Oracle® Configurator

Methodologies

Release 11*i*

Part No. B10618-01

Februrary 2003

This book describes particular methods of using Oracle Configurator functionality for specialized purposes.

ORACLE®

Oracle Configurator Methodologies, Release 11*i*

Part No. B10618-01

Primary Author: Tina Brand, Mark Sawtelle

Contents

List of Examples

List of Figures

List of Tables

Send Us Your Comments

Oracle Configurator Methodologies, Release 11*i*

Part No. B10618-01

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

This manual presents tasks and information useful in implementing Oracle Configurator. See the *Oracle Configurator Installation Guide* for installation information and the *Oracle Configurator Developer User's Guide* for information about developing configuration models in Oracle Configurator Developer.

The *Oracle Configurator Custom Web Deployment Guide* of previous releases has been included in this manual.

Intended Audience

Anyone responsible for supporting use of Oracle Configurator should read this book. That includes supporting the development environment (Oracle Configurator Developer) as well as the runtime environment that is created for deployment.

Ordinarily, the tasks presented in this book are performed by a Database Administrator (DBA) or an Oracle Configurator implementer with DBA experience.

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 manual contains a table of contents, lists of examples, tables and figures, a reader comment form, several chapters, appendix, glossary, and index. The chapters are organized in parts; each part contains the chapters related to a particular methodology.

Within the chapters, information is organized in numbered sections of several levels. Note that level does not imply importance or degree of detail. For instance, third-level sections in one chapter (x.x.x) may not contain information of equivalent detail to the third-level sections in another chapter.

- Part I, "Configuration Attributes"
  - Chapter 1, "Configuration Attributes" describes the methodology for using certain configuration features of Oracle Configurator and host applications to capture and exchange data that is not standard inventory information.
- Part II, "Appendices"
  - Appendix A, "Code Examples" contains code examples that support other chapters of this document. These examples are fuller and longer than the examples provided in the rest of this document, which are often fragments.
- "Glossary of Terms and Acronyms" contains definitions that you may need while working with Oracle Configurator documentation.

  The Index provides an alternative method of searching for key concepts and product details.

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 Configuration Interface Object (CIO) Developer's Guide*
- *Oracle Configurator Performance Guide*

For more information, see the documentation for Oracle Applications (Release 11*i*) Oracle RDBMS (Release 8*i* or 9*i*), the *Oracle Applications Library*, the product-specific Release Notes for releases supported to work with Oracle Configurator, and the Configurator eTRM on Metalink, Oracle's technical support Web site.

Conventions

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

The following conventions are also used in this manual:

Convention	Meaning
. . .	Vertical ellipsis points in an example mean that information not directly related to the example has been omitted.
. . .	Horizontal ellipsis points in statements or commands mean that parts of the statement or command not directly related to the example have been omitted
boldface text	Boldface type in text indicates a new term, a term defined in the glossary, specific keys, and labels of user interface objects. Boldface type also indicates a menu, command, or option, especially within procedures
italics	Italic type in text, tables, or code examples indicates user-supplied text. Replace these placeholders with a specific value or string.
[ ]	Brackets enclose optional clauses from which you can choose one or none.

Convention	Meaning
>	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 at:

`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:

`www.oracle.com/support/`

Part I

Configuration Attributes

Part I contains the following chapters:

1

Configuration Attributes

This chapter describes a methodology for using configuration attributes.

1.1 Overview of Configuration Attributes

The term **configuration attributes** denotes a methodology for using certain existing features of Oracle Configurator and host applications to capture and exchange data that is not standard inventory information.

1.1.1 Purpose

There are situations in which you need the ability to capture and pass certain miscellaneous items of data between Oracle Configurator and a host application. Such miscellaneous items can include free-form text (such as addresses) or computed values (such as dimensions). The inventory information used by the runtime Oracle Configurator does not always provide the structure needed to store such non-standard data.

1.1.1.1 Typical Problems To Be Solved

This list of problems is by no means exhaustive. It is meant to illustrate the problems that this methodology addresses.

- In fabrication industries, raw material is often purchased in standard sizes. A problem occurs when the unit of measurement of the item in inventory (typically "Each") is different from the unit of measurement (UOM) required by the manufacturing process on the factory floor. Configuration rules can determine how many inventory items are needed for the process, but cannot capture the difference between the inventory measurement and the process measurement.

Example: A fabrication company stocks the inventory items listed in the following table:

Item ID	Description	UOM
AA248	2 x 4 Pine Stud, 8’	Each
AA148	1 x 4 Pine Stud, 8’	Each

Oracle Configurator may determine that a specific fabricated item requires 3 2x4s, each 61 inches long. While the quantity (3) and item ID (AA248) can be passed back to the host application, the additional information as to the required length of the item is not. The information that needs to be communicated to the manufacturing floor is "Get 3 of Item AA248 and cut all three down to 61 inches each".

- In many applications, end users need to enter or compute multiple attributes for a given item. This is common in the metals industry where multiple parameters may be required in order to specify or "formulate" a metal component. For example, a number of attributes may be required to specify a sheet of aluminum, including its length, width, gauge, and the percentages of the various components that make up the metal.
- It may be necessary to enter or determine attribute information and have that information associated with many items. For example, an attribute such as "voltage" may apply to a number of items within the configuration that share electrical properties.

1.1.1.2 Solutions

To solve the problems listed in Section 1.1.1.1, you can use Oracle Configurator configuration attributes. These are attributes that you design and attach to certain nodes of existing configuration models.

There are two distinct strategies for using configuration attributes, which you can implement independently of each other:

- **input**, described in Section 1.1.1.3
- **output**, described in Section 1.1.1.4

You can use the two strategies together, but there is no requirement to do so, and there are no special provisions in Oracle Configurator for making them interact.

As background for the output strategy, you should familiarize yourself with descriptive flexfields, which are a feature of Oracle Applications that allow you to capture non-standard information.

1.1.1.3 Use of Configuration Attributes for Input

The use of configuration attributes for input permits values that are stored by a host application in a database to be retrieved by Oracle Configurator and inserted into the configuration model at the beginning of a configuration session, as the initial values of specified Features. Unlike predefined initial values, configuration attribute values are obtained dynamically at runtime, and reflect the latest state of the host application.

During the configuration session, the input values can be modified in the normal way, and are subject to configuration rules. Note that the values changed during the session are not provided back to the host application.

Implementing the Strategy

Section 1.1.2.1 on page 1-5 provides an overview of how the input strategy works, and the steps required to implement it.

Section 1.2 on page 1-7 provides the details for implementing the input strategy.

1.1.1.4 Use of Configuration Attributes for Output

The use of configuration attributes for output permits values to be captured as part of an Oracle Configurator configuration session and, after the session, be provided back to a host application for further use in the downstream process.

By using configuration attributes for output, you can capture miscellaneous text or numeric data during the configuration session, store it in text or numeric Features in the configuration model, and then provide it to the hosting application so that it is still associated with the configured item.

At design time, you can:

- Supplement model structure in Oracle Configurator Developer (OCD) with attributes for storing additional parameters, such as: dimensions, location names, pricing annotations, and manufacturing notes.
- Write configuration rules against the added attributes
- Specify the level of association of the attribute values:
  - Only where attached in the configuration model

- Where attached plus immediate selected model children
- Where attached plus all selected model descendents

At runtime, you can:

- Associate the configuration attributes' values to specific line items, through descriptive flexfield context definitions
- Write configuration attributes with different contexts for the same line item
- Associate a single configuration attribute value:
  - to a single line item
  - to many line items
- Associate many configuration attribute values to a single line item

Context for Processing Output

To understand the requirements that are filled by using configuration attributes, it is helpful to summarize the events in a configuration session.

When a sales order or quote is created at runtime in a host application (for instance, Order Management, *i*Store, or Oracle Quoting), the end user is asked to specify a particular item to be configured. In the runtime Oracle Configurator, the end user selects predefined options and enters numeric values, such as dimensions. Oracle Configurator uses the configuration rules written for the configuration model in Oracle Configurator Developer to automatically select or deselect additional items as being required or excluded by the explicit user selections. After the configuration session is completed (with the Done button), a complete configuration is saved, in the Oracle Configurator schema of the application's database.

After the configuration is saved, the host application performs downstream processing that varies according to the specifics of that application. Consider Order Management as an example here. When booking the order, Order Management validates the configuration and explodes the configuration items onto the order, creating additional order lines for each selected subcomponent of the configurable model. Through invoking the Auto-Create Final Assembly concurrent program, the order (that is, all order lines) can be passed to Oracle Manufacturing to be scheduled and executed.

In addition to ensuring the valid configuration of selections and quantities in the BOM, Oracle Configurator can provide additional configuration attributes attached to the selected components that may or may not be directly related to the quantities.

Through a custom procedure (which must be written as part of this methodology), a downstream application can access the configuration attribute data.

The methodology described in this chapter provides a solution for:

- Defining these configuration attributes
- End-user entry of data for these configuration attributes
- Saving the values of these configuration attributes

Implementing the Strategy

Section 1.1.2.2 on page 1-6 provides an overview of how the output strategy works, and the steps required to implement it.

Section 1.3 on page 1-25 provides the details for implementing the output strategy.

1.1.2 Overviews of Implementing Configuration Attributes

See Section 1.1.1.3 on page 1-3 and Section 1.1.1.4 on page 1-3 for overviews of the use of these strategies.

1.1.2.1 Overview of Implementing Configuration Attributes for Input

See Section 1.2 on page 1-7 for the details on implementation.

How the Solution Works

A Model is specially modified to contain configuration attribute data. During the initialization of an end user configuration session with that Model, a Functional Companion retrieves the configuration attribute data from the host application's tables and inserts it into the Model so that it can be used during configuration session.

What You Do to Implement the Solution

To use configuration attributes for the input of data from a host application to Oracle Configurator, you must:

- If the host application is a custom application, modify the initialization message to Oracle Configurator to specify the source of the configuration attribute data. See Section 1.2.2 on page 1-9.
- Modify the structure of the configuration model so that it includes elements that can receive the data (such as numeric Features). You must also associate a Functional Companion with the structure. See Section 1.2.3 on page 1-13.

- Modify the example Functional Companion to pass the data from the host application to the locations that you have created in the modified model. See Section 1.2.4 on page 1-15.
- Deploy the Model and the Functional Companion. See Section 1.1.3 on page 1-6.

1.1.2.2 Overview of Implementing Configuration Attributes for Output

See Section 1.3 on page 1-25 for the details on implementation.

How the Solution Works

A configuration model is specially modified to contain configuration attribute data. After an end user configuration session with that model terminates, a Functional Companion captures the configuration attribute data from the model and inserts it in the table CZ_CONFIG_ATTRIBUTES. A custom procedure retrieves the data from this table and inserts it into descriptive flexfield segments, where it can be accessed by the Oracle Applications host.

What You Do to Implement the Solution

To use configuration attributes for the output of data from Oracle Configurator to a host application, you must:

- Understand the structure of the table CZ_CONFIG_ATTRIBUTES. See Section 1.3.1 on page 1-27.
- Modify Oracle Applications to add flexfield definitions. See Section 1.3.2 on page 1-29.
- Modify the configuration model to capture configuration attribute data. See Section 1.3.3 on page 1-30.
- Possibly modify the example Functional Companion to match changes to the configuration model. See Section 1.3.4 on page 1-42.
- Deploy the Model and the Functional Companion. See Section 1.1.3 on page 1-6.
- Write a custom procedure and customize the downstream application so that it can use the configuration attribute data stored in CZ_CONFIG_ATTRIBUTES. See Section 1.3.5 on page 1-43.

1.1.3 Deploying the Solution

The tasks for deploying your configuration attributes solution are the same for both input and output.

After you have modified the host application, the Model, and the Functional Companion, you must:

1. Using Oracle Configurator Developer, generate the Active Model and unit-test it with the Test module. See the *Oracle Configurator Developer User's Guide* for details.
2. Publish the modified Model. See *Oracle Configurator Implementation Guide* for information on publishing.
3. Compile the Functional Companion, as described in the *Oracle Configuration Interface Object (CIO) Developer's Guide*.
4. Install the Functional Companion, as described in the *Oracle Configurator Installation Guide*. This involves placing the Java class file for the Functional Companion in the class path of the Oracle Configurator Servlet, updating the OC Servlet's configuration files, and restarting the OC Servlet.

1.2 Implementing Configuration Attributes for Input

This section describes the details for implementing a configuration attributes input solution.

See Section 1.1.2.2, "Overview of Implementing Configuration Attributes for Output" on page 1-5 for an overview.

See the following sections for details on the tasks required for implementing configuration attributes for input:

- Section 1.2.1, "Example for Implementing Input Configuration Attributes" on page 1-7.
- Section 1.2.2, "Host Application Integration with Oracle Configurator" on page 1-9.
- Section 1.2.3, "Modifying the Model" on page 1-13.
- Section 1.2.4, "Modifying the Functional Companion Example" on page 1-15.
- Section 1.1.3, "Deploying the Solution" on page 1-6.

1.2.1 Example for Implementing Input Configuration Attributes

The methodology for implementing input configuration attributes is illustrated throughout Section 1.2 by use of a simple hypothetical example. This section describes that example.

Note: This chapter, Chapter 1, "Configuration Attributes", discusses methodological examples of implementing configuration attributes, and provides examples of Functional Companion coding. The term *example* is used for both.

Assumptions for the Example

Assume that you have the following requirements:

- The product that you are selling through your host application must be configured partly on the basis of the U.S. state to which it is shipped.
- You want to use Oracle Configurator to decide which version of your product must be selected, based on the U.S. state.
- The host application is Oracle Quoting.
- For a given quote line, you want to capture the identity of the U.S. state for the customer associated with the quote, so that you can pass it to Oracle Configurator. Assume that the address is associated with the order header.

Methodology for the Example

The example uses the following method to fulfill your requirements:

1. The host application (Oracle Quoting) calls the runtime Oracle Configurator when the end user clicks the Configure button. The initialization message posted from *Oracle Quoting* to OC includes a parameter that identifies the current quote line.
2. Oracle Configurator creates a new configuration. In doing so, it triggers the execution of the `onNew()` method of the Functional Companion associated with the configuration model.
3. The Functional Companion uses a custom query to extract the data associated with quote line, using the quote header to access the U.S. state for the customer's address.
4. The Functional Companion processes the result set returned by the query to isolate the value for the U.S. state name.
5. The Functional Companion stores the value of the U.S. state name in a local variable.
6. The configuration model includes a Feature named `US_State` dedicated to storing the name of a U.S. state, so that state-specific configuration rules can be

written against it. Each Option of this Feature is named for a U.S. state, in a way that matches the state name stored in the customer record used by the host application (that is, with the state's two-letter postal abbreviation).

7. The Functional Companion uses a custom method to locate the Feature named `US_State`. The search starts from the node with which the Functional Companion has been previously associated in Configurator Developer.
8. The Functional Companion uses the same custom method to locate the Option whose name matches the U.S. state name that was extracted from the database.
9. The Functional Companion selects that particular Option. Now, any configuration rules written against that Option will be invoked or set or applied.

