ACF Customization Guide

Release 5.2.1

Model N

Document Revised: June 12, 2007

Table of Contents

List of Code Samples

List of Figures and Tables

1 About This Book

The *ACF Customization Guide* contains background information, programming tips, and simple scenarios for customizing and extending the Application Component Framework (ACF). The ACF is the user interface framework which can be customized with XML and Java to meet the needs of the customer.

This guide provides step-by-step instructions for common customization scenarios. Also included are code listings and other important reference information.

1.0 Audience

This book is intended for the following audiences:

- System Integrator partners and IT Programmers who customize and maintain the Model N applications at a customer's site
- Model N Professional Services representatives who are planning the implementation and customization of a Model N application.

2.0 Using This Guide

This guide is a starting point for performing simple user interface customizations at the customer's site. The ability to write Java and XML code is required to perform these customizations.

This guide assumes the customizer has access to the source code for a customer's Model N system and has the ability to both build and deploy software.

3.0 Conventions

The following table describes the typographical and other conventions used in this book:

Table 1-1: Conventions

Conventions	Descriptions
File names and code objects	Names of files and code objects are in `Courier New` font style.
Field and dialog box names	Names of fields and dialog boxes are first letter (of each word) capitalized, with no other special formatting. For example: • Information in the Customer ID field must be valid. • Information in the External Data dialog box is read-only.
Line breaks in code	Some code examples may contain lines which are longer than can be displayed on the width of a page. Lines which have been broken for page formatting reasons will be indicated with the pilcrow (¶) symbol. **Warning:** Do not separate these lines during actual configuration because doing so may cause the application to fail to initialize correctly.
System outputs	System outputs, such as confirmation messages or alerts, are first letter capitalized. For example: The system displays the following message: Do you want to save your changes before moving on?
Referenced topics	Referenced topic headings are in blue and underlined. For example: See Conventions for more information.
Lifecycle statuses	The lifecycle statuses are in `Courier New` font style. For example: Users cannot override the `Ignore` status or the `Fatal` severity level.

Table 1-1: Conventions (Continued)

Conventions	Descriptions
Notes	Notes are contained within two horizontal lines. For example: **Note:** This is a note.
Blank pages	There will be a blank page with header and footer at the end of the chapters ending on odd page numbers. This format is designed to accommodate printing multiple chapters on duplex paper and keeping the collation even.

4.0 Related Documents/Resources

The following related documents are available:

- The *Installation Guide* for each version describes how to install/deploy the product.
- The *Developer's Guides* for each application contains configuration information and Java extensions.
- The *Application Switches Reference* for each verison describes the application switches which are available to configure the product.

5.0 Feedback

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

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

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

- E-mail: technicaldocs@modeln.com
- FAX: (650) 610 - 4699. Attn: Product Documentation
- Postal service:

  Model N
  Product Documentation
  1800 Bridge Parkway
  Redwood City, CA 94065

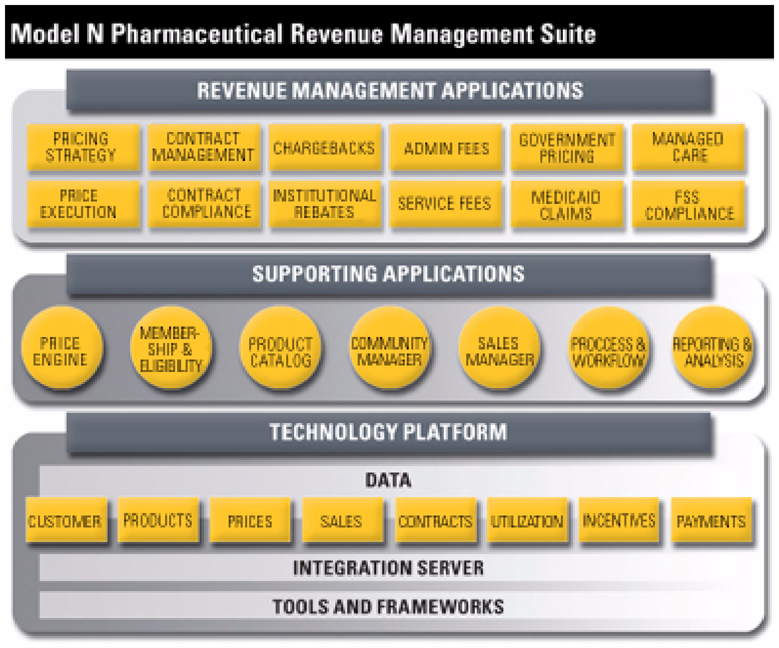
At Model N, we are dedicated to improving the documentation. We are not able to reply to your comments individually, but we will use your advice to improve our services.

2 Architecture Overview

This chapter provides a high level overview of the architecture of Model N applications, together with a description of the Application Component Framework. The Model N family of Revenue Management products are browser-based applications built upon a cohesive layered architecture. Model N is an n-tier, J2EE standards-based product.

1.0 Model N Architecture and the Application Component Framework

The following diagram illustrates the layers of the Model N application:



The Application Component Framework (ACF) is a powerful, flexible, extensible and cohesive toolset residing in the Technology Platform, Tools and Frameworks layer. It allows the rapid creation of consistent n-tier thin-client user interfaces. This framework allows Model N applications to be developed from reusable visual components and follows the well-accepted Model-View-Control (MVC) paradigm.

2.0 Understanding Components

This section provides an introduction to the architecture of the ACF and the structure of components. It also provides a brief description of common components.

2.1 Component Architecture

The Application Component Framework (ACF) is a user interface framework containing reusable, extendable components. The ACF provides a consistent and reusable component library, with a published API.

The ACF consists of two distinct parts, as follows:

- The underlying runtime framework that enables the definition and instantiation of components in general.
- The library of components that accelerates application development and user interface customization.

The runtime framework is implemented entirely in Java and runs on the server side to generate the JSP that in turn produces the HTML/DHTML pages that are rendered in a standard Web browser. Like other visual component libraries, such as Java Swing, individual ACF components are assembled in a hierarchical manner to construct a user interface. The top-level containers, intermediate containers, and atomic components are nested to form a component hierarchy that allows greater structure and re-usability within the application interface. The component framework API provides a published set of methods and properties that form the foundation of a consistent and reusable component library.

The library of reusable user interface components range from simple widgets (such as buttons, labels, textboxes, and selection lists), to more complex components (such as tables, trees, and dialogs) and up to sophisticated components (such as an entity list, folder/finder and detail components). The ACF components support a mapping to backing data, allowing customizable formatting and validation through XML configuration files. The XML configuration files provide the initialization states of the components.

Customizations and extensions to the components are accomplished using the XML configuration files, allowing for a full Web application to be developed without writing any HTML or JSP code. Application UI changes such as changes to layout, color, and display fields occur through simple XML and CSS modifications.

The framework allows for heavier customization where the behavior of existing components, at a varying level of granularity, can be changed by writing Java code. In the same manner, new components may be developed to add new and create new applications with a consistent user interface design.

2.2 Component Hierarchy

The Model N application hierarchy consists of a set of nested containers, as follows:

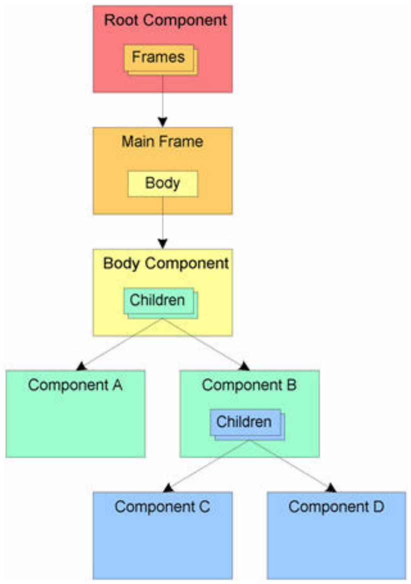
- Top-level containers
- Intermediate containers
- Atomic components

Top-level containers are the starting point for constructing an application interface. Intermediate containers and atomic components are placed in the top-level container, where they can be loaded and rendered to form a unified application. Intermediate containers allow logical component groups to be created and managed as a collective unit. In this way, all components within the container can be accessed as a single unit or contained with other logically related components. Atomic components form the building blocks of the application. These user interface elements have published API's that define their use and functionality.

The container hierarchy refers to the component relationship structure used within the ACF. Components are organized in a hierarchical structure where the root component becomes the top-level container to which all other components are added.

The term container refers to any non-interactive component which is used to group and visually display related components. `CMnContainerComp` is an example of a component which is used to group other components. These container components can exist at any level of the component hierarchy. Widgets are interactive components that form the leaf nodes of the component hierarchy. These components are atomic visual elements which cannot act as containers for other components. A simplified diagram of the component hierarchy is shown in Figure 2-1.

Figure 2-1: Component Hierarchy

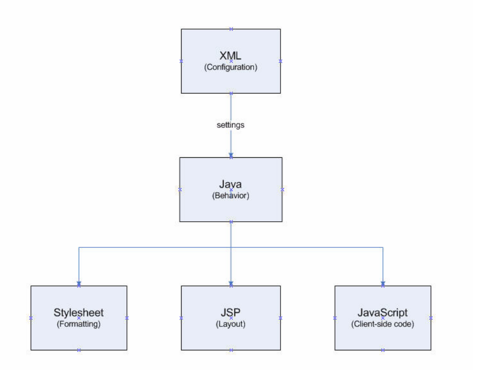


The hierarchical component structure provides a mechanism for maintaining parent-child relationships between containers and components.

2.3 Component Structure

Components are composed of three parts: a backing Java class, an XML configuration file, and a JSP file. Optional JavaScript files may be included to add functionality, and Stylesheets may be added for additional formatting.

Figure 2-2: Component Structure



2.3.1 Component Files

Several files are associated with a component. A Java class is required to provide functionality. A JSP file is required to format, render, and display the component in the browser. An XML file is recommended to configure the Java class attributes. JavaScript and CSS files are optional for additional support if needed. Table 2-1 provides a list of the files, a brief description, and the location of the file.

Table 2-1: Component Files

Sample File	Definition	Location	Customized?
`CMnCheckBoxComp.java`	Recommended (required if no XML); the backing Java file provides the functionality	classpath	Typically
`CMnCheckBoxComp.xml`	Recommended (required if no Java); the XML file provides an extendable configuration method	classpath	Typically
`CMnCheckBoxCompJsp.jsp`	Optional; the JSP file renders the component in the browser	docroot	Rarely
`CMnCheckBoxComp.css`	Optional; a stylesheet may be used to configure the appearance of the page	docroot	Infrequently
`CMnCheckBoxComp.js`	Optional; JavaScript may be used to add post-rendering functionality such as validations.	docroot	Infrequently

The component framework manages the loading and configuration of components, and therefore can locate the correct class files and XML configuration files by searching the classpath. The remaining support files must be accessible to the application server or Web server, and must therefore be located in a document root directory which is accessible to incoming HTTP requests. The directories in which the files reside are expected to be identical. For example, if a component is `com.foo.CMnMyComp`, then all files are expected to be in the `com/foo` directory.

