Oracle® Configuration Interface Object (CIO)

Developer's Guide

Release 11*i*

Part No. B10617-01

Februrary 2003

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

ORACLE®

Oracle Configuration Interface Object (CIO) Developer's Guide, Release 11*i*

Part No. B10617-01

Primary Author: Mark Sawtelle

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

Contents

2 Using Functional Companions

3 Using the Configuration Interface Object (CIO)

List of Figures

List of Examples

List of Tables

Send Us Your Comments

Oracle Configuration Interface Object (CIO) Developer's Guide, Release 11*i*

Part No. B10617-01

Oracle Corporation 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 Functional Companions to augment the functionality of your runtime Oracle Configurator beyond what is provided by Oracle Configurator Developer. You create Functional Companion objects, which use the Configuration Interface Object (CIO) to perform various tasks, including accessing the Model, setting and getting logic states, and adding optional 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 Functional Companions. The language recommended for developing Functional Companions is Java.

This manual assumes that you are an experienced programmer and that you understand Oracle databases, the SQL and Java programming languages, and the principles of JDBC.

> **Note:** Be sure to check Section 1.1.3, "Prerequisite Experience for Developing Functional Companions" on page 1-5, which describes the Java development skills required for success with Functional Companions.

This manual also provides background and reference information on the CIO, which is needed by developers of applications having customized user interfaces that need access to the Oracle Configurator Active Model.

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. Standards will continue to evolve over time, and Oracle Corporation 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 additional information, visit the Oracle Accessibility Program Web site at `http://www.oracle.com/accessibility/`.

Accessibility of Code Examples in Documentation JAWS, a Windows screen reader, 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, JAWS 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 Corporation does not own or control. Oracle Corporation neither evaluates nor makes any representations regarding the accessibility of these Web sites.

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.

Functional Companion Basics

Provides the essentials for implementing any Functional Companions. Explains what Functional Companions are, and the different types available. Explains the relationship of Functional Companions and the CIO. Describes the basics of building a Functional Companion with Java. Describes how to incorporate a Functional Companion in your deployed Oracle Configurator.

Using Functional Companions

Collects instructions on how to use Functional Companions for specific tasks, such as controlling elements of a User Interface, handling User Interface events, and controlling windows.

Using the Configuration Interface Object (CIO)

Explains the Oracle Configuration Interface Object (CIO) and how to use it. Explains how to work with runtime configuration instances, and the nodes of the runtime Model; how to use logic transactions; how to validate configurations and handle contradictions; how to use requests; the effects of decimal quantities.

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.

Glossary of Terms and Acronyms

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:

- *Oracle Configurator Release Notes*
- *Oracle Configurator Implementation Guide*
- *Oracle Configurator Installation Guide*
- *Oracle Configurator Developer User's Guide*
- *Oracle Configurator Methodologies*
- *Oracle Configurator Performance 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 table below lists other conventions that 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 per cent 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:

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

Log into 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:

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

1

Functional Companion Basics

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

This chapter describes the essential techniques needed to create all Functional Companions.

Note: Be sure to check Section 1.1.3, "Prerequisite Experience for Developing Functional Companions" on page 1-5, which describes the Java development skills required for success with Functional Companions.

Note: Be sure to review the *Oracle Configurator Performance Guide* for information the performance impacts of Functional Companions.

1.1 What Are Functional Companions?

A Functional Companion is a programming object that you attach to your Model in order to extend the functionality of your runtime Oracle Configurator in ways that are not provided by Oracle Configurator Developer.

You write a Functional Companion object in the Java programming language. The Functional Companion communicates with your Model through an API (application programming interface) called the Configuration Interface Object (CIO). The Oracle Configuration Interface Object is also written in Java. See Chapter 3, "Using the Configuration Interface Object (CIO)".

You connect Functional Companions to specific nodes in your Model using Oracle Configurator Developer. You also specify the type of action that you want the specified Functional Companion to perform when your end users select its associated node. Then you generate the logic and user interface, as you normally do for your runtime Oracle Configurator. This generation action associates the Functional Companion with your application so that when your end users select a node in the Model, the Functional Companion on that node is automatically invoked.

1.1.1 Types of Functional Companions

You can assign a Functional Companion to perform any or all of the types of actions listed in Table 1–1 on page 1-2:

Table 1–1 Functional Companion Types

Type	Description
Auto-Configuration	Configures the state of the Model. You can use this to modify the shape of the Model tree, and the state of its nodes. For instance, your application might gather initial needs assessment information and use it to set up the appropriate set of choices for your end user to make. In your runtime Oracle Configurator, your end user will explicitly choose to run an Auto-Configuration Functional Companion. See Section 3.12.2, "The autoConfigure() Interface Method" on page 3-44.

Table 1–1 (Cont.) Functional Companion Types

Type	Description
Validation	Validates the logical choice that the end user has just made. The Functional Companion can perform complex operations beyond the scope of what you can develop in Oracle Configurator Developer. For instance, you can perform sophisticated numeric comparisons. A Functional Companion returns null if the validation is successful. If the validation fails, it returns a List of CompanionValidationFailure objects. In your runtime Oracle Configurator, all validation Functional Companions are run every time your end user chooses an Option. After each action, the end user gets the collection of strings returned by each Functional Companion that failed. Validation companions query the Model to determine validity, but should *not* modify the Model. Modifying the Model in a validation Functional Companion can cause unexpected application failures. See Section 3.12.3, "The validate() Interface Method" on page 3-45.
Output	Generates some form of output from the configuration. This output might be a report, a performance graph, a geometric rendering, or a graphical representation of the configuration. In your runtime Oracle Configurator, your end user will explicitly choose to run an output Functional Companion. See Section 3.12.4, "The generateOutput() Interface Method" on page 3-46.
Event-Driven	Automatically performs a specified action when a configuration is processed in certain ways (such as created or saved). See Section 3.4.8, "Automatic Behavior for Configurations" on page 3-14.

1.1.2 Important Facts About Functional Companions

- To build a Functional Companion, you implement an object class in Java. Oracle requires that Functional Companions be implemented in Java. Functional Companions can run on any Oracle platform that supports Java.

- When the runtime Oracle Configurator starts up, it creates an instance of the CIO, which creates runtime instances of all the mandatory Components in the Model. Then Oracle Configurator creates, for each instance of each Component, an instance of the class that you defined for your Functional Companion and associated with the Component in Oracle Configurator Developer; Oracle Configurator then attaches the Functional Companion instance to the Component.
- You can associate more than one Functional Companion with a particular Component; the CIO will create instances of all of them.
- If any Functional Companions cannot be loaded when you create a new configuration (for instance, due to internal errors or an incorrect CLASSPATH), the configuration will fail to open.
- In addition to Components, you can also associate Functional Companions with Models and BOM Models, which are considered equivalent entities. You can also associate Functional Companions with Connectors.
- In Java, you implement a class that extends `oracle.apps.cz.cio.FunctionalCompanion`. See Section 1.3, "Building Functional Companions" on page 1-7.

  You also implement one or more of the standard interface methods of `oracle.apps.cz.cio.IFunctionalCompanion`, which are described in Section 3.12, "Standard Interface Methods for Functional Companions" on page 3-41.

  For event-driven Functional Companions, you extend the class `oracle.apps.cz.cio.AutoFunctionalCompanion`, and implement one or more of its methods, which are described in Section 3.4.8, "Automatic Behavior for Configurations" on page 3-14.
- In order to communicate with your application's Model, the Functional Companion 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 1.2, "Functional Companions and the CIO" on page 1-5, and all of Chapter 3, "Using 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.

1.1.3 Prerequisite Experience for Developing Functional Companions

To effectively develop a Functional Companion, 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 Functional Companion.

In general, the Functional Companion 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
  - Exception handling
  - Using Java Interfaces
  - Java Event model (for writing Functional Companions that interact with the user interface)
  - HTML and the Java class HttpServletResponse (for writing Functional Companions that generate custom output)
- A working understanding of Oracle databases, including:
  - the PL/SQL programming language
  - the principles of JDBC

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

1.2 Functional Companions and the CIO

Functional Companions depend on the CIO for access to the Active Model. For more background, see Section 3.1.2, "The CIO and Functional Companions" on page 3-2.

1.2.1 Using the CIO Interface

Your Functional Companion is a client of the CIO. When you program against the CIO, you create instances of a set of public interface objects, which are defined in `oracle.apps.cz.cio`.

Your code should refer only to these public interface objects. See Section 3.2, "The CIO's Runtime Node Interfaces" on page 3-2.

1.2.2 Implementing Standard Interface Methods

You provide functionality for your Functional Companion by implementing body code for one or more of the methods listed below.

- For Functional Companions that extend `FunctionalCompanion`, implement:
  - `initialize()`

    You should also call `super.initialize()`
  - `autoConfigure()`
  - `validate()`
  - `generateOutput()`
  - `terminate()`

  These methods are described in Section 3.12, "Standard Interface Methods for Functional Companions" on page 3-41.
- For Functional Companions that extend `AutoFunctionalCompanion`, implement:
  - `oNew()`
  - `onSave()`
  - `onRestore()`
  - `afterSave()`
  - `onSummary()`

  These methods are described in Section 3.4.8, "Automatic Behavior for Configurations" on page 3-14.

For details and examples, see Section 1.3, "Building Functional Companions" on page 1-7.

1.3 Building Functional Companions

This section describes how to build a Functional Companion object class, and how to associate it with your configuration model so that it is used by the runtime Oracle Configurator.

1.3.1 Procedure for Building Functional Companions

Here is an overview of the tasks for Building Functional Companions in Java. See also Section 1.3.2, "Installation Requirements for Functional Companions" on page 1-9.

1. Use a Java development environment or text editor to create a `.java` file in which to define a Java class.
2. Import the classes from the CIO that your Functional Companion requires to do its work. See Chapter 3, "Using the Configuration Interface Object (CIO)"for background. Typical examples:

   ```
   import oracle.apps.cz.cio.FunctionalCompanion;
   import oracle.apps.cz.cio.IRuntimeNode;
   ```

3. Define a class in which to determine the behavior of your Functional Companion.

   ```
   public class MyClass extends FunctionalCompanion {
   ```

   When you define your Functional Companion class, you extend the base class for Functional Companions (`FunctionalCompanion`) and override only the particular methods that you need. In this case, you gain the functionality of the `FunctionalCompanion` base class. This functionality includes: saving references to the runtime node with which the Functional Companion is associated (with the `FunctionalCompanion.getRuntimeNode()` method), and returning the name of the Functional Companion (with the `FunctionalCompanion.getName()` method).
4. You may want to override `FunctionalCompanion.initialize()`. (See Section 3.12.1, "The initialize() Interface Method" on page 3-42.)

   Ordinarily, you should never directly call `FunctionalCompanion.initialize()`, since the CIO does that for you. However, if your Functional Companion overrides `FunctionalCompanion` as its base class, then the `initialize()` method of *your* class should call `super.initialize()`. This passes some necessary variables to the superclass

(`FunctionalCompanion`) so that its methods will work. This technique is illustrated by the following code fragment:

```
public void initialize(IRuntimeNode comp_node, String name, String
description, int id)
    {
    super.initialize(comp_node, name, description, id);
    this.comp_node = comp_node;
    }
```

5. Override one or more of the other interface methods of `FunctionalCompanion` (see Section 3.12, "Standard Interface Methods for Functional Companions" on page 3-41):

   ```
   initialize
   autoConfigure
   validate
   generateOutput
   terminate
   ```

6. Optionally, call the methods of the other interfaces of the CIO (see Section 3.2, "The CIO's Runtime Node Interfaces" on page 3-2).

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

   Use the correct version of the Sun JDK for your platform. See Section 1.3.2.1, "Installation Requirements for Developing Functional Companions" on page 1-9.

   If you use a class from the collections library, such as `List`, you must import it using this syntax:

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

8. Put the resulting `.class` file in the class path of the Oracle Configurator Servlet, or into a JAR file in that class path.

   See the *Oracle Configurator Installation Guide* for information on installing the OC Servlet.

9. Run Oracle Configurator Developer.

   Associate your Functional Companion with a Component in your Model. See Section 1.4, "Incorporating Functional Companions in Your Configurator" on page 1-13.

   Generate the Active Model and User Interface.

10. To test your Functional Companion, click the Test button in Oracle Configurator Developer.

    When the runtime Oracle Configurator starts up, click the buttons that have been generated in the UI for activating your Functional Companions. See Section 1.4.3, "Testing Functional Companions in the Runtime Oracle Configurator" on page 1-19.

1.3.2 Installation Requirements for Functional Companions

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

1.3.2.1 Installation Requirements for Developing Functional Companions

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

- The latest version of Oracle JDeveloper
- The latest version of JDK 1.3 for your platform

You do not need JDBC drivers or database access to compile a Functional Companion, although these are required to run one. The required driver classes are contained in the `apps.zip` file installed with Oracle Applications.

If you use a class from the collections library, such as `List`, you must import it using this syntax:

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

1.3.2.2 Installation Requirements for Compiling Functional Companions

In order to compile Functional Companions, you must have access to the classes in `apps.zip`. To get this access, you can:

- Copy `apps.zip` to your development machine.
- Map the location of `apps.zip` to a drive on your development machine. This requires NFS mounting.
- Perform your compilation on the server machine on which `apps.zip` is installed.

All of these methods require you to modify the Java class path on your development machine to include `apps.zip`. You must use the same `apps.zip` that is used by the Oracle Configurator Servlet. You can determine the location of `apps.zip` by examining the Apache configuration file `jserv.properties` for the setting of the OC Servlet's class path. For example:

```
wrapper.classpath=/d01/oracle/viscomn/java/apps.zip
```

The runtime Oracle Configurator automatically sets up a JDBC database connection for use by the CIO. Custom user interfaces that take the place of the runtime Oracle Configurator must perform this task. See Section 3.3, "Initializing the CIO" on page 3-4 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 Functional Companions 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*.

In order to compile Functional Companions, the files described in Table 1–2 on page 1-10 must be installed and recognized by your operating system environment in the appropriate locations.

Table 1–2 Required Software for Functional Companions

File	For Platform	Comment
Java Classes		
`apps.zip`	All	Includes all the Java classes required for the runtime Oracle Configurator, in addition to those for Oracle Applications.
`xmlparserv2.zip`	All	Includes the Java classes for the XML parser.
Shared Objects		Needed for CIO and Functional Companion access to the Oracle Configurator logic engine.
`czlce.dll`	Windows NT	Must be in the PATH system environment variable on the host machine on which the OC Servlet is installed. This should be set by the OC installation program.

Table 1–2 (Cont.) Required Software for Functional Companions

File	For Platform	Comment
`libczlce.so`	Solaris	Must be in the LD_LIBRARY_PATH environment variable parameter for the OC 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*.

1.3.2.3 Installation Requirements for Testing Java Functional Companions

If you have installed and set up the Oracle Configurator Servlet and Oracle Configurator Developer so that the Test button produces a DHTML test window, then this setup should also be correct for testing Functional Companions.

The classes that implement your Functional Companions must be installed in the Servlet directory used by the OC Servlet and added to the OC Servlet's class path. Otherwise, you are likely to get an error message like the following when you try to create a new configuration:

```
New Configuration: Cannot create configuration:
oracle.apps.cz.cio.FuncCompCreationException:
java.lang.ClassNotFoundException: classname
```

Where *classname* is the name of the first Functional Companion to be loaded.

1.3.3 Minimal Example of a Functional Companion

Example 1–1 provides a template for coding a Functional Companion that does not perform any work.

Example 1–1 Template for Functional Companion Code (MyClass.java)

```
import oracle.apps.cz.cio.*;
import com.sun.java.util.collections.List;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

/**
 * Template for Functional Companion code
 */
```

```
public class MyClass extends FunctionalCompanion {

  // constructor
  public MyClass() {
  }

  public void initialize(IRuntimeNode node, String name, String description, int
id) {
      super.initialize(node, name, description, id);
      // implement body, if necesary
  }

  // implement method, if necessary
  public void autoConfigure() throws LogicalException {
  }

  // implement method, if necessary
  public List validate() {
      return null;
  }

  // implement method, if necessary
  public void generateOutput(HttpServletResponse response) throws IOException {
  }

  public void terminate() {
      super.terminate();
      // implement body, if necesary
  }
}
```

Notes on Example 1–1

Regarding the following line in the example:

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

You must use this syntax if you import a class from the collections library, such as `List`.

Regarding the following line in the example:

```
public class MyClass extends FunctionalCompanion {
```

This class extends the base class for Functional Companions: `FunctionalCompanion`. See the explanation under Step 3. on page 1-7.

1.4 Incorporating Functional Companions in Your Configurator

To incorporate a Functional Companion into your configurator, you must:

1. Create a configuration rule that associates the Functional Companion with a Component (or equivalent node). See Section 1.4.1, "Associating Functional Companions with your Model" on page 1-13.
2. Update the User Interface to reflect the Functional Companion. See Section 1.4.2, "Updating the User Interface" on page 1-17.
3. Test the operation of the Functional Companion. See Section 1.4.3, "Testing Functional Companions in the Runtime Oracle Configurator" on page 1-19.