1.2.2 Host Application Integration with Oracle Configurator

The responsibilities for this task depend on the integration role:

- **Custom Application**: If you are integrating Oracle Configurator with a custom host application, see Section 1.2.2.1, "Responsibilities For Custom Host Application" on page 1-9.
- **Oracle Applications**: If you are integrating Oracle Configurator with Oracle Applications, see Section 1.2.2.2, "Responsibilities For Oracle Applications Application Host" on page 1-11.
- **Functional Companion**: If you are implementing the Functional Companion used for input configuration attributes, see Section 1.2.2.3, "Responsibilities For Functional Companion Implementer" on page 1-12.

See Section 1.2.1, "Example for Implementing Input Configuration Attributes" on page 1-7 for background.

1.2.2.1 Responsibilities For Custom Host Application

A custom host application must pass to Oracle Configurator the information that identifies the line item that is associated with the desired configuration attribute data.

The mechanism for passing this identification information is a set of parameters in the initialization message that is sent by the Oracle Applications host to the Oracle Configurator Servlet.

- The choice of parameters is described in Specifying Initialization Parameters for Input Configuration Attributes on page 1-10.

- The parameters themselves are described in Initialization Parameter Descriptions on page 1-10.
- The task of adding the parameters is described in Adding the Parameters to the Initialization Message on page 1-11.

Specifying Initialization Parameters for Input Configuration Attributes

The set of parameters that must be added to the initialization message depends on the requirements of the host application.

The host application must always specify its identity, with the parameter `calling_application_id` (see calling_application_id on page 1-10).

In addition to specifying its identity, the host application must identify the line item that is associated with the desired configuration attribute data. This identification should be performed using as many of the following parameters as necessary:

- client_header
- client_line
- client_line_detail

Initialization Parameter Descriptions

The values of these parameters depend on the host application. For some examples, see Section 1.2.2.2, "Responsibilities For Oracle Applications Application Host" on page 1-11.

calling_application_id

Use this parameter to identify the Oracle Applications host that is invoking Oracle Configurator. The value of the ID for your application is obtained from FND_APPLICATION.APPLICATION_ID. (See the description of `calling_application_id` in the *Oracle Configurator Implementation Guide* for other information about this parameter.) See Section 1.2.4.4, "Identifying the Host Application" on page 1-19 for information about determining the value to use for this parameter.

client_header

Use this parameter to specify the unit of work for the application (for example, an order or quote).

client_line

The particular part of the order or quote that the configuration is initiated against.

client_line_detail
This parameter is used to provide additional information if client_line does not provide enough.

Adding the Parameters to the Initialization Message

Example 1–1 on page 1-11 shows sample parameters for identifying a line item in the initialization message sent by a host application to the runtime Oracle Configurator. The parameters not related to configuration attributes are omitted from this example. The values for `client_header` and `client_line` are hypothetical; the value for `calling_application_id` applies to the Order Management application.

Example 1–1 Configuration Attribute Parameters in the Initialization Message

```
<initialize>
...
<param name="calling_application_id">660</param>
...
<param name="client_header">12345</param>
<param name="client_line">1000</param>
...
</initialize>
```

For general background and specific details on initialization parameters and the initialization message, see the *Oracle Configurator Implementation Guide*.

1.2.2.2 Responsibilities For Oracle Applications Application Host

Some Oracle Applications support the use of configuration attributes in their initialization message. See the Release Notes for a particular application to see whether it does.

Oracle Applications use the same set of possible initialization parameters as custom host applications. See Specifying Initialization Parameters for Input Configuration Attributes on page 1-10.

For the sake of example, Table 1–1, Table 1–2, and Table 1–3 list the potential sources, in certain host Oracle Applications, for the data that can be provided for the parameters client_header, client_line, and client_line_detail.

Table 1–1 Potential Sources for Values of the client_header Parameter

Application	Source for Parameter Data
Oracle Order Management (ONT)	OE_ORDER_LINES_ALL.HEADER_ID
Oracle Quoting (ASO)	Probably not used.
Oracle Contracts Core (OKC)	Probably not used.

Table 1–2 Potential Sources for Values of the client_line Parameter

Application	Source for Parameter Data
Oracle Order Management (ONT)	OE_ORDER_LINES_ALL.LINE_ID
Oracle Quoting (ASO)	ASO_QUOTE_LINE.QUOTE_LINE_ID
Oracle Contracts Core (OKC)	OKC_K_LINES.ID

Table 1–3 Potential Sources for Values of the client_line_detail Parameter

Application	Source for Parameter Data
Oracle Order Management (ONT)	Not used.
Oracle Quoting (ASO)	ASO_QUOTE_LINE.LINE_NUMBER
Oracle Contracts Core (OKC)	OKC_K_LINES.LINE_NUMBER

1.2.2.3 Responsibilities For Functional Companion Implementer

When you modify the example Functional Companion (Example A–1 on page A-2):

- You must know which of the configuration attribute initialization parameters are being passed in the initialization message. The recommended parameters are described in Specifying Initialization Parameters for Input Configuration Attributes on page 1-10.
- You must get the names and values of the passed parameters, including `calling_application_id`, as described in Section 1.2.4.3 on page 1-18.
- You must write a query that extracts the desired data from the tables associated with the quote or order line, as described in Section 1.2.4.5 on page 1-20.
- See Section 1.2.4 on page 1-15 for details on the rest of the tasks required when modifying the Functional Companion.

1.2.3 Modifying the Model

See Section 1.2.1, "Example for Implementing Input Configuration Attributes" on page 1-7 for background.

To create configuration rules that operate on the input configuration attribute data, your Model must include some additional structure that receives the data when it is transferred to the runtime Model by your Functional Companion.

To create this structure, you use Oracle Configurator Developer, as described in Section 1.2.3.1 and following. See the *Oracle Configurator Developer User's Guide* for details on creating structure.

1.2.3.1 Creating Attribute Features

In order to write configuration rules that involve configuration attribute data, the attributes are represented in the configuration model by Features. In the solution described here, these Features are called **attribute Features**. The value of each configuration attribute is stored in an attribute Feature. Each attribute Feature corresponds, more or less, to a field in the host application's database.

Attribute Features are not a new type of Feature. The term attribute Feature simply means a Feature that is used to contain the data for a configuration attribute.

To work with the example, create a Feature named `US_State`, with a **Type** set to List of Options, and **Minimum** and **Maximum** both set to 1.

The name of the Feature must agree exactly with the name that you specify for the `findFirstNodeByName()` method of the Functional Companion, since the Feature is located by a case-sensitive string search on each character of the name. See Section 1.2.4.6, "Transferring Data to Features" on page 1-21.

You can use this Feature as a participant in a configuration rule that specifies a particular U.S. state.

1.2.3.2 Creating Options

The name of each Option must match the data value stored in the customer record used by the host application.

To work with the example, create a list of Options for the Feature named `US_State`. Set the **Name** of each Option to be the postal abbreviation for a U.S. state (for example, CA, NY, MA, RI).

The name of each Option must agree exactly with one of the values retrieved from the host application's database by the custom query in the Functional Companion.

The retrieved value is passed as an argument to the `findFirstNodeByName()` method of the Functional Companion,which performs a case-sensitive string search on each character of the name. See Section 1.2.4.6, "Transferring Data to Features" on page 1-21.

You can use this Option as a participant in a configuration rule that specifies a particular U.S. state.

1.2.3.3 Creating Functional Companion Rules

The Functional Companion must be associated with the node in the Model's structure that is the starting point for locating the Feature that contains the configuration attribute Options.

To work with the example, create a Functional Companion rule and associate it with a parent node of the Feature named `US_State`. Choose a node that is certain to contain the Feature, so that the search for it by the `findFirstNodeByName()` method of the Functional Companion will be successful and efficient. Consider that choosing a node high in the Model's hierarchical structure will result in a longer search, and may result in locating the wrong Feature if there is another one with the same name.

Set the **Definition** of the Functional Companion rule as shown in the following table:

Rule Definition Attribute	Definition Value
Base Component	The node from which the Functional Companion will search for the specified Feature.
Type	Event-Driven
Implementation	Java
Program String	The name of the Java class that implements the input Functional Companion. For the example, this is `CfgInputExample` (Example A–1 on page A-2).

For more details on defining Functional Companion rules, see the *Oracle Configuration Interface Object (CIO) Developer's Guide*. For details on defining other kinds of configuration rules, see the *Oracle Configurator Developer User's Guide*.

1.2.4 Modifying the Functional Companion Example

For the complete Java code example that supports this section, see Example A–1 on page A-2, in Section A.1, which demonstrates a Functional Companion that uses configuration attributes for input to a configuration model

This section includes excerpts from Example A–1.

Caution: Example A–1 demonstrates a Functional Companion that uses configuration attributes for input to a configuration model. This example is only a template for your own solution to the problem; *you must modify the code in the example* to suit your own configuration model and host application.

For a summary of the flow of data and control in the Functional Companion example, see Methodology for the Example under Section 1.2.1, "Example for Implementing Input Configuration Attributes" on page 1-7.

When you customize the Functional Companion, keep in mind that it should only perform the function of transferring data into your configuration model. For simplicity and maintainability, you should not perform other operations in this Functional Companion.

This Functional Companion example is customized to work with *Oracle Quoting*. The customizations are identified and explained where they occur in the following sections, so that you can customize your Functional Companion for your own host application.

Note: Be sure to check the *Oracle Configuration Interface Object (CIO) Developer's Guide* for a description of the Java development skills required for success with Functional Companions.

1.2.4.1 Implementing AutoFunctionalCompanion Methods

For your Functional Companion to be run automatically when the host application invokes the runtime Oracle Configurator, you must extend the class `AutoFunctionalCompanion`, as shown in the following code fragment.

```
public class CfgInputExample extends AutoFunctionalCompanion {
    // body of class defined here
...
}
```

You must also override the `onNew()` and `onRestore()` methods of `AutoFunctionalCompanion`, to provide functionality when the configuration associated with the Functional Companion is either created, or restored, respectively. The following code fragment shows the structure of this overriding.

```
public void onNew() throws LogicalException {
   // functionality of method defined here
   doInputAttributeTransfer();
 }

public void onRestore() throws LogicalException {
   // functionality of method defined here
   doInputAttributeTransfer();
 }
```

Your Functional Companion can define functionality for both methods. The appropriate method will be triggered by the runtime Oracle Configurator when the configuration associated with the Functional Companion is either created or restored.

The method `doInputAttributeTransfer()` is part of the example, and is explained in Section 1.2.4.2, "Structuring the Behavior" on page 1-17.

You should be careful in deciding what functionality to define in `onNew()`, in `onRestore()`, or in both.

- If you use only `onNew()`, then the input configuration attribute data is retrieved only once, when the configuration is created. You should do this if you want the configuration attribute data to remain the same throughout the life of the configuration.

  If you want to ensure that the original input data is never changed, and to specify an unchangeable set of logical assertions against that data, you should use initial requests (which are described in the *Oracle Configuration Interface Object (CIO) Developer's Guide*).

- If you use only `onRestore()`, then the input configuration attribute data is retrieved only whenever the saved configuration is restored from the database. It is unlikely that you would want to use only `onRestore()`, since the result would be that no configuration attribute data would be input into your configuration model when a configuration is created.

- If you use both `onNew()` and `onRestore()`, then the input configuration attribute data is retrieved when the configuration is created and also whenever

it is restored. (Note that if you used initial requests in `onNew()`, then `onRestore()` will not be able to change those values.)

The effect is that the current source data in the host application's database is always put into your configuration model. However, be aware that if the host application has changed the source data since the configuration was created, that changed data will be put into your configuration model, and may produce different results when it interacts with your configuration rules.

1.2.4.2 Structuring the Behavior

The main tasks that must be performed by the Functional Companion are:

- Getting the session parameters from the initialization message. Section 1.2.4.3 on page 1-18.
- Identifying the host application. See Section 1.2.4.4 on page 1-19.
- Extracting the input configuration attribute data for the quote (or order) line associated with the configuration. See Section 1.2.4.5 on page 1-20.
- Transferring the configuration attribute data to the configuration model. See Section 1.2.4.6 on page 1-21.

In the example, the methods for performing these tasks are gathered together for convenience in the method `doInputAttributeTransfer()`, as shown in the following code fragment.

```
private void doInputAttributeTransfer() throws LogicalException {

// Get the session parameters
  getSessionParameters();
  if (mOrderLineNumber == null) return;

// Identify the host application
  if (mApplicationId.longValue() != 697) return;

// Specify the quote/order line and extract the attribute data
  getInputAttributes();

// Transfer the attribute data to the configuration model
  transferInputAttributesIntoModel();
}
```

See Section 1.2.4.4, "Identifying the Host Application" on page 1-19 for background on the value of `mApplicationId`.

1.2.4.3 Getting Session Parameters

The Functional Companion must extract the parameter values it needs from the initialization message posted to Oracle Configurator by the host application. The parameters used for obtaining configuration attribute data are described in Specifying Initialization Parameters for Input Configuration Attributes on page 1-10.

The example defines the method `getSessionParameters()`, as shown in the following code fragment.

```
private void getSessionParameters() {

  // 1. Get the parameters in the initialization message
  NameValuePairSet initParams =
getRuntimeNode().getConfiguration().getInitParameters();
String paramValue = null;

  // 2. Get the value of the parameter "client_line"
  paramValue = (String)initParams.getValueByName("client_line");

  // 3. Store the parameter value
  if (paramValue != null) mOrderLineNumber = Long.valueOf(paramValue);

  // 4. Get and store the value of the parameter "calling_application_id"
  paramValue = (String)initParams.getValueByName("calling_application_id");
  if (paramValue != null) mApplicationId = Long.valueOf(paramValue);
  }
```

The method `getSessionParameters()` performs the following tasks:

1. Gets the parameters of the initialization message, with `Configuration.getInitParameters()`.
2. Gets the value of the parameter that identifies the quote line or order line (`client_line`), using `NameValuePairSet.getValueByName()`. You may need to customize this to extract other parameter values.
3. Stores the value of the parameter in the variable `mOrderLineNumber`, if it is not null. You may have to change the data type of the variable depending on the type of the initialization parameter data being passed.
4. Gets the value of the parameter that identifies the host application (`calling_application_id`), using `NameValuePairSet.getValueByName()`. If it is not null, store it in the variable `mApplicationId`.

1.2.4.4 Identifying the Host Application

The Functional Companion must check the identity of the host application that posted the initialization message to the runtime Oracle Configurator. This allows you to process the initialization parameters and configuration attribute data differently, depending on the host application.

The Oracle Applications host is supposed to pass the identity of the application in the initialization parameter `calling_application_id`.

If your host application is not part of Oracle Applications, then you must design your own mechanism for identifying the host application to the Functional Companion.

