# Oracle® Configurator

Modeling Guide

Release 12.2

**Part No. E48818-01**

September 2013

Oracle Configurator Modeling Guide, Release 12.2

Part No. E48818-01

Primary Author: Margot Murray

Contributor: Tom Myers

# Contents

# Send Us Your Comments

**Oracle Configurator Modeling Guide, Release 12.2**

**Part No. E48818-01**

Oracle welcomes customers' comments and suggestions on the quality and usefulness of this document. Your feedback is important, and helps us to best meet your needs as a user of our products. For example:

- Are the implementation steps correct and complete?
- Did you understand the context of the procedures?
- Did you find any errors in the information?
- Does the structure of the information help you with your tasks?
- Do you need different information or graphics? If so, where, and in what format?
- Are the examples correct? Do you need more examples?

If you find any errors or have any other suggestions for improvement, then please tell us your name, the name of the company who has licensed our products, the title and part number of the documentation and the chapter, section, and page number (if available).

Note: Before sending us your comments, you might like to check that you have the latest version of the document and if any concerns are already addressed. To do this, access the new Oracle E-Business Suite Release Online Documentation CD available on My Oracle Support and www.oracle.com. It contains the most current Documentation Library plus all documents revised or released recently.

Send your comments to us using the electronic mail address: appsdoc_us@oracle.com

Please give your name, address, electronic mail address, and telephone number (optional).

If you need assistance with Oracle software, then please contact your support representative or Oracle Support Services.

If you require training or instruction in using Oracle software, then please contact your Oracle local office and inquire about our Oracle University offerings. A list of Oracle offices is available on our Web site at www.oracle.com.

# Preface

## Intended Audience

Welcome to Release 12.2 of the *Oracle Configurator Modeling Guide.*

This book contains information you need for designing configuration models that are best suited to Oracle Configurator. This book focuses on model design and does not present other information about planning Oracle Configurator projects such as preparing your site and team for implementation and scheduling tasks.

Use this document together with the other books in the Oracle Configurator documentation set to prepare for and implement high performance configuration model designs.

This preface describes how the book is organized, who the intended audience is, and how to interpret the typographic conventions.

This guide is intended for Oracle Consultants and implementers of Oracle Configurator who have completed Oracle Configurator training or have experience using Oracle Configurator Developer. Oracle Configurator training is available through Oracle University.

Before using this book, you should already have a working knowledge of your business processes, how to create configuration models using Oracle Applications, and the other books in the Oracle Configurator documentation set.

See Related Information Sources on page xiii for more Oracle E-Business Suite product information.

## Documentation Accessibility

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

## Access to Oracle Support

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

## Structure

**1 Introduction**

**2 Planning Your Model Design**

This chapter describes the high level flow of starting a project and designing configuration models. The chapter also presents guidelines to help you plan your Oracle Configurator project and determine which design questions you should ask.

For information about planning and starting an Oracle Configurator project, see the Oracle Configurator training available through Oracle University.

**3 Starting Your Model Design**

Before reading this chapter, review the planning guidelines in Chapter 2 *Planning Your Model Design* to help identify areas of special considerations in an Oracle Configurator project.

This chapter presents the design questions you should ask yourself in order to identify which best practices apply to your project.

Read through the questions on the following pages to help you identify relevant design decisions that are presented as some of the best practices and case studies in the following chapters. This list of design questions is intended as a starting point and does not include all the questions that are useful to ask as you begin your Oracle Configurator project.

**4 Best Practices**

Before reading this chapter, review the design questions in Starting Your Model Design, page 4-1 to help you identify best practices that are relevant to your Oracle Configurator project.

Applying the following best practices in various combinations will improve the performance, usability, maintenance, and scalability of your configuration models.

Many of these best practices include detailed instructions. You can gain useful information by reading all of them, even though some are not directly connected to a design question. To understand these best practices, you must be familiar with the specifics of creating configuration models. See the *Oracle Configurator Developer User's Guide* for details.

**5 Many Large BOM Models**

This chapter describes an Oracle Configurator project involving many large BOM Models with much explicit and repetitive structure that is best modeled as a single top-level BOM Model containing a deep hierarchy of generic structure and abstractions.

**6 Many BOM Items**

This chapter describes an Oracle Configurator project involving many BOM Items that are not orderable and could be better implemented as Features or configuration attributes. Configuration attributes are explained in *Oracle Configurator Methodologies.*

**A SQL Queries**

This appendix contains some SQL*Plus queries that are useful for calculating the total number of nodes in a configuration model, and the number of different types of nodes in a configuration model.

**Common Glossary for Oracle Configurator**

## Related Information Sources

For a full list of documentation resources for Oracle Configurator, see the Oracle Configurator Release Notes for this release.

For a full list of documentation resources for Oracle Applications, see Oracle Applications Documentation Resources, on MetaLink, Oracle's technical support Web site.

Additionally, be sure you are familiar with current release or patch information for Oracle Configurator on MetaLink, Oracle's technical support Web site.

### Integration Repository

The Oracle Integration Repository is a compilation of information about the service endpoints exposed by the Oracle E-Business Suite of applications. It provides a complete catalog of Oracle E-Business Suite's business service interfaces. The tool lets users easily discover and deploy the appropriate business service interface for integration with any system, application, or business partner.

The Oracle Integration Repository is shipped as part of the E-Business Suite. As your instance is patched, the repository is automatically updated with content appropriate for the precise revisions of interfaces in your environment.

You can navigate to the Oracle Integration Repository through Oracle E-Business Suite Integrated SOA Gateway.

## Do Not Use Database Tools to Modify Oracle E-Business Suite Data

Oracle STRONGLY RECOMMENDS that you never use SQL*Plus, Oracle Data Browser, database triggers, or any other tool to modify Oracle E-Business Suite data unless otherwise instructed.

Oracle provides powerful tools you can use to create, store, change, retrieve, and maintain information in an Oracle database. But if you use Oracle tools such as SQL*Plus to modify Oracle E-Business Suite data, you risk destroying the integrity of your data and you lose the ability to audit changes to your data.

Because Oracle E-Business Suite tables are interrelated, any change you make using an

Oracle E-Business Suite form can update many tables at once. But when you modify Oracle E-Business Suite data using anything other than Oracle E-Business Suite, you may change a row in one table without making corresponding changes in related tables. If your tables get out of synchronization with each other, you risk retrieving erroneous information and you risk unpredictable results throughout Oracle E-Business Suite.

When you use Oracle E-Business Suite to modify your data, Oracle E-Business Suite automatically checks that your changes are valid. Oracle E-Business Suite also keeps track of who changes information. If you enter information into database tables using database tools, you may store invalid information. You also lose the ability to track who has changed your information because SQL*Plus and other database tools do not keep a record of changes.

# Part 1

## Modeling Guidelines

Oracle Configurator provides tools for a wide range of solutions. This section consists of chapters that are designed to lead you from relevant planning guidelines through important design questions to best practices that leverage the strengths and requirements of Oracle Configurator.

# 1 Introduction

This chapter covers the following topics:

- Scope of this Guide
- Conventions
- Product Support

## Scope of this Guide

This book contains a table of contents, examples, tables, figures, a reader comment form, a preface, several chapters, a glossary, and an index.

- Chapter 1 - Introduction presents an overview of the materials covered in each chapter, Product Support, and Troubleshooting information.
- Chapter 2 - Planning Your Model Design describes the high level flow of starting a project and designing configuration models. The chapter also presents guidelines to help you plan your Oracle Configurator project and determine which design questions you should ask.
- Chapter 3 - Best Practices explains best practices for designing a configuration model with optimal performance, maintainability, and scalability.
- Chapter 4 - Many Large BOM Models describes an Oracle Configurator project involving many large BOM Models with a great deal of explicit and repetitive structure that is best modeled as a single top-level BOM Model containing a deep hierarchy of generic structure and abstractions.
- Chapter 5 - Many BOM Items describes an Oracle Configurator project involving many BOM Items that are not orderable and could be better implemented as Features or configuration attributes.
- The Glossary contains terms that are used throughout the Oracle Configurator

documentation set.

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

## Conventions

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

In rule examples, the order in which the rules are presented does not indicate a sequence in which Oracle Configurator processes the rules. You cannot assume a particular sequence, and the sequence can differ each time the rules are processed by the Oracle Configurator engine.

The following conventions are also used in this manual:

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

| Convention | Meaning |
|---|---|
| $ | The dollar sign represents the DIGITAL Command Language prompt in Windows and the Bourne shell prompt in Digital UNIX. |
| % | The percent sign alone represents the UNIX prompt. |
| `name( )` | In text other than code examples, the names of programming language methods and functions are shown with trailing parentheses. The parentheses are always shown as empty. For the actual argument or parameter list, see the reference documentation. This convention is not used in code examples. |

## Product Support

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

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

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

Log 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:
    1. Choose **Product**: Oracle Configurator or Oracle Configurator Developer
    2. Choose **Type of Problem**: Oracle Configurator Generic Issue template
5. Provide the information requested in the iTAR template.

You can also find product-specific documentation and other useful information using Metalink. For a complete listing of available Oracle Support Services and phone numbers, see:

www.oracle.com/support/

## Troubleshooting

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

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

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

- The *Oracle E-Business Suite System Administrator's Guide* for descriptions of the Oracle Applications Manager UI screens that allow System Administrators to set up logging profiles, review Java system properties, search for log messages, and so on.
- The *Oracle E-Business Suite Supportability Guide*, which includes logging guidelines for both System Administrators and developers, and related topics.
- The Oracle Application Framework Documentation Resources, Release 12 on Metalink.

# 2

# Planning Your Model Design

This chapter describes the high level flow of starting a project and designing configuration models. The chapter also presents guidelines to help you plan your Oracle Configurator project and determine which design questions you should ask.

For information about planning and starting an Oracle Configurator project, see the Oracle Configurator training available through Oracle University.

This chapter covers the following topics:

- Overview of Designing a Configuration Model
- Planning Guidelines Relevant to Model Design

## Overview of Designing a Configuration Model

A proven, repeatable methodology for designing and creating configuration models includes the following design considerations:

1. Model structure design
2. User Interface design
3. Rule design

Begin by designing the Model structure, which may involve leveraging BOM Model data that already exists in Oracle Bills of Material, and adding guided buying or selling Features and Options.

As you design your Model structure, consider your requirements for the user interface. A runtime UI may be automatically generated in Configurator Developer from your Model structure, and your UI requirements often may drive the structure of your guided buying or selling components. However, the templates that you use to generate UIs in Configurator Developer ultimately provide a great deal of control over the UI's appearance and behavior.

Once you have designed an initial Model structure, apply rules among its elements.

Rules typically represent the most complex aspect of your configuration model and should be well thought-out during the design process.

The flow of steps to design a configuration model is often iterative, but by following the suggestions in this Modeling Guide you should be able to reduce the number of iterations during the design process. Planning Guidelines Relevant to Model Design, page 2-2 describes a set of guidelines to follow as you plan your implementation and identify what design decisions are necessary.

> **Caution:** Do not use Oracle Configurator to emulate or replace functionality that is missing from other products or processes. For example, using Oracle Configurator to emulate the Configure-To-Order business process because a specific requirement is not addressed in the current release could have undesirable consequences.

## Planning Guidelines Relevant to Model Design

This section presents some of the planning guidelines that can help you make appropriate model design decisions in the following areas:

- BOM Model Design or Redesign, page 2-2
- End-User Expectations , page 2-5
- Rule Design, page 2-5

These planning guidelines are intended as a starting point for designing your configuration model. You may need to make additional planning decisions that are specific to your business or Oracle Configurator project.

> **Note:** Even if your implementation does not leverage BOM Models, review the design questions and best practices listed in BOM Model Design or Redesign, page 2-2 before beginning your configuration model design.

After reading the following planning guidelines, read through the design questions presented in Chapter 3, *Starting Your Model Design* to help identify best practices that are relevant to your project. For more information, see Best Practices, page 4-1.

### BOM Model Design or Redesign

The guidelines provided for BOM Model design assume that your Oracle Configurator project is integrated with the Oracle *e*Business Suite.

The BOM is what is configured and ordered. While it needs to be structured for your Enterprise Resource Planning (ERP) process, it also may need to be simplified for end user comprehension and usability during configuration. Even if you believe there is no flexibility for changing your BOM Models, the consequences in implementation effort, maintenance costs, usability issues, and performance may persuade you to make some adjustments. In this case, be aware of the limitations this lack of flexibility imposes and plan your runtime architecture accordingly.

The table below lists some common BOM Model characteristics and specific adjustments that must be made to maintain Oracle Configurator performance and usability.

***BOM Model Design and Impact on Runtime Architecture***

| BOM Model Design Characteristic | Metric | Recommended Adjustment to Runtime Architecture |
|---|---|---|
| Many BOM Option Classes and BOM Standard Items | Typically, BOM Models with over 10,000 items. | More JVMs<br>More dedicated JVMs<br>Increased (large) JVM heap size<br>Model structure abstraction to hide the BOM from the UI (easier option selection) |
| Large BOM Option Classes | Typically, those containing over 100 BOM Standard Items. | More JVMs<br>More dedicated JVMs<br>Increased (large) JVM heap size<br>Model structure abstraction to hide the BOM from the UI (easier option selection) |
| Multiple BOM Models with similar structure | References to two or more BOM Models that contain only minor differences (few unique items) | Additional memory<br>Additional hardware<br>Increased (large) JVM heap size |
| BOM Option Classes that are repeated in multiple BOM Models | Any duplication of BOM Option Classes, (especially those containing over 100 Standard Items) | Additional memory<br>Additional hardware<br>Increased (large) JVM heap size |

## Planning Guidelines

Always begin your Oracle Configurator project by examining the structure of your BOM Models. If your BOM Models exhibit any of the characteristics listed in BOM Model Design and Impact on Runtime Architecture, refer to the design questions listed in Chapter 3, Starting Your Model Design, to review best practices that may apply to your project.

Another important consideration related to BOM Model design is that the size of the configuration model directly affects the time it takes to load the model and display the UI in a runtime Oracle Configurator. In other words, the bigger the model, the longer the initial load. For this reason, Oracle recommends preloading models so that end users experience better performance throughout the configuration session. For more information on preloading models, see the *Oracle Configurator Performance Guide*.

The database size of a configuration model is measured in total number of nodes. Appendix A, SQL Queries, page A-1 presents queries used to find out how many nodes are in your configuration model.

The goal is to streamline your BOM Models so that the resulting configuration models are highly maintainable and scalable as your business grows, with optimal usability and performance at runtime.

> **Note:** Both the name and description of BOM items are imported into Configurator Developer with the BOM Model. If either is used to generate UI captions, make sure they are meaningful to end users.
>
> Be aware that only BOM Standard Items that are defined as optional in Oracle Bills of Material appear in Oracle Configurator; BOM Standard Items that are mandatory are not displayed.

BOM Model redesign may involve changes such as:

- Reducing repetitive or similar structure into abstract elements that should be referenced (abstraction)
- Changing entity types: For example, changing a commonly used Option Class to a BOM Model.
- Eliminating unnecessary items: For example, removing BOM Standard Items from the BOM and converting them to attributes.

These and other BOM Model redesign best practices are discussed in Chapter 4, *Best Practices*.

Note that any changes to the BOM Model may have an impact on downstream operations. Optimal BOM Model design balances the needs of all applications that leverage it. Every item in a BOM is ordered, and either gets manufactured or provisioned. Therefore, any changes to the BOM to improve Oracle Configurator performance may require additional rework to support requirements in other applications, such as Pricing, Order Management, Install Base, Shop Floor Management, and so on.

## End-User Expectations

A single configuration model can support multiple User Interfaces (UIs). Typically, Oracle Configurator Developer is used to automatically generate end-user UIs. Since you may base the UI structure on the Model structure, keep UI considerations and end-user expectations in mind as you design your configuration model. If your expectations for the UI are based on a legacy system, it may be difficult to leverage some of the default features and functionality that is provided "out-of-the-box" by a UI generated in Oracle Configurator Developer. In other words, legacy UI design and behaviors may require you to either extensively customize a generated UI or create a completely custom UI from scratch. Writing Configurator Extensions to implement custom behavior may also be necessary.

### Planning Guidelines

Examine and understand how your end users will interact with the configuration UI. The following circumstances are relevant to model design:

- Your runtime Oracle Configurator needs to exhibit any of the following behaviors:
  - UI opens with options already selected
  - Product complexity hidden from end users by high-level questions or minimal required selections
- Your end users need to perform any of the following actions in the UI:
  - Select from a long list of values
  - Enter as input the characteristics of a part (such as dimensions, instructions, a monogram, or similar information)
  - Add or remove elements of the configuration

## Rule Design

One of the most critical and potentially time-consuming activities in constructing your configuration model is designing and constructing the rules that govern what the end user can select to make a valid configuration. You need to build rules that express relations and behaviors among the Components, Features, Options, BOM Option Classes, and BOM Standard Items of your Model.

## Planning Guidelines

The number and complexity of the rules are a factor in determining the size of your configuration model. Independent of whether structure is a factor, fewer than 500 rules is generally a small model, 500 to 2,000 rules is a medium model, and over 2,000 rules is a large model. When working with models containing a large number of rules, performance and usability can be a concern.

It is important to examine and understand all of the rules that define your business or constrain your products before proceeding with your Configurator implementation. You may also want to consider using Configuration Attributes to, for example, insert values that are stored by the host application into the configuration model at the beginning of a configuration session, as the initial values of specified Features. For more information about configuration attributes, see *Oracle Configurator Methodologies.*

> **Note:** Prototyping a configuration model based on a preliminary and incomplete rule information will likely result in a sub-optimal model, or require additional redesign iterations.

The following circumstances are relevant to model design:

- You have over 2,000 rules, or you have rules that exhibit any of the following characteristics:
    - Complexity with subexpressions, especially if the subexpressions are used in multiple rules
    - Specific order or sequence in which the rules must be executed
- You use the following types of rules in your configuration model:
    - Compatibility
    - Comparison
    - Defaults
    - Rules that use the NotTrue operator
- You are importing rules from a legacy system

  Before importing rules from a legacy system, be sure to carefully analyze them and consider whether they could be more efficiently implemented by creating new rules from scratch in Oracle Configurator Developer. Importing legacy rules directly into Configurator Developer will likely result in less than optimal rules that contain duplicate sub-expressions.

## Lifecycle and Maintainability Expectations

It is important to carefully consider how long your configurable product will be available, plan for how you will maintain it until it is removed from production, and set company expectations accordingly. You should understand the maintenance requirements of your product - such as how often new items will be added, existing items will be modified, and so on - and the impact these tasks will have on your overall business. For example, when the model needs to be updated, will it need to be taken out of production? If so, for how long?

Consider reviewing your business processes to determine if there are areas that can be changed to minimize the impact that configuration model maintenance has on your business, and plan for the kinds of changes that will occur throughout the life cycle of your product.

# 3 Starting Your Model Design

Before reading this chapter, review the planning guidelines in Chapter 2 *Planning Your Model Design* to help identify areas of special considerations in an Oracle Configurator project.

This chapter presents the design questions you should ask yourself in order to identify which best practices apply to your project.

Read through the questions on the following pages to help you identify relevant design decisions that are presented as some of the best practices and case studies in the following chapters. This list of design questions is intended as a starting point and does not include all the questions that are useful to ask as you begin your Oracle Configurator project.

This chapter covers the following topics:

## Do You Expect Configurator to Display Large Lists of Options?

Many options means more than 100 selectable items in one list of options. Hundreds or thousands of selectable options displayed for end users cause usability and performance problems.

### Other Ways To Phrase This Question

Does your BOM Model contain any BOM Option Classes with more than 100 selectable items?

What is an efficient way to design a BOM Option Class or Option Feature containing a large number of option selections?

### Structure Decisions

Apply the following best practices when optimizing the design of large numbers of options or inventoried parts:

- Optional and Multiple Instantiation, page 4-9
- Explicit Model Structure Versus Abstractions, page 4-2
- Grouped Versus Ungrouped Items, page 4-18

### Rule Decisions

Apply the following best practices when optimizing the rule design of large numbers of options:

- Large Option Features and Option Classes, page 4-18

### UI Decisions

Apply the following best practices when optimizing the UI design of large numbers of options:

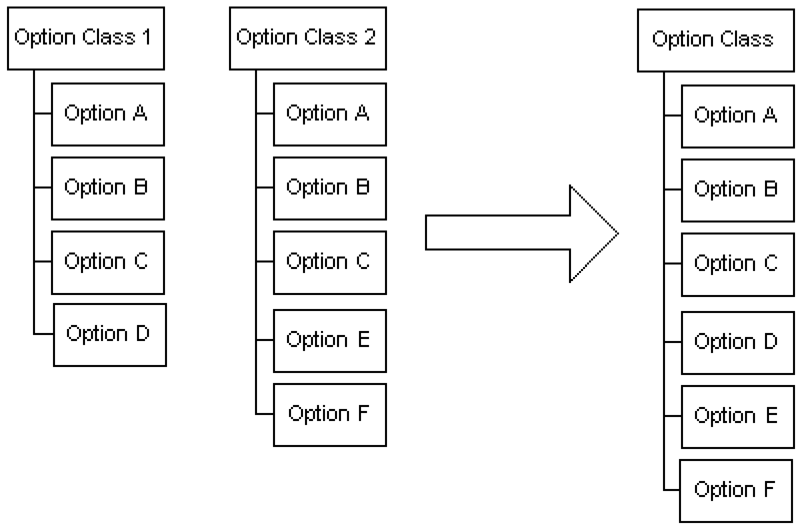
- Optimizing User Interface Performance, page 4-40

## Are the Same Product Elements Repeated in Separate Models?

Any identical or similar model structure that is duplicated at least twice in your Model can be considered as a repeating product element. For example, BOM Option Classes that are similar share some options in common, but also contain different options, as shown in Repeating Similar Product Elements, page 3-3 graphic.

The graphic shows two BOM Option Classes, "Option Class 1," and "Option Class 2." Each BOM Option Class contains Options, some of which are repeated in the two BOM Option Classes, and some of which are not. "Option Class 1" contains Options A, B, C, and D, while "Option Class 2" contains Options A, B, C, E, and F. An arrow points to a third BOM Option Class that contains a union of the Options A, B, C, D, E, and F.

***Repeating Similar Product Elements***



Repeating Similar Product Elements, page 3-3 shows that when the separate and duplicated structure is combined, the total structure that has to be loaded, instantiated, or maintained is smaller. In this example, the number of nodes has been reduced from 11 to 7, a reduction of 36%. When combining structure in this way, you need appropriate rules to ensure that only the relevant options are visible or available where the abstraction appears in the runtime Oracle Configurator.

## Other Ways To Phrase This Question

Are there high degrees of similarity across multiple models?

Are the same BOM Option Classes used in multiple BOM Models?

