

LISTEN.
THINK.
SOLVE.®

PharmaSuite®



CONFIGURATION AND EXTENSION - VOLUME 3

RELEASE 8.4

TECHNICAL MANUAL

PUBLICATION PSCEV3-GR008E-EN-E-DECEMBER-2017

Supersedes publication PSCEV3-GR008D-EN-E



Allen-Bradley • Rockwell Software

Rockwell
Automation

Contact Rockwell See contact information provided in your maintenance contract.

Copyright Notice © 2017 Rockwell Automation Technologies, Inc. All rights reserved.
This document and any accompanying Rockwell Software products are copyrighted by Rockwell Automation Technologies, Inc. Any reproduction and/or distribution without prior written consent from Rockwell Automation Technologies, Inc. is strictly prohibited. Please refer to the license agreement for details.

Trademark Notices FactoryTalk, PharmaSuite, Rockwell Automation, Rockwell Software, and the Rockwell Software logo are registered trademarks of Rockwell Automation, Inc.

The following logos and products are trademarks of Rockwell Automation, Inc.:

FactoryTalk Shop Operations Server, FactoryTalk ProductionCentre, FactoryTalk Administration Console, FactoryTalk Automation Platform, and FactoryTalk Security.
Operational Data Store, ODS, Plant Operations, Process Designer, Shop Operations, Rockwell Software CPGSuite, and Rockwell Software AutoSuite.

Other Trademarks ActiveX, Microsoft, Microsoft Access, SQL Server, Visual Basic, Visual C++, Visual SourceSafe, Windows, Windows 7 Professional, Windows Server 2008, Windows Server 2012, and Windows Server 2016 are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

Adobe, Acrobat, and Reader are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States and/or other countries.

ControlNet is a registered trademark of ControlNet International.

DeviceNet is a trademark of the Open DeviceNet Vendor Association, Inc. (ODVA).

Ethernet is a registered trademark of Digital Equipment Corporation, Intel, and Xerox Corporation.

OLE for Process Control (OPC) is a registered trademark of the OPC Foundation.

Oracle, SQL*Net, and SQL*Plus are registered trademarks of Oracle Corporation.

All other trademarks are the property of their respective holders and are hereby acknowledged.

Warranty This product is warranted in accordance with the product license. The product's performance may be affected by system configuration, the application being performed, operator control, maintenance, and other related factors. Rockwell Automation is not responsible for these intervening factors. The instructions in this document do not cover all the details or variations in the equipment, procedure, or process described, nor do they provide directions for meeting every possible contingency during installation, operation, or maintenance. This product's implementation may vary among users.

This document is current as of the time of release of the product; however, the accompanying software may have changed since the release. Rockwell Automation, Inc. reserves the right to change any information contained in this document or the software at any time without prior notice. It is your responsibility to obtain the most current information available from Rockwell when installing or using this product.

Chapter 1	Introduction	1
	Intended Audience	4
	Typographical Conventions	5
Chapter 2	Extension and Naming Conventions	7
	Guidelines to Control the Migration Effort	7
	Building Block-specific Conventions	9
	Oracle Database Data Types: varchar2 and nvarchar2	10
Chapter 3	Adding Numeric Fields to the Database.....	11
Chapter 4	Configuring the Universe of Recipe and Workflow Designer	13
	What Is the Universe?	13
	What Is a Universe Object Type?	13
	What Is a Universe Object?	13
	What Is the Universe Explorer?	14
	What Is the Universe Configuration?	16
	What Is the Open Dialog?	17
	What Is the Select Material Dialog?	17
	What Is the Data Dictionary?	17
	Introduction to Universe Configuration	18
	Step 1: Simple Universe Configuration for One Universe Object Type	19
	Step 2: Universe Configuration for One Universe Object Type with Database Information Inside	21
	Step 3: Universe Configuration for One Universe Object Type with Some Database Information.....	23

	Step 4: Complex Universe Configuration for a Single Universe Object Type	24
	Step 5: Universe Configuration for Multiple Universe Object Types	26
	Elements of the Universe Configuration	31
	Parameters in the Universe Configuration	33
	Adjusting the Configuration	35
	Tips and Tricks	36
Chapter 5	Configuring the Setlist of Recipe and Workflow Designer	39
	What Is the Setlist?	39
	Adjusting the Configuration	40
	Adjusting the Configuration of Custom Attributes	41
Chapter 6	Configuring the Property Windows of Recipe and Workflow Designer	43
	What Is a Property Window?	43
	Adjusting the Configuration	44
	Adjusting the Configuration of Custom Attributes (Header and Element Property Window)	45
	Adjusting the Packaging Level-related Attributes (Header Property Window).....	47
	Adjusting the Configuration of Custom Attributes (Source Property Window).....	49
Chapter 7	Configuring the Parameter Panel of Recipe and Workflow Designer	53
	What Is the Parameter Panel?	53
	Adjusting the Configuration	54
	Adjusting the Configuration of Custom Attributes	55
	Adjusting the Packaging Level-related Attributes	56
Chapter 8	Managing the Layout of Recipe and Workflow Designer and Data Manager	59
	What Is Layout Management?	59
	Removing/Adding Items from/to the Menu Bar, Toolbar, Context Menu.....	60
	Changing the Layout of Floating Windows.....	60

Chapter 9	Configuring the Expression Editor of Recipe and Workflow Designer and Data Manager	63
	What Is the Expression Editor?	63
	Providing a Context-related Function.....	64
	Adding a New Function	64
	Enabling Localization and Online Help.....	67
	Providing an Arbitrary Function.....	68
	Creating a Function Evaluation Class.....	68
	Implementing a Type Inference Class	69
	Exporting the New Function via Annotation.....	70
	Configuring PharmaSuite	71
	Enabling Localization and Online Help	72
Chapter 10	Using ERP BOMs in Recipe and Workflow Designer	75
	Technical Details	75
Chapter 11	Configuring the Initialization of Material Parameters in Recipe and Workflow Designer	77
	Using the IMaterialParameterCustomAttributesHandler Interface	77
	Configuring the Custom Attributes Handler for Material Parameters	78
	Example Implementation.....	79
Chapter 12	Configuring Menu and Toolbar of Recipe and Workflow Designer.....	81
	What Are Actions in Recipe and Workflow Designer?	81
	Implementing an Action Class	82
	Configuring Action Classes	82
	Configuring Actions	84
	Configuring an Action for the Menu Bar.....	84
	Configuring an Action for the Toolbar	85
	Adding a Context to an Action	85
Chapter 13	Configuring the Details Window of Data Manager - Work Center.....	87
	What Is the Details Window?.....	87
	Adjusting the Configuration	88

Adjusting the Configuration of Custom Attributes	89
Chapter 14 Configuring the Change History of Data Manager - Work Center	91
Which Events Are Tracked by the Change History?	91
Adjusting the Configuration	92
Adjusting the Configuration of Custom Attributes	92
Monitoring Work Center Objects in Non-PharmaSuite Applications	94
Chapter 15 Configuring FSMs for Equipment Properties	95
Configuring an FSM	95
Configuring Semantic Properties	96
Chapter 16 Providing Images for Equipment Classes in Data Manager	99
Providing an Image	99
Chapter 17 Providing Report Designs for (Template) Equipment Entities in Data Manager	101
Providing a Report Design	101
Chapter 18 Adding an Equipment Logbook Category Accessible by Phase Building Blocks	103
Extending the Category Choice List	103
Writing Equipment Logbook Entries Using New Categories	104
Accessing Equipment Logbook Entries	105
Chapter 19 Retrieving Specific Information from the Equipment Logbook	107
What Is a Last Product?	107
Retrieving the Last Product Information	109
Retrieving the Last Product Logbook Entry	109
Implementing the Expression Editor Function	111
Implementing the Type Inference Class	111
Chapter 20 Configuring Additional Purposes for the Equipment Type Technical Property Type	113
Add a Purpose and Extend the Validation Check	113

Chapter 21 Adding a Workflow Type to Workflow Designer and the Production Execution Client	115
Creating a Workflow Type	115
Adding a Group to the Production Execution Client	116
Chapter 22 Managing Cockpit Actions of the Production Execution Client.....	117
What Are Cockpit Actions?	117
Implementing an Action Class	118
Actions without a User Interface.....	118
Actions Displaying a Dialog.....	119
Actions Displaying a Form	119
Actions Displaying a Panel	120
Configuring Action Classes	123
Configuring Actions	124
Actions with Restricted Access Rights.....	125
Filtering Actions by Means of Access Privileges	126
Adapting the ChangeStationAction (Register at Station) Action	127
Enabling of Scrolling in Forms	127
Optional Cockpit Actions	128
Invisible Cockpit Actions	129
Scanner Actions in the Cockpit.....	130
Running Processes in the Cockpit	133
Chapter 23 Configuring Menu and Toolbar of the Production Response Client	137
What Are Actions in the Production Response Client?.....	137
Implementing an Action Class	138
Configuring Action Classes	138
Configuring Actions	139
Configuring an Action for the Menu Bar.....	139
Configuring an Action for the Toolbar	140