The following test from the example's `doInputAttributeTransfer()` method compares the `mApplicationId` obtained from `getSessionParameters()`to the application ID for Oracle Quoting, and returns immediately out of the method if it does not match.

```
if (mApplicationId.longValue() != 697) return;
```

The following test from the example's `getInputAttributes()` method checks whether the `mApplicationId` matches the application ID for *Oracle Quoting*, and proceeds if it does.

```
if (mApplicationId.longValue() == 697) {
...
}
```

The SQL query in Example 1–2 on page 1-19 lists the IDs, short names, and long names for Oracle Applications. You can examine the results of the query to determine the application ID for your own host application.

Example 1–2 Query for Names of Oracle Applications

```
SELECT SUBSTR(application_short_name,1,10) sName,
       SUBSTR(application_name,1,32) application,
       application_id as id
FROM   fnd_application_vl
ORDER BY application_short_name;
```

The IDs and short names for hosting Oracle Applications are listed in the *Oracle Configurator Release Notes*.

1.2.4.5 Extracting Input Attribute Data for the Specified Quote Line

The Functional Companion extracts the configuration attribute data from the host application's database by means of a customized SQL query that specifies the `mOrderLineNumber` that was obtained by `getSessionParameters()` (see Section 1.2.4.3 on page 1-18). To perform this data extraction, the example defines the method `getInputAttributes()`, as shown in the following fragment.

```
private void getInputAttributes() {

  Connection conn =
getRuntimeNode().getConfiguration().getContext().getJDBCConnection();
  PreparedStatement pStmt = null;
  ResultSet rs;
  int ret;
...
// Define a custom query to extract the U.S. state for a quote line
        String sql = "select p.state "
                   + "from   hz_parties p, "
                   + "       aso_quote_headers_all_v q, "
                   + "       ASO_QUOTE_LINES_ALL_V l "
                   + "where  p.party_id = q.cust_account_id "
                   + "and    l.quote_header_id = q.quote_header_id "
                   + "and    l.quote_line_id = ?";

        pStmt = conn.prepareStatement(sql);

// Put the line number into the first (and only) parameter
// of the PreparedStatement, for quote_line_id
          pStmt.setLong(1, mOrderLineNumber.longValue());

// Execute the query, putting the results in a ResultSet
          rs = pStmt.executeQuery();
// Iterate through the ResultSet and get the first value
...
            mUsState = rs.getString(1);
...
}
```

Note: The code fragment shown here omits error handling, for clarity. The full coding of the example for `getInputAttributes()` includes error handling for the query and processing of the results. Be sure to examine Example A–1 on page A-2.

1.2.4.6 Transferring Data to Features

At this point, the Functional Companion has obtained the configuration attribute data (the abbreviation for a U.S. state), and stored it in a local variable (`mUsState`). Now the value must be transferred into a Feature of your configuration model (named `US_State`) that is dedicated to storing the configuration attribute data (the abbreviation of the U.S. state).

To perform this transfer, the example first defines a general transfer method `transferInputAttributesIntoModel()`, which is called in `doInputAttributeTransfer()`. This general method in turn calls a more specific transfer method, `transferUsState()`, which you must customize or replace, to transfer the configuration attribute data to the dedicated Feature of your configuration model.

This arrangement, which is intended to provide modularity and ease of customization, is shown in the following fragment.

```
private void transferInputAttributesIntoModel() throws LogicalException {
  transferUsState();
}

private void transferUsState()throws LogicalException {

  if (mUsState == null) return;

// Find the Feature dedicated to the attribute data
IRuntimeNodertNode=findFirstNodeByName(getRuntimeNode(),"US_State");

  if (rtNode == null) return;

  if (rtNode.getType() != IRuntimeNode.OPTION_FEATURE) return;

// Use the attribute value to find the Option with the matching name
  IOption option = (IOption)findFirstNodeByName(rtNode, mUsState);
  if (option == null) return;
...
```

```
// Select the option to set its value
  option.select();
...
}
```

Starting from the runtime node to which it has been previously associated, the transfer method uses a custom method (`findFirstNodeByName()`) to locate the Feature named `US_State`.

The transfer method then uses `findFirstNodeByName()` to locate the Option whose name matches the U.S. state name that was extracted from the database.

Finally, the transfer method selects that particular Option. Now, any configuration rules written against that Option will be able to operate on it.

The utility method `findFirstNodeByName()` is defined in the full example. See Example A–1 on page A-2 for the definition of this method.

Note: The code fragment shown here omits some error handling, for simplicity. Be sure to examine Example A–1 on page A-2.

1.2.4.7 Guidelines for the Functional Companion

This section lists guidelines for writing a Functional Companion for input configuration attributes. See the preceding subsections of Section 1.2.4 for illustrations of these guidelines.

General

- The sole purpose of the Functional Companion should be to transfer configuration attribute data from a host application to the runtime Oracle Configurator.

  Do not perform any calculations on the configuration attribute data in the Functional Companion. Instead, perform the calculations through the structure and rules of the configuration model. This practice greatly improves the maintainability of the configuration rules, since the rules can be modified by using Oracle Configurator Developer, instead of by programming.

- Assign only one Functional Companion that implements `AutoFunctionalCompanion.onNew()` for configuration attribute transfer per configuration model.

  The same guideline applies to a Functional Companion that implements `AutoFunctionalCompanion.onRestore()`.

- Design the Functional Companion so that it works with all of your relevant host applications that include configuration models. The example demonstrates how to test for different applications.
- The Functional Companion is not able to return messages resulting from exceptions, since an exception (which produces a message) returned from any Functional Companion that runs during the initialization of the runtime Oracle Configurator is treated as a fatal exception. The end-user session is terminated, displaying the message thrown from the exception. Consequently, structure the configuration model so that validation failures, rather than exceptions, will be produced when configuration attribute values have changed between sessions (assuming that you want your end user to know about this change during the configuration session). See the *Oracle Configuration Interface Object (CIO) Developer's Guide* for background on validating configurations.
- You can write a single Functional Companion that focuses on a single configuration attribute, and it will work with all configuration models that are modified to include that configuration attribute's data. You do not have to create a separate Functional Companion for each of these configuration models. This is true because the Functional Companion is written to search for the Feature that corresponds to the configuration attribute, regardless of where it is placed in a Model.
- Because the example uses the `findFirstNodeByName()` method to locate the attribute Feature, you cannot write a single Functional Companion to provide input to more than one configuration attribute.
- For simplicity, the example Functional Companion is written to provide data to one attribute Feature. By implementing separate blocks of code, you can modify the Functional Companion so that it provides input data to multiple attribute Feature.

When Implementing the onNew() Method

See Section 1.2.4.1, "Implementing AutoFunctionalCompanion Methods" on page 1-15 for coverage of the `onNew()` method.

- If your Functional Companion's custom query to the host application's database does not find a value for the configuration attribute, then the Functional Companion should not attempt to provide a default value. Instead, your configuration model should include a Defaults rule that provides the default value. See the *Oracle Configurator Developer User's Guide* for details on Defaults rules. The Functional Companion should report an error, as is shown in the full example (Example A–1 on page A-2).

When Implementing the onRestore() Method

See Section 1.2.4.1, "Implementing AutoFunctionalCompanion Methods" on page 1-15 for coverage of the `onRestore()` method.

- Not all previously saved attributes need to be loaded. For example, if a Feature value is calculated from several configuration attributes, then only the inputs to the calculation need to be restored. It may also be that these inputs may be the values of BOM items, which don't need to be restored through the Functional Companion.
- You must decide when to update configuration attribute data: whether to restore it from the saved configuration or by extracting it again from the host application. Getting the most current value from the host application is usually the best course. However, if you don't want the most current value from the host application, then the value is already restored, and you don't need to take further action.
- You must consider the host application domain when restoring a configuration into a different host application. For example, when the configuration is saved in Oracle Quoting, submitted to Order Management, and later is restored in Order Management (with batch validation), there can be different treatment of an input configuration attribute, but only if you code your Functional Companion to do that.

1.2.5 Runtime Behavior

See Methodology for the Example under Section 1.2.1, "Example for Implementing Input Configuration Attributes" on page 1-7 for a summary of the behavior of the host application and Oracle Configurator at runtime.

Interactive Configuration Session Scenario

When the runtime Oracle Configurator is invoked by the host application, the user interface is loaded and presented to the user. If there are any validation failures resulting from creating the configuration, they are displayed to the user. The session then waits to interact with the user.

Any errors other than validation failures cause the session to terminate.

Batch Validation Scenario

When the host application performs batch validation (for example, when Order Management books the order), any validation failures are recorded into the CZ_CONFIG_MESSAGES table. Then each user input, if any, is set by Order

Management. All contradictions are recorded in CZ_CONFIG_MESSAGES. Overridable contradictions are overridden.

Any errors other than validation failures cause the session to terminate.

1.3 Implementing Configuration Attributes for Output

This section describes the details for implementing a configuration attributes output solution.

See Section 1.1.2.2 on page 1-6 for an overview.

See the following sections for details on the tasks required for implementing configuration attributes for output:

- Section 1.3.1, "Database Tables for Configuration Attributes" on page 1-27.
- Section 1.3.2, "Defining Descriptive Flexfields" on page 1-29.
- Section 1.3.3, "Modifying the Model" on page 1-30.
- Section 1.3.4, "The Output Functional Companion" on page 1-42.
- Section 1.1.3, "Deploying the Solution" on page 1-6.
- Section 1.3.5, "Using Configuration Attributes in the Downstream Application" on page 1-43.

Recommended Control and Data Flows

The control and data flow at design time is shown in Figure 1–1 on page 1-26.

The control and data flow at runtime is shown in Figure 1–2 on page 1-27.

Note: The flows shown here illustrate the methodology recommended in this chapter for Oracle Applications. If you are integrating with a custom host application, then you must make the appropriate adjustments to your flow, and also to the Functional Companion used to help implement it.

Figure 1–1 Control and Data Flow at Design Time for Output

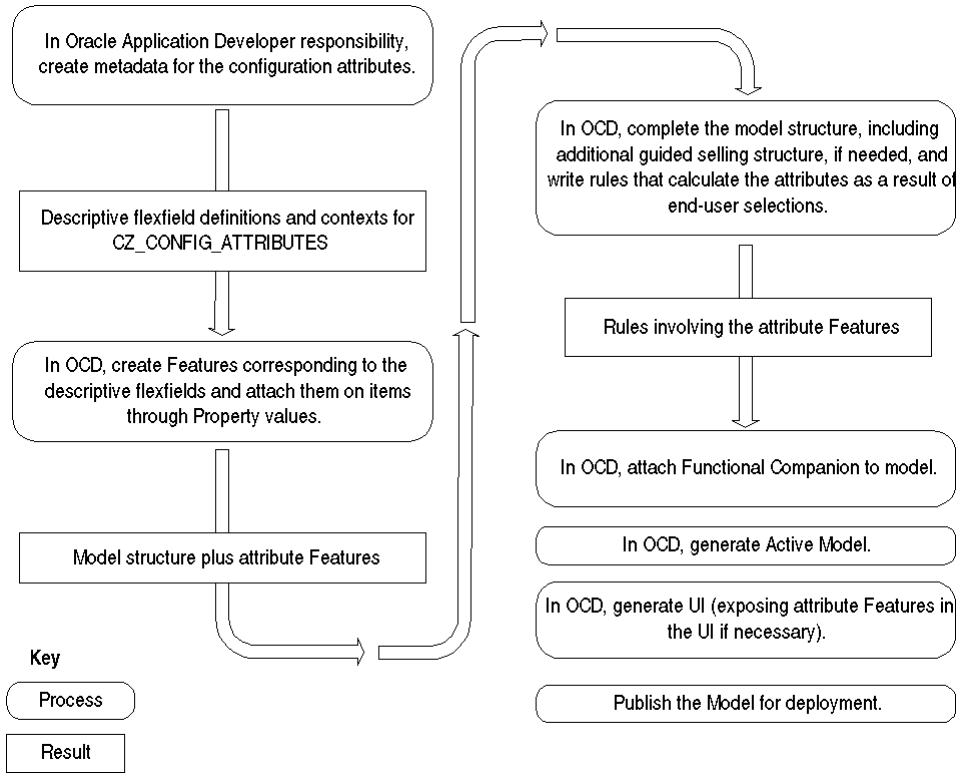
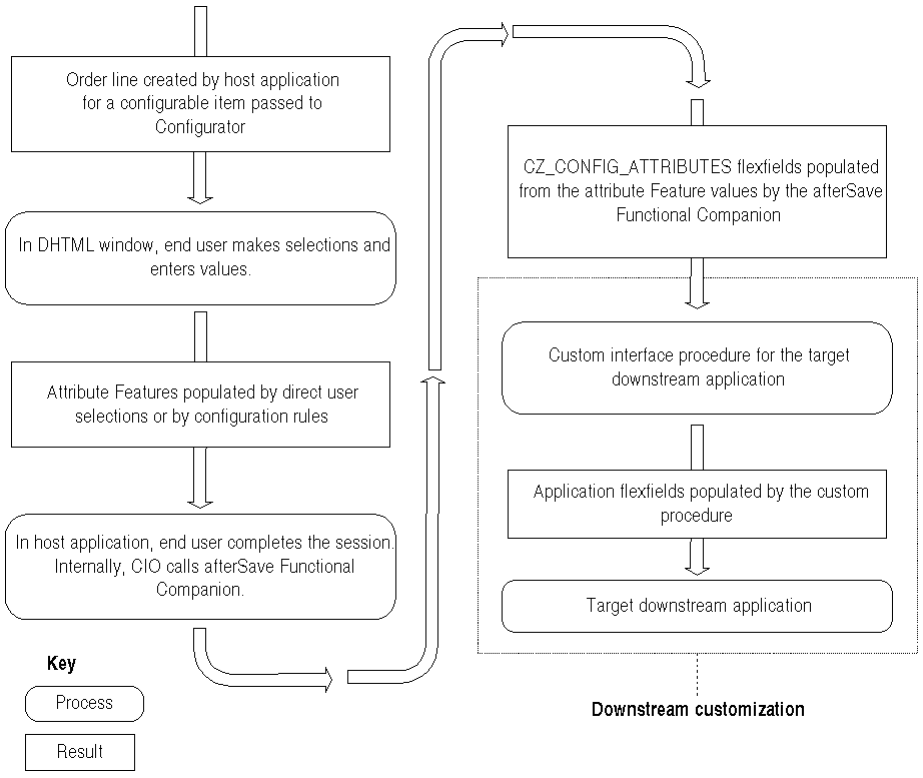


Figure 1–2 Control and Data Flow at Run Time for Output



1.3.1 Database Tables for Configuration Attributes

Certain tables in the Oracle Applications database are used to hold configuration attribute data written by the runtime Oracle Configurator, so that it can be used by downstream applications. For downstream applications, Oracle Configurator uses the table CZ_CONFIG_ATTRIBUTES. See Section 1.3.1.1 on page 1-27.