Where in your BOM Model structure can you take advantage of referencing?

Would combining several similar BOM Option Classes into one BOM Option Class containing a union of all possible options decrease the overall number of options that need to be loaded or instantiated in Oracle Configurator?

## Structure Decisions

If the same product elements repeat in separate models, use the following best practices to optimize your model design:

- Explicit Model Structure Versus Abstractions, page 4-2
- Explicit Model Structure Versus References, page 4-7
- Shallow Versus Nested or Deep Hierarchy, page 4-14

## Are You Modeling Many Related Products?

Hundreds or thousands of separate products with similar or duplicate items cause maintenance, performance, and scalability problems.

### Other Ways To Phrase This Question

Do your end users configure similar products from a large product line?

Do your products contain many common elements, repetitive structure, or similar sets of selections?

Do you use multiple BOM Models to represent different combinations of the same general product?

When should you use generic BOM Models and when should you maintain a set of explicit BOM Models for the same structure?

### Structure Decisions

Apply the following best practices when optimizing the design of large numbers of related products:

- Explicit Model Structure Versus Abstractions, page 4-2
- Explicit Model Structure Versus References, page 4-7
- Optional and Multiple Instantiation, page 4-9
- Shallow Versus Nested or Deep Hierarchy, page 4-14

## Do You Need Default Values Set Automatically?

Default values present suggested or likely values that end users could change if desired but do not need to change to complete the configuration. There are many ways to implement the effect of default values, some of which, such as Defaults rules, are more costly to Configurator performance than others.

### Other Ways To Phrase This Question

Do end users expect to see default, initial, or recommended values already filled in when they start up Oracle Configurator?

Do end users expect to see default values filled in automatically to accelerate completion of the configuration as they make selections?

When is it efficient to add defaults to end user requests?

### Rule Decisions

Apply the following best practices when designing pre-selected or default values:

- Defaults Rules Versus Alternatives to Default Selections, page 4-20

## Does Your End User Need to See the Bill of Materials?

If configurations are based on a BOM Model, you need to determine if end users need the BOM items to appear in Oracle Configurator exactly as they appear in Inventory and Bills of Material. With very large BOMs or novice end users, displaying the entire BOM is not optimal or desirable.

### Other Ways To Phrase This Question

Is the end user a product expert who expects to make selections of parts by their part numbers or part descriptions?

Does the end user need guidance in making appropriate selections to create a valid configuration?

When is it efficient to define a guided buying or selling model instead of exposing the BOM items for selection?

### Structure Decisions

If your end users expect to see the BOM, use the following best practices to optimize your model design, especially if your BOM is large:

- Grouped Versus Ungrouped Items, page 4-18

If your end users do not expect to see the BOM, use the following best practices to optimize your model design:

- Guided Buying or Selling, page 4-12

### UI Decisions

If your end users do not expect to see the BOM, use the following best practices to optimize your model design:

- Optimizing User Interface Performance, page 4-40

## Will Configurations Contain Instances of a Component?

The word "component" refers not only to Configurator Developer Components, but any configurable element in a configuration model, including instances of BOM Models, Models, and Components.

Hundreds or thousands of instances that end users must add interactively to the configuration, or that must be instantiated at startup cause usability and performance problems. Adding an instance is more costly to performance than selecting an option.

### Other Ways To Phrase This Question

Do you have many instances of a component with no constraints among the selections within the configuration of each instance?

Do you need your end users to add many instances of a component, or can instances be added programmatically?

### Structure Decisions

Apply the following best practices when optimizing the design of adding instances:

- Optional and Multiple Instantiation, page 4-9

### Rule Decisions

Apply the following best practices when optimizing the rule design of adding instances with a Configurator Extension:

- Configurator Extension Design, page 4-42

### User Interface Decisions

Apply the following best practice when optimizing the design of UIs:

- When adding instances programmatically, add them early in the configuration session, prior to the end user making many user selections. Each time an instance is added, the user's inputs are retracted and reasserted thus affecting performance. By adding them at the beginning of the configuration session, this performance problem can be mitigated.

## Will Your Configurator Collect Many End-User Inputs?

This question identifies hundreds or thousands of inputs that are only passed as parameters and contribute to but are not constrained by rules in the configuration session. Designing Oracle Configurator to include collecting such end-user inputs can cause usability and performance problems. In some cases such data input can occur more efficiently outside the configuration session.

### Other Ways To Phrase This Question

Do you expect end-user interactions to be extensive and repetitive?

Do you have end-user inputs that do not participate directly in rules or calculations?

Do you feel that your end users would be able to enter data into a spreadsheet?

Will your end user enter data as option selections or as text or numeric inputs?

### UI Decisions

Apply the following best practices when optimizing the UI design of models with large numbers of end-user inputs:

- Large Amounts of End-User Data Collected Using Configurator Extensions, page 4-42

## Does Configurator Depend on Information Collected Upstream?

Information that is collected by the host application for the configuration session, but is not needed in downstream processes, should be passed to Oracle Configurator as configuration attributes. Structuring this information as orderable items can cause usability and maintenance problems because it bloats the BOM and must be dealt with as order line items. Configuration attributes are explained in *Oracle Configurator Methodologies.*

### Other Ways To Phrase This Question

Do you expect attributes or parameters to be passed into Oracle Configurator?

Is there data besides items and quantities of items that is needed for computation in Oracle Configurator?

Is there non-item information on the order line before configuration that Oracle Configurator needs in computations?

### Structure Decisions

Apply the following best practice when optimizing structure that includes passing attributes into Oracle Configurator:

- Items Versus Alternatives to Items, page 4-14

## Does Configurator Pass Non-Orderable Information Downstream?

Information collected from the configuration session that is needed in downstream processes should be passed as configuration attributes. Structuring this information as orderable items can cause usability and maintenance problems because it bloats the BOM and must be dealt with as order line items.

For more information about configuration attributes, see *Oracle Configurator Methodologies.*

### Other Ways To Phrase This Question

Do you expect attributes or parameters to be passed out of Oracle Configurator?

Is non-item information, meaning not items or their quantities, needed on the order line?

Are the results of computations in Oracle Configurator needed in downstream processing?

Are items used to represent non-manufacturable entities to indicate details about another item or operation? For example, modeling color choices as a set of options, or a range of allowable dimensions for a product as discrete items.

### Structure Decisions

Apply the following best practice when optimizing structure that includes passing attributes out of Oracle Configurator:

- Items Versus Alternatives to Items, page 4-14

## Are Some Selections Disallowed Until Other Selections Are Made?

The purpose of this question is to find out whether you are implementing any Statement Rules that use the NotTrue operator. This is not recommended as using NotTrue may cause propagation issues under certain circumstances if it imposes order in the rules.

### Other Ways To Phrase This Question

Do you have rules that make some options not selected until some other option is selected?

Are you planning on using NotTrue in Statement Rules (rules written in CDL)?

Do you want to disallow some option selections when other options are not selected?

Do you need to impose dependencies among option selections?

### Rule Decisions

Apply the following best practice when optimizing a rule design that imposes order on option selections, or locks the initial value of components:

- NotTrue Logical Function Imposes Order and Causes Locking, page 4-28

### User Interface Decisions

Do not use the NotTrue operator to control the selection state or visibility of options in the runtime User Interface. For example, you may want to disallow some option selections in the UI when other options are not selected, or define specific dependencies among option selections. Such dynamic behaviors can be easily implemented in a User Interface created in Configurator Developer by using display conditions. For details, see the *Oracle Configurator Developer User's Guide*.

## Will Your Rules Include Repeating Patterns or Redundancy?

Repeating patterns or redundancy means that several rules include the same subexpressions or have the same result. This could cause performance issues.

### Other Ways To Phrase This Question

Does your model require calculations that contribute to other calculations?

Are the same Options or Features that are used in AnyTrue or AllTrue expressions used in multiple rules?

### Rule Decisions

Apply the following best practices when optimizing the rule design of models containing common subexpressions or repeated patterns:

- Repetitive Rule Patterns and Redundancy, page 4-24

## Are Your Configuration Rules Based on Legacy Rules?

New rules that are based existing rules from a legacy system, or rules imported directly from a legacy system, are much less likely to perform well in Oracle Configurator. Often, legacy rules contain repeating patterns or introduce redundancy, which means that several rules include the same subexpressions or have the same result.

For best runtime performance, Oracle recommends defining constraints and relations in Configurator Developer, following the general recommendations described in Chapter 4, *Best Practices.*

### Rule Decisions

Apply the following best practices when optimizing the rule design of models containing common subexpressions or repeated patterns:

- Repetitive Rule Patterns and Redundancy, page 4-24

## Do You Need to Express Compatibilities in Your Model?

This means you have many options that require and exclude other options. There are several ways to implement this to achieve optimal performance and maintainability.

### Other Ways To Phrase This Question

Do you need to express incompatibilities in your model?

Are relationships between options or items documented as tables?

Do the Options or Items that are involved in compatibility relationships have relevant property or catalog data?

### Rule Decisions

Apply the following best practices when optimizing a rule design that expresses compatibilities or incompatibilities among options:

- Compatibility Rules, page 4-30

## Do You Need to Express Comparisons in Your Model?

This means you need to compare, balance, or rate one set of options against another set of options. Comparison rules could affect usability by requiring end users to retract selections before being able to continue. Under specific circumstances involving initial values, Comparison rules can also lock during propagation.

### Other Ways To Phrase This Question

Does your model contain comparison logic?

### Rule Decisions

Apply the following best practices when optimizing a rule design that expresses comparisons:

- Comparison Rules, page 4-35

# 4

# Best Practices

Before reading this chapter, review the design questions in Starting Your Model Design, page 4-1 to help you identify best practices that are relevant to your Oracle Configurator project.

Applying the following best practices in various combinations will improve the performance, usability, maintenance, and scalability of your configuration models.

Many of these best practices include detailed instructions. You can gain useful information by reading all of them, even though some are not directly connected to a design question. To understand these best practices, you must be familiar with the specifics of creating configuration models. See the *Oracle Configurator Developer User's Guide* for details.

This chapter covers the following topics:

- Explicit Model Structure Versus Abstractions
- Explicit Model Structure Versus References
- Optional and Multiple Instantiation
- Guided Buying or Selling
- Shallow Versus Nested or Deep Hierarchy
- Items Versus Alternatives to Items
- Large Option Features and Option Classes
- Defaults Rules Versus Alternatives to Default Selections
- Repetitive Rule Patterns and Redundancy
- Number and Complexity of Rules
- NotTrue Logical Function Imposes Order and Causes Locking
- Compatibility Rules
- Comparison Rules
- Connectors with Connection Rules

- Optimizing User Interface Performance
- Large Amounts of End-User Data Collected Using Configurator Extensions
- Configurator Extension Design

## Explicit Model Structure Versus Abstractions

Abstraction is a design approach used to optimize performance and usability of a configuration model by reducing many related products or product elements to generic elements and eliminating repetition. Abstraction may also be as simple as merging overlapping sets of data, as described in Are the Same Product Elements Repeated in Separate Models, page 3-2.

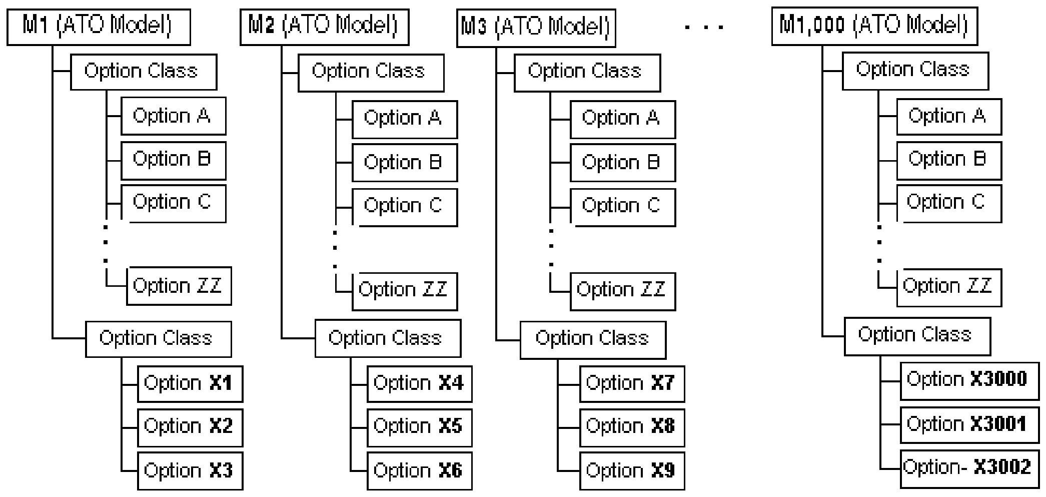
Many companies define BOM Models as an explicit representation of their products with no abstractions to streamline the structure or volume of items. (See the graphic Explicit Structure, page 4-2 for an example.) Loading all BOM Models and presenting entire product lines to end users for selection may result in poor performance and usability. An alternative approach transforms many explicit BOMs into one root BOM referencing a smaller number of configurable components that are abstractions of related structure. Such a model loads faster, thereby improving performance. Configurator usability is also improved by confining end-user access to only those optionally added instances of the configurable components that are needed in the configuration.

### Explicit Structure

Explicit Model structure contains no abstractions and no References. The graphic Explicit Model Structure, page 4-3 shows a series of explicit Models, each containing the same BOM Option Class.

The graphic shows a simple Model repeated 1,000 times. Each version consists of a BOM Option Class with a large number of the same Options and another BOM Option Class containing a few unique Options.

***Explicit Model Structure***



Explicit Model Structure, page 4-3 illustrates 1,000 Models repeating explicitly, each containing repeating elements. One BOM Option Class in each Model contains Options that repeat across the 1,000 BOM Option Classes, while another BOM Option Class contains Options unique to each of the 1,000 BOM Model.

Duplicate structure or common elements that are explicitly repeated in hundreds or thousands of Models do not scale well as your business grows. Many explicit Models with repetitive structure require repetitive maintenance.

It may not be possible or necessary to change your explicit Models to take advantage of abstractions and references, for any of the following reasons:

- You wish to preserve existing BOM Model and routing definitions
- You need to source each explicit Model to a different organization, supplier, or flow line.
- You can adjust your hardware and memory to accommodate many or large explicit Models
- The number of explicit Models is not large enough to significantly affect maintenance, scalability, and performance
- The explicit Models are not large enough to significantly affect maintenance, scalability, and performance

## Permutation

The term permutation refers to a poor modeling practice that adversely affects runtime performance and significantly adds to how much effort is required to maintain a configuration model. This practice can be avoided using the abstraction technique described later in this section.

For example, an end user can select from the following characteristics when configuring a hypothetical product:

- Product Type: (4 options)
- Speed (20 options)
- Service Level (3 options)

Based on these options, there are 240 permutations available to the end user (4 x 20 x 3 = 240). If your solution to this configuration problem is to construct a single Model with Standard Items for each permutation, then the configuration will contain every possible orderable item, although the user will never pick more than one. Such a configuration model will scale very poorly when new items are added in the future. For example, adding a single item (such as a new service level) increases the number of permutations from 240 to 320.

Constructing a Model as an abstraction with 3 Option Classes (one each for Type, Speed, and Service Level) and 27 Standard Items is preferable and will scale much better than the previous example (that is, defining 240 Standard Items). However, downstream applications for manufacturing or provisioning the product or service will require different setups for such a modeling approach. This is one of the trade-offs to consider when evaluating modeling styles.

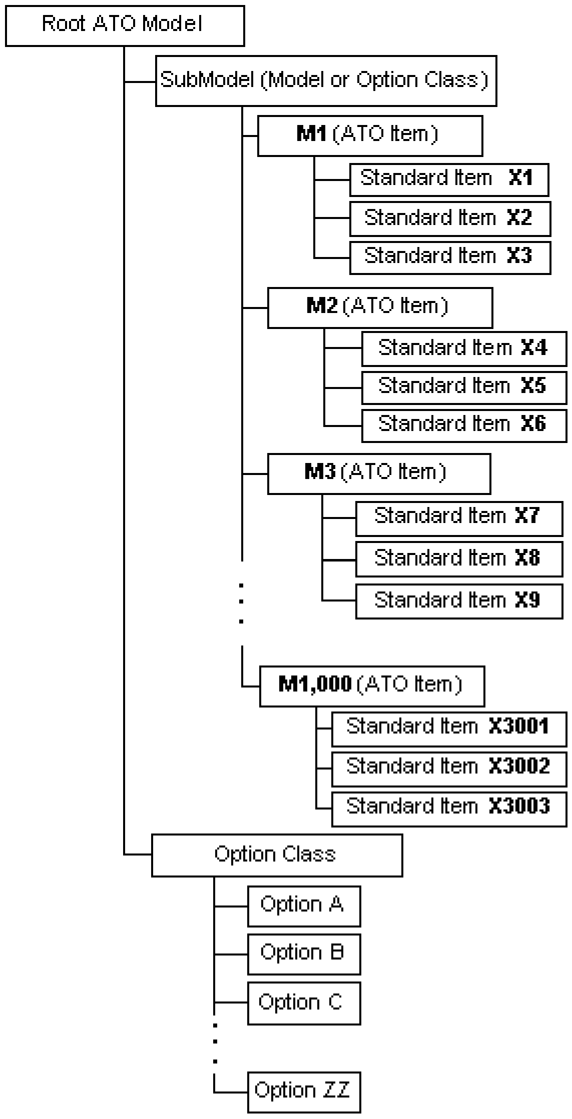
Another approach is to create a single Model with three attributes that collect information about the Type, Speed, and Service Level that the end user requires. Using configuration attributes is also preferable to the first example, but adds difficulty to the implementation and increases the reliance of downstream applications to process the information that is passed from Oracle Configurator. Configuration attributes are explained in *Oracle Configurator Methodologies*.

## Abstractions

An abstraction is a generic part that expresses all the essential characteristics of a specific part. For example, the related models shown in Explicit Model Structure, page 4-3 can be redefined as one top-level root Model containing a BOM Option Class or BOM Model submodel whose contents is M1 through M1,000 redefined as ATO Items. Abstraction of Related Products, page 4-5 shows M1 through M1,000 each containing their unique elements as BOM Standard Items. The repeating elements occur only once as a child BOM Option Class under the top-level root Model.

The following graphic shows a top-level root Model abstraction of the 1,000 Models presented in Explicit Model Structure, page 4-3. The repetitive BOM Option Class with the large number of the same Options occurs only once, and another BOM Option Class containing a few unique Options.

*Abstraction of Related Products*



The redesign with abstractions shown in Abstraction of Related Products, page 4-5 is easier to maintain than the Models in Explicit Model Structure, page 4-3. The number of nodes (per the graphic) has been reduced from 40 to 23, a reduction of 42.5%. See Downstream Consequences in Other Oracle Applications, page 4-6 for a discussion of the possible trade-off when choosing this kind of redesign.

Together with optional instantiation and referencing, abstractions perform and scale better than structure containing explicit, related product definitions. For details about referencing and optional instantiation, see Optional and Multiple Instantiation, page 4-9 and Explicit Model Structure Versus References, page 4-7. Additionally, turning non-orderable items into alternatives such as Features could further optimize the structure. For details about alternatives to items, see Items Versus Alternatives to Items, page 4-14.

## Downstream Consequences in Other Oracle Applications

A separate, explicit ATO Model is sourced from only a single organization or supplier. Multiple sources and option-dependent sourcing are not supported. In the Abstraction of Related Products, page 4-5 graphic, all the configured items **M1** to **M1000** are sourced in the same organization. If you need configurations of these Models to come from different organizations, you need to use an explicit structure such as shown in Explicit Model Structure, page 4-3.

In Abstraction of Related Products, page 4-5, all the configured items **M1** to **M1000** are built on the same flow line defined in Flow Manufacturing. If you need configurations of these Models to be built on different flow lines, you need to use an explicit structure such as shown in Explicit Model Structure, page 4-3.

Models **M1** to **M1000** are unique because they contain components (such as Options **X1**, **X2**, and so on) available only to one of the Models (**M1**). The abstraction of Models **M1** to **M1000** as ATO items in Abstraction of Related Products, page 4-5 must include those components as BOM Standard Items under the Option Classes M1 to M1000. For a configuration to contain selections from Options A to ZZ, as well as one of the ATO items, change the BOM Models so Options **X1**, **X2**, and so on do not appear under an Option Class. In configurator, you would only see options **M1** to **M1000** and A to ZZ. All mandatory components (X1, X2, X3 and so on) just appear on the BOM under **M1**.

Each configuration of the explicit Models **M1** to **M1000** results in one bill of material and routing definition to build the entire configuration. After moving the abstract top-level root Model, each configuration receives its own bill of material and routing, and each of the ATO item options (M1-M1000) maintains its own bill of material and routing. This can be avoided if M1 to M100 are defined as phantom items. However, phantom items do not participate in Available To Promise (ATP) and Advanced Planning.

## Related Best Practices and Relevant Case Studies

Top-level root Models typically access abstract structure through referencing. See Explicit Model Structure Versus References, page 4-7.

Referencing and abstractions alone do not address performance and memory issues associated with instantiating many components at runtime. See Optional and Multiple Instantiation, page 4-9 and Grouped Versus Ungrouped Items, page 4-18. See also Items Versus Alternatives to Items, page 4-14.

The case study described in Many Large BOM Models, page 5-1 illustrates the use of redesigning explicit structure as abstractions.

## Explicit Model Structure Versus References

It is good modeling practice to use referencing instead of explicit structure if your configuration models contain:

- Duplicate or repeating product information
- Common elements or product definition

### Explicit Structure

Explicit Model Structure, page 4-3 shows examples of many related products defined explicitly in Models.

### Model References

Models with References are easier to maintain and require less memory to load than structure containing explicit common product definitions.