Chapter 24	Adding Filter Attributes to the Exception Dashboard of the Production Response Client	141
	What Are Filter Attributes in the Production Response Client?	141
	Supported Database Objects	142
	Data Types Supported as Filter Attributes	144
	Implementing a Filter Attribute.....	147
	Methods Common to All Data Types	148
	String Data Type	148
	String List Data Type	150
	Option List Data Type.....	152
	Choice List Data Type.....	156
	Boolean Data Type	158
	Date Data Type	160
	Configuring Filter Attributes.....	162
Chapter 25	Material Handling for DCS.....	163
Chapter 26	Adapting User Documentation	165
	Accessing and Configuring the Author-it Database.....	165
	Generating Help URLs in PharmaSuite	166
	Adapting the Help Path.....	169
Chapter 27	Reference Documents	171
Chapter 28	Revision History.....	173
Index	179

Figure 1: Universe Explorer with no optional filters applied.....	15
Figure 2: Universe Explorer with all filters applied	16
Figure 3: Open dialog, working the same way as the Universe Explorer	17
Figure 4: Data Dictionary Manager displaying all properties of a Java Bean class.....	18
Figure 5: Universe Explorer for the simple configuration	20
Figure 6: Data Dictionary Manager displaying all database information that is required	21
Figure 7: Universe Explorer for a simple configuration including database information.....	22
Figure 8: Universe Explorer for a simple configuration including placeholders for additional database information.....	24
Figure 9: Universe Explorer for a more complex configuration, still containing a single object type	26
Figure 10: Universe Explorer for a complex configuration containing multiple object types.....	31
Figure 11: Editing the red_universeConfiguration list.....	35
Figure 12: Setlist	40
Figure 13: Header property window	44
Figure 14: Example: Customized bean class for the Source Property Window	49
Figure 15: Example: Customized factory class for the Source Property Window	49
Figure 16: Example: Registration of a customer factory class for the Source Property Window	50
Figure 17: Creating a new Data Dictionary Class for customer bean classes	51
Figure 18: Parameter Panel (material parameters)	53
Figure 19: Modifying the red_buttonConfiguration list.....	60
Figure 20: Modifying the red_defaultLayoutSettings list	61
Figure 21: Custom functions in the Functions tree panel.....	73
Figure 22: MYCMenuAction action in the MYCMenu sub-menu	84
Figure 23: MYCToolbarAction action toolbar MYCToolbar toolbar	84

Figure 24: MYCMenuAction with context information	85
Figure 25: Basic tab of Details window for stations	88
Figure 26: Adding a choice element in Choice List Manager	103
Figure 27: Additional categories in equipment logbook of Data Manager	104
Figure 28: Example of equipment logbook (binding)	107
Figure 29: Example of equipment logbook (context)	108
Figure 30: New workflow type in Workflow Designer and the Production Execution Client	115
Figure 31: Adding a choice element in Choice List Manager	116
Figure 32: Configuring the MYCAction action for the Cockpit	124
Figure 33: MYCAction action in the Cockpit	125
Figure 34: Access privilege for new action	127
Figure 35: Group of invisible startable items in group of startable items	129
Figure 36: Invisible startable item in group of invisible startable items	129
Figure 37: Station-specific configuration with ActivitySet/SetStartableItemGroups=StartableItemsGroup	132
Figure 38: Station-specific configuration with ActivitySet/SetStartableItemGroups=StartableItemsGroup-NoInvisible	132
Figure 39: Group of invisible startable items in group of startable items	133
Figure 40: No group of invisible startable items in group of startable items	133
Figure 41: Invisible startable item in group of invisible startable items	133
Figure 42: MYCMenuAction action in the menu	139
Figure 43: MYCToolbarAction action in the toolbar	139
Figure 44: Filter Definition panel with filter attributes	142
Figure 45: Editor for text-based attributes	144
Figure 46: Editor for string list attributes	145
Figure 47: Editor for option list attributes	145
Figure 48: Editor for choice list attributes	146
Figure 49: Editor for boolean attributes	146
Figure 50: Editor for date attributes (Date)	147
Figure 51: Editor for date attributes (Duration)	147
Figure 52: MasterRecipeCommentFilterAttribute in the Filter Definition panel	150
Figure 53: MaterialTypeFilterAttribute in the Filter Definition panel	158

Figure 54: WeighAndDispenseUnitProcedureFilterAttribute in the Filter Definition panel	159
Figure 55: BatchExpiryDateFilterAttribute in the Filter Definition panel	162
Figure 56: Help path for Production Execution Client	167
Figure 57: Help path for Production Management Client	168

Introduction

This documentation contains important information about how to configure and adapt a PharmaSuite system including information about actions that need to be performed in FactoryTalk® ProductionCentre.

The information is structured in the following sections:

VOLUME 1: STYLE AND LAYOUT

In this volume you will find information about standard styles and layouts that are defined for the user interface of PharmaSuite. You will also learn how to modify the elements of the user interface to fit your purposes.

It consists of the following chapters:

- Changing the General Appearance of PharmaSuite
- Creating and Adapting Forms
- Using Forms in the Production Execution Client
- Using Forms in the Production Management Client
- Adapting the Workflows of the Production Execution Client
- Adapting the Use Cases of the Production Management Client
- Changing the Layout of Forms

VOLUME 2: SEMANTIC (AFFECTING THE WHOLE SYSTEM)

In this volume you will find information about such system functionalities as field attributes, version management, services, audit trail, etc. You will also learn how you can adapt the mechanisms available in PharmaSuite to match your needs, as well as maintain and manage certain elements of the system, such as units of measures, signatures, or users.

It consists of the following chapters:

- Adapting and Adding Field Attributes
- Creating and Extending GUI Activities
- Creating Java Artifacts to Access Application Tables
- Configuring Flexible State Models
- Adapting Versioning Graphs
- Localizing Date and Time Representation

- Managing Services
- Adapting the Order Explosion Service
- Documenting the Change History of Order Definitions
- Managing Electronic Signatures and Access Rights
- Modifying and Adding Constraints
- Modifying and Adding Checks
- Managing Audit Trail
- Adding Units of Measure and Global Conversion Definitions
- Changing Number Generation Schemes
- Changing or Adding New Reports
- Changing or Adding New Labels
- Working with the Batch Record API
- Adapting the Export for Archive Process
- Adapting the Purge Process
- Defining the Risk Level for Exceptions
- Supporting External Authentication Systems
- Supporting External Exception Review
- Defining Units of Measure for Potency
- Adapting the About Dialog
- Adding New Scale Drivers
- Providing Localization to Support Different Locales

VOLUME 3: SEMANTIC (AFFECTING SPECIFIC AREAS)

In this volume you will find information about providing additional information in Recipe and Workflow Designer or Production Execution Client, etc. You will also learn how you can adapt windows in Recipe and Workflow Designer, Cockpit actions, or user documentation.

It consists of the following chapters:

- Adding Numeric Fields to the Database (page [11](#))
- Configuring the Universe of Recipe and Workflow Designer (page [13](#))
- Configuring the Setlist of Recipe and Workflow Designer (page [39](#))
- Configuring the Property Windows of Recipe and Workflow Designer (page [43](#))
- Configuring the Parameter Panel of Recipe and Workflow Designer (page [53](#))

- Managing the Layout of Recipe and Workflow Designer and Data Manager (page [59](#))
- Configuring the Expression Editor of Recipe and Workflow Designer and Data Manager (page [63](#))
- Using ERP BOMs in Recipe and Workflow Designer (page [75](#))
- Configuring the Initialization of Material Parameters in Recipe and Workflow Designer (page [77](#))
- Configuring Menu and Toolbar of Recipe and Workflow Designer (page [81](#))
- Configuring the Details Window of Data Manager - Work Center (page [87](#))
- Configuring the Change History of Data Manager - Work Center (page [91](#))
- Configuring FSMs for Equipment Properties (page [95](#))
- Providing Images for Equipment Classes in Data Manager (page [99](#))
- Providing Report Designs for (Template) Equipment Entities in Data Manager (page [101](#))
- Adding an Equipment Logbook Category Accessible by Phase Building Blocks (page [103](#))
- Retrieving Specific Information from the Equipment Logbook (page [107](#))
- Configuring Additional Purposes for the Equipment Type Technical Property Type (page [113](#))
- Adding a Workflow Type to Workflow Designer and the Production Execution Client (page [115](#))
- Managing Cockpit Actions of the Production Execution Client (page [117](#))
- Configuring Menu and Toolbar of the Production Response Client (page [137](#))
- Adding Filter Attributes to the Exception Dashboard of the Production Response Client (page [141](#))
- Material Handling for DCS (page [163](#))
- Adapting User Documentation (page [165](#))

VOLUME 4: CONFIGURATION FRAMEWORK

In this volume you will find information about the methods for managing application configurations, logging and debugging. You will also learn how to administer configuration keys, and which configuration keys are available in PharmaSuite. It consists of the following chapters:

- Managing Configurations
- Logging and Debugging

■ Configuration Keys of PharmaSuite

VOLUME 5: EXTENSION USE CASES

In this volume you will find step-by-step instructions for selected extension use cases. As a trained FactoryTalk ProductionCentre and PharmaSuite system integrator, you will learn how to adapt PharmaSuite.

