10.3.3 Specifying DeltaValidators

Create DeltaValidator objects for the change types that you want to track during the configuration session. Then add them to a list that can be used to register the DeltaValidators for a DeltaRegion (shown in Example 10–5 on page 10-7).

Example 10–4 Specifying DeltaValidators

```
...
// Create a Navigation (Tree) region. This is interested in watching
// all runtime nodes for instance name, instantiation, and unsatisfaction
// changes.
List dvList = new ArrayList();

dvList.add(deltaMgr.getDeltaValidator(DeltaValidator.INSTANCE_NAME_DV));
dvList.add(deltaMgr.getDeltaValidator(DeltaValidator.INSTANTIATION_DV));
dvList.add(deltaMgr.getDeltaValidator(DeltaValidator.UNSATISFACTION_DV));
...
```

10.3.4 Registering DeltaRegions

Register a DeltaRegion with the DeltaManager, passing it the list of nodes to watch and the list of DeltaValidators to watch them with (`dvList`, defined in Example 10–4 on page 10-6).

You can also register an individual DeltaValidator, using `DeltaManager.registerDeltaValidator()`.

Example 10–5 shows the registration of a region using all of the runtime nodes in the Configuration (`config.getRuntimeNodes()`). If you want to use some subset of the runtime nodes (such as only the nodes visible in the user interface), then you must implement a custom method to do so. This alternative is shown in Example 10–6, using the hypothetical custom method `getRuntimeNodesInSelectedComponent()`.

Example 10–5 Registering a DeltaRegion: All Nodes

```
...
List watchedNodes = config.getRuntimeNodes();

DeltaRegion treeRegion = deltaMgr.registerRegion(watchedNodes, dvList,
"MyTreeRegion");
...
```

Example 10–6 Registering a DeltaRegion: Subset of Nodes

```
...
// Create a component region. This region displays a Component screen and is
// interested in watching all nodes in that component for availability, count,
// price, state and unsatisfaction changes
dvList.clear();
dvList.add(deltaMgr.getDeltaValidator(DeltaValidator.AVAILABILITY_DV));
dvList.add(deltaMgr.getDeltaValidator(DeltaValidator.COUNT_DV));
dvList.add(deltaMgr.getDeltaValidator(DeltaValidator.PRICE_DV));
dvList.add(deltaMgr.getDeltaValidator(DeltaValidator.STATE_DV));
dvList.add(deltaMgr.getDeltaValidator(DeltaValidator.UNSATISFACTION_DV));

watchedNodes = getRuntimeNodesInSelectedComponent(); // a custom method, not
defined here

DeltaRegion compRegion = deltaMgr.registerRegion(watchedNodes, dvList,
"MyCompRegion");
...
```

10.4 Tracking Session Changes

Your client should perform the following steps each time it needs to track a session change to the current configuration. Most of the code examples shown in this section are shown in a more complete context in Example B–5, "Tracking Session Changes (DeltaExample.java)" on page B-8.

1. Begin a configuration transaction. See Chapter 7, "Using Logic Transactions" for background.

   ```
   ConfigTransaction tran = config.beginConfigTransaction();
   ```

2. Perform the change, by making an assertion. For background details, see Section 6.4, "Getting and Setting Logic States" on page 6-5 and Section 6.5, "Getting and Setting Numeric Values" on page 6-7. The following example fragment shows how to select the Option node named `Option1`.

   ```
   // Make an assertion to change the current configuration
   try {
   Option option1 =
   (Option)config.getRootComponent().getChildByName("Feature").getChildByName("Opt
   ion1");
   option1.select();
    } catch (LogicalException loe) { }
   ```

3. Close the configuration transaction.

```
config.commitConfigTransaction(tran);
```

4. Query the configuration for the changes of interest. Update the list of changes that you can use to update a region that you registered. The following example updates the change map for the region registered in Example 10–5 on page 10-7.

```
// Get the deltas due to this assertion and update the tree and component
regions
Map treeChanges = deltaMgr.getUpdateMapForRegion("MyTreeRegion");
```

5. Update the region that you registered (as in Example 10–5 on page 10-7), using a custom method. The custom method `updateTreeRegion()` is described in Section 10.5, "Updating a Region" on page 10-8.

```
// Now update the tree region cache and UI with treeChanges
updateTreeRegion(treeChanges);
```

10.5 Updating a Region

When you need to update a region with the a list of the changes that your client has been tracking with the DeltaManager, you can invoke a custom method such as `updateTreeRegion()`, whose definition is shown in Example 10–7 on page 10-9. This method operates as follows:

1. Take as an argument the `changes` object that is a Map of the changed nodes in the registered region (`MyTreeRegion`). See Section 10.4, "Tracking Session Changes" on page 10-7 for a description of when this updating takes place.

   The map of changed nodes consists of a set of pairs, in which the key is a `RuntimeNode` object, and the value is a collection of `IValidatorChange` objects.

2. Iterate over the nodes in the Map of changed nodes. Use a custom method, such as `getUiNode()` in the example, to get access to the UI node corresponding to the changed node object.

3. Iterate over the List of changes for the node, using each change to set the value of the `IValidatorChange` object `change`.

4. For each change, call `IValidatorChange.getType()`, which returns the type of the change in a form that corresponds to one of the change types defined in the class DeltaValidator, such as `INSTANCE_NAME_DV`.

5. Using a `switch` control structure, switch on the change type. For each change type, cast the `change` object to the actual implementing class of the change, such as `InstanceNameChange`.

```
InstanceNameDeltaValidator.InstanceNameChange nameChange =
InstanceNameDeltaValidator.InstanceNameChange)change;
```

6. Using the particular change object for the change, use a custom method to update the UI node corresponding to the changed node object.

```
String newName = nameChange.getInstanceName();
uiNode.setName(newName); // custom method on uiNode
```

Example 10–7 Custom Method to Update a Region

```
public static void updateTreeRegion(Map changes) {
  for (Iterator iter = changes.keySet().iterator(); iter.hasNext();) {
    RuntimeNode changedNode = (RuntimeNode)iter.next();
    uiNode = getUiNode(changedNode); // custom method
    Collection nodeChanges = (Collection)changes.get(changedNode);
    for (Iterator iter2 = nodeChanges.iterator(); iter2.hasNext();) {
      IValidatorChange change = (IValidatorChange)iter2.next();
      switch (change.getType()) {
        case DeltaValidator.INSTANCE_NAME_DV:
          InstanceNameDeltaValidator.InstanceNameChange nameChange =
InstanceNameDeltaValidator.InstanceNameChange)change;
          String newName = nameChange.getInstanceName();
          uiNode.setName(newName); // custom method on uiNode
          break;
        case DeltaValidator.INSTANTIATION_DV:
          InstantiationDeltaValidator.InstantiationChange iChange =
(InstantiationDeltaValidator.InstantiationChange) change;
          Collection added = iChange.getNewlyAddedInstances();
          Collection deleted = iChange.getNewlyDeletedInstances();
          uiNode.updateInstances(added, deleted); // custom method on uiNode
          break;
        case DeltaValidator.UNSATISFACTION_DV:
          UnsatisfactionDeltaValidator.UnsatisfactionChange uChange =
(UnsatisfactionDeltaValidator.UnsatisfactionChange) change;
          boolean unsatisfied = uChange.isUnsatisfied();
          uiNode.setUnsatisfied(unsatisfied); // custom method on uiNode
          break;
      }
    }
  }
```

10.6 Handling Screen Changes

When a screen change (such as a screen flip to another UI page) occurs in your client's user interface, you should update the list of watched nodes in each DeltaRegion, so that you can get a list of the changes made to the nodes whenever you need such a list.

The manner in which you update the watched nodes depends on how extensive are the changes to the region you are watching.

- If the general layout of the region is unchanged, and only the set of nodes in the region may have changed, you can simply clear the list of watched nodes, get the list of currently interesting nodes, then add that list to the region's list of nodes to watch. This approach is shown in Example 10–8.

Example 10–8 Updating Watched Nodes: Screen Format Unchanged

```
...
rgn1.clearWatchedNodes();
List visibleNodes = getCurrentVisibleNodes(); // custom method
rgn1.addWatchedNodes(visibleNodes);
...
```

  You must define the custom method used to get the visible nodes, `getCurrentVisibleNodes()`.

- If the general layout of the region has **changed** significantly, then you should unregister the region, rebuild the list of DeltaValidators, and register the region, specifying all the nodes and the list of DeltaValidators. This approach is shown in Example 10–9.

Example 10–9 Updating Watched Nodes: Screen Format Changed Significantly

```
...
mgr.unRegisterRegion(rgn1.getId());
List dvList = new ArrayList();
dvList.add(dm.getDeltaValidator(DeltaValidator.PRICE_DV));
dvList.add(dm.getDeltaValidator(DeltaValidator.AVAILABILITY_DV));
rgn1 = mgr.registerRegion(config.getRuntimeNodes(), dvList, null);
...
```

10.7 Creating a Custom DeltaValidator

It is possible, but unlikely, that you may need to write custom DeltaValidators for change types that are not already defined in the CIO. See Section 10.2.4, "Role of DeltaValidators" on page 10-4 for an explanation of DeltaValidators and a description of the default DeltaValidators provided with the CIO.