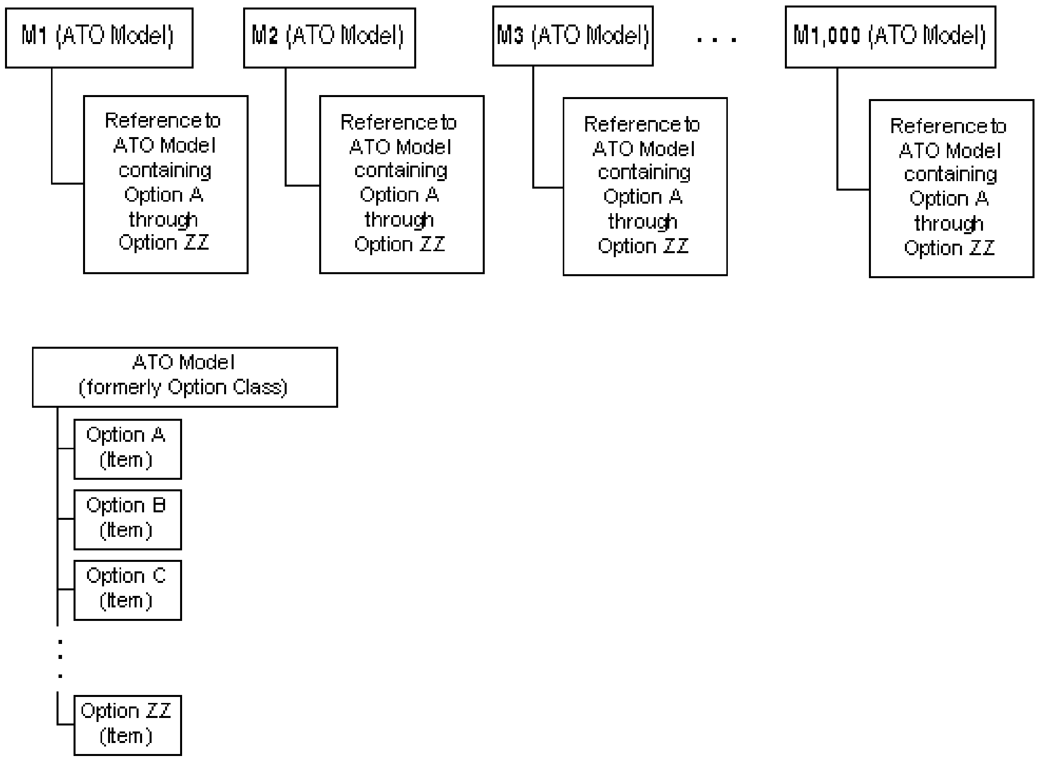
When importing BOMs, all submodels in the top-level root BOM Model are imported as references. See the *Oracle Configurator Implementation Guide* for information about BOM structure after it has been imported into the CZ schema.

#### Referencing BOM Option Classes

BOM Option Classes cannot be shared by reference. Redesigning the BOM Option Class in M1 through M1,000 in Explicit Model Structure, page 4-3 so that it can be referenced requires turning it into a phantom ATO model containing Option A through Option ZZ. Another Possible Abstraction of Related Products, page 4-8 shows the BOM Option Class redefined as a phantom ATO model so that it can be referenced in M1 through M1,000.

The following graphic shows the BOM Option Class with the large number of the same Options in the previous figure now changed into an ATO Model. The 1,000 explicit Models each contain a reference to the ATO Model.

***Another Possible Abstraction of Related Products***


M1 (ATO Model)
Reference to ATO Model containing Option A through Option ZZ
M2 (ATO Model)
Reference to ATO Model containing Option A through Option ZZ
M3 (ATO Model)
Reference to ATO Model containing Option A through Option ZZ
. . .
M1,000 (ATO Model)
Reference to ATO Model containing Option A through Option ZZ
ATO Model (formerly Option Class)
Option A (Item)
Option B (Item)
Option C (Item)
Option ZZ (Item)

Changing BOM Option Classes into BOM Models has the following consequences within Oracle Configurator:

- BOM Models cannot express a mutually exclusive relationship
- BOM Model Items cannot participate in Compatibility Rules because Compatibility Rules enumerate *all* of the allowed combinations of Options from the participant's Features. If a selection is made from each participant Feature and the selections do not correspond to one of the rows in the Explicit Compatibility table or do not satisfy the Property-based criteria, then there is a contradiction.
- BOM Models with a logic state of true do not allow the last available Option to be set to true

> **Note:** If you have redesigned BOM Option Classes as phantom ATO Models, you cannot specify a value greater than 1 to the **Instances Maximum.**

## Non-Imported Model References

Referencing is a technique used to optimize development and maintenance of a configuration model in which the same submodel appears multiple times in the structure. Replacing each explicit occurrence of the submodel with References to a separate Model that represents the submodel can also improve runtime performance. Additionally, it is a good idea to place a Component with repetitive structure within a non-imported Model and then create References to that Model. See the *Oracle Configurator Developer User's Guide* for information about using Model referencing in configuration models.

### Downstream Consequences in Other Oracle Applications

A BOM submodel behaves like a BOM Option Class in most of the Oracle Applications if you leave the **Supply Type** of the BOM submodel set to phantom on its parent BOM Model.

In Flow Manufacturing, some conveniences in defining BOM Option Classes are not available when defining BOM Models. For example, you cannot create a common routing for all BOM Models as you can for all BOM Option Classes under a parent BOM Model.

### Related Best Practices and Relevant Case Studies

Referencing is used when creating abstractions. See Explicit Model Structure Versus Abstractions, page 4-2.

See Many Large BOM Models, page 5-1.

## Optional and Multiple Instantiation

Optional instantiation is an implementation approach used to optimize performance of a configuration model by creating a component instance only if and when it is needed. This prevents Oracle Configurator from loading model elements at initialization that may not be selected or needed in the configuration session.

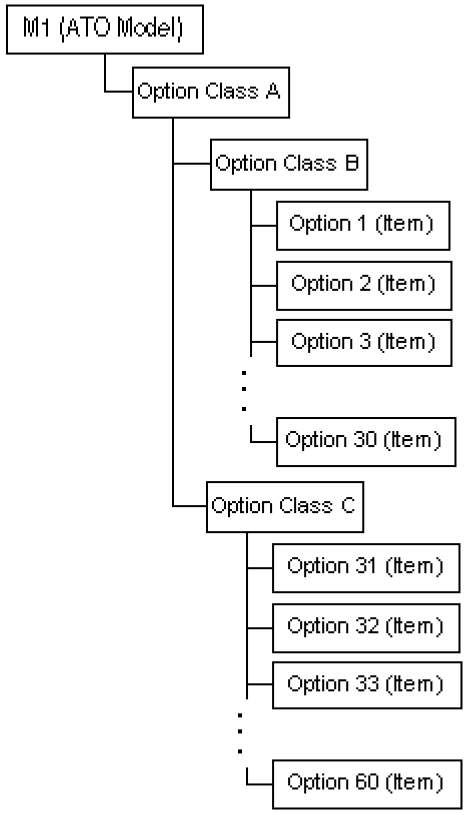
Multiple instantiation is an implementation approach used to optimize usability by allowing end users to create and individually configure multiple occurrences of a Model or Component at runtime, as needed. For more information about multiple instantiation, see the *Oracle Configurator Developer User's Guide*.

### Optional Instantiation of BOM Option Classes

It is good modeling practice to convert a large hierarchy of BOM Option Classes to a hierarchy of optionally instantiable BOM Models. For guidance in converting BOM Option Classes into BOM Models, see Referencing BOM Option Classes, page 4-7. The selective instantiation improves runtime memory usage and reduces caching of redundant data. For example, if M1 through M1,000 in Explicit Model Structure, page 4-3 contained more structure, as shown in Explicit Nested Hierarchy, page 4-10, the resulting large number of Model elements that need to be loaded and instantiated could affect performance and usability adversely.

The following graphic shows an ATO Model containing a BOM Option Class with several child BOM Option Classes, each with unique Options.

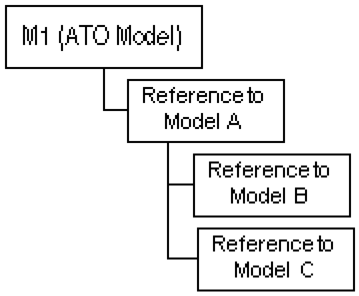
***Explicit Nested Hierarchy***



Rather than load all the explicit, nested BOM Option Classes of Explicit Nested Hierarchy, page 4-10, define them once as BOM Models and include them by reference in M1 through M1,000, as shown in Referenced Nested Hierarchy, page 4-11. Be aware of the consequences of changing BOM Option Classes into BOM Models, as described in Referencing BOM Option Classes, page 4-7.

The graphic Referenced Nested Hierarchy, page 4-11 shows the ATO Model from the previous figure now with references to a submodel containing references to child Models that in the previous figure were the nested BOM Option Classes.

*Referenced Nested Hierarchy*



To save additional memory and improve runtime performance, make the References in Referenced Nested Hierarchy, page 4-11optionally instantiable.

Not loading instances at start up and instead implementing optional instantiation can result in significant performance improvement. However, requiring end users to instantiate many components interactively one at a time may affect usability. Instead, implement guided buying or selling to find out how many instances the end user needs and implement a Configurator Extension that instantiates that number of components all at the same time.

With optional instantiation, as with multiple instantiation, instances cannot be created using rules. Either the minimum is set, the end user clicks an **Add** button, or a Configurator Extension adds instances based on inputs. Using optional instantiation in a generated UI of the BOM structure results in **Add** buttons that end users must understand how to use to complete a configuration. If your end users are not product experts, consider using guided buying or selling questions. See Guided Buying or Selling, page 4-12 for details.

## Setting Node Values After Adding Instances

Design your UI flow and Configurator Extension so that instances are added when there are as few settings as possible. Adding an instance causes all previous requests to be retracted, after which the requests are added and reasserted. This occurs with each component addition. It is good modeling practice to delay setting the states or values of nodes until after all instances are added either at startup (with a Configurator Extension bound to the `postConfigNew` event) or early in the end-user flow of a configuration session. See also The postConfigNew Configurator Extension, page 4-22.

Adding an instance is particularly expensive when there are default values set by configuration rules. Retracting default assertions is time-consuming and iterative. See Defaults Rules Versus Alternatives to Default Selections, page 4-20 for details. The initial values set in Configurator Developer for Boolean Features should also be regarded as default values.

### Downstream Consequences in Other Oracle Applications

See Downstream Consequences in Other Oracle Applications, page 4-9 for the effects on Oracle Applications of changing a BOM Option Class into a BOM Model.

Multiple instantiation of submodels is only available if the parent ATO BOM Model's **Supply Type** is set to non-phantom. This means that each configured instance of the Model receives its own configuration Item number, bill of material, and routing, and is built or bought individually.

### Large Numbers of Instances

If your Model has large numbers of instances, check for similarity among instances. If many sibling components are nearly identical, meaning they only differ in the value of their attributes and are not distinguished by connectivity, consider representing the entire set of instances as a single instance. You can store the set of attribute values offline or in a custom table.

### Related Best Practices and Relevant Case Studies

See Explicit Model Structure Versus References, page 4-7.

See Large Option Features and Option Classes, page 4-18.

See Many Large BOM Models, page 5-1.

## Guided Buying or Selling

Guided buying or selling questions are intended to guide end users to specific selections and valid configurations.

The size or complexity of your Model may cause usability or performance problems at runtime. Large models, even if not complex, may require large amounts of memory to load, or require end users to make many selections or page through many options. Complex models, even if not large, may require long end-user think time and complicated navigation. Under these circumstances, you can simplify or streamline the end-user's experience by defining guided buying or selling structure.

Guided buying or selling questions gather user inputs to accomplish any of the following scenarios:

- End users reach a valid configuration solely by answering guided buying or selling questions
- End users complete a configuration by making additional selections after their answers to guided buying or selling questions make an initial set of selections
- End users only see a narrowed set of selections after answering guided buying or

selling questions

You implement these behaviors by adding rules that tie the end-user inputs to the selection or exclusion of options.

Generating a default UI for a large imported BOM may result in an unreasonably large and poorly performing runtime session. You can use guided buying or selling UI structure to control which elements of the imported BOM you want exposed and visible in the UI. This requires defining rules that associate the guided buying or selling UI structure to the BOM Model items and disabling the UI visibility attribute of the BOM nodes before generating the UI.

## Manufacturing vs. Sales View of a Model

In many cases, the Sales and the Manufacturing organizations of a company have very different views of the product definition. This can driven by several factors, including:

- The need to display customer-friendly item names rather than part numbers
- The need to simplify the product structure for customers
- The need to sell functions instead of options and features.

  For example, electrical motors are sold based on their output characteristics, not based on their components.
- Marketing requirements

  For example, the name of an option must remain the same from a sales and marketing perspective, whereas the manufacturing option may change
- Sales packages

  For example, different groups of manufacturing options may be included in different sales packages.

For companies that were previously using two separate configuration solutions for their Sales and Manufacturing operations, the result may be two separate configuration models and the need to "translate" Sales configurations into Manufacturing configurations when an order is sent to production.

With Oracle Configurator, the same configuration model is shared by the entire Enterprise, and requirements from the different divisions that are involved in the configuration process need to be considered throughout the configuration model design phase.

As a Configurator model designer, you may want to use your current Manufacturing structure as the basis for your configuration model, and create additional structure to satisfy Sales requirements, adding configuration rules to map the Sales options to Manufacturing rules. This approach can lead to performance issues as it increases the size of the model and the number of rules. Instead, the recommended approach is to review and modify (if necessary) your Manufacturing structure to accommodate Sales requirements, thereby optimizing your configuration model's performance and usability. For example, you can expose Manufacturing options to end users by giving the related item a user-friendly description and then displaying the description (rather than the item name) in the Configurator User Interface.

In most cases, using the current Manufacturing structure is not recommended. Be sure to evaluate whether functionality in Oracle Configure to Order (CTO) or other applications will permit changes to the model design that benefits deployment and minimizes the amount of future maintenance.

## Shallow Versus Nested or Deep Hierarchy

The depth of Model structure is determined by the levels of hierarchy contained under the root node. Shallow structure is acceptable when not displaying BOM nodes in the UI. A shallow structure generally decreases the size of the configuration model and potentially eases downstream processing.

Deeply nested structure takes advantage of the performance gains made when using optional instantiation to load only those parts of the Model structure that are needed. Nesting items into deeper structure can also ease Model maintenance. However, deep nesting may distribute end-user selections across UI pages in a manner that requires end users to flip back and forth as they check previous selections. When this is the case, guided buying or selling questions can significantly ease navigation flow. See Guided Buying or Selling, page 4-12 for additional information about guided buying or selling.

### Related Best Practices

See Explicit Model Structure Versus References, page 4-7 for descriptions of nested structure. Nested or deep hierarchy affects performance when used with optional instantiation. See Optional and Multiple Instantiation, page 4-9.

### Relevant Case Studies

See Many Large BOM Models, page 5-1 and Many BOM Items, page 6-1.

## Items Versus Alternatives to Items

Items either need to appear on the order line, or they need to represent characteristics that are needed as input somewhere in the process. Some item information is only needed as a temporary value during configuration.

You should define Inventory Items only for elements of your structure that need to be ordered or routed downstream from Oracle Configurator. For example, elements of Model structure that must appear on an order line and are picked or assembled for shipping need to be Inventory Items.

Alternatives to Inventory Items are Option Features or configuration attributes. For upstream item information that is needed in Configurator computations or for downstream processing such as calculations or passing along information about an item, you should define an alternative to Items.

For example, if you have raw materials that are ordered by lengths, do not create an item for each length. Instead, define a single item for each raw material with an attribute: Length. Then capture the needed length from the end user by a numeric input or from a List of Options, and associate that input with the attribute.

For another example of changing explicit items into Features, compare Explicit Model Structure, page 4-3 with Abstraction of Related Products, page 4-5.

For more information about configuration attributes, see *Oracle Configurator Methodologies.*

## Values Needed For Configuration Only

If the value is not needed in any way downstream but only for completing the configuration, use non-BOM Features and Options. These values cannot participate in downstream calculations.

For information about leveraging the power of Properties, see *Oracle Configurator Developer User's Guide*.

## Values Needed Downstream

If the value is needed downstream, such as in Fulfillment or Billing Integration, and represents a Configurator computation or inputs from the end user, you can define configuration attributes to pass along those inputs. For performance reasons, it is optimal to define as few attributes as possible. Omit Feature and attribute definitions from your Model structure whose values can be referenced from other Model structure or computed in custom code downstream from other attributes. If a value cannot be referenced or computed using custom code, minimize the set of attributes and Features you define in the Model structure. If an attribute is shared among multiple components, use inheritance to share the Feature in which the attribute is defined.

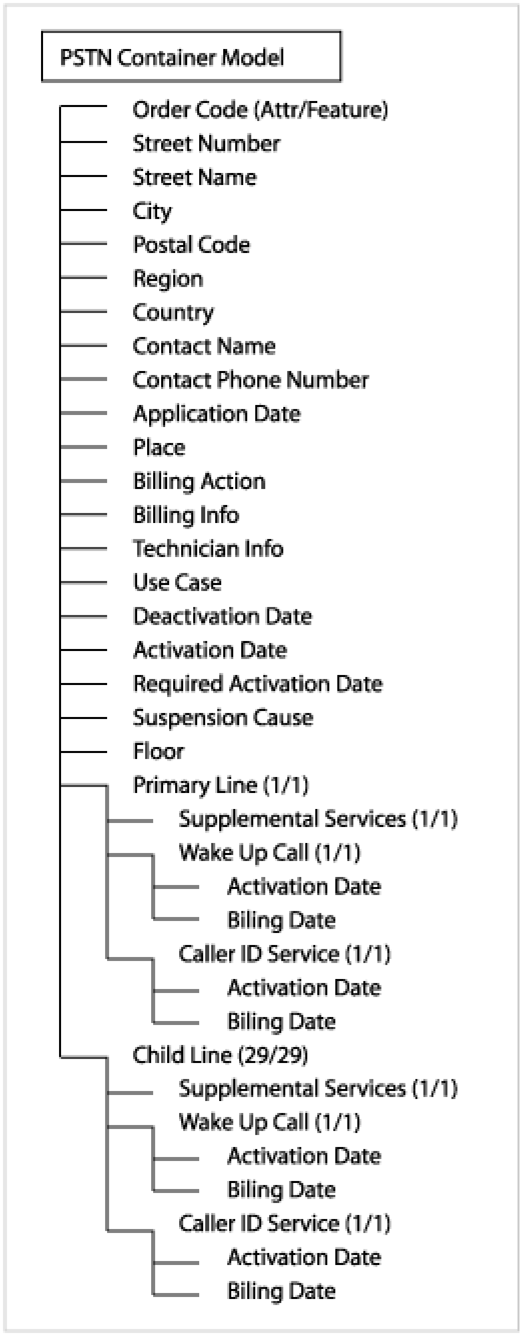
See *Oracle Configurator Methodologies* for details about implementing configuration attributes. Note that configuration attributes are not accessible to downstream applications without customization.

For example, consider a Container Model that defines a configurable Public Switched Telephone network (PSTN). The top-level Model contains two Components: Primary Line (1/1) and Child Line (29/29). Each Component contains Items (such as Application Date and Billing Date) that have been modeled as attributes of Features and are required for downstream processing. All the Features are hidden in the runtime UI. A Configurator Extension rule gets the required attribute values from custom code outside Configurator and copies the values for the model and its submodels. The duplication of data across many models and the process of copying the values causes poor performance when the configuration is saved, restored, or during batch validation.

For details about using the type of attributes referred to in this example, refer to the method for using configuration attributes with Install Base that is described in the *Oracle Telecommunications Service Ordering Process Guide.*

The graphic Model Design with Minimal Set of Features, page 4-17, shows a better Model design that moves all the duplicated common attribute Features to the top-level Container model.

**Model Design with Minimal Set of Features**



The design includes attribute Features for unique attributes at the individual Model level. A Configurator Extension copies the value from the custom application to the top-level Features. Attribute Mode set to ALL propagates the value from the top-level Model to all the submodels.

### Related Best Practices and Relevant Case Studies

See Many BOM Items, page 6-1.

## Large Option Features and Option Classes

If your Model contains large BOM Option Classes or Option Features, evaluate the following best practices for possible use in your project:

- Grouped Versus Ungrouped Items, page 4-18
- Maximum Selections on Large Option Classes or Features, page 4-18
- Alternatives to Option Features With Many Options, page 4-20

If your design cannot eliminate the need for configuring BOM Option Classes with large numbers (for example, hundreds) of Items, be aware that the Generic Configurator UI does not perform faster than a generated Configurator UI. With the generated Configurator UI you can use multiple screens to display BOM Option Classes with large numbers of Items. If you are using guided buying or selling questions, hide all the Items and have the guided selling questions drive the selections. See Guided Buying or Selling, page 4-12 for more information.

### Grouped Versus Ungrouped Items

In cases where there are many options to choose from, the end user's experience and Oracle Configurator performance can be improved by grouping items into BOM Option Classes. Typically, you can improve usability and performance by combining grouping with the use of optional instantiation. This requires changing BOM Option Classes into BOM Models. See Optional Instantiation of BOM Option Classes, page 4-9 for details.

For example, a manufacturer of upholstered furniture lets customers select fabrics from a huge inventory, including solids, stripes, floral prints, and geometric patterns. Rather than have end users flip through screen after screen of swatches in no particular order, the configuration experience can be organized by types of fabrics, and only the group that is selected needs to be loaded into memory. End users select one from a subset of fabrics, thereby reducing the number of displayed options. The inventory that does not need to be displayed to the end user is not instantiated.

Grouping items can also ease Model maintenance.

### Maximum Selections on Large Option Classes or Features

Defining a **Maximum Selections** on a large BOM Option Class or Option Feature can adversely affect performance. If an Option Feature contains an Option that participates in a rule, propagation of that rule triggers a count of the current number of selected Options. If the maximum number of Options specified by the value of **Maximum**

**Selections** is reached, this rule propagates **Logic False** to all unselected Options. If the number of Options is large, propagating Logic False or retracting and reasserting the logic state on all the Options can cause slow performance.

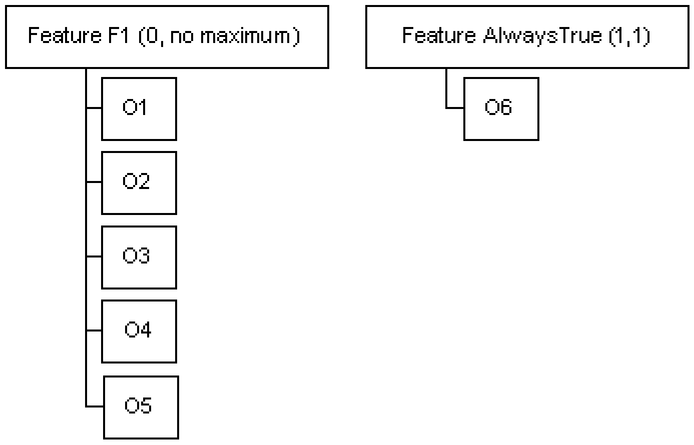
The best way to improve performance is to restructure the Model or BOM Model to contain Option Features or BOM Option Classes with fewer Options or Items.

If your business requirements demand many options and a maximum number that can be selected, and you need the remaining options to be **Logic False** after the maximum is reached, then you must set the **Maximum Selections** and incur the performance cost.

However, since the runtime display icons for both **Logic False** (from maximum reached) and **Unknown** states indicate that an Option is available for selection, your logic requirements may permit you to set the **Maximum Selections** to no value and define the number of allowable selections by using a rule that counts the selected Options. For example, if you want only three Options to be selected from a Feature, impose that maximum by defining a Total, a Numeric Rule, and a Logic Rule with CDL or as a Statement Rule as shown in Maximum Selections Imposed by a Rule, page 4-19.