It consists of the following chapters:

- Processing Materials with a Second Potency
- Adding the Second Potency Attribute to ERP BOMs
- Adding the Second Potency Function to the Expression Editor
- Managing Batches in the ConditionallyReleased Status
- Adapting the Approval Workflow of Master Recipes
- Adding the Material Balance Section to the Batch Report
- Displaying a Retest Date Column in the Overview Panel of the Exception Dashboard
- Adding the Print FDA Report Function to PharmaSuite Clients
- Adding the Print QA Report Function to Recipe and Workflow Designer
- Automatic Archiving and Purging of Specific Orders and Workflows
- Displaying Custom Columns in PharmaSuite's Audit Trail
- Using a Locale with Unicode Fonts

REFERENCE DOCUMENTS

Finally, the Reference Documents (page [171](#)) section provides a list of all the documentation that is referenced in this manual.

In Volume 5, each extension use case description comes with its own Reference Documents sections.

Intended Audience

This manual is intended for system engineers who implement customer-specific configurations and extensions to a PharmaSuite standard system.

The system engineers need to have a thorough working knowledge of FactoryTalk ProductionCentre, PharmaSuite, and other components relevant to the configuration or extension use cases.

Some use case require a fully set up PharmaSuite development environment and profound Java know-how.

Typographical Conventions

This documentation uses typographical conventions to enhance the readability of the information it presents. The following kinds of formatting indicate specific information:

Bold typeface	Designates user interface texts, such as <ul style="list-style-type: none">■ window and dialog titles■ menu functions■ panel, tab, and button names■ box labels■ object properties and their values (e.g. status).
<i>Italic typeface</i>	Designates technical background information, such as <ul style="list-style-type: none">■ path, folder, and file names■ methods■ classes.
CAPITALS	Designate keyboard-related information, such as <ul style="list-style-type: none">■ key names■ keyboard shortcuts.
Monospaced typeface	Designates code examples.

Extension and Naming Conventions

This section describes how to control efforts for migrating your PharmaSuite installation to upcoming versions. If you cannot observe the guidelines for technical reasons, please report the issue to your dedicated delivery team of Rockwell Automation or your system integrator.

PharmaSuite artifacts fall into two main groups: building blocks and PharmaSuite core artifacts.

- When you wish to modify a **building block**, use a copy of the building block. For details, see "Technical Manual Developing System Building Blocks" [A1] (page [171](#)).
- When you wish to modify a **PharmaSuite core artifact**, the extension strategy depends on the artifact itself.
Modify **DSX objects** and copy **all other PharmaSuite artifacts** (e.g. XML configurations for services).

In all cases, please follow the guidelines to control the migration effort (page [7](#)).

Guidelines to Control the Migration Effort

Please ensure that you observe the guidelines listed below.

1. Retain the published API
The published API of PharmaSuite is accessible via the PharmaSuite start page ("PharmaSuite-related Java Documentation" [C1] (page [171](#))).
 - If you adapt PharmaSuite on Java level, you must only use the published PharmaSuite Java API.
Published interfaces will only be changed if necessary. Changes to these interfaces and classes will be announced in future release notes.
 - Avoid using methods and classes that are not published, classes from the implementation package (...*impl*...), or Pnuts functions from any PharmaSuite subroutine.
These classes and functions may change in future versions of PharmaSuite without notification.

2. Rather modify than copy

Whenever you wish to extend/modify any object of PharmaSuite in Process Designer (DSX objects), just override the object, except for the following objects:

- For the **Application** object **Default Configuration**, apply the mechanism of nested configurations in order to reduce a potential migration effort (see chapter "Managing Configurations" in Volume 4 of the "Technical Manual Configuration and Extension").
- For **FSMs** (flexible state models), structural changes (i.e. changes to states and transitions) are not allowed in order to enable a later system migration. Modifications of semantic properties can be applied to the standard FSM. They do not impact a system migration.

When you migrate your PharmaSuite installation to another version, PharmaSuite Update and Migration displays a warning if a standard object was changed. Subsequently, you can decide whether to adapt your extension, ignore the changes delivered with the new version, or replace your extension with the new version (if applicable).

3. Mark your objects

When naming your artifacts (e.g. objects in Process Designer, classes, interfaces, methods, functions, building blocks), use specific prefixes for your objects. The main purpose of the naming conventions is to prevent naming conflicts with deliverables from other vendors or with other versions.

- Define and make use of a vendor code consisting of up to three uppercase letters as prefix (e.g. MYC for the My Company vendor code). The **X_** and **RS_** prefixes are reserved for PharmaSuite and PharmaSuite-specific product building blocks, respectively.
- Do not reuse any of the prefixes of PharmaSuite objects in Process Designer in order to avoid conflicts during migration.
- This guideline also applies to UDA definitions and column names of application tables.
- Additional conventions apply to building blocks (page 9).

If you do not observe the guidelines, an update process during system migration may fail due to conflicts.

Building Block-specific Conventions

Besides the general conventions (page 7), additional conventions apply to building blocks related to vendor code, version number, and length restrictions:

1. Vendor code

- It must be appended to the name of a phase or parameter class used in the UI (enclosed in round brackets).
- It must be used as prefix for the AT definitions.
- The package name must also contain a vendor reference. You can either use the vendor code or write out the company's full name.

2. Version number

- The version number consists of two components, an integral part to refer to a major version and a fractional part to refer to a minor version, e.g. 2.1.
- It must be appended to the name of the phase or parameter class used in the UI and to the base name used for the generated artifacts.
- For the UI, the version number is enclosed in square brackets, e.g. [2.1].
- Internal names must not contain brackets and dots, since Java does not allow the usage of these characters. Therefore, the last four characters are reserved for the version number, with digits 1 and 2 representing the major version and digits 3 and 4 representing the minor version. The format is xxyy, e.g. 0201 for version [2.1], 0100 for version [1.0], or 0113 for version [1.13].

3. Length restrictions

- The maximum length of the **name** of a phase or parameter class used in the UI is 64 characters.
- The maximum length of the **base name** of a phase building block or parameter class is 18 characters (14 for the name, 4 for the version).

Examples:

■ Hello World phase of My Company with vendor code MYC in version 2.1

```
<Name>Hello World Phase (MYC) [2.1]</Name>
<PhaseLibBaseName>HelloWorld0201</PhaseLibBaseName>
<ATDefinitionPrefix>MYC</ATDefinitionPrefix>
<PackageName>com.mycompany.phase.helloworld</PackageName>
```

■ My Parameter parameter class of My Company with vendor code MYC in version 1.0

```
<Name>My Parameter (MYC) [1.0]</Name>
<ParamClassBaseName>MyParam0100</ParamClassBaseName>
<ATDefinitionPrefix>MYC</ATDefinitionPrefix>
<PackageName>com.mycompany.parameter.myparam</PackageName>
```

Oracle Database Data Types: **varchar2** and **nvarchar2**

When you define text fields for FactoryTalk ProductionCentre application tables or UDAs, you should be aware that there are significant differences between the **varchar2** and **nvarchar2** data types used by Oracle databases.

- The maximum field length for both database data types is restricted to 4000 bytes.
- A field of the **nvarchar2** data type works as expected: For a 4000 characters text field, you can only insert the maximum of 2000 2-byte UTF8 characters or 1300 3-byte UTF8 characters. PharmaSuite text input fields check the number of bytes.
- For an Oracle 11 database, a field of the **varchar2** data type can only handle 1000 2-byte UTF8 characters or 667 3-byte UTF8 characters. In this case, PharmaSuite text input fields check the byte length and prevent the database exception "ORA-01704: string literal too long".

TIP

Previous versions of FactoryTalk ProductionCentre (prior to 9.3) and PharmaSuite (prior to 5.0) contained **varchar2** database data type definitions instead of **nvarchar2**. Therefore a migrated PharmaSuite system may contain **varchar2** definitions and the maximum length of a **varchar2** database field may be defined in maximum number of **bytes**. In this case, if a text does not only contain 1-byte UTF8 characters, the **byte** length is greater than the **char** length and can cause a "value too large exception" (ORA-01401: inserted value too large for column, ORA-12899: value too large for column).