In order to create a custom DeltaValidator, you must do the following:

- Define a subclass that extends `DeltaValidator`. This class is your custom DeltaValidator. For example:

  ```
  public class MyCustomDeltaValidator extends DeltaValidator {
    // constructor
    protected MyCustomDeltaValidator() {
      setType(MY_CUSTOM_DV);
    }
  ```

- Define a change object that represents the type of change that your custom DeltaValidator is designed to track. This change object class must implement the interface `IValidatorChange`. See Section 10.2.5, "Role of the IValidatorChange Interface" on page 10-5.

  ```
  public class MyCustomChange extends ValidatorChange {
  // Implement your change object here
  }
  ```

  In the DeltaValidators defined in the CIO, the change object is defined as an inner class, but this design decision is not mandatory.

- In the custom DeltaValidator, define a constant that designates your custom type of DeltaValidator and the change type that it tracks. The value of the constant must be greater than `DeltaValidator.CUSTOM_DV` (which is currently defined as 1000, though you should not directly reference that value). Example:

  ```
  public static final int MY_CUSTOM_DV = DeltaValidator.CUSTOM_DV + 1;
  ```

- In the custom DeltaValidator, implement the method `isChanged()`, which is defined as `abstract` in `DeltaValidator`:

  ```
  protected abstract boolean isChanged(IRuntimeNode node, DeltaRegion region)
  ```

  Your implementation must determine if there are any changes to be reported for the runtime node by this DeltaValidator, for the given region.

- In the custom DeltaValidator, implement the method `getChange()`, defined as `abstract` in DeltaValidator:

  ```
  protected abstract IValidatorChange getChange(IRuntimeNode node, DeltaRegion
  region)
  ```

  Your implementation must get the change object for this node. For example:

  ```
  protected IValidatorChange getChange(IRuntimeNode node, DeltaRegion region) {
    MyCustomChange change = new MyCustomChange();
    return change;
  }
  ```

- In the change-object class, implement the method `getType()` from the interface `IValidatorChange`. Your implementation must return the change type, which corresponds to the custom DeltaValidator type that you defined. For example:

  ```
  public int getType() {
    return MyCustomDeltaValidator.MY_CUSTOM_DV;
  }
  ```

- Include your custom DeltaValidator in list of DeltaValidators passed to `DeltaManager.registerRegion()`. See Section 10.3.4, "Registering DeltaRegions" on page 10-6. You can also register a custom DeltaValidator independently, using `DeltaManager.registerDeltaValidator()`, which adds a DeltaValidator to the list of existing ones. This will enable different regions to use the same instance of your custom DeltaValidator.

10.8 Unified Code Example

The code in Example B–5, "Tracking Session Changes (DeltaExample.java)" on page B-8 assembles together the fragmentary examples shown elsewhere in this chapter.

11 Logging Through the CIO

This chapter provides basic information about logging the operations you perform with the CIO, especially those inside Configurator Extensions.

The sections of this chapter are:

11.1 Overview of Logging

Oracle Configurator and the Oracle Configuration Interface Object use the Oracle Applications Logging Framework to provide a convenient and uniform interface for logging their activity.

For references to Oracle documentation about the Oracle Applications Logging Framework, see "Troubleshooting" on page xvii.

Logging through the CIO requires these essential actions:

- Section 11.2, "Enabling Logging Scope" on page 11-1
- Section 11.3, "Creating Entries in the Log" on page 11-3
- Section 11.4, "Recommended Practices for Logging" on page 11-4

These actions are illustrated together by Section 11.5, "Example of Logging" on page 11-5.

Note: Logging through the CIO is primarily intended for use within Configurator Extensions, but you can also use it in custom applications that use the CIO directly.

11.2 Enabling Logging Scope

In order to enable the creation of log entries through the CIO you must set the following parameters for the Oracle Applications Logging Framework:

- AFLOG_ENABLED, to turn on logging.

- AFLOG_MODULE, to specify the Java packages or classes that you wish to log, using the parameters described in Table 11–1 on page 11-2.
- AFLOG_LEVEL, to specify the level of entries that you wish to log, using the parameters described in Table 11–2 on page 11-2.
- AFLOG_FILENAME, to specify the file where middle-tier log messages are written.
- AFLOG_ECHO, to optionally echo all filtered logging messages to STDERR.

These parameters can be set as middle-tier properties or as database profile options. The parameter names listed here are for middle-tier properties. See the *Oracle Applications Supportability Guide* for information on how to set these parameters as database profile options.

Table 11–1 lists the strings that you can include in the AFLOG_MODULE parameter to identify the Java packages or classes that you wish to log. The AFLOG_MODULE parameter is a comma-delimited filter against which the module names of log messages are compared.

Table 11–1 Values for AFLOG_MODULE

Value	Description

`cz%`	Logs with attribution to the log-writing method `Configuration.writeCXLogEntry()`. This setting logs all activity by Oracle Configurator during a configuration session, regardless of which class in your Configurator Extension or custom application caused the entry to be written. Allows you to examine the activity of your classes in the context of Oracle Configurator activity.
	The Oracle Applications Logging Framework ignores `oracle.apps.` at the beginning of a package name, so to specify `oracle.apps.cz.cio`, you only specify `cz.cio`.
	Examples:
	`cz%`
	`cz.cio%`

`packagepath``%`	Logs with attribution to the methods in your own Configurator Extension or custom application classes that caused the entry to be written. This setting logs only activity by your Configurator Extension or custom application during a configuration session and omits the surrounding activity by Oracle Configurator.
	Examples:
	`acme%`
	`acme.rocket%`

Table 11–2 lists the Oracle Applications Logging Framework logging levels in order of increasing severity. You must specify one of the supported levels when enabling logging through the CIO.

Table 11–2 Values for AFLOG_LEVEL

Value	Description
`STATEMENT`	Used for low-level progress reporting.
`PROCEDURE`	Used for API-level progress reporting.
`EVENT`	Used for high-level progress reporting.
`EXCEPTION`	**Not supported.** Indicates a handled internal software failure.

Table 11–2 (Cont.) Values for AFLOG_LEVEL

Value	Description
`ERROR`	**Not supported.** Indicates an external end user error.
`UNEXPECTED`	**Not supported.** Indicates unhandled internal software failure.

Caution: Logging through the CIO does *not* support use of the more severe logging levels provided by the Oracle Applications Logging Framework, namely: EXCEPTION, ERROR, and UNEXPECTED.

See "Troubleshooting" on page xvii for references to more information about AFLOG_MODULE.

11.3 Creating Entries in the Log

Creating entries in the log requires performing these essential actions in your Configurator Extension or custom application code:

- Testing Whether Logging Is Enabled
- Writing Log Entries

In the Oracle Applications Logging Framework, the term *module* refers to a Java class when it is applied to a Java framework, so that term is used for consistency in the descriptions in this section.

11.3.1 Testing Whether Logging Is Enabled

You test whether logging is enabled by calling the method `Configuration.isCXLogEnabled()`. The syntax for this method is as follows:

```
public final boolean isCXLogEnabled(module, logLevel)
```

Table 11–3 describes the parameters for this method. Notice that the parameter `module` can be either an Object or a String. There are separate signatures of `isCXLogEnabled()` for each data type.

Example 11–1, "Logging Through the CIO" on page 11-5 provides an example of how to use this method.

Table 11–3 Parameters for isCXLogEnabled()

Data Type	Parameter	Description
Object or String	`module`	If you pass an Object, this parameter specifies the Java class to which the log entry will attributed. The typical value for this parameter is the Java keyword `this`. If you pass a String, this parameter specifies the fully-qualified name of the Java class, including its package, to which the log entry will attributed. This form is provided for use with static methods, since Java technology does not allow the use of the keyword `this` in static methods. A runtime exception is raised if this parameter is null. This description also applies to the parameter of the same name in Table 11–4, " Parameters for writeCXLogEntry()" on page 11-4.

Table 11–3 (Cont.) Parameters for isCXLogEnabled()

Data Type	Parameter	Description

int	`logLevel`	The level of detail at which logging is enabled. Must be one of the following constants:
		`Configuration.CXLOG_STATEMENT`
		`Configuration.CXLOG_PROCEDURE`
		`Configuration.CXLOG_EVENT`
		
		The specified level must correspond to one of the supported levels specified for AFLOG_LEVEL, as listed in Table 11–2 on page 11-2. For example, if you specify `Configuration.CXLOG_STATEMENT` for this parameter, then AFLOG_LEVEL must specify `STATEMENT`.
		A runtime exception is raised if this parameter specifies an unsupported level.
		This description also applies to the parameter of the same name in Table 11–4, " Parameters for writeCXLogEntry()" on page 11-4.