2.3.2 Java Class

At the core of every component is a Java Bean which defines the component properties and behavior. The Java class contains attributes such as the component name or display value. Methods within the Java class provide support for managing properties, dispatching events and errors, and ultimately rendering the component.

XML is used as the extension mechanism, to provide the ability to configure component behavior without requiring Java coding.

2.3.3 XML Configuration File

The XML file provides the means for dynamically configuring the component through the published get and set methods provided by the corresponding Java class.

XML files are used to configure the ACF components by setting component properties. These properties control the behavior of the component and its relationship with other components. The application then uses these XML files to construct the component hierarchy and set the properties of each component. In most cases, creating or modifying a component configuration XML file is sufficient to create the desired behavior.

In addition to setting properties, you can register events, add child components, and apply access controls through the XML file. The XML structure is also expressive enough to create arbitrary types. The arbitrary types must be Java Beans.

2.3.4 JSP File

The specification for how a component is rendered is be provided by a JSP file. Each component will have a corresponding JSP page or some other view, defined explicitly or through inheritance, which provides the HTML tags and dynamic content used to render the component.

The rendered view is usually implemented as a JSP, but can take other forms. It is leveraged by using the `javax.servlet.RequestDispatcher.include()` mechanism. As long as the view is accessible through a URI, it can be displayed.

3.0 Types of Components

The ACF component library is a collection of visual and non-visual components, containers, and layout managers. These components include simple widgets such as buttons and text boxes, as well as more complex components such as tables, trees, details, and folder-finders. The layout managers are containers that organize the other components on the Web page.

The base class for all visual components is `com.modeln.ui`. Layouts are contained in `com.modeln.ui.layout`.

This section provides an overview for each the following types of components:

- Layouts and Sections
- Simple Widgets, for example, Input Fields, Labels, Request/Action, and Selection components
- Complex components, including Trees, Tables, Dialogs, Details, and Folder-Finders

See *Appendix A – Component Reference* for a description of specific components.

See Component Reference for a description of specific components.

3.1 Layouts

Layouts are a mechanism to maintain the position of components within the user interface. Layouts direct the ACF to automatically display the child components correctly rather than creating an explicit layout in the JSP file.

Every component contains a primary layout that determines how the component and its children will be rendered in relation to one another. A component may also contain other layouts, called sections, that act upon an explicitly defined subset of the component's children.

The display order is specified in the layout, which determines the position of the user interface element on the screen.

The layout components are defined in the `com.modeln.ui.layout` package.

3.2 Simple Widgets

Simple widgets are basic, atomic components that are combined to create a page.

3.2.1 Input Fields

Input fields are used to display text or receive text from an input form. With the exception of `CMnTextAreaComp`, input fields have a naming convention of `...BoxComp`. Input field components extend from `CMnBaseTextInputComp`.

Following is a list of the input field widgets for each type of data:

- Strings: `CMnTextBoxComp, CMnTextAreaComp`
- Numbers: `CMnBDPercentBoxComp, CMnBigDecimalBoxComp, CMnDoubleBoxComp, CMnFloatBoxComp, CMnIntegerBoxComp, CMnLongBoxComp, CMnPercentBoxComp, CMnShortBoxComp`
- Dates: `CMnDateBoxComp, CMnDateTimeBoxComp, CMnDateRangeComp, CMnEffectiveStartDateBoxComp, CMnEffectiveStartDateTimeBoxComp, CMnEffectiveEndDateBoxComp, CMnEffectiveEndDateTimeBoxComp`
- Currency: `CMnSingleCurrencyMoneyBoxComp`

3.2.2 Labels

Labels are the display fields for a component. Label components extend from `CMnBasetTextComp`.

Following is a list of label widgets for each type of data:

- Strings: `CMnTextFieldComp, CMnEnumFieldComp`
- Numbers: `CMnBDPercentFieldComp, CMnBigDecimalFieldComp, CMnDoubleFieldComp, CMnFloatFieldComp, CMnIntegerFieldComp, CMnLongFieldComp, CMnPercentFieldComp, CMnShortFieldComp`
- Dates: `CMnDateFieldComp, CMnDateTimeFieldComp, CMnEffectiveStartDateFieldComp, CMnEffectiveStartDateTimeFieldComp, CMnEffectiveEndDateFieldComp, CMnEffectiveEndDateTimeFieldComp`
- Currency: `CMnSingleCurrencyMoneyFieldComp`

3.2.3 Request/Action Components

Request components initiate a user action or request. Request components may also contain a confirmation component to force the user to confirm the action. Request components extend `CMnBaseRequestComp`.

Following is a list of Request/Action components:

- `CMnButtonComp`

- `CMnAnchorComp`
- `CMnFgoAnchorComp`

3.2.4 Selection Components

Selection components provide the user with a list of items for selection within a form. The data for each component is managed by a list selection model, which maintains a name/value relationship for each item in the list. All list selection models extend from the `CMnBaseListSelModel` class, which encapsulates both the data list that is submitted to the application and the selection list that is visible to user.

Following is the list of selection components:

- `CMnDropDownSelComp`
- `CMnCheckBoxSelComp`
- `CMnRadioButtonSelComp`
- `CMnAddRemoveSelComp`

3.2.5 Miscellaneous Components

Miscellaneous components include components for uploading and downloading files, and the component for inserting an image. File components initiate file transfers to or from the server. There is an upload component to transfer files to the server and a download component to transfer files from the server. These components generally rely on the browser for selection of files and directories.

- `CMnImageComp`
- `CMnFileUploadComp`
- `CMnFileDownloadComp`
- `CMnJspDownloadComp`

3.3 Complex Components

A complex component is a collection of atomic components that work together as a single entity to provide more complex functionality. Examples of complex components are trees, dialogs, tables, details, and folder-finders.

3 Programming Tips

One of the goals of the structured, flexible design of the ACF and the Model N architecture in general is to allow for ease of customization. The ACF can be extended using XML or a combination of XML and Java. Many customizations can be performed using XML entirely, since much of the underlying architecture is wrapped in ready-to-use components.

In addition, the configuration of the applications can be controlled using property files and application switches.

1.0 Types of Customizations

The Model N architecture offers three levels of customization. The levels are determined by the amount of coding effort required to complete the customization.

- Configuration involves loading content, setting application switches, and entering values in property files to configure the business logic.
- Extension includes modifying XML files to customize application behavior and writing short Java plug-ins to add functionality.
- Customization includes developing new components, business objects, and

application logic using Java

Note: This guide covers extensions using XML overlays and light Java customizations. For information on configuration, refer to the developer guide for a module. For example, the *Contract Management Developer's Guide* contains configuration instructions for contracts.

1.1 Property File and Application Switch Configuration

Configuration is a level of customization that requires no coding.

It includes such configuration changes as the following:

- Updating property files
- Setting application switches
- Loading content
- Updating lookup lists (for example, reason codes)
- Configuring notifications, alerts, and access control rules
- Other application settings

1.2 XML Extensions and Java Plug-ins

Extensions are user interface customizations requiring minimal coding. Extensions primarily use XML in the form of component overlays. Extensions may include some Java and/or Java plug-ins.

Component overlays are XML files used to customize the ACF by setting component properties. These properties control the behavior of the component and its relationship with other components. The application then uses these XML files to construct the component hierarchy and set the properties of each component. In most cases, creating or modifying a component configuration XML file is sufficient to create the desired behavior.

Further customization can be accomplished using Java plug-ins. Java plug-ins allow components to be customized in Java. The most common use for a Java plug-in is to configure properties. This is done by specifying the plug-in through either the component XML configuration or the component plugin API.

1.3 Application Customizations

Application customizations require more significant Java coding effort than extensions and may also require tuning and optimization. Adding new pages, new POF objects, editing queries, and adding new application functionality are examples of these high level customizations.

2.0 Introduction to XML Overlays

In the Model N component architecture, XML files configure component behavior by setting component attributes. The component behavior can be customized using XML files to extend a component's original behavior. XML overlays are the primary method for customizing Model N applications.

XML overlays are XML files that extend, rather than replace, current Model N functionality. XML overlays are used for customizations to the user interface, such as changing labels of existing fields, adding new fields, adding search parameters, and adding columns to tables.

The most common reason to overlay a component is to add XML to it, although you can customize existing XML as well. The XML overlay represents any changes made to the component relative to the original XML component file.

In an overlay, only enough information is required to locate what is being changed. Thus, for example, the type attribute is often not specified in an overlay because the value in the base component is maintained.

An ACF component may be extended with multiple XML overlays. The overlay contains a `configures` statement pointing to the ACF component being extended. The overlays are registered in a property file, so that the XML can be merged into a single component tree. If more than one overlay is registered, then each overlay is applied in the order they appear in the property files. However, in all cases, the overlay specifies that it is extending the original ACF component.

When a component is customized with an overlay, the new component maintains the backing Java and JSP of the original component.

Important: Overlays always specify an ACF component in the `configures` statement. It is not valid for an overlay to extend another overlay.

See Customization Scenarios for step-by-step instructions for common extension scenarios. This chapter provides general instructions for creating an XML overlay.

2.1 Sample XML Overlays

This section provides an overview of the method for creating overlays. Examples are provided for both a single overlay and for multiple overlays.

Detailed examples for typical customizations are available in Customization Scenarios.

2.1.1 Create a Single Overlay

The following example is an overlay to extend the fictitious `LoginComp` component to add a Create Account link:

Code 3-1: Sample Overlay to Add an Anchor

```
<component name="UpgradedLoginComp" configures="LoginComp">

  <child name="CreateAccount" component="Anchor" order="30">

    <string name="value">Create Account</string>

  </child>
```

Code 3-1: Sample Overlay to Add an Anchor

```
</component>
```

An XML overlay creates a new component that configures an existing component. The original component is loaded first and then the attributes of the overlay are loaded (overlaid) on top of the original component.

In this example, the name of the new component is `UpgradedLoginComp`. The existing component being extended is `LoginComp`. The `CreateAccount` component will be added to the `LoginComp` component as an `Anchor` with the label `Create Account`.

The ACF knows how to combine the XML for `UpgradedLoginComp` with the XML for `LoginComp` (and other ancestors) to create a single object tree which it then uses to create the Login component. This provides the capability to overlay properties already defined in ancestor XML files.

In an XML overlay, information is provided to locate what is being changed. Attributes that are not specified will remain as they are in the base component (because they are not being overlaid). All properties are included in the overlay, but only those with attributes will be customized.

Any ACF component may be overlaid with any number of overlays. Each overlay is simply an XML extension of the component. In other words, the overlay itself is an XML document that provides the incremental changes over the original component. If more than one overlay is registered, then each overlay is applied in order. This means that overlays can depend on each other and extend previous extensions. However, in ALL cases, the configures attribute of the overlay specifies that it is extending the original ACF component.

The overlay mechanism requires that the customizer (a) provide the XML overlay document and (b) register the overlay, so that the system knows that the specific overlay exists.

2.1.2 Register the Overlay

Each overlay requires an entry in a properties file to alert the ACF that a component is being overlaid. The properties file builds up a list of overlays for each component type using the property prefix `com.modeln.App.componentOverlays`.

The form of the property file entry is: `com.modeln.App.componentOverlays.<existing.component>+=overlay.component>`. The += syntax concatenates multiple attributes.