1.4.1 Associating Functional Companions with your Model

To enable your Functional Companion to work with your runtime Oracle Configurator, you must associate it with a Component or Connector (or equivalent node) in your Model. You create this association in Oracle Configurator Developer, as a type of configuration rule that specifies the Functional Companion method(s) that you have implemented in your executable Functional Companion, and the path to be used by the runtime Oracle Configurator to locate the file containing the executable Functional Companion.

To create an association between a Component and a Functional Companion:

1. Click on the **Rules** button on the main toolbar.

   A list of the configuration rule types appears in the lower-left pane.
2. Choose **New Functional Companion** from the Create menu. You can also highlight the **Functional Companions** node, click on the right mouse button, and select **New Functional Companion** from the popup menu.
3. In the **Name** field in the right pane, type a name for the Functional Companion rule.

   This name will be used to reference the rule when you create User Interface objects.
4. In the **Description** section, type a short explanation of the Functional Companion rule. If necessary, open the **Description** section by clicking on the blue arrow to the left of it.

5. If necessary, open the **Definition** section by clicking on the blue arrow to the left of it.

   In the Model view, select the node that you want to associate with the Functional Companion. Drag the node with the left-hand mouse button to the **Base Component** field in the Definition section. Only one Base Component may be specified per rule. You can choose the following types of Model nodes:

   - Component
   - Model
   - BOM Model
   - Connector

   The location of the node you choose determines the instantiation scope of the Functional Companion, based on the instance of the associated runtime component. However, it does not affect the execution scope of the Functional Companion class, which is global. Generally, you should choose a node as close to the root as possible.

6. Choose one or more types for the Functional Companion. The choices are listed in Table 1–3 on page 1-14:

Table 1–3 Functional Companion Types and Corresponding Methods

Type	Corresponding Functional Companion method
Validation	`validate()`
Auto-Configuration	`autoConfigure()`
Output	`generateOutput()`
Event-Driven	All methods of `AutoFunctionalCompanion`

See Section 1.1.1, "Types of Functional Companions" on page 1-2 and Section 3.12, "Standard Interface Methods for Functional Companions" on page 3-41 for background information.

Note: Note that you do not associate the `initialize()` and `terminate()` methods, since they are invoked automatically by the runtime Oracle Configurator

---

7. Under **Implementation**, drop down the list and chose the language in which the Functional Companion is implemented.

   - For all Functional Companions, choose "Java".

     (Java is the only implementation supported.)

8. Under **Program String**, type in the string that locates the executable Functional Companion:

   - For a Java implementation, the Program string is the name of the class that implements the Functional Companion, such as:

     ```
     FuncCompTest1
     ```

     The Program String must enable Oracle Configurator Developer and the OC Servlet to locate the class file, using your Java class path. The standard Java rules for specifying classes in archives apply. For example, if you had placed your Functional Companion `FuncCompTest1.class` file in a Java archive (JAR) file named `tests.jar`, using this path:

     ```
     com\java\tests\FuncCompTest1.class
     ```

     then you would specify the Functional Companion class this way as the **Program String** in Configurator Developer:

     ```
     com.java.tests.FuncCompTest1
     ```

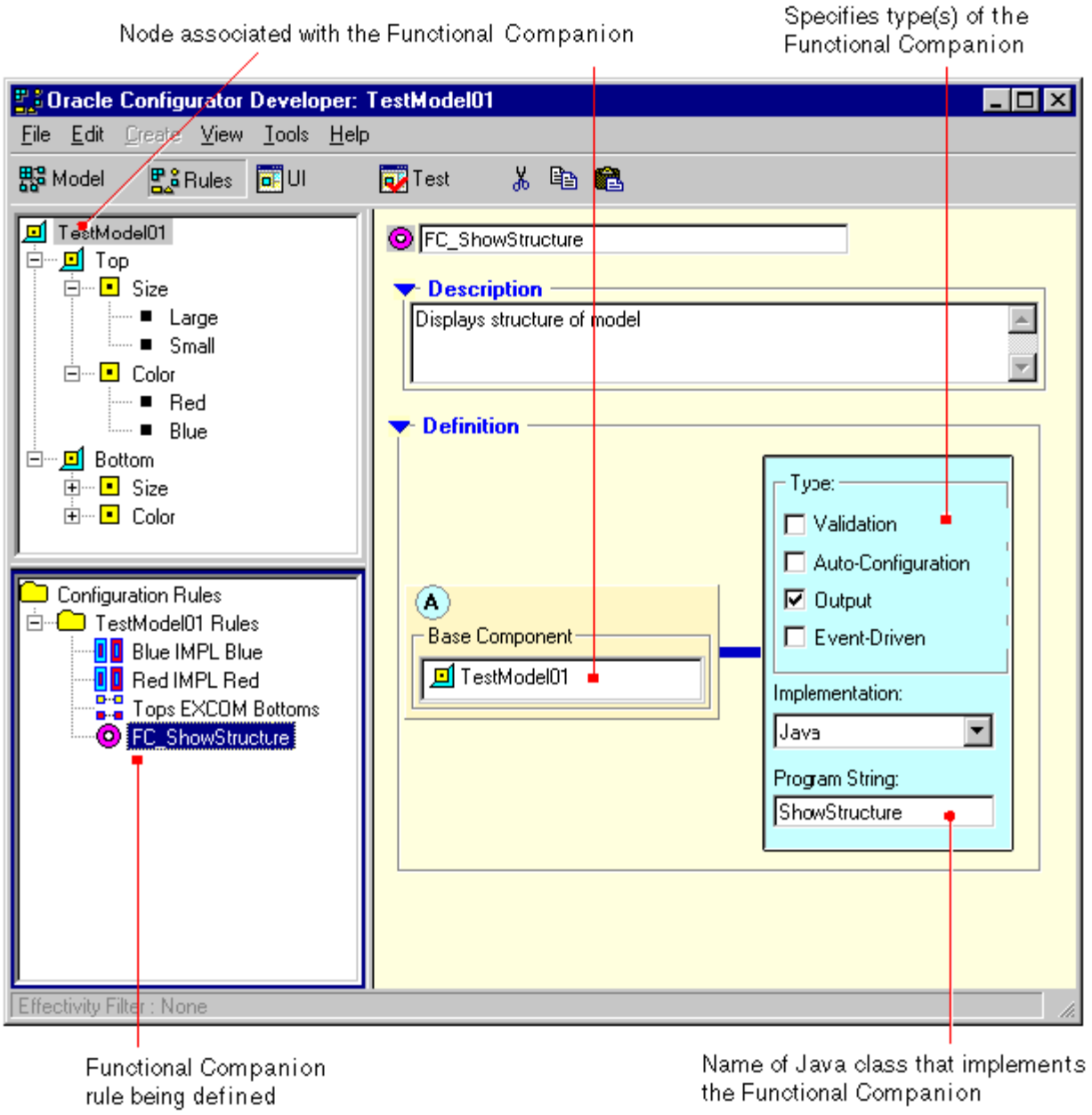
     See Step 8. on page 1-8 under Section 1.3.1, "Procedure for Building Functional Companions".

   Figure 1–1 on page 1-16 shows what the Rules module screen of Oracle Configurator Developer might look like after you associate a Component with a Functional Companion.

9. If you have recently modified your Model structure or Rules (not including creating a Functional Companion rule), then choose **Generate Active Model** from the **Tools** menu.

10. After the **Generate Active Model** command completes successfully, click on the **UI** button on the main toolbar to switch to the User Interface module.

Figure 1–1 Associating a Component with a Functional Companion



1.4.2 Updating the User Interface

After you create a Functional Companion rule that associates your executable Functional Companion with a Component or equivalent node in your Model, you must update the User Interface, so that it includes buttons for any Auto-Configuration or Output Functional Companions that you have defined.

Updating the User Interface consists of:

- Generating a new UI or refreshing an existing one. See Section 1.4.2.1 on page 1-17.
- Optionally, modifying the default UI that is generated by Oracle Configurator Developer. See Section 1.4.2.2 on page 1-17.

See the chapter on the User Interface in the *Oracle Configurator Developer User's Guide* for full information on these larger topics, especially the modification of UIs. The information presented here is minimal.

1.4.2.1 Generating or Refreshing a User Interface

To generate a User Interface:

1. If you have not already created a User Interface for your Model, select the root node ("User Interfaces") in the lower-left pane and choose **New User Interface** from the **Create** menu.

   You can generate any number of UIs, and choose the one to use in your application.

2. Wait for the notice that the creation of the User Interface was successful.

To refresh a User Interface:

1. If you have already created a User Interface for your Model, select the UI node for that UI and choose **Refresh** from the **Edit** menu.

   You can select any existing UI to refresh.

2. Wait for the notice that the refreshing of the User Interface was successful.

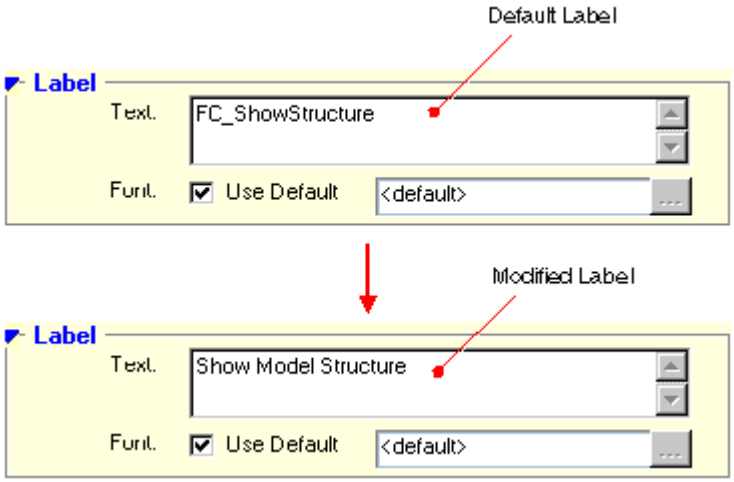
1.4.2.2 Modifying the Default User Interface

You can customize many elements of your User Interface in several ways. There are a few that are most relevant to Functional Companions.

- If your Functional Companion rule is of the Output or Auto-Configuration type, then a Button for running the Functional Companion is automatically

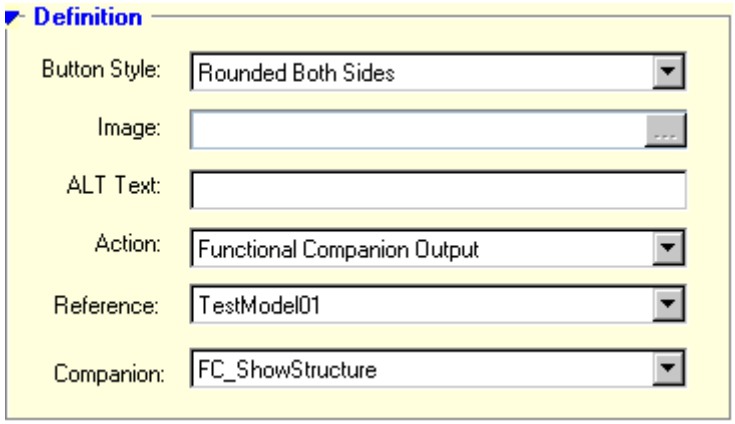
created as a node in the UI tree. The default label on the button is fabricated from the name of the Functional Companion rule. You will probably want to change this label to something that will give your end user a better idea of what the button will do. You do this by editing the **Text** field in the **Label** section of the attributes for the Button, as shown in Figure 1–2 on page 1-18

Figure 1–2 Changing the Label for a Functional Companion Button



- In the Definition section of the attributes for the Button, you may want to change one or more of the defaults shown in Figure 1–3 on page 1-19.

Figure 1–3 User Interface Definition for a Functional Companion Button



- The **Action** field is set to the type of Functional Companion. This type is Output or Auto-Configuration; the Validation and Event-Driven types do not generate a button. You can change the type if the implementation of your executable Functional Companion changes.
- The **Reference** field is set to the Model node that you associated with the Functional Companion rule. You can change it to attach the Button to a different Model node while keeping it on the same UI screen.
- The **Companion** field is set to the Functional Companion rule that the Button will run.You can change it, so that the same button triggers a different rule.

See the *Oracle Configurator Developer User's Guide* for details on these procedures.

1.4.3 Testing Functional Companions in the Runtime Oracle Configurator

After you generate the Active Model and UI, you can test your Functional Companions in the runtime Oracle Configurator.

1.4.3.1 Testing with Oracle Configurator Developer

To start the runtime Oracle Configurator from Oracle Configurator Developer, and test your Functional Companion:

1. Ensure that the Java class path for the OC Servlet includes the Functional Companion class file that you built. See Section 1.3.2.3, "Installation Requirements for Testing Java Functional Companions" on page 1-11.

2. If you have changed and recompiled the Functional Companion since the OC Servlet was last started, then you must restart the OC Servlet. Otherwise, the JVM running in the OC Servlet will not load the changed class file.
3. Choose **Tools** > **Options**, select the **Test** tab, then choose **Dynamic HTML in a browser** as your test environment.
4. In the **Servlet URL** field, enter the URL of your Oracle Configurator OC Servlet, using this syntax:

   `http://host:port/configurator/oracle.apps.cz.servlet.UiServlet`

   Where *host* and *port* are specific to the installation of the Oracle Configurator OC Servlet at your site.
5. Close the Options dialog.
6. Click the **Test** button.

   A web browser window opens, containing a DHTML rendering of the User Interface for your Model.
7. Click the button(s) that have been generated in the User Interface for your Functional Companion rules.

For more information, see the documentation indicated in Table 1–4.

Table 1–4 Documentation Related to Testing Functional Companions

For details on ...	See this section ..	In this document ...
Testing in Oracle Configurator Developer	Test/Debug	*Oracle Configurator Developer User's Guide*
Configuring items in runtime Oracle Configurator	Test/Debug	
Modifying buttons in the User Interface	The User Interface	
Setting InitServletURL in spx.ini	[Test]	*Oracle Configurator Implementation Guide*
Installing the OC Servlet	OC Servlet Considerations	*Oracle Configurator Installation Guide*
Setting up an HTML test page		*Oracle Configurator Installation Guide*

1.4.3.2 Testing with an HTML Test Page

You can test your runtime Oracle Configurator without having to run either Oracle Configurator Developer or your application. You can construct a simple HTML test page that takes the place of a host application. Opening the page in a web browser starts up the runtime Oracle Configurator in a predefined frame set, using the Model and User Interface that you specify inside that frame.

In order to use such a test page:

- You must provide an initialization message and its parameters in the page, as described in the *Oracle Configurator Implementation Guide*.
- You must have installed and started the Oracle Configurator OC Servlet, as described in the *Oracle Configurator Installation Guide*.

Examples of HTML test pages are provided in the *Oracle Configurator Implementation Guide* and the *Oracle Configurator Installation Guide*.

1.4.3.3 Test Functionality in the Runtime Oracle Configurator

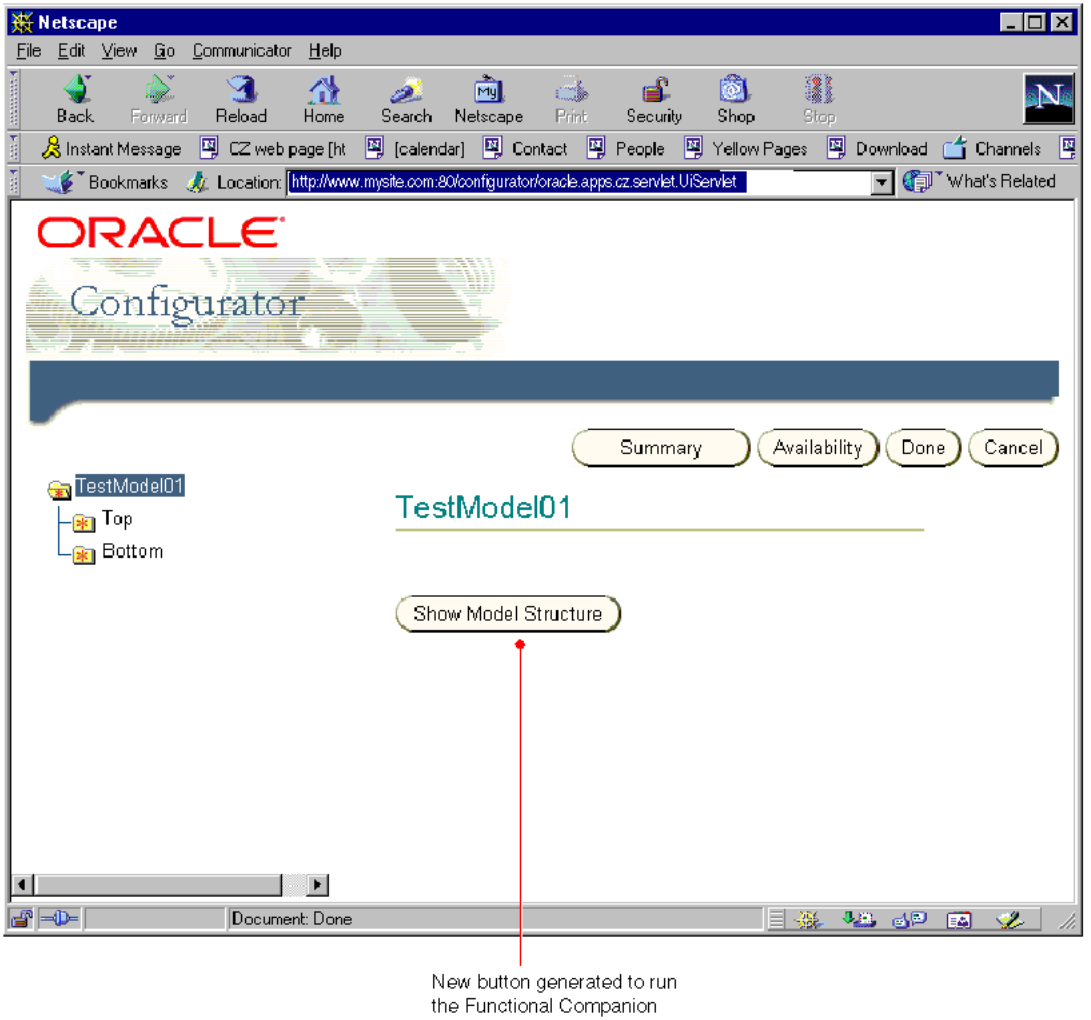
The User Interface for the runtime Oracle Configurator allows you to test your Functional Companions with the appropriate method listed in Table 1–5:

Table 1–5 Functional Companion Types and User Interface Features

Type	User Interface feature
Auto-Configuration	A button allows the user to run the `autoConfigure()` method.
Validation	The `validate()` method is called automatically when the user selects, deselcts, or changes the value of an option.
Output	A button allows the user to run the `generateOuput()` method.
Event-Driven	The actions specified in the overridden methods of `AutoFunctionalCompanion` are performed when a configuration is processed in the specified way (such as being created or saved).