For this reason, we recommend to migrate all **varchar2** fields to **nvarchar2** fields. A migrated FactoryTalk ProductionCentre database should have the same database schema as the database of a newly installed FactoryTalk ProductionCentre system and only contain **nvarchar2** database data type field definitions.

Adding Numeric Fields to the Database

This section describes restrictions that apply to numeric data types.

Generally, we do not recommend to use the **Decimal** type of FactoryTalk ProductionCentre when creating ATRows, phase building blocks, etc.

Although the decimal values are represented by BigDecimals in Java, when using the PharmaSuite class generators, they are stored as Float values in the database.

This may result in a loss of precision which is especially critical for values used in calculations.

Please use a unitless MeasuredValue instead and get its BigDecimal value by its **getValue()** method.

TIP

When using a MeasuredValue, please be aware of its range limitation (15 integral, 9 fractional digits). Please refer to the "Recipe and Workflow Designer User Documentation" [C2] (page [171](#)) and the "Data Manager User Documentation" [C3] (page [171](#)).

Configuring the Universe of Recipe and Workflow Designer

This section contains general information about the Universe (page 13) and related topics, provides an introduction into the Universe configuration (page 16), and guidelines how to adjust the configuration (page 35).

To configure the Universe, you have to be familiar with

- XML and XSD. A basic know-how is sufficient.
- PharmaSuite database
For information on the PharmaSuite database, please refer to the FactoryTalk ProductionCentre documentation. In particular, see section "Data Dictionary Overview" in "Process Designer Online Help" [B1] (page 171).
- Table and column names
- Java Beans encapsulating the data to be read from the database
- Data Dictionary Manager (see chapter "Adapting and Adding Field Attributes" in Volume 2 of the "Technical Manual Configuration and Extension" [A3] (page 171)).
- Process Designer

What Is the Universe?

The Universe is a tool that allows a user to access the complete set of objects stored in the PharmaSuite database. An object can be any object that can be used in Recipe and Workflow Designer, e.g. material, phase, master recipe, or master workflow.

What Is a Universe Object Type?

A Universe Object Type is one logical part of the Universe, representing a subset of similar objects of the same class, like material, equipment, or parameter.

What Is a Universe Object?

A Universe Object is a concrete instance of one of the objects that build the Universe, e.g. the specific material with name **MyMaterial**.

What Is the Universe Explorer?

The Universe Explorer is a user interface dialog which allows a user to access the data of the Universe. The Universe Explorer can be used to search, filter, and then select objects to work with. It consists of 4 major parts:

- **Object Type List**
This list displays the set of currently accessible Universe Object Types. Only one item of the list can be selected at a time. This is the first, coarse-grained filter that is always applied to restrict the set of Universe Objects displayed in the Result List.
- **Quick Search**
This optional filter can be applied to restrict the set of Universe Objects displayed in the Result List table by entering arbitrary text. This text is matched against the values stored in the objects' properties, as displayed in the columns of the Result List.
- **Quick Filter**
This is another optional filter that can be applied to further restrict the set of Universe Objects displayed in the result table by selecting values of a set of pre-defined object properties. The property values selected are matched against the appropriate values stored in the objects' properties, according to the columns displayed in the Result List.
- **Result List**
This table displays all Universe Objects that match the current filter criteria and can therefore be selected in Recipe and Workflow Designer.

In the figure below, you can see the Universe Explorer with Material as selected Universe Object Type. No further filters are selected, thus all materials stored in the database are displayed in the Result List.

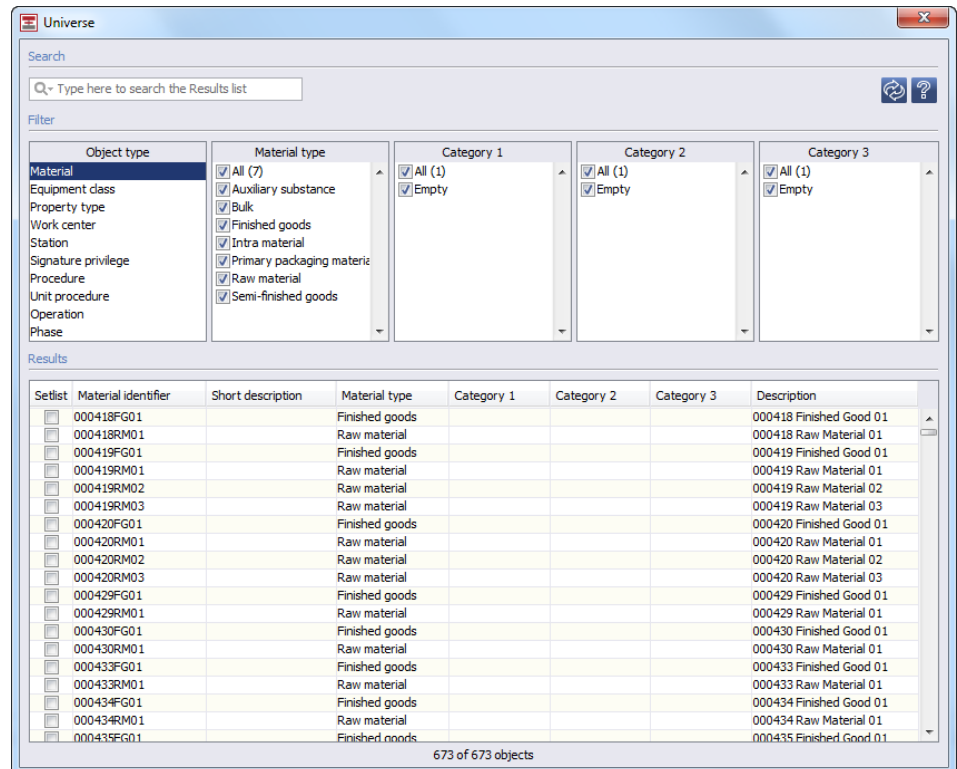


Figure 1: Universe Explorer with no optional filters applied

Applying the different filter criteria allows to conveniently restrict the set of objects contained in the Result List and to find the object you are looking for.

The figure again shows the Universe Explorer with material selected as Universe Object Type, but with Quick Search and Quick Filter filters applied. Therefore, only raw materials containing the substring S88 in one of their properties are displayed in the Result List. In this example, there is only one material left that matches the current filter.

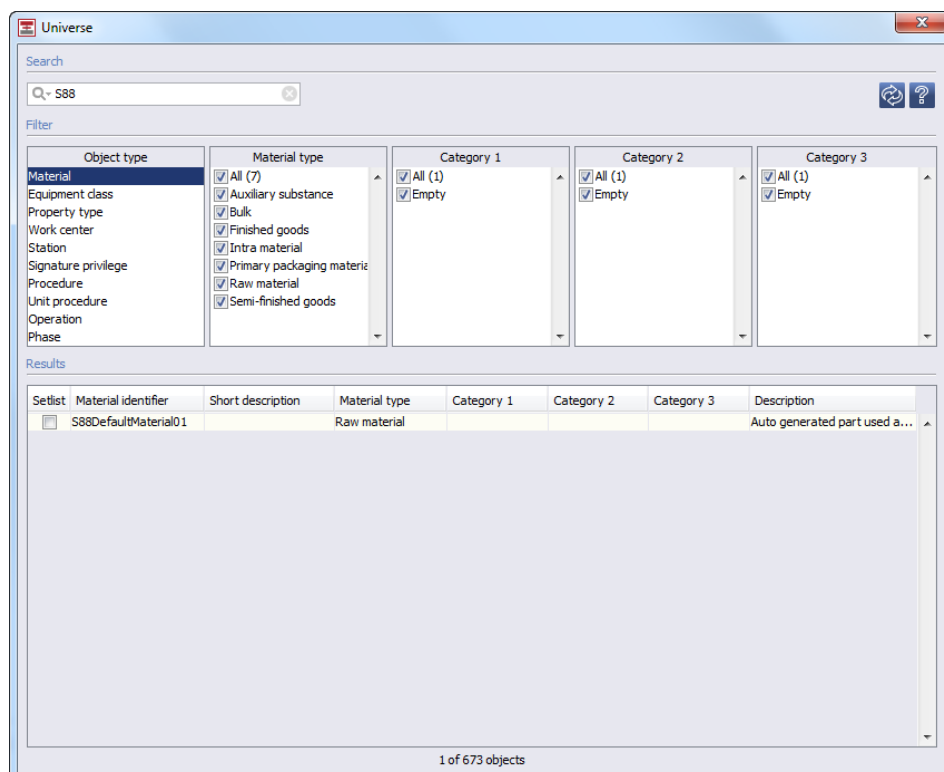


Figure 2: Universe Explorer with all filters applied

What Is the Universe Configuration?

The Universe Explorer is a fully configurable user interface dialog. Which data is read from the database at all, the set of Universe Object Types offered for selection, the Universe Object properties displayed in the Result List and up to four of these Universe Object Properties used for the Quick Filter - all of these characteristics can be configured through the Universe Configuration, without touching any code.