1.3.1.1 The CZ_CONFIG_ATTRIBUTES Table

The table CZ_CONFIG_ATTRIBUTES is used as the intermediate store of configuration attribute data between Oracle Configurator and the host application. The runtime Oracle Configurator writes output data to the table by means of the Functional Companion described in Section 1.3.4 on page 1-42. The data is available to a host application by means of the descriptive flexfields you define, as described

in Section 1.3.2 on page 1-29. A custom procedure (which you must write) reads the data from CZ_CONFIG_ATTRIBUTES and inserts it into the tables used by a specific downstream application.

The layout and description of the CZ_CONFIG_ATTRIBUTES table is provided in Table 1–4.

Table 1–4 The CZ_CONFIG_ATTRIBUTES Table

Column Name	Null?	PK?	Type	Comments
CONFIG_HDR_ID	N	Y	NUMBER(9)	Configuration header ID
CONFIG_REV_NBR	N	Y	NUMBER(9)	Configuration revision number
CONFIG_ITEM_ID	N	Y	NUMBER(9)	Configuration item ID
ATTRIBUTE_CATEGORY	N	Y	VARCHAR2(30)	Name of the flexfield context
ATTRIBUTE1	Y	N	VARCHAR2(240)	Flexfield segment value for the specified context
ATTRIBUTE2	Y	N	VARCHAR2(240)	Flexfield segment value for the specified context
...	...	...	...	...
ATTRIBUTE30	Y	N	VARCHAR2(240)	Flexfield segment value for the specified context

The columns CONFIG_HDR_ID, CONFIG_REV_NBR, CONFIG_ITEM_ID and ATTRIBUTE_CATEGORY constitute the primary key for CZ_CONFIG_ ATTRIBUTES.

All of the columns ATTRIBUTE1 through ATTRIBUTE30 are defined identically, and the columns between ATTRIBUTE2 and ATTRIBUTE30 have been omitted from Table 1–4 for brevity.

The standard columns such as LAST_UPDATE_DATE have also been omitted for brevity.

1.3.1.2 General Information about Tables

For information about the Oracle Configurator schema, see the *Oracle Configurator Implementation Guide*. For technical details about CZ_CONFIG_ATTRIBUTES, CZ_ CONFIG_EXT_ATTRIBUTES, and other tables, see the Configurator eTRM on Metalink, Oracle's technical support Web site.

1.3.2 Defining Descriptive Flexfields

Configuration attributes are provided by defining descriptive flexfield segments that are used in conjunction with attribute Features. See Section 1.3.3.1, "Design Principles" on page 1-30 for an explanation of attribute Features.

To define the metadata for the flexfields, log into Oracle Applications with the Oracle Application Developer responsibility. Metadata describes how data is organized. See the *Oracle Applications Flexfields Guide* for details on defining descriptive flexfields.

When you create the flexfields, use the settings shown in Table 1–5 for all contexts that you define.

Table 1–5 Flexfield Settings for All Contexts

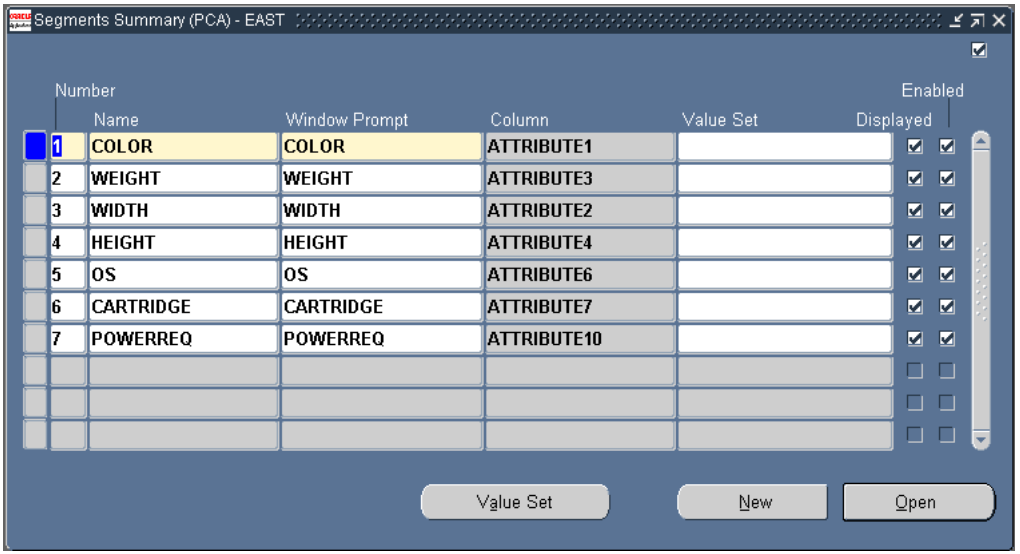
Setting	Value
Application	Oracle Configurator
Table Application	Oracle Configurator
Table Name	CZ_CONFIG_ATTRIBUTES

When you create the flexfields, you define whatever contexts are required by the host application. Then, for each segment in a context, you need to specify the ATTRIBUTE*n* column in CZ_CONFIG_ATTRIBUTES in which that segment's data will be allocated (see Table 1–4 on page 1-28 for the available column names).

Example of Flexfield Definition

For a context named EAST, define that the segment named COLOR stores its data in ATTRIBUTE1 of the designated table (CZ_CONFIG_ATTRIBUTES), that a segment named WEIGHT stores its data in ATTRIBUTE2, and so on. Figure 1–3 on page 1-30 shows a summary of such segment definitions in Oracle Applications.

Figure 1–3 Segments Summary for a Context



Guidelines

If you want to replicate the attribute values as flexfields on a particular downstream application table (for example, MRP_FLOW_SCHEDULES), then it is good practice to match all the context and segment names as well as using the same column names for the CZ_CONFIG_ATTRIBUTES definitions. In other words, all flexfield definitions should be duplicated with the only difference being the Application and the Table Name.

1.3.3 Modifying the Model

To use configuration attributes, you must modify your configuration model as described in this section.

1.3.3.1 Design Principles

In order to write configuration rules that involve configuration attribute data, the attributes are represented in the configuration model by Features. In the solution described here, these Features are called **attribute Features**. The value of each configuration attribute is stored in an attribute Feature. Each attribute Feature corresponds to a descriptive flexfield segment (see Section 1.3.2, "Defining Descriptive Flexfields" on page 1-29).

Attribute Features are not a new type of Feature. The term attribute Feature simply means a Feature that is used to contain the data for a configuration attribute.

The data types of the attribute Features must match the data types of the corresponding descriptive flexfields: an Integer Feature for an integer flexfield, an Option Feature for a LOV flexfield, and so on.

Caution: If you set the **Maximum** of any attribute Option Feature to a value greater than 1, then you must modify the example Functional Companion (Example A–2) to interpret the end user's selection into a single attribute assignment.

To control the flow of configuration attribute data from the Model to the CZ_CONFIG_ATTRIBUTES table (and subsequently to the descriptive flexfields used by a downstream application), you define Properties on nodes in the Model structure:

- To assign an attribute Feature to the corresponding descriptive flexfield segment, you must define a Property on the Feature that contains the name of the flexfield segment. This Property should be called ATTR_NAME.
- To assign an attribute Feature to the correct flexfield context for the corresponding segment, you must define a Property on the Feature that contains the name of the flexfield context. This Property should be called ATTR_CONTEXT.
- To assign the configuration attribute for an Item to the attribute Feature that stores the attribute's value, you must define a Property on the Item that defines the node path to the attribute Feature in the Model. This Property should be called ATTR_*n*_PATH, where *n* is an integer that makes the name unique within the scope of the current node. You can create multiple ATTR_*n*_PATH Properties for the same node, with different values of *n*, to assign multiple attribute Features to the node. This means that you can assign multiple configuration attributes to the same Item. You can use the same ATTR_*n*_PATH names in different nodes without conflict.
- To optionally propagate the value of a configuration attribute to the selected children of an Item, you can define a Property on the Item that specifies a propagation mode. This Property should be called ATTR_MODE.
- To optionally override, for a particular Item, the default flexfield segment name, context, or mode specified by ATTR_NAME, ATTR_CONTEXT, or ATTR_MODE, you can define Properties on the Item called ATTR_*n*_NAME, ATTR_*n*_

CONTEXT, or ATTR_*n*_MODE, respectively. In these Property names, *n* must match *n* in the ATTR_*n*_PATH Property that points to the attribute Feature whose segment name, context, or mode you are overriding.

Table 1–6 on page 1-32 provides details on the Properties used for defining configuration attributes.

If you need to use different names for the Properties, then you must modify the example Functional Companion, as described in Section 1.3.4 on page 1-42.

Table 1–6 Properties for Defining Configuration Attributes

Property Name	Type	Description
ATTR_NAME	Text	Default flexfield name that identifies the attribute Feature metadata. The value of the Property is the name of the corresponding segment in the descriptive flexfield. This Property is mandatory for the methodology to work. The ATTR_NAME values used in Example 1–6 on page 1-34 correspond to the attribute names in Example 1–4 on page 1-34.
ATTR_CONTEXT	Text	Default flexfield context that identifies the attribute Feature metadata. The value of the Property is the flexfield context of all the corresponding segments in the descriptive flexfield. This Property is mandatory for the methodology to work.
ATTR_*n*_PATH	Text	Flexfield assignment for the Item. The value of the Property is the node path to the attribute Feature that stores the configuration attribute value for the Item. The node path is relative to the Model that contains the Item. In naming this Property, use the integer indicated by *n* to make the name of the Property unique within the scope of the current node. You can define otherProperties with the same name in other nodes without conflict. This Property is mandatory for the methodology to work.

Table 1–6 (Cont.) Properties for Defining Configuration Attributes

Property Name	Type	Description
ATTR_MODE	Integer	Default propagation mode of the configuration attribute value. The value of the Property indicates the depth to which the value of the current Item is propagated in the Model. Propagation means that the Functional Companion writes the value of the configuration attribute for the current Item into the CZ_CONFIG_ATTRIBUTES table for all the Items specified by the propagation mode. The modes are: ■ 0 = Propagate the value only to the current Item. (This is the default mode, if this Property is omitted.) ■ 1 = Propagate the value to the current Item, and to its immediate selected children. ■ 2 = Propagate the value to the current Item, and to all its selected descendents.
ATTR_*n*_NAME	Text	Overriding flexfield name. A Property with this name overrides the default value specified by ATTR_NAME. Use ATTR_*n*_NAME to specify a different flexfield segment for the Item to which the Property ATTR_*n*_PATH is attached, using the same value of *n*.
ATTR_*n*_CONTEXT	Text	Overriding flexfield context. A Property with this name overrides the default value specified by ATTR_CONTEXT. Use ATTR_*n*_CONTEXT to specify a different flexfield context for the Item to which the Property ATTR_*n*_PATH is attached, using the same value of *n*.
ATTR_*n*_MODE	Integer	Overriding propagation mode. A Property with this name overrides the default value specified by ATTR_MODE. Use ATTR_*n*_MODE to specify a different flexfield context for the Item to which the Property ATTR_*n*_PATH is attached, using the same value of *n*.

1.3.3.2 Example of Model Structure

Example Structure

Assume the structure of the sample BOM Model shown in Example 1–3:

Example 1–3 Structure of Sample BOM Model

```
BOM-ATO1-Model
 |__BOM-OC-VINYL
 |   |__BOM-Item1
```

```
|   |__BOM-Item2
|__BOM-OC-WOOD
|   |__BOM-Item3
|   |__BOM-Item4
|__BOM-OC-HANDLE
    |__BOM-PTO-Item5
    |__BOM-PTO-Item6
```

Assume that you want to create the configuration attributes on that BOM Model shown in Example 1–4:

Example 1–4 Configuration Attributes for Sample BOM Model

```
BOM-ATO1-Model
        COMMON-ATTR-1
        COMMON-ATTR-2
BOM-OC-VINYL
        VINYL-ATTR-1
        VINYL-ATTR-2
BOM-OC-WOOD
        WOOD-ATTR-1
        WOOD-ATTR-2
```

Assume that you have defined the descriptive flexfield contexts and segments for these configuration attributes shown in Example 1–5:

Example 1–5 Flexfield Contexts and Segments for Sample BOM Model

```
COMMON
        COMMON-ATTRIB1
        COMMON-ATTRIB2
VINYL
        VINYL-ATTRIB1
        VINYL-ATTRIB2
WOOD
        WOOD-ATTRIB1
        WOOD-ATTRIB2
```

Given the preceding assumptions, you would use Oracle Configurator Developer to create model structure like that shown in Example 1–6:

Example 1–6 Configuration Model Structure for Sample BOM Model

```
BOM-ATO1-Model
             |              <ATTR_1_PATH="COMPONENT_ATTRIBUTES.Feature1">
```

```
|              <ATTR_2_PATH="COMPONENT_ATTRIBUTES.Feature2">
|              <ATTR_MODE=1>
|__BOM-OC-VINYL
|   |          <ATTR_1_PATH="COMPONENT_ATTRIBUTES.Feature3">
|   |          <ATTR_1_MODE=1>
|   |          <ATTR_2_PATH="COMPONENT_ATTRIBUTES.Feature4">
|   |__BOM-Item1
|   |__BOM-Item2
|__BOM-OC-WOOD
|   |          <ATTR_1_PATH="COMPONENT_ATTRIBUTES.Feature5">
|   |          <ATTR_2_PATH="COMPONENT_ATTRIBUTES.Feature6">
|   |__BOM-Item3
|   |__BOM-Item4
|__BOM-OC-HANDLE
|   |__BOM-PTO-Item5
|   |__BOM-PTO-Item6
|__COMPONENT-ATTRIBUTES
    |__Feature1
    |          <ATTR_CONTEXT="COMMON">
    |          <ATTR_NAME="COMMON-ATTRIB1">
    |__Feature2
    |          <ATTR_CONTEXT="COMMON">
    |          <ATTR_NAME="COMMON-ATTRIB2">
    |__Feature3
    |          <ATTR_CONTEXT="VINYL">
    |          <ATTR_NAME="VINYL-ATTRIB1">
    |__Feature4
    |          <ATTR_CONTEXT="VINYL">
    |          <ATTR_NAME="VINYL-ATTRIB2">
    |__Feature5
    |          <ATTR_CONTEXT="WOOD">
    |          <ATTR_NAME="WOOD-ATTRIB1">
    |__Feature6
               <ATTR_CONTEXT="WOOD">
               <ATTR_NAME="WOOD-ATTRIB2">
```

Explanation of Example

This explanation applies to Example 1–6 and the similar examples that follow in this section.

- Properties of the nodes in the Model are indicated by the following convention:

  <*property_name*="property_value">

The angle brackets (< >) are not part of the Property name; their purpose is to visually distinguish the Properties from the other elements of the Model structure. The quotation marks around *property_value* are not themselves part of the value; they indicate that the type of the Property value is Text.