11.3.2 Writing Log Entries

You write an entry by calling the method `Configuration.writeCXLogEntry()`. The syntax for this method is as follows:

```
public final void writeCXLogEntry(module, methodName, label, message, logLevel)
```

Table 11–4 describes the parameters for this method. Notice that the parameter `module` can be either an Object or a String. There are separate signatures of `writeCXLogEntry()` for each data type.

Example 11–1, "Logging Through the CIO" on page 11-5 provides an example of how to use this method.

Table 11–4 Parameters for writeCXLogEntry()

Data Type	Parameter	Description
Object or String	`module`	See the description of the parameter of the same name in Table 11–3, " Parameters for isCXLogEnabled()" on page 11-3.
String	`methodName`	The name of your Java method that is calling `writeCXLogEntry()`. This name is written to the log. A runtime exception is raised if this parameter is null or consists of white space.
String	`label`	An optional string. Use to provide additional context for the entry in the log.
String	`message`	An optional string. Use to write the log message that describes the situation being logged.
int	`logLevel`	See the description of the parameter of the same name in Table 11–3, " Parameters for isCXLogEnabled()" on page 11-3.

11.4 Recommended Practices for Logging

When logging through the CIO, you should follow these practices:

- When writing a log entry with `writeCXLogEntry()`, always wrap that invocation with a test that uses `isCXLogEnabled()`. This prevents the unnecessary invocation of `writeCXLogEntry()` when logging is not enabled, which can affect performance.

See Section 11.5, "Example of Logging" on page 11-5 for an example of this practice.

- If you are handling an exception, you can add an explicit invocation of `writeCXLogEntry()` in the catch block of your exception handling routine, specifying any of the supported logging levels listed in Table 11–2, " Values for AFLOG_LEVEL". Note that the CIO logs exceptions even if you do not add this explicit invocation, but adding it may ease your debugging work.
- Set the `logLevel` parameter for `writeCXLogEntry()` to the level that provides you with the most useful information. See Table 11–5 for guidance.

Table 11–5 Values for the logLevel Parameter

Value	Description
`CXLOG_STATEMENT`	Use for low-level progress reporting. Most of your log data will be written at this level. Note that using this level can affect performance, since it requires more logging activity.
`CXLOG_PROCEDURE`	Use for API-level progress reporting. Log at this level to report the entrance into or exit from a Java method of particular interest.
`CXLOG_EVENT`	Use for high-level reporting of significant configuration session events, such as the restoring of a configuration or the selection of a particular Model node. This level is not necessarily equivalent to an event that triggers a Configurator Extension, though you can choose to log such events at this level. This level provides the best logging performance.

11.5 Example of Logging

Example 11–1 illustrates how your code can use the logging methods described in Section 11.3, "Creating Entries in the Log" on page 11-3. These methods are highlighted typographically in Example 11–1. The example also highlights these requirements:

- The methodName parameter must match the name of the enclosing method.
- The logLevel parameter must agree with the setting of AFLOG_LEVEL, which is assumed to be STATEMENT, in this example.

Example 11–1 Logging Through the CIO

```
package acme.code;
import oracle.apps.cz.cio.Configuration;
import oracle.apps.cz.cio.IRuntimeNode;

public class MyClass {
    // ... other code here to interact with configuration ...
    public void selectIt(IRuntimeNode rtNode) {
        Configuration cfg = rtNode.getConfiguration();
        // ... other code here to select a node ...
        if (cfg.isCXLogEnabled(this, Configuration.CXLOG_STATEMENT)) {
            cfg.writeCXLogEntry(this,
                                "selectIt",
                                        null,
                                        "Selecting a node.",
                                        Configuration.CXLOG_STATEMENT);
            }
        }
}
```

Example 11–2 and Example 11–3 show the log entries produced by the code fragment in Example 11–1, with differing settings for AFLOG_MODULE, as described in Section 11.2, "Enabling Logging Scope" on page 11-1.

- Example 11–2 on page 11-6 shows the effect of setting AFLOG_MODULE to `cz%`.
- Example 11–3 on page 11-6 shows the effect of setting AFLOG_MODULE to `acme%`.

Example 11–2 Log File Entry When AFLOG_MODULE Includes cz%

```
[Oct 28, 2004 9:53:58 AM PDT] :
1098982438703:Thread[HttpRequestHandler-94,5,main]: -1: -1: ap723jdv:
139.185.20.44: -1:-1: STATEMENT:[cz.cio.Configuration.writeCXLogEntry]:[null_
4e9d2fd_2 1 2] CXLog> [acme.code.MyClass.selectIt] Selecting Option 1
```

In Example 11–2:

- The entry begins with standard Oracle Applications Logging Framework information.

  ```
  [Oct 28, 2004 9:53:58 AM PDT] :
  1098982438703:Thread[HttpRequestHandler-94,5,main]: -1: -1: ap723jdv:
  139.185.20.44: -1:-1: STATEMENT
  ```

- The next part of the entry shows the attributing class and method:

  ```
  :[cz.cio.Configuration.writeCXLogEntry]
  ```

  Notice that the attributing class and method are `Configuration` and `writeCXLogEntry()`, which are the ones that actually wrote the entry.

- The final part of the entry shows the logging message (which begins with the standard logging footprint text for Oracle Configurator):

  ```
  :[null_4e9d2fd_2 1 2] CXLog> [acme.code.MyClass.selectIt] Selecting a node.
  ```

  Notice that the message includes the prefix `CXLog>` and the full path to your method that called `writeCXLogEntry()`.

Example 11–3 Log File Entry When AFLOG_MODULE Includes acme%

```
[Oct 28, 2004 9:53:58 AM PDT] :
1098982438703:Thread[HttpRequestHandler-94,5,main]:-1: -1: ap723jdv:
139.185.20.44: -1:-1: STATEMENT:[acme.code.MyClass.selectIt]:[null_4e9d2fd_2 1 3]
Selecting Option 1
```

In Example 11–3:

- The entry begins with standard Oracle Applications Logging Framework information.

  ```
  [Oct 28, 2004 9:53:58 AM PDT] :
  1098982438703:Thread[HttpRequestHandler-94,5,main]: -1: -1: ap723jdv:
  139.185.20.44: -1:-1: STATEMENT
  ```

- The next part of the entry shows the attributing class and method:

  ```
  :[acme.code.MyClass.selectIt]
  ```

  Notice that the message shows the full path to your method that called `writeCXLogEntry()`.

- The final part of the entry shows the logging message (which begins with the standard logging footprint text for Oracle Configurator):

  ```
  :[null_4e9d2fd_2 1 3] Selecting a node.
  ```

  Notice that the message shows only the text that you passed as an argument to the `message` parameter of `writeCXLogEntry()`.

Part III

Appendixes

Part III contains the following chapters:

A

Reference Documentation for the CIO

Reference documentation for the Oracle Configuration Interface Object is provided in the form of pages generated by the Javadoc tool from the source code for the CIO.

For the location of Javadoc pages for the this release, see the *About Oracle Configurator* documentation on Metalink, Oracle's technical support Web site.

B Code Examples

This chapter contains code examples illustrating the use of Configurator Extensions and the CIO. These examples are fuller and longer than the examples provided in the rest of this document, which are often fragments. For each example, see the cited background sections for explanatory details.

The sections of this chapter are:

- Generating Output Related to Model Structure
- Using Requests
- Sharing a Configuration Session in a Child Window
- Tracking Configuration Session Changes

B.1 Generating Output Related to Model Structure

This Configurator Extension produces an HTML representation of the runtime Model tree, beginning at a node specified in the Configurator Extension binding.

For the detailed procedure for creating a Configurator Extension Rule, see Chapter 2, "Building Configurator Extensions" and the *Oracle Configurator Developer User's Guide*. For specific information on building a Configurator Extension for generating custom output, see Section 3.2, "Generating Custom Output" on page 3-2.

Here is a summary of the tasks specific to this example:

- Use the Java source code in Example B–1 on page B-2 for your Java archive file and Configurator Extension Archive.
- When you define your Configurator Extension rule, use the options listed in the following table:

Option	Choose ...
Model Node	The node of your Model on which you want the button for the command event to be placed by Oracle Configurator. This node is independent of the node in the Model tree from which the Configurator Extension begins showing structure.
Java Class	`ShowStructureCX`, from your Configurator Extension Archive
Java Class Instantiation	With Model Node Instance

- When you define your event binding, use the options listed in the following table:

Option	Choose ...
Event	`onCommand`
Command Name	A string that you choose as a command. For example: `Show Structure`. Do not enclose the string in quotation marks. The string can contain spaces.
Event Scope	Your choice of scope. Try repeating the example with different scopes to see the effect when you test it.
Method Name	`showModelStructure`

- When you define your argument bindings, use the options listed in the following tables:

Option	Choose ...
Argument Type	`javax.servlet.http.HttpServletResponse`
Argument Specification	Event Parameter
Binding	HttpServletResponse

Option	Choose ...
Argument Type	`oracle.apps.cz.cio.IRuntimeNode`
Argument Specification	Model Node or Property
Binding	The node of your Model from which you want to begin showing hierarchical Model structure.