The Universe Configuration is an XML document, that

- describes the view to the Universe the user will see in Recipe and Workflow Designer,
- provides the information to access the required data from the database, and
- describes the representation of the UI for each Universe Object Type.

You can find the XML document of the Universe Configuration in the PharmaSuite database as a **List** object.

The Universe Configuration allows to configure:

- the list of Universe Object Types
- criteria of the Quick Filter (for each Universe Object Type)
- columns of the Result List (for each Universe Object Type).

What Is the Open Dialog?

The **Open** dialog provides the functionality to open master recipes, master workflows, or other building blocks. In contrast to the Universe Explorer (page 14), the **Open** dialog only offers access to a restricted set of objects (that can be opened by Recipe and Workflow Designer). However, the way it looks and works is still very similar to the Universe Explorer. Therefore, it is based on the same elements and can be configured in the same way.

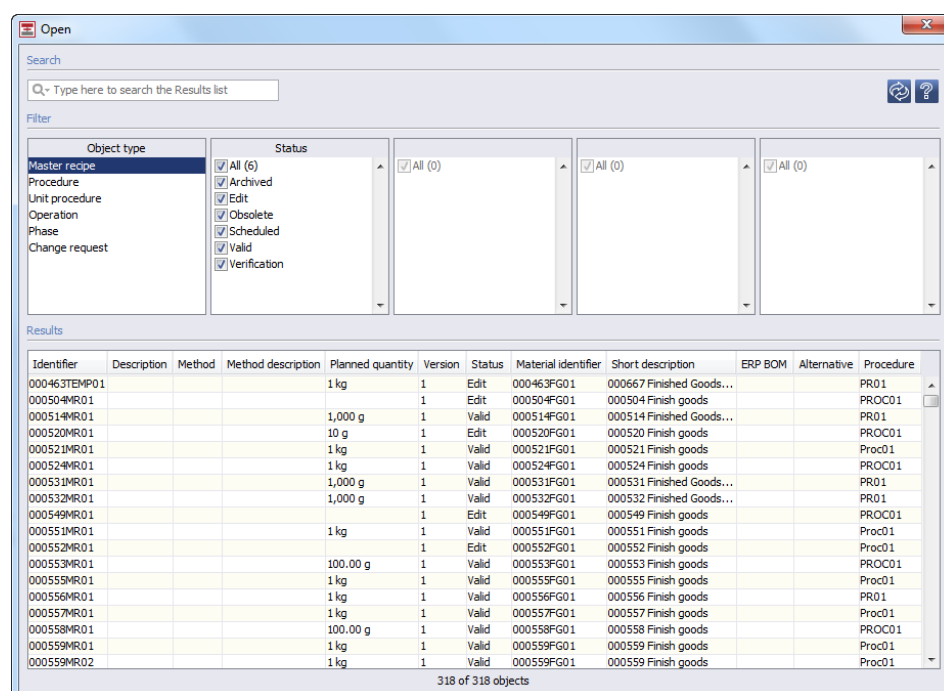


Figure 3: Open dialog, working the same way as the Universe Explorer

What Is the Select Material Dialog?

The **Select material** dialog provides the functionality to select materials to be used when creating master recipes. It looks and works very similar to the Universe Explorer (page 14). Therefore, it is based on the same elements and can be configured in the same way.

What Is the Data Dictionary?

The Data Dictionary (DD) is the central repository of all Java bean classes used in the UI of PharmaSuite. It stores information about the beans and each of their properties.

You can use the **Data Dictionary Manager** to view and manage contents of the Data Dictionary (see chapter "Adapting and Adding Field Attributes" in Volume 2 of the "Technical Manual Configuration and Extension" [A3] (page 171).

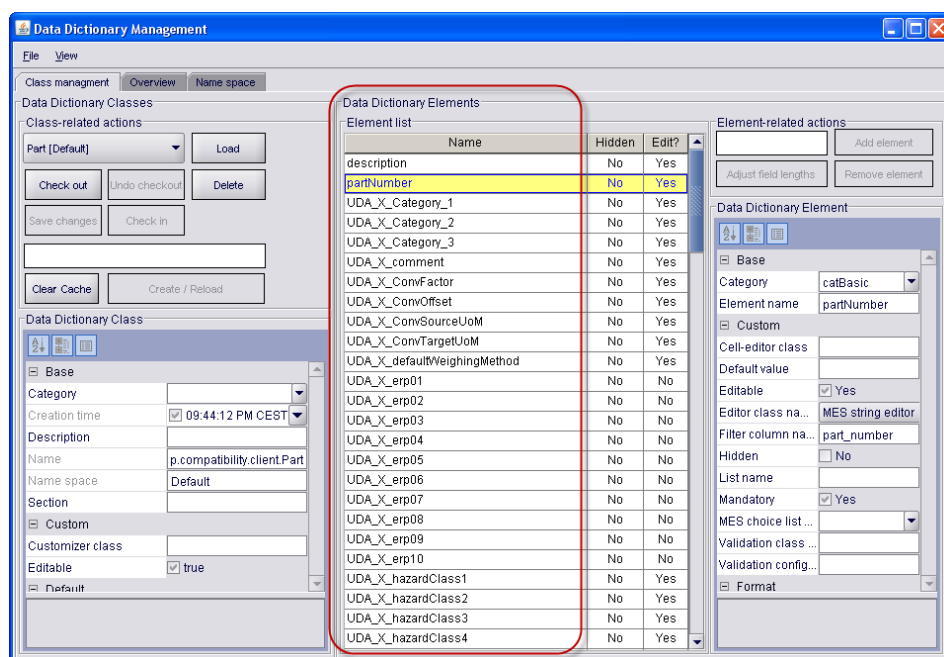


Figure 4: Data Dictionary Manager displaying all properties of a Java Bean class

Introduction to Universe Configuration

Before you adjust the XML file holding the Universe Configuration, please read this section carefully, because it provides a solid background on this topic and various configuration variants.

- Step 1: Simple Universe Configuration for one Universe Object Type (page 19)
- Step 2: Universe Configuration for one Universe Object Type with database information inside (page 21)
- Step 3: Universe Configuration for one Universe Object Type with some database information (page 23)
- Step 4: Complex Universe Configuration for a single Universe Object Type (page 24)
- Step 5: Universe Configuration for multiple Universe Object Types (page 26)
- Elements of the Universe Configuration (page 31)

Step 1: Simple Universe Configuration for One Universe Object Type

The best introduction to the Universe Configuration is a simple example. Look at following XML document.

```
<Universe>
  <UniverseObject name="material">
    <DBTable boundClass="com.datasweep.compatibility.client.Part">
      <DatabaseColumns>
        <DBColumn propertyName="key" />
        <DBColumn propertyName="partNumber" identifier="number" />
        <DBColumn propertyName="description" />
        <DBColumn propertyName="UDA_X_materialType"
          identifier="type" />
        <DBColumn propertyName="category" />
      </DatabaseColumns>
    </DBTable>
    <FilterCriteria>
      <Criterion identifier="type" />
      <Criterion identifier="category" />
    </FilterCriteria>
    <GridColumns>
      <GridColumn identifier="number" />
      <GridColumn identifier="description" />
      <GridColumn identifier="type" />
      <GridColumn identifier="category" />
    </GridColumns>
  </UniverseObject>
</Universe>
```

The XML document contains only one Universe Object Type, the **material** type. Use a meaningful name for the object name. Each Universe Object Type needs a database table with a bound class. The bound class is the Java Bean class that stores the records read from the database. If you look at this bound class in the "PharmaSuite-related Java Documentation" [C1] (page 171) or in the Data Dictionary Manager, you can see all available properties you can use in the column list (see figure in section "What Is the Data Dictionary?" (page 17)). The key column is mandatory in the list of database columns.

In the **FilterCriteria** and **GridColumns** XML sections you can add the properties to the list you wish to see on the user interface. The identifier must be equal to the identifier of the column list (element <DBColumn>). If a database column has no explicit identifier, the property name is used instead as implicit identifier.