- `Feature1` through `Feature6` are the attribute Features. These correspond to the descriptive flexfield segments that you would have defined. The ATTR_NAME values are the flexfield segment names shown in Example 1–5 on page 1-34.
- The Properties of the attribute Features shown in Example 1–6 are described in Table 1–6 on page 1-32.
- `COMPONENT-ATTRIBUTES` is a Component with Instances set to Minimum = 1 and Maximum = 1. Its purpose is to contain the attribute Features in a single location, for convenient access. This is an optional design technique.

Note: It is not necessary to group attribute Features in a Component, although that design is used in this example. The attribute Features can be located anywhere in the Model, as long as the ATTR_*n*_PATH Properties point to them. See Section 1.3.3.4 on page 1-40 for an explanation of the rules for placement of attribute Features.

Database Results

With reference to the explanation of the table CZ_CONFIG_ATTRIBUTES given in Section 1.3.1 on page 1-27, the design in Example 1–6 on page 1-34 would produce database records like those shown in Table 1–7 on page 1-37. For the sake of compactness, the combination of the columns CONFIG_HDR_ID, CONFIG_REV_NBR, and CONFIG_ITEM_ID is represented here by the column named **Item**, ATTRIBUTE_CATEGORY is represented here by the column named **Context**, and only a few of the ATTRIBUTE*n* columns are shown. The ATTRIBUTE*n* columns contain the name of the attribute Feature that contains their value, rather than the value itself.

Table 1–7 Example Attribute Properties (Database Results)

Item	Context	Attribute1	Attribute2	Attribute3	Attribute4
BOM-ATO1-Model	COMMON	(Feature1)	(Feature2)		
BOM-OC-VINYL	COMMON	(Feature1)	(Feature2)		
BOM-OC-VINYL	VINYL	(Feature3)	(Feature4)		
BOM-Item1	VINYL	(Feature3)			
BOM-Item2	VINYL	(Feature3)			
BOM-OC-WOOD	COMMON	(Feature1)	(Feature2)		
BOM-OC-WOOD	WOOD	(Feature5)	(Feature6)		
BOM-OC-HANDLE	COMMON	(Feature1)	(Feature2)		

1.3.3.3 Alternative Modeling Strategies

Reusing an Attribute Value for Multiple Items

It may be required that more than one Item needs the same configuration attribute value. You can accomplish this by making the Items point to the same instance of the attribute Feature that contains that value. You do this by setting the same value for the attribute Property ATTR_*n*_PATH in each Item.

This strategy is shown in Example 1–7.

Example 1–7 Reusing an Attribute Value for Multiple Items

```
...
|   |__BOM-Item1
|   |          <ATTR_1_PATH="COMPONENT_ATTRIBUTES.Feature14">
|   |__BOM-Item2
|              <ATTR_1_PATH="COMPONENT_ATTRIBUTES.Feature14">
|__COMPONENT-ATTRIBUTES
    |__Feature14
    |          <ATTR_CONTEXT="VINYL">
    |          <ATTR_NAME="VINYL-ATTRIB3">
   ...
```

In this example, both BOM-Item1 and BOM-Item2 point to Feature14, so they both define the value of ATTR_1_PATH as COMPONENT_ATTRIBUTES.Feature14. The result is that the same configuration attribute value is written in CZ_CONFIG_ATTRIBUTES for both of these Items.

Database Results

With reference to the explanation for Table 1–7 on page 1-37, the design in Example 1–7 on page 1-37 would produce database records like those shown in Table 1–8 on page 1-38.

Table 1–8 Reusing an Attribute Value for Multiple Items (Database Results)

Item	Context	Attribute1	Attribute2	Attribute3	Attribute4
...					
BOM-Item1	VINYL			(Feature14)	
BOM-Item2	VINYL			(Feature14)	
...					

As an alternative, you could assign the attribute to the parent node in the Model and use the ATTR_MODE property to propagate the value to all selected children.

Reusing an Attribute Value for Multiple Contexts

It may be required that the same configuration attribute value has to be written as the value of more than one descriptive flexfield segment. You can accomplish this by making the Item specify an overriding context (ATTR_*n*_CONTEXT) and attribute name (ATTR_*n*_NAME) as part of the assignment.

This strategy is shown in Example 1–8.

Example 1–8 Reusing an Attribute Value for Multiple Contexts

```
...
|   |__BOM-Item1
|              <ATTR_1_PATH="COMPONENT_ATTRIBUTES.Feature14">
|              <ATTR_2_PATH="COMPONENT_ATTRIBUTES.Feature14">
|              <ATTR_2_CONTEXT="SHIPPING">
|              <ATTR_2_NAME="SHIPPING-ATTRIB1">
|__COMPONENT-ATTRIBUTES
    |__Feature14
    |          <ATTR_CONTEXT="VINYL">
    |          <ATTR_NAME="VINYL-ATTRIB3">
   ...
```

Database Results

With reference to the explanation for Table 1–7 on page 1-37, the design in Example 1–8 on page 1-38 would produce database records like those shown in Table 1–9 on page 1-39.

Table 1–9 Reusing an Attribute Value for Multiple Contexts (Database Results)

Item	Context	Attribute1	Attribute2	Attribute3	Attribute4
...					
BOM-Item1	VINYL			(Feature14)	
BOM-Item1	SHIPPING	(Feature14)			
...					

Using Different Contexts with Different Values

Multiple contexts assigned to the same Item should be modeled as multiple attribute Features with different ATTR_CONTEXT Property values. Then, the Item will have ATTR_NAME Properties for each of those Features.

This strategy is shown in Example 1–9.

Example 1–9 Using Different Contexts with Different Values

```
    ...
|   |__BOM-Item3
|              <ATTR_1_PATH="COMPONENT_ATTRIBUTES.Feature15">
|              <ATTR_2_PATH="COMPONENT_ATTRIBUTES.Feature16">
|              <ATTR_3_PATH="COMPONENT_ATTRIBUTES.Feature17">
|__COMPONENT-ATTRIBUTES
    |__Feature15
    |           <ATTR_CONTEXT="WOOD">
    |           <ATTR_NAME="WOOD-ATTRIB3">
    |__Feature16
    |           <ATTR_CONTEXT="PRICING">
    |           <ATTR_NAME="PRICING-ATTRIB1">
    |__Feature17
    |           <ATTR_CONTEXT="MFG">
    |           <ATTR_NAME="MFG-ATTRIB1">
    ...
```

Database Results

With reference to the explanation for Table 1–7 on page 1-37, the design in Example 1–9 on page 1-39 would produce database records like those shown in Table 1–10 on page 1-40.

Table 1–10 Using Different Contexts with Different Values (Database Results)

Item	Context	Attribute1	Attribute2	Attribute3	Attribute4
...					
BOM-Item3	WOOD			(Feature15)	
BOM-Item3	PRICING	(Feature16)			
BOM-Item3	MFG	(Feature17)			
...					

1.3.3.4 Special Considerations

This section covers some important considerations that may affect your configuration model's structure.

Referenced Models

Your Model can include child Models that are connected through References (as is the case with most imported BOM Models). If this is the case, then the output Functional Companion (described in Section 1.3.4 on page 1-42) traverses the entire structure of the root Model and all other Models that it references, collecting configuration attribute data from any attribute Properties whose names conform to the conventions in Table 1–6 on page 1-32.

Location of Attribute Features

You must give careful thought to where you create attribute Features.

An attribute Feature can be located anywhere in a Model, providing that the location is a node path that can be specified by the Property ATTR_*n*_PATH. Consequently, the node path to an attribute Feature must be specified as relative to the Model that contains the Item.

The output Functional Companion (described in Section 1.3.4 on page 1-42) first finds the node that is the root of the entire configuration model, and then searches from there for Properties whose names conform to the conventions in Table 1–6 on page 1-32.

Because the Functional Companion can find Features in child Models that are connected through References (see Referenced Models on page 1-40), it is possible to assign a configuration attribute value to an Item in a parent Model by pointing (with the Property ATTR_*n*_PATH) to an attribute Feature in a referenced child Model. Such a node path must include the names of any child Models between the Model that contains the Item and the attribute Feature. However, a node path that you can express as a Text Property value can only point down the tree of Models, not upwards. Consequently, it is not possible to point to an attribute Feature in a parent Model, and therefore you cannot assign a configuration attribute value from a parent Model to an Item in a child Model.

To summarize: parent Models can use attribute Features defined in referenced child Models; child Models cannot use attribute Features defined in parent Models.

If follows from the preceding facts that if you intend to use a referenced child Model without the parent Model, then you must ensure that it contains all the attribute Features that are employed as the configuration attributes on the Items in that Model.

Multiple Component Instances in the Node Path

The node path to an attribute Feature, which is specified by the Property ATTR_*n*_PATH, cannot include any Components that can be instantiated multiple times. (Such a Component is one that has its Instances set to a Maximum greater than 1, so that more than one runtime instance of the Component can be created and configured.) The existence of multiple instances of a Component in a node path makes the path ambiguous. Consequently, you cannot place any attribute Features inside such a Component, because the output Functional Companion (described in Section 1.3.4 on page 1-42) cannot resolve the correct path to that attribute Feature.

Required Items

Configuration attributes cannot be defined for required Items in a BOM Model, because such Items are not configurable, and consequently are not imported when you import the BOM Model into Oracle Configurator Developer to define a configuration model. (See the *Oracle Configurator Implementation Guide* for details about importing.)

1.3.3.5 Creating Functional Companion Rules

Set the **Definition** of the Functional Companion rule as shown in the following table:

Rule Definition Attribute	Definition Value
Base Component	The node from which the Functional Companion will search for configuration attribute Properties. See Section 1.3.4 on page 1-42 and Section 1.3.3.4 on page 1-40 for background.
Type	Event-Driven
Implementation	Java
Program String	The name of the Java class that implements the output Functional Companion. For the example, this is `WriteAttributes` (Example A–2 on page A-11).

For more details on defining Functional Companion rules, see the *Oracle Configuration Interface Object (CIO) Developer's Guide*. For details on defining other kinds of configuration rules, see the *Oracle Configurator Developer User's Guide*.

1.3.4 The Output Functional Companion

The output Functional Companion performs the work of transferring the configuration attribute data from a configured runtime instance of the Model to the CZ_CONFIG_ATTRIBUTES table.

A Functional Companion has been specially written to support the configuration attribute methodology described in this chapter. The source code for this Functional Companion is provided in Example A–2 on page A-11. This section describes its operation at a high level.

For the Functional Companion to operate on the configuration model, you must create a configuration rule that associates the Functional Companion with the Model that contains the configuration attribute Properties, as illustrated in Example 1–6 on page 1-34, and described in Section 1.3.3 on page 1-30.

The example Functional Companion does not have to be modified, *if* you modify your Model in the manner described in Section 1.3.3. In modifying your Model, you must follow the conventions described for naming Properties in order for the Functional Companion to be able to collect values for the attribute Features.

Note: The example Functional Companion assumes that your Model is a BOM Model. If it is not, then see Section 1.3.7.1, "Modifying the Functional Companion" on page 1-49.

Here is a high-level description of the operation of the output Functional Companion:

1. During a configuration session, the end user selects options that, through configuration rules, result in the entry of configuration attribute data in the attribute Features of your Model.
2. The end user terminates the session successfully by clicking the Done button, which saves the configuration.
3. The Functional Companion overrides the `afterSave()` method of `AutoFunctionalCompanion`, so it is triggered after the configuration is successfully saved.
4. The Functional Companion clears the CZ_CONFIG_ATTRIBUTES table of any configuration attribute data for the Items in the current configuration model. This prevents previously entered data from conflicting with data from the latest configuration session.
5. The Functional Companion traverses the entire configuration model, collecting the values of all the attribute Features by using the Properties described in Section 1.3.3 on page 1-30. This collection includes the names of the descriptive flexfield contexts and segment names for the flexfield data, recorded through the ATTR_NAME and ATTR_CONTEXT Properties.
6. The Functional Companion queries the descriptive flexfield tables in the Oracle Applications database for the name of each ATTRIBUTE*n* column in the CZ_CONFIG_ATTRIBUTES table that corresponds to the context and segment name for each configuration attribute value.
7. The Functional Companion prepares and executes a SQL statement that inserts into the CZ_CONFIG_ATTRIBUTES table all of the configuration attribute data from the latest configuration session for your configuration model.

1.3.5 Using Configuration Attributes in the Downstream Application

In order to use the configuration attribute data that is collected from a configuration session, you must implement some significant customizations to your downstream application. The specifics vary according to the particular application. This section begins with an explanation of how the data is stored into the CZ_CONFIG_ATTRIBUTES table (Section 1.3.5.1), and then outlines some strategies for customizing the downstream application.

1.3.5.1 Storing Output Data for Downstream Use

At the end of a configuration session, the end user closes the runtime Oracle Configurator by clicking the Done button, which saves the configuration, triggering the output Functional Companion (described in Section 1.3.4 on page 1-42). The Functional Companion writes the saved configuration attribute data into the CZ_CONFIG_ATTRIBUTES table in the following way:

- Traverse the tree of the configuration model, visiting each configuration item.
- For each configuration item, collect the values of all the attribute Properties that help define configuration attributes, and the values of all the configuration attributes themselves. The rules that govern how the attribute Properties point to attribute Features are described in Table 1–6 on page 1-32.
- Obtain the identity of the current item in the configured model's structure from the runtime configuration instance, and write these values into the columns CONFIG_HDR_ID, CONFIG_REV_NBR, CONFIG_ITEM_ID.
- Obtain the flexfield context for the configuration attribute from the value defined for the Property ATTR_CONTEXT on the attribute Feature, and write this value into ATTRIBUTE_CATEGORY.

  The combination of CONFIG_HDR_ID, CONFIG_REV_NBR, CONFIG_ITEM_ID, and ATTRIBUTE_CATEGORY constitutes the primary key for the table, which uniquely identifies the configuration attribute value that applies to the given item for the given context.
- Query the Oracle Applications flexfield tables to determine which ATTRIBUTE*n* column of the CZ_CONFIG_ATTRIBUTES table is associated with the value of the Property ATTR_NAME on the attribute Feature, and write the value of the Feature (that is, the value of the configuration attribute) into that ATTRIBUTE*n* column.

This description has been somewhat simplified for clarity. For details on how the Functional Companion implements this database update, see the commented source code in Example A–2 on page A-11.

Example of Storing Output Data

Assume a flexfield definition like the one shown in Figure 1–3 on page 1-30. Assume, for some configuration item, that the value of the attribute Property ATTR_CONTEXT is `EAST`, the value of the attribute Property ATTR_NAME is `COLOR`, and that the attribute Property ATTR_*n*_PATH points to an attribute Feature named `COLOR`, whose value is `WHITE`. The Functional Companion queries the flexfield tables, specifying the value of the flexfield context as `EAST`, and the value

of the flexfield segment (that is, the configuration attribute) as COLOR. According to the flexfield definition in Figure 1–3, the column in which to write the value of the configuration attribute is ATTRIBUTE1, so the Functional Companion writes the selected value of COLOR into ATTRIBUTE1, for the current configuration item.