The example first calls the `response.setContentType()` method of the HttpServletResponse class, passing "text/html" as the output type.

The following line is required for compatibility with Microsoft Internet Explorer:

```
response.setHeader ("Expires", "-1");
```

Then the example calls `response.getWriter()` to get an output stream to which the Configurator Extension can write HTML.

You can also write non-HTML output by setting a different content type (a MIME type) and writing appropriate data to the output stream.

In the private method `generateNode()`, you can call either `IRuntimeNode.getCaption()`, as shown, or `IRuntimeNode.getName()`. However, `getCaption()` reflects changes to the name of a component instance made with `Component.setInstanceName()`, as described in Section 6.2.2, "Renaming Instances of Components" on page 6-3, while `getName()` does not.

Example B–1 Generating Output with a Configurator Extension (ShowStructureCX.java)

```
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.http.HttpServletResponse;
import com.sun.java.util.collections.Iterator;
import oracle.apps.cz.cio.IRuntimeNode;

/**
 * Displays a textual rendition of the model structure tree.
 *
 */

public class ShowStructureCX {

/**
 * Bind node parameter to the node from which to start rendering model structure.
 */

  public void showModelStructure(HttpServletResponse response, IRuntimeNode node) throws
IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("<html>");
    out.println("<head>");
    out.println("<title>Runtime Model Structure</title>");
    out.println("</head>");
    out.println("<body>");
    out.println("<h3>Runtime Model Structure</h3>");
    generateNode(out, node, 0);
    out.println("</body>");
    out.println("</html>");
  }

  private static void generateNode(PrintWriter out, IRuntimeNode node, int level) throws
IOException {
    for (int i = 0; i < level; ++i) {
      out.print("--");
    }
    //    out.println(node.getName() + " <br> "); // doesn't get changed instance names
    out.println(node.getCaption() + " <br> ");
    for (Iterator i = node.getChildren().iterator(); i.hasNext(); ) {
      IRuntimeNode childNode = (IRuntimeNode)i.next();
      generateNode(out, childNode, (level + 1));
    }
  }
}
```

B.2 Using Requests

For background, see Chapter 9, "Using Requests".

B.2.1 Setting Nonoverridable Requests

This example shows how to designate a group of requests as nonoverridable requests, by using `ConfigTransaction.useNonOverridableRequests()`.

For background, see Section 9.4, "Nonoverridable Requests" on page 9-2.

Example B–2 Setting Nonoverridable Requests (NonOverridableTest.java)

```
import oracle.apps.cz.cio.*;
import com.sun.java.util.collections.Iterator;

public class NonOverridableTest
{
  public void testOverride(Configuration config, IRuntimeNode comp) throws LogicalException {
    ConfigTransaction itr = null;
    try {
```

```
// Begin transaction that uses nonoverridable requests
itr = config.beginConfigTransaction();
itr.useNonOverridableRequests();

// Try setting an Option Feature with mutually exclusive Options.
IRuntimeNode of1 = comp.getChildByName("option_feature_1");
// Select option_1
ConfigTransaction tr = config.beginConfigTransaction();
((IOption)of1.getChildByName("option_1")).select();
config.commitConfigTransaction(tr);
// Select option_2
tr = config.beginConfigTransaction();
((IOption)of1.getChildByName("option_2")).select();
config.commitConfigTransaction(tr);

// Try setting a value for an Integer Feature.
tr = config.beginConfigTransaction();
((IInteger)comp.getChildByName("integer_feature_1")).setIntValue(33);
config.commitConfigTransaction(tr);

// Try overriding a Boolean value.
// boolean_feature_1 negates boolean_feature_2. This should produce a contradiction.
tr = config.beginConfigTransaction();
try {
  ((BooleanFeature)comp.getChildByName("boolean_feature_1")).setState(IState.TRUE);
  ((BooleanFeature)comp.getChildByName("boolean_feature_2")).setState(IState.TRUE);
} catch (LogicalOverridableException loe) {
  loe.override();
}
config.commitConfigTransaction(tr);

// Get next Component in Component set.
ComponentSet cset = (ComponentSet)comp.getParent().getChildByName("component_set_1");
Component cset_comp_1 = null;
Iterator iter = cset.getChildren().iterator();
if (iter.hasNext()) {
  cset_comp_1 = ((Component)iter.next());
}

// Try deleting a Component from a Component set.
// This is not allowed, and should produce a contradiction.
try {
  tr = config.beginConfigTransaction();
  cset.delete(cset_comp_1);
  config.commitConfigTransaction(tr);
} catch (Exception e) {
  config.rollbackConfigTransaction(tr);
  System.out.println("Expected exception in deleting component " + e);
}

// Try adding a Component to a Component set.
// This is not allowed, and should produce a contradiction.
try {
  tr = config.beginConfigTransaction();
  cset.add();
  config.commitConfigTransaction(tr);
} catch (Exception e) {
  config.rollbackConfigTransaction(tr);
  System.out.println("Expected exception in adding component " + e);
}
```

```
      // Try setting value of a Text Feature of Component in Component set
      tr = config.beginConfigTransaction();
      IRuntimeNode featText = cset_comp_1.getChildByName("text_feature_1");
      ((IText)featText).setTextValue("any_text");
      config.commitConfigTransaction(tr);

      // Try overriding default value of an Integer Feature of Component in Component set
      IRuntimeNode intFeatDef = comp.getParent().getChildByName("integer_feature_default");
      tr = config.beginConfigTransaction();
      ((IInteger)intFeatDef).setIntValue(50);  // Default value was 25
      config.commitConfigTransaction(tr);

      // Commit the transaction that used nonoverridable requests
      config.commitConfigTransaction(itr);

      // Try setting a nonoverridable request after a user request
      // Make an ordinary user request:
      tr = config.beginConfigTransaction();
      ((IState)comp.getChildByName("boolean_feature_3")).setState(IState.TRUE);
      config.commitConfigTransaction(tr);
      try {
    } catch (Exception e) {
      e.printStackTrace();
    } catch (Throwable t) {
        t.printStackTrace();
    }
  }
}
```

B.2.2 Getting a List of Failed Requests

This example shows how to use `LogicalOverridableException.override()` to override a logical contradiction and return a List of Request objects that represent all the previously asserted user requests that failed due to the override that you are performing.

For background, see Section 9.5, "Failed Requests" on page 9-4.

Example B–3 Getting a List of Failed Requests (OverrideTest.java)

```
import oracle.apps.cz.cio.*;
import oracle.apps.cz.common.*;
import oracle.apps.fnd.common.*;
import java.util.*;
import com.sun.java.util.collections.List;
import com.sun.java.util.collections.Iterator;

public class OverrideTest
{

  public static void main(String[] args)
  {
    ConfigTransaction tr = null;
    Configuration config = null;
    try {
      Class.forName("oracle.jdbc.driver.OracleDriver");
      WebAppsContext ctx = new WebAppsContext("server01_sid02"); // Use DBC file for context
      CIO cio = new CIO();
      config = cio.createConfiguration("overrideTest", ctx, null, Calendar.getInstance(),
Calendar.getInstance(), null, null);
      OptionFeature of = (OptionFeature)config.getRootComponent().getChildByName("Feature1");
      Option o1 = (Option) of.getChildByName("Option1");
      Option o2 = (Option) of.getChildByName("Option2");

      try {
        tr = config.beginConfigTransaction();
        o1.select();
        o2.deselect();
        config.commitConfigTransaction(tr);
      } catch (LogicalOverridableException loe) {
        try {
          // Get list of failed requests, if any
          List list = loe.override();
          System.out.println("Option1: " + o1+ " State: " + o1.getState());
          System.out.println("Option2: " + o2+ " State: " + o2.getState());
          printList(list);
          config.commitConfigTransaction(tr);
        } catch (Exception re) {
          re.printStackTrace();
          config.rollbackConfigTransaction(tr);
        }
      } catch (LogicalException le) {
          le.printStackTrace();
          config.rollbackConfigTransaction(tr);
      }

    } catch (Exception e) {
      e.printStackTrace();
    }
  }

  public static void printList(List list) {
    Iterator iter = list.iterator();
    while (iter.hasNext()) {
      System.out.println("Node: " + iter.next());
    }
    System.out.println("***************\n");
  }
}
```

B.3 Sharing a Configuration Session in a Child Window

This example must use a child window of the kind described in Section 5.10, "Sharing a Configuration Session" on page 5-10, which describes the background and purpose of the example. The child window must be created with the HTML-based version of Oracle Configurator Developer and run with a generated Configurator UI for the runtime Oracle Configurator.

This JSP generates the contents of a child window and performs the following tasks:

- Imports the necessary user classes by importing the CIO. Session-related classes, such as `PageContext`, are supplied by your servlet/JSP container.
- Gets the session's `Configuration` object (`cfg`) through the session key `configurationObject`. This allows the child window to modify the same configuration as the parent window.