Following is an example of the property to register the XML overlay for Sample Overlay to Add an Anchor:

```
com.modeln.App.componentOverlays.LoginComp+=\

#This line is the original component.

com.model.App.MyProject.UpgradedLoginComp

#This line is the customized component.
```

2.1.3 Create Multiple Overlays

There may be more than one XML overlay for a component. The configures statements must all point to the base component. The configures statement of an overlay may not point to another overlay. The overlays will be registered in separate property files.

A good example of multiple overlays is the Contracts module in the Life Sciences application. Multiple packages overlay the Contracts folder-finder component to add subfolders specific to that package. The following examples illustrate multiple overlays.

Following is the XML configuration file for the initial component, component A, which will be overlaid with multiple XML overlays:

Code 3-2: Initial Component for Multiple Overlays Example

```
<component name="A">

  <property name="folders">

    <property name="contracts" type="folder">

     . . .

    </property>

  </property>

</component>
```

Following is an example of a possible first XML file that overlays the folders property to create the Rebates folder:

Code 3-3: First XML Overlay for Multiple Overlays Example

```
<component name="B" configures="A">

  <property name="folders">

    <property name="rebates" type="folder">

     . . .

    </property>

  </property>

</component>
```

Following is an example of a possible second XML file that overlays the folders property to create the My Contracts folder:

Code 3-4: Second XML Overlay for Multiple Overlays Example

```
<component name="C" configures="A">

  <property name="folders">

    <property name="contracts">

      <!-- Can add XML at arbitrary points -->

      <property name="myContracts" type="folder">
```

Code 3-4: Second XML Overlay for Multiple Overlays Example

```
       </property>

      . . .

     </property>

   </property>

</component>
```

2.1.4 Register the Overlay

To register multiple overlays, you would probably create two overlay properties in two different properties files. Following are examples of overlay properties:

```
com.modeln.App.componentOverlays.A+=B

com.modeln.App.componentOverlays.A+=C
```

When the ACF constructs a component of type A, it will obtain the `com.modeln.App.componentOverlays.A` properties and combine A's XML tree with B and C before creating the component instance.

Merge order for multiple overlays is taken from order in properties file.

2.2 Rules for Troubleshooting Overlays

Following are some rules that may help when troubleshooting overlays:

- Overlays must configure a component; you cannot have an overlay of an overlay.
- Overlaid components cannot be dynamically instanced (for example, through `addChildOfType`).
- XML errors in overlays or overlaid components can get masked by stack traces. If you have a funny stack trace on an overlaid component, try deleting everything inside the `<component>` tags in each overlay until you isolate the overlay that is an issue. Then delete segments of the XML until you find the offending segment.

3.0 Developer Tools

When the Model N application is running in Development Mode, an interactive tool called the Component Viewer displays information about the code behind every component. You can view a summary of the Java and XML for a component, view a list of properties and their values, view the Java class hierarchy and other information by simply pressing CTL/Right Click.

3.1 Activating Development Mode

For internal users, Development Mode is automatically enabled when running the application from the Verification Environment.

Follow these steps to activate Development Mode from an installed version of the application:

1. Locate the `/users/<USER_NAME>/classes/<USER_NAME>_base_en_US.properties` file. The `/users` directory is located in the root directory of the application.
2. Set the `com.modeln.App.development` property to **true**.
3. Restart the application. Development Mode will be activated and you can access the Component Viewer.

Note: The Component Viewer is an in-house development tool, meant for internal use only. It is not a supported commercial product.

3.2 Using the Component Viewer

The Componenet Viewer is a development tool that displays the code and properties behind a component.

Follow these steps to display the code and properties for a component:

1. Place the mouse over the component. You can select a single field, a search builder component, or an entire table.
2. Press Ctrl/Right Click simultaneously.

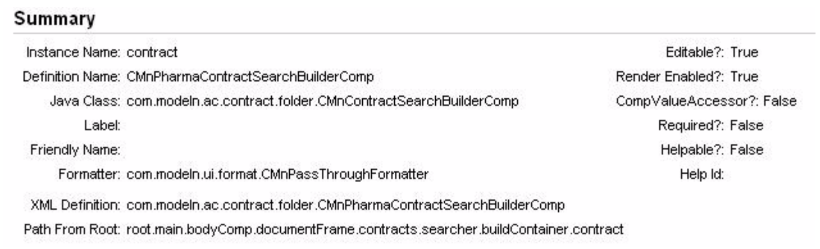
   The chosen component will be outlined in red and the information about the component is displayed.
3. Select the code you want to view in the left navigation menu: Summary, Mapping, Validation, Layout, JavaScript, CSS, Properties, Java Classes, Ancestors and Children.

3.2.1 Component Viewer Examples

This section provides samples of the information available in the Component Viewer. These selections are available in the left navigation bar.

Summary

The Summary provides the name of Java class and XML definition.

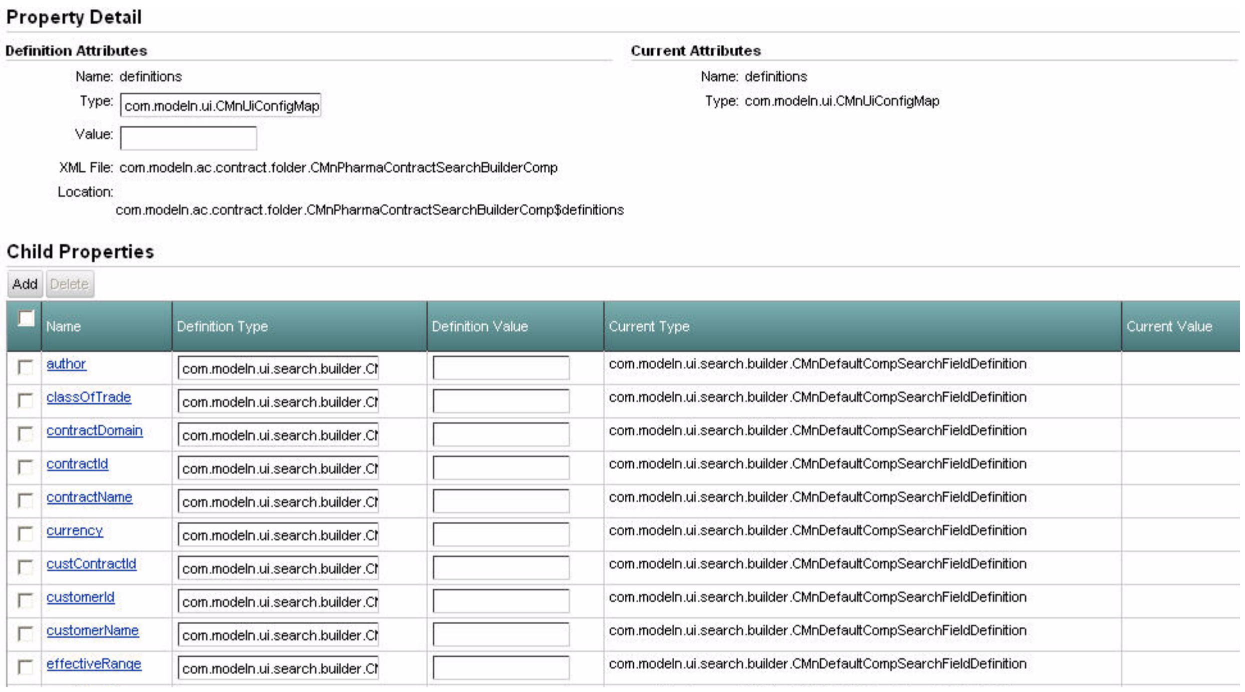


Summary

Instance Name:	contract	Editable?:	True
Definition Name:	CMnPharmaContractSearchBuilderComp	Render Enabled?:	True
Java Class:	com.modeln.ac.contract.folder.CMnContractSearchBuilderComp	CompValueAccessor?:	False
Label:		Required?:	False
Friendly Name:		Helpable?:	False
Formatter:	com.modeln.ui.format.CMnPassThroughFormatter	Help Id:	
XML Definition:	com.modeln.ac.contract.folder.CMnPharmaContractSearchBuilderComp		
Path From Root:	root.main.bodyComp.documentFrame.contracts.searcher.buildContainer.contract		

Field definitions

The Field Definitions property provides access to details about the fields on the page.



Property Detail

Definition Attributes

Name: definitions
Type: com.modeln.ui.CMnUiConfigMap
Value:
XML File: com.modeln.ac.contract.folder.CMnPharmaContractSearchBuilderComp
Location: com.modeln.ac.contract.folder.CMnPharmaContractSearchBuilderComp$definitions

Current Attributes

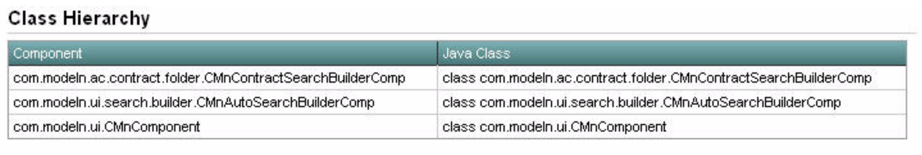
Name: definitions
Type: com.modeln.ui.CMnUiConfigMap

Child Properties

	Name	Definition Type	Definition Value	Current Type	Current Value
☐	author	com.modeln.ui.search.builder.CI		com.modeln.ui.search.builder.CMnDefaultCompSearchFieldDefinition	
☐	classOfTrade	com.modeln.ui.search.builder.CI		com.modeln.ui.search.builder.CMnDefaultCompSearchFieldDefinition	
☐	contractDomain	com.modeln.ui.search.builder.CI		com.modeln.ui.search.builder.CMnDefaultCompSearchFieldDefinition	
☐	contractId	com.modeln.ui.search.builder.CI		com.modeln.ui.search.builder.CMnDefaultCompSearchFieldDefinition	
☐	contractName	com.modeln.ui.search.builder.CI		com.modeln.ui.search.builder.CMnDefaultCompSearchFieldDefinition	
☐	currency	com.modeln.ui.search.builder.CI		com.modeln.ui.search.builder.CMnDefaultCompSearchFieldDefinition	
☐	custContractId	com.modeln.ui.search.builder.CI		com.modeln.ui.search.builder.CMnDefaultCompSearchFieldDefinition	
☐	customerId	com.modeln.ui.search.builder.CI		com.modeln.ui.search.builder.CMnDefaultCompSearchFieldDefinition	
☐	customerName	com.modeln.ui.search.builder.CI		com.modeln.ui.search.builder.CMnDefaultCompSearchFieldDefinition	
☐	effectiveRange	com.modeln.ui.search.builder.CI		com.modeln.ui.search.builder.CMnDefaultCompSearchFieldDefinition	

Java classes

The Java classes link displays the Java class hierarchy.