Example 1–10 on page 1-45 shows a SQL query that displays the value of ATTRIBUTE1 for a particular configuration item (which Oracle Configurator identifies internally by a configuration header ID, revision number, and item ID), and Example 1–11 on page 1-45 shows a result of the query.

Example 1–10 Query for Value of ATTRIBUTE1

```
SELECT
  config_hdr_id, config_rev_nbr, config_item_id, attribute_category, attribute1
FROM
  cz_config_attributes
WHERE  -- a configuration item
  config_hdr_id  = 18900 AND config_rev_nbr = 1 AND config_item_id = 34
AND  -- the flexfield context
  attribute_category = 'EAST';
```

Example 1–11 Value of ATTRIBUTE1

```
CONFIG_HDR_ID CONFIG_REV_NBR CONFIG_ITEM_ID ATTRIBUTE_CA ATTRIBUTE1
------------- ------------- ------------- ------------ ------------
        18900             1            34 EAST         WHITE
```

1.3.5.2 Using Output Data in Downstream Applications

The configuration attribute output data stored in the CZ_CONFIG_ATTRIBUTES table is available for use by downstream applications. However, you must put the data into a form that your particular application can use, which probably requires some degree of customization, depending on the specific characteristics of the application. This section identifies some strategies for customization.

- You must write a custom procedure (see Section 1.3.5.3 on page 1-46) to retrieve the data from the CZ_CONFIG_ATTRIBUTES table and insert it into the descriptive flexfields used by the downstream application.
- You may decide to read the configuration attribute values directly from the CZ_CONFIG_ATTRIBUTES table without inserting them into the downstream application's flexfields. However, doing so requires customizing the

downstream application's forms to read the attributes and assigning them to the correct data.

- For the Oracle Advanced Pricing application, the attributes must be stored as flexfields on the order lines.
- Advanced Pricing is designed to read data from any table, and does not require special customization to do so. For other Oracle Applications the attributes must be stored in the interface table flexfields for those applications.
- For configuration attributes to be displayed in a form that displays an auto-created final assembly (such as Flow Workstation), they need to be stored as attributes on the flow schedules.

  In Oracle Flow Manufacturing, the Flow Workstation calls an API that passes out the Flow schedule number and returns the displayable fields. The API constructs the relationship between the Flow schedule, a sales order, and the configuration attributes related to that order, which are stored in the CZ_CONFIG_ATTRIBUTES table.
- If you need to define configuration attributes on required BOM items in a downstream application, then you must explicitly join them with their parents and populate them to get attribute values.

  The reason behind this requirement is that required items in a BOM are not configurable, and consequently cannot be imported into an Oracle Configurator configuration model. Therefore, they are not present during a configuration session, and cannot be written into the CZ_CONFIG_ATTRIBUTES table by the output Functional Companion.

1.3.5.3 Linking Configuration Attributes to Flexfields

You can use the following approach in designing a custom procedure (which you must write) that links the information in a configuration attribute to a flexfield for a downstream application:

- A line ID is passed as an argument to the custom procedure. The line ID is a primary key that points to the host application's equivalent of a configurable line item in an order or quote.
- The line item contains a foreign key formed from the columns CONFIG_HDR_ID, CONFIG_REV_NBR, CONFIG_ITEM_ID in CZ_CONFIG_ATTRIBUTES. This key points to the set of records in CZ_CONFIG_ATTRIBUTES associated with the selected items in the order or quote.

- Use CZ_CONFIG_ATTRIBUTES.ATTRIBUTE_CATEGORY to provide a description of the flexfields that apply to the context for the particular component from the order or quote.
- The flexfield definition contains information about the names of the columns (ATTRIBUTE*n*, and so on) in CZ_CONFIG_ATTRIBUTES that contain the desired configuration attributes.

  Alternatively, depending on your implementation, the reading code in your custom procedure could assume certain hardcoded column names for each segment in a particular context.

1.3.5.4 Downstream User Interfaces

Downstream, these are the ways to use configuration attributes in an application's user interface:

- Using the standard Oracle Applications UI with the application flexfields (see Oracle Applications User Interface with Flexfields on page 1-47)
- Directly feeding the configuration attribute table into a customized host application UI (see Custom Application User Interface on page 1-48)

The choice of UI depends on your needs.

Oracle Applications User Interface with Flexfields

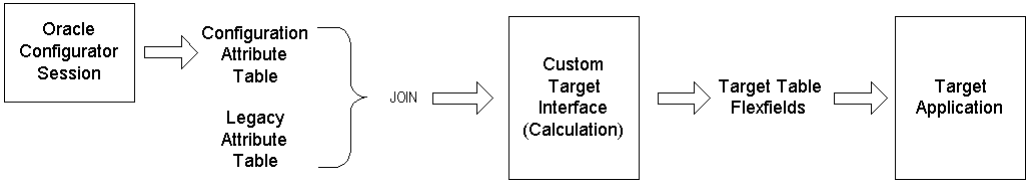
The data in the populated table CZ_CONFIG_ATTRIBUTES needs to be inserted into the flexfields on the downstream application table. For that purpose you must write a custom procedure to populate those flexfields on the target records. See the information on Oracle Workflow APIs in the *Oracle Workflow Guide* for details. See Figure 1–4 on page 1-47.

Figure 1–4 Flow for Oracle Applications UI with Flexfields

It is also acceptable to join the configuration attribute table before populating it in the application's schema, if the resulting data is a derivation from the configuration and legacy attributes. In that case, writing to the downstream table's flexfields after calculations could be more efficient than separating the steps in different tables. See Figure 1–5 on page 1-48.

Note: The Oracle Advanced Pricing application is designed to read data from any table, and does not require special customization to do so.

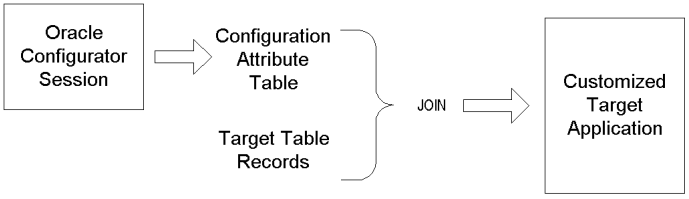
Figure 1–5 Flow for Oracle Applications UI with Flexfields with Joined Tables



Custom Application User Interface

For custom-built UIs, there may be no need to populate the flexfields on the downstream application's table. Instead, you might be able to join the application table with CZ_CONFIG_ATTRIBUTES. This is a valid approach if the data that is represented by the configuration attributes is used by the downstream application only for display purposes. See Figure 1–6 on page 1-48.

Figure 1–6 Flow for Custom Application UI



Note: Details about possible customizations depend on the downstream application and your implementation. Please refer to the relevant documentation.

1.3.6 Maintaining the Output Solution

You must *manually* update your Model in Oracle Configurator Developer if the following are changed:

- The flexfield definitions in Oracle Applications. Changes to the flexfield definitions might include changes to context or attribute column assignments.
- The BOM (in Oracle Bills of Materials) that is the basis for your imported Model. Changes to the Bill of Material might include the addition of an Item that has configuration attributes.

Bill of Material changes are reflected by refreshing your Model. See the *Oracle Configurator Implementation Guide* for details on refreshing models. Performing a refresh of your model may result in requiring changes to the structure of Features and Properties described in Section 1.3.3, "Modifying the Model" on page 1-30.

1.3.7 Optional Flows

This section describes optional tasks that you may need to perform that are not part of the flow described in Figure 1–1 on page 1-26.

Caution: If you do not follow the model design described in Section 1.3.3 on page 1-30, the output Functional Companion, as provided, will not work.

1.3.7.1 Modifying the Functional Companion

This section describes potential modifications to the example Functional Companion.

The example Functional Companion provided with this methodology (Example A–2 on page A-11) does not have to be modified, *if* you modify your Model in the manner described in Section 1.3.3 on page 1-30, and *if* your Model is a BOM Model. If your Model is a Component (non-BOM) type, then see Using Non-BOM Models on page 1-50.

Note: Be sure to check the *Oracle Configuration Interface Object (CIO) Developer's Guide* for a description of the Java development skills required for success with Functional Companions.

Modifying the Property Names

In modifying your Model, you must follow the conventions for naming Properties described in Table 1–6 on page 1-32 in order for the Functional Companion to be able to collect values for the attribute Features. If you want to use different Property

names, then you must modify certain character strings, which are typographically highlighted in Example 1–12 on page 1-50, so that they match your own Property names.

Example 1–12 Strings for Property Names in the Functional Companion

```
...
  private Map getAttributes(BomNode node) {
...
      if (name.startsWith("ATTR_")) {
...
        int beginningIndex = new String("ATTR_"). length();
        StringTokenizer tokens = new
StringTokenizer(name.substring(beginningIndex), "_", false);
...
        if (name.endsWith("PATH")) {
...
        else if (name.endsWith("MODE")) {
...
        else if (name.endsWith("CONTEXT")) {
...
        else if (name.endsWith("NAME")) {
...
  private class Attribute implements Comparable {
...
        Property prop = m_feature.getPropertyByName("ATTR_CONTEXT");
...
        Property prop = m_feature.getPropertyByName("ATTR_NAME");
```

Using Non-BOM Models

The example Functional Companion assumes that your Model is a BOM Model, like the Model in Example 1–6 on page 1-34. If your Model is a Component type, then you must modify the `traverseBomTree()` method of the Functional Companion to test whether the Component containing a given configuration attribute Item is a Model, so that the value of the ATTR_*n*_PATH Property can be resolved correctly. See the description of the ATTR_*n*_PATH Property described in Table 1–6 on page 1-32 for background. (For clarity, you may also wish to rename the `traverseBomTree()` method to indicate that it is not traversing a BOM Model.)

In order to determine whether a Component is a Model (that is, the root of the configuration, or a Reference), use the following test:

```
private void traverseBomTree(RuntimeNode node, List parentAttributes) {
```

```
...
  if (node.getReferringOrDatabaseID() != node.getDatabaseID() ) {
    // the Component is a Model, so collect its attributes
...
 }
...
}
```

Part II

Appendices

Part II contains the following appendices:

- Code Examples

A

Code Examples

This chapter contains code examples that support other chapters of this document. These examples are fuller and longer than the examples provided in the rest of this document, which are often fragments. See the cited background sections for details.

> **Note:** Consult the *Oracle Configurator Implementation Guide* and the *Oracle Configuration Interface Object (CIO) Developer's Guide* for examples that support other features of Oracle Configurator.

Table A–1 Code Examples Provided

Purpose of Example	Example
Section A.1, "Using Configuration Attributes for Input"	Example A–1, "Using Configuration Attributes for Input (CfgInputExample.java)"
Section A.2, "Using Configuration Attributes for Output"	Example A–2, "Using Configuration Attributes for Output (WriteAttributes.java)"

You should consult these other documents for details on the tasks described in this section:

- For information on how to write and compile Functional Companions, and on how to incorporate them into your configuration model, see the *Oracle Configuration Interface Object (CIO) Developer's Guide*.
- For information on how to install Functional Companions, see the *Oracle Configurator Installation Guide*.
- For an explanation of updating configurations, see the *Oracle Configurator Developer User's Guide*.

- For an details on how to build a configuration model that enables you to update configurations, see the *Oracle Configurator Developer User's Guide*.

A.1 Using Configuration Attributes for Input

This example demonstrates how to use a Functional Companion to transfer configuration attribute data into a configuration model from a host application.

- To use the example, you must modify it in accordance with the instructions in Section 1.2.4, "Modifying the Functional Companion Example" on page 1-15.
- For details about the specific example that is supported by this code, see Section 1.2.1, "Example for Implementing Input Configuration Attributes" on page 1-7.
- For general background, see Section 1.2, "Implementing Configuration Attributes for Input" on page 1-7.

> **Note:** If you have installed Oracle Configurator Developer and the Oracle Configurator documentation, then the source code for Example A–1 is available as the file `CfgInputExample.java`, in the folder *`OC_Developer_installation`*`\doc\code_examples`.