- Gets the URL of the runtime Configurator in the parent window (`retUrl`) through the session key `czReturnToConfiguratorUrl`, so that control can return to it when the child window is closed.
- Modifies the state of the current configuration.

  Example code for modifying the runtime configuration from the child window is shown after the comment // Start configuration changes here.. For simplicity, this code illustrates only basic interaction with the configuration model. For true interaction with the configuration model, you must tailor the code to your own circumstances.

  The example here locates a node named `Boolean Feature-1`, checks whether it exists and is a Boolean Feature, and, if so, toggles its state. This action is performed when the end user clicks a button like that described in Table 5–4, " UI Specifications for Invoking Child Window" on page 5-10.

  For background on modifying the runtime configuration model, see Chapter 6, "Working with Model Entities". For details on toggling state, see Example 6–5, "Setting the State of a Node" on page 6-6 in Section 6.4, "Getting and Setting Logic States".
- Provides a button (labeled `Close`), which refreshes the parent window with the results of the child window's actions then closes the child window. This button calls a function, `refreshMainWdw()`, that uses the URL of the parent window (`retUrl`) to return control to it.

Example B–4 Sharing a Configuration Session in a Child Window (TestChildWin.jsp)

```
<%@ page contentType="text/html;charset=windows-1252"
    import="oracle.apps.cz.cio.*"%>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=windows-1252">
<title>
Test Child Window
</title>
</head>
<body>
<%
  // Get the session's configuration object, through javax.servlet.jsp.PageContext
  Configuration cfg = (Configuration)pageContext.getAttribute("configurationObject",
PageContext.SESSION_SCOPE);
  // Get URL of the runtime Configurator, so we can return to it.
  String retUrl = (String)pageContext.getAttribute("czReturnToConfiguratorUrl",
PageContext.SESSION_SCOPE);
  if (cfg != null) {
    out.println("<p>Got Configuration object from HTTP session. Can now modify the
configuration.</p>");

  // Start configuration changes here.
    IRuntimeNode node = cfg.getRootComponent().getChildByName("Boolean Feature-1");
    if (node != null && node instanceof BooleanFeature) {
      ((BooleanFeature)node).setState(IState.TOGGLE);
    }
  // End configuration changes here.

  }
%>
<script>
  function refreshMainWdw() {
    opener.location="<%= retUrl %>";
    window.close();

  }
</script>
<form>
  <input type="button" name="b1" value="Close" onclick="javascript:refreshMainWdw();">
</form>
</body>
</html>
```

B.4 Tracking Configuration Session Changes

The code in Example B–5 assembles together the fragmentary examples shown in Chapter 10, "Configuration Session Change Tracking".

Example B–5 Tracking Session Changes (DeltaExample.java)

```
import com.sun.java.util.collections.*;
import oracle.apps.cz.cio.*;
import oracle.apps.cz.common.CZWebAppsContext;
import oracle.apps.fnd.common.Context;

public class DeltaExample
{

   public static void main(String [] args) {
    // Define some constants
    int modelId = 1234;
    String dbcFilename = "dbcFile.dbc";
    String user = "scott";
    String pwd = "tiger";

    try {
      // Load the JDBC Driver and create Context, CIO
      Class.forName("oracle.jdbc.driver.OracleDriver");
      Context context = new CZWebAppsContext(dbcFilename);
      context.getSessionManager().validateLogin( user, pwd);
      CIO cio = new CIO();
      cio.initializeAppsSession(context);

      // Create a new Configuration and DeltaManager
      ConfigParameters params = new ConfigParameters(modelId);
      Configuration config = cio.startConfiguration(params, context);
      DeltaManager deltaMgr = config.createDeltaManager("MyDeltaMgr");

      // Create a Navigation (Tree) region. This is interested in watching
      // all runtime nodes for instance name, instantiation, and unsatisfaction
      // changes.
      List dvList = new ArrayList();

      dvList.add(deltaMgr.getDeltaValidator(DeltaValidator.INSTANCE_NAME_DV));
      dvList.add(deltaMgr.getDeltaValidator(DeltaValidator.INSTANTIATION_DV));
      dvList.add(deltaMgr.getDeltaValidator(DeltaValidator.UNSATISFACTION_DV));

      List watchedNodes = config.getRuntimeNodes();

      DeltaRegion treeRegion = deltaMgr.registerRegion(watchedNodes, dvList, "MyTreeRegion");
      // Create a component region. This region displays a Component screen and is
      // interested in watching all nodes in that component for availability, count,
      // price, state and unsatisfaction changes
      dvList.clear();
      dvList.add(deltaMgr.getDeltaValidator(DeltaValidator.AVAILABILITY_DV));
      dvList.add(deltaMgr.getDeltaValidator(DeltaValidator.COUNT_DV));
      dvList.add(deltaMgr.getDeltaValidator(DeltaValidator.PRICE_DV));
      dvList.add(deltaMgr.getDeltaValidator(DeltaValidator.STATE_DV));
      dvList.add(deltaMgr.getDeltaValidator(DeltaValidator.UNSATISFACTION_DV));

      watchedNodes = getRuntimeNodesInSelectedComponent(); // a custom method, not defined here

      DeltaRegion compRegion = deltaMgr.registerRegion(watchedNodes, dvList, "MyCompRegion");

      // Make an assertion to change the current configuration
      Option option1 =
(Option)config.getRootComponent().getChildByName("Feature").getChildByName("Option1");
      option1.select();

      // Get the deltas due to this assertion and update the tree and component regions
      Map treeChanges = deltaMgr.getUpdateMapForRegion("MyTreeRegion");

      // Now update the tree region cache and UI with treeChanges
      updateTreeRegion(treeChanges);

      Map compChanges = compRegion.getUpdateMap();
      updateCompRegion(compChanges); // a custom method, not defined here
    } catch (Exception e) {
      e.printStackTrace();
    }
  }

  public static void updateTreeRegion(Map changes) {
    for (Iterator iter = changes.keySet().iterator(); iter.hasNext();) {
      RuntimeNode changedNode = (RuntimeNode)iter.next();
      uiNode = getUiNode(changedNode); // custom method
      Collection nodeChanges = (Collection)changes.get(changedNode);
      for (Iterator iter2 = nodeChanges.iterator(); iter2.hasNext();) {
        IValidatorChange change = (IValidatorChange)iter2.next();
        switch (change.getType()) {
          case DeltaValidator.INSTANCE_NAME_DV:
            InstanceNameDeltaValidator.InstanceNameChange nameChange =
(InstanceNameDeltaValidator.InstanceNameChange)change;
            String newName = nameChange.getInstanceName();
            uiNode.setName(newName); // custom method on uiNode
            break;
          case DeltaValidator.INSTANTIATION_DV:
            InstantiationDeltaValidator.InstantiationChange iChange =
(InstantiationDeltaValidator.InstantiationChange) change;
            Collection added = iChange.getNewlyAddedInstances();
            Collection deleted = iChange.getNewlyDeletedInstances();
            uiNode.updateInstances(added, deleted); // custom method on uiNode
            break;
          case DeltaValidator.UNSATISFACTION_DV:
            UnsatisfactionDeltaValidator.UnsatisfactionChange uChange =
(UnsatisfactionDeltaValidator.UnsatisfactionChange) change;
            boolean unsatisfied = uChange.isUnsatisfied();
            uiNode.setUnsatisfied(unsatisfied); // custom method on uiNode
            break;
        }
```

```
         }
      }
   }
}
```

C

Java Parameter Types for Configurator Extensions

When you are creating Configurator Extensions with Oracle Configurator Developer, you must be able to bind an entity in your Model as an argument to a parameter in the Java method that you have selected.

The Java types of the parameters of your method must agree with the types of Model entities that are eligible for event binding. For a list of the Java classes that you can use in event bindings, see Example C–1, "Valid Java Types for Parameters". on page C-1.

For information on developing Java methods for Configurator Extensions, see Section 2.2, "Developing Java Classes and Archives" on page 2-4.