Figure 1–4 on page 1-22 illustrates testing a Functional Companion in a runtime Oracle Configurator generated by the Test button in Oracle Configurator Developer. The Functional Companion illustrated is the one defined in the example in Section B.1, "Thin-Client generateOutput() Functional Companion" on page B-1.

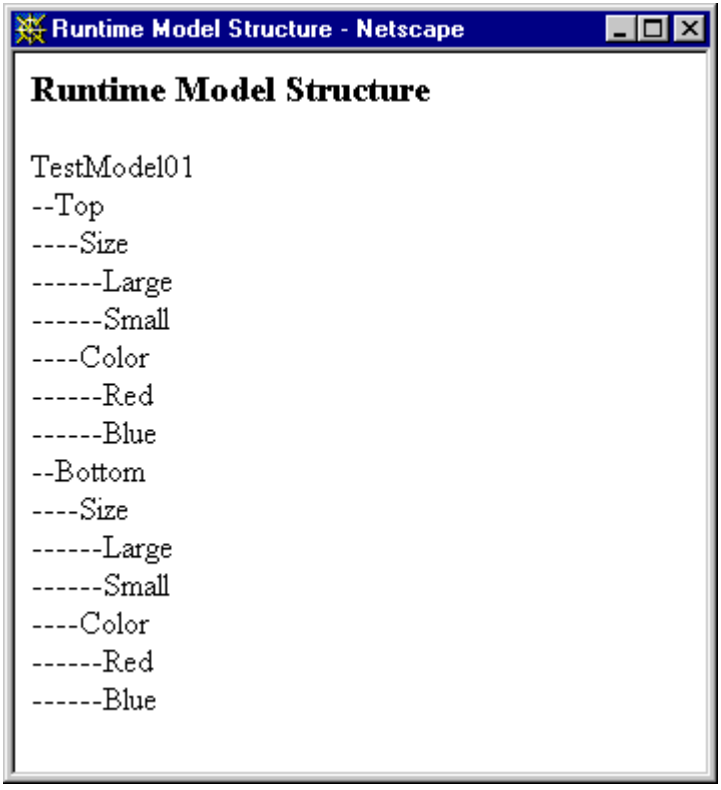
Figure 1–4 Testing a Functional Companion in the Runtime Oracle Configurator



- The Output Functional Companion button has been relabeled **Show Model Structure**, as suggested in Section 1.4.2.2, "Modifying the Default User Interface" on page 1-17.
- Clicking the Functional Companion button produces a window that displays the structure of the Model tree, beginning at the Base Component associated

with the Functional Companion rule. This output is shown in Figure 1–5 on page 1-23.

Figure 1–5 Output Window Generated by Functional Companion Button



- The user interface never contains a button for a Validation Functional Companion. The `validate()` method is run whenever there is a change in the value or state of an Option. If the change produces a validation failure, then the runtime Oracle Configurator displays a message, as described in the *Oracle Configurator Developer User's Guide*. The example Functional Companion shown here does not include Validation.
- The user interface never contains a button for an Event-Driven Functional Companion. The appropriate methods of `AutoFunctionalCompanion` are run when a configuration is processed in the specified way (such as being

created or saved). The example Functional Companion shown here is not Event-Driven.

- The user interface for an Auto-Configuration Functional Companion includes a button that modifies the configuration model in the specified way (for instance by changing the value of a numeric Feature). The example Functional Companion shown here does not include auto-configuration.

2

Using Functional Companions

This chapter describes some advanced techniques for using Functional Companions effectively in your application.

2.1 Controlling User Interface Elements

> **Note:** As you read this chapter, refer to the examples in Section B.4, "User Interface Interaction" on page B-8.

The CIO provides a set of Java interfaces for creating Functional Companions that interact with a Dynamic HTML user interface in a browser. Example 2–1 shows the hierarchy of these interfaces.

Example 2–1 Hierarchy of IUserInterface and Related Interfaces

```
IUserInterface
IUserInterfaceNode
    IUserInterfaceScreen
    IUserInterfaceControl
        IUserInterfaceImage
        IUserInterfaceLabel
IUserInterfaceEvent
IUserInterfaceEventListener
```

These interfaces provide the kinds of interaction listed in the following table:

Type of interaction	Examples
Access to UI session properties	Getting current screen or user information

Type of interaction	Examples
Access to UI controls and some of their properties	Change or hide the labels and images of controls such as images, captions, or combo boxes. (To hide or disable entire controls, see Section 2.2 on page 2-14 and Section 2.3 on page 2-16.)
Custom UI functionality callbacks from DHTML controls	Saving/restoring, navigation
Functional Companion callbacks	Callbacks for standard and custom events
Dynamic control over the navigation of UI screens being displayed	Navigation
Dynamic control over images or labels in the screens	Changing images, changing captions
Direct communication channels with the user	Displaying message boxes, generating custom HTML content

2.1.1 How the CIO Interacts with User Interfaces

The display of a user interface for Oracle Configurator is performed by the Oracle Configurator UI Server, which runs inside the Oracle Configurator Servlet.

As part of processing the initialization message for each configuration session, the UI Server is responsible for:

1. Creating or restoring a configuration (which results in creating a runtime tree view of the configuration Model).
2. Creating the UI nodes on the runtime tree view.
3. Connecting the UI nodes with the corresponding nodes in the runtime instance of the configuration Model provided by the CIO.

As part of creating or restoring a configuration, the UI Server also has to pass an argument to set a high-level UI interface reference, in the form of an IUserInterface object, such as `uiSession` in the following code fragment:

```
/*** UI SERVER code ***/

 Configuration cioConfig;
 IUserInterface uiSession;
 ...
 cioConfig = cio.createConfiguration(rootNodeId, uiSession);
 ...
```

All Functional Companions are able to get that reference back and use it to access other UI interfaces, or to register themselves to listen to UI events:

```
/*** FUNCTIONAL COMPANION example code ***/
 public void  initialize(...) {
   ...
   uiSession = cioConfig.getUserInterface();
   uiSession.addUserInterfaceEventListener(this);
   ...
 }
```

> **Note:** User interface interaction through `IUserInterface` and related interfaces is only available when your user interface uses Dynamic HTML in a browser. It is not available with Java Applet or custom-coded interfaces, or when using batch validation (which does not have a user interface).

Although a number of interfaces related to `IUserInterface` are provided, Functional Companions should only implement the interfaces `IUserInterfaceEventListener` and `IUserInterfaceNode`, as shown in the following code fragment:

```
/*** FUNCTIONAL COMPANION example code ***/

public class MyUiChanger extends FunctionalCompanion implements
IUserInterfaceEventListener, IUserInterfaceNode {
...
 public void  initialize(...) {
   ...
   uiSession = cioConfig.getUserInterface();
   uiSession.addUserInterfaceEventListener(this);
   ...
 }
}
```

The members of the other related interfaces may be called, but not implemented. See the fuller examples in Section B.4, "User Interface Interaction" on page B-8, which implement the required interfaces.

If Functional Companions need to perform some UI tasks, they have to register themselves in the Functional Companion's `initialize()` method to listen for activation events.

Such events are dispatched whenever a new UI tree branch has been built, such as when the UI tree is initially created, or when a component instance is added.

After this point, every user interaction causes a UI event that is dispatched through the `uiSession` object to all interested listeners (see Section 2.1.9, "Handling Interface Events" on page 2-8 for details).

As your Functional Companion code makes requests to manipulate the user interface (such as changing an image, or navigating to another screen), the UI Server caches the requests and keeps track of the current requested state of the UI.

Since it is possible that more than one Functional Companion is manipulating the user interface at the same time, the UI Server does not immediately render the results of your UI requests. The CIO checks the list of Functional Companions that are interacting with the UI; when all the Functional Companions are done with their UI requests, the results from the last Functional Companion are used, and the cached requests are loaded into a final result set. Finally, the result set is rendered in the client browser.

Note: It is not possible to guarantee the order in which multiple Functional Companions execute. Consequently, it is not possible to predict in advance which UI interaction will be rendered.

2.1.2 Getting the Current Screen

Having a reference to `IUserInterface` allows a Functional Companion to get access to various properties of the `uiSession` object, such as style, session ID, or user account information.

```
/*** FUNCTIONAL COMPANION example code ***/
 ...
 style = uiSession.getType();
 sessionID = uiSession.getUiSessionID();
 ...
```

Also available on the `IUserInterface` reference is the current display UI screen and UI nodes from the UI runtime tree

```
 /*** FUNCTIONAL COMPANION example code ***/
 ...
 IUserInterfaceScreen  uiScreen = uiSession.getCurrentScreen();
 List                  uiCombos = uiSession.getNode("Feature-321",uiScreen);
```

```
// Now walk the list of nodes and check the types, to get the target node
...
IUserInterfaceControl  uiCombo321 = uiCombos.get(targetNode);
 ...
```

Note that because there can be many UI nodes having the same name, `uiSession.getNode()` returns a List. All nodes in such a list will share the same name. You must walk the list and determine which node has the desired type, then get it.

2.1.3 Access to Nodes

UI node-specific properties are accessible through the `IUserInterfaceNode` interface:

```
/*** FUNCTIONAL COMPANION example code ***/
 ...
 IUserInterfaceNode  myNode;
 ...
int type = myNode.getType();
 String name = myNode.getName();
 String caption = myNode.getUiCaption();
 ...
```

You can check the type of the node, using the types provided by the `IUserInterfaceNode` interface:

```
...
if (myNode.getType() == (IUserInterfaceNode.COMBO_BOX))
...
```

See the documentation for `IUserInterfaceNode` for a list of the types.

You can also use `IRuntime.getRuntimeNode()` to obtain the runtime node, if any, that is associated with a UI node.

2.1.4 Access to Screens and Controls

Any one of the sub-interfaces (such as `IUserInterfaceScreen` or `IUserInterfaceControl`) has UI-accessible properties:

```
  /*** FUNCTIONAL COMPANION example code ***/
 ...
 IUserInterfaceControl  uiCombo321;
 IUserInterfaceScreen  uiScreen = uiCombo.getScreen();
```

```
List                controls = uiScreen.getControls();
...
```

2.1.5 Access to Images and Labels

Having a reference through `IUserInterfaceImage` allows access to the filename of the displayed image. An `IUserInterfaceLabel` reference can be used to access to the UI caption of a label (the method `getUiCaption()` is available on all UI nodes):

```
/*** FUNCTIONAL COMPANION example code ***/
...
IUserInterfaceImage    image;
String filename = image.getFileName();
...
IUserInterfaceLabel    label;
String caption = label.getUiCaption();
...
```

2.1.6 Controlling Navigation

An important use of UI control through the CIO is the ability to navigate to different screens. It allows a Functional Companion to control the sequence of screens that your end user goes through, based on their selections. Based on preliminary screening selections, you can allow your end user to skip over screens that contain irrelevant choices.

Having a reference through IUserInterface allows you to navigate to any screen for which there is a given screen name. For this purpose the `uiSession` object will try to find a screen with the given name and navigate to it:

```
/*** FUNCTIONAL COMPANION example code ***/
...
uiSession.navigateToScreen("Screen322");
...
```

Note that the UI Server now regards "Screen322" as the "current screen", though it may not have rendered its display yet. See Section 2.1.1, "How the CIO Interacts with User Interfaces" on page 2-2.

But this navigation is more convenient if there is a reference to the screen:

```
/*** FUNCTIONAL COMPANION example code ***/
```

```
 ...
 IUserInterfaceControl  uiCombo321;
 IUserInterfaceScreen  uiScreen = uiCombo.getScreen();
 uiScreen.navigate();
 ...
```

2.1.7 Controlling Images and Labels

Being able to change the content of an image or label allows Functional Companions to control the way the end user sees the configured product based on their selections. For example, end users may want to see a different-colored image when they select from a combo box asking "Which color do you prefer?"

In order to be able to change the content of an image dynamically, it is necessary to have a reference to an image UI node, through `IUserInterfaceImage`:

```
/*** FUNCTIONAL COMPANION example code ***/
 ...
 IUserInterfaceScreen  uiScreen = uiSession.getCurrentScreen();
 List                  uiImages = uiSession.getNode("Image-322",uiScreen);
 IuserInterfaceImage    uiImage322 = uiImages.get(0);
...
 uiImage322.setFileName("Image322-1.jpg");
...
```

For a full example, including deployment considerations, see Section B.4.3, "Changing an Image" on page B-13.

Similarly, the content of a label can be dynamically changed through a reference to a `IUserInterfaceLabel`:

```
/*** FUNCTIONAL COMPANION example code ***/
 ...
 IUserInterfaceScreen  uiScreen = uiSession.getCurrentScreen();
 List                  uiLabels = uiSession.getNode("Label-323",uiScreen);
 IuserInterfaceLabel    uiLabel323 = uiLabels.get(0);
...
 uiLabel323.setUiCaption("ABC");
...
uiLabel323.setUiCaption("DEF");
 ...
```

2.1.8 Generating Messages and Other Output

A useful capability is being able to control the result of a user click into a different frame. This allows you to enhance the look and feel of Oracle Configurator by customizing the UI into separate frames along with other information that you think might be useful to the user.

2.1.8.1 Message Boxes

In order to be able to create simple message boxes, the Functional Companion needs a IUserInterface reference:

```
 /*** FUNCTIONAL COMPANION example code ***/
 ...
 IUserInterface uiSession;
 ...
uiSession.addMessageBox(0,
                    "We don't have this in stock.",
                    "Warning");
...
```

2.1.8.2 Custom HTML Output

If your Functional Companion needs to control the look of a separate frame you can output the custom HTML output directly, using the `IUserInterface` reference:

```
 /*** FUNCTIONAL COMPANION example code ***/
 ...
 IUserInterface uiSession;
 ...
 uiSession.outputToFrame("Root:Frame1",
                    "<HTML> ... </HTML>");
...
```

Note that the complex HTML output required for this approach is not illustrated in the code fragment shown here.

2.1.9 Handling Interface Events

It is effective to trigger your Functional Companion in response to events taking place in the user interface.

The examples in Section B.4, "User Interface Interaction" on page B-8 all use event handling, by implementing `IUserInterfaceEventListener`.

2.1.9.1 Types of Events

Any kind of user action on the client machine is recorded and encapsulated into a UI event that the UI Server receives and dispatches to the "interested" Functional Companions.

The events that are recognized (such as POST_SAVE_CONFIGURATION or POST_CLICK) are fields of IUserInterfaceEvent.

2.1.9.2 Listener Registration

Any Functional Companion object can be notified of a UI event, by implementing the interface `IUserInterfaceEventListener` and registering itself as an event listener.

Note that listening is available for the Functional Companion as a whole, but not on individual UI nodes.

The following code fragment illustrates the registration of a UI event listener:

```
...
IUserInterface uiSession;
...
public void initialize(IRuntimeNode node, String name, String descrip, int id) {
...
  uiSession = this.getRuntimeNode().getConfiguration().getUserInterface();
  uiSession.addUserInterfaceEventListener(IUserInterfaceEvent.POST_CLICK, this);
}
...
```

See Section B.4, "User Interface Interaction" for fuller examples in context, such as Example B–4 on page B-9.

Event listeners registered on the `IUserInterface` reference can listen to events for all UI nodes.

The method `addUserInterfaceEventListener()` is available on the `IUserInterface` interface and adds a listener to the specified dispatcher.

The complementary method, `deleteUserInterfaceEventListener()` is also available, to unregister a listener in order to stop handling events.

Whenever an event is dispatched, each listener's `handleUserInterfaceEvent()` method is called with the event object as an argument.