Class Hierarchy

Component	Java Class
com.modeln.ac.contract.folder.CMnContractSearchBuilderComp	class com.modeln.ac.contract.folder.CMnContractSearchBuilderComp
com.modeln.ui.search.builder.CMnAutoSearchBuilderComp	class com.modeln.ui.search.builder.CMnAutoSearchBuilderComp
com.modeln.ui.CMnComponent	class com.modeln.ui.CMnComponent

4 Customization Scenarios

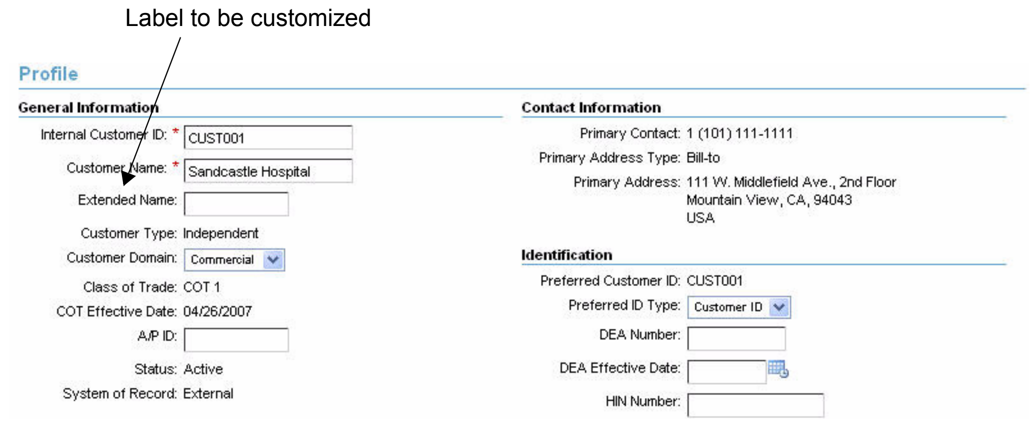
This chapter contains instructions and code examples for common user interface customizations that are performed by extending components using XML.

1.0 Customizing Labels

A simple extension is to change the labels on a page, using an XML overlay.

The goal of this extension scenario is to change the label of the Extended Name: field on the Customer Profile page. An example of the page is shown in Table 4-1:

Figure 4-1: Customer Profile Page Before Label Customization



Profile

General Information

Internal Customer ID: *	CUST001
Customer Name: *	Sandcastle Hospital
Extended Name:	
Customer Type:	Independent
Customer Domain:	Commercial
Class of Trade:	COT 1
COT Effective Date:	04/26/2007
A/P ID:	
Status:	Active
System of Record:	External

Contact Information

Primary Contact:	1 (101) 111-1111
Primary Address Type:	Bill-to
Primary Address:	111 W. Middlefield Ave., 2nd Floor Mountain View, CA, 94043 USA

Identification

Preferred Customer ID:	CUST001
Preferred ID Type:	Customer ID
DEA Number:	
DEA Effective Date:	
HIN Number:	

To change the label of the component, create an XML file to overlay the current component and then create a property file to register the custom component.

This scenario applies to any label attribute.

1.1 Create an XML Overlay

Following is an annotated example of the XML overlay for customizing the Extended Name: label on the Customer Profile page.

The XML overlay file will contain only the new value for the label attribute. All other attributes are loaded from the component that is being extended.

Code 4-1: XML Overlay to Change a Label

```
<component
name="com.sample.ac.cmty.customers.CSampleLabelCustomerProfileComp"¶
configures="com.modeln.ac.cmty.customers.CMnCustomerProfileComp">
```

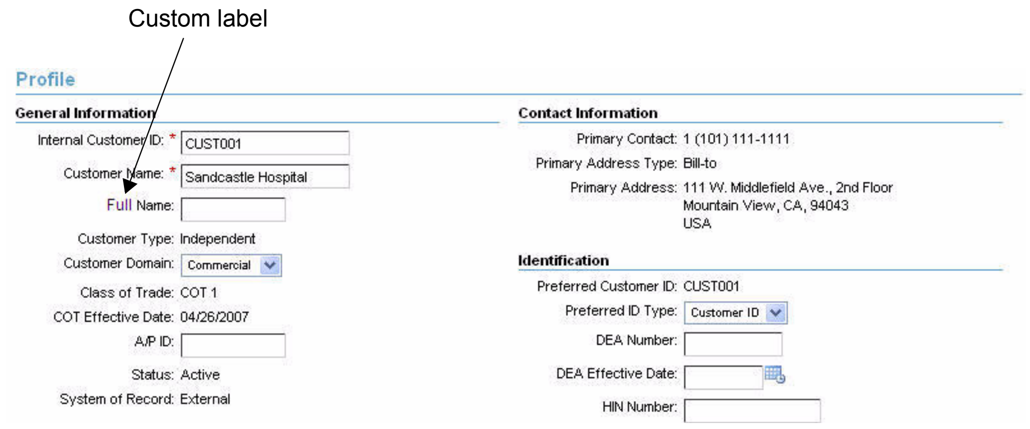
The `component name` is the name of the new component containing the customizations. The `configures` statement points to the existing component that is being customized.

```
- <child name="customerExtendedName">

          <string name="label">Full Name:</string>

        </child>

</component>
```

The attribute `customerExtendedName` indicates the component to be overlaid. The string `Full Name:` is the customized label for the component. This value will overlay the label of the `customerExtendedName` field in the `com.modeln.ac.cmty.customers.CMnCustomerProfileComp` component.

The new label on the Customer Profile page is similar to the example shown in Table 4-2:

Figure 4-2: Customer Profile Page after Label Customization



Profile

General Information

Internal Customer ID: *	CUST001
Customer Name: *	Sandcastle Hospital
Full Name:	
Customer Type:	Independent
Customer Domain:	Commercial
Class of Trade:	COT 1
COT Effective Date:	04/26/2007
A/P ID:	
Status:	Active
System of Record:	External

Contact Information

Primary Contact:	1 (101) 111-1111
Primary Address Type:	Bill-to
Primary Address:	111 W. Middlefield Ave., 2nd Floor Mountain View, CA, 94043 USA

Identification

Preferred Customer ID:	CUST001
Preferred ID Type:	Customer ID
DEA Number:	
DEA Effective Date:	
HIN Number:	

1.2 Register the Overlay

The next step is to register the overlay in a property file so that the application can use the new file. Create an entry similar to the following one in the property file used for registering the overlays for the project, for example, `CustomerProfile_compOverlays.properties`:

Code 4-2: Property for Registering the Label Overlay

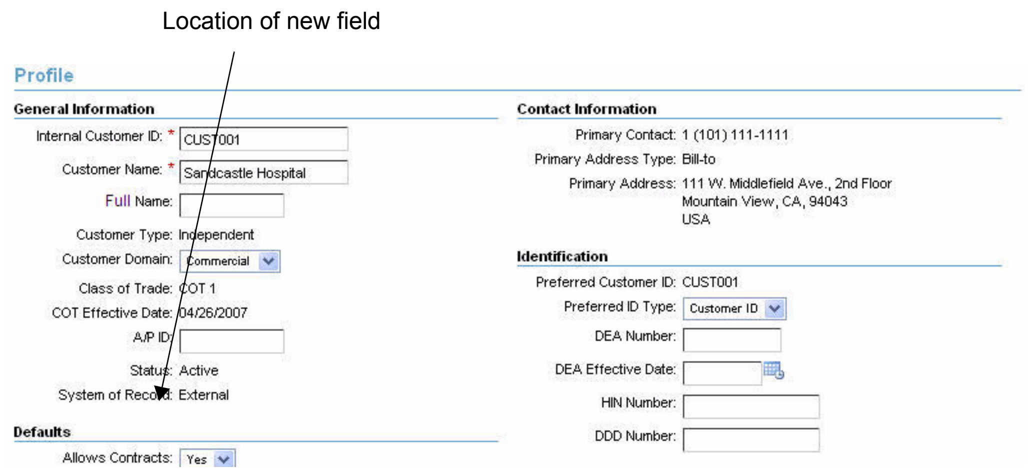
```
com.modeln.App.componentOverlays.com.modeln.ac.cmty.customers.¶
CMnCustomerProfileComp+=¶
\com.sample.ac.cmty.customers.CSampleLabelCustomerProfileComp
```

2.0 Adding a New Field to a Component

Another common extension performed is adding a new field to a page. Fields that currently exist in the database can be added to a component using an XML overlay.

The goal of this extension scenario is to add a field named State Tax ID: after the System of Record: field to the Customer Profile page.

Figure 4-3: Customer Profile Page Before Adding a New Field



Profile

General Information

Internal Customer ID: *	CUST001
Customer Name: *	Sandcastle Hospital
Full Name:	
Customer Type:	Independent
Customer Domain:	Commercial
Class of Trade:	COT 1
COT Effective Date:	04/26/2007
A/P ID:	
Status:	Active
System of Record:	External

Defaults

Allows Contracts:	Yes

Contact Information

Primary Contact:	1 (101) 111-1111
Primary Address Type:	Bill-to
Primary Address:	111 W. Middlefield Ave., 2nd Floor Mountain View, CA, 94043 USA

Identification

Preferred Customer ID:	CUST001
Preferred ID Type:	Customer ID
DEA Number:	
DEA Effective Date:	
HIN Number:	
DDD Number:	

2.1 Create an XML Overlay

Following is an annotated example of an XML overlay for adding a State Tax ID: field to the Customer Profile page. When adding a new field, the overlay specifies such attributes as the component type, location in the layout, and mapping to the backing data object.

All other attributes are loaded from the component that is being extended.

Code 4-3: XML Overlay to Add a Field

```
<component
name="com.sample.ac.cmty.customers.CSampleFieldCustomerProfileComp"¶
configures="com.modeln.ac.cmty.customers.CMnCustomerProfileComp">
```

The `component name` is the name of the new component containing the customizations. The `configures` statement points to the omponent that is being customized.

```
<import name="imported_class"/>

<import name="imported_class"/>

<!-- one class per import statement -->
```

The `import` statements load the supporting components.

```
-   <section name="panel" order="20" component="CMnPanelLayout">

-    <section name="leftCol" order="10" component="CMnLinearLayout">

-      <section name="generalSection " order="10"¶
       component="CMnFieldSetLayout">
```

The `section` attributes provide the location of the new component on the screen, using layout components. The new field will be added to the section named `generalSection`. You can now, in your overlay, just make a reference to the sections by name and define the new component as a child:

- `CMnPanelLayout` determines the horizontal panels on the page. An order value of 20 indicates that this component is the second one rendered.
- `CMnLinearLayout` defines whether child components in the section are rendered in a horizontal or vertical fashion.
- `CMnFieldSetLayout` is the container for the field. This component is the third one rendered. The `CMnTextBoxComp` component for the field will be embedded in this container.

```
<child name="stateTaxId" order="101" component="CMnTextBoxComp">

          <int name="size">20</int>

          <string name="label">State Tax ID:</string>

          <property name="fieldMapping" type="CMnFieldMapping">

            <string name="dataAttrPath">m.viewObj.SamplestateTaxId</
string>

          </property>

          <boolean name="required">true</boolean>

        </child>
```

Table 4-1: Sample Attributes in an XML Overlay

Attribute	Description
name	The `name` attribute designates the name (`stateTaxId`) of the field as it appears in the database.
order	The order attribute designates the display order of the child components within the section. Display order values are interpreted sequentially within a section to determine the location of the component on the page. To determine the order value, review the code of the component. The display order of the new State Tax Id: field is 101, meaning that the new field will be rendered below the System of Record: field (which has a display order of 100).
component	The component type is `CMnTextBoxComp,` a text input field.
size	Indicates the size of text box for the field.