Example C–1 Valid Java Types for Parameters

```
boolean
com.sun.java.util.collections.Collection
com.sun.java.util.collections.List
double
float
int
java.lang.Integer
java.lang.Long
java.lang.Object
java.lang.String
java.long.Double
java.long.Float
java.text.DecimalFormat
java.utils.Date
javax.servlet.http.HttpServletResponse
long
oracle.apps.cz.cio.BomInstance
oracle.apps.cz.cio.BomModel
oracle.apps.cz.cio.BomNode
oracle.apps.cz.cio.BomOptionClass
oracle.apps.cz.cio.BomStdItem
oracle.apps.cz.cio.BooleanFeature
oracle.apps.cz.cio.CXEvent
oracle.apps.cz.cio.CXRule
oracle.apps.cz.cio.Component
oracle.apps.cz.cio.ComponentInstance
oracle.apps.cz.cio.ComponentSet
oracle.apps.cz.cio.Configuration
oracle.apps.cz.cio.Connector
oracle.apps.cz.cio.CountFeature
oracle.apps.cz.cio.DecimalFeature
oracle.apps.cz.cio.DecimalNode
oracle.apps.cz.cio.IAtp
oracle.apps.cz.cio.IBomItem
oracle.apps.cz.cio.ICount
oracle.apps.cz.cio.IDecimal
oracle.apps.cz.cio.IDecimalMinMax
oracle.apps.cz.cio.IInstance
oracle.apps.cz.cio.IInteger
oracle.apps.cz.cio.IIntegerMinMax
oracle.apps.cz.cio.IOption
oracle.apps.cz.cio.IOptionFeature
oracle.apps.cz.cio.IPrice
oracle.apps.cz.cio.IReadOnlyDecimal
oracle.apps.cz.cio.IRuntimeNode
oracle.apps.cz.cio.IState
oracle.apps.cz.cio.IText
oracle.apps.cz.cio.IntegerFeature
oracle.apps.cz.cio.IntegerNode
oracle.apps.cz.cio.Option
oracle.apps.cz.cio.OptionFeature
oracle.apps.cz.cio.OptionFeatureNode
oracle.apps.cz.cio.OptionNode
oracle.apps.cz.cio.PricedNode
oracle.apps.cz.cio.ReadOnlyDecimalNode
oracle.apps.cz.cio.Resource
oracle.apps.cz.cio.RuntimeNode
oracle.apps.cz.cio.TextFeature
oracle.apps.cz.cio.TextNode
oracle.apps.cz.cio.Total
void
```

Glossary

This glossary contains definitions that you may need while working with Oracle Configurator.

API

Application Programming Interface

applet

A Java application running inside a Web browser. *See also* **Java** and **servlet**.

Archive Path

The ordered sequence of **Configurator Extension Archive**s for a **Model** that determines which **Java class**es are loaded for **Configurator Extension**s and in what order.

argument

A data value or object that is passed to a method or a **Java class** so that the method can operate.

ATO

Assemble to Order

ATP

Available to Promise

base node

The **node** in a **Model** that is associated with a **Configurator Extension** Rule. Used to determine the **event** scope for a **Configurator Extension**.

bill of material

A list of Items associated with a parent Item, such as an assembly, and information about how each Item relates to that parent Item.

Bills of Material

The application in Oracle Applications in which you define a **bill of material**.

binding

Part of a **Configurator Extension** Rule that associates a specified event with a chosen **method** of a **Java class**. *See also* **event**.

BOM

See **bill of material**.

BOM item

The **node** imported into **Oracle Configurator Developer** that corresponds to an Oracle **Bills of Material** item. Can be a **BOM Model**, **BOM Option Class node**, or **BOM Standard Item node**.

BOM Model

A model that you import from Oracle **Bills of Material** into **Oracle Configurator Developer**. When you import a BOM Model, effective dates, **ATO** rules, and other data are also imported into Configurator Developer. In Configurator Developer, you can extend the structure of the BOM Model, but you cannot modify the BOM Model itself or any of its attribute**s**.

BOM Model node

The imported **node** in **Oracle Configurator Developer** that corresponds to a **BOM Model** created in Oracle **Bills of Material**.

BOM Option Class node

The imported **node** in **Oracle Configurator Developer** that corresponds to a BOM Option Class created in Oracle **Bills of Material**.

BOM Standard Item node

The imported **node** in **Oracle Configurator Developer** that corresponds to a BOM Standard Item created in Oracle **Bills of Material**.

Boolean Feature

An **element** of a **component** in the **Model** that has two **option**s: true or false.

bug

See **defect**.

build

A specific **instance** of an application during its construction. A build must have an install program early in the project so that application **implementer**s can **unit test** their latest work in the context of the entire available application.

CDL

See **Constraint Definition Language**.

CIO

See **Oracle Configuration Interface Object (CIO)**.

command event

An **event** that is defined by a character string, which is considered the command for which **listener**s are listening.

Comparison Rule

An **Oracle Configurator Developer** rule type that establishes a relationship to determine the selection state of a logical **Item** (Option, Boolean Feature, or List-of-Options Feature) based on a comparison of two numeric values (numeric **Feature**s, **Total**s, **Resource**s, **Option** counts, or numeric constants). The numeric values being compared can be computed or they can be discrete intervals in a continuous numeric input.

Compatibility Rule

An **Oracle Configurator Developer** rule type that establishes a relationship among **Features** in the Model to control the allowable combinations of **Options**. *See also*, **Property-based Compatibility Rule**.

Compatibility Table

A kind of Explicit Compatibility Rule. For example, a type of compatibility relationship where the allowable combination of **Options** are explicitly enumerated.

component

A piece of something or a configurable element in a **model** such as a **BOM Model**, **Model**, or **Component**.

Component

An element of the **model structure**, typically containing **Features**, that is configurable and instantiable. An **Oracle Configurator Developer** node type that represents a configurable element of a **Model**. Corresponds to one UI screen of selections in a runtime **Oracle Configurator**.

Component Set

An element of the **Model** that contains a number of instantiated **Components** of the same type, where each Component of the set is independently configured.

concurrent program

Executable code (usually written in SQL*Plus or Pro*C) that performs the function(s) of a requested task. Concurrent programs are stored procedures that perform actions such as generating reports and copying data to and from a database.

configuration

A specific set of specifications for a product, resulting from selections made in a runtime **configurator**.

configuration attribute

A characteristic of an **item** that is defined in the **host application** (outside of its inventory of items), in the **Model**, or captured during a **configuration session**. Configuration attributes are inputs from or outputs to the host application at initialization and termination of the configuration session, respectively.

configuration engine

The part of the runtime **Oracle Configurator** that uses **configuration rules** to validate a **configuration**. Compare **generated logic**.

Configuration Interface Object

See **Oracle Configuration Interface Object (CIO)**.

configuration model

Represents all possible configurations of the available **options**, and consists of **model structure** and **rules**. It also commonly includes **User Interface** definitions and **Configurator Extensions**. A configuration model is usually accessed in a **runtime Oracle Configurator window**. *See also* **model**.

configuration rule

A **Logic Rule**, **Compatibility Rule**, **Comparison Rule**, **Numeric Rule**, **Design Chart**, **Statement Rule**, or **Configurator Extension** rule available in **Oracle Configurator Developer** for defining **configurations**. *See also* **rules**.

configuration session

The time from launching or invoking to exiting **Oracle Configurator**, during which **end users** make selections to configure an orderable product. A configuration session is limited to one **configuration model** that is loaded when the session is initialized.

configurator

The part of an application that provides custom configuration capabilities. Commonly, a window that can be launched from a host application so **end users** can make selections resulting in valid **configurations**. *Compare* **Oracle Configurator**.

Configurator Extension

An extension to the **configuration model** beyond what can be implemented in Configurator Developer.

A type of **configuration rule** that associates a **node**, **Java class**, and event **binding** so that the rule operates when an **event** occurs during a **configuration session**.

A **Java class** that provides methods that can be used to perform configuration actions.

Configurator Extension Archive

An **object** in the **Repository** that stores one or more compiled **Java class**es that implement **Configurator Extension**s.

connectivity

The connection between client and database that allows data communication.

The connection across components of a model that allows modeling such products as networks and material processing systems.

Connector

The **node** in the **model structure** that enables an **end user** at **runtime** to connect the Connector node's parent to a referenced **Model**.

Constraint Definition Language

A language for entering **configuration rule**s as text rather than assembling them interactively in Oracle Configurator Developer. CDL can express more complex constraining relationships than interactively defined configuration rules can.

Container Model

A type of **BOM Model** that you import from Oracle **Bills of Material** into **Oracle Configurator Developer** to create configuration models containing **connectivity** and trackable components. Configurations created from Container Models can be tracked and updated in Oracle Install Base

Contributes to

A relation used to create a specific type of **Numeric Rule** that accumulates a total value. *See also* **Total**.

Consumes from

A relation used to create a specific type of **Numeric Rule** that decrements a total value, such as specifying the quantity of a **Resource** used.

count

The number or quantity of something, such as selected **options**. *Compare* **instance**.

CTO

Configure to Order

customer

The person for whom products are configured by **end users** of the **Oracle Configurator** or other **ERP** and CRM applications. Also the end users themselves directly accessing **Oracle Configurator** in a Web store or kiosk.

customer requirements

The needs of the customer that serve as the basis for determining the configuration of products, **systems**, and services. Also called needs assessment. *See* **guided buying or selling**.

CZ

The product shortname for **Oracle Configurator** in Oracle Applications.

CZ schema

The implementation version of the standard runtime **Oracle Configurator** data-warehousing schema that manages data for the **configuration model**. The implementation schema includes all the data required for the **runtime** system, as well as specific tables used during the construction of the **configurator**.

data import

Populating the **CZ schema** with enterprise data from **ERP** or legacy systems via **import tables**.

data source

A programmatic reference to a database. Referred to by a data source name (DSN).