Note: It is not possible to guarantee the order in which multiple Functional Companions execute. Consequently, it is not possible to predict in advance the order in which the listeners will be invoked.

Any number of listeners can be registered at any point to listen to events.

If more than one listener is registered, then the Functional Companion can only differentiate between different events by getting the types of the events. The method to use is `IUserInterfaceEvent.getName()`, which returns the type of the event. This is shown in the following code fragment.

```
...
public void handleUserInterfaceEvent(IUserInterfaceEvent event) {
...
String eventType = event.getName();
...
}
...
```

An event listener can also be registered to listen only to specific types of events, by specifying an event ID as well as a listener. This prevents all other types of events from being dispatched to this listener. For example, if a Functional Companion needs to listen only to POST_CLICK events, then it has to register itself as shown in the following code fragment.

```
...
uiSession.addUserInterfaceEventListener(IUserInterfaceEvent.POST_CLICK, this);
...
```

The events that are recognized represented as fields of `IUserInterfaceEvent`. See Section 2.1.9.1, "Types of Events" on page 2-9.

2.1.9.3 Events in a Configuration Network

The ability to update a network of installed configured components introduces the following events:

- ON_NAVIGATE
- POST_ACTIVATE_ACTION

Handling these events requires you to register a listener, as described in Section 2.1.9.2 on page 2-9.

For background on updating networks of installed configured components, see the documentation for the Telecommunications Services Ordering solution.

ON_NAVIGATE

Handling the ON_NAVIGATE event enables you to alter the UI screen that is about to be presented to the end user.

You should consider the situations under which a screen may be navigated to. For example, if the screen the end user navigates to might be a read-only component, then you may want to render the UI controls in a manner that is different from that for an editable component.

The UI Server notifies ON_NAVIGATE event listeners when a screen navigation is about to occur. Here are details about that notification:

- If the configuration session is being started, the UI Server notifies ON_NAVIGATE event listeners after the POST_RESTORE_CONFIGURATION and POST_NEW_CONFIGURATION events have been processed. After the listeners for ON_NAVIGATE have been invoked, any changes made by your Functional Companion will be displayed when the new UI screen is rendered by a navigational screen change.
- During the configuration session, the UI Server notifies ON_NAVIGATE event listeners after all other event listeners are processed, and if there is a navigational screen change to be made. Screen changes can be caused when an end user clicks a navigation button, or by navigation requested by a Functional Companion.
- If the configuration session is being terminated, the UI Server notifies ON_NAVIGATE event listeners under one of the following scenarios:

  First scenario:

  1. After the PRE_SAVE_CONFIGURATION events have been processed.
  2. If your Functional Companion listening for PRE_SAVE_CONFIGURATION events aborts the "save" operation by throwing a FuncCompMessageException.
  3. If there is a navigational screen change to be made.

  Second scenario:

  1. After the PRE_CANCEL_CONFIGURATION events have been processed.

2. If your Functional Companion listening for PRE_SAVE_CONFIGURATION events aborts the "cancel" operation by throwing a FuncCompMessageException.
3. If there is a navigational screen change to be made.

After either of these scenarios, the changes are added to the UI result set, so that they can be rendered.

Note that the UI Server does *not* notify ON_NAVIGATE event listeners after a POST_SAVE_CONFIGURATION event.

You must distinguish the purpose of ON_NAVIGATE from that of POST_NAVIGATION_EVENT_EXECUTION. The ON_NAVIGATE event enables you to customize the screen that is about to be displayed, while POST_NAVIGATION_EVENT_EXECUTION enables you to validate the screen that the end user is leaving, or to choose which screen to navigate to next. The POST_NAVIGATION_EVENT_EXECUTION event can be handled for the following purposes:

- To cancel navigation if certain UI controls do not have any values (for example, a text field that should contain the name of the component but does not)
- To cancel navigation if certain UI controls do not have the desired values, or if the format of the values is not correct
- To decide which screen to navigate to based on the current state of the model

POST_ACTIVATE_ACTION

Handling the POST_ACTIVATE_ACTION event enables you to change the display state of controls on the screen that is current displayed.

The UI Server notifies POST_ACTIVATE_ACTION event listeners when either of these events happens:

- The end user clicks an **Activate Instance** UI control
- A read-only component is activated through a Functional Companion *and* that component is currently displayed

Here are some details about that notification of POST_ACTIVATE_ACTION event listeners:

- During the configuration session, the UI Server notifies POST_ACTIVATE_ACTION listeners under either of these conditions:
  - After all other event listeners are processed, but before the ON_NAVIGATE listeners

  - If the UI Server is currently processing the event that activates a read-only component
- If the configuration session is being terminated, the UI Server does not notify POST_ACTIVATE_ACTION listeners.

2.1.10 Testing for a User Interface

It is possible to use a Functional Companion in conjunction with batch validation, which invokes the runtime Oracle Configurator without a UI. Batch validation, which is described in the *Oracle Configurator Implementation Guide*, is used by Oracle Applications for certain tasks.

> **Warning: When working with `IUserInterface` and related CIO interfaces, you should always check for the existence of a user interface. Not doing so may result in a NullPointerException in certain circumstances, such as during batch validation.**

Invoking the runtime Oracle Configurator with a UI results in a situation in which the `getUserInterface()` method returns a null object. The symptom of this null return object is a `NullPointerException`, the reporting of a `FuncCompCreationException` in the OC Servlet logs, and a configuration validation error in the UI. The following code fragment demonstrates how to check for the existence of a user interface in the `initialize()` method of a Functional Companion:

```
...
IUserInterface uiSession;
...
public void initialize(IRuntimeNode node, String name, String descrip, int id) {
...
  uiSession = this.getRuntimeNode().getConfiguration().getUserInterface();
  if (uiSession != null) {
    uiSession.addUserInterfaceEventListener(IUserInterfaceEvent.POST_CLICK,
                                            this);
...
    }
}
...
```

2.2 Hiding User Interface Controls

By default, all controls are visible in a DHTML user interface that is generated with Oracle Configurator Developer.

You can hide a specified control by setting its **hidden flag**, using the following methods of the interface `IUserInterfaceControl` that are listed in Table 2–1.

Table 2–1 Methods for Hiding User Interface Controls

Method	Description
`setHiddenFlag(boolean bFlag)`	Sets the hidden flag for the UI control. Set `bFlag` to TRUE to hide the control. A runtime exception is thrown if the operation is terminated for some reason.
`getHiddenFlag()`	Return the current value of the hidden flag for the control.
`isVisible()`	Returns the current visibility state of the control, taking into account the availability of the node corresponding to the control. Returns TRUE if the control is to be rendered as visible.

You cannot control the visibility of controls in the Java applet user interface.

2.2.1 Principles of Hiding User Interface Controls

Whenever there is a screen refresh or a navigation event, new UI controls are created, corresponding to the nodes in the User Interface for the Model that is being configured. When controls are created, the UI Server checks the availability state of each node in the Model that needs to be displayed. If a node is available, the hidden flag for its control is processed.

The availability for a node is determined by:

- The Effectivity attribute of the nodes of the Model
- The Visibility attribute of the nodes of the Model
- The User Interface attributes that govern dynamic visibility (such as **Hide unselectable Controls and Options**)

All of these attributes are controlled with Configurator Developer. See the *Oracle Configurator Developer User's Guide* for details.

Table 2–2 summarizes the interaction between the availability of a node and the hidden flag setting for the corresponding UI control. A control must be available in order for the hidden flag to take effect.

Table 2–2 Factors Affecting the Visibility of a Control

Availability	Hidden Flag[1]	Resulting Visibility of Control
True	True	Hidden
True	False	Visible
False	True	Hidden
False	False	Hidden

[1] As set by `setHiddenFlag()`.

Unsatisfied Rules and Visibility

In the runtime Oracle Configurator, the visibility of the graphic that designates an unsatisfied selection depends on the visibility of the associated control. If you have hidden a control (or a screen) for a selection that is unsatisfied, then its unsatisfied rule message is not displayed. To alert the end user that a rule is unsatisfied, the parent node of the hidden unsatisfied control in the Navigation tree is marked as unsatisfied. In addition, a warning message is displayed when the end user terminates the session by clicking the Done button.

User Interface Layout and Visibility

In the runtime Oracle Configurator, the positioning of screen elements is not changed by the hiding or showing of controls. If you hide a control, then an empty space remains on the screen in its former location; adjacent controls are not repositioned to fill in the space.

2.2.2 Example of Hiding User Interface Controls

Example 2–2 shows a fragment of Functional Companion code that handles a click event from a button named `hide` and uses it to hide a label named `HideMe`.

Example 2–2 Controlling User Interface Visibility

```
...
public void initialize(IRuntimeNode node, String name, String description, int id) {
    super.initialize(node, name, description, id);

    mUi = this.getRuntimeNode().getConfiguration().getUserInterface();
```

```
    mUi.addUserInterfaceEventListener(IUserInterfaceEvent.POST_CLICK, this);
}


public void handleUserInterfaceEvent(IUserInterfaceEvent event) {
    if (event.getUiNode() != null ) {
        IUserInterfaceScreen scr = (IUserInterfaceScreen)mUi.getCurrentScreen();
        List ctls = scr.getControls();
        int len = ctls.size();
        // Find a button named "hide"
        if ((event.getUiNode().getType() == IUserInterfaceNode.BUTTON) && (
event.getUiNode().getName().equals("hide"))){
            for ( int i = 0; i<len; i++ ){
                IUserInterfaceControl ctl = (IUserInterfaceControl)ctls.get(i);
                // Find a label named "HideMe"
                if ( ctl.getType() == IUserInterfaceNode.LABEL && ctl.getName().equals("HideMe")
){
                    System.err.println("Found the label to hide");
                    try{
                        // Hide the control
                        ctl.setHiddenFlag(true);
                    }catch(Exception e){
                        System.err.println("The following controls were not found.")
                            }
                    return;
                }
            }
        }
    }
}
...
```

2.3 Disabling User Interface Controls

By default, all controls are enabled in a DHTML user interface that is generated with Oracle Configurator Developer.

You can disable a specified control by setting its **disabled flag**, using the following methods of the interface `IUserInterfaceControl` that are listed in Table 2–3. Disabling a control makes it read-only. Enabling a control allows it to be used.

Table 2–3 Methods for Disabling User Interface Controls

Method	Description
`setDisabledFlag(boolean bFlag)`	Sets the disabled flag for the UI control. Set `bFlag` to TRUE to disable the control.
`getDisabledFlag()`	Return the current value of the disabled flag for the control.
`isEnabled()`	Returns the current disabled state of the control, taking into whether the control is on the active screen. Returns TRUE if the control is to be rendered as enabled.

You cannot disable individual controls in the Java applet user interface. However, you can open the Java applet in a read-only state by passing `true` as the value of the `read_only` parameter in the initialization message.

Table 2–2 summarizes the interaction between the availability of a node and the disabled flag setting for the corresponding UI control.

A control must be on the screen that is currently active in order for the disabled flag to take effect. You can disable an entire screen, since a screen itself is also a UI control.

Table 2–4 Factors Affecting the Enabled State of a Control

Active Screen	Disabled Flag[1]	Resulting Disabled State of Control
True	True	True
False	True	True
True	False	False
False	False	True

[1] As set by `setDisabledFlag()`.

2.3.1 Example of Disabling User Interface Controls

Example 2–3 shows a fragment of Functional Companion code that handles a click event from a button named `disable` and uses it to hide a label named `DisableMe`.

Example 2–3 Disabling a User Interface Control

```
...
```

```
public void initialize(IRuntimeNode node, String name, String description, int id) {
    super.initialize(node, name, description, id);

    mUi = this.getRuntimeNode().getConfiguration().getUserInterface();
    mUi.addUserInterfaceEventListener(IUserInterfaceEvent.POST_CLICK, this);
}


public void handleUserInterfaceEvent(IUserInterfaceEvent event) {
    if (event.getUiNode() != null ) {
        IUserInterfaceScreen scr = (IUserInterfaceScreen)mUi.getCurrentScreen();
        List ctls = scr.getControls();
        int len = ctls.size();
        // Find a button named "disable"
        if ((event.getUiNode().getType() == IUserInterfaceNode.BUTTON) && (
event.getUiNode().getName().equals("disable"))){
            for ( int i = 0; i<len; i++ ){
                IUserInterfaceControl ctl = (IUserInterfaceControl)ctls.get(i);
                // Find a label named "DisableMe"
                if ( ctl.getType() == IUserInterfaceNode.LABEL &&
ctl.getName().equals("DisableMe") ){
                    System.err.println("Found the Label to disable");
                    try{
                        // Disable the control
                        ctl.setDisableFlag(true);
                    }catch(Exception e){
                        System.err.println("The following screens were not found.")
                            }
                    return;
                }
            }
        }
    }
}
...
```

2.4 Controlling Parent and Child Windows

Your application may require the ability to launch a child custom user interface from the Oracle Configurator DHTML 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 Oracle Configurator DHTML interface.

Implementing the solution could require the features described in these sections:

- Section 2.4.1, "Locking, Unlocking and Refreshing Windows" on page 2-19
- Section 2.4.2, "Accessing a Shared Configuration Model" on page 2-19

2.4.1 Locking, Unlocking and Refreshing Windows

During the period of user interaction with the child UI window, you would probably wish to 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.

You can use methods available through the Source Frame of the Oracle Configurator DHTML window to control the child and parent windows. The following table lists these methods:

Method	Action
`lockEventManagers()`	Locks the parent window while the child window is in use
`unlockEventManagers()`	Unlocks the parent window when the child window is closed
`refreshFrames()`	Refreshes the parent window with the results of changes made in the child window

The use of these methods is illustrated in Section B.5.1, "Locking, Unlocking and Refreshing Windows" on page B-16.

2.4.2 Accessing a Shared Configuration Model

If your application opens a child window in order to perform interaction with the Model that is not available in the Oracle Configurator DHTML window, other servlets that you may use in the same session should have access to the state of the configuration Model.

The example in Section B.5.2, "Accessing a Shared Configuration Model" on page B-17 illustrates how it is possible, during the initialization of a configuration session with the Oracle Configurator Servlet, to store the Configuration object under a new key in the HTTP session.

2.5 Filtering for Connectivity

You can define a Connection Filter Functional Companion 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.

2.5.1 Defining a Connection Filter Functional Companion

To define a Connection Filter Functional Companion:

1. Define a class that extends FunctionalCompanion.

   See Section 2.5.3, "Example of a Connection Filter Functional Companion" on page 2-21.

2. Override `FunctionalCompanion.validateEligibleTarget()`.

   In the body of your implementation of `validateEligibleTarget()`, define the test that filters potential target instances for the Connection.

3. In Oracle Configurator Developer, associate your Functional Companion with 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.

4. Select choose **Event-Driven** as the Type of the Functional Companion.

   See Section 1.4, "Incorporating Functional Companions in Your Configurator" on page 1-13 for background.

2.5.2 Behavior of Connection Filter Functional Companions

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 Functional Companions associated with these instances. If one of these Functional Companions has implemented `validateEligibleTarget()`, then its filtering test is executed. If the target instance fails the test, 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 Functional Companion 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 Functional Companions can be associated with the same Connector. Example:
  - Model_A includes Connector_A
  - In Model_A, Functional Companion FC_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, Functional Companion FC_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 FC_1 and FC_2.

2.5.3 Example of a Connection Filter Functional Companion

For an example of a Connection Filter Functional Companion, see Example 2–4 on page 2-21. This Functional Companion 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 Functional Companion 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 2–4 Filtering for Connectivity (FilterTarget.java)

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

public class FilterTarget extends FunctionalCompanion {

  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;
      }
    }
  }
```

3

Using the Configuration Interface Object (CIO)

This chapter explains the Oracle Configuration Interface Object (CIO) and how to use it.

3.1 Background

This section describes what the CIO is, and its relationship to Functional Companions.

3.1.1 What is the CIO?

The Configuration Interface Object (CIO) is an API (application programming interface) that provides your 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 also used by Functional Companions. See Section 1.2, "Functional Companions and the CIO" on page 1-5. Be sure to review the *Oracle Configurator Performance Guide* for information the performance impacts of Functional Companions.

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.

The Oracle Configuration Interface Object is written in Java, and implemented as a Java package `oracle.apps.cz.cio`, from which you will need to import classes.

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

3.1.2 The CIO and Functional Companions

A Functional Companion is Java code that calls the CIO.

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

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

3.2 The CIO's Runtime Node Interfaces

When you program against the CIO, you only create instances of the classes `CIO` (see Section 3.3, "Initializing the CIO" on page 3-4) and `Configuration` (see Section 3.4, "Access to Configurations" on page 3-5). You then use the public interfaces of the CIO, such as those listed in Table 3–1 on page 3-3, 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.

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

These interfaces are all defined in the Java package `oracle.apps.cz.cio`.

Table 3–1 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.
`IUserInterface`	Implemented by applications that want to provide access to the user interface in Functional Companions. There are related interfaces, described in Chapter 2, "Using Functional Companions".

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 will remain constant.

3.3 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.

If you are using the CIO in a custom user interface, you will have to initialize the CIO.

> **Note:** When you run Functional Companions through the runtime Oracle Configurator (or test them by using the **Test** button in Oracle Configurator Developer), this initialization and connection work is automatically handled for you by the application; you do not have to write your own code 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, load *one* of the following:

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

3. Create a context object and pass to it the information needed to make a database connection: the database URL, the user ID and password of the current user, and the owner of the database. 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 ("webServerHostname", "portNumber",
"dbcFileName");
```

4. Create a CIO object.

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

Example 3–1 on page 3-7 shows how some of these steps are employed.