### Maximum Selections Imposed by a Rule

One Feature, F1, contains five Options, and a second Feature, AlwaysTrue, contains one Option, as shown in the following design:



The rules:

```
Total T1 tracks the number of options selected.
Numeric Rule: EachOf (Options of F1) Contributes 1 to T1
Statement Rule expressing a Logic Rule with the following expression for
Operand 2: AlwaysTrue Requires Total < 4
```

At runtime, when the end user selects O1, O2, and O3, the number of selections is less than 4. When the end user selects O4, the Logic Rule displays a violation message. The violation message can be customized to explain that the maximum number of allowable selections has been exceeded.

If your business requirements demand many options and a maximum number that can be selected, and it does not matter to your implementation whether Options are **Unknown** or **Logic False**, then having them be **Unknown** is better. The Oracle

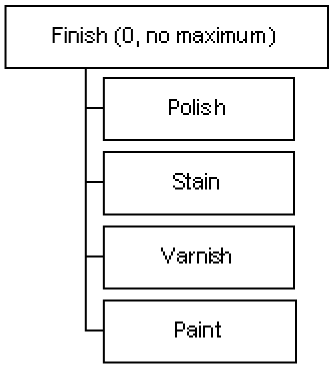
Configurator engine does not push the **Unknown** state to other options that are connected through rules, except if NotTrue is used. Design your configuration model so that valid configurations are allowed even when numerous Options are **Unknown**. That way the CIO and the Oracle Configurator engine process fewer changes on each end-user action. For example, use the suggested rule for counting selected options to minimize the propagated **Logic False** states, instead of the regular **Maximum Selections.**

### Alternatives to Option Features With Many Options

When an Option from a large Option Feature participates in a rule, the propagation of that rule triggers an evaluation of all that Feature's other Options. (This does not apply to BOM Option Classes and BOM Standard Items.) However, if no Options participate in rules or the Option Feature is not constrained by a maximum number of Options that can be selected, redesign the Options as Boolean Features. Create a Component containing the Boolean Features or design all selectable Options as BOM structure.

#### Redesigning Unconstrained Options as Boolean Features

A Feature called Finish contains four independent furniture finishing Options, Polish, Stain, Varnish, and Paint, as shown in the following design:



Since end users can select any Option independent of the other Option selections, it is better to define a Component containing a Boolean Feature for each Option.

### Relevant Case Studies

See Many BOM Items, page 6-1.

## Defaults Rules Versus Alternatives to Default Selections

Defaults Logic Rules reduce the number of required end-user selections by providing the following:

- Initial values when Oracle Configurator is launched

- Filled-in values during the configuration session based on end user selections
- Filled-in values at the end of the configuration session to complete the configuration

Some implementations of Defaults relation can cause performance problems at runtime, especially when propagation of the defaulted items affects many other items. This is because the Oracle Configurator engine retracts default values before applying or reapplying user requests or adding or deleting components.

## Evaluating the Need for Default Selections

When designing a configuration model, add Defaults Logic Rules at the end of the design process and weigh the performance cost of adding them against the perceived benefit to the end user.

The following considerations can help you determine the usefulness of setting initial values:

- Analyze how often the end user changes the default selections that are automatically made by these rules. If you expect that default selections are often changed, then you will gain runtime performance by removing those Defaults relations.
- Analyze the runtime performance when Defaults relations are active, compared to when the Defaults relations are disabled and end users have to make initial selections explicitly.

## Activating Defaults on End User Request

If you intend to apply default values to complete the configuration session rather than set selections during the configuration session, design the Defaults Logic Rules so they are not processed at runtime until the end user requests them. For example, create a Boolean Feature at the root of the Model for this purpose. This Boolean Feature could be called ApplyDefaults.

1. Create a Statement Rule with Operand 1 of the Defaults relation defining a condition.
2. In every Defaults relation, replace

   **Example**
   ```
   X Defaults Y
   ```
   where `X` is the condition, and `Y` is the result, with

   **Example**
   ```
   AllTrue (X, ApplyDefaults) Defaults Y
   ```
   Note that it is enough to define this rule as `AllTrue (Apply Defaults)` if X is

always true.

3. Write a Configurator Extension bound to the `preConfigSave`event that sets the ApplyDefaults Boolean Feature to True. This causes applicable Defaults to be applied so their consequences appear in the output.

4. Write a Configurator Extension bound to the `postConfigRestore` event that unsets ApplyDefaults. This allows the configuration session to proceed in the absence of defaults. Since this Configurator Extension could cause unnecessary work during batch validation, consider using Step 5.

5. Optionally, make the ApplyDefaults Boolean Feature visible in the User Interface so the end user can turn it on and see the consequences of the Defaults. Since this is expensive in terms of performance, the end user might elect to use it sparingly.

If you do not make the ApplyDefaults Boolean Feature visible, line items and prices may appear in the order that the end user was not aware of during the configuration session.

## Boolean Features With Initial Values

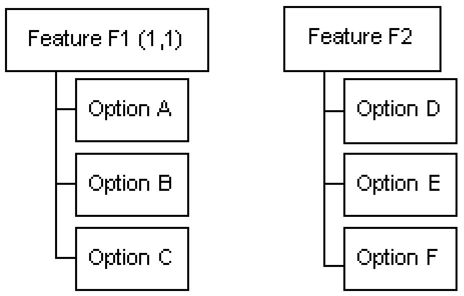
Boolean Features with an initial value (True or False) cause slow performance because they behave like Defaults Logic Rules. Leaving the initial value Unknown improves performance.

## The postConfigNew Configurator Extension

A Configurator Extension bound to the `postConfigNew` event simulates the behavior of setting initial default values, but avoids the cost of repeatedly retracting and reapplying default values after the end user makes selections involving Defaults relation participants. However, there are consequences to using the Configurator Extension that you should consider. Defaults Rule Behavior at Runtime, page 4-22 and Configurator Extension That Simulates Defaults Rule, page 4-23 show the differences in behavior during the runtime selection process between Defaults relations and Configurator Extensions that simulate Defaults relations.

### Defaults Rule Behavior at Runtime

Two Features contain different Options, as shown in the following design: Feature F1 contains Options A, B, and C. Feature F2 contains Options D, E, and F.



The rules:

```
F1 has a Maximum Selections set to 1
F1 Defaults A
E Requires B
```

At startup, A is selected. When the end user selects E, Option B is selected and A is deselected. No contradiction occurs in this example.

#### Configurator Extension That Simulates Defaults Rule

Using the same two Features shown in Defaults Rule Behavior at Runtime, page 4-22, define the following rules:

```
Configurator Extension selects A
E Requires B
```

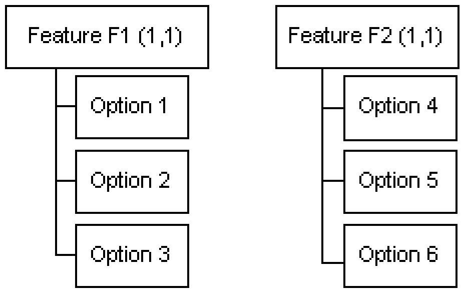
When the configuration session begins, Option A is selected. The end user selects Option E. Oracle Configurator displays a contradiction message asking whether the end user wants to override the contradiction and select Option B. This is because settings made by the Configurator Extension have the status of an end-user request.

### Implies Relation Instead of Defaults Relation

In some rare cases, an Implies relation can be used instead of a Defaults relation if your end users do not need the flexibility to change the default value.

#### Implies Rule Provides Behavior of an Unoverridable Defaults Rule

Two Features contain different Options, as shown in the following design: Feature F1 contains Options 1, 2, and 3. Feature F2 contains Options 4, 5, and 6.



The rules:

```
Option 1 Defaults Option 4
or
Option 1 Implies Option 4
```

At runtime, both rules cause Option 4 to become selected when the end user selects Option 1. The Defaults relation gives end users the flexibility to override the default value and select Option 5 or Option 6 when Option 1 is selected. If you your end users do not need that flexibility, use the Implies relation and only Option 4 can be selected when Option 1 is selected.

### Default Rule Idiosyncrasies

When designing Defaults rules, avoid overlapping rules as this can cause unexpected behavior at runtime. Rules overlap when they both contain the same node (or nodes) as participants.

For example, in Model A, Features Y and Z are siblings (that is, they appear at the same level in the Model structure). You define the following rules:

```
All True (A, B) Defaults Y
All True (A, B, C) Defaults Z
```

When A, B and C are all true at runtime, neither Y nor Z will be selected by default consistently. In other words, sometimes Y will be selected by default and sometimes Z will be selected by default. To avoid this kind of unintended behavior, be very explicit when defining Defaults rules.

To implement the rules in this example properly, define the first rule as follows:

```
All True (A, B, Not (C)) Defaults Y
```

## Repetitive Rule Patterns and Redundancy

Repetitive patterns or redundancy means that several rules include the same subexpressions or have the same result. This could cause performance issues.

### Repetitive Patterns and Common Subexpressions

The Oracle Configurator engine separately evaluates and propagates each instance of the subexpressions of every rule, even if there are commonly used patterns of operators and operands in those subexpressions. Consequently, a large number of rules with common subexpressions impairs performance by triggering redundant calculations of the subexpressions. (A subexpression provides a calculation result that is subsequently used as a term in a parent expression.)

Subexpressions are defined within a pair of parentheses, and every pair of parentheses is treated as a subexpression using the conventional order of operations.

Rules With Common Subexpressions, page 4-25 shows two rules containing a common calculation contributing to a Resource.

**Rules With Common Subexpressions**

[(A + B) * C] contributes to R1

[(A + B) * C] + D contributes to R2

Create an intermediate node (I1) to contain the value of the commonly used subexpression and create an intermediate rule that contains the subexpression [(A + B) * C] and the intermediate node:

[(A + B) * C] contributes to I1

The original rules can then be rewritten by replacing the common subexpression with the new node that expresses the intermediate rule.

I1 contributes to R1

I1 + D contributes to R2

Although this technique creates an additional rule (the definition of the common subexpression), the effect of replacing repeating patterns in rules with nodes that represent those patterns is that Oracle Configurator only computes the subexpression once, thereby reducing the calculation required for each rule containing the common subexpression.

> **Note:** When you use the technique of replacing common subexpressions with intermediate nodes, you must customize the violation message of the intermediate rule to explain the contradiction in terms of the rules that contain the intermediate rule. In other words, the default violation message for I1 in Rules With Common Subexpressions, page 4-25, which is displayed to the end user when Oracle Configurator encounters a problem with R1 or R2, describes a contradiction in I1 unless you customize the message to explain the error in terms of R1 and R2.

## Redundancy

Redundancy occurs when you unintentionally create multiple rules that perform the same function in a configuration model. Executing redundant rules at runtime is inefficient and contributes to poor performance. Redundant rules are most often Logic Rules, and when they are of this type they are more difficult to detect when unit testing.

Redundant Rules: Simple Example, page 4-25 shows two redundant rules.

**Redundant Rules: Simple Example**

- Rule 1: A Requires B
- Rule 2: B Requires A

Note, however, that the following rules are *not* redundant (since the Implies relation is not bidirectional):

- `Rule 1: A Implies B`
- `Rule 2: B Implies A`

The following example shows two more rules that perform the same purpose.

**Redundant Subexpression**

- `Rule 1: (A,B) Requires (X,Y)`
- `Rule 2: AllTrue (A,B,C) Requires AllTrue (X,Y,Z)`

In Redundant Subexpression, page 4-26, Rule 2 overlaps Rule 1 because it contains the same options (A, B, X, and Y) and performs the same purpose as Rule 1 (they both ensure that options X and Y are selected when A and B are selected).

It is also important to understand what kinds of runtime behavior Oracle Confgurator inherently provides to be sure that you do not create rules that accomplish the same tasks. For example, Rules that Perform Unnecessary Actions, page 4-26 shows a rule that performs an action that Oracle Configurator does implicitly.

**Rules that Perform Unnecessary Actions**

Model A contains Feature X, and Feature X has three Options: A, B, and C. Model A also has the following rule:

```
AnyTrue('Option A', 'Option B', 'Option C') Implies AnyTrue('Option
Feature X')
```

When an end user selects any of a Feature X's Options, Oracle Configurator automatically selects Feature X. It is therefore not necessary to define a rule that does the same thing.

## Circular Propagation

Circular rules may involve both Logic and Numeric Rules. In some cases this may result in Oracle Configurator not being able to settle on a final result. Circular Rules (Logic and Numeric), page 4-26 shows four rules that are circular.

**Circular Rules (Logic and Numeric)**

A Contributes A.count() to B

B Contributes B.count() to C

C Requires D

D Contributes D.count() to A

Examine all possible actions or inputs and their results to show where the propagation path does not settle on a result but cycles through the rule again and again. In Circular Rules (Logic and Numeric), page 4-26, the following sequence of events occur when you set A to 3:

**Example**

```
Set A =3
B is set to 3, and true
C is set to 3, and true
D is set to 1, and true
A is set to 4, and true
B is set to 4, and true
C is set to 4, and true
```

The circularity stops at this point, because D is already set to 1, and true.

Circular Numeric Rules may cause numeric cycles, as shown in Numeric Cycles, page 4-27.

**Numeric Cycles**

A Contributes A.count() to B

B Contributes B.count() to C

C Contributes C.count() to A

In Numeric Cycles, page 4-27, the following sequence of events occur when you set A to 3:

**Example**

```
Set A =3
B is set to 3, and True
C is set to 3, and True
A is set to 6, and True
B is set to 6, and True
C is set to 6, and True
A is set to 9, and True
```

Examining all possible actions or inputs and their results suggests that these rules would continue to propagate without end. However, Oracle Configurator catches the error in Numeric Cycles, page 4-27 as a runtime numeric cycle failure.

The circular rules may not be obvious. Imported BOM Models automatically enforce Bill of Material quantity cascade, which can be thought of as Numeric Rules. Attempting to contribute the quantity of a Standard Item to it's parent Option Class or any other ancestor node in the model hierarchy will result in a circular rule.

Optimize the configuration problem to avoid creating rules that create cycles.

One way to debug redundancy and cycles is to turn rules on or off by explicitly disabling them at the rule or Folder level. For a review of logic states and other information related to rules, see the *Oracle Configurator Developer User's Guide*. See the *Oracle Configurator Performance Guide* for a discussion about the effect of the quantity and complexity of rules on runtime performance.

## Number and Complexity of Rules

Complex or large numbers of rules within a configuration model could cause slow performance. Whenever possible, use effectivity dates or Usages to turn off rules that are not necessary in certain contexts of the calling application. That lets the Oracle Configurator engine ignore those rules and propagate among the enabled rules more efficiently. When debugging configuration models, disable rules explicitly at the rule or rule folder level to let the Oracle Configurator engine ignore them. For information on Property-based Compatibility rules, see Minimizing the Size and Complexity of Property-based Compatibility Rules, page 4-33.

Large numbers of rules with commonly used subexpressions also cause slow performance. When the Oracle Configurator engine propagates such rules, even if there are commonly used patterns of operators and operands, the engine needs to evaluate and propagate each instance of the common subexpression separately.

Consider Rules With Common Subexpressions, page 4-28 showing two rules with a common calculation contributing to a Resource.

**Rules With Common Subexpressions**

[(A + B) * C] contributes to R1

[(A + B) * C] + D contributes to R2

Create an intermediate node (I1) for the commonly used subexpression and create an intermediate rule that contains the subexpression [(A+B)*C] and the intermediate node.

[(A+B*C] contributes to I1

The original rules can then be rewritten by replacing the common subexpression with the new node that expresses the intermediary rule.

I1 contributes to R1

I1+D contributes to R2

Although this technique creates an additional rule, the net effect of replacing repeating patterns in rules with nodes that represent those patterns is that the Oracle Configurator only computes the subexpression once, thereby reducing the amount of calculation required for each rule that contains the common subexpression.

> **Warning:** When you use the technique of replacing common subexpressions with intermediate nodes, you must customize the violation message of the intermediary rule to explain the contradiction in terms of the rules that contain the intermediary rule. In other words, the default violation message for I1 in Rules With Common Subexpressions, page 4-28, which is displayed to the end user when Oracle Configurator encounters a problem with R1 or R2, describes a contradiction in I1 unless you customize the message to explain the error in terms of R1 and R2.

## NotTrue Logical Function Imposes Order and Causes Locking

The logical function NotTrue is used in CDL and Statement Rules.

## Order Dependency Caused By NotTrue

Using NotTrue in a Statement Rule can impose an order for rule propagation that makes the configuration model harder to design and use. Rule order diminishes the flexibility with which end users make selections. For example, if the intention is to have one Option, Y, not be selected until another Option, X, is selected, then using NotTrue achieves this result, but alternative rule definitions have a similar effect without causing order dependency.

If the rule is NotTrue(X) Excludes Y, the following occurs at runtime:

1. Initially, X is unknown and Y is not available for selection (false).
2. When X is selected (true), Y becomes unknown, or available for selection. If Y is selected, X remains true.
3. If X is deselected (false), Y becomes false.

If you want Y to be unavailable for selection until X is selected, you need to use NotTrue(X) and incur the cost of imposed order in the rule propagation. If you only need to make sure that Y is never selected when X is not selected, you can express this with less restrictive alternative rules

For example, Not(X) Excludes Y preserves the original intent of ensuring that Y cannot be true without X being true, but X can be true without Y being true. If the rule is Not(X) Excludes Y, the following occurs at runtime:

1. Initially, X is unknown and Y is unknown, or available for selection.
2. When X is selected (true), Y remains unknown. If Y is selected (true), X becomes selected (true).
3. If X is deselected (false), Y becomes deselected (false).

Both these rules use a double negative (which may be difficult to understand) and a subexpression (which compromises efficiency).

An optimal alternative rule is Y Implies X, which imposes no order and avoids the double negative and subexpression of the previous alternative. If the rule is Y Implies X, the following occurs at runtime:

1. Initially both X and Y are unknown and available for selection.
2. When Y is selected (true), X becomes selected (true).
3. If X is deselected (false), Y becomes deselected (false).

Although this rule preserves the original intent of ensuring that Y cannot be in a sales order without X, it does not allow X to be in the sales order without Y.

Note that in release 11.5.10 and later, a simpler and more efficient approach is to define a display condition to prevent Y from appearing in the UI until X is selected. Display conditions are explained in the *Oracle Configurator Developer User's Guide*.

### Locked States Caused By NotTrue

Using NotTrue can result in a locked state for the initial values of some items. In the following example, Model X contains the following structure:

Option Feature A, containing some Options

Option Feature B, containing some Options

Your project requires that Feature B should be false and not displayed to the end user until an Option in Feature A is selected.

When an Option in Feature A is selected, Feature B should be true and displayed to the end user. End users must select an Option from Feature B to satisfy the configuration.

The following rules fulfill these requirements:

**Example**

```
Rule 1: NotTrue(A) Excludes B
Rule 2: A Implies B
```

Initially, as a consequence of Rule 1, A is unknown and B is false. As a consequence of Rule 2, A is made false when B is false. Rule 1 and Rule 2 together result in both A and B being false, which is a locked state. Changing the state of either A or B results in a contradiction.

To accomplish the intent of Rule 1 without locking, create a display condition in Oracle Configurator Developer that hides Feature B until an Option in Feature A is selected.

Locking also results from embedding a NotTrue expression within a series of rules involving the same operands. Consider the following rule:

**Example**

```
NotTrue(A) Implies B
```

If A is an Option of a Feature whose other Options participate in other rules that side effect A and B (such as through a maximum number of selections setting), then A being NotTrue locks the series of rules from completing. To avoid this locking, do not use the participants of a NotTrue expression in any other rules.

For more information about logic states, display conditions, and other information related to rules, see the *Oracle Configurator Developer User's Guide*.

## Compatibility Rules

When you want to express compatibility, use Compatibility Rules instead of Logic Rules. The Oracle Configurator engine performs calculations over a large number of Options faster with Compatibility Rules than with any other rule type.

However, consider the following when designing compatibilities or incompatibilities among items in your Model:

- Expressing Compatibility Using Properties, page 4-31
- Minimizing Participants in a Compatibility , page 4-32
- Using the Excludes Relation to Express Incompatibilities , page 4-33

These guidelines are explained in the following sections.

## Expressing Compatibility Using Properties

Explicit Compatibility Rules defined with a large number of participants are difficult to maintain and can cause performance problems at runtime. It may be better to express the compatibility among options by using a Property-based Compatibility Rule.

For example, power supply voltage for a personal computer is determined by the voltage used in the country where the computer will be used. The United States is compatible with a 110 volt power supply, while France and India are compatible with a 220 volt power supply. You can create a Property-based Compatibility Rule to enforce this relationship at runtime.

The Model for this example includes a Feature named "Country" and a BOM Option Class named "Power Supply Type." Preliminary steps in creating this rule are to define a Property named "Voltage Needed" for the Feature and to define a Property named "Voltage Supplied" for the BOM Option Class.

The table below lists the Options of the Feature called "Country," as well as each Option's Property and Property Values.

| Option Name | Property | Value |
|---|---|---|
| USA | Voltage Value | 110 |
| France | Voltage Value | 220 |
| India | Voltage Value | 220 |

The following table lists the options within the Power Supply Type BOM Option Class, as well as each option's Property and Property Value.

| Option Name | Property | Value |
|---|---|---|
| 110V | Voltage Value | 110 |

| Option Name | Property | Value |
|---|---|---|
| 220V | Voltage Value | 220 |

The final step is to create the following Property-based Compatibility Rule that links the Country Feature selected at runtime to the Power Supply Type BOM Option Class voltage specified for the computer:

Operand 1:

- Feature: Country
- Property: Voltage Value

Operator:

- Equals

Operand 2:

- BOM Option Class: Power Supply Type
- Property: Voltage Value

At runtime, when the end user makes a selection for the Country (Feature), the value for the Voltage Value Property is determined. The Property-based Compatibility Rule matches the value of the BOM Option Class Voltage Value to the value of the Feature Property Voltage Value.

In this example, a Property-based Compatibility Rule design performs better than an Explicit Compatibility Rule defining particular power supply types and the voltages or countries with which they are incompatible. Maintenance is also streamlined. For example, when you add a new country, you add only the Voltage Value Property rather than modifying the rule to add the new compatibility. Or if a country changed its line voltage, the change could be reflected in your Model by changing the value of the Voltage Value Property.

If you needed to express compatibility between country and currency, the problem would be more complex and the maintenance advantages of the Property-based Compatibility solution even greater.

## Minimizing Participants in a Compatibility

To express compatibility among Options by using Compatibility Rules, it is better to design several Compatibility Rules with fewer participants than to define one Compatibility Rule with many participants.