DBMS

Database Management System

default

A predefined value. In a **configuration**, the automatic selection of an **option** based on the **preselection** rules or the selection of another option.

Defaults relation

An **Oracle Configurator Developer** Logic Rule relation that determines the logic state of **Features** or **Options** in a default relation to other Features and Options. For example, if A Defaults B, and you select A, B becomes Logic True (selected) if it is available (not Logic False).

defect

A failure in a product to satisfy the **users'** requirements. Defects are prioritized as critical, major, or minor, and fixes range from corrections or workarounds to enhancements. Also known as a bug.

Design Chart

An **Oracle Configurator Developer** rule type for defining advanced Explicit Compatibilities interactively in a table view.

developer

The person who uses **Oracle Configurator Developer** to create a **configurator**. *See also* **implementer** and **user**.

Developer

The tool (**Oracle Configurator Developer**) used to create **configuration models**.

DHTML

Dynamic Hypertext Markup Language

discontinued item

A discontinued item is one that exists in an installed configuration of a component (as recorded in Oracle Install Base), but has been removed from the instance of the component being reconfigured, either by deletion or by deselection.

element

Any entity within a **model**, such as **Options**, **Totals**, **Resources**, UI controls, and **components**.

end user

The ultimate user of the runtime **Oracle Configurator**. The types of end users vary by project but may include salespeople or distributors, administrative office staff, marketing personnel, order entry personnel, product engineers, or customers directly accessing the application via a Web browser or kiosk. *Compare* **user**.

enterprise

The **systems** and **resources** of a business.

environment

The arena in which software tools are used, such as operating system, applications, and **server** processes.

ERP

Enterprise Resource Planning. A software system and process that provides automation for the customer's back-room operations, including order processing.

event

An action or condition that occurs in a **configuration session** and can be detected by a **listener**. Example events are a change in the value of a **node**, the creation of a component **instance**, or the saving of a **configuration**. The part of **model structure** inside which a **listener** listens for an event is called the event **binding** scope. The part of model structure that is the source of an event is called the event execution scope. *See also* **command event**.

Excludes relation

An **Oracle Configurator Developer Logic Rule** type that determines the logic state of **Features** or **Options** in an excluding relation to other Features and Options. For example, if A Excludes B, and if you select A, B becomes Logic False, since it is not allowed when A is true (either User or Logic True). If you deselect A (set to User

False), there is no effect on B, meaning it could be User or Logic True, User or Logic False, or **Unknown**. *See* **Negates relation**.

feature

A characteristic of something, or a configurable element of a **component** at **runtime**.

Feature

An element of the **model structure**. Features can either have a value (numeric or Boolean) or enumerated **Options**.

functional specification

Document describing the functionality of the application based on **user** requirements.

generated logic

The compiled structure and rules of a **configuration model** that is loaded into memory on the Web server at **configuration session** initialization and used by the **Oracle Configurator engine** to validate runtime selections. The logic must be generated either in **Oracle Configurator Developer** or programmatically in order to access the configuration model at **runtime**.

guided buying or selling

Needs assessment questions in the **runtime** UI to guide and facilitate the configuration process. Also, the **model structure** that defines these questions. Typically, guided selling questions trigger **configuration rule** that automatically select some product **options** and exclude others based on the **end user's** responses.

host application

An application within which **Oracle Configurator** is embedded as integrated functionality, such as Order Management or *i*Store.

HTML

Hypertext Markup Language

implementation

The stage in a project between defining the problem by selecting a configuration technology vendor, such as Oracle, and deploying the completed configuration application. The implementation stage includes gathering requirements, defining test cases, designing the application, constructing and testing the application, and delivering it to **end users**. *See also* **developer** and **user**.

implementer

The person who uses **Oracle Configurator Developer** to build the **model structure**, **rules**, and UI customizations that make up a **runtime** Oracle Configurator. Commonly also responsible for enabling the integration of **Oracle Configurator** in a **host application**.

Implies relation

An **Oracle Configurator Developer Logic Rule** type that determines the logic state of **Features** or **Options** in an implied relation to other Features and Options. For example, if A Implies B, and you select A, B becomes Logic True. If you deselect A (set to User False), there is no effect on B, meaning it could be User or Logic True, User or Logic False, or **Unknown**. *See* **Requires relation**.

import server

A database **instance** that serves as a source of data for **Oracle Configurator**'s Populate, Refresh, and Synchronization concurrent processes. The import server is sometimes referred to as the remote server.

import tables

Tables mirroring the CZ schemaItem Master structure, but without integrity constraints. Import tables allow batch population of the CZ schema's Item Master. Import tables also store extractions from Oracle Applications or **legacy data** that create, update, or delete records in the CZ schema **Item Master**.

initialization message

The **XML** message sent from a **host application** to the **Oracle Configurator Servlet**, containing data needed to initialize the runtime Oracle Configurator. *See also* **termination message**.

Instance

An **Oracle Configurator Developer** attribute of a **component's node** that specifies a minimum and maximum value. *See also* **instance**.

instance

A **runtime** occurrence of a **component** in a configuration. *See also* **instantiate**. *Compare* **count**.

Also, the memory and processes of a database.

instantiate

To create an instance of something. Commonly, to create an **instance** of a **component** in the runtime **user interface** of a **configuration model**.

integration

The process of combining multiple software **components** and making them work together.

integration testing

Testing the interaction among software programs that have been integrated into an application or **system**. Also called system testing. *Compare* **unit test**.

item

A product or part of a product that is in inventory and can be delivered to customers.

Item

A Model or part of a Model that is defined in the **Item Master**. Also data defined in Oracle Inventory.

Item Master

Data stored to structure the Model. Data in the **CZ schema** Item Master is either entered manually in **Oracle Configurator Developer** or imported from Oracle Applications or a legacy system.

Item Type

Data used to classify the Items in the Item Master. Item Catalogs imported from Oracle Inventory are Item Types in **Oracle Configurator Developer**.

Java

An object-oriented programming language commonly used in internet applications, where Java applications run inside Web browsers and **server**s. Used to implement the behavior of **Configurator Extension**s. *See also* **applet** and **servlet**.

Java class

The compiled version of a **Java** source code file. The **method**s of a Java class are used to implement the behavior of **Configurator Extension**s.

JavaServer Pages

Web pages that combine static presentation elements with dynamic content that is rendered by Java **servlet**s.

JSP

See **JavaServer Pages**.

legacy data

Data that cannot be imported without creating custom extraction programs.

listener

A class in the **CIO** that detects the occurrence of specified **event**s in a **configuration session**.

load

Storing the **configuration model** data in the **Oracle Configurator Servlet** on the Web server. Also, the time it takes to initialize and display a configuration model if it is not preloaded.

The burden of transactions on a **system**, commonly caused by the ratio of **user** connections to CPUs or available memory.

log file

A file containing errors, warnings, and other information that is output by the running application.

Logic Rule

An **Oracle Configurator Developer** rule type that expresses constraint among model elements in terms of logic relationships. Logic Rules directly or indirectly set the logical state (User or Logic True, User or Logic False, or **Unknown**) of **Features** and **Options** in the Model.

There are four primary Logic Rule relations: Implies, Requires, Excludes, and Negates. Each of these rules takes a list of Features or Options as operands. *See also* **Implies relation**, **Requires relation**, **Excludes relation**, and **Negates relation**.

maintainability

The characteristic of a product or process to allow straightforward **maintenance**, alteration, and extension. Maintainability must be built into the product or process from inception.

maintenance

The effort of keeping a **system** running once it has been deployed, through **defect** fixes, procedure changes, infrastructure adjustments, data replication schedules, and so on.

Metalink

Oracle's technical support Web site at:

`http://www.oracle.com/support/metalink/`

method

A function that is defined in a **Java class**. Methods perform some action and often accept parameters.

Model

The entire hierarchical "tree" view of all the data required for **configurations**, including **model structure**, variables such as **Resources** and **Totals**, and elements in support of intermediary rules. Includes both imported **BOM Models** and Models created in Configurator Developer. May consist of BOM Option Classes and BOM Standard Items.

model

A generic term for data representing products. A model contains **elements** that correspond to **items**. Elements may be **components** of other objects used to define products. A **configuration model** is a specific kind of model whose elements can be configured by accessing an **Oracle Configurator window**.

model-driven UI

The graphical views of the **model structure** and **rules** generated by **Oracle Configurator Developer** to present **end users** with interactive product selection based on **configuration models**.

model structure

Hierarchical "tree" view of data composed of **elements** (**Models**, **Components**, **Features**, **Options**, **BOM Models**, **BOM Option Class nodes**, **BOM Standard Item nodes**, **Resources**, and **Totals**). May include reusable **components** (**References**).