3.4 Access to Configurations

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 3–2:

Table 3–2 Typical Methods of the Configuration Object

To obtain this ...	Use this method of `Configuration` ...
Access to the CIO object that contains the Configuration object	`getCIO()`
Access to the component object for which the Configuration object represents a configuration	`getRootComponent()`
Access to a collection of current validation failures	`getValidationFailures()`
An indication of whether the complete configuration is satisfied	`isUnsatisfied()` `getUnsatisfiedItems()`
A collection of the selected nodes in the configuration	`getSelectedItems()`

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 Section 3.7, "Logic Transactions" on page 3-28.

3.4.1 Creating Configurations

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 will not be able to 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 3.3 on page 3-4 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 3–1 on page 3-7.

- 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 Oracle Configurator Servlet. The correspondences are shown in Table 3–3 on page 3-7. See the *Oracle Configurator Implementation Guide* for more information on the initialization parameters.

Table 3–3 Correspondence of Configuration 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 3–1 shows a technique for creating a Configuration object. For clarity, it omits some important tasks, such as using transactions and fully handling exceptions.

Example 3–1 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 webServerHostname = "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( webServerHostname, 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
        }
    }
}
```

3.4.2 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.

3.4.3 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 3.4.5 on page 3-11.

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.

Note: Do not save a Configuration object during a logic transaction (see Section 3.7, "Logic Transactions" on page 3-28). You may miss some validation messages that are not available until the transaction is committed.

3.4.4 Monitoring Changes to Configurations

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

3.4.4.1 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 Functional Companions 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 Functional Companion 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 3.4.4.2, "How You Can Monitor Changes to Configurations" on page 3-11.

3.4.4.2 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 Functional Companion to create and rename of an instance of an optional 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.

3.4.5 Restoring Configurations

You restore a configuration in order to operate on if it has been saved and closed (as described in Section 3.4.3, "Saving Configurations" on page 3-9).

To restore a Configuration object from the Oracle Configurator schema, use `CIO.startConfiguration()`. For details about that method, see Section 3.4.1 on page 3-5 and Example 3–1 on page 3-7.

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 will not be able to use any new functionality.

When you restore a configuration, any user requests (see Section 3.10.2, "User Requests" on page 3-37) that cannot be applied are reported as validation failures. See Section 3.10.4, "Failed Requests".

You may be able to improve performance by restarting the current configuration, instead of restoring it. See Section 3.4.6, "Restarting Configurations" on page 3-13.

You must be aware of the possible effects of changing the model structure or configuration rules between the time you save a configuration and the time you restore it. 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.

Example:

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 3.5.4, "Getting and Setting Logic States" on page 3-20 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 appear in the order.

3.4.6 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 (such as the DHTML UI).

You may be able to save significant processing time by not re-validating the configuration after restarting it, since you have validated it once already.

3.4.7 Renaming Instances

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 modify 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 optional component. The component to be renamed must not be a mandatory component. The name that you set will persist 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 3–2.

Example 3–2 Renaming an Instance

```
...
 String inputText = "My Instance Name";
 Component c  = (Component)mUi.getCurrentScreen().getRuntimeNode();
 c.setInstanceName(inputText);
...
```

For a full example of how to change the name of an instance, see the Functional Companion example named `ChangeInstanceName.java` in the *Oracle Configurator Implementation Guide*.

3.4.8 Automatic Behavior for Configurations

You can define behavior that will be executed whenever a configuration is processed in certain ways, by extending the class `AutoFunctionalCompanion` and overriding its methods. Table 3–4 on page 3-14 describes these methods, and the circumstances under which you should use them.

Examples

For examples of Functional Companion code that uses the methods of `AutoFunctionalCompanion`, see Appendix B, "Code Examples", and some of the Functional Companion examples in *Oracle Configurator Methodologies*. The specific examples for each method are indicated in Table 3–4.

Table 3–4 Methods of AutoFunctionalCompanion

Method to Override	Executed ...	Comments and Examples
`onNew()`	When a newly-created configuration is activated	See also Section 3.10.3, "Initial Requests" on page 3-37 for a discussion of using the `onNew()` method. Examples: Example B–2 on page B-4, and `CfgInputExample.java` in *Oracle Configurator Methodologies*.

Table 3–4 (Cont.) Methods of AutoFunctionalCompanion

Method to Override	Executed ...	Comments and Examples
`onSave()`	Before a configuration is saved	The `onSave()` method should only be used with a user interface generated by Oracle Configurator Developer, since the method relies on the Oracle Configurator UI Server to determine whether the configuration being saved is valid or not.
`afterSave()`	After a configuration is saved	Clicking the **Done** button in the runtime Oracle Configurator terminates the configuration session and saves the configuration, if it is valid. Example: `WriteAttributes.java` in *Oracle Configurator Methodologies*.
`onRestore()`	After a configuration is restored	Example: `CfgInputExample.java` in the *Oracle Configurator Implementation Guide*.
`onSummary()`	When the Summary screen is displayed	Clicking the **Summary** button in the runtime Oracle Configurator displays the Summary screen.

In Oracle Configurator Developer, associate the Functional Companion that extends the class `AutoFunctionalCompanion` with the root node of your Model, and choose **Event-Driven** as the Type. See Section 1.4.1, "Associating Functional Companions with your Model" on page 1-13 for details.

In the runtime Oracle Configurator, the Functional Companion runs when one of the events listed in Table 3–4 on page 3-14 is executed (such as after a configuration is saved). When the Functional Companion runs, the CIO automatically determines the correct method to invoke, based on the method that has already been internally registered for the event.

> **Note:** Do not confuse a Functional Companion that extends the class `AutoFunctionalCompanion` with an Auto-Configuration Functional Companion, which extends `FunctionalCompanion` and overrides `autoConfigure()`. See Section 3.12.2 on page 3-44.

3.4.9 Access to Configuration Parameters

If you are using Oracle Configurator in a Web deployment, you can use a Functional Companion 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 Functional Companion that calls `Configuration.getUserParameters()`, which returns a `NameValuePairSet` object. This object contains all the parameter names and values stored by the Oracle Configurator UI Server when it processes the initialization message sent by the host application to the Oracle Configurator Servlet.

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.

3.5 Access to Nodes of the Model at Runtime

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 3.6, "Introspection through IRuntimeNode" on page 3-26.

Use `IRuntimeNode.getConfiguration()` to get the configuration to which a node belongs.

3.5.1 Opportunities for Modifying the Configuration

There are some restrictions on when you can modify the state of a configuration with a Functional Companion.

- Normally, you should use an Auto-Configuration Functional Companion to perform such modifications. To do so, you perform the actions in the body of the `IFunctionalCompanion.autoConfigure()` method. Your end user invokes the Auto-Configuration companion by clicking a button in the User Interface. See Section 3.12.2, "The autoConfigure() Interface Method" on page 3-44.
- If you want to modify the configuration automatically when an internally registered event occurs (such as creating or saving a configuration), you must use the methods of the class `AutoFunctionalCompanion`. For instance, you may want to perform certain selections when the configuration session begins or ends.
- If you want to apply a set of logic requests to a configuration when it is created, you define a set of initial requests. See Section 3.10.3, "Initial Requests" on page 3-37.
- You should never modify the configuration in an Initialization Functional Companion (that is, in the `IFunctionalCompanion.initialize()` method). See Section 3.12.1, "The initialize() Interface Method" on page 3-42.
- You should never modify the configuration in a Validation Functional Companion (that is, in the `IFunctionalCompanion.validate()` method). See Section 3.12.3, "The validate() Interface Method" on page 3-45.

Modifying the configuration through a Functional Companion is also referred to as **side-effecting** it.

3.5.2 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 set the **Instances** attribute of a Model, BOM Model, or Component to 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 mandatory components. Components that have a minimum number of instances of 0 and a maximum number of instances of 1 or more are called optional components. See the *Oracle*

Configurator Developer User's Guide for details about mandatory and optional components.

3.5.2.1 Adding and Deleting Optional Components

It is most likely that you would add or delete optional components in an Auto-Configuration Functional Companion. See Section 3.12.2, "The autoConfigure() Interface Method" on page 3-44.

Use `ComponentSet.add()` to add an optional 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 optional 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 the DHTML user interface, when the end user adds an instance of an optional component that is a 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 Functional Companion to add an instance of a BOM Model, that instance is *not* automatically selected. If you want your Functional Companion to select the instance, you must do it explicitly, as shown in Example 3–3 on page 3-18. Optional components that do not represent BOM Models cannot be selected.

Example 3–3 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();
    }
...
```

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

3.5.2.2 Getting the Functional Companions for a Component

Use `Component.getFunctionalCompanions()` to return a list of all the Functional Companions associated with a component. Remember that each instantiated component has its own instances of every Functional Companion associated with it. See Section 1.1.2, "Important Facts About Functional Companions" on page 1-3.

3.5.3 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 3.6, "Introspection through IRuntimeNode" on page 3-26). Each type also implements its own interface with appropriately specialized methods.

- TextFeatures have a string value.
- BooleanFeatures have a boolean (true/false/unknown) state.
- Features are represented by OptionFeature objects. OptionFeatures have a logic value, and a set of options as children. 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 Options, but not the Features they belong to. However, it is possible to use `select()` to select an `OptionFeature` object itself when using a Functional Companion. You should avoid selecting `OptionFeature` objects. If you do so and save the configuration, then this selection will not be applied if you later restore the configuration.
- DecimalFeatures have a floating point value.

- IntegerFeatures have an integer numeric value. The value can be positive, negative, or zero.
- CountFeatures an associated integer-valued numeric count, and are a special case of IntegerFeatures, with a count greater than zero. CountFeatures behave like counted options in an OptionFeature (a Feature whose values are a List of Options).

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.

3.5.4 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 input states, used to specify a new state for an object, listed in Table 3–5:

Table 3–5 Input 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 output states, returned when querying an object for its state listed in Table 3–6:

Table 3–6 Output 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.
UNKNOWN	The Unknown output state, indicating that there is no current state.
UTRUE	The User True output state, indicating that a user has set this object to true.

- A set of methods for getting and setting the object's state listed in Table 3–7:

Table 3–7 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 toward 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 use- specified state.
isLogic()	Tells whether this feature is in a logically specified state.
isUnknown()	Tells whether this feature is in unknown or known state.

- The code fragment in Example 3–4 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 3–4 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);
```

- Using `isUnknown()` is important when a node is cast to a 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 3–5, which uses `setState()` with `TOGGLE`, toggles the state of the selected item in the Model tree.

Example 3–5 Setting the state of a node

```
private void toggleSelectedItem() {
    IState node = (IState)getSelectedNode();
      node.setState(IState.TOGGLE);
    }
    catch (LogicalException le) {}
    catch (TransactionException te) {}
```

You should not use the `TOGGLE` state unless you are working with a user interface (see Section 2.1, "Controlling User Interface Elements" on page 2-1). 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 will throw a non-overridable 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 non-overridable logical contradiction:

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

- When determining if the state of a node is true (and when you aren't interested in the difference between UTRUE and LTRUE), the proper way to do this is to call `StateNode.isTrueState(int state)`.

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

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

  then the test will only return `true` if the logic state is UTRUE, but not if it is LTRUE.

3.5.5 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 Appendix 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 3–6 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 3–6 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 3.9, "Validating Configurations" on page 3-33 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 Min/Max 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 in which a Min/Max violation has occurred. Here is a suggested method for dealing with this situation:

1. Open a transaction.
2. Set the new value.
3. Close the transaction.
4. Get the collection of validation failures for the configuration.
5. If the last transaction caused a Min/Max violation, then call `Configuration.undo()`, which retracts the last 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.

3.5.6 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.

3.5.7 Access to Options

Option features have special methods for selecting options and querying for selected options. The `selectOption()` method implements mutual exclusion behavior for option features with a min/max of 1/1 by deselecting a currently selected option before selecting the new option. The `getSelectedOption()` method throws the TooManySelectedException if more than one option is selected in the feature.

An option is a child of an option feature which supports a true/false logic state and a count. Options implement the `IRuntimeNode` interface.

You can use the interface `IOption` to select, deselect, and determine the selection state of Options. Table 3–8 on page 3-25 lists these methods.

Table 3–8 Methods of the Interface IOption

Method	Action
`deselect()`	Deselect this Option.
`isSelected()`	Returns true if this Option is selected, and false otherwise.
`select()`	Select this Option.

The code fragment in Example 3–7 displays a "check" icon if an Option of a runtime node is selected, and displays an "unsatisfied" icon if the node is logically unsatisfied:

Example 3–7 Testing whether an option is selected, or satisfied

```
IRuntimeNode rtNode = (IRuntimeNode)value;
    if (value instanceof IOption) {
      IOption optionNode = (IOption)value;
      if (optionNode.isSelected()) {
        setIcon(checkIcon);
      }
    } else if (rtNode.isUnsatisfied()) {
      setIcon(unsatIcon);
    }
```

```
return this;
```

In this example, assume that `checkIcon` and `unsatIcon` point to icon files, and that `setIcon()` is a custom method that displays them.

3.6 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" Functional Companions, which can interact with a Model tree dynamically, without having prior knowledge of its structure. Table 3–9 on page 3-26 lists some of the more important of these methods.

Table 3–9 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.
`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.

Table 3–9 (Cont.) Important Methods of the interface IRuntimeNode

Method	Action
`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 will nevertheless be 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.
`isUnsatisfiedNode()`	Returns true if this particular node has not been completely configured.

The code fragment in Example 3–8 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 3–8 Getting a child node by name

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

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

The code fragment in Example 3–9 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 3–9 Collecting all child nodes by type

```
IRuntimeNode comp;
TextFeature textFeat;
com.sun.java.util.collections.List textFeatList;
Iterator iter;
//get all the text features
            textFeatList = comp.getChildrenByType(IRuntimeNode.TEXT_FEATURE);
            traverseTree(comp.getChildren(),
                         TEXT_FEATURE,
                         textFeatList);
            iter = textFeatList.iterator();
```

The code fragment in Example 3–10 uses the `isUiVisible()` method to determine whether the current runtime node is visible in the user interface.

Example 3–10 Determining UI visibility

```
Configuration config = node.getConfiguration();
...
        IRuntimeNode root = (IRuntimeNode)config.getRootComponent();
        boolean isVisbl = root.isUiVisible();
```

3.7 Logic Transactions

In order to help you maintain consistency in interactions with the Oracle Configurator logic engine, you can use *configuration-level logic transactions*. A logic transaction comprises all the logical assertions that constitute a user interaction. At the end of a transaction, the CIO returns a list of all validation failures. See Section 3.9, "Validating Configurations" on page 3-33.

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 3.5.4 on page 3-20 and Section 3.5.5 on page 3-23).

- 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.endConfigTransaction()` ends the transaction that was started with `beginConfigTransaction()`, without committing it (thus skipping validation checking).
- `Configuration.commitConfigTransaction()` commits the given transaction or series of nested transactions, propagates the effect of user

selections throughout the configuration Model, and triggers validation checking (see Section 3.9, "Validating Configurations" on page 3-33).

- `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 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 initial requests. See Section 3.10.3, "Initial Requests" on page 3-37.

There are situations in which you must take care to commit a transaction at the appropriate time. The fragmentary code in Example 3–11 on page 3-29 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, "Setting Initial Requests (InitialRequestTest.java)" on page B-4 also illustrates the use of transactions.

Example 3–11 Using a logic transaction with a deletion

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

// ---------------------------------------------------------
// This sequence produces unintended results:
...
// Add a component:
comp = compSet.add();
...
// User selects a child of compSet (interactively).
...
// Delete the component:
compSet.delete(comp);
...
// 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.
```

3.8 Handling Logical Contradictions

When you make a logic request to modify the state of a logic network, for instance by using `IState.setState()`, the result may be a failure of the request because of a logical contradiction. Such a failure will create and throw a *logical exception*, accessed through either the `LogicalException` or `LogicalOverridableException` objects. A `LogicalException` cannot be overridden.

See Section 3.8.2, "Overriding Contradictions" on page 3-31 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 strings for the failure.

- Use `LogicalException.getMessage()` to provide a message containing either the cause or the reasons.

3.8.1 Generating Error Messages from Contradictions

You can use the `Reason` object to wrap the information returned by a contradiction, in order to include information about internal error messages.

- You must provide the constructor of the `Reason` class with all of the arguments listed in the following table:

Table 3–10 Arguments for the Reason Constructor

Argument	Meaning
`type`	What type of reason this is.
`node`	The node that caused the problem.
`msg`	The message returned.

- Use `Reason.getMsg()` 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.

3.8.2 Overriding Contradictions

Your runtime Oracle Configurator or Functional Companion 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 back 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. Example 3–12 on page 3-32 is a code fragment 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 Functional Companion should try to handle the situation.