## Using the Excludes Relation to Express Incompatibilities

If your Compatibility Rule defines a larger number of compatibilities than incompatibilities, consider defining the incompatibilities by using Excludes relations. For example, a Compatibility table contains compatibility between six Features, each with two Options. This is shown in Compatibility Table, page 4-33.

The following table lists the compatibilities among the two Options (A and B) and six Features. A1 and A2 are compatible with B1 and B2. A3 is compatible with B1 only.

***Compatibility Table***

| A | B |
|---|---|
| A1 | B1 |
| A1 | B2 |
| A2 | B1 |
| A2 | B2 |
| A3 | B1 |

In this Compatibility table, the only incompatibility exists between Options A3 and B2. In this case, it would be better to express the incompatibility between A3 and B2 in an Excludes relation.

## Minimizing the Size and Complexity of Property-based Compatibility Rules

Property-based relationships between large BOM Option Classes can be implemented more efficiently by introducing Property Features that correspond directly to Properties, with options corresponding to the Property values. Relationships between Property values are then expressed on these smaller Features instead of directly on the Option Classes. Property-based Compatibility Rules are then used to relate the Option Classes and the Property Features. This can result in improved runtime performance and shorter times to generate logic and load a configuration model. Property Features may also provide a shortcut to a needs-based Configurator User Interface.

One of the best ways to make a configuration model simple to define and easy to maintain is to base as many of the rules as possible on Property relationships and avoid the explicit use of options unless absolutely necessary. A possible solution is the use of the Constraint Definition Language's (CDL) 'FOR ALL' expression with Property-based WHERE clauses.

However, there is a negative aspect to this approach. The statement of a Property-based Compatibility or the CDL's FOR ALL...WHERE rule can be deceptively simple. If the Features or Option Classes used in the rule have a large number of options, the rule can expand into large and complex logic within the Configurator Engine. This large and complex logic can result in time-consuming generation of logic in Configurator Developer and poor runtime performance. The best way to avoid this situation is to keep in mind the number of explicit relationships implied by the Property-based expression and take steps to minimize it.

**Large Number of Options**

A model has two Option Classes, A and B. Each Option Class has a large number of options, for example 1000. The options of A have a Property X, the options of B have a Property Y, and an option of A is compatible with an option of B when A.X = B.Y. This relationship can be expressed as a Property-based Compatibility Rule.

Now suppose that X and Y each have 20 distinct values, more or less evenly distributed, so on average each option of A is compatible with 50 options of B. Expressing the relationship as an Explicit Compatibility Rule between A and B would result in each option of A appearing in 50 rows of the table, and the table would contain 50,000 rows (1000 x 50). This illustrates the complexity of the rule as compiled by logic generation. Because this is a very large rule, compiling the logic takes a long time and runtime performance is adversely affected.

The logic of this relationship can be simplified by creating "Property Features." A Property Feature should have Options that correspond to each possible value of the Property, and these Options should each contain the Property in question. Typically, a Property Feature represents a Property of the items of a specific Option Class. It is also possible for a single Property Feature to correspond to multiple Option Classes whose options come from the same catalog. A Property Feature is related to its corresponding Option Class via a Property-based Compatibility Rule, with a condition 'PropertyFeature.Property = OptionClass.Property'.

Property Features can be created easily using a Populator.

1. Manually create a Feature.

2. Define a Populator on the Feature: Select Property Values as the Source and Options as the Destination. Select the Item Type of the Items in the corresponding Option Class and a Property of the selected Item Type. The Populator will automatically attach the specified Property to the Options it creates.

After creating the Populator, create the Property-based Compatibility Rule that links the Property Feature to its corresponding Option Class.

Whenever new options are added to the Option Class in Oracle Inventory (possibly including new values for Properties), Property Features can be brought up to date by rerunning the Populator(s).

### How do Property Features Simplify Property-based Relationships between Large Option Classes?

Any direct property-based relationship between Option Classes is equivalent to a Property-based Compatibility Rule between Property Features, together with the structural Property-based Compatibilities. The complexity of a structural Property-based Compatibility Rule is equivalent to that of an Explicit Compatibility Rule with only as many rows as the Option Class has options. Since the Property Features have a small number of options, a property-based relationship between them will generally correspond to a fairly small number of compatibility table rows or explicit relationships. The total complexity of such a set of indirect relationships is usually less than the complexity of the original direct relationship.

In the Large Number of Options, page 4-34 example, Property Features FeatureX and FeatureY are created. Populators create 20 Options each in FeatureX and FeatureY, which correspond to the 20 different values of the Properties. Each Option bears the corresponding Property and value. The structural Property-based Compatibility would specify 'FeatureX.X = A.X' and 'FeatureY.Y = B.Y'. The property-based relationship requiring X and Y to be equal would specify 'FeatureX.X = FeatureY.Y'. The net effect of these three relationships is equivalent to the original relationship 'A.X = B.Y'. But now, instead of 50,000 equivalent explicit compatibility rows, there are only 2020 (1000 + 1000 + 20).

One other possible benefit of Property Features is that Property values often characterize the form, fit, and function of the Items in an Option Class, and may be closely related to the end user's reasons for preferring one Item over another. In an ideal case, the Properties (or a subset of them) directly represent the user's needs. By presenting Property Features in the runtime UI, it may be possible to implement a needs-based Configurator that requires little or no ongoing maintenance aside from defining Items and BOM Models in Oracle Inventory.

## Comparison Rules

Depending on your requirements, it is generally good modeling practice to avoid rules that raise contradictions. Instead, define rules that cause only validation failures or warnings and therefore allow end users to proceed.

When defining Comparison rules, avoid involving the initial value of a Total in the comparison. The initial value can cause locking when retracted during a numeric cycle.

The examples in this section show best practices for building rules that avoid contradictions and problems caused by intermediate values (also known as locking). However, it is important to remember that subsequent changes to any nodes that participate in the rule may alter a rule's behavior at runtime.

### Comparison Rules That Raise Warnings

When defining Comparison Rules, construct your rules so they lead to a resource violation rather than a contradiction. Rules that raise contradictions are much less flexible because they require one or more previous selections to be deselected before the end user can continue. A resource violation, however, only displays a warning indicating that a specific value has been exceeded, does not deselect any options, and allows the end user to acknowledge the warning and proceed with the configuration.

For an example, see Avoiding Unexpected Contradictions from Intermediate Values, page 4-37.

## Using Intermediate Values Effectively With Comparison Rules

Comparison Rules that raise contradictions can also lead to problems caused by an intermediate value. Oracle Configurator triggers each rule in a configuration model sequentially (that is, one after another), rather than propagating all rules simultaneously. As a result, a rule may be violated when it reaches a specific value, even if you defined another rule to prevent the violation from occurring in the first place. However, since the intermediate value is reached before the other rule propagates, a violation occurs. (See Unexpected Contradictions from Intermediate Values, page 4-36). See also Connectors with Connection Rules, page 4-37 for additional reasons to avoid rule order.

Problems with intermediate values can also occur when:

- The initial value of a Total participates in the comparison
- A Statement Rule contains a comparison-based expression

  For example:

  ```
  'OC58102' = 1 Excludes AnyTrue('OC29315')
  ```

  > **Note:** The purpose of this example is to show a Statement Rule that contains a comparison-based expression. Whether such a rule would cause an intermediate value problem depends on other rules defined in your Model, as described in the preceding paragraph.

### Unexpected Contradictions from Intermediate Values

Model A contains the following:

- Boolean Feature: X
- Option Feature: F1 containing one Option1
- Option Feature: F2 (0,2) containing two Options: Option2 and Option3
- Numeric Feature: Z

The rules:

```
Logic: Boolean Feature X Requires AllOf (OptionsOf (Option Feature F2))
Numeric: EachOf (OptionsOf (Option Feature F2)) Contributes to Numeric
Feature Z
Comparison: Numeric Feature Z equal to 1 Excludes Option Feature F1
```

At runtime, the end user selects Boolean Feature X, making the state of X true. This makes F2 true. Propagation of Option2 makes the value of Numeric Feature Z equal 1. The Comparison Rule causes the propagation of Z=1 to try to push the value of Option Feature F1 false. Even though the current value of Numeric Feature Z is an intermediate value, and the propagation of Option2 will result in a value of Z=2, a contradiction occurs. To avoid the contradiction, consume from a Resource to keep Z=1 when Option2 is true.

### Avoiding Unexpected Contradictions from Intermediate Values

This example shows how to avoid contradictions raised by Comparison Rules because of intermediate values.

In addition to the nodes described in the previous example, above, Model A also contains the following:

- Boolean Feature: Temp
- Resource: Res

To avoid a contradiction from an intermediate value, define the following rules (note that you define these *instead of* the rules shown in Unexpected Contradictions from Immediate Values, page 4-36, not in addition to those rules):

```
Logic: Boolean Feature X Requires AllOf (OptionsOf (Option Feature F2))
Numeric: EachOf (OptionsOf (Option Feature F2)) Contributes to Numeric
Feature Z
Comparison: Numeric Feature Z equalto 1 Implies Temp
Numeric: Temp Consumes from Res
```

At runtime, the end user selects Boolean Feature X, making the state of X true. This makes F2 true. Propagation of Option2 makes the value of Numeric Feature Z equal 1. The Comparison Rule causes the propagation of Z=1 to push Boolean Feature Temp true. Temp consumes 1 from Resource Res. Since Oracle Configurator checks validation failures at the end of propagation, Option3 can become true without any validation failures.

## Connectors with Connection Rules

Connection rules that involve Connector nodes in the Model structure operate only when component instances are connected at runtime, and only among the connected instances. The logic generated by connection rules is not initially loaded with the configuration model at runtime. Connection rules are loaded and executed only when the Connectors in the parent instance have been assigned to corresponding instances of the target Models.

When an end user makes a connection, the following happens:

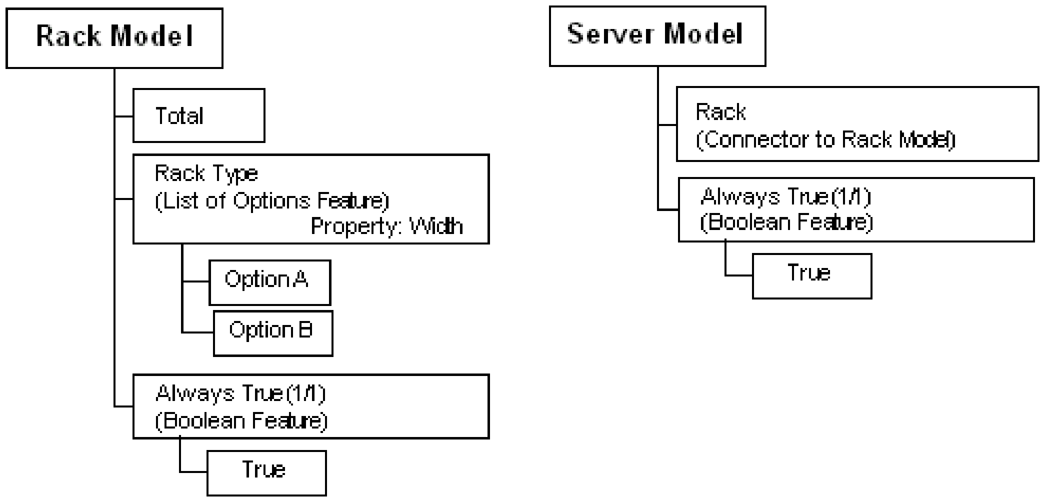
1. All the end-user inputs are retracted.
2. The connection is made.
3. Connection rules are loaded and executed.
4. All the end-user inputs are applied again.

This process leads to changes in values and logic that conflict with the configuration as it was before the connection was made. The sequence in which the rules are loaded and the end-user inputs are reapplied could cause unexpected behavior due to invalid intermediate values. Such situations could cause fatal errors and prevent the end user from proceeding with the configuration.

To avoid inconsistency or errors, ensure that all component instantiations and connections can be performed independent of end-user inputs. Additionally, as with all configuration rules, write rules that are order independent.

Consider the two models in Model with a Connector, page 4-38. The Rack Model contains a Total, a Boolean Feature that is Always True, and an Options Feature named Rack Type with Option A and B. The Rack Type Options Feature has a Property named Width. The Server Model contains a Connector to the Rack Model and a Boolean Feature that is Always True.

***Model with a Connector***


Rack Model
Total
Rack Type (List of Options Feature) Property: Width
Option A
Option B
Always True (1/1) (Boolean Feature)
True
Server Model
Rack (Connector to Rack Model)
Always True (1/1) (Boolean Feature)
True

The following sections use these Models to demonstrate:

- Connection Rules That Depend on End-User Input, page 4-39
- Order Independent Connection Rules , page 4-39
- Restoring Configurations With Connections, page 4-40

## Connection Rules That Depend on End-User Input

The rules in the Rack Model are as follows:

**Example**

```
Rule 1: Total < 0 Excludes Always True
Rule 2: Each Of (Options Of(Rack Type)) * Property.Weight Contributes To
Total
CDL Rule 1: CONSTRAIN (Total < 0) EXCLUDES "Always True"
CDL Rule 2: CONTRIBUTE (EachOf (OptionsOf(Rack Type)) * Property.Weight)
TO Total
```

The intent of the CONSTRAIN rule ensures that the value of Total never drops below 0 (zero).

The rule in the Server Model is the Connection rule:

**Example**

```
Rule 3: Always True * Constant(Value) Consumes From Total
CDL Rule 3: CONTRIBUTE ((Always True * Constant(Value))* -1) TO Total
```

At runtime, the end user selects a Rack Type. Oracle Configurator executes Rule 2, contributing the selected Rack Type's Property Weight to Total. The end user then connects to the Rack Model in the Server Model. Oracle Configurator retracts the selected Rack Type, which resets Total to 0. Oracle Configurator loads the connection rule, Rule 3, and consumes Value from Total (0), causing Rule 1 to be invalid. This results in a non-overridable contradiction and the connection fails.

## Order Independent Connection Rules

You can avoid the dependence on end-user input described in Connection Rules That Depend on End-User Input, page 4-39 by creating a rule rather than an end-user input that makes the contribution. Rewrite Rule 2 in the above case as:

**Example**

```
Rule 1: Total < 0 Excludes Always True
Rule 2: Always True * Constant(Value) Contributes To Total
CDL Rule 1: CONSTRAIN (Total < 0) EXCLUDES "Always True"
CDL Rule 2: CONTRIBUTE "Always True" * Constant(Value) TO Total
```

When Oracle Configurator starts up, a configuration is created. The end user connects the Server Model to a Rack Model. Oracle Configurator retracts all end-user inputs, but does not retract the Value of Total because it has been set by a rule, not an end-user input. Then Oracle Configurator loads the connection rule, Rule 3, and consumes from Total without a contradiction occurring.

**Example**

```
Rule 3: Always True * Constant(Value) Consumes From Total
CDL Rule 3: CONTRIBUTE (("Always True" * Constant(Value))* -1) TO Total
```

This solution avoids order dependence but may still encounter intermittent failures when a configuration is restored. See Restoring Configurations With Connections, page 4-40 for details.

### Restoring Configurations With Connections

Restoring a configuration created using the rules described in Order Independent Connection Rules , page 4-39 intermittently results in failures. When Oracle Configurator restores a configuration, the Oracle Configurator engine activates all the rules and the order of assertions is no longer guaranteed. In some cases, Rule 1 and Rule 3 might be executed before Rule 2, resulting in a fatal error.

You can avoid the fatal error either by populating a Total with an initial value, or using a Resource instead of a Total. Over-consuming a resource does not result in a fatal validation failure, which allows the end user to proceed with the configuration session.

**Example**

```
Rule 1: Resource < 0 Excludes Always True
Rule 2: Always True * Constant(Value) Contributes To Resource
Rule 3: Always True * Constant(Value) Consumes From Resource
CDL Rule 1: CONSTRAIN (Total < 0) EXCLUDES "Always True"
CDL Rule 2: CONTRIBUTE "Always True" * Constant(Value) TO Total
CDL Rule 3: CONTRIBUTE (("Always True" * Constant(Value))* -1) TO
Resource
```

When Oracle Configurator starts up, a configuration is created. The end user connects the Server Model to a Rack Model. Oracle Configurator retracts all end-user inputs, but does not retract the Value of Resource because it has been set by a rule, not an end-user input. Then Oracle Configurator loads the connection rule, Rule 3, and consumes from Resource without a contradiction occurring. The connection is made. Now, when the end user restores this configuration, only a validation failure would occur if Rule 1 and Rule 3 were executed before Rule 2. The validation failure would not prevent the configuration from being restored.

## Optimizing User Interface Performance

The following design considerations can help you improve performance of a runtime User Interface:

- Visibility Settings, page 4-40
- Graphics, page 4-41
- Number and Type of Pages and Controls, page 4-41
- Custom User Interface, page 4-42

For details about creating a User Interface, see the *Oracle Configurator Developer User's Guide*.

### Display Conditions

Using display conditions to hide items is generally more expensive than allowing all options to be displayed. However, the overall effect on performance is minimal when only a small number of items on a page have display conditions. For best performance, avoid placing many items with display conditions on the same page.

## Graphics

The number and size of GIFs in a page does not increase the time needed to render the screen on the server. However, reducing the number of controls and GIFs on the page may improve the performance of rendering the browser page on the client machine.

## Number and Type of Pages and Controls

The number of Oracle Configurator pages and the number of UI controls on each page influences runtime performance. Increasing the number of controls increases the time needed to render a page. This is due to browser resource limitations, not an inherent limitation in Oracle Configurator. For DHTML UIs, Oracle recommends a maximum of 8 to 10 UI controls and 5 graphics per page. The performance of an HTML UI is not adversely affected by pages containing a large number of UI controls.

The type of controls on a page does not influence server performance. For example, Drop-down Lists and Selection Lists take the same amount of time to render. However, In a DHTML UI, Dropdown Lists may render faster than Selection Lists, and the time it takes to render a Selection List is dependent on how many Options are displayed in the list.

## Configuration Summary User Interface Template

By default, the predefined UI Master Templates use the Summary with Status Region Content Template to display all orderable items in the Configuration Summary page at runtime. The default template displays the Summary Table fully expanded, which can negatively affect performance in a large configuration.

To improve performance of the Configuration Summary page:

1. Create a custom configuration summary UI Content Template. (Refer to the predefined template to see which UI elements are required and understand the structure.)

   When creating the Summary Table UI element, deselect **Expand All Levels on Entry.**

2. Create a custom UI Master Template.

3. In the Utility Templates section of your UI Master Template, select your custom configuration summary template for the Configuration Summary/Preview setting.

4. Use your custom UI Master Template to generate the UI.

### Custom User Interface

A Custom User Interface is created outside of Oracle Configurator Developer. It is usually coded as a set of Java Server Pages that display model content via the Configuration Interface Object (CIO). You create a custom UI by writing custom code that allows your configuration model to interface with the Configuration Interface Object (CIO).

Because it interacts with the CIO, it is important to optimize the CIO calls for best performance. For more information, see Configurator Extension Design, page 4-42.

## Large Amounts of End-User Data Collected Using Configurator Extensions

In rare cases, you may have unconstrained end-user data, meaning it is not constrained by or does not participate in rules. Collecting large amounts of that kind of data from end users, especially if it is repetitive, can degrade the usability of the Oracle Configurator UI. If you can identify data that could be collected outside the main interactive configuration session, especially if it lends itself to being collected in a tabular or spreadsheet form and is not orderable or constrained by rules, consider the following implementations:

- If you only need to collect moderate amounts of end-user data, design a Configurator Extension bound to the `onCommand` event to launch a separate child window during the main interactive configuration session that allows end users to enter data. Moderate amounts of end-user data might be no more than 100 rows, or take no more than a few minutes to enter.

- If end users have to enter a large volume of data or do not have time to enter data during configuration sessions, create a separate application for collecting the data outside Oracle Configurator. Implement a Configurator Extension to collect this data for inclusion in the configuration. In cases where the collected data needs to be validated, consider the following possible implementations:

  - Design a Configurator Extension bound to the `onCommand` event to launch a separate child window that displays the previously collected data for editing and then validates the modified data.

  - Design a Configurator Extension bound to the `onCommand` event that alerts end users to the invalid data so they can correct entries in the separate application used for collecting data and begin a new configuration session with valid data.

## Configurator Extension Design

The design of Configurator Extensions and how they interact with the configuration model as well as other software may affect performance. This section presents the following topics:

- Avoiding Unnecessary Interactions, page 4-43
- Accessing Runtime Nodes, page 4-43
- Components and Requests, page 4-45
- Adding and Deleting Instantiable Components, page 4-45
- Impact of Making Connections Among Components, page 4-46
- Optimization of Configurator Extensions that Change Model Structure, page 4-47
- Optimization of Validation Configurator Extensions, page 4-52

Configurator Extensions must use the Configuration Interface Object (CIO) to interact with the configuration model. See the *Oracle Configurator Extensions and Interface Object Developer's Guide* for additional information about the CIO and Configurator Extensions.

## Avoiding Unnecessary Interactions

Design Configurator Extension to avoid any unnecessary interactions, such as recursively processing unselected Model subtrees or creating large numbers of Configurator Extension instances by associating Configurator Extension instantiation component instances that occur in large numbers.

## Accessing Runtime Nodes

- The method `RuntimeNode.getChildByID()` is likely to cause slow performance if used on runtime nodes with many children. It is better to use `RuntimeNode.getChildByName()`, even on runtime nodes with few children.

- Use Java `HashMap` objects to map names, IDs, and paths to RuntimeNodes so that you can use `RuntimeNode.getChildren()`. If `HashMap`s cannot be used, use a localized search by starting only at the Components where you can certainly find the nodes. For better performance, use a localized search only once. In other words, do not construct your Configurator Extension such that it navigates the tree multiple times, or unnecessarily.

  Additionally, be sure to always cache the runtime node to allow faster access on subsequent calls to the extension.

- Keeping track of all changed runtime nodes after a user action is an expensive operation. Use `Configuration.getDeltaListIterator()` immediately before calling `Configurator.commitConfigTransaction()`. This iterator returns only the changed nodes with their new and old states and counts.

- Configurator Extension code that invokes the CIO to make unnecessary state and count assertions should be avoided. Use `IState.getState()` or `IState.getCount()` to determine the state or count, and do not reset the state or count if it is already as desired.

  In some cases, setting the state of a node may cause a contradiction. For example, setting the state of a node that is Logic False due to an Excludes Logic Rule. If you know that setting the node's state will cause a contradiction and you have no intention of overriding the assertion, then do not set the state.

  > **Warning:** When you use this technique of ignoring assertions, the contradiction is not obvious for opposite states if the contradictory state is due to a Default Logic Rule, in which case the assertion might succeed. For example, the state of a Boolean Feature is Logic True. If this node participates in a Defaults Logic Rule, then its logic state could be caused by this rule, and you can change the state to False without causing a contradiction. However, if the True state is due to an Excludes, Requires, or Negates Logic Rule, then a contradiction will occur when you change the node's state.