Example A–1 Using Configuration Attributes for Input (CfgInputExample.java)

```
/*===========================================================================+
 |      Copyright (c) 2002 Oracle Corporation, Redwood Shores, CA, USA       |
 |                         All rights reserved.                              |
 +===========================================================================+
 |  FILENAME
 |      CfgInputExample.java
 +===========================================================================*/

import oracle.apps.cz.cio.AutoFunctionalCompanion;
import oracle.apps.cz.cio.IRuntimeNode;
import oracle.apps.cz.cio.IOption;
import oracle.apps.cz.cio.IOptionFeature;
import oracle.apps.cz.cio.IState;
import oracle.apps.cz.cio.LogicalException;
import oracle.apps.cz.cio.NoSuchChildException;
```

```
import oracle.apps.cz.cio.TransactionException;
import oracle.apps.cz.utilities.Assert;
import oracle.apps.cz.utilities.CheckedToUncheckedException;
import oracle.apps.cz.utilities.NameValuePairSet;

import java.sql.*;

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


/**
 *  CfgInputExample provides the skeleton code needed to transfer
 *  parameters from the database into the model.
 *
 *  This Functional Companion must be attached to each model that it
 *  is to work with.  Use Oracle Configurator Developer to create the
 *  relationship between model and companion.
*/

// It is strongly recommended that only input parameters be transferred
// from the database to the model. That is, this companion should not
// make calculations or decide validity of a configuration. The model
// should use rules to perform those operations. The maintenance of the model
// will be made easier if the Functional Companion is kept simple.

// This companion is written to run from Oracle Quoting.
//
// The model that this FC is attached to should have an Option Feature
// named 'US_State' (case sensitive) with some, if not all, Options
// that represent each state of the United States. The names of the
// Options should be the uppercase two-letter postal codes for each state
// (for example, CA, NY, MA, RI).

// You must extend AutoFunctionalCompanion in order to use the onNew()
// and onRestore() methods.

public class CfgInputExample extends AutoFunctionalCompanion {

  private Long   mOrderLineNumber = null; // Retrieved from the user's session
  private Long   mApplicationId = null; // From the user session information
  private String mUsState = null;

  public void onNew() throws LogicalException {
    doInputAttributeTransfer();
```

```
}

public void onRestore() throws LogicalException {

  // Implementers have to decide if the transfer of
  // attributes needs to occur during the restoration of
  // a saved model. In most cases, the best practice is to
  // retrieve the current value from the database since
  // the value can be changed from the host application.

  // If an attribute is truly only set once, when the configuration
  // is newly created, then the attribute should not be
  // restored. Use inititial requests, which are described in the
  // Oracle CIO Developer's Guide.

  doInputAttributeTransfer();

}

/**
 * Transfers input attribute data into a configuration model.
 *
 */
private void doInputAttributeTransfer() throws LogicalException {

  // The general steps in passing config attributes are:
  // 1. Get the session information.
  // 2. Use the database to get the information to be transferred into the model.
  // 3. Set the model values.

  getSessionParameters();

  // If there are no parameters for this session then continue.

  if (mOrderLineNumber == null) return;

  // Implementers have to decide if attributes should be treated
  // differently, depending on which application is calling Oracle
  // Configurator.  In this example assume that the information
  // should be transferred only for Oracle Quoting.

  if (mApplicationId.longValue() != 697) return;

  getInputAttributes();
  transferInputAttributesIntoModel();
```

```
}

/**
 * Retrieves the session information from the configuration
 * and makes the information available.
 */
private void getSessionParameters() {

  // The initialization message holds the parameters supplied by the
  // application when Oracle Configurator was started.

  NameValuePairSet initParams = getRuntimeNode().getConfiguration().getInitParameters();

  String paramValue = null;

  String paramValue = (String)initParams.getValueByName("client_line");

  // Each host application calling Oracle Configurator can pass this parameter
  // with different content. You need to cast the type of the parameter value
  // if the host application does not pass a string for the client_line.

  if (paramValue != null) mOrderLineNumber = Long.valueOf(paramValue);

  // This example only uses the client_line parameter.
  // To use other parameters, write additional get and set statements for them.

  // The other parameters that may be useful, depending on the application, are:
  //    client_header
  //    client_line_detail
  // The remainder of the initialization parameters are described
  // in the Session Initialization chapter.

  // You must identify the host application that called Oracle Configurator.

  paramValue = (String)initParams.getValueByName("calling_application_id");
  if (paramValue != null) mApplicationId = Long.valueOf(paramValue);
}

/**
 * Retrieves the input attributes from the database.
 */
private void getInputAttributes() {

  Connection conn = getRuntimeNode().getConfiguration().getContext().getJDBCConnection();
  PreparedStatement pStmt = null;
```

```
ResultSet rs;
int ret;

try {

  // Define a custom query to extract the desired database value
  // for the configuration attribute.
  // In this case, get the value for the U.S. state associated
  // with a quote line.

  try {

    if (mApplicationId.longValue() == 697) {

      String sql = "select p.state "
                 + "from   hz_parties p, "
                 + "       aso_quote_headers_all_v q, "
                 + "       ASO_QUOTE_LINES_ALL_V l "
                 + "where  p.party_id = q.cust_account_id "
                 + "and    l.quote_header_id = q.quote_header_id "
                 + "and    l.quote_line_id = ?";

      // Prepared statements generally provide good performance.

      pStmt = conn.prepareStatement(sql);

      // Put the line number into the first (and only) parameter
      // of the PreparedStatement, for quote_line_id.

      pStmt.setLong(1, mOrderLineNumber.longValue());

      // Execute the query, putting the results into the rows of the ResultSet.

      rs = pStmt.executeQuery();

      // Iterate through the ResultSet and get the first value.

      try {
        if (rs.next()) {

          // If a record exists, retrieve the value(s) and consider if
          // there are nulls.

          // The first (and only) return value in this example is from HZ_PARTIES.STATE.
          // Store the value from column 1 of the ResultSet into mUsState.
```

```
        mUsState = rs.getString(1);

        // Check to see if the value was null. This check is critical
        // for integer and other data types.

        if (rs.wasNull()) mUsState = null;
      }
      else {

        // If no record exists in the ResultSet, raise an exception and
        // provide a message.

        // Since this Functional Companion is being called as
        // part of a regular process, the FC expects that the
        // values of client_line and other parameters are
        // correct. They may not be correct, or (more likely)
        // the query is not accurate.

        // Put as much useful information into the error message as possible.

        String s = "CfgInputExample.getInputAttributes: Did not find record for "
                 + "client_line " + mOrderLineNumber + ".\n\n"
                 + "query = " + sql;

        throw new RuntimeException(s);
      }
    }
    finally {

      // Make sure that all ResultSets are closed. If they are
      // not closed, then it is easy to use up the allotment of
      // database cursors.

      rs.close();
    }
  }
}
finally {

  // Make sure that all prepared statements are closed. If they are
  // not closed, then it is easy to use up various database resources.

  if (pStmt != null) pStmt.close();
}
```

```
    }
    catch (SQLException se) {

      // Any problem with the database will
      // stop the session. The error will be logged.

      throw new RuntimeException("CfgInputExample.getInputAttributes: " + se.toString());
    }
  }

  /**
   * Transfers the information from the database into the model structure.
   */
  private void transferInputAttributesIntoModel() throws LogicalException {

    // Use a method customized for the specific configuration attribute.

    transferUsState();
  }

  /**
   * Specialized method to transfer attribute data
   * into an Option Feature with a specified name.
   */
  private void transferUsState()throws LogicalException {

    // Find the model element(s) that will be set with the
    // configuration attribute value.

    // In this example, search for a Feature named "US_State". This
    // feature has Options corresponding to the postal codes for some
    // individual states in the United States.

    // This FC starts its search from the model node that the FC is
    // associated with, and stops when it finds the first object named
    // "US_State".

    // If, for any reason, there is not a "US_State" then return. The model's
    // configuration rules or defaults will provide the value.

    // If an attribute is truly only set once, when the
    // configuration is newly created, then use initial requests,
    // which are described in the Oracle CIO Developer's Guide.

    if (mUsState == null) return;
```

```
    IRuntimeNode rtNode = findFirstNodeByName(getRuntimeNode(), "US_State");

    // Implementers have to decide whether failing to find the node
    // should be treated as an exception. In this example, it is not
    // an issue if the model lacks the model element.

    if (rtNode == null) return;

    // Implementers should verify that the node is what is expected.
    // Changes to the configuration model can lead to runtime errors,
    // such as ClassCastExceptions, if there is no verification.

    if (rtNode.getType() != IRuntimeNode.OPTION_FEATURE) return;

    // Find the option for the US State. In this example, if the option is not found
    // do not consider it a problem. The model's configuration rules or defaults
    // will decide what to do.

    // Use the attribute value retrieved from the database to find the
    // Option with the matching name.

    IOption option = (IOption)findFirstNodeByName(rtNode, mUsState);
    if (option == null) return;

    // Set the value of the attribute Feature, by selecting the Option.

    try {
      option.select();
    }
    catch(TransactionException te) {

      // There should be never be a problem with CIO transactions. If one
      // occurs then provide as much informaiton as possible.

     throw new RuntimeException("CfgInputExample.transferInputAttributesIntoModel: " +
te.toString());

    }
  }


  /**
   * Utility method to locate the first model node with a specified name
   * found in a search starting from the current node.
```

```
   */
  private IRuntimeNode findFirstNodeByName(IRuntimeNode parent, String name) {

    IRuntimeNode found = null;

    if (parent == null) return null;
    if (!parent.hasChildren()) return null;

    try {
      found = parent.getChildByName(name);
    }
    catch (NoSuchChildException nsce) {

      // Recurse through the children

      Iterator it = parent.getChildren().iterator();
      while (it.hasNext()) {
        found = findFirstNodeByName((IRuntimeNode)it.next(), name);
        if (found != null) return found;
      }
    }
    return found;
  }
}
```

A.2 Using Configuration Attributes for Output

This example demonstrates how to use a Functional Companion to transfer configuration attribute data from a configuration model into a host application.

- To use the example, you may modify it in accordance with the instructions in Section 1.3.4, "The Output Functional Companion" on page 1-42.
- For general background, see Section 1.3, "Implementing Configuration Attributes for Output" on page 1-25.

> **Note:** If you have installed Oracle Configurator Developer and the Oracle Configurator documentation, then the source code for Example A–1 is available as the file `WriteAttributes.java`, in the folder *`OC_Developer_installation`*`\doc\code_examples`.