The result of our simple configuration is shown below:

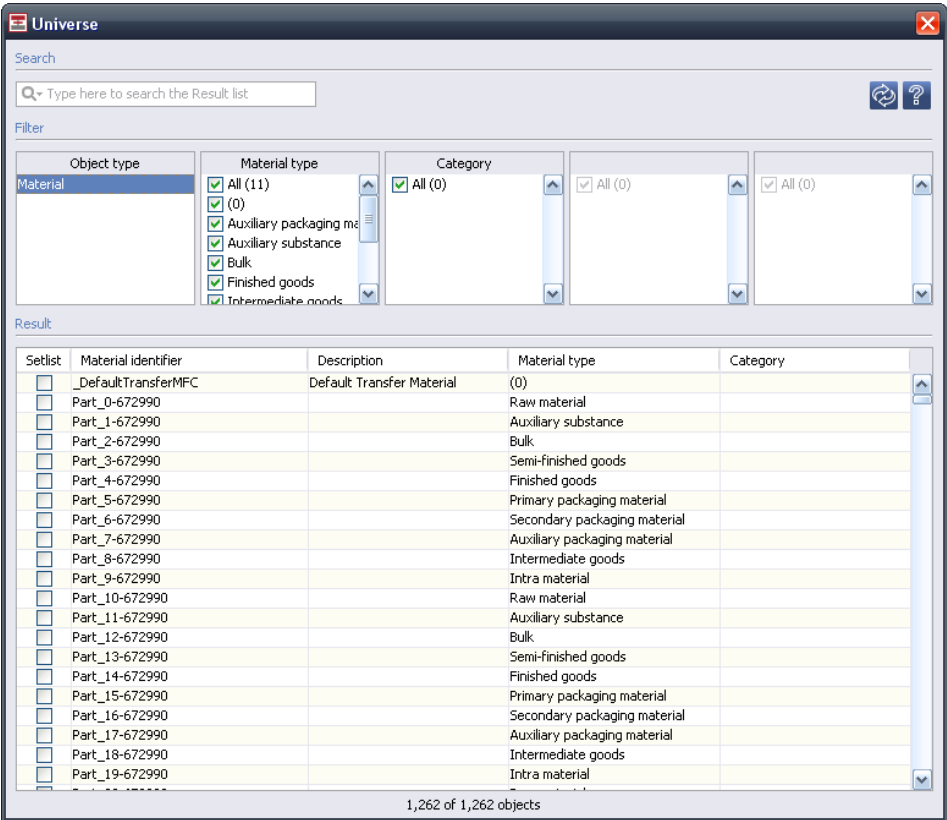


Figure 5: Universe Explorer for the simple configuration

In the XML document of this simple example you cannot find any database information. How is it possible, that the system could fetch the data from the database? The answer is that all database information must be available in the Data Dictionary of the bound class of the Universe Object Type, see figure below. The DD class provides the database table name, the UDA table name and the key column name. Each DD element provides the database column name (also called **Filter column name**).

TIP
Please add this information to the Data Dictionary if it is missing.

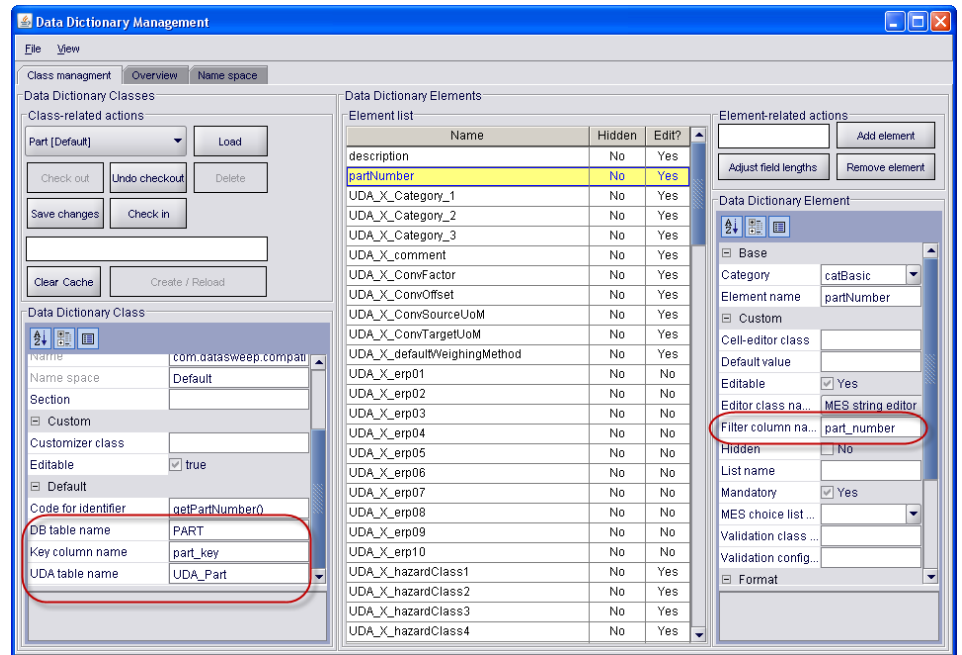


Figure 6: Data Dictionary Manager displaying all database information that is required

Step 2: Universe Configuration for One Universe Object Type with Database Information Inside

The difference of this example compared to Step 1 (page 19) is that the database information is given in the XML document. This is useful if the Data Dictionary is not available, during testing, or if you wish to overwrite the settings of the Data Dictionary. Look at following XML document; it works without any database information from the Data Dictionary.

```
<Universe>
  <UniverseObject name="material">
    <DBTable boundClass="com.datasweep.compatibility.client.Part"
      tableName="PART" orderBy="PART.part number"
      whereCondition="UDA_Part.X_materialType_I > 0">
      <DatabaseColumns>
        <DBColumn propertyName="key" isKey="true"
          columnName="part_key"/>
        <DBColumn propertyName="partNumber"
          columnName="part_number" identifier="number" />
        <DBColumn propertyName="description" />
        <DBColumn propertyName="category" />
        <DBColumn propertyName="UDA_X_materialType"
          columnName="X_materialType_I" identifier="type" />
      </DatabaseColumns>
    </DBTable>
    <FilterCriteria>
      <Criterion identifier="type" />
      <Criterion identifier="category" />
    </FilterCriteria>
    <GridColumns>
      <GridColumn identifier="number" />
    </GridColumns>
  </UniverseObject>
</Universe>
```

```

    <GridColumn identifier="description" />
    <GridColumn identifier="type" />
    <GridColumn identifier="category" />
  </GridColumn>
</GridColumns>
</UniverseObject>
</Universe>

```

If you look at the red colored XML attributes, you find the database-internal information which was kept in the Data Dictionary in Step 1 (page 19). Besides, you can find new key words like **whereCondition** and **orderBy**, which cannot be stored in the Data Dictionary, because they are related to this special user interface element and not of common interest.

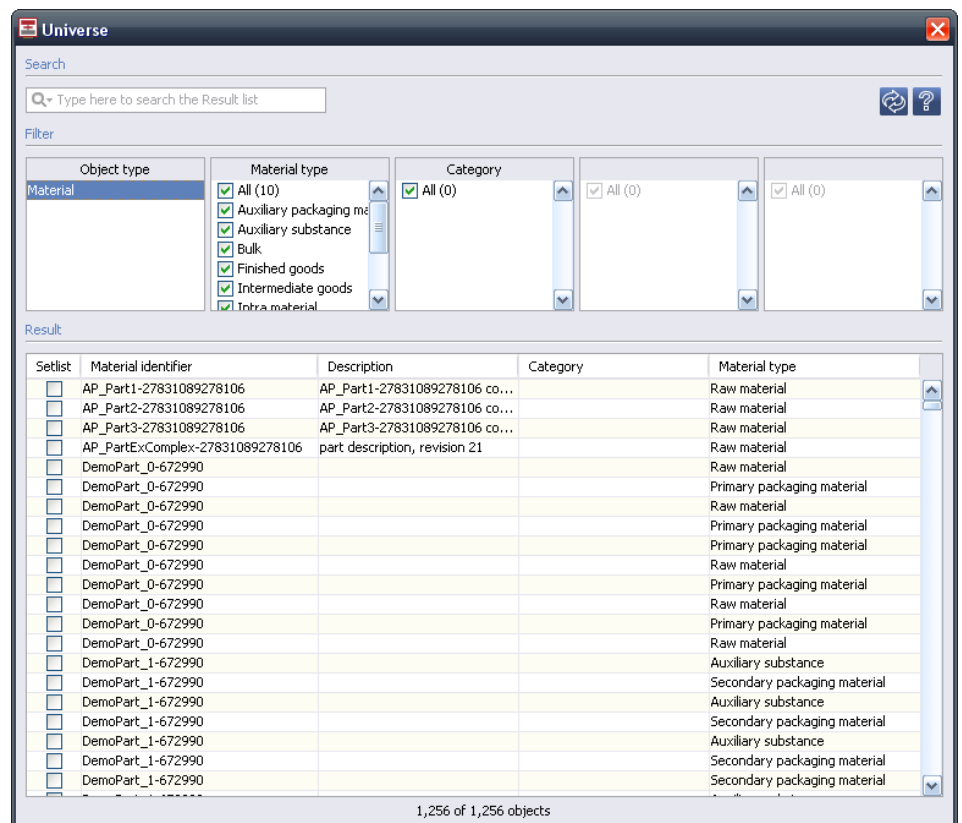


Figure 7: Universe Explorer for a simple configuration including database information

The result is similar to Step 1 (page 19) that did not contain database information. It differs in

- the initial sort order of the elements in the table and
- the order of some of the columns of the result table,

as defined in the configuration.