- Configurator Extensions and custom CIO code making many calls to `IState.getState()` to check for certain states can be slow.

  For each node on which to check the state, call `IState.getState()` only once. Then use the following static methods on the returned `int`:

  - `StateNode.isTrueState()`
  - `StateNode.isFalseState()`
  - `StateNode.isUserState()`
  - `StateNode.isLogicState()`
  - `StateNode.isUnknownState()`

  You can also use the following methods to combine state checks:

  - `StateNode.isSelected()`
  - `StateNode.isTrue()`
  - `StateNode.isFalse()`
  - `StateNode.isUser()`
  - `StateNode.isLogic()`
  - `StateNode.isUnknown()`

- Configurator Extensions and custom code invoking the CIO that makes many calls to `IState.getState()` or `OptionFeature.getSelectedOptions()` to get the Options selected as a result of a transaction can be slow. Instead, use `Configuration.getSelectedItems()`, which uses already-calculated information about all the TRUE items in the whole configuration.

  If you are also making assertions (with `setState()`, `setCount()`, `setValue()`, and so on), wrap all those assertions in a transaction.

  In order for the CIO mechanism to be able to update the selected item list, make sure you commit the transaction before calling `Configuration.getSelectedItems()`.

- CIO methods may or may not be expensive, depending on the circumstances of the configuration model. Making many calls to any CIO method that is expensive can be slow. Instead, call the method once and then cache the results for reuse. For example, the information from `Configuration.getAvailableNodes()` should be queried only when necessary, such as on screen flips and after user assertions. (Note that you can also obtain availability information by using `AvailabilityDeltaValidator`.)

  Use profiling tools (such as JProbe) to evaluate the performance cost of called CIO methods.

## Components and Requests

Programmatic changes to the configuration model are dependent on the sequence of events. For example, to instantiate some Components and programmatically set some Features in each Component, you could use either of these approaches:

- Set the Features as you add each Component
- Add all Components and then set all the Features

The second approach is faster, because the creation of each Component requires the retraction of all previous inputs (all user selections and all default selections). In the second approach, you can assert inputs and defaults only once, between when you add all the Components and when you set all the Features in all the added Components. The expensive events that you are avoiding in this second approach are the retractions *and reassertions* of inputs and defaults, which are processed every time that a Component is added.

If you are modifying the structure of a Model, you can improve performance by using the technique described in Optimization of Configurator Extensions that Change Model Structure, page 4-47.

## Adding and Deleting Instantiable Components

Adding or deleting an instance of a `ComponentSet` can be very expensive in terms of performance. For example:

- Adding an instance causes all previous requests to be retracted, then reasserted (that is, added and asserted again).

- Adding an instance is particularly expensive when there are default values set by configuration rules. Retracting default assertions is time-consuming and iterative. The initial values set in Configurator Developer for Boolean Features should also be regarded as default values.

- Deleting instances is a very expensive operation, and may not be necessary at the end of the Configurator Extension event sequence, since the Configurator Extension may need to add the same number of Components back into the Component Set anyway.

To avoid these performance problems, follow these guidelines for adding instances of a `ComponentSet`:

- Try to delay setting the states or values of nodes until after all instances are added. Add an instance when there are as few settings as possible.

- When deleting instances (if you have retracted all the requests), you can reuse the instantiated Components by computing the number of children in the `ComponentSet` and keeping track of the index number of the Component that you are setting Feature values for. You can then delete any Components that you have not reused. To determine the number of children, you can use `IRuntimeNode.getChildren().size()` or `ComponentSet.getCount()`.

  You may also want to consider using the method `ConfigTransaction.allowOnlyStructuralChanges ()`. For details, see Optimization of Configurator Extensions that Change Model Structure, page 4-47.

## Impact of Making Connections Among Components

Connectivity among components can add further complexity to the suggested sequence described in Detailed Sequence, page 4-47. The best approach depends on understanding the number of connections you plan to make, and how many you expect to fail. Completing a connection involves the creation and loading of all rules that are enforced as a result of the connection. That means all current requests have to be retracted and reapplied after the net is added.

The optimum point at which to make the connection is between the retraction of all requests and adding or reusing component instances.

If you do not expect the end user requests or the new values to make a difference on whether connection is allowed, then you need to add or reuse component instances, or reassert applicable requests before the connection.

The number of connections may cause you to adjust your thinking, due to the impact of retracting inputs.

You may also want to consider using the method `ConfigTransaction.allowOnlyStructuralChanges ()`. For details, see Optimization of Configurator Extensions that Change Model Structure, page 4-47.

## Optimization of Configurator Extensions that Change Model Structure

Use the following general strategy to optimize performance of a Configurator Extension that changes model structure:

1. Store all needed values.
2. Begin a transaction and call `ConfigTransaction.allowOnlyStructuralChanges()`.
3. Make changes to model structure. Model structure changes are defined as:
   - Adding or deleting instantiable components
   - Connecting or disconnecting component instances
4. Delete extra component instances.
5. Commit or roll back the transaction.
6. Set values on components.

The structural change transaction enables you to mark a transaction as being only for modifying the product structure. When you use the method `ConfigTransaction.allowOnlyStructuralChanges()`, Oracle Configurator automatically retracts the inputs and stores them until you close the transaction. If the end user changes any inputs, Configurator displays an exception.

The principle behind this strategy is that making structural changes in a block, inside a transaction that suspends the usual retraction and reassertion of requests, optimizes the event sequence. Detailed Sequence, page 4-47 explains these events in more detail.

### Detailed Sequence

Based on the preceding guidelines, optimize the performance of a Configurator Extension that changes model structure by designing it according to the following sequence:

1. Store all needed values and logic states in temporary variables.

   These are the values that are needed for later calculations and which would be lost by retracting requests.

2. Begin a transaction. Use `Configuration.beginConfigTransaction()`.

3. Call `ConfigTransaction.allowOnlyStructuralChanges()`.

   This method puts the configuration into a state in which the only changes allowed are those that modify the structure of the Model. Model structure changes are defined as:

   - Adding or deleting instantiable components
   - Connecting or disconnecting component instances

   You cannot make any requests (changes to values or logic states in the configuration) while in the structural-changes-only state. If you make a request, a non-overridable logical exception is thrown.

   The `allowOnlyStructuralChanges()` method retracts all user and non-overridable requests. The requests are stored until your transaction is committed or rolled back. If a contradiction occurs during the retraction, then the retraction is aborted, and the transaction loses its structural-changes-only state.

   Any nested transactions that you create inherit the structural-changes-only state of the parent transaction.

   > **Caution:** Do not call any methods that get values or logic states in the configuration while it is in the structural-changes-only state. Because the `allowOnlyStructuralChanges()` method previously retracted all requests, any values or states in the configuration returned by a query will be incorrect.

4. Add or reuse component instances, as suggested in Adding and Deleting Instantiable Components, page 4-45. To add a component instance, use `ComponentSet.add()`.

5. If you know that you have extra component instances, you may be able to delete them, as suggested in Adding and Deleting Instantiable Components, page 4-45. To delete a component instance, use `ComponentSet.delete()`.

6. Commit or roll back the transaction. Use `Configuration.commitConfigTransaction()` or `Configuration.rollbackConfigTransaction()`.

   When you commit or roll back the transaction, the CIO removes the configuration from the structural-changes-only state, and reasserts all the previously retracted user and non-overridable requests. If a contradiction occurs during the reassertions, then a `LogicalOverridableException` is thrown. You can override the exception and examine the failed requests. See the *Oracle Configurator Extensions and Interface Object Developer's Guide* for details on exceptions and failed requests.

7. Set values on Component instances or their children. Use `setState()`,

`setCount()`, `setValue()`, and so on.

## Comparison of Coding Approaches

Consider an example of creating instances of a certain number of instantiable Components, and setting the values of Features in each one. Setting Components and Feature Values One at a Time (Slower), page 4-49 creates a Component, then sets its Features before creating the next Component. Setting All Components and Then All Feature Values (Faster) , page 4-49 creates all the Components, then sets all of their Features.

**Setting Components and Feature Values One at a Time (Slower)**

```
// Block for 1st component
Add Component1 to ComponentSet
Set Feature1 in Component1
Set Feature2 in Component1
// Block for 2nd component
Add Component2 to ComponentSet
Set Feature3 in Component2
...
```

**Setting All Components and Then All Feature Values (Faster)**

```
// Block for all structure changes
Begin strucure-only transaction
Add Component1 to ComponentSet
Add Component2 to ComponentSet
Commit strucure-only transaction
// Block for all assertions
Set Feature1 in Component1
Set Feature2 in Component1
Set Feature3 in Component2
...
```

Setting Components and Feature Values One at a Time (Slower), page 4-49 is subject to the negative performance effects identified in Optional and Multiple Instantiation, page 4-9. Setting All Components and Then All Feature Values (Faster) , page 4-49 results in significantly faster runtime performance.

The reason why Setting Components and Feature Values One at a Time (Slower), page 4-49 is slower is shown by comparing Detailed Slower Approach, page 4-50 with Detailed Faster Approach, page 4-50. The operations that have to be performed by the CIO when adding `ComponentSet` instances are highlighted in **boldface**. repeats *all* of the operations for *each* addition of an instance, so that each addition takes longer than the preceding one, and the total processing time is proportional to the number of instances to be added. If there are many instances to be added, the impact is great. Detailed Faster Approach, page 4-50 performs the operations once, greatly lessening the effect of adding many instances.

### Detailed Slower Approach

```
...
Set Feature1 in Component1
Set Feature3 in Component2
// Block for 1st component
Retract requests // (2 requests)
Add Component1 to ComponentSet
Reassert requests // (2 requests)
Set Feature1 in Component1
Set Feature2 in Component1
...
// Block for 2nd component
Retract requests // (2 existing + 2 new = 4 requests)
Add Component2 to ComponentSet
Reassert requests // (4 requests)
Set Feature3 in Component2
...
```

For *n* components added, this approach results in *n* retractions and reassertions, each of which is an expensive operation. Note also that the queue of requests grows over time, further increasing the expense of retraction and reassertion.

Contrast this with the result for Detailed Faster Approach, page 4-50.

### Detailed Faster Approach

```
...
Set Feature1 in Component1
Set Feature3 in Component2
Set Feature1 in Component1Set Feature2 in Component1
Begin strucure-only transaction // retracts all existing requests (4
requests)
Add Component1 to ComponentSet
Add Component2 to ComponentSet
Commit strucure-only transaction // reasserts all existing requests (4
requests)
...
Set Feature3 in Component2
...
```

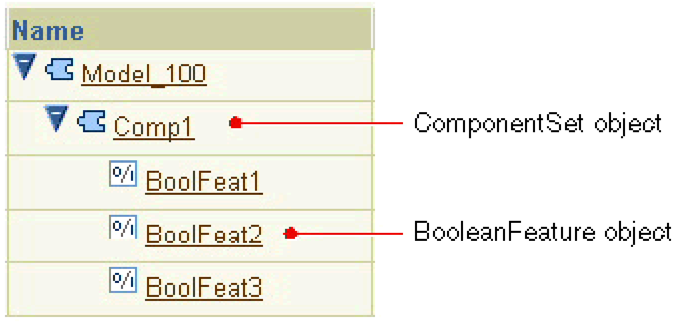
For *n* components added, this approach results in only 1 retraction and reassertion. Contrast this with the result for Detailed Slower Approach, page 4-50. You could further improve performance by performing structural changes before making requests, thus reducing the size of the queue of requests.

## Code Example

Consider the simple runtime Model structure shown in Model Structure for Adding Components, page 4-51. At runtime, this Model contains a `ComponentSet` object and several `BooleanFeature` objects.

The following graphic shows a simple Model, consisting of a `ComponentSet` object named `Comp1`, which contains three `BooleanFeature` objects called `BoolFeat1`, `BoolFeat2`, and `BoolFeat3`.

***Model Structure for Adding Components***



Assume that you have written a Configurator Extension bound to the `onCommand` event that sets the state of the Boolean Features as illustrated in Setting Features (Slower Code), page 4-51.

**Setting Features (Slower Code)**

```
...for(int i=0; i<100; i++) {
    comp = Comp1.add();

((BooleanFeature)comp.getChildByName("BoolFeat1")).setState(IState.TRUE)
;

((BooleanFeature)comp.getChildByName("BoolFeat2")).setState(IState.FALSE
);

((BooleanFeature)comp.getChildByName("BoolFeat3")).setState(IState.TRUE)
;
}
...
```

You can significantly improve the performance of this operation by modifying the code as shown in Setting Features (Faster Code), page 4-51. The differences are highlighted in **boldface**.

**Setting Features (Faster Code)**

```
...
ConfigTransaction tr = config.beginConfigTransaction();
ConfigTransaction.allowOnlyStructuralChanges();
for(int i=0; i<100; i++){
    comp[i] = Comp1.add();
}config.commitConfigTransaction(tr);for(int i=0; i<100; i++){
    ((BooleanFeature)comp[i]
.getChildByName("BoolFeat1")).setState(IState.TRUE);
    ((BooleanFeature)comp[i]
.getChildByName("BoolFeat2")).setState(IState.FALSE);
    ((BooleanFeature)comp[i]
.getChildByName("BoolFeat3")).setState(IState.TRUE);
}
...
```

Setting Features (Faster Code), page 4-51 improves performance by adding all the `ComponentSet` instances, then setting all the `BooleanFeature` values. This follows the principles identified in Optional and Multiple Instantiation, page 4-9, and the example shown in Setting All Components and Then All Feature Values (Faster) , page

4-49 under Comparison of Coding Approaches, page 4-49.

## Optimization of Validation Configurator Extensions

In general, Oracle Configurator applies validation tests and defaults more often than would be expected. The `onConfigValidate` event is called whenever the end user selects an option or enters an input in the runtime Oracle Configurator. In a Configurator Extension bound to this event, defaults should be used as judiciously as possible, as described in Defaults Rules Versus Alternatives to Default Selections, page 4-20. Validation tests must also be minimal. You should arrange the code so that `onConfigValidate` is only called when necessary, and the condition test is quick.

The examples Minimizing Validation Tests on a Configuration Model, page 4-52 and Causing More Validation Tests on a Configuration Model, page 4-52 show two ways of applying a validation test. Minimizing Validation Tests on a Configuration Model, page 4-52 performs better because it performs the task of setting up the validation (by finding the node in each call) in a method bound to the `postCXInit` event, which occurs once, and then performs the validation test only after all other calls are completed.

The example Causing More Validation Tests on a Configuration Model, page 4-52 sets up the validation as part of the validation test, so the setup is performed every time the `onConfigValidate` event occurs.

If you are using Oracle Configurator release 11.5.9 or earlier, use `onConfigValidate`. If you are using release 11.5.10 or later - and only the end user can modify the Feature - then use the `postValueChange` event instead. This event has a more limited scope and is much more efficient than `onConfigValidate`. Refer to the *Oracle Configurator Extensions and Interface Object Developer's Guide*for more information.

**Minimizing Validation Tests on a Configuration Model**

```
OptionFeature f = null;
doPostCXInit() {     // bind to postCXInit event
      f = (OptionFeature) root.getChildByName("feature_name");
   }
doOnConfigValidate () { // bind to onConfigValidate event
  if (f.isTrue()) {
  }
}
```

**Causing More Validation Tests on a Configuration Model**

```
doPostCXInit() {     // bind to postCXInit event
   // empty
   }
doOnConfigValidate () { // bind to onConfigValidate event
  OptionFeature f = (OptionFeature) root.getChildByName("feature_name");
  if (f.isTrue()) {
  }
}
```

# Part 2

## Case Studies

To solve your configuration problem, Oracle Configurator may require a combination of best practices that is not obvious. This section presents some examples of common configuration problems and optimal design solutions best suited to an Oracle Configurator implementation.

# 5

# Many Large BOM Models

This chapter describes an Oracle Configurator project involving many large BOM Models with much explicit and repetitive structure that is best modeled as a single top-level BOM Model containing a deep hierarchy of generic structure and abstractions.

This chapter covers the following topics:

- Overview
- Project Description
- A Deficient Modeling Approach
- The Suggested Modeling Approach

## Overview

This case study explores redesigning a project consisting of many large BOM Models (such as 90,000), each with a large number of options for selection (such as 150,000). The goal is to fulfill performance and usage expectations in a way best suited to the strengths and characteristics of Oracle Configurator.

This project illustrates the following best practices:

- Explicit Model Structure Versus Abstractions, page 4-2
- Explicit Model Structure Versus References, page 4-7
- Optional and Multiple Instantiation, page 4-9
- Shallow Versus Nested or Deep Hierarchy, page 4-14
- Items Versus Alternatives to Items, page 4-14
- Large Option Features and Option Classes, page 4-18
- Optimizing User Interface Performance, page 4-40

## Project Description

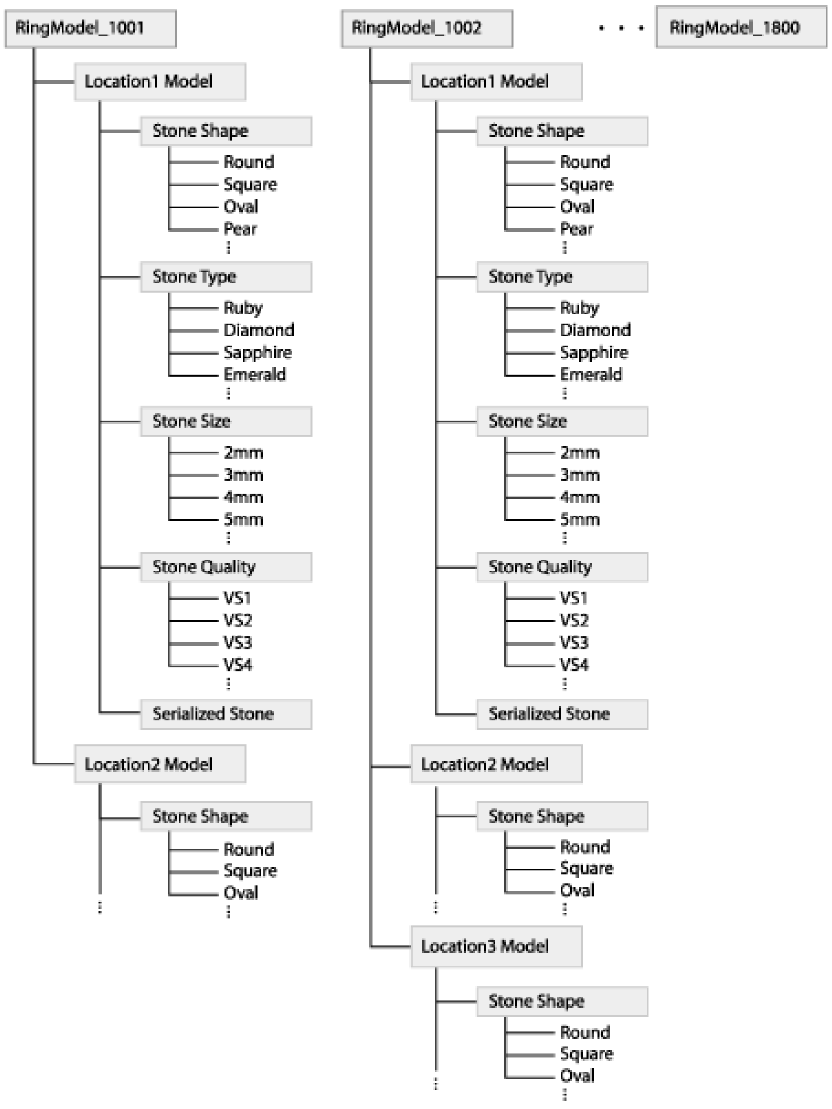
A manufacturer and wholesaler of jewelry offers customers the opportunity to custom configure rings and bracelets. Configuring a ring consists of placing stones with characteristics such as shape, size, type, clarity, quality, into a specified location on the ring. Each ring consists of up to 60 locations. For example, a customer orders a 3-setting ring by configuring a 2mm round ruby at location 1 and 3, and a 3mm square emerald at location 2.

## A Deficient Modeling Approach

An initial modeling approach might be to have a specific BOM for each ring model. Each ring model contains explicit submodels for each of 60 possible locations. Each Location submodel contains the specifications for the stone placed in that location. Stone specifications are shape, type, and so on. The 60 locations can be arranged in one of many possible settings. So for instance, if a ring is defined with three locations, the locations can be arranged in a line with two bevels flanking a peg. The locations on a 60-location ring could be arranged in many more possible settings. Multiplying the 60 rings containing from 1 to 60 Location submodels by the various possible settings results in 90,000 large, explicit BOM Models with up to 150,000 options in each BOM Model. Many Explicit Ring Models, page 5-3 shows an example of such explicit, flat Model structure across many Models.

The graphic Many Explicit Ring Models, page 5-3 shows RingModel 1000, RingModel 1001, and RingModel 1800, and shows that 800 RingModels contain repetitive structure.

***Many Explicit Ring Models***



Each Location has the same kind of configurable characteristics, such as particular stones of specific shapes and sizes. In the individual RingModels, the structure for configuring Locations is repeated over and over again, up to the maximum number of 60 locations. For example, if 100 rings are defined with 3 locations and another 100 rings are defined with 4 locations, then in 200 ring models the submodel structure for Location is repeated 700 times.

The ring configuration can contain the same configuration of a stone in several locations or a stone with different characteristics in different locations.

### Perceived Advantages

By defining each possible model explicitly, the manufacturer can maintain and sell each model independently of all the other models. This includes sourcing items to separate organizations and preserving existing routings.

By not defining abstractions, the manufacturer does not have to create a large number of rules to capture the large number of valid combinations because each model identifies explicit options.

### Overwhelming Disadvantages

A design consisting of 90,000 individual Models, each representing a unique combination, does not leverage the power of Oracle Configurator and causes the following:

- Performance problems when importing 90,000 BOM Models into Oracle Configurator Developer
- Performance problems at runtime caused by the large number of items (up to 150,000 options per model) that must be instantiated
- Costly maintenance of 90,000 explicit Models with repetitive structure of many similar items across many of the models
- Costly memory usage during preload, initialization, and UI screen display
- Scalability issues as the business expands to more models and more options within those models

The runtime performance issues at a minimum are prohibitive.

## The Suggested Modeling Approach

The suggested modeling approach avoids the problems presented in A Deficient Modeling Approach, page 5-2 and applies numerous best practices described in Best Practices, page 4-1. Rather than define separate explicit models repeatedly for each possible ring, a well-designed implementation defines the duplicated structure by references and abstractions, using optional instantiation to optimize performance.

### Applying Best Practices to Your Model Structure

This case involves several separate models that represent unique rings. However, the ring models contain a large amount of similar structure, such as settings and locations for mountings on the rings.