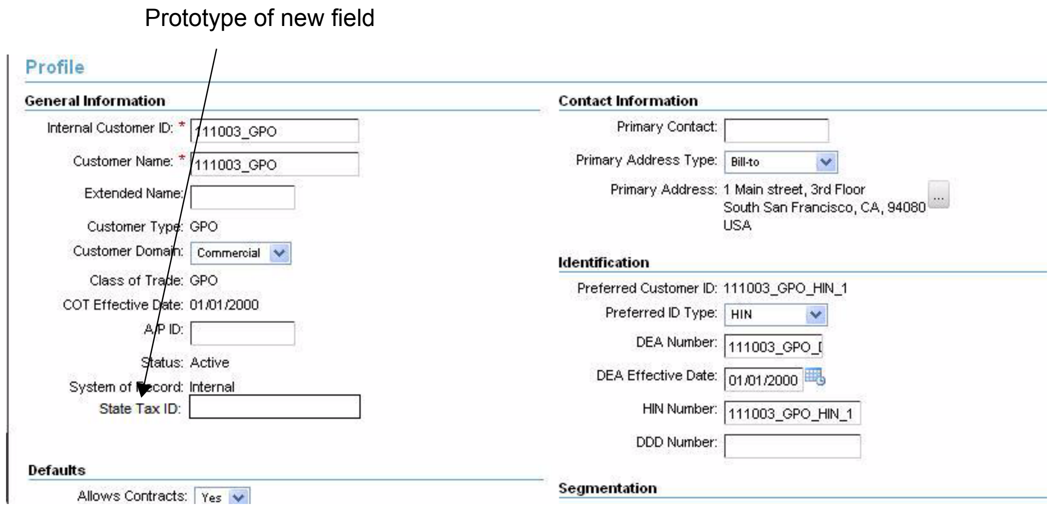
Table 4-1: Sample Attributes in an XML Overlay (Continued)

Attribute	Description
label	Indicates the label of the field.
fieldMapping	The `CMnFieldMapping` property maps the field to the backing object, called an FGO (Fine Grained Object).
dataAttrPath	The `dataAttrPath` attribute points to the backing data value for the field. The starting point is the component that the field mapping is setup on. In the following dataAttrPath: `m.viewObj.SamplestateTaxId` `m` = the mapping component, usually the detail component `viewObj` = references the backing object `SampleStateTaxid` = a reference to a specific method in a Java class that returns the mapped value. This overlay component (`CSampleFieldCustomerProfileComp`) is pointing to a public method on the class being overlaid (`CMnCustomerProfileComp`). The method is named `getSampleStateTaxId()`. Note that the “get” is understood
required	Finally, this is a required field, determined by setting the required attribute to true.

These are the required closing tags.

```
            </section>

        </section>

    </section>

</component>
```

Figure 4-4: Wireframe for Customer Profile Page After Adding State Tax ID Field



Profile

General Information

Internal Customer ID: * 111003_GPO
Customer Name: * 111003_GPO
Extended Name:
Customer Type: GPO
Customer Domain: Commercial
Class of Trade: GPO
COT Effective Date: 01/01/2000
A/P ID:
Status: Active
System of Record: Internal
State Tax ID:

Defaults

Allows Contracts: Yes

Contact Information

Primary Contact:
Primary Address Type: Bill-to
Primary Address: 1 Main street, 3rd Floor
South San Francisco, CA, 94080
USA

Identification

Preferred Customer ID: 111003_GPO_HIN_1
Preferred ID Type: HIN
DEA Number: 111003_GPO_[
DEA Effective Date: 01/01/2000
HIN Number: 111003_GPO_HIN_1
DDD Number:

Segmentation

2.2 Register the Overlay

The next step is to register the overlay in a property file so that the application can use the new file. This is done in a project property file as follows:

Code 4-4: Property for Registering the Overlay

```
com.modeln.App.componentOverlays.com.modeln.ac.cmty.customers.¶
CMnCustomerProfileComp+=¶
\com.sample.ac.cmty.customers.CSampleFieldCustomerProfileComp
```

2.2.1 Project Property File for Registering Overlays

A project property file may contain all the overlays for the project. For a large amount of overlays, divide the property file into sections to organize the overlays.

Two extensions are now registered in the `CustomerProfile_compOverlays.properties` file, as shown in the following example:

Code 4-5: Sample Property File for Registering Overlays

```
com.modeln.App.componentOverlays.com.modeln.ac.cmty.customers.¶
CMnCustomerProfileComp+=¶
\com.sample.ac.cmty.customers.CSampleLabelCustomerProfileComp

com.modeln.App.componentOverlays.com.modeln.ac.cmty.customers.¶
CMnCustomerProfileComp+=¶
\com.sample.ac.cmty.customers.CSampleFieldCustomerProfileComp
```

3.0 Hiding a Field

Fields that are displayed on a page may be hidden using the `renderEnabled` property. Whether a field is displayed in the page is controlled by the `renderEnabled` property. Setting `renderEnabled` to false hides the field; setting `renderEnabled` to true exposes the field.

3.1 Create an XML Overlay

This scenario describes how to hide the page size control in the footer of a table. However, the renderEnabled property may be used to hide any field in a component.

By default, a `table` component contains a page size control in the table footer. This control is used to select the number of rows that are displayed. The page size adjusts to accomodate the number of rows. By hiding the page size controls, the user will not be able to adjust the size of the table.

The following example hides the page size controls at the bottom of a multiple-page table:

Code 4-6: Setting renderEnabled to Hide a Field

```
<component name="com.modeln.myProject.ac.contract.offer.¶
CmyProjectPharmaOfferListTableComp"¶
configures="com.modeln.ac.contract.offer.CMnPharmaOfferListTableComp">

   <child name="myTable" component="CMnPharmaOfferListTableComp">

      <child name="controlBar">

         <child name="pagingInfoComp">

            <child name="pageSizeLabel">

               <boolean name="renderEnabled">false</boolean>

            </child>

            <child name="pageSizeSelector">

            <boolean name="renderEnabled">false</boolean>

            </child>

         </child>

      </child>

[other table configuration here...]

   </child>

</component>
```

3.2 Register the Overlay

Following is an example of the property to register the `renderEnabled` overlay:

Code 4-7: Property for Registering the renderEnabled Overlay

```
com.modeln.App.componentOverlays.com.modeln.ac.contract.offer.¶
CMnPharmaOfferListTableComp+=¶
\com.modeln.myProject.ac.contract.offer.¶
CmyProjectPharmaOfferListTableComp
```

4.0 Setting the Initial Value of a Drop-down Menu

The initial value of a drop-down menu can be customized. Usually, the intial value of a drop-down menu is set to a null value, to force the user to make a selection (for a required field) or to submit a null value (for an optional field).

4.1 Create an XML Overlay

The goal of this scenario is to set the initial value of a drop-down menu to a null value. For required fields, the user is forced to make a selection. For optional fields, the user may select the null value or make a selection from the menu.

To set the initial value to null, create an XML overlay to customize the `initialLabel` attribute of a specific instance of the `CMnDropDownSelComp` component, for example, `CMnStateDropDownSelComp`.

This example is appropriate for a required field, as the initial value is Select a Value, to instruct the user to make a selection. To set the initial value, you configure the `initialLabel` attribute. In this example, the drop-down will display Select a Value as its initial value, to instruct users to make a selection.

Following is an example of an XML overlay for setting the initial value of the drop-down menu:

Code 4-8: XML Overlay to Set the Initial Value of a Drop-down Menu

```
<component¶
name="com.modeln.ac.cmty.helpers.myProject.CMnMyDropDownSelComp"¶
configures="com.modeln.ac.cmty.helpers.CMnStateDropDownSelComp">

   <child name="dropdown" component="CMnStateDropDownSelComp">

      <string name="initialLabel">Select a Value</string>

         <property name="model" type="CMnListSelModel">

. . .

         </property>

   </child>

</component>
```

4.2 Register the Overlay

The following property registers the overlay for use by the application:

Code 4-9: Property for Registering the Initial Value Overlay

```
com.modeln.App.componentOverlays.com.myProject.CMnDropDownSelComp+=¶
\com.myProject.CMnMyDropDownSelComp
```

5.0 Adding an Option to a List Selection

This scenario describes how to add an option to a list selection component, an instance of `CMnListSelModel`. Components in the ListSel package (for example, radio buttons, checkboxes, and drop-down menus) use the `IMnListSelModel` to store the options from which the user can select.

`CMnListSelModel` allows each option for the component to be specified.

Note: Most list selection components are dynamically created using an Enum Java class. XML overlays would not be used to customize a list selection component that is created using an Enum class.

5.1 Create an XML Overlay

This scenario adds options to a drop-down menu, an instance of `CMnDropDownSelComp`. The options are instances of `CMnLabeledOption`.

Code 4-10: XML Overlay to Add Options to a Selection Component

```
<component name="com.myProject.CMnOptionsDropDownSelComp"¶
configures="CMnDropDownSelComp">

    <child name="dropdown" component="CMnDropDownSelComp">

        <property name="model" type="CMnListSelModel">

        <property name="options">

    <collection>

        <property type="CMnLabeledOption">

            <string name="label">Option 1</string>

            <string name="value">option1</string>

        </property>

        <property type="CMnLabeledOption">

            <string name="label">Option 2</string>

            <string name="value">option2</string>

        </property>

    </collection>
```

Code 4-10: XML Overlay to Add Options to a Selection Component (Continued)

```
        </property>

        </property>

    </child>
```

5.2 Register the Overlay

The following property registers the overlay for use by the application:

Code 4-11: Property for Registering the Options Overlay

```
com.modeln.App.componentOverlays.com.myProject.CMnDropDownSelComp+=¶
\com.myProject.CMnOptionsDropDownSelComp
```

6.0 Exporting a Large Data Set

The application provides an export feature to export the results of a query to CSV files. However, by default, only the first 500 rows are retrieved per query, regardless of the total number of rows in the bucket (result set). This was designed to improve performance.

In some cases, more than 500 rows need to be exported. The default behavior can be customized using an XML overlay.

Note: The Sales Submissions and Managed Care Modules in Release 5.2.1 contain an Exporter Framework that uses streaming to export large data sets. Refer to the *Sales Submissions Developer's Guide* for more information on the Exporter Framework.

6.1 Create an XML Overlay

In this example, the `com.modeln.ac.gp.plugins.CMnTransactionDetailListComp` component is extended to set the boolean property `formatAllResults` to **true**. For example:

Code 4-12: Exporting a Large Data Set

```
<component
name="com.myproject.ac.gp.plugins.CSampleTransactionDetailListComp"¶
configures="com.modeln.ac.gp.plugins.CMnTransactionDetailListComp">

    <boolean name="formatAllResults">true</boolean>

</component>
```

Note: A Cognos Report may also be used to generate a report for a large data set.

Important: The `formatAllResults` property stores the exported data in the memory buffer. For large data sets, there is risk of throwing an

`OutOfMemoryException`. Care must be taken to balance the potential size of exports and the available memory.