Example 3–12 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) {
      rollbackConfigTransaction();
    }
    else {
      try {
          // override the contradiction and ...
        loe.override();
          // ... finish the transaction
        commitConfigTransaction();
      }
      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);
  }
```

3.8.3 Raising Exceptions

When a Functional Companion is invoked, the Oracle Configurator UI Server wraps a transaction around this invocation.

If a contradiction occurs, and it is not handled in your code, and is not rolled back inside the Functional Companion, this may result in a fatal exception. If there is a fatal exception, the UI Server kills the configuration session (dropping any open transactions), and throws a `FunctionalCompanionException`, which displays a message to the end user. The user's configuration is ended.

If your Functional Companion handles an error by throwing a `FuncCompErrorException`, the UI Server displays the message that you specify to the end user, and rolls back the transaction. The user's configuration session is allowed to continue.

If the error throws a `LogicalException`, the UI Server displays a message to the end user, and rolls back the transaction. The user's configuration session is allowed to continue.

If the error throws a `RuntimeException`, the UI Server kills the user's configuration session. The UI Server continues to operate, and your end user can begin a new configuration session.

In a previous version of the CIO, you could throw a `FuncCompMessageException`, which is now deprecated, but is retained for backward compatibility with existing 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 UI Server 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. This situation could be a particular problem for Auto-Configuration Functional Companions.

Warning: The class `FuncCompMessageException` is now deprecated, but is retained for backward compatibility with existing code.

3.9 Validating Configurations

You want to be able to check whether a Configuration is valid (that is, does not violate the rules associated with it).

The CIO validates a Configuration after all logical assertions that constitute a user interaction are performed. This corresponds exactly to the length of a logical transaction. See Section 3.7, "Logic Transactions" on page 3-28.

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.

At this point, you can call the methods of `oracle.apps.cz.cio.Configuration` listed in Table 3–11:

Table 3–11 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.
`getSelectedItems()`	Returns a collection of selected items as a `StatusInfo` structure indicating the set of selected (true) items in the Configuration.
`getUnsatisfiedItems()`	Returns a collection of unsatisfied items as a `StatusInfo` structure indicating the set of unsatisfied items in the Configuration.

As nodes become selected they have a status of `STATUS_NEW`. If they continue to be selected since the last transaction their status is `STATUS_EXISTING`. If they become unselected, their status becomes `STATUS_DELETED` until the next transaction at which time they will be removed from the collection.

If you are writing a Functional Companion, the `validate()` method should return a list of `CompanionValidationFailure` objects in the event of a validation failure. This allows you to return more than one failure. Your `validate()` method can include several tests; you can track which ones failed, and why. (See Section 3.12.3 on page 3-45 for information about the `validate()` method.)

If the validation fails, then information about the failure is gathered by the CIO in a List of `CompanionValidationFailure` objects. In general, if a Functional Companion needs to return a violation message about a particular runtime node, you have to create a CompanionValidationFailure object and pass it the runtime node and the message. The code fragment in Example 3–13 illustrates this point.

Example 3–13 Returning a list of validation failures

```
public class Test extends FunctionalCompanion implements IFunctionalCompanion
{
...
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 CompanionValidationFailure("Value less than minimum",
                                                 node,
                                                  this) );
    if(failures.isEmpty())
        return null;
    else
        return failures;
...
}
```

If, though, the violation persists after the next user action, the Functional Companion shouldn't need to create a new `CompanionValidationFailure`, but should instead return the same object. This prevents the CIO from returning the already known violation message as a new violation message, which might be annoying for the user.

3.10 Using Requests

A logic 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 3–12 on page 3-35 lists some methods of this type:

Table 3–12 Methods typically used to make requests

Method	Described In ...
`IState.setState()`	Getting and Setting Logic States on page 3-20
`ICount.setCount()`	Getting and Setting Numeric Values on page 3-23
`IOPtion.select()`	Access to Options on page 3-25

- Requests that set a state or value, such as those listed in Table 3–12, are called *user requests*. See Section 3.10.2, "User Requests" on page 3-37.

- You can code a set of user requests that are applied to a new configuration when it is created. These are called *initial requests*. See Section 3.10.3, "Initial Requests" on page 3-37.
- When user requests fail, due to an override, the CIO generates a list of these *failed requests*. See Section 3.10.4, "Failed Requests" on page 3-40.
- You can get information about a request, by interrogating an instance of the Request object. See Section 3.10.1, "Getting Information about Requests" on page 3-36.

3.10.1 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, as described in Section 3.10, "Using Requests" on page 3-35.

You can create a `Request` object and pass its constructor the runtime node on which the request has been made, and a string that is the value of the request.

```
Request req = new Request(node, value);
```

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 3–13.

Table 3–13 Type methods of the class Request

This returns TRUE if ...	... the request made was for ...	The value[1] of the request is ...
`isNumericRequest()`	changing the numeric value of a runtime node	a Number
`isStateRequest()`	changing the state of a runtime node	"t", "true", "f", "false", "toggle", or "unknown"
`isTrueStateRequest()`	changing the state of a runtime node to True	"t" or "true"

Table 3–13 (Cont.) Type methods of the class Request

This returns TRUE if ...	... the request made was for ...	The value[1] of the request is ...
`isFalseStateRequest()`	changing the state of a runtime node to False	"f" or "false"
`isToggleStateRequest()`	toggling the state of a runtime node	"toggle"
`isUnknownStateRequest()`	unsetting the state of a runtime node	"unknown"

[1] The test for the value of the request is case-insensitive.

3.10.2 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 Functional Companion.

```
...
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();
}
...
```

3.10.3 Initial Requests

You can specify a set of logic requests to be applied to every new configuration as the configuration is created. Such requests are called initial requests.

You apply initial requests automatically on the creation of a configuration, following the practice illustrated in Example 3–14 on page 3-38 and in the following steps:

1. Define a Functional Companion that extends the class `AutoFunctionalCompanion`.

   ```
   public class compInitReq extends AutoFunctionalCompanion { ... }
   ```

2. In the class, invoke the `onNew()` method of `AutoFunctionalCompanion`, which is called right after the activation of a newly-created configuration.

   ```
   public void onNew() throws LogicalException { ... }
   ```

3. In the `onNew()` method, begin a configuration transaction, using `Configuration.beginConfigTransaction()`.

   ```
   IRuntimeNode node = getRuntimeNode();
   ConfigTransaction tr = node.getConfiguration().beginConfigTransaction();
   ```

   See Section 3.7, "Logic Transactions" on page 3-28 for details about transactions.

4. Specify that the transaction will contain initial requests, using `ConfigTransaction.useInitialRequests()`.

   ```
   tr.useInitialRequests();
   ```

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

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

   See Section 3.10, "Using Requests" on page 3-35 for details about setting logic requests.

6. When you have set all the desired initial requests, commit the logic transaction.

   ```
   node.getConfiguration().commitConfigTransaction(tr);
   ```

These steps are combined in Example 3–14. For a fuller example of using initial requests, see Example B–2, "Setting Initial Requests (InitialRequestTest.java)" on page B-4.

Example 3–14 Using initial requests

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

public class compInitReq extends AutoFunctionalCompanion
{
  public void onNew() throws LogicalException {
    try {
      IRuntimeNode node = getRuntimeNode();
      ConfigTransaction tr = node.getConfiguration().beginConfigTransaction();
      tr.useInitialRequests();
      BooleanFeature feat = (BooleanFeature)node.getChildByName("Feature_1234");
      feat.setState(IState.TRUE);
```

```
      node.getConfiguration().commitConfigTransaction(tr);
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

3.10.3.1 Usage Notes on Initial Requests

- You can think of a transaction that includes `ConfigTransaction.useInitialRequests()` (as illustrated in Step 4. on page 3-38) as putting the CIO in "initial request mode". You can nest any number of subtransactions within this transaction; the requests in these subtransactions all inherit this mode of being initial requests. You can perform overrides and rollbacks as you would with ordinary user requests. You must commit or roll back the initial-request transaction, as in step 6., to indicate the conclusion of the initial requests. You can then specify other user requests in your Functional Companion.
- When you save a configuration that includes initial requests, the initial requests are saved as part of the configuration. When you restore such a configuration, with `CIO.restoreConfiguration()`, the initial requests are reapplied to the configuration.

3.10.3.2 Limitations on Initial Requests

- Initial requests can only be specified when a new configuration is activated, by using the `onNew()` method of `AutoFunctionalCompanion`.
- After you apply initial requests to a configuration, you cannot override any of the initial requests with user requests. Attempting to override an initial request will result in a `LogicException`.
- You must specify all initial requests before specifying any user requests. Attempting to open a transaction with initial requests after specifying an ordinary user request will result in an `InitialRequestException`.
- If you use the `onRestore()` method of `AutoFunctionalCompanion`, which is called after a configuration is restored, you cannot specify any additional initial requests. The existing set of initial requests on the saved configuration is restored.
- You cannot use initial requests to add or delete components.

3.10.4 Failed Requests

When you use `LogicalOverridableException.override()` to override a logical contradiction (see Section 3.8.2, "Overriding Contradictions" on page 3-31), 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.3.2, "Getting a List of Failed Requests" on page B-7 for an example.

3.11 Working with Decimal Quantities

In previous versions of Oracle Configurator, all quantities in imported BOM Standard Items were treated as integers. Beginning with 11*i* Patchset F, quantities for imported BOM Standard Items can be either integers or decimals.

Table 3–14, " Methods for Integer and Decimal Nodes" on page 3-40 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 will raise 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 3–14 Methods for Integer and Decimal Nodes

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

Table 3–14 (Cont.) Methods for Integer and Decimal Nodes

Class/Interface	Integer Method	Decimal Method
StateCountNode	getCount()	getDecimalCount()
	setCount()	setDecimalCount()

3.12 Standard Interface Methods for Functional Companions

You provide functionality for your Functional Companion by implementing body code for the methods described in this section. These methods are listed in Table 3–15 on page 3-41.

For particulars that apply to the languages currently supported by the CIO, and examples, see Section 1.3, "Building Functional Companions" on page 1-7.

These methods are invoked by the runtime Oracle Configurator, through the CIO, in response to program events or the actions of end users. The type of method invoked for each component is determined when you associate the component with a Functional Companion in Oracle Configurator Developer. See Section 1.4, "Incorporating Functional Companions in Your Configurator" on page 1-13 for details.

These methods are invoked by the CIO for each Functional Companion object that it creates for the components in your Model. Note that your code does not invoke these methods directly; that is done by the CIO. Rather, you implement the body of each method, using the API provided by the CIO to communicate with your Model.

The body of any or all of these methods can be empty. Your Functional Companion object has to implement only those methods indicated in Oracle Configurator Developer.

The interface that defines these methods is:

`oracle.apps.cz.cio.IFunctionalCompanion`

Table 3–15 Standard methods of the IFunctionalCompanion interface

Method	Purpose	Details in
initialize()	Saves information about the Model and performs any actions needed to initialize the Functional Companion.	Section 3.12.1

Table 3–15 (Cont.) Standard methods of the IFunctionalCompanion interface

Method	Purpose	Details in
`autoConfigure()`	Performs a programmatic configuration step.	Section 3.12.2
`validate()`	Programmatically checks that a configuration is valid and throws a LogicalException object if the Model is not valid.	Section 3.12.3
`generateOutput()`	Generates output for this component, for either a thick or thin client.	Section 3.12.4
`terminate()`	Performs any cleanup on this Functional Companion that needs to occur before the Companion is destroyed.	Section 3.12.5

3.12.1 The initialize() Interface Method

The `IFunctionalCompanion.initialize()` method is called when the companion is created. It connects a Functional Companion object to its configuration modeling environment (for example, a running instance of the runtime Oracle Configurator). Be aware that Functional Companions are created and initialized after all subcomponent instances are created for the current component instance.

Your implementation of `initialize()` can include tasks that you wish to perform when the Functional Companion is first created. For example, you might wish to start writing audit messages to a log file, tracking the actions performed by your end users.

When a runtime Oracle Configurator runs, it creates runtime instances of all the components in the Model and their associated Functional Companions. When a Functional Companion object is created, the CIO calls `initialize()` and passes the input parameters listed in Table 3–16:

Table 3–16 Input Parameters for the initialize() method

Name	Type	Description
`node`	IRuntimeNode	The node instance associated with the Functional Companion being created. Specified in Configurator Developer.
`name`	String	The name of the Functional Companion. Specified in Configurator Developer.
`description`	String	A description of the Functional Companion. Specified in Configurator Developer.

Table 3–16 (Cont.) Input Parameters for the initialize() method

Name	Type	Description
`id`	int	The database ID of the Functional Companion. Created internally.

Note: It is worth emphasizing that the node passed as the first input parameter to `initialize()` is specified in Oracle Configurator Developer, when you create the Functional Companion rule that associates a Model node with your Functional Companion.

Your Functional Companion should ordinarily never directly call `FunctionalCompanion.initialize()`, since the CIO does that for you automatically. However, if your Functional Companion extends `FunctionalCompanion` as its base class, and you wish to perform some specialized initialization tasks, then the overriding `initialize()` method in *your* class should call `super.initialize()`. This passes some necessary variables to the superclass (`oracle.apps.cz.cio.FunctionalCompanion`) so that its methods will work.

It is not normally necessary to implement your own `initialize()` method in your Functional Companion. If you need to obtain the values of the input parameters of `FunctionalCompanion.initialize()` for use elsewhere in your Functional Companion, you can use the set of accessor methods of `FunctionalCompanion` already provided in the `oracle.apps.cz.cio.FunctionalCompanion` base class. Each of these methods, which are listed in Table 3–17 on page 3-43, returns the value of the corresponding input parameter:

Table 3–17 Methods for Returning Input Parameters

Method	Description
`getRuntimeNode()`	Returns the runtime node to which this functional is associated.
`getName()`	Returns the name of the functional companion.
`getDescription()`	Returns the description of the functional companion.
`getID()`	Returns the database ID of the functional companion.

Note: Currently, in Configurator Developer, you can only associate a Functional Companion with certain types of Model nodes, which correspond to the `node` parameter of `initialize()`. However, to accommodate possible future enhancement of Configurator Developer, the `IFunctionalCompanion` interface allows any runtime node to be associated with your Functional Companion.

Warning: You should not modify the Model in the `initialize()` method. Doing so can cause unexpected application failures.

3.12.2 The autoConfigure() Interface Method

The `IFunctionalCompanion.autoConfigure()` method is called at the request of the controlling User Interface, and can set states in the Model, add optional components, and other tasks that modify the Model. (Modifying the configuration through a Functional Companion is also referred to as side-effecting it.)

Note: Using an Auto-Configuration Functional Companion is the only approved way of modifying model structure.

This method performs an automatic configuration on the Model. This action can include changing the logical state of Options, or adding nodes underneath the selected component instance in the Model tree.

Your implementation of `autoConfigure()` can include configuration actions that you wish to be performed before your end users arrive at a certain point in a configuration session, or as the result of certain choices that they make.

Note: Do not confuse an Auto-Configuration Functional Companion, which extends `FunctionalCompanion` and overrides `autoConfigure()`, with a Functional Companion that extends the class `AutoFunctionalCompanion`. See Section 3.4.8 on page 3-14.

Note: In the current version of Release 11*i*, a contradiction that is not handled in your code, and is not rolled back, may result in a fatal exception, since the Oracle Configurator UI Server may not be able to commit the overarching transaction, or provide adequate feedback. This can be an issue for Auto-Configuration Functional Companions. For reference, see `FuncCompMessageException`.

Warning: Ordinarily, you cannot use an external program to modify an open configuration asynchronously. Doing so circumvents the Oracle Configurator UI Server, which can result in problems with multi-threading and updating of the display in the user interface. The key exception to this restriction is the execution of the body of the `autoConfigure()` method. When you use `autoConfigure()` to modify the Model, you must finish all modifications before returning from the method. This allows the UI Server to update the display to reflect any modifications.

3.12.3 The validate() Interface Method

The `IFunctionalCompanion.validate()` method is called automatically when a logical transaction takes place, and should return a `List` of `CompanionValidationFailure` objects if the Model is not valid.

This method performs a functional validation for the selected component instance each time the end user selects a node in the Model.

Warning: You should not modify the Model in the `validate()` method. Doing so can cause unexpected application failures.

Your implementation of `validate()` can include tasks that you wish to perform whenever your end users make any selection. For example, you might wish to perform a calculation based on the object count of the selected component, and present the end user with a notification if the result is outside a range that you define.

If the validation fails, then information about the failure is gathered by the CIO in a `List` of `CompanionValidationFailure` objects. See Section 3.9, "Validating Configurations" on page 3-33 for details.

The general structure of your implementation of `validate()` should be:

1. Collect inputs from the Model.
2. Call a generic validation function that you define outside the body of `validate()`.
3. Propagate the result back as the value of the function, either null or a `List` of `CompanionValidationFailure` objects.

Optimization factors to keep in mind are:

- Validation tests and defaults are applied more often than you may expect. Remember that `validate()` is called whenever the end user selects a node in the runtime Oracle Configurator.
- Defaults should be used very judiciously, if possible.
- Validation tests must be minimal. Design your code that `validate()` is called only when necessary. Design the validation test so that it is performed quickly.

3.12.4 The generateOutput() Interface Method

The `generateOutput()` method is invoked at the request of the controlling User Interface.

Your implementation of `generateOutput()` might include tasks such as writing to a database, creating a report, or producing a visualization of the end user's configuration choices.

There are two versions of `generateOutput()`:

- "thick client" version