To leverage the advantages of deep hierarchy and optional instantiation, you need to create abstractions of the structure that is similar across all ring models, as follows:

1. Combine the separate models into a single top-level model for configuring rings. RingModel_1001 through RingModel_1800 in Many Explicit Ring Models, page 5-3 become an abstract model called Ring Model in Top-level Ring Model with Abstractions, page 5-7. Instead of searching from a long list of rings with many pre-defined characteristics in RingModel_1001 through RingModel_1800, the end-user starts by selecting the characteristics of an undefined ring. The undefined ring is represented by Ring Model and the characteristics are the settings and the stones.

   This step applies the best practice:

   - Explicit Model Structure Versus Abstractions, page 4-2

2. Creating a single top-level Ring Model containing the settings structure, initially results in one huge, flat BOM Option Class containing all possible settings, of which only one will be selected. Not only are all settings loaded when the top-level Ring Model is loaded, but the end user is faced with selecting a setting from this very large list of options.

   To help end users see only those settings options that are relevant to their configuration, organize settings into groups by some criterion such as popularity or the ring's intended function. A Settings Model contains not all possible settings but BOM Option Classes of related or grouped settings.

   Each group of settings is a BOM Option Class containing only those options that belong in that group. Ideally, a specific Settings option should not appear in more than one group.

   Create a Settings Model that contains all the Setting Group BOM Option Classes.

   This step applies the best practices:

   - Explicit Model Structure Versus Abstractions, page 4-2

   - Grouped Versus Ungrouped Items, page 4-18

3. To involve only that group of settings in the configuration that contains relevant settings, change the Settings Group BOM Option Classes into BOM Models and make them optionally instantiable, which means the value of **Instance Minimum** is 0, **Maximum** is 1. See Top-level Ring Model with Abstractions, page 5-7.

   This step applies the best practice:

   - Optional Instantiation of BOM Option Classes, page 4-9

4. In the top-level Ring Model, you can significantly decrease the amount of repetitive structure by defining an ATO model for the various Stone Types, each containing BOM Option Classes for the configurable characteristics of the Stone such as size

and quality.

This step applies the best practice:

- Explicit Model Structure Versus Abstractions, page 4-2

5. To specify the location where the Stone will be placed, create a Location Option Feature for the Stone Model. The Location List of Options contains the maximum number of Locations allowed for any ring. Making Location an attribute of the Stone Model rather than an Item in the Ring Model is appropriate because the Stone is configurable and orderable, not the Location.

   This step applies the best practice described in:

   - Items Versus Alternatives to Items, page 4-14

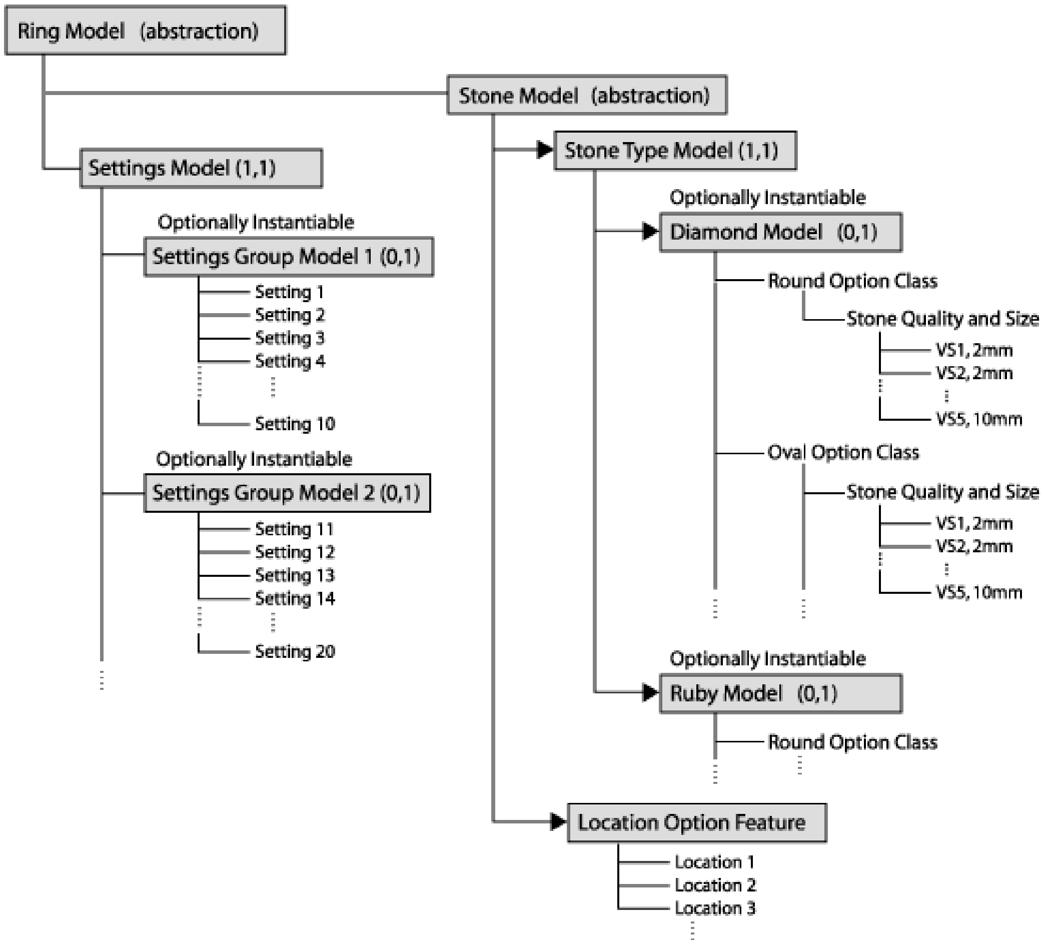
6. In the Stone Model, refine your design further by creating a Stone Type -BOM Model containing each stone type (Diamond, Ruby, and so one) as a separate Model that is optionally instantiable. Each Model in the Stone Type -BOM Model contains BOM Option Classes of the configurable characteristics of that stone type, such as shape and size.

   This applies the following best practices:

   - Explicit Model Structure Versus Abstractions, page 4-2
   - Explicit Model Structure Versus References, page 4-7
   - Shallow Versus Nested or Deep Hierarchy, page 4-14

The graphic Top-level Ring Model with Abstractions, page 5-7 shows the Ring Model that represents an abstraction of RingModel 1000 to 1800.

***Top-level Ring Model with Abstractions***



## Applying Best Practices to Further Optimize the End-User Experience

After completing the steps in the previous section (Applying Best Practices to Your Model Structure, page 5-4), continue with the following steps:

1. To allow end users to select only the Locations relevant to their ring configurations, write rules that disallow selecting Locations that are not allowed for a particular Setting. For example, when the end user has selected a 3-location setting, only 3 Locations should be selectable.

2. To further enhance usability, define a display condition so only valid Locations are displayed in the UI.

   This applies the best practice:

   - Optimizing User Interface Performance, page 4-40

3. You can also write rules to ensure that each required Location for the selected Setting is selected and that, as the end user configures the Stones for a ring, only those Locations remain available for selection that have not yet been selected.

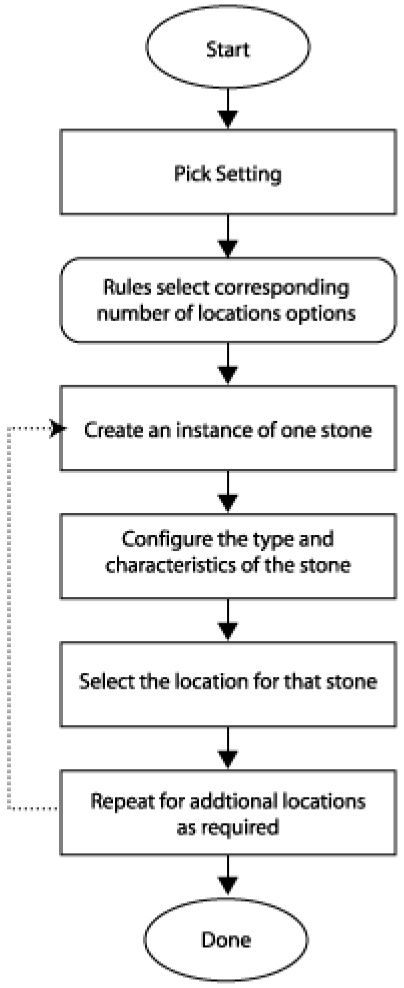
## The Resulting End-User Flow

In the Order Management Sales Order Pad, the end user selects the top-level Ring Model and clicks the **Configurator** button to start configuring a ring. At start up, no Settings or Stone Models are instantiated. In the Oracle Configurator UI, the end user selects a Settings Group. For example, the end user picks a 3-setting ring. The end user must then configure between one and three Stones for the 3 Locations of the 3-setting ring. The first instance of the Stone Model allows the end user to configure the Stone Type and select from the three available Locations. If the end users picks all three Locations, the configuration is complete and the same Stone configuration will be set in each of the three locations. If not all three Locations are selected, the end user adds additional instances of the Stone Model, configures them and selects remaining available Locations until the ring is fully configured.

The diagram in End User Flow for Configuring a Ring Model, page 5-9 shows a typical end-user flow.

The graphic End User Flow for Configuring a Ring Model, page 5-9is an end-user flow chart for ordering the Ring Model. The end user picks a setting from the Settings submodel and rules select the corresponding number of Locations. The end user then creates an instance of a Stone and configures its characteristics. Finally, the end user completes the Stone configuration by selecting a location for that Stone. These tasks repeat for all locations required by the selected Setting.

***End User Flow for Configuring a Ring Model***


Start
Pick Setting
Rules select corresponding number of locations options
Create an instance of one stone
Configure the type and characteristics of the stone
Select the location for that stone
Repeat for addtional locations as required
Done

## Advantages of This Modeling Approach

### Perceived Disadvantages

Creating deeper hierarchy and abstractions may require greater designing effort and more rule definitions than an explicit design approach.

A single BOM Model can be sourced to only one organization.

Downstream ERP applications may require additional setup for option-dependent routings or using configuration attributes. Configuration attributes are explained in *Oracle Configurator Methodologies.*

### Overwhelming Advantages

Comparing this approach to the deficient one presented in A Deficient Modeling

Approach, page 5-2, a single top-level BOM Model with structure that is only instantiated as needed provides the following advantages:

- Importing a single BOM Model with abstractions is significantly faster than importing 90,000 large BOM Models with explicit, repetitive structure

- Instantiating only the substructures required by the current configuration is significantly faster at runtime than instantiating all items

- Maintaining a single top-level BOM Model with abstractions is quicker, more flexible to change, and less prone to error than maintaining 90,000 explicit models with repetitive structure of many similar items across many of the models

- Preloading, initializing, and displaying the UI screens for a single BOM Model with optionally instantiated items uses significantly less memory than the same operations for 90,000 explicit models

- Scaling the single top-level BOM Model with abstractions to accommodate exponentially more items as the business grows does not significantly affect the performance baseline

# 6

# Many BOM Items

This chapter describes an Oracle Configurator project involving many BOM Items that are not orderable and could be better implemented as Features or configuration attributes. Configuration attributes are explained in *Oracle Configurator Methodologies*.

This chapter covers the following topics:

- Overview
- Project Description
- A Deficient Modeling Approach
- The Suggested Modeling Approach

## Overview

This case study explores redesigning a project that contains a profusion of items not all of which are part of the order.

This project illustrates the following best practices:

- Items Versus Alternatives to Items, page 4-14
- Grouped Versus Ungrouped Items, page 4-18
- Optional and Multiple Instantiation, page 4-9
- Optimizing User Interface Performance, page 4-40

## Project Description

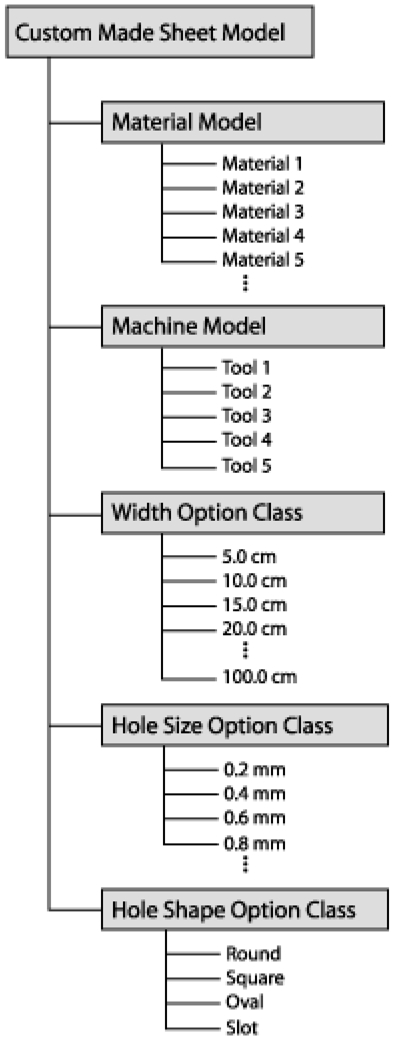
A manufacturer and retail supplier of perforated metal sheets offers customers both standard products available from local stock and custom-made products. When ordering metal sheets, customers must specify the type and grade of the material, its width, length, and thickness, and various characteristics about the perforations. From this information, the manufacturer must determine the machines and tools to use to produce the metal sheets.

## A Deficient Modeling Approach

An initial modeling approach might be to define all possible characteristics of a metal sheet as items in the BOM Model. This would result in a top-level BOM structure for Metal Sheet, which contains submodels for specifying material and machines, and contains BOM Option Classes for specifying width, length, thickness, material grade, hole type, hole size, tools, and so on. Each BOM Option Class contains a large number of selectable items. Widths, lengths, thicknesses, material grades, hole types, hole sizes, and tools are all defined as items. For example, the Width BOM Option Class contains items that represent every possible selectable width for the metal sheet. Model With Too Many Items, page 6-3 shows an excerpt of such a model.

The graphic Model With Too Many Items, page 6-3 shows a Custom Metal Sheet Model containing submodels for Material and Machine, and BOM Option Classes for Width, Hole Size, and Hole Shape.

***Model With Too Many Items***



## Perceived Advantage

- By defining each characteristic of the product as an item, the manufacturer does not have to create rules to capture a large number of valid combinations of characteristics
- Defining all of the information as part of the BOM Model ensures that all of the information is passed back to Order Management and to the downstream manufacturing applications.
- Defining each characteristic as an item provides a straightforward means by which to associate a price with each characteristic.

### Overwhelming Disadvantages

A design that forces all of this information into the BOM Model inflates the size of the model structure, does not leverage the flexibility of Oracle Configurator, and causes the following problems:

- Poor performance when importing large numbers of items into Oracle Configurator Developer
- Poor performance at runtime caused by the large number of items (more than 10,000) that must be loaded into memory and displayed
- Poor usability of a UI that requires finding the desired item among a large number of items
- Insufficient scalability and maintainability as the business expands to more characteristics and dimensions

## The Suggested Modeling Approach

Rather than defining thousands of BOM items to capture the characteristics of the configured item, a well-designed implementation reviews the requirement for this large number of items and based on the item's use, redefines it either as a configuration attribute or Feature.

### Applying Best Practices to Your Model Structure

The following suggested modeling approach applies best practices to achieve improved performance and usability:

1. Redesign the BOM Model for perforated metal sheets by analyzing the purpose of each submodel and BOM Option Class. By finding alternatives to items where possible, you can make the BOM structure significantly smaller and improve performance and maintainability.
    1. Identify which items appear on an order line and will be picked or assembled. For example, items in the Materials submodel must appear on the order line. These items must remain in your model as BOM Standard Items.
    2. Of the items that do not need to appear on the order line, determine whether you need end user input for the value of the item, or if the item has static value used in calculations or Compatibility rules. Define static values as Properties of the items they define.

       For example, the material grade has a static value (Premium, Medium, and so on) which can be defined as Properties of the Materials items.

   3. Define items whose values result from end user input as Features or as attributes on items. For example, hole size and width.

   4. Of the items that require end user input, determine whether they should be defined as Features or as configuration attributes. Those inputs that are required for downstream operations must be defined as configuration attributes. Write attribute values collected during the configuration session into the CZ_CONFIG_ATTRIBUTES table.

      For example, width, length, and thickness contribute to items that specify material. These values can be modeled as configuration attributes and associated with the Materials items that need to go to manufacturing.

      For details about implementing configuration attributes, see*Oracle Configurator Methodologies*. Note that configuration attributes are not accessible to downstream applications without customization.

   5. If the items are only needed during the configuration session, define them as Features.

      For example, the hole size and type affect the selection of tools needed for creating custom-made sheets. Tool selection occurs during the configuration session. Add a Numeric Feature to get the hole size and a Feature with a List of Options to get the hole shape from the end user.

   6. Define Configurator Extensions for selecting the appropriate tool based on end users selections for hole shape and size.

   This step applies the best practice:

   - Items Versus Alternatives to Items, page 4-14

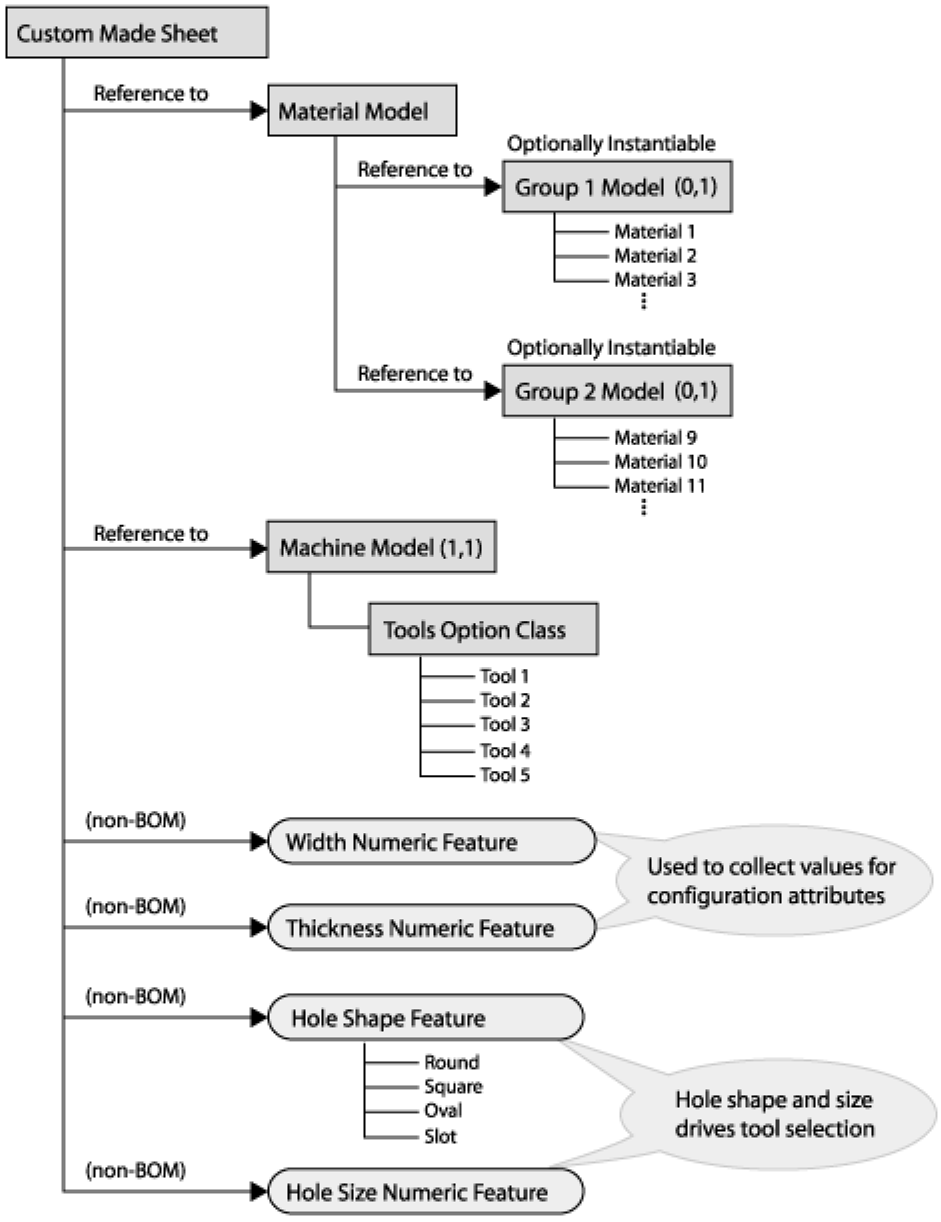
2. The Material submodel contains a large number of items. Organize these items under some logical grouping by making each logical group a separate model. Add non-BOM Features to get user inputs that determine which Material group to load. Each group model contains only items belonging to that group. For instance, if the group were based on material type, then add a Feature to capture a material type selection such as aluminum or steel.

   This step applies the best practice:

   - Grouped Versus Ungrouped Items, page 4-18

The graphic Redesigned Model With Fewer Items, page 6-6 shows a top-level root Custom Metal Sheet Model that contains references to a submodels for Material and Machine, as well as Features for Width, Thickness, Hole Shape, and Hole Size.

***Redesigned Model With Fewer Items***



## Applying Best Practices to Further Optimize the End-User Experience

After completing the steps in the previous section (Applying Best Practices to Your Model Structure, page 6-4), continue with the following steps:

1. To enhance the runtime performance, load only those groups of items that are needed by making instantiation of each group model optional, which means the value of **Instance Minimum** is 0, **Maximum** is 1. For example, a particular

submodel group of the Material model is instantiated based on certain end-user criteria.

This step applies the best practice:

- Optional and Multiple Instantiation, page 4-9

2. Hide the parts of the BOM that do not require user interaction. For example, hide the Machine submodel.

   This step applies the best practice:

   - Optimizing User Interface Performance, page 4-40

## The Resulting End-User Flow

The end user starts the order in the Order Management Sales Order Pad, selects the appropriate item to be configured, and launches Oracle Configurator. The Oracle Configurator UI starts with a page or pages for entering high-level order characteristics, such as material type, hole size, and hole shape. Based on the end user's selection of a material type, the Configurator Extension loads the appropriate Material model. The end user selects the material and then enters the dimensions for the material, such as height, width, and length. These dimensions are collected as configuration attributes. The end user then selects a tool selection button which invokes a Configurator Extension to determine the appropriate tool for processing the ordered specifications.

At the end of the configuration session, the Configurator Extension writes the dimension attributes to the CZ_CONFIG_ATTRIBUTES table and populates the order line with the ordered material and tool.

For more information about configuration attributes, see *Oracle Configurator Methodologies.*

## Advantages of This Modeling Approach

### Perceived Disadvantages

- Differentiating whether items are orderable or merely participants in completing a configuration may require greater designing effort and more rule definitions than simply defining all characteristics as individual items.