NOTE

Due to an additional **whereCondition**, some objects are omitted. Thus, when selecting the Universe Object Type **material** in this scenario, the complete (i.e. unfiltered) result set does only contain a pre-filtered subset of all Materials stored in the database.

Step 3: Universe Configuration for One Universe Object Type with Some Database Information

Now we combine Step 1 (page 19) and Step 2 (page 21). We intend to avoid any database information in XML, but use constructs like **whereCondition** and **orderBy**. The solution is to use placeholders as you can see in the following XML document.

```
<Universe>
  <UniverseObject name="material">
    <DBTable boundClass="com.datasweep.compatibility.client.Part"
      orderBy="{number}" whereCondition="{type} > 0">
      <DatabaseColumns>
        <DBColumn propertyName="key" />
        <DBColumn propertyName="partNumber" identifier="number" />
        <DBColumn propertyName="description" />
        <DBColumn propertyName="UDA_X_materialType"
          identifier="type" />
        <DBColumn propertyName="category" />
      </DatabaseColumns>
    </DBTable>
    <FilterCriteria>
      <Criterion identifier="type" />
      <Criterion identifier="category" />
    </FilterCriteria>
    <GridColumn>
      <GridColumn identifier="number" />
      <GridColumn identifier="description" />
      <GridColumn identifier="type" />
      <GridColumn identifier="category" />
    </GridColumn>
  </UniverseObject>
</Universe>
```

If you look at the red XML attributes you find placeholders in curly brackets (**number**, **type**). These placeholders are the property names which will be replaced by the real column names (if the column name is available in the Data Dictionary).

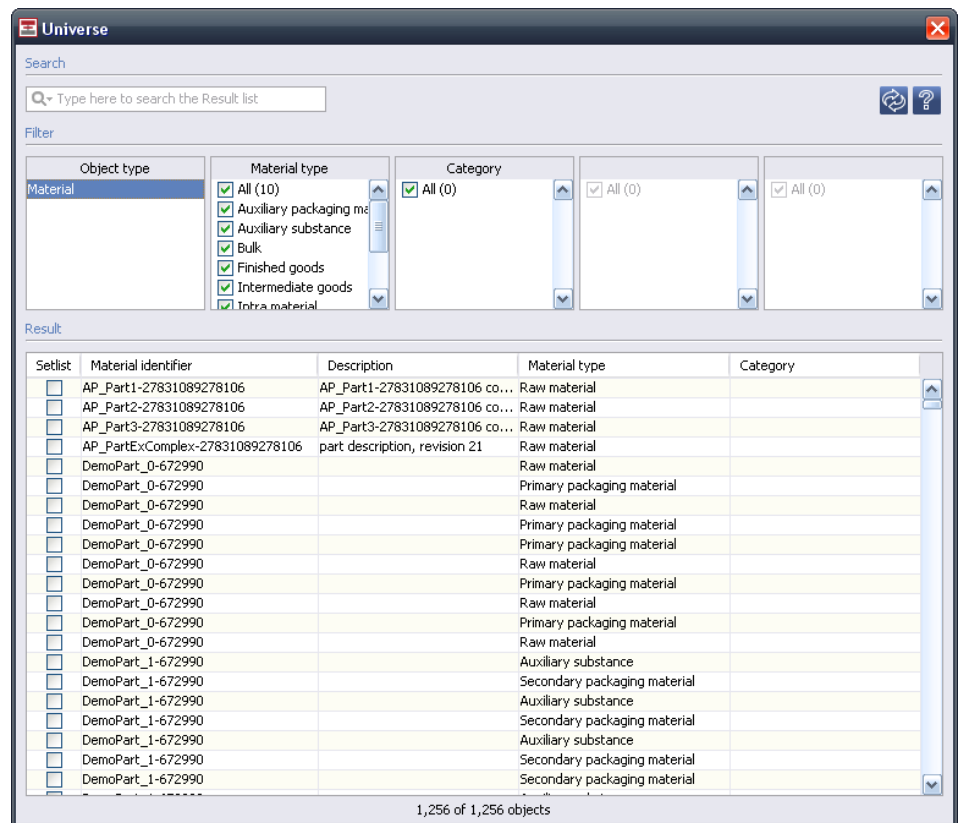


Figure 8: Universe Explorer for a simple configuration including placeholders for additional database information

Except for the sort order of some columns, the result is identical to Step 2 (page 21).

Step 4: Complex Universe Configuration for a Single Universe Object Type

In each of the preceding steps, the Universe Object Type maps exactly to one database table and to one Java Bean class. This scenario is not always given. Now, we build a view to more than one table. The main table (here: **ProcessBillOfMaterials**) is joined to some other tables of related objects (here: **Part**, **ObjectState**, **Status**).

```
<Universe>
  <UniverseObject name="bom">
    <DBTable boundClass="com.datasweep.compatibility.client.ProcessBillOfMaterials">
      <DatabaseColumns>
        <DBCColumn propertyName="key" />
        <DBCColumn propertyName="bomName" identifier="name" />
        <DBCColumn propertyName="description" />
        <DBCColumn propertyName="extendedRevision" identifier="revision" />
        <DBCColumn propertyName="UDA_X_method" identifier="method" />
        <DBCColumn propertyName="state" isState="true"
          titleMsgID="versionState ShortLabel" />
        <DBCColumn propertyName="partProduced" />
      </DatabaseColumns>
    <JoinTable boundClass="com.datasweep.compatibility.client.Part"
      joinCondition="{partProduced} = {partKey}">
      <DatabaseColumns>
        <DBCColumn propertyName="key" identifier="partKey" />
      </DatabaseColumns>
    </JoinTable>
  </UniverseObject>
</Universe>
```

```

        <DBColumn propertyName="partNumber" />
        <DBColumn propertyName="description"
            identifier="partDescription" />
        <DBColumn propertyName="UDA_X_materialType"
            identifier="materialType" />
    </DatabaseColumns>
</JoinTable>
</DBTable>
<FilterCriteria>
    <Criterion identifier="method" />
    <Criterion identifier="state" />
    <Criterion identifier="materialType" />
</FilterCriteria>
<GridColumns>
    <GridColumn identifier="name" />
    <GridColumn identifier="description" />
    <GridColumn identifier="revision" />
    <GridColumn identifier="state" />
    <GridColumn identifier="method" />
    <GridColumn identifier="partNumber" />
    <GridColumn identifier="partDescription" />
    <GridColumn identifier="materialType" />
</GridColumns>
</UniverseObject>
</Universe>

```

Look at the **JoinTable** XML section. It is necessary to read data from other tables than the main database table. When changing the Universe Configuration, you must be familiar with the database and know how to join the tables.

Another example is the **state** database column. It is a dummy column to get the object status using a complex database query, which is built automatically by the system. Make sure the **isState** XML attribute is set to true. We recommend to add the message ID for the column title manually because the state is not managed in the Data Dictionary of the bound class (see **titleMsgID** XML attribute).

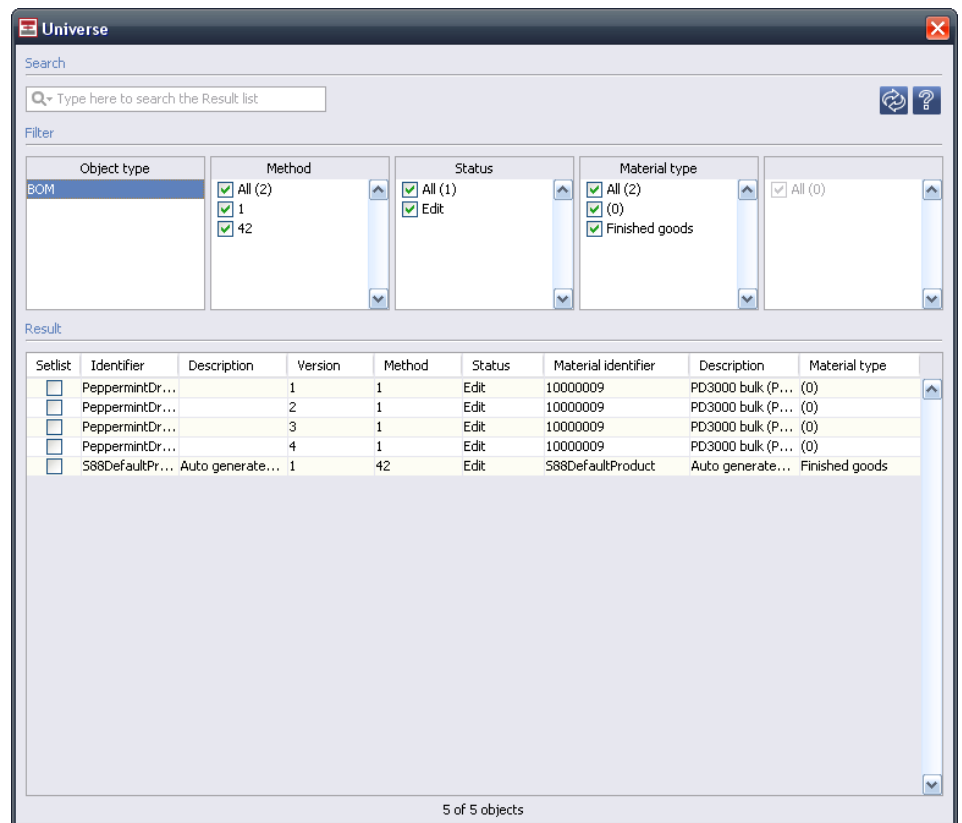


Figure 9: Universe Explorer for a more complex configuration, still containing a single object type