  ```
  public String generateOutput();
  ```

  A thick client architecture is one in which the configuration Model, and the user interface for manipulating it, both reside on the same client machine. The thick client architecture allows your Functional Companion's Output method to produce output windows directly on the client machine.

- "thin client" version

  ```
  public void generateOutput(HttpServletResponse response) throws IOException
  ```

A thin client, browser-based architecture is one in which the configuration Model resides on a server, and the user interface resides on a client machine's web browser. The thin-client architecture allows your Functional Companion's Output method to produce output in web-browser windows.

This version is invoked when your Functional Companion operates in a Web deployment, such as Dynamic HTML in a browser.

See Section B.1, "Thin-Client generateOutput() Functional Companion" on page B-1 for an example.

Currently, there is no mechanism for output generated through `generateOutput()` to provide feedback to the User Interface or the runtime Model.

Warning: You should not modify the Model in the `generateOutput()` method. Doing so can cause unexpected application failures.

3.12.5 The terminate() Interface Method

The `IFunctionalCompanion.terminate()` method is called automatically by the CIO when the component that the Functional Companion is attached to is deleted from the running Model.

Your implementation of this method can include tasks that you wish to perform when the Functional Companion is deleted. For example, if `initialize()` opens a file and reads some data, `terminate()` would close the file.

Your Functional Companion should ordinarily never directly call `FunctionalCompanion.terminate()`, since the CIO does that for you automatically. However, if your Functional Companion extends `FunctionalCompanion` as its base class, and you wish to perform some specialized termination tasks, then the overriding `terminate()` method in *your* class should call `super.terminate()`.

Warning: You should not modify the Model in the `terminate()` method. Doing so can cause unexpected application failures.

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. These pages are installed with Oracle Configurator Developer, and are available through the Windows Start menu as part of the documentation for Oracle Configurator. See the *Oracle Configurator Installation Guide* for information about installing Oracle Configurator Developer.

B

Code Examples

This chapter contains code examples illustrating the use of Functional Companions 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.

B.1 Thin-Client generateOutput() Functional Companion

This Functional Companion uses the "thin-client" version of `generateOutput()` (see Section 3.12.4 on page 3-46). When you invoke the Functional Companion from a web browser, it produces an HTML representation of the runtime Model tree, beginning at the node to which the companion is attached.

To use this type of Functional Companion, you must use Oracle Configurator in a Web deployment. See the *Oracle Configurator Implementation Guide* for details not covered in this document. Here is a summary of the tasks:

- Compile the Java source code into a class file.
- In Configurator Developer, define a Functional Companion rule with the options listed in the following table:

Option	Choice
Type	Output
Base Component	The Component to which you want to attach the Functional Companion
Implementation language	Java
Program String	The name of the class file

- In Oracle Configurator Developer's User Interface module, define a button for the Component that invokes the Functional Companion.
- With your internet server, create an Oracle Configurator OC Servlet. See the *Oracle Configurator Installation Guide* for details.
- Add the new class file for the Functional Companion to the CLASSPATH environment variable for the servlet.
- You can test the Functional Companion from Configurator Developer, by specifying the URL of the servlet (in **Tools**>**Options**>**Test**>**Servlet URL**, where you also must select the Dynamic HTML in a browser option) and clicking the **Test** button. This opens a web browser, passing it a URL that includes an XMLmsg parameter containing the necessary OC initialization message. This message contains database connection and login strings, and specifies the Model to display, by means of the `ui_def_id` parameter that identifies the User Interface definition you created in Configurator Developer.
- You can test the Functional Companion outside Configurator Developer, by creating an HTML test page that substitutes for your host application. (Examples are provided in the *Oracle Configurator Implementation Guide*.) This page sends an OC initialization message that specifies database connection and login information, and the Model containing the Component. You can copy these parameters from the temporary file `C:\temp\dhtmlhtm.htm` that is generated by the Test button. Test the Functional Companion by opening the HTML test page.

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 Functional Companion 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.

Example B–1 Thin-client Output Functional Companion

```
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.FunctionalCompanion;
import oracle.apps.cz.cio.IRuntimeNode;

public class ShowStructure extends FunctionalCompanion {

  public void generateOutput(HttpServletResponse response) throws IOException {
    response.setContentType("text/html");
    response.setHeader ("Expires", "-1");
    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>");
    IRuntimeNode rootNode = getRuntimeNode();
    generateNode(out, rootNode, 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> ");
    for (Iterator i = node.getChildren().iterator(); i.hasNext(); ) {
      IRuntimeNode childNode = (IRuntimeNode)i.next();
      generateNode(out, childNode, (level + 1));
    }
  }
}
```

B.2 Saving and Exiting a Configuration

This example defines an Auto-Configuration Functional Companion that can be used to save and exit a configuration. This Functional Companion writes JavaScript code to a frame on the client browser, using the method `IUserInterface.outputToFrame()`.This method takes as parameters the path of the frame to write the JavaScript code to and the text of the code to write to the frame.

```
import oracle.apps.cz.cio.*;
public class Save extends FunctionalCompanion {
    public void autoConfigure() {
        IUserInterface  ui;
        ui = getRuntimeNode().getConfiguration().getUserInterface();
        String script = " <html><body onload='postEvent()'>";
        script += " <script>";
        script += " function postEvent() {";
        script += " bkr = parent.frames.czSrc.getUiBroker();";
        script += "     bkr.postClientMessage('<save exit=\"true\"></save>');";
        script += "} <\\/script></body></html>";
        ui.outputToFrame("parent.frames.rightBorder", script);
    }
}
```

B.3 Using Requests

For background, see Section 3.10, "Using Requests" on page 3-35.

B.3.1 Setting Initial Requests

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

For background, see Section 3.10.3, "Initial Requests" on page 3-37.

Example B–2 Setting Initial Requests (InitialRequestTest.java)

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

public class InitialRequestTest extends AutoFunctionalCcompanion
{
  public void onNew() throws LogicalException {
    IRuntimeNode comp = getRuntimeNode();
    Configuration config = comp.getConfiguration();
    ConfigTransaction itr = null;
    try {

      // Begin transaction that uses initial requests
      itr = config.beginConfigTransaction();
      itr.useInitialRequests();

      // 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 initial requests
    config.commitConfigTransaction(itr);

    // Try setting an initial 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 {
      // Attempt to use initial requests after an ordinary user request:
      // This is not allowed, and should produce a contradiction.
      tr = config.beginConfigTransaction();
      tr.useInitialRequests();
      ((IState)comp.getChildByName("boolean_feature_4")).setState(IState.TRUE);
      config.commitConfigTransaction(tr);
    } catch (InitialRequestException ire) {
      System.out.println("InitialRequestException: " + ire);
      config.rollbackConfigTransaction(tr);
      System.out.println("Expected exception in attempting initial requests " + ire);
    }
  } catch (Exception e) {
    e.printStackTrace();
```

```
    } catch (Throwable t) {
        t.printStackTrace();
    }
  }
}
```

B.3.2 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 3.10.4, "Failed Requests" on page 3-40.

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.4 User Interface Interaction

To use Functional Companions that interact with the User Interface, you must use Oracle Configurator in a Web deployment. See the *Oracle Configurator Implementation Guide* for details not covered in this document.

For a summary of the tasks required to implement a Functional Companion in a web deployment, see the description under Section B.1, "Thin-Client generateOutput() Functional Companion" on page B-1, with these differences:

- Define each Functional Companion rule as a Validation rule, so that it is invoked each time the end user selects a node associated with the rule.

- Do not create a button in the User Interface for these Functional Companions, since Validation rules are not invoked by buttons.

For details, see the descriptions below for each example.

B.4.1 Navigating to a Screen

This Functional Companion demonstrates the use of these methods to programmatically navigate from one Model tree node to another:

- `IUserInterface.navigateToScreen()`
- `IUserInterface.getCurrentScreen()`
- `IUserInterface.getOriginalScreen()`

To use this example, substitute the variables listed in the following table (and highlighted in the example code) with names of nodes in your own Model tree:

Variable	Usage
`optsel = "O4"`	Name of an option, which when selected navigates to a particular screen
`screenname = "C21"`	Component screen name to which one wants to navigate when `optsel` is selected.

Example B–4 Navigating to a Screen (ScreenNav.java)

```
import javax.servlet.http.HttpServletResponse;
import java.io.PrintWriter;
import java.io.*;
import oracle.apps.cz.cio.FunctionalCompanion;
import oracle.apps.cz.cio.AutoFunctionalCompanion;
import oracle.apps.cz.cio.IUserInterface;
import oracle.apps.cz.cio.*;
import com.sun.java.util.collections.List;
import com.sun.java.util.collections.Iterator;
import oracle.apps.cz.utilities.CheckedToUncheckedException ;
import java.awt.Toolkit;


public class ScreenNav extends FunctionalCompanion implements IUserInterfaceEventListener {

  IUserInterface  mUi;
  Component rootnode;
  PrintWriter p = null;
```

```
  String optsel = "O4";
  String screenname = "C21";


  public void initialize(IRuntimeNode node, String name, String description, int id) {
  try{
  p = new PrintWriter(new FileWriter("D:\\servlets\\screennav.txt",true));
  }
  catch(Exception e)
  {
    e.printStackTrace();
  }
    super.initialize(node, name, description, id);

    mUi = this.getRuntimeNode().getConfiguration().getUserInterface();
    mUi.addUserInterfaceEventListener(IUserInterfaceEvent.POST_CLICK, this);
    mUi.getCurrentScreen();
    rootnode = (Component)this.getRuntimeNode();
     }


    public void handleUserInterfaceEvent(IUserInterfaceEvent event) {
      try{
              IUserInterfaceScreen  currscreen = mUi.getCurrentScreen();
          if (event.getUiNode() == null) return;
          if ((event.getUiNode().getType() == IUserInterfaceNode.OPTION) &&
(event.getUiNode().getName().equalsIgnoreCase(optsel)))
             {
           p.println("Screen before navigating is " + mUi.getCurrentScreen().getName());
           IOption optf  = (IOption)event.getUiNode().getRuntimeNode();
           if (optf.isTrue()){
             p.println("Navigating to screen " + screenname);
             mUi.navigateToScreen(screenname);
             p.println("Current screen is " + mUi.getCurrentScreen().getName());
               p.println("Original screen was " + mUi.getOriginalScreen().getName());
           }
          }

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

   }

      public IRuntimeNode treetrav (IRuntimeNode rtnode,String nodename){
```

```
        List children =  rtnode.getChildren();
        Iterator iter = (Iterator)children.iterator();
      while (iter.hasNext()){
          IRuntimeNode currchild = (IRuntimeNode)iter.next();
          if (currchild.getName().equalsIgnoreCase(nodename))
            return currchild;
          if ((treetrav (currchild,nodename)) != null)
            return currchild;
      }
      return null;
    }
}
```

B.4.2 Changing the Caption of a Node

This Functional Companion demonstrates how to change the caption on a node in the Model tree view (of type TREELINK), using these classes and methods:

- `IUserInterfaceNode`
- `IUserInterfaceLabel.setUiCaption()`

To use this example, attach this Functional Companion to the root node of your Model tree, and substitute the variables listed in the following table (and highlighted in the example code) with names of nodes in your own Model tree:

Variable	Usage
`ChangeCapOf="C11"`	Name of the TREELINK caption that you want to change.
`ChangeCapTo="Comp11"`	Name that you want to change the TREELINK caption to.

Example B–5 Changing the Caption of a Node (CaptionChng.java)

```
import javax.servlet.http.HttpServletResponse;
import java.io.PrintWriter;
import java.io.*;
import oracle.apps.cz.cio.FunctionalCompanion;
import oracle.apps.cz.cio.AutoFunctionalCompanion;
import oracle.apps.cz.cio.IUserInterface;
import oracle.apps.cz.cio.*;
import com.sun.java.util.collections.List;
import oracle.apps.cz.uiserver.engine.*;
import com.sun.java.util.collections.Iterator;
import oracle.apps.cz.utilities.CheckedToUncheckedException ;
```

```
import oracle.apps.fnd.common.Context;
import java.awt.Toolkit;
import java.util.*;

public class CaptionChng extends FunctionalCompanion implements
IUserInterfaceEventListener,IUserInterfaceNode{
  IUserInterface  mUi;
  Component rootnode;
  String ChangeCapOf="C11";
  String ChangeCapTo="Comp11";


  public void initialize(IRuntimeNode node, String name, String description, int id) {
    super.initialize(node, name, description, id);

    mUi = this.getRuntimeNode().getConfiguration().getUserInterface();
    mUi.addUserInterfaceEventListener(IUserInterfaceEvent.POST_CLICK, this);
    rootnode = (Component)this.getRuntimeNode();
     }

   public void handleUserInterfaceEvent(IUserInterfaceEvent event) {
      try{

        rootnode = (Component)this.getRuntimeNode();
        if (event.getUiNode() == null) return;
        if (mUi.getCurrentScreen().getName().equals(ChangeCapOf))
        {
          IRuntimeNode nodefound = (IRuntimeNode)treetrav (rootnode,ChangeCapOf);
          Iterator iternode = mUi.getNode(nodefound).iterator();
          while (iternode.hasNext()){
          IUserInterfaceNode unode = (IUserInterfaceNode)iternode.next();
            if (unode.getType()== (IUserInterfaceNode.TREELINK))
            {
              ((IUserInterfaceLabel)unode).setUiCaption(ChangeCapTo);
              break;
            }
             }
        }        }
              catch(Exception e){e.printStackTrace();}

   }

      public IRuntimeNode treetrav (IRuntimeNode rootnode,String nodename){
        IRuntimeNode retNode = null;
        List children =  rootnode.getChildren();
```

```
        Iterator iter = (Iterator)children.iterator();
      while (iter.hasNext()){
              IRuntimeNode currchild = (IRuntimeNode)iter.next();
          if (currchild.getName().equalsIgnoreCase(nodename))
            return currchild;
          if (( retNode=treetrav (currchild,nodename)) != null)
            return retNode;
      }
      return null;
    }


      public int getType(){return 0;};
      public String getUiCaption(){return null;};
      public IUserInterfaceScreen getScreen(){return null;};
}
```

B.4.3 Changing an Image

This Functional Companion demonstrates how to change the image associated with a particular Option, using these classes:

- `IUserInterfaceImage`
- `IUserInterfaceScreen`

To use this example, in Oracle Configurator Developer:

1. In the Model module, create or use a Component `C1`.
2. In `C1`, create Feature `F1`.
3. In `F1`, create Options `O1`, `O2`, and `O3`

   Presume that you have installed the GIF files listed in the following table, and that all the images are the same size:

File name	Description
rightarr.gif	An image of a right-pointing arrow
lefttarr.gif	An image of a left-pointing arrow
blank.gif	A blank placeholder image

4. In O1 and O3, add **Properties** named img1, each with the value rightarr.gif.
5. In O2, add a Property named img1, with the value lefttarr.gif.
6. In the Rules module, define a Functional Companion rule referring to this example, and associate it with Component C1.
7. Choose **Tools** > **Generate Active Model**.
8. In the UI module, **Refresh** or **Create** a User Interface for the Model.
9. In the UI node for C1, add a Picture object, and name the Picture object placeholder. Refer to the Picture by providing its image path (for example, D:\orant\OC\Shared\ActiveMedia\blank.gif.)
10. Adjust the position and size of the Picture object to fit the image files.
11. Place these GIF files in the ActiveMedia folder in your Developer installation, and also in the OA_MEDIA directory of your Oracle Configurator Servlet.
12. Select the User Interface and choose **Edit** > **Refresh**.
13. Use the **Test** button to view the Dynamic HTML in a browser, and click on C1.
14. When you select O1 or O3 (of F1), a right-pointing arrow appears on the screen.
15. When you select O2, a left-pointing arrow appears on the screen.

Example B–6 Changing an Image (ImageChange.java)

```
import javax.servlet.http.HttpServletResponse;
import java.io.PrintWriter;
import java.io.*;
import oracle.apps.cz.cio.FunctionalCompanion;
import oracle.apps.cz.cio.AutoFunctionalCompanion;
import oracle.apps.cz.cio.IUserInterface;
import oracle.apps.cz.cio.*;
import com.sun.java.util.collections.List;
import com.sun.java.util.collections.Iterator;
import oracle.apps.cz.utilities.CheckedToUncheckedException ;
import java.awt.Toolkit;

public class ImageChange extends FunctionalCompanion implements IUserInterfaceEventListener {

  IUserInterface  mUi;
  Component rootnode;
  PrintWriter p = null;
```

```
  public void initialize(IRuntimeNode node, String name, String description, int id) {
    super.initialize(node, name, description, id);

    mUi = this.getRuntimeNode().getConfiguration().getUserInterface();
    mUi.addUserInterfaceEventListener(IUserInterfaceEvent.POST_CLICK, this);
    rootnode = (Component)this.getRuntimeNode();
   }



  public void handleUserInterfaceEvent(IUserInterfaceEvent event) {
      try{
        if (event.getUiNode().getType() == IUserInterfaceNode.OPTION);
          {
          Option optsel = (Option)event.getUiNode().getRuntimeNode();
          String imagename = optsel.getPropertyByName("img1").getStringValue();
          IUserInterfaceScreen currscr = mUi.getCurrentScreen();
              List images = currscr.getNode("placeholder");
          IUserInterfaceImage placeholder = (IUserInterfaceImage)images.get(0);
          placeholder.setFileName(imagename);

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

    }


}
```

B.5 Controlling Parent and Child Windows

To use Functional Companions that control parent and child windows, you must use Oracle Configurator in a Web deployment. See the *Oracle Configurator Implementation Guide* for details not covered in this document.