6.2 Register the Overlay

Next, define an overlay so that the application can use the new file. This is done by adding a property in the project property file. For example:

Code 4-13: Property for Registering the Large Data Set Overlay

```
com.modeln.App.componentOverlays.com.modeln.ac.gp.plugins.¶
CMnTransactionDetailListComp+=¶
\com.myproject.ac.gp.plugins.CMPTransactionDetailListComp
```

7.0 Customizing Tables

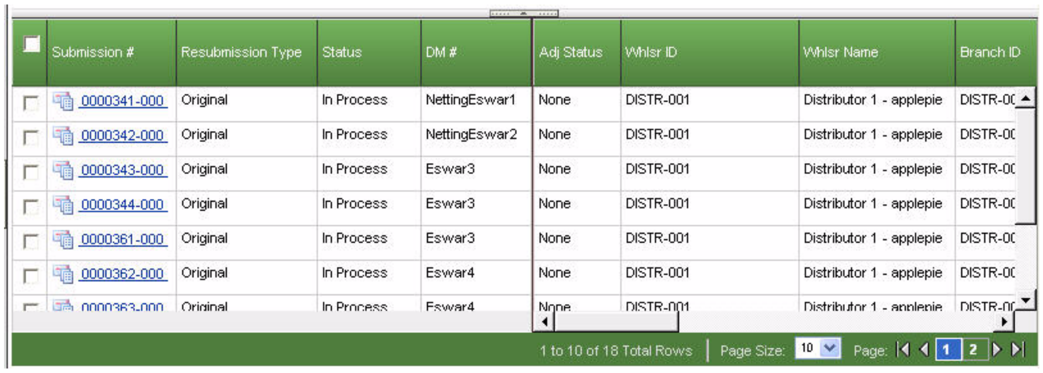
7.1 Configuring Horizontal Scrolling (Scroll Locks)

If a table has too many columns to render within a normally sized browser window, horizontal scrolling (also called Scroll Locks) can be defined on a subset of the columns. When configuring a table for horizontal scrolling, a range of columns are set up to be fixed. Usually, the columns that are fixed are one or more columns on the left. The remaining columns continue to scroll. The information in the fixed columns is always in view and can be matched to the information in the scrolling columns on the right.

This technique is used extensively in the Sales Module.

Note: The Indirect Sales Table contains both horizontal and vertical scrolling. The following example describes horizontal scrolling only.

Figure 4-5: Horizontal Scrolling on Indirect Sales Table



	Submission #	Resubmission Type	Status	DM #	Adj Status	Whlsr ID	Whlsr Name	Branch ID
☐	0000341-000	Original	In Process	NettingEswar1	None	DISTR-001	Distributor 1 - applepie	DISTR-0
☐	0000342-000	Original	In Process	NettingEswar2	None	DISTR-001	Distributor 1 - applepie	DISTR-0
☐	0000343-000	Original	In Process	Eswar3	None	DISTR-001	Distributor 1 - applepie	DISTR-0
☐	0000344-000	Original	In Process	Eswar3	None	DISTR-001	Distributor 1 - applepie	DISTR-0
☐	0000361-000	Original	In Process	Eswar3	None	DISTR-001	Distributor 1 - applepie	DISTR-0
☐	0000362-000	Original	In Process	Eswar4	None	DISTR-001	Distributor 1 - applepie	DISTR-0
☐	0000363-000	Original	In Process	Eswar4	None	DISTR-001	Distributor 1 - applepie	DISTR-0

1 to 10 of 18 Total Rows | Page Size: 10 Page: 1 2

7.1.1 Create an XML Overlay

If a table contains many columns, adding horizontal scrolling makes the information easier to read. There are three properties that need to be set in the config section of the table component to configure horizontal scrolling:

- `scrollWidth` - Sets the width of the scrolling area, either in pixels or a relative percentage.

- `numColsBeforeScroll` - Sets the number of columns that must be displayed in the scrolling area before the horizontal scrollbar appears and scrolling is activated.
- `firstColName` - Sets the name of the column (cell) where scrolling will begin if the `numColsBeforeScroll` value is reached.

Following is an example for an XML overlay to add horizontal scrolling to a table:

Code 4-14: XML Overlay to Add Horizontal Scrolling to a Table

```
<component¶
name="com.myproject.ac.sales.submission.direct.CSampleDirectSales¶
SubmissionListComp"¶
configures="com.modeln.ac.sales.submission.direct.CMnDirectSales¶
SubmissionListComp">

<property name="config" type="CMnTableConfig">

   ...Table Row Definitions...


   ...Table Configuration...


   <!-- Hortizontal Scrolling Definition -->

   <property name="colsConfig" type="CMnTableColsConfig">

     <property name="cols" type="CMnUiConfigMap">

       <property name="colGroup" type="CMnTableColGroupConfig">


         <!-- Width of scrolling area -->

         <string name="scrollWidth">70%</string>


         <!-- # of columns to have before scrolling is used -->

         <int name="numColsBeforeScroll">2</int>


         <!-- Name of first column in scrolling region -->

         <string name="firstColName">reversalCell</string>

       </property>

     </property>

   </property>

</property>

</component>
```

7.1.2 Register the Overlay

Next, register the overlay so that the application can use the new file. This is done by adding a property in the project property file. For example:

Code 4-15: Property for Registering the Horizontal Scrolling Overlay

```
com.modeln.App.componentOverlays.com.modeln.ac.sales.submission.¶
direct.CMnDirectSalesSubmissionListComp+=\com.myproject.ac.sales.¶
submission.direct.CSampleDirectSalesSubmissionListComp
```

7.1.3 Alternate Method for Configuring Horizontal Scrolling

An alternate method for configuring horizontal scrolling exists. Although the previous method [Configuring Horizontal Scrolling (Scroll Locks)] is the recommended method to configure horizontal scrolling, there is an alternate, dynamic method. In this method, the `scrollWidth` and `numColsBeforeScroll` attributes are required. However, the `firstColName` attribute is omitted. Instead, a `cols` node is defined within the `scrollingColGroup` node. The `cols` node lists all the columns to be included in the scrolling section in the correct order.

This method is used when the columns' position and visibility are configurable, so that there is no absolute first column for the scrolling section.

The following example is taken from the `CMnMcdClaimLinesComp.xml` component in the Medicaid module.

Code 4-16: Example for Horizontal Scrolling Alternate Method

```
        <property name="scrollingColGroup"¶
type="CMnTableColGroupConfig">

          <int name="numColsBeforeScroll">1</int>

          <string name="scrollWidth">70%</string>

          <property name="cols" type="CMnUiConfigMap">

<--All columns in order.-->

            <property name="calcUraCell" type="CMnTableColConfig"/>

            <property name="uraInvoicedCell" type="CMnTableColConfig">

              <string name="domAttr">invURAColTag</string>

            </property>

            <property name="uraAdjCell" type="CMnTableColConfig">

              <string name="domAttr">invURAAdjColTag</string>

            </property>

           <property name="unitsInvoicedCell" type="CMnTableColConfig">

              <boolean name="totalsEnabled">true</boolean>

              <string name="domAttr">invUnitsColTag</string>

            </property>

          </property>
```

Code 4-16: Example for Horizontal Scrolling Alternate Method (Continued)

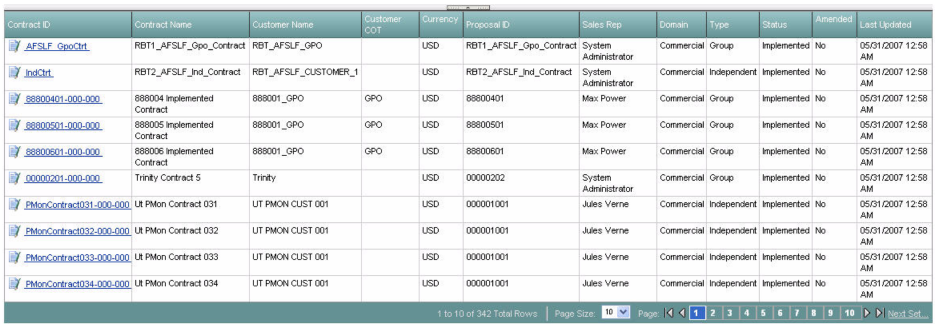
```
        </property>
```

7.2 Hiding a Table Column

In addition to adding scrolling, another common customization technique for wide tables is to hide unnecessary columns. For example, the Contracts table is wider than the browser, so it is a good candidate for hiding a column (or horizontal scrolling).

Note: Although the Contact Name column is hidden in the following overlay example, it probably is a necessary column in the Contracts table and would not be hidden.

Figure 4-6: Contracts Table



Contract ID	Contract Name	Customer Name	Customer COT	Currency	Proposal ID	Sales Rep	Domain	Type	Status	Amended	Last Updated
AFSLF_GpoCtrt	RBT1_AFSLF_Gpo_Contract	RBT_AFSLF_GPO		USD	RBT1_AFSLF_Gpo_Contract	System Administrator	Commercial	Group	Implemented	No	05/31/2007 12:58 AM
IndCtrt	RBT2_AFSLF_Ind_Contract	RBT_AFSLF_CUSTOMER_1		USD	RBT2_AFSLF_Ind_Contract	System Administrator	Commercial	Independent	Implemented	No	05/31/2007 12:58 AM
88800401-000-000	888004 Implemented Contract	888001_GPO	GPO	USD	88800401	Max Power	Commercial	Group	Implemented	No	05/31/2007 12:58 AM
88800501-000-000	888005 Implemented Contract	888001_GPO	GPO	USD	88800501	Max Power	Commercial	Group	Implemented	No	05/31/2007 12:58 AM
88800601-000-000	888006 Implemented Contract	888001_GPO	GPO	USD	88800601	Max Power	Commercial	Group	Implemented	No	05/31/2007 12:58 AM
00000201-000-000	Trinity Contract 5	Trinity		USD	00000202	System Administrator	Commercial	Group	Implemented	No	05/31/2007 12:58 AM
PMonContract031-000-000	Ut PMon Contract 031	UT PMON CUST 001		USD	000001001	Jules Verne	Commercial	Independent	Implemented	No	05/31/2007 12:58 AM
PMonContract032-000-000	Ut PMon Contract 032	UT PMON CUST 001		USD	000001001	Jules Verne	Commercial	Independent	Implemented	No	05/31/2007 12:58 AM
PMonContract033-000-000	Ut PMon Contract 033	UT PMON CUST 001		USD	000001001	Jules Verne	Commercial	Independent	Implemented	No	05/31/2007 12:58 AM
PMonContract034-000-000	Ut PMon Contract 034	UT PMON CUST 001		USD	000001001	Jules Verne	Commercial	Independent	Implemented	No	05/31/2007 12:58 AM

7.2.1 Create an XML Overlay

Following is an example of an XML overlay to hide a column in a table. This example hides a Proposal Id column.