Example A–2 Using Configuration Attributes for Output (WriteAttributes.java)

```
/*=========================================================================+
 |      Copyright (c) 2002 Oracle Corporation, Redwood Shores, CA, USA      |
 |                         All rights reserved.                            |
 +=========================================================================+
 |  FILENAME                                                               |
 |      WriteAttributes.java                                               |
 |  DESCRIPTION                                                            |
 |                                                                         |
 |  NOTES                                                                  |
 |                                                                         |
 |  DEPENDENCIES                                                           |
 |                                                                         |
 |  HISTORY                                                                |
 |       26-Nov-01 Anupam Miharia     Created.                             |
 +=========================================================================*/

import oracle.apps.fnd.common.Context;
import oracle.apps.cz.cio.AutoFunctionalCompanion;
import oracle.apps.cz.cio.Property;
import oracle.apps.cz.cio.OptionFeatureNode;
import oracle.apps.cz.cio.BooleanFeature;
import oracle.apps.cz.cio.Configuration;
import oracle.apps.cz.cio.CountFeature;
import oracle.apps.cz.cio.IntegerNode;
import oracle.apps.cz.cio.DecimalNode;
import oracle.apps.cz.cio.ComponentSet;
import oracle.apps.cz.cio.NoSuchChildException;
import oracle.apps.cz.cio.BomNode;
import oracle.apps.cz.cio.TextNode;
import oracle.apps.cz.cio.BomModel;
import oracle.apps.cz.cio.RuntimeNode;
import oracle.apps.cz.cio.IRuntimeNode;
import com.sun.java.util.collections.Map;
import com.sun.java.util.collections.Collection;
import com.sun.java.util.collections.HashMap;
import com.sun.java.util.collections.ArrayList;
import com.sun.java.util.collections.Iterator;
import com.sun.java.util.collections.TreeSet;
import com.sun.java.util.collections.List;
import java.util.StringTokenizer;
import java.sql.Connection;
import java.sql.PreparedStatement;
```

```
import java.sql.SQLException;
import java.sql.ResultSet;

/**
 * @author Anupam Miharia
 *
 * This is a sample implementation of an afterSave Functional Companion
 * that is used to write the attributes associated with BOM nodes to
 * the CZ_CONFIG_ATTRIBUTES table. The implementation is based on the
 * attribute association as described in the documentation.
 */
public class WriteAttributes extends AutoFunctionalCompanion
{
  public static final int ATTR_MODE_CURRENT = 0;
  public static final int ATTR_MODE_CURRENT_AND_IMMEDIATE_CHILDREN = 1;
  public static final int ATTR_MODE_CURRENT_AND_ALL_CHILDREN = 2;
  Configuration config = null;
  /**
   * An afterSave FC is called after saving the configuration.
   * This collects all the attributes from the BOM nodes and writes
   * them to the database.
   */
  public void afterSave() {
    config = getRuntimeNode().getConfiguration();
    // First, clear all attribute data for this configuration from
    // the CZ_CONFIG_ATTRIBUTES table.
    try {
      clearConfigAttributes(config);
    } catch (SQLException sqle) {
      throw new RuntimeException("Error in clearing the previous attribute data");
    }
    // Starting from the root, call traverseBomTree on all top level BOM nodes.
    RuntimeNode root = (RuntimeNode)config.getRootComponent();
    if (root instanceof BomNode) {
      traverseBomTree(root, new ArrayList());
    } else {
      Iterator iter = root.getChildren().iterator();
      while (iter.hasNext()) {
        RuntimeNode child = (RuntimeNode)iter.next();
        if (child instanceof BomNode || child instanceof ComponentSet) {
          traverseBomTree(child, new ArrayList());
        }
      }
    }
  }
```

```
/**
 * Iterates over the BOM tree and writes the attributes for each BOM node.
 * @param node Can either be a BomNode or a ComponentSet.
 * @param parentAttributes List of attributes from ancestors that is to be
 * applied to this node.
 */
private void traverseBomTree(RuntimeNode node, List parentAttributes) {
  List childAttribs = null;

  if (!(node instanceof ComponentSet)) {
    BomNode bomNode = (BomNode)node;

    // Write attribute data only for selected BOM nodes
    if (!bomNode.isSelected()) return;

    // First get the attributes corresponding to this BOM node
    Map attributes = getAttributes(bomNode);

    // Then group the attrbutes of this node with those from its ancestors
    // and write the combined set to the CZ_CONFIG_ATTRIBUTES table.
    // Also get the list of attributes that have to be applied to this
    // node's children.
    childAttribs = groupAndWriteAttributes(bomNode, attributes, parentAttributes);
  } else {
    // Pass along the parent's attributes to the children for a ComponentSet.
    childAttribs = parentAttributes;
  }

  // Now recursively visit all the children subtrees passing along the list
  // of attributes that have to be applied to its children.
  Iterator iter = node.getChildren().iterator();
  while (iter.hasNext()) {
    RuntimeNode child = (RuntimeNode)iter.next();
    if (child instanceof BomNode || child instanceof ComponentSet) {
      traverseBomTree(child, childAttribs);
    }
  }
}

/**
 * Group the attributes together by:
 * 1. Combining the attributes passed from the ancestors with the current set.
 * 2. Combining the attributes that have a commmon context in a single list.
 * 3. Creating a list of child Attributes to be applied to the node's children.
```

```
   * @param node The BOM node whose attributes are to be written
   * @param attributes Attributes corresponding to the given BOM node
   * @param parentAttributes List of attributes from ancestors that is to be
   * applied to the given BOM node.
   */
  private List groupAndWriteAttributes(BomNode node, Map attributes, List parentAttributes) {
    // Attributes from parent go only one level deep if their mode = 1.
    // Set their mode = 0 here so that they don't propagate further.
    Iterator iter = parentAttributes.iterator();
    while (iter.hasNext()) {
      Attribute attr = (Attribute)iter.next();
      if (attr.getMode() == ATTR_MODE_CURRENT_AND_IMMEDIATE_CHILDREN) {
        attr.setMode(ATTR_MODE_CURRENT);
      }
    }
    // List of attributes from this node to be applied to children.
    List childAttributes = new ArrayList();
    // Create an aggregate map of attributes collected by context because
    // we can have multiple attributes for this node having a common context.
    // The key of this map is the attribute context and the value is
    // a HashMap of attribute name and attribute value.
    HashMap ctxAttrsMap = new HashMap();
    // Combine the list of attributes for this node with that from the parent.
    // Make it a Set so that all duplicate attributes are eliminated.
    // Look at definition of Attribute.equals().
    Collection allAttributes = new TreeSet();
    allAttributes.addAll(attributes.values());
    allAttributes.addAll(parentAttributes);

    // Iterate over the combined list of attributes.
    iter = allAttributes.iterator();
    while (iter.hasNext()) {
      Attribute attribute = (Attribute)iter.next();

      // If the attribute mode is not CURRENT-only then add it to the
      // childAttributes list
      if (attribute.getMode() != ATTR_MODE_CURRENT) {
        childAttributes.add(attribute);
      }
      // We have to combine all attributes belonging to this node by context
      // as well, so that it can be written as a single row in the
      // CZ_CONFIG_ATTRIBUTES table.
      String ctx = attribute.getAttrContext();

      HashMap values = (HashMap)ctxAttrsMap.get(ctx);
```

```
    if (values == null) {
      values = new HashMap();
      ctxAttrsMap.put(ctx, values);
    }
    if (attribute.hasBooleanValue()) {
      values.put(attribute.getName(), new Boolean(attribute.getBooleanValue()));
    } else if (attribute.hasIntegerValue()) {
      values.put(attribute.getName(), new Integer(attribute.getIntValue()));
    } else if (attribute.hasDecimalValue()) {
      values.put(attribute.getName(), new Double(attribute.getDecimalValue()));
    } else if (attribute.hasStringValue()) {
      values.put(attribute.getName(), attribute.getStringValue());
    } else {
      values.put(attribute.getName(), "NULL");
    }
  }

  // Write all the attributes to the database after we are done grouping
  // them appropriately.
  writeAttributes(node, ctxAttrsMap);
  return childAttributes;
}

/**
 * Writes the attributes for the given node to the database.
 * The parameter 'attributes' is a Map where the key is the attribute context
 * and the value is a map of attribute name and attribute value.
 */
private void writeAttributes(BomNode node, Map attributes) {
  Iterator iter = attributes.keySet().iterator();
  while (iter.hasNext()) {
    String context = (String)iter.next();
    Map nameValuePairs = (Map) attributes.get(context);
    insertAttributes(node, context, nameValuePairs, config.getContext());
  }
}

/**
 * Inserts the attributes belonging to a given node and context into
 * the CZ_CONFIG_ATTRIBUTES table.
 */
public int insertAttributes (BomNode node, String attrContext, Map attributes, Context ctx) {
  Connection conn = ctx.getJDBCConnection(ctx);
  PreparedStatement pStmt = null;
  int ret;
```

```
    try {
      try {
        String insertString = " INSERT INTO CZ_CONFIG_ATTRIBUTES (CONFIG_HDR_ID, CONFIG_REV_NBR,
CONFIG_ITEM_ID, ATTRIBUTE_CATEGORY";
        String valueString = " VALUES(?,?,?,?";

        // Build an insert string based on the attribute name for the attribute context.
        Collection keys = attributes.keySet();
        Iterator keyIter = keys.iterator();
        while (keyIter.hasNext()) {
          insertString = insertString + "," + getAttributeFieldName (attrContext,
(String)keyIter.next(), ctx);
          valueString = valueString + ",?";
        }
        insertString = insertString + ")";
        valueString = valueString + ")";

        String sql = insertString + valueString;
        pStmt = conn.prepareStatement(sql);

        //Bind/set values to the parameters
        // Add config_hdr_id.
        pStmt.setLong(1,config.getConfigHeaderIdLong());
        // Add config_rev_nbr.
        pStmt.setLong(2,config.getConfigHeaderRevisionLong());
        // Add config_item_id.
        pStmt.setLong(3,node.getConfigItemID());
        // Add flexfield context for attribute.
        pStmt.setString(4,attrContext );

        // Iterate over attributes.
        int n = 5;
        List attrValues = new ArrayList();
        attrValues.addAll(attributes.values());
        Iterator iter = attrValues.iterator();

        while (iter.hasNext()) {
          pStmt.setString(n++, iter.next().toString());
        }

        ret = pStmt.executeUpdate();
      } finally {
        if(pStmt != null) pStmt.close();
      }
```

```
    } catch (SQLException e) {
      throw new RuntimeException("Error Inserting into CZ_CONFIG_ATTRIBUTES table" + e);
    }
    return ret;
  }


  /**
   * Return the column name in CZ_CONFIG_ATTRIBUTES to write this attribute value to,
   * given an attribute context name, attribute name, and database context.
   */
  public String getAttributeFieldName (String attrContext, String attribute, Context ctx) {
    Connection conn = ctx.getJDBCConnection(ctx);
    ResultSet rs = null;
    PreparedStatement pStmt = null;
    String sql, columnName = null;

    try {
      try {
        sql = "SELECT dfu.application_column_name " +
              "FROM FND_DESCR_FLEX_COLUMN_USAGES DFU, " +
              "FND_DESCRIPTIVE_FLEXS FDL , FND_APPLICATION FAN " +
              "WHERE fan.application_short_name = 'CZ' " +
                          "AND dfu.application_id = fan.application_id " +
              "AND fdl.application_id = fan.application_id " +
              "AND fdl.APPLICATION_TABLE_NAME = 'CZ_CONFIG_ATTRIBUTES' " +
              "AND dfu.descriptive_flexfield_name = fdl.descriptive_flexfield_name " +
              "AND dfu.descriptive_flex_context_code = ? " +
              "AND end_user_column_name  = ? ";

        pStmt = conn.prepareStatement(sql);
        pStmt.setString(1,attrContext );
        pStmt.setString(2,attribute );

        rs = pStmt.executeQuery();
        if (rs.next()) {
          columnName = rs.getString(1);
        } else {
          throw new RuntimeException("No column name found for context = " + attrContext + " and
attribute = " + attribute);
        }
      } finally {
        // close all
        if (rs != null) rs.close();
        if (pStmt != null) pStmt.close();
```

```
    }
  } catch (SQLException e) {
    throw new RuntimeException("Error querying attribute names");
  }

  return columnName;
}

/**
 * Clears all attribute data for this configuration from
 * the CZ_CONFIG_ATTRIBUTES table.
 */
public void clearConfigAttributes(Configuration config) throws SQLException {
  long configHdrId = config.getConfigHeaderIdLong();
  long configHdrRev = config.getConfigHeaderRevisionLong();
  Connection conn = config.getContext().getJDBCConnection();
  PreparedStatement pStmt = null;
  int ret;
  try {
    String sql = "DELETE FROM CZ_CONFIG_ATTRIBUTES WHERE CONFIG_HDR_ID=? AND CONFIG_REV_NBR=?";
    pStmt = conn.prepareStatement(sql);
    pStmt.setLong(1, configHdrId);
    pStmt.setLong(2, configHdrRev);
    ret = pStmt.executeUpdate();
  } finally {
    if (pStmt != null) pStmt.close();
  }
}


/**
 * Returns a map of attributes associated with this BOM node by reading
 * its properties from the database. The map is keyed by the attribute
 * index and the values are Attribute objects.
 */
private Map getAttributes(BomNode node) {
  HashMap attributes =  new HashMap();
  int defaultMode = 0;
  // Iterate over the properties and see if we have any attribute properties.
  Iterator iter = node.getProperties().iterator();
  while (iter.hasNext()) {
    Property prop = (Property)iter.next();
    String name = prop.getName();
    // All attribute properties start with "ATTR_".
    if (name.startsWith("ATTR_")) {
```

```
        // Get the index of this attribute.
        int index = 0;
        int beginningIndex = new String("ATTR_"). length();
        StringTokenizer tokens = new StringTokenizer(name.substring(beginningIndex), "_", false);
        if (tokens.hasMoreTokens()) {
          try {
            index = Integer.parseInt(tokens.nextToken());
          } catch (NumberFormatException nfe) {
            // This could be the default mode.
            if (name.endsWith("MODE")) {
              defaultMode = prop.getIntValue();
              continue;
            } else {
              throw new RuntimeException("Attribute properties on BOM nodes should follow " +
                                         "the pattern ATTR_int_* where 'int' is an integer. ");
            }
          }
        }
        // If we do not already have an attribute with this index, create one
        // and put it in the map.
        Attribute attr = (Attribute)attributes.get(new Integer(index));
        if (attr == null) {
          attr = new Attribute(node, index);
          attributes.put(new Integer(index), attr);
        }
        // Assign the appropriate property to this attribute.
        if (name.endsWith("PATH")) {
          attr.setPath(prop.getStringValue());
        }
        else if (name.endsWith("MODE")) {
          attr.setMode(prop.getIntValue());
        }
        else if (name.endsWith("CONTEXT")) {
          attr.setAttrContext(prop.getStringValue());
        }
        else if (name.endsWith("NAME")) {
          attr.setName(prop.getStringValue());
        }
        else {
          throw new RuntimeException("Attribute properties on BOM nodes should end up with " +
                                     "with either \"PATH\", \"MODE\", \"CONTEXT\", or
\"NAME\"");
        }
      }
    }
```

```
  // Apply the default mode on all attributes.
  iter = attributes.values().iterator();
  while (iter.hasNext()) {
    Attribute attribute = (Attribute)iter.next();
    if (!attribute.hasMode()) {
      attribute.setMode(defaultMode);
    }
  }

  return attributes;
}



/**
 * Class to represent an Attribute of a BOM node.
 */
private class Attribute implements com.sun.java.util.collections.Comparable {
  private BomNode m_node;
  private int m_index;
  private String m_path;
  private int m_mode;
  private String m_attrContext;
  private String m_name;
  private RuntimeNode m_feature;
  private boolean found = false;

  protected Attribute(BomNode node, int index) {
    m_node = node;
    m_index = index;
    m_path = null;
    m_mode = -1;
    m_attrContext = null;
    m_name = null;
    m_feature = null;
  }
  public String getPath() {
    return m_path;
  }

  public boolean hasMode() {
    return (m_mode != -1);
  }
```

```
public int getMode() {
  return m_mode;
}

public String getAttrContext() {
  if (m_attrContext == null && getFeature() != null) {
    Property prop = m_feature.getPropertyByName("ATTR_CONTEXT");
    if (prop != null) {
      m_attrContext = prop.getStringValue();
    }
  }
  return m_attrContext;
}

public String getName() {
  if (m_name == null && getFeature() != null) {
    Property prop = m_feature.getPropertyByName("ATTR_NAME");
    if (prop != null) {
      m_name = prop.getStringValue();
    }
  }
  return m_name;
}
public RuntimeNode getFeature() {
  // If the feature has not been found previously then this method will first
  // try to find it. This is done as follows:
  // First traverse up the tree from this BomNode to find the Bom Model that
  // it belongs to. Then get each child node's name from the path (the names are
  // separated by ".") and do a getChildByName() to traverse down from the model
  // node until the feature is found at the end of the path.
  if (!found) findFeature();
  return m_feature;
}


/**
 * Returns the attribute's value as an integer.
 */
public int getIntValue() {
  if (hasIntegerValue()) {
    if (m_feature instanceof IntegerNode) {
      return ((IntegerNode)m_feature).getIntValue();
    } else {
      return ((CountFeature)m_feature).getCount();
    }
```

```
      } else {
        throw new RuntimeException("This attribute does not have integer value");
      }
    }

    /**
     * Returns the attribute's value as a string.
     */
    public String getStringValue() {
      if (hasStringValue()) {
        if (m_feature instanceof TextNode) {
        return ((TextNode)m_feature).getTextValue();
        } else {
          // Return the name of the only selected option in is this option feature
          return ((IRuntimeNode)(((OptionFeatureNode)m_
feature).getSelectedOptions().get(0))).getName();
        }
      } else {
        throw new RuntimeException("This attribute does not have String value");
      }
    }

    /**
     * Returns the attribute's value as a double.
     */
    public double getDecimalValue() {
      if (hasDecimalValue()) {
        return ((DecimalNode)m_feature).getDecimalValue();
      } else {
        throw new RuntimeException("This attribute does not have decimal value");
      }
    }

    /**
     * Returns the attribute's value as a boolean.
     */
    public boolean getBooleanValue() {
      if (hasBooleanValue()) {
        return ((BooleanFeature)m_feature).isTrue();
      } else {
        throw new RuntimeException("This attribute does not have boolean value");
      }
    }

    /**
```

```
     * Returns true if attribute feature has an integer value.
     */
    public boolean hasIntegerValue() {
      RuntimeNode feature = getFeature();
      return (feature instanceof IntegerNode || feature instanceof CountFeature);
    }


    /**
     * Returns true if attribute feature has a decimal value.
     */
    public boolean hasDecimalValue() {
      RuntimeNode feature = getFeature();
      return (feature instanceof DecimalNode);
    }

    /**
     * Returns true if attribute feature has a boolean value.
     */
    public boolean hasBooleanValue() {
      RuntimeNode feature = getFeature();
      return (feature instanceof BooleanFeature);
    }


    /**
     * Returns true if attribute feature has a string value.
     */
    public boolean hasStringValue() {
      RuntimeNode feature = getFeature();
      return (feature instanceof TextNode ||
              (feature instanceof OptionFeatureNode &&
((OptionFeatureNode)feature).getSelectedOptions().size() == 1));
    }

    protected void setPath(String path) {
      m_path = path;
    }

    protected void setMode(int mode) {
      m_mode = mode;
    }

    protected void setAttrContext(String context) {
      m_attrContext = context;
```

```
}

protected void setName(String name) {
  m_name = name;
}

private void findFeature() {
  // Set the found flag to true once the path has been set
  // so that we do not attempt to find it again
  if (getPath() != null) {
    found = true;
  } else return;

  RuntimeNode node = m_node;
  while (!(node instanceof BomModel)) {
    node = (RuntimeNode)node.getParent();
  }

  if (node instanceof BomModel) { // We found the model
    StringTokenizer tokenizer = new StringTokenizer(getPath(), ".", false);
    while (tokenizer.hasMoreTokens()) {
      try {
        node = (RuntimeNode)node.getChildByName(tokenizer.nextToken());
        if (node instanceof ComponentSet) {
          throw new RuntimeException("Cannot refer to a multiply instantiable Component: " +
            node.getName() + " in the Attribute Path");
        }
      } catch (NoSuchChildException nsce) {
        throw new RuntimeException("No attribute feature found in Path " +
          getPath() + " for Node " + m_node.getName());
      }
    }
    m_feature = node;
  }
}

public String toString() {
  return getAttrContext() + getName();
}

public boolean equals(Object obj) {
  if (obj instanceof Attribute) {
    String myAttr = getAttrContext();
    String myName = getName();
    String objAttr = ((Attribute)obj).getAttrContext();
```

```
        String objName = ((Attribute)obj).getName();
        if (myAttr != null && objAttr != null && myAttr.equals(objAttr) &&
            myName != null && objName != null && myName.equals(objName)) {
          return true;
        }
      }
      return false;
    }

    public int compareTo(Object o) {
      if (equals(o)) {
        return 0;
      } else {
        return this.toString().compareTo(o.toString());
      }
    }

  }

}
```

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 users** 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 **attributes**.

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 **configurations**. *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 attributes**.

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 **options**. *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 models**.

DHTML

Dynamic Hypertext Markup Language

DIO

See **Data Integration Object**.

distributed computing

Running various **components** 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 **Options**, **Totals**, **Resources**, UI controls, and **components**.

end user

The ultimate user of the runtime **Oracle Configurator**. The types of end users vary by project but may include salespeople or distributors, administrative office staff, marketing personnel, order entry personnel, product engineers, or customers directly accessing the application via a Web browser or kiosk. *Compare* **user**.

enterprise

The **systems** and **resources** of a business.

environment

The arena in which software tools are used, such as operating system, applications, and **server** processes.

ERP

Enterprise Resource Planning. A software system and process that provides automation for the customer's back-room operations, including order processing.

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

P

R

S

T

U

V

W