- Using configuration attributes requires customization to retrieve the dimension attributes information from CZ_CONFIG_ATTRIBUTES for use in downstream manufacturing applications. This customization must be reviewed for possible modification when you upgrade Oracle Applications.

- Associating prices with characteristics that are defined as configuration attributes requires customization and the use of Advanced Pricing rules.

## Overwhelming Advantages

Compared to the deficient approach presented in A Deficient Modeling Approach, page 6-2, the suggested approach provides the following advantages:

- Importing fewer BOM items is significantly faster
- Instantiating only groups of items at runtime that are needed by the current configuration is significantly faster than instantiating all items
- Maintaining a smaller BOM Model with fewer items is easier
- A relatively smaller BOM Model can be scaled better as the business grows without irreparably degrading performance

# A

# SQL Queries

This appendix contains some SQL*Plus queries that are useful for calculating the total number of nodes in a configuration model, and the number of different types of nodes in a configuration model.

This appendix covers the following topics:

- Number of Nodes by Type
- Number of Features by Type in a Configuration Model
- Number of Rules by Type in a Configuration Model
- Number of Rules by Relation in a Configuration Model

## Number of Nodes by Type

Use the Determine Number of Nodes by Type in a configuration model, page A-2 query to find out how many nodes of each type exist in a configuration model. The Numeric Identification of PS_NODE_TYPE table, page A-1 provides the numeric identification of the node types that need to be inserted in the query. For example, in the example that follows the table below, replace `ps_node_type ID` with 259 in the script to find out how many Component nodes exist in a configuration model.

***Numeric Identification of PS_NODE_TYPE***

| PS_NODE_TYPE | Description |
|---|---|
| 259 | Component |
| 261 | Feature |
| 262 | Option |

| PS_NODE_TYPE | Description |
|---|---|
| 263 | Model Reference (BOM or Non-BOM) |
| 264 | Model Connector |
| 272 | Total |
| 273 | Resource |
| 436 | BOM Model Root node |
| 437 | BOM Option Class |
| 438 | BOM Standard Item |

### Determine Number of Nodes by Type in a Configuration Model (not including Model References)

This query returns the number of nodes by type in a configuration model, but does not include nodes that belong to referenced Models. .

```
SQL> SELECT count(*)
FROM cz_ps_nodes
WHERE deleted_flag = '0'
and ps_node_type = ps_node_type ID
and devl_project_id = devl_project_id
/
```

### Determine Number of Nodes by Type in a Configuration Model (including Model References)

This query returns the number of nodes by type in a configuration model, including nodes that belong to referenced Models.

```
SQL> SELECT count(*)
FROM cz_ps_nodes
WHERE deleted_flag = '0'
and ps_node_type = ps_node_type ID
and devl_project_id in (select distinct component_id from
cz_model_ref_expls where deleted_flag = '0'
and model_id = devl_project_id)
/
```

In the examples above, insert a number from the Numeric Identification of PS_NODE_TYPE, page A-1 table for the variable `ps_node_type ID`, and insert the configuration model ID for `devl_project_id`. To determine a configuration model's `DEVL_PROJECT_ID`, see the Determine DEVL_PROJECT_ID of a Configuration Model, page A-3 example.

The following examples show queries that use the 'in' and 'not in' functions to find out how many BOM and non-BOM nodes exist in a configuration model.

#### References)

```
SQL> SELECT count(*)
FROM cz_ps_nodes
WHERE deleted_flag = '0'
and ps_node_type in (436,437,438)
and devl_project_id = devl_project_id
/
```

#### Determine Number of BOM Nodes in a Configuration Model (including Model References)

```
SQL> SELECT count(*)
FROM cz_ps_nodes
WHERE deleted_flag = '0'
and ps_node_type in (436,437,438)
and devl_project_id in (select distinct component_id from
cz_model_ref_expls where deleted_flag = '0'
and model_id = devl_project_id
/
```

#### Determine Number of non-BOM Nodes in a Configuration Model (not including Model References)

```
SQL> SELECT count(*)
FROM cz_ps_nodes
WHERE deleted_flag = '0'
and ps_node_type not in (436,437,438)
and devl_project_id = devl_project_id
/
```

#### Determine Number of non-BOM Nodes in a Configuration Model (including Model References)

```
SQL> SELECT count(*)
FROM cz_ps_nodes
WHERE deleted_flag = '0'
and ps_node_type not in (436,437,438)
and devl_project_id in (select distinct component_id from
cz_model_ref_expls where deleted_flag = '0'
and model_id = devl_project_id
/
```

#### Determine DEVL_PROJECT_ID of a Configuration Model

```
SQL> SELECT devl_project_id, name
FROM cz_devl_projects
WHERE name like '%Sen%'
and deleted_flag='0'
```

In this example, replace `Sen` with part of your model's name. The query returns both the Model's `devl_project_id` and its name, as shown below:

```
DEVL_PROJECT_ID            NAME
2020                       Sentinel Custom Desktop(204 137)
```

## Number of Features by Type in a Configuration Model

Use the queries in this section to find out how many Features of a specific type exist in a configuration model. For each query, refer to the table below to determine the value to enter for the *`feature_type ID`* variable. Refer to Determine DEVL_PROJECT_ID of a Configuration Model, page A-3to determine your configuration model's `devl_project_id`.

***Numeric Identification of FEATURE_TYPE***

| FEATURE_TYPE | Description |
|---|---|
| 0 | Option |
| 1 | Integer |
| 2 | Decimal |
| 3 | Boolean |
| 4 | Text |

### Determine Number of Features by Type in a Configuration Model (not including Model References)

```
SQL> SELECT count(*)
FROM cz_ps_nodes
WHERE deleted_flag = '0'
and ps_node_type = 261
and feature_type = feature_type ID
and devl_project_id = devl_project_id
/
```

### Determine Number of Features by Type in a Configuration Model (including Model References)

```
SQL> SELECT count(*)
FROM cz_ps_nodes
WHERE deleted_flag = '0'
and ps_node_type = 261
and feature_type = feature_type ID
and devl_project_id in (select distinct component_id from
cz_model_ref_expls where deleted_flag = '0'
and model_id = devl_project_id
/
```

## Number of Rules by Type in a Configuration Model

Use the query below to find out how many rules of each type exist in a configuration model.

When executing the query, insert the `model_id` of one or more models where indicated in the example, and enter a value from the table below for *`rule_type ID`*.

***Numeric Identification of RULE_TYPE***

| RULE_TYPE | Description |
|---|---|
| 21 | Logic Rule |
| 22 | Numeric Rule |
| 23 | Compatibility Rule |
| 24 | Compatibility Table |
| 25 | Func Preselect |
| 26 | Func Validate |
| 27 | Comparison Rule |
| 29 | Functional Companion |
| 30 | Design Chart |
| 33 | Runtime DISPLAY Condition |
| 34 | Runtime ENABLED Condition |
| 200 | Statement Rule |
| 300 | Configurator Extension Rule |

### Determine Number of Rules by Type in a Configuration Model

```
select
  model_id,
  rule_type,
  count (distinct rule_id) as distinct_rules,
  count (*) as rule_instances_all_models
from
  cz_model_ref_expls,
  cz_rules
where
  cz_model_ref_expls.DELETED_FLAG = '0' and
  cz_model_ref_expls.component_id = cz_rules.devl_project_id and
  cz_rules.deleted_flag = '0' and
  ps_node_type != 264 and
  cz_rules.rule_type = rule_type ID  and
  model_id in (list one or more models here, separated by commas)
group by
  model_id, rule_type
;
```

There are two rule counts in the above example. The rule counts usually return the same amount but not always. The reason for this is that it is possible for the same child model to be referenced more than once. In this case, a rule is likely to be loaded and active once for each place its model is included in the overall root model. The data storage volume is indicated by the DISTINCT_RULES calculation; this counts each unique rule only once. The runtime memory and processing load will usually be better indicated by the RULE_INSTANCES_ALL_MODELS calculation.

## Number of Rules by Relation in a Configuration Model

The Determine Number of Rules by Relation in a Configuration Model, page A-7 example shows the query for determining the number of rules by relation type in a configuration model.

***Numeric Identification of EXPR_RULE_TYPE***

| EXPR_RULE_TYPE | Description |
|---|---|
| 1 | Requires |
| 2 | Implies |
| 3 | Excludes |
| 4 | Negates |
| 5 | Defaults |

| EXPR_RULE_TYPE | Description |
|---|---|
| 6 | NumSelections |
| 8 | Contributes |
| 10 | Consumes |

### Determine Number of Rules by Relation in a Configuration Model

```
SQL> SELECT count(*)
FROM cz_rules
WHERE rule_type = rule_type
and expr_rule_type = expr_rule_type
and devl_project_id in (
SELECT distinct ps_node_id
FROM cz_ps_nodes
WHERE deleted_flag = '0'
and ps_node_type in(436,263)
START WITH ps_node_id = devl_project_id
CONNECT BY DECODE(PRIOR ps_node_type,263,PRIOR reference_id, PRIOR
ps_node_id)=
DECODE(PRIOR ps_node_type,263,ps_node_id,parent_id))
/
```

In the Determine Number of Rules by Relation in a Configuration Model, page A-7 example, insert the `RULE_TYPE` number shown in the Numeric Identification of RULE_TYPE, page A-5 table for the variable *`rule_type`*, insert the `EXPR_RULE_TYPE` number shown in the Numeric Identification of EXPR_RULE_TYPE, page A-6 table for the variable `expr_rule_type`, and insert the `DEVL_PROJECT_ID` of the configuration model for the variable `devl_project_id` (see the Determine DEVL_PROJECT_ID of a Configuration Model, page A-3 example).

# Glossary

This glossary contains definitions relevant to working with Oracle Configurator.

**A**

### Archive Path

The ordered sequence of Configurator Extension Archives for a Model that determines which Java classes are loaded for Configurator Extensions and in what order.

**B**

### base node

The node in a Model that is associated with a Configurator Extension Rule. Used to determine the event scope for a Configurator Extension.

### batch validation

A background process for validating selections in a configuration.

### binding

Part of a Configurator Extension Rule that associates a specified event with a chosen method of a Java class. *See also* event.

### 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 (Assemble To Order) 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 options: true or false.

## C

### CDL (Constraint Definition Language)

A language for entering configuration rules as text rather than assembling them interactively in Oracle Configurator Developer. CDL can express more complex constraining relationships than interactively defined configuration rules can.

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

### CIO (Oracle Configuration Interface Object)

A server in the runtime application that creates and manages the interface between the client (usually a user interface) and the underlying representation of model structure and rules in the generated logic.

### command event

An event that is defined by a character string and detected by a command listener.

### 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 Features, Totals, Resources, Option counts, or numeric constants). The numeric values being compared can be computed or they can be discrete intervals in a continuous numeric input.

### Compatibility Rule

An Oracle Configurator Developer rule type that establishes a relationship among Features in the Model to control the allowable combinations of Options. *See also,* Property-based Compatibility Rule.

## Compatibility Table

A kind of Explicit Compatibility Rule. For example, a type of compatibility relationship where the allowable combination of Options are explicitly enumerated.

## component

A piece of something or a configurable element in a model such as a BOM Model, Model, or Component.

## Component

An element of the model structure, typically containing Features, that is configurable and instantiable. An Oracle Configurator Developer node type that represents a configurable element of a Model.

## Component Set

An element of the Model that contains a number of instantiated Components of the same type, where each Component of the set is independently configured.

## 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 model

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

## configuration rule

A Logic Rule, Compatibility Rule, Comparison Rule, Numeric Rule, Design Chart, Statement Rule, or Configurator Extension rule available in Oracle Configurator Developer for defining configurations. *See also* rules.

## configuration session

The time from launching or invoking to exiting Oracle Configurator, during which end users make selections to configure an orderable product. A configuration session is limited to one configuration model that is loaded when the session is initialized.

**configurator**

The part of an application that provides custom configuration capabilities. Commonly, a window that can be launched from a host application so end users can make selections resulting in valid configurations. *Compare* Oracle Configurator.

**Configurator Developer**

*See* OCD.

**Configurator Extension**

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

A type of configuration rule that associates a node, Java class, and event binding so that the rule operates when an event occurs during a configuration session.

A Java class that provides methods that can be used to perform configuration actions.

**Configurator Extension Archive**

An object in the Repository that stores one or more compiled Java classes that implement Configurator Extensions.

**connectivity**

The connection across components of a model that allows modeling such products as networks and material processing systems.

**Connector**

The node in the model structure that enables an end user at runtime to connect the Connector node's parent to a referenced Model.

**Constraint Definition Language**

*See* CDL

**Container Model**

A type of BOM Model that you import from Oracle Bills of Material into Oracle Configurator Developer to create configuration models that support connectivity and contain trackable components. Configurations created from Container Models can be tracked and updated in Oracle Install Base

**Contributes to**

A relation used to create a specific type of Numeric Rule that accumulates a total value. *See also* Total.

### Consumes from

A relation used to create a specific type of Numeric Rule that decrements a total value, such as specifying the quantity of a Resource used.

### count

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

### CZ

The product shortname for Oracle Configurator in Oracle Applications.

### CZ schema

The implementation version of the standard runtime Oracle Configurator data-warehousing schema that manages data for the configuration model. The implementation schema includes all the data required for the runtime system, as well as specific tables used during the construction of the configurator.

## D

### default

In a configuration, the automatic selection of an option based on the preselection rules or the selection of another option.

### Defaults relation

An Oracle Configurator Developer Logic Rule relation that determines the logic state of Features or Options in a default relation to other Features and Options. For example, if A Defaults B, and you select A, B becomes Logic True (selected) if it is available (not Logic False).

### Design Chart

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

## E

### 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.

### event

An action or condition that occurs in a configuration session and can be detected by a listener. Example events are a change in the value of a node, the creation of a component instance, or the saving of a configuration. The part of model structure inside which a listener listens for an event is called the event binding scope. The part of model structure that is the source of an event is called the event execution scope. *See also* command event.

### Excludes relation

An Oracle Configurator Developer Logic Rule type that determines the logic state of Features or Options in an excluding relation to other Features and Options. For example, if A Excludes B, and if you select A, B becomes Logic False, since it is not allowed when A is true (either User or Logic True). If you deselect A (set to User False), there is no effect on B, meaning it could be User or Logic True, User or Logic False, or Unknown. *See* Negates relation.

## F

### feature

A characteristic of something, or a configurable element of a component at runtime.

### Feature

An element of the model structure. Features can either have a value (numeric or Boolean) or enumerated Options.

## G

### generated logic

The compiled structure and rules of a configuration model that is loaded into memory on the Web server at configuration session initialization and used by the Oracle Configurator engine to validate runtime selections. The logic must be generated either in Oracle Configurator Developer or programmatically in order to access the configuration model at runtime.

### guided buying or selling

Needs assessment questions in the runtime UI to guide and facilitate the configuration process. Also, the model structure that defines these questions. Typically, guided selling questions trigger configuration rules that automatically select some product options and exclude others based on the end user's responses.

## H

**host application**

An application within which Oracle Configurator is embedded as integrated functionality, such as Order Management or *i*Store.

**I**

**implementer**

The person who uses Oracle Configurator Developer to build the model structure, rules, and UI customizations that make up a runtime Oracle Configurator. Commonly also responsible for enabling the integration of Oracle Configurator in a host application.

**Implies relation**

An Oracle Configurator Developer Logic Rule type that determines the logic state of Features or Options in an implied relation to other Features and Options. For example, if A Implies B, and you select A, B becomes Logic True. If you deselect A (set to User False), there is no effect on B, meaning it could be User or Logic True, User or Logic False, or Unknown. *See* Requires relation.

**import server**

A database instance that serves as a source of data for Oracle Configurator's Populate, Refresh, Migrate, and Synchronization concurrent processes. The import server is sometimes referred to as the remote server.

**initialization message**

The XML (Extensible Markup Language) message sent from a host application to the Oracle Configurator Servlet, containing data needed to initialize the runtime Oracle Configurator. *See also* termination message.

**instance**

A runtime occurrence of a component in a configuration that is determined by the component node's Instance attribute specifying a minimum and maximum value. *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.

**item**

A product or part of a product that is in inventory and can be delivered to customers.

### Item

A Model or part of a Model that is defined in the Item Master. Also data defined in Oracle Inventory.

### Item Master

Data stored to structure the Model. Data in the CZ schema Item Master is either entered manually in Oracle Configurator Developer or imported from Oracle Applications or a legacy system.

### Item Type

Data used to classify the Items in the Item Master. Item Catalogs imported from Oracle Inventory are Item Types in Oracle Configurator Developer.

## L

### listener

A class in the CIO that detects the occurrence of specified events in a configuration session.

### Logic Rule

An Oracle Configurator Developer rule type that expresses constraint among model elements in terms of logic relationships. Logic Rules directly or indirectly set the logical state (User or Logic True, User or Logic False, or Unknown) of Features and Options in the Model.

There are four primary Logic Rule relations: Implies, Requires, Excludes, and Negates. Each of these rules takes a list of Features or Options as operands. *See also* Implies relation, Requires relation, Excludes relation, and Negates relation.

## M

### 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

The entire hierarchical "tree" view of all the data required for configurations, including model structure, variables such as Resources and Totals, and elements in support of intermediary rules. Includes both imported BOM Models and Models created in Configurator Developer. May consist of BOM Option Classes and BOM Standard Items.

### model structure

Hierarchical "tree" view of data composed of elements (Models, Components, Features, Options, BOM Models, BOM Option Class nodes, BOM Standard Item nodes, Resources, and Totals). May include reusable components (References).

**N**

### Negates relation

A type of Oracle Configurator Developer Logic Rule type that determines the logic state of Features or Options in a negating relation to other Features and Options. For example, if one option in the relationship is selected, the other option must be Logic False (not selected). Similarly, if you deselect one option in the relationship, the other option must be Logic True (selected). *Compare* Excludes relation.

### node

The icon or location in a Model tree in Oracle Configurator Developer that represents a Component, Feature, Option or variable (Total or Resource), Connector, Reference, BOM Model, BOM Option Class node, or BOM Standard Item.

### Numeric Rule

An Oracle Configurator Developer rule type that expresses constraint among model elements in terms of numeric relationships. *See also*, Contributes to and Consumes from.

**O**

### object

Entities in Oracle Configurator Developer, such as Models, Usages, Properties, Effectivity Sets, UI Templates, and so on. *See also* element.

### OCD

*See* Oracle Configurator Developer.

### option

A logical selection made in the Model Debugger or a runtime Oracle Configurator by the end user or a rule when configuring a component.

### Option

An element of the Model. A choice for the value of an enumerated Feature.

### Oracle Configurator

The product consisting of development tools and runtime applications such as the CZ schema, Oracle Configurator Developer, and runtime Oracle Configurator. Also the runtime Oracle Configurator variously packaged for use in networked or Web deployments.

### Oracle Configurator Developer

The tool in the Oracle Configurator product used for constructing and maintaining configuration models.

### Oracle Configurator engine

The part of the Oracle Configurator product that uses configuration rules to validate runtime selections. Compare generated logic. *See also* generated logic.

### Oracle Configurator schema

*See* CZ schema.

### Oracle Configurator Servlet

A Java servlet that participates in rendering legacy user interfaces for Oracle Configurator.

### Oracle Configurator window

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

## P

### Populator

An entity in Oracle Configurator Developer that creates Component, Feature, and Option nodes from information in the Item Master.

### Property

A named value associated with a node in the Model or the Item Master. A set of Properties may be associated with an Item Type. After importing a BOM Model, Oracle Inventory Catalog Descriptive Elements are Properties in Oracle Configurator Developer.

### Property-based Compatibility Rule

An Oracle Configurator Developer Compatibility Rule type that expresses a kind of compatibility relationship where the allowable combinations of Options are specified implicitly by relationships among Property values of the Options.

### publication

A unique deployment of a configuration model (and optionally a user interface) that enables a developer to control its availability from host applications such as Oracle Order Management or *i*Store. Multiple publications can exist for the same configuration model, but each publication corresponds to only one Model and User Interface.

### publishing

The process of creating a publication record in Oracle Configurator Developer, which includes specifying applicability parameters to control runtime availability and running an Oracle Applications concurrent process to copy data to a specific database.

## R

### reference

The ability to reuse an existing Model or Component within the structure of another Model (for example, as a subassembly).

### Reference

An Oracle Configurator Developer node type that denotes a reference to another Model.

### Repository

Set of pages in Oracle Configurator Developer that contains areas for organizing and maintaining Models and shared objects in a single location.

### Requires relation

An Oracle Configurator Developer Logic Rule relationship that determines the logic state of Features or Options in a requirement relation to other Features and Options. For example, if A Requires B, and if you select A, B is set to Logic True (selected). Similarly, if you deselect A, B is set to Logic False (deselected). *See* Implies relation.

### Resource

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.

### rules

Also called business rules or configuration rules. In the context of Oracle Configurator and CDL, a rule is not a business rule. Constraints applied among elements of the product to ensure that defined relationships are preserved during configuration. Elements of the product are Components, Features, and Options. Rules express logic, numeric parameters, implicit compatibility, or explicit compatibility. Rules provide preselection and validation capability in Oracle Configurator.

*See also* Comparison Rule, Compatibility Rule, Design Chart, Logic Rule and Numeric Rule.

### runtime

The environment in which an implementer (tester), end user, or customer configures a product whose model was developed in Oracle Configurator Developer. *See also* configuration session.

## S

### Statement Rule

An Oracle Configurator Developer rule type defined by using the Oracle Configurator Constraint Definition Language (text) rather than interactively assembling the rule's elements.

## T

### termination message

The XML (Extensible Markup Language) message sent from the Oracle Configurator Servlet to a host application after a configuration session, containing configuration outputs. *See also* initialization message.

### Total

A variable in the Model used to accumulate a numeric total, such as total price or total weight.

Also a specific node type in Oracle Configurator Developer. *See also* node.

## U

### UI

*See* User Interface.

### UI Templates

Templates available in Oracle Configurator Developer for specifying UI definitions.

### Unknown

The logic state that is neither true nor false, but unknown at the time a configuration session begins or when a Logic Rule is executed. This logic state is also referred to as Available, especially when considered from the point of view of the runtime Oracle Configurator end user.

### 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 visible part of the application, including menus, dialog boxes, and other on-screen elements. The part of a system where the user interacts with the software. Not necessarily generated in Oracle Configurator Developer. *See also* User Interface.

### User Interface

The part of an Oracle Configurator implementation that provides the graphical views necessary to create configurations interactively. A user interface is generated from the model structure. It interacts with the model definition and the generated logic to give end users access to customer requirements gathering, product selection, and any extensions that may have been implemented. *See also* UI Templates.

## V

### validation

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

## W

### Workbench

Set of pages in Oracle Configurator Developer for creating, editing, and working with Repository objects such as Models and UI Templates.

# Index