Code 4-17: XML Overlay to Hide a Column

```
<component name="com.myProject.ac.contract.offer.CMyContractTable"¶
configures="com.modeln.ac.contract.offer.CMnOfferListTableComp">
    <property name="config">
        <property name="rowdefs">
            <property name="proposalIdCell"
                <boolean name="visible">false</boolean>
            </property>
        </property>
    </property>
</component>
```

7.2.2 Register the Overlay

Next, register the overlay so that the application can use the new file. This is done by adding a property in the project property file. For example:

Code 4-18: Property for Registering the Overlay to Hide a Column

```
com.modeln.App.componentOverlays.com.modeln.ac.contract.offer.¶
CMnOfferListTableComp+=\com.myProject.ac.contract.offer.¶
CMyContractTable
```

7.3 Changing Column Order in a Table

To change the column order, create an XML overlay of the base list table to change the display order of the desired column using the `order` attribute. For a description of the `order` attribute, see Table 4-1.

When changing the display order for a column, you are required to modify both the header and body cell definitions for the column. If a footer is present, it must be modified also. In the following example, the Status column is moved by overlaying the definition of `statusCell`.

Important: To change the column order in a table, modify the display order for the table header and body (required), and the table footer (if present), cell definitions. It is recommended that you assign the same order to all three elements (header, body, footer).

Code 4-19: XML Overlay for Changing Column Order in a Table

```
<component name="com.myProject.ac.contract.offer.CMyTable"¶

configures="com.modeln.ac.contract.offer.CMnOfferListTableComp">

  <property name="config">

    <property name="rowDefs">

      <property name="header">

        <!--

            Change the display order of the status column in the header.

          -->

        <property name="statusCell" order="11"/>


      <property name="body">

        <!--

             Change the display order of the status column in the body.

          -->

        <property name="statusCell" order="11"/>
```

Code 4-19: XML Overlay for Changing Column Order in a Table (Continued)

```
       <property name="footer">

         <!--

             Change the display order of the status column in the footer.

           -->

         <property name="statusCell" order="11"/>


       </property>

     </property>

   </property>
```

7.3.1 Register the Overlay

Next, register the overlay so that the application can use the new file. This is done by adding a property in the project property file. For example:

Code 4-20: Property for Registering the Overlay to Change the Column Order

```
com.modeln.App.componentOverlays.com.modeln.ac.contract.offer.¶
CMnOfferListTableComp+=\com.myProject.ac.contract.offer.¶
CMyTable
```

7.4 Changing the Default Number of Rows in a Table

The default number of rows loaded into a table is 10. To set a new default, use an XML overlay for the table.

7.4.1 Create an XML Overlay

To change the default number of rows in a table, modify the `pageSize` property within the `bodyConfig` portion of the table configuration.

Following is an example of an XML overlay to set the default number of rows in an Alerts table to 30:

Code 4-21: XML Overlay to Change the Default Number of Rows in a Table

```
<component name="com.sample.ac.application.CSampleAlertsTableComp"¶
configures="com.modeln.ac.application.CMnAlertsTableComp">

   <property name="bodyConfig">

      <int name="pageSize">30</int>

   </property>

</component>
```

Important: A large page size to increase the number of rows in a table may cause the page to load slowly.

Note: To change the default number of rows for every table in the system, edit the following Application Switch: `com.modeln.AppSwitch.table.defaultPageSize`.

7.4.2 Register the Overlay

Create an entry similar to the following one in the overlay property file for the project.

Code 4-22: Property for Registering the Number of Rows in a Table Overlay

```
com.modeln.App.componentOverlays.com.modeln.ac.cmty.customers.CMnCusto¶
merProfileComp+=\com.sample.ac.cmty.customers.CSampleLabelCustomerProf¶
ileComp
```

7.5 Changing Maximum Number of Records in a Table

The default maximum number of records retrieved is 500. This applies to the user interface (not exporting to CSV).

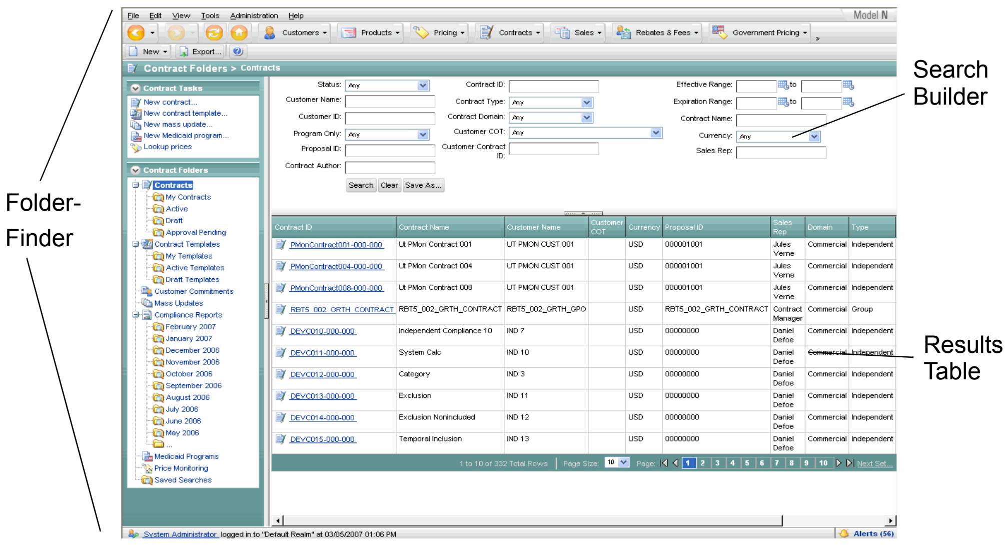
There is a property setting that can be added or modified to accomplish this. Please note that this is global to the whole application. Also, a very high number of records could lead to memory problems and perfomance degradation.

```
com.modeln.selectQuery.resultLimit=5000
```

8.0 Adding a Search Field to a Folder-Finder Component

Search fields can be added to any folder-finder component. A folder-finder is a complex component containing several smaller components.

Figure 4-7: Contracts Folder-Finder Page



Status:	Any	Contract ID:		Effective Range:	to
Customer Name:		Contract Type:	Any	Expiration Range:	to
Customer ID:		Contract Domain:	Any	Contract Name:	
Program Only:	Any	Customer COT:	Any	Currency:	Any
Proposal ID:		Customer Contract ID:		Sales Rep:	
Contract Author:					

Contract ID	Contract Name	Customer Name	Customer COT	Currency	Proposal ID	Sales Rep	Domain	Type
PMonContract001-000-000	Ut PMon Contract 001	UT PMON CUST 001		USD	000001001	Jules Verne	Commercial	Independent
PMonContract004-000-000	Ut PMon Contract 004	UT PMON CUST 001		USD	000001001	Jules Verne	Commercial	Independent
PMonContract008-000-000	Ut PMon Contract 008	UT PMON CUST 001		USD	000001001	Jules Verne	Commercial	Independent
RBT5_002_GRTH_CONTRACT	RBT5_002_GRTH_CONTRACT	RBT5_002_GRTH_GPO		USD	RBT5_002_GRTH_CONTRACT	Contract Manager	Commercial	Group
DEVC010-000-000	Independent Compliance 10	IND 7		USD	00000000	Daniel Defoe	Commercial	Independent
DEVC011-000-000	System Calc	IND 10		USD	00000000	Daniel Defoe	Commercial	Independent
DEVC012-000-000	Category	IND 3		USD	00000000	Daniel Defoe	Commercial	Independent
DEVC013-000-000	Exclusion	IND 11		USD	00000000	Daniel Defoe	Commercial	Independent
DEVC014-000-000	Exclusion Nonincluded	IND 12		USD	00000000	Daniel Defoe	Commercial	Independent
DEVC015-000-000	Temporal Inclusion	IND 13		USD	00000000	Daniel Defoe	Commercial	Independent

1 to 10 of 332 Total Rows | Page Size: 10 Page: 1 2 3 4 5 6 7 8 9 10 Next Set...

System Administrator logged in to "Default Realm" at 03/05/2007 01:06 PM Alerts (56)

The following customizations are required to add a search field:

- An extension of the SearchBuilder component to add the search field.
- An overlay of `CMnRootConfComp` to register the new search field. (This is an additional overlay required for registering extensions to dynamically instantiated components.)
- An overlay of the Results Table component to add a column to display the results.

Note: There is currently a limitation in the ACF that prevents overlays of dynamically instantiated components such as a SearchBuilder. Therefore, a new component that extends the CMnContractSearchBuilderComp component is created. The extension is then registered in an overlay of the CMnAppRootConfComp component. The code to create an extension is identical to an overlay; the difference is that the extension must be registered in CMnAppRootConfComp.

8.1 Extend the SearchBuilder

In the following example, the Contract SearchBuilder component is extended to add the ability to search by the Activated Date. Note that the type is specified because the Activated Date field does not currently exist in the component.

Code 4-23: XML Overlay to Add a Search Field

```
<component name=¶
"com.MyProject.ac.contract.folder.CSampleContractSearchBuilderComp"¶
configures=¶
"com.modeln.ac.contract.folder.CMnContractSearchBuilderComp">



   <import name="com.modeln.ui.CMnDropDownSelComp"/>

   <import name="com.modeln.ui.listsel.CMnBooleanListSelModel"/>

   <import name="com.modeln.ui.listsel.CMnEnumListSelModel"/>

   <import name="com.modeln.ui.mapping.CMnFieldMapping"/>

   <import name="com.modeln.ui.search.model.CMnStringOperator"/>

   <import name="com.modeln.ui.search.model.CMnEnumOperator"/>

   <import name=¶
   "com.modeln.ui.search.builder.CMnDefaultCompSearchField
   Definition"/>¶



 <property name="definitions">

    <property name="ActivatedDate"¶
type="CMnDefaultCompSearchFieldDefinition" order="40">

      <string name="label">Activated Date:</string>

      <string name="fullName">ActivatedDate</string>

      <class name="type">java.lang.Date</class>

      <boolean name="operatorVisible">false</boolean>

          <property name="operator" type="CMnDateOperator">

             <value>EQUALS</value>

          </property>

   </property>

 </property>

</component>
```

8.1.1 Register the Extension

Since the `CMnContractSearchBuilderComp` is a dynamically instantiated component, the extension to add a new search field must be registered. To do this, create an overlay of the `CMnPharmaContractsRootConfComp` file (which is itself an overlay of `CMnRootConfComp`). The extension must configure `CMnRootConfComp` because an overlay cannot configure another overlay. The overlay modifies the search definition to use the new search builder component `CSampleContractSearchBuilderComp`.

Code 4-24: XML Overlay to Register the Extension

```
<component name=¶
"com.MyProject.ac.contract.folder.CExtensionRootConfComp"¶

 configures="com.modeln.ui.CMnRootConfComp">



  <property name="userModel">

    <property name="search">



      <property name="contractSearchBuilder">

 <!-- Replace Search Builder component definition with extension -->

        <string name="searchBuilder">¶
          com.MyProject.ac.contract.folder.¶
          CSampleContractSearchBuilderComp</string>¶

      </property>

    </property>

  </property>



</component>
```

8.1.2 Register the Overlay

The extension containing the new search field `(CExtensionRootConfComp)` configured the `CMnRootConfComp` in an overlay. This overlay must be registered in a property file, using a property similar to the following:

Code 4-25: Property for Registering CExtensionRootConfComp

```
com.modeln.App.componentOverlays.com.modeln.ui.CMnRootConfComp+=\¶

 com.MyProject.ac.contract.folder.CExtensionRootConfComp
```

Note: The `RootConfComp` file contains many overlaid configurations for a module. Add the property to the existing `RootConfComp`; there is no need to create a new `RootConfComp`.