For a summary of the tasks required to implement a Functional Companion in a web deployment, see the description under Section B.1, "Thin-Client generateOutput() Functional Companion" on page B-1.

B.5.1 Locking, Unlocking and Refreshing Windows

For background on this example, see Section 2.4.1, "Locking, Unlocking and Refreshing Windows" on page 2-19.

When you define your Functional Companion rule, associate it with the compiled code from Example B–7.

The values for the host name and port number shown in the example must be tailored to your own site:

```
http://server01:8800/configurator/Replace
```

The methods that lock, unlock, and refresh the windows are highlighted.

The code for the servlet that implements the "update" function is the subject of Example B–8 under Section B.5.2, "Accessing a Shared Configuration Model" on page B-17.

Example B–7 Launching a Child Window (Launch.java)

```
import javax.servlet.http.HttpServletResponse;
import java.io.PrintWriter;
import oracle.apps.cz.cio.FunctionalCompanion;
import oracle.apps.cz.cio.AutoFunctionalCompanion;

public class Launch extends AutoFunctionalCompanion
{
    /**
     * Constructor:
     * Can be used for any necessary setup.
     */
    public Launch()
    {
    }

    public void generateOutput(HttpServletResponse response)
    {
      try
      {
          response.setContentType("text/html");
          PrintWriter out = response.getWriter();
          out.println("<html>");
          out.println("<head>");
          out.println("<script language=\"javascript\">");
          out.println("var winHandle = opener;");
```

```
          // The "opener" is the czSource.htm frame.
          // The methods are made public here.
          out.println("function lock() { opener.lockEventManagers();}");
          out.println("function unlock() { opener.unlockEventManagers();}");
          out.println("function refresh() { opener.refreshFrames();}");
          out.println("function update() { lock(); document.form1.submit();}");
          out.println("</script></head>");
          out.println("<body><form name='form1'
action='http://server01:8800/configurator/Replace'>");
          out.println("<input type='button' value='lock' onClick='lock();'>");
          out.println("<input type='button' value='unlock' onClick='unlock();'>");
          out.println("<input type='button' value='refresh' onClick='refresh();'>");
          out.println("<input type='button' value='update' onClick='update();'>");
          out.println("</form></body>");
          out.println("</html>");
      }
      catch(Exception e){e.printStackTrace();}
    }

    public void onNew()
    {

    }
}
```

B.5.2 Accessing a Shared Configuration Model

For background on this example, see Section 2.4.2, "Accessing a Shared Configuration Model" on page 2-19.

This servlet code implements the "update" function in Example B–7 under Section B.5, "Controlling Parent and Child Windows" on page B-15.

The names of the Model nodes shown in the example are:

```
Component-11820
Feature-11821
Option-11822
Option-11825
```

These names must be replaced by the names used in your own Model.

Example B–8 Accessing a Shared Configuration Model (Replace.java)

```
import javax.servlet.*;
import javax.servlet.http.*;
```

```
import java.io.*;
import java.util.Iterator;
import java.util.ArrayList;
import oracle.apps.cz.cio.Configuration;
import oracle.apps.cz.cio.Component;
import oracle.apps.cz.cio.IRuntimeNode;
import oracle.apps.cz.cio.RuntimeNode;
import oracle.apps.cz.cio.ComponentNode;
import oracle.apps.cz.cio.OptionFeature;
import oracle.apps.cz.cio.Option;
import oracle.apps.cz.cio.NoSuchChildException;
import oracle.apps.cz.cio.LogicalException;
import oracle.apps.cz.cio.TransactionException;
import oracle.apps.cz.cio.IState;
import oracle.apps.cz.cio.ConfigTransaction;

public class Replace extends HttpServlet {

//Initialize global variables
  public void init(ServletConfig config) throws ServletException {
    super.init(config);
  }

//Process the HTTP Get request
  public void doGet(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
    response.setContentType("text/html");
    ArrayList list = new ArrayList();
    // Here we are updating the Model.
    try{
        HttpSession ses = request.getSession(true);
        // Here we are getting the configuration Object stored in the http session.
        Configuration conf = (Configuration)ses.getValue("configurationObject");
        Component root = conf.getRootComponent();
        IRuntimeNode node = root.getChildByName("Component-11820");
        OptionFeature child = (OptionFeature)node.getChildByName("Feature-11821");
        Option opt1 = (Option)child.getChildByName("Option-11822");
        Option opt2 = (Option)child.getChildByName("Option-11825");
        ConfigTransaction ct = conf.beginConfigTransaction();
        try{
            try{
                opt1.setState(IState.UTRUE);
                opt2.setState(IState.UTRUE);
                conf.commitConfigTransaction(ct);
            }catch(LogicalException le){}
```

```
        }catch(TransactionException le){}

    }catch(NoSuchChildException e){}

    PrintWriter out = new PrintWriter (response.getOutputStream());
    out.println("<html>");
    out.println("<head>");
    out.println("<script language=\"javascript\">");
    out.println("var winHandle = opener;");
    out.println("function init() { winHandle.unlockEventManagers(); winHandle.refreshFrames();
}");
    out.println("</script></head>");
    out.println("<body onLoad='init();'>We have just updated and refreshed the model.</body>");
    out.println("</html>");
    out.close();
  }

//Process the HTTP Post request
  public void doPost(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = new PrintWriter (response.getOutputStream());
    out.println("<html>");
    out.println("<head><title>Replace</title></head>");
    out.println("<body>");
    out.println("</body></html>");
    out.close();
  }
//Get Servlet information

  public String getServletInfo() {
    return "Information about Replace";
  }
}
```

Glossary of Terms and Acronyms

This glossary contains definitions that you may need while working with Oracle Configurator.

Active Model

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 Active Model must be generated either in **Oracle Configurator Developer** or programmatically in order to access the configuration model at **runtime**.

API

Application Programming Interface

applet

A Java application running inside a Web browser. *See also* **Java** and **servlet**.

application architecture

The software structure of an application at **runtime**. Architecture affects how an application is used, maintained, extended, and changed.

architecture

See **application architecture**.

ATO

Assemble to Order

ATP

Available to Promise

attribute

The defining characteristic of something. Models have attributes such as Effectivity Sets and Usage. Components, Features, and Options have attributes such as Name, Description, and Effectivity.

benchmark

Represents performance data collected during **runtime** tests under various conditions that emulate expected and extreme use of the product.

beta

An external release, delivered as an installable application, and subject to acceptance, **validation**, and **integration testing**. Specially selected and prepared **end user**s may participate in beta testing.

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**.

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.

CIO

See **Oracle Configuration Interface Object (CIO)**.

client

A **runtime** program using a **server** to access functionality shared with other clients.

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 **Feature**s in the Model to control the allowable combinations of **Option**s. *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 **Option**s 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 **Feature**s, 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 **Component**s of the same type, where each Component of the set is independently configured.

concurrent manager

A process manager that coordinates the **concurrent process**es generated by **user**s' **concurrent request**s. An Oracle Applications product group can have several concurrent managers.

concurrent process

A task that can be scheduled and is run by a **concurrent manager**. A concurrent process runs simultaneously with interactive functions and other concurrent processes.

concurrent processing facility

An Oracle Applications facility that runs time-consuming, non-interactive tasks in the background.

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.

concurrent request

A user-initiated request issued to the concurrent processing facility to submit a non-interactive task, such as running a report.

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 Interface Object

See **Oracle Configuration Interface Object (CIO)**.

configuration model

Represents all possible configurations of the available **option**s, and consists of **model structure** and **rules**. It also commonly includes **User Interface** definitions and **Functional Companion**s. A configuration model is usually accessed in a **runtime Oracle Configurator window**. *See also* **model**.

configuration rules

The **Oracle Configurator Developer Logic rules**, **Compatibility rule**s, **Comparison rule**s, **Numeric rule**s, and **Design Chart**s available for defining **configuration**s. *See also* **rules**.

configuration session

The time from launching or invoking to exiting **Oracle Configurator**, during which **end user**s 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 hosting application so **end user**s

can make selections resulting in valid **configuration**s. *Compare* **Oracle Configurator**.

connectivity

The connection between **client** and database **server** 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**.

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

context

The surrounding text or conditions of something.

Determines which context-sensitive segments of a flexfield in the Oracle Applications database are available to an application or **user**. Used in defining **configuration attribute**s.

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 decrementing a total value, such as specifying the quantity of a **Resource** used.

count

The number or quantity of something, such as selected **option**s. *Compare* **instance**.

CRM

See **Customer Relationship Management**

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-centric extensions

See **customer-centric views**.

customer-centric views

Optional extensions to core functionality that supplement configuration activities with **rules** for **preselection**, **validation**, and **intelligent views**. View capabilities include generative geometry, drawings, sketches and schematics, charts, performance analyses, and **ROI** calculations.

Customer Relationship Management

The aspect of the enterprise that involves contact with customers, from lead generation to support services.

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.

data import

Populating the **Oracle Configurator schema** with enterprise data from **ERP** or legacy systems via **import tables**.

Data Integration Object

Also known as the DIO, the Data Integration Object is a **server** in the **runtime** application that creates and manages the interface between the **client** (usually a **user interface**) and the **Oracle Configurator schema**.

data maintenance environment

The environment in which the runtime **Oracle Configurator** data is maintained.

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 rule

An **Oracle Configurator Developer** Logic rule that determines the logic state of **Feature**s or **Option**s 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 **user**s' requirements. Defects are prioritized as critical, major, or minor, and fixes range from corrections or workarounds to enhancements. Also known as a bug.

defect tracking

A system of identifying defects for managing additional tests, testing, and approval for release to **user**s.

deliverable

A work product that is specified for review and delivery.

demonstration

A presentation of the tested application, showing a particular usage scenario.

Design Chart

An **Oracle Configurator Developer** rule type for defining advanced Explicit Compatibilities interactively in a table view.

design review

A technical review that focuses on application or **system** design.

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 model**s.

DHTML

Dynamic Hypertext Markup Language

DIO

See **Data Integration Object**.

distributed computing

Running various **component**s of a **system** on separate machines in one network, such as the database on a database **server** machine and the application software on a Web server machine.

DLL

Dynamically Linked Library

DSN

See **data source**.

element

Any entity within a **model**, such as **Option**s, **Total**s, **Resource**s, UI controls, and **component**s.

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 **system**s and **resource**s 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.

Excludes rule

An **Oracle Configurator Developer** Logic rule determines the logic state of **Feature**s or **Option**s 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 rule**.

extended functionality

A release after delivery of core functionality that extends that core functionality with **customer-centric views**, more complex proposal generation, discounting, quoting, and expanded integration with **ERP**, **CRM**, and third-party software.

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 **Option**s.

Functional Companion

An extension to the **configuration model** beyond what can be implemented in Configurator Developer.

An object associated with a **Component** that supplies methods that can be used to initialize, validate, and generate **customer-centric views** and outputs for the **configuration**.

functional specification

Document describing the functionality of the application based on **user** requirements.

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 rules** that automatically select some product **option**s 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

ICX

Inter-Cartridge Exchange

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 user**s. *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 rule

An **Oracle Configurator Developer** Logic rule that determines the logic state of **Feature**s or **Option**s 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 rule**.

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 Oracle Configurator schema Item Master structure, but without integrity constraints. Import tables allow batch population of the Oracle Configurator schema's Item Master. Import tables also store extractions from Oracle Applications or **legacy data** that create, update, or delete records in the Oracle Configurator schema **Item Master**.

incremental construction

The process of organizing the construction of the application into **build**s, where each build is designed to meet a specified portion of the overall requirements and is **unit test**ed.

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**.

install program

Software that sets up the local machine and installs the application for testing and use.

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 **component**s and making them work together.

integration testing

Testing the interaction among software programs that have been integrated into an application or **system**. *Compare* **unit test**.

intelligent views

Configuration output, such as reports, graphs, schematics, and diagrams, that help to illustrate the value proposition of what is being sold.

IS

Information Services

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 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. *See also* **applet** and **servlet**.

LAN

Local Area Network

LCE

Logical Configuration Engine. *Compare* **Active Model**.

legacy data

Data that cannot be imported without creating custom extraction programs.

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 rules

Logic rules directly or indirectly set the logical state (User or Logic True, User or Logic False, or **Unknown**) of **Feature**s and **Option**s 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 rule**, **Requires rule**, **Excludes rule**, and **Negates rule**.

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/`

Model

The entire hierarchical "tree" view of all the data required for **configuration**s, including **model structure**, variables such as **Resource**s and **Total**s, 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 nodees**, **BOM Standard Item nodes**, **Resources**, and **Totals**). May include reusable **components** (**References**).

MS

Microsoft Corporation

Negates rule

A type of **Oracle Configurator Developer** Logic rule 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 rule**.

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 express constraint among model elements in terms of numeric relationships. *See also*, **Contributes to** and **Consumes from**.

OC

See **Oracle Configurator**.

ODBC

Open Database Connectivity. A database access method that uses drivers to translate an application's data queries into **DBMS** commands.

OCD

See **Oracle Configurator Developer**.

opportunity

The workspace in Oracle Sales Online in which products, **system**s, and services are configured, quotes and proposals are generated, and orders are submitted.

option

A logical selection made by the **end user** 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 **Active Model**.

The CIO is the **API** that supports creating and navigating the Model, querying and modifying selection states, and saving and restoring **configuration**s.

Oracle Configurator

The product consisting of development tools and **runtime** applications such as the **Oracle Configurator 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 **Active Model**, and the **Oracle Configurator schema**. The application development architecture consists of **Oracle Configurator Developer** and the Oracle Configurator 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 **configurator**s.

Oracle Configurator engine

Also **LCE**. *Compare* **Active Model**.

Oracle Configurator 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**.

Oracle Configurator Servlet

Vehicle for **Oracle Configurator** containing the UI Server.

Oracle Configurator window

The **user interface** that is launched by accessing a **configuration model** and used by **end user**s to make the selections of a **configuration**.

Oracle SellingPoint Application

No longer available or supported.

output

The output generated by a **configurator**, such as quotes, proposals, and **customer-centric views**.

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 node**s from information in the **Item Master**.

preselection

The default state in a **configurator** that defines an initial selection of **Component**s, **Feature**s, and **Option**s 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.

project manager

A member of the project team who is responsible for directing the project during implementation.

project plan

A document that outlines the logistics of successfully implementing the project, including the schedule.

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

A kind of compatibility relationship where the allowable combinations of **Option**s 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 hosting 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.

QA

Quality Assurance

RAD

Rapid Application Development

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**.

regression test

An automated test that ensures the newest **build** still meets previously tested requirements and functionality. *See also* **incremental construction**.

Requires rule

An **Oracle Configurator Developer** Logic rule that determines the logic state of **Feature**s or **Option**s 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 rule**.

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.

RFQ

Request for Quote

ROI

Return on Investment

rules

Also called business rules or **configuration rules**. 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 rules** 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**.

sales configuration

A part of the sales process to which configuration technology has been applied in order to increase sales effectiveness and decrease order errors. Commonly identifies **customer requirements** and product configuration.

schema

The tables and objects of a data model that serve a particular product or business process. *See* **Oracle Configurator schema**.

SCM

Supply Chain Management

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**.

SFA

Sales Force Automation

solution

The deployed **system** as a response to a problem or problems.

SQA

Software Quality Assurance

SQL

Structured Query Language

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**.

test case

A description of inputs, execution instructions, and expected results that are created to determine whether a specific software feature works correctly or a specific requirement has been met.

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**.

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 **Oracle Configurator architecture runtime** architecture that is generated from the **model structure** and provides the graphical views necessary to create **configuration**s interactively. Interacts with the **Active Model** and data to give **end user**s access to customer requirements gathering, product selection, and **customer-centric views**.

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**.

user requirements

A description of what the **configurator** is expected to do from the **end user's** perspective.

user's guide

Documentation on using the application or **configurator** to solve the intended problem.

validation

Tests that ensure that configured **component**s will meet specific criteria set by an enterprise, such as that the components can be ordered or manufactured.

Validation

A type of **Functional Companion** that is implemented to ensure that the configured **component**s will meet specific criteria.

VAR

Value-Added Reseller

variable

Parts of the **Model** that are represented by **Total**s, **Resource**s, or numeric **Feature**s.

VB

Microsoft Visual Basic. Programming language in which portions of **Oracle Configurator Developer** are written.

verification

Tests that check whether the result agrees with the specification.

WAN

Wide Area Network

Web

The portion of the Internet that is the World Wide Web.

WIP

Work In Progress

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

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W

X