Negates relation

A type of **Oracle Configurator Developer Logic Rule** type that determines the logic state of **Features** or **Options** in a negating relation to other Features and Options. For example, if one **option** in the relationship is selected, the other option must be Logic False (not selected). Similarly, if you deselect one option in the relationship, the other option must be Logic True (selected). *See* **Excludes relation**.

node

The icon or location in a **Model** tree in **Oracle Configurator Developer** that represents a **Component**, **Feature**, **Option** or variable (**Total** or **Resource**), **Connector**, **Reference**, **BOM Model**, **BOM Option Class node**, or **BOM Standard Item node**.

Numeric Rule

An **Oracle Configurator Developer** rule type that expresses constraint among model elements in terms of numeric relationships. *See also*, **Contributes to** and **Consumes from**.

object

Entities in **Oracle Configurator Developer**, such as **Models**, Usages, Properties, Effectivity Sets, UI Templates, and so on. *See also* **element**.

OC

See **Oracle Configurator**.

OCD

See **Oracle Configurator Developer**.

option

A logical selection made in the Model Debugger or a runtime Oracle Configurator by the **end user** or a rule when configuring a **component**.

Option

An element of the **Model**. A choice for the value of an enumerated **Feature**.

Oracle Configuration Interface Object (CIO)

A **server** in the **runtime** application that creates and manages the interface between the client (usually a **user interface**) and the underlying representation of **model structure** and **rules** in the **generated logic**.

The CIO is the **API** that supports creating and navigating the Model, querying and modifying selection states, and saving and restoring **configurations**.

Oracle Configurator

The product consisting of development tools and **runtime** applications such as the **CZ schema**, **Oracle Configurator Developer**, and runtime Oracle Configurator. Also the runtime Oracle Configurator variously packaged for use in networked or Web deployments.

Oracle Configurator architecture

The three-tier **runtime** architecture consists of the **User Interface**, the **generated logic**, and the **CZ schema**. The application development architecture consists of **Oracle Configurator Developer** and the CZ schema, with test instances of a runtime **Oracle Configurator**.

Oracle Configurator Developer

The suite of tools in the **Oracle Configurator** product for constructing and maintaining **configurators**.

Oracle Configurator engine

The part of the **Oracle Configurator** product that validates runtime selections. *See also* **generated logic**.

Oracle Configurator schema

See **CZ schema**.

Oracle Configurator Servlet

A **Java** servlet that participates in rendering Legacy user interfaces for **Oracle Configurator**.

Oracle Configurator window

The **user interface** that is launched by accessing a **configuration model** and used by **end users** to make the selections of a **configuration**.

performance

The operation of a product, measured in throughput and other data.

Populator

An entity in **Oracle Configurator Developer** that creates **Component**, **Feature**, and **Option nodes** from information in the **Item Master**.

preselection

The default state in a **configurator** that defines an initial selection of **Components**, **Features**, and **Options** for configuration.

A process that is implemented to select the initial element(s) of the **configuration**.

product

Whatever is ordered and delivered to customers, such as the output of having configured something based on a model. Products include intangible entities such as services or contracts.

Property

A named value associated with a **node** in the **Model** or the **Item Master**. A set of Properties may be associated with an Item Type. After importing a BOM Model, Oracle Inventory Catalog Descriptive Elements are Properties in **Oracle Configurator Developer**.

Property-based Compatibility Rule

An **Oracle Configurator Developer** Compatibility Rule type that expresses a kind of compatibility relationship where the allowable combinations of **Options** are specified implicitly by relationships among Property values of the Options.

prototype

A construction technique in which a preliminary version of the application, or part of the application, is built to facilitate **user** feedback, prove feasibility, or examine other implementation issues.

PTO

Pick to Order

publication

A unique deployment of a **configuration model** (and optionally a **user interface**) that enables a developer to control its availability from host applications such as Oracle Order Management or *i*Store. Multiple publications can exist for the same configuration model, but each publication corresponds to only one **Model** and **User Interface**.

publishing

The process of creating a **publication** record in **Oracle Configurator Developer**, which includes specifying applicability parameters to control **runtime** availability and running an Oracle Applications concurrent process to copy data to a specific database.

RDBMS

Relational Database Management System

reference

The ability to reuse an existing **Model** or **Component** within the structure of another Model (for example, as a subassembly).

Reference

An **Oracle Configurator Developer** node type that denotes a **reference** to another **Model**.

Repository

Set of pages in **Oracle Configurator Developer** that contains areas for organizing and maintaining **Model**s and shared **object**s in a single location.

Requires relation

An **Oracle Configurator Developer** Logic Rule relationship that determines the logic state of **Features** or **Options** in a requirement relation to other Features and Options. For example, if A Requires B, and if you select A, B is set to Logic True (selected). Similarly, if you deselect A, B is set to Logic False (deselected). See **Implies relation**.

resource

Staff or equipment available or needed within an enterprise.

Resource

A variable in the **Model** used to keep track of a quantity or supply, such as the amount of memory in a computer. The value of a Resource can be positive or zero, and can have an Initial Value setting. An error message appears at **runtime** when the value of a Resource becomes negative, which indicates it has been over-consumed. Use **Numeric Rules** to contribute to and consume from a Resource.

Also a specific node type in **Oracle Configurator Developer**. *See also* **node**.

reusable component

See **reference** and **model structure**.

reusability

The extent to and ease with which parts of a **system** can be put to use in other systems.

rules

Also called business rules or **configuration rule**. In the context of Oracle Configurator and **CDL**, a rule is not a "business rule." Constraints applied among elements of the product to ensure that defined relationships are preserved during configuration. Elements of the product are **Components**, **Features**, and **Options**. Rules express logic, numeric parameters, implicit compatibility, or explicit compatibility. Rules provide **preselection** and **validation** capability in **Oracle Configurator**.

See also **Comparison Rule**, **Compatibility Rule**, **Design Chart**, **Logic Rule** and **Numeric Rule**.

runtime

The environment and context in which applications are run, tested, or used, rather than developed.

The environment in which an **implementer** (tester), **end user**, or **customer** configures a product whose model was developed in **Oracle Configurator Developer**. *See also* **configuration session**.

schema

The tables and objects of a data model that serve a particular product or business process. *See also* **CZ schema**.

server

Centrally located software processes or hardware, shared by client**s**.

servlet

A Java application running inside a Web server. *See also* **Java**, **applet**, and **Oracle Configurator Servlet**.

solution

The deployed **system** as a response to a problem or problems.

SQL

Structured Query Language

Statement Rule

An **Oracle Configurator Developer** rule type defined by using the Oracle Configurator **Constraint Definition Language** (text) rather than interactively assembling the rule's elements.

system

The hardware and software **component**s and infrastructure integrated to satisfy functional and **performance** requirements.

termination message

The **XML** message sent from the **Oracle Configurator Servlet** to a **host application** after a **configuration session**, containing configuration outputs. *See also* **initialization message**.

Total

A variable in the **Model** used to accumulate a numeric total, such as total price or total weight.

Also a specific node type in **Oracle Configurator Developer**. *See also* **node**.

UI

See **User Interface**.

UI Templates

Templates available in **Oracle Configurator Developer** for specifying UI definitions.

Unknown

The logic state that is neither true nor false, but unknown at the time a **configuration session** begins or when a Logic Rule is executed. This logic state is also referred to as Available, especially when considered from the point of view of the **runtime Oracle Configurator end user**.

unit test

Execution of individual routines and modules by the application **implementer** or by an independent test consultant to find and resolve **defect**s in the application. *Compare* **integration testing**.

update

Moving to a new version of something, independent of software release. For instance, moving a production **configurator** to a new version of a **configuration model**, or changing a **configuration** independent of a model **update**.

upgrade

Moving to a new release of **Oracle Configurator** or **Oracle Configurator Developer**.

user

The person using a product or system. Used to describe the person using **Oracle Configurator Developer** tools and methods to build a **runtime Oracle Configurator**. *Compare* **end user**.

User Interface

The part of an **Oracle Configurator** implementation that provides the graphical views necessary to create **configuration**s interactively. A **user interface** is generated from the **model structure**. It interacts with the model definition and the **generated logic** to give **end user**s access to customer requirements gathering, product selection, and any extensions that may have been implemented. *See also* **UI Templates**.

user interface

The visible part of the application, including menus, dialog boxes, and other on-screen elements. The part of a **system** where the **user** interacts with the software. Not necessarily generated in **Oracle Configurator Developer**. *See also* **User Interface**.

user requirements

A description of what the **configurator** is expected to do from the **end user's** perspective.

validation

Tests that ensure that configured **components** will meet specific criteria set by an enterprise, such as that the components can be ordered or manufactured.

variable

Parts of the **Model** that are represented by **Totals**, **Resources**, or numeric **Features**.

verification

Tests that check whether the result agrees with the specification.

Web

The portion of the Internet that is the World Wide Web.

Workbench

Set of pages in **Oracle Configurator Developer** for creating, editing, and working with **Repository objects** such as **Models** and **UI Templates**.

XML

Extensible Markup Language, a highly flexible markup language for transferring data between **Web** applications. Used for the **initialization message** and **termination message** of the **Oracle Configurator Servlet**.

Index