8.2 Add a Column

A column must be added to the results table on the Contracts folder-finder page to display the results of the new search field. The following overlay adds the Activated Date column to the results table. The column must be added to the header, body, and optional footer of the table. In addition, the display order must be specified to define the position of the new column.

Code 4-26: XML Overlay to Add a Column for the New Search Field

```
<component name="com.myProject.ac.contract.offer.CActDateTable"¶

configures="com.modeln.ac.contract.offer.CMnOfferListTableComp">

  <property name="config">

    <property name="rowDefs">

      <property name="header">


        <!--

             This is the new header cell with an order of 15 so it

             is between the oppNum and userCell columns

          -->

        <property name="actDateCell" order="15"
type="CMnTableCellConfig">

          <string name="renderComp">com.modeln.ui.CMnTextFieldComp</
string>

          <boolean name="literal">true</boolean>

          <string name="literalValue">ActivatedDate</string>

          <property name="sortDef"
type="CMnPObjQueryTableColumnSortDef">

            <string name="attrPath">ActivatedDate</string>

          </property>

        </property>


      </property>

      <property name="body">


        <!--

             This is the new body cell with an order of 15 so it

             is between the oppNum and userCell columns.  We use the

             extension proxy so that it calls the "getActivatedDate()"

             method.
```

Code 4-26: XML Overlay to Add a Column for the New Search Field (Continued)

```
          -->

        <property name="actDateCell" order="15"
type="CMnTableCellConfig">

          <string name="renderComp">com.modeln.ui.CMnTextFieldComp</
string>

          <string name="attrPath">extnProxy.ActivatedDate</string>

          <boolean name="editable">false</boolean>

        </property>


      </property>

    </property>


  </property>
```

8.2.1 Register the Overlay

Create an entry similar to the following one in the overlay property file for the project.

Code 4-27: Property for Registering the Overlay to Add a Column

```
com.modeln.App.componentOverlays.com.modeln.ac.contract.offer.¶
CMnOfferListTableComp+=\com.myProject.ac.contract.offer.CActDateTable
```

A

Component Reference

The following table provides a description of the most common components. It provides the following information:

- Component type
- Component name
- Sample image
- Brief description and Java class

Table A-1: Component Overview

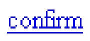
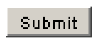
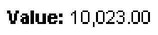
Type	Component	Image	Description and Java Class
Request	`CMnAnchorComp`		Displays a hypertext link for opening a URL. `com.modeln.ui.CMnAnchorComp`
	`CMnButtonComp`		Displays a button for a user-initiated action. The image and label of the button can be customized. `com.modeln.ui.CMnButtonComp`
Label	`CMnTextFieldComp`	 **Value:** 10,023.00	A read-only field often used as a label or a read-only data value. • `com.modeln.ui.CMnTextFieldComp` • `com.modeln.ui.CMnEnumFieldComp` • `com.modeln.ui.number.CMnIntegerFieldComp` • `com.modeln.ui.number.CMnDoubleFieldComp`

Table A-1: Component Overview (Continued)

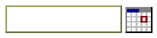
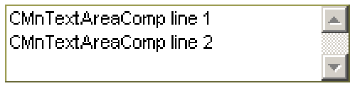
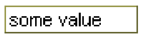
Type	Component	Image	Description and Java Class
Input	`CMnDateBoxComp`		Provides a single input line with a validation to ensure correct date format. A JavaScript calendar is also associated with the field. • `com.modeln.ui.date` • `com.modeln.ui.date.CMnDateBoxComp` • `com.modeln.ui.date.CMnDateTimeBoxComp`
	`CMnTextAreaComp`	 CMnTextAreaComp line 1 CMnTextAreaComp line 2	Multi-line input field. `com.modeln.ui.CMnTextAreaComp`
	`CMnTextBoxComp`	 some value	Single line input field. • `com.modeln.ui.CMnTextBoxComp` • `com.modeln.ui.number.CMnIntegerBoxComp` • `com.modeln.ui.number.CMnDoubleBoxComp` • `com.modeln.ui.currency.CMnSingleCurrencyMoneyBoxComp`

Table A-1: Component Overview (Continued)

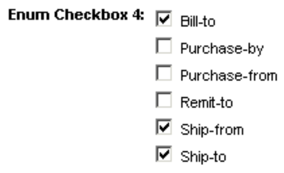
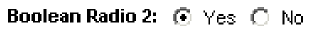
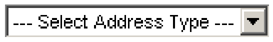
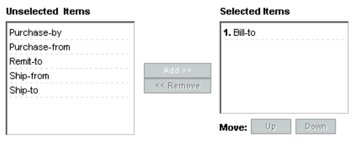
Type	Component	Image	Description and Java Class
Selection	`CMnCheckBoxSelComp`	 Enum Checkbox 4: ✓ Bill-to; ☐ Purchase-by; ☐ Purchase-from; ☐ Remit-to; ✓ Ship-from; ✓ Ship-to	Provides a list of check boxes that correspond to static data items or a backing data model. The list allows multiple items to be selected. • `com.modeln.ui.listsel` • `com.modeln.ui.listsel.CMnCheckBoxSelComp`
	`CMnRadioButtonSelComp`	 Boolean Radio 2: [✓ Yes] No	Provides a list of radio buttons that correspond to static data items or a backing data model. Only one item from the list can be selected at a given time. • `com.modeln.ui.listsel` • `com.modeln.ui.listsel.CMnRadioButtonSelComp`
	`CMnDropDownSelComp`	 --- Select Address Type ---	Provides a collapsible list of items that correspond to static data items or a backing data model. Only a single item from the list can be selected. An initial label can be specified. • `com.modeln.ui.listsel` • `com.modeln.ui.listsel.CMnDropDownSelComp`
	`CMnAddRemoveSelComp`	 Unselected Items: Purchase-by; Purchase-from; Remit-to; Ship-from; Ship-to — Selected Items: 1. Bill-to	Manages a list of selected and unselected items by separating the items into two list boxes. Unselected items appear in the left column and selected items appear in the right column. JavaScript allows the user to move items between the two lists on the client side, submitting the final selections only when saving. • `com.modeln.ui.listsel` • `com.modeln.ui.listsel.CMnAddRemoveSelComp`

Table A-1: Component Overview (Continued)

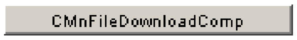
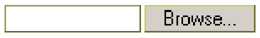
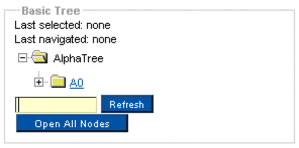
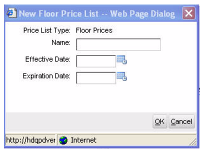
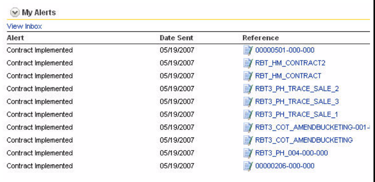
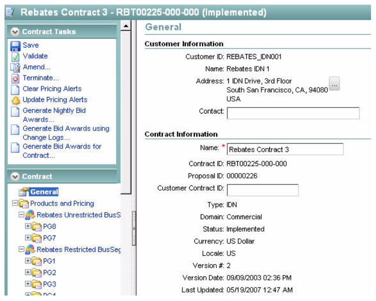
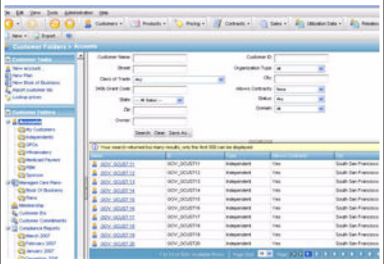
Type	Component	Image	Description and Java Class
Misc.	`CMnFileDownloadComp`		Initiates an HTTP file transfer from the server to the client. `com.modeln.ui.CMnFileDownloadComp`
	`CMnFileUploadComp`		Uploads a file from a local filesystem to the application. `com.modeln.ui.CMnFileUploadComp`

Table A-1: Component Overview (Continued)

Type	Component	Image	Description and Java Class
Complex	`CMnTreeComp`	 Basic Tree Last selected: none Last navigated: none AlphaTree A0 Refresh Open All Nodes	Provides a navigable representation of hierarchical data. • `com.modeln.ui.tree` • `com.modeln.ui.tree.CMnTreeComp`
	`CMnDialogComp`	 New Floor Price List -- Web Page Dialog Price List Type: Floor Prices Name: Effective Date: Expiration Date: OK Cancel http://hdqpdver Internet	Provides a dialog box that is embedded in a popup window. • `com.modeln.ui.dialog` • `com.modeln.ui.dialog.CMnDialogComp`
	`CMnTableComp`	 My Alerts View Inbox Alert / Date Sent / Reference Contract Implemented / 05/19/2007 / 00000501-000-000 Contract Implemented / 05/19/2007 / RBT_HM_CONTRACT2 Contract Implemented / 05/19/2007 / RBT_HM_CONTRACT Contract Implemented / 05/19/2007 / RBT3_PH_TRACE_SALE_2 Contract Implemented / 05/19/2007 / RBT3_PH_TRACE_SALE_3 Contract Implemented / 05/19/2007 / RBT3_PH_TRACE_SALE_1 Contract Implemented / 05/19/2007 / RBT3_COT_AMENDBUCKETING-001- Contract Implemented / 05/19/2007 / RBT3_COT_AMENDBUCKETING Contract Implemented / 05/19/2007 / RBT3_PH_004-000-000 Contract Implemented / 05/19/2007 / 00000206-000-000	Provides a table component. • `com.modeln.ui.table` • `com.modeln.ui.table.CMnDialogComp`
	`CMnDetailComp`	 Rebates Contract 3 - RBT00225-000-000 (Implemented) Contract Tasks: Save, Validate, Amend..., Terminate..., Clear Pricing Alerts, Update Pricing Alerts, Generate Nightly Bid Awards..., Generate Bid Awards using Change Logs..., Generate Bid Awards for Contract... Contract: General, Products and Pricing, Rebates Unrestricted BusS, PG8, PG7, Rebates Restricted BusSeg, PG1, PG2, PG3 General Customer Information Customer ID: REBATES_IDN001 Name: Rebates IDN 1 Address: 1 IDN Drive, 3rd Floor South San Francisco, CA, 94080 USA Contact: Contract Information Name: * Rebates Contract 3 Contract ID: RBT00225-000-000 Proposal ID: 00000226 Customer Contract ID: Type: IDN Domain: Commercial Status: Implemented Currency: US Dollar Locale: US Version #: 2 Version Date: 09/09/2003 02:36 PM Last Updated: 05/19/2007 12:47 AM	The Detail component allows users to locate, view, and edit a specific data object through the use of navigation components and an object detail view. • `com.modeln.ui.detail` • `com.modeln.ui.detail.CMnDetailComp`
	`CMnFolderFinderComp`		Provides support for an application module. Contains a search model. • `com.modeln.ui.folderfinder` • `com.modeln.ui.folderfinder.CMnFolderFinderComp`