Step 5: Universe Configuration for Multiple Universe Object Types

When you are familiar with the previous steps, it will be easy to extend the Universe Configuration to more than one object type. Look at the following example, which uses 3 different Universe Object Types.

```
<!-- Test configuration of the universe dialog. -->
<Universe xsi:noNamespaceSchemaLocation="universe.xsd"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

  <!-- Simple example, one table with UDA, without joins and status -->
  <UniverseObject name="material">
    <DBTable boundClass="com.datasweep.compatibility.client.Part"
      orderBy="{number}" whereCondition="{type} > 0">
      <DatabaseColumns>
        <DBCColumn propertyName="key" />
        <DBCColumn propertyName="partNumber" identifier="number" />
        <DBCColumn propertyName="description" />
        <DBCColumn propertyName="category" />
        <DBCColumn propertyName="UDA_X materialType" identifier="type" />
        <DBCColumn propertyName="UDA_X_erp01" identifier="segmentA" />
        <DBCColumn propertyName="UDA_X_erp02" identifier="segmentB" />
        <DBCColumn propertyName="UDA_X_erp03" identifier="segmentC" />
      </DatabaseColumns>
    </DBTable>
  </UniverseObject>
</Universe>
```



```

<FilterCriteria>
  <Criterion identifier="type" />
  <Criterion identifier="segmentA" />
  <Criterion identifier="segmentB" />
  <Criterion identifier="segmentC" />
</FilterCriteria>

<GridColumns>
  <GridColumn identifier="number" />
  <GridColumn identifier="description" />
  <GridColumn identifier="type" />
  <GridColumn identifier="segmentA" />
  <GridColumn identifier="segmentB" />
  <GridColumn identifier="segmentC" />
</GridColumns>

</UniverseObject>

<!-- Same example with explicit database information -->
<UniverseObject name="material">
  <DBTable boundClass="com.datasweep.compatibility.client.Part"
    tableName="PART" orderBy="PART.part_number"
    whereCondition="UDA_Part.X_materialType_I > 0">
    <DatabaseColumns>
      <DBCColumn propertyName="key" isKey="true" columnName="part_key" />
      <DBCColumn propertyName="partNumber" columnName="part_number"
        identifier="number" />
      <DBCColumn propertyName="description" />
      <DBCColumn propertyName="category" />
      <DBCColumn propertyName="UDA_X_materialType" columnName="X_materialType_I"
        identifier="type" />
      <DBCColumn propertyName="UDA_X_erp01" identifier="segmentA" />
      <DBCColumn propertyName="UDA_X_erp02" identifier="segmentB" />
      <DBCColumn propertyName="UDA_X_erp03" identifier="segmentC" />
    </DatabaseColumns>
  </DBTable>

  <FilterCriteria>
    <Criterion identifier="type" />
    <Criterion identifier="segmentA" />
    <Criterion identifier="segmentB" />
    <Criterion identifier="segmentC" />
  </FilterCriteria>

  <GridColumns>
    <GridColumn identifier="number" />
    <GridColumn identifier="description" />
    <GridColumn identifier="type" />
    <GridColumn identifier="segmentA" />
    <GridColumn identifier="segmentB" />
    <GridColumn identifier="segmentC" />
  </GridColumns>

</UniverseObject>

<!-- More complex example, one table with UDA, joins, status and table alias -->
<UniverseObject name="bom">
  <DBTable boundClass="com.datasweep.compatibility.client.ProcessBillofMaterials"
    aliasTableName="pbom">
    <DatabaseColumns>
      <DBCColumn propertyName="key" />
      <DBCColumn propertyName="bomName" identifier="name" />
    </DatabaseColumns>
  </DBTable>

```

```

        <DBColumn propertyName="description" />
        <DBColumn propertyName="extendedRevision" identifier="revision" />
        <DBColumn propertyName="UDA_X_method" identifier="method" />
        <DBColumn propertyName="partProduced" />
    </DatabaseColumns>
    <JoinTable boundClass="com.datasweep.compatibility.client.Part"
        aliasTableName="bomMat" joinCondition="{partProduced} = {partKey}">
        <DatabaseColumns>
            <DBColumn propertyName="key" identifier="partKey" />
            <DBColumn propertyName="partNumber" />
            <DBColumn propertyName="description" identifier="partDescription" />
            <DBColumn propertyName="UDA_X_materialType" identifier="materialType" />
        </DatabaseColumns>
    </JoinTable>
</DBTable>

<FilterCriteria>
    <Criterion identifier="method" />
    <Criterion identifier="materialType" />
</FilterCriteria>

<GridColumns>
    <GridColumn identifier="name" />
    <GridColumn identifier="description" />
    <GridColumn identifier="revision" />
    <GridColumn identifier="method" />
    <GridColumn identifier="partNumber" />
    <GridColumn identifier="partDescription" />
    <GridColumn identifier="materialType" />
</GridColumns>
</UniverseObject>

<!-- More complex example, one table with UDA, joins and status -->
<UniverseObject name="masterRecipe">
    <DBTable boundClass="com.datasweep.compatibility.client.MasterRecipe"
        aliasTableName="mr">
        <DatabaseColumns>
            <DBColumn propertyName="key" />
            <DBColumn propertyName="name" />
            <DBColumn propertyName="description" />
            <DBColumn propertyName="extendedRevision" identifier="revision" />
            <DBColumn propertyName="UDA_X_method" identifier="method" />
            <DBColumn propertyName="state" isState="true"
                titleMsgID="versionState_ShortLabel" />
            <DBColumn propertyName="route" />
            <DBColumn propertyName="processBom" />
        </DatabaseColumns>
    <JoinTable boundClass="com.datasweep.compatibility.client.Route"
        joinCondition="{route} = {routeKey}">
        <DatabaseColumns>
            <DBColumn propertyName="key" identifier="routeKey" />
            <DBColumn propertyName="name" identifier="routeName" />
            <DBColumn propertyName="description" identifier="routeDescription" />
            <DBColumn propertyName="UDA_X_method" identifier="routeMethod" />
        </DatabaseColumns>
    </JoinTable>
    <JoinTable
        boundClass="com.datasweep.compatibility.client.ProcessBillofMaterials"
        joinCondition="{processBom} = {bomKey}">
        <DatabaseColumns>
            <DBColumn propertyName="key" identifier="bomKey" />
            <DBColumn propertyName="bomName" />

```

```

        <DBColumn propertyName="description" identifier="bomDescription" />
        <DBColumn propertyName="UDA_X_method" identifier="bomMethod" />
        <DBColumn propertyName="partProduced" />
    </DatabaseColumns>
    <JoinTable boundClass="com.datasweep.compatibility.client.Part"
        joinCondition="{partProduced} = {partKey}" aliasSuffix="produced">
        <DatabaseColumns>
            <DBColumn propertyName="key" identifier="partKey" />
            <DBColumn propertyName="partNumber" />
            <DBColumn propertyName="description" identifier="partDescription" />
            <DBColumn propertyName="UDA_X_materialType" identifier="materialType" />
        </DatabaseColumns>
    </JoinTable>
</JoinTable>
</DBTable>

<FilterCriteria>
    <Criterion identifier="method" />
    <Criterion identifier="state" />
    <Criterion identifier="materialType" />
</FilterCriteria>

<GridColumns>
    <GridColumn identifier="name" />
    <GridColumn identifier="description" />
    <GridColumn identifier="revision" />
    <GridColumn identifier="state" />
    <GridColumn identifier="method" />
    <GridColumn identifier="routeName" />
    <GridColumn identifier="routeDescription" />
    <GridColumn identifier="routeMethod" />
    <GridColumn identifier="bomName" />
    <GridColumn identifier="bomDescription" />
    <GridColumn identifier="bomMethod" />
    <GridColumn identifier="partNumber" />
    <GridColumn identifier="partDescription" />
    <GridColumn identifier="materialType" />
</GridColumns>
</UniverseObject>

<!-- example for AT-table -->
<UniverseObject name="basicPhase">
    <DBTable boundClass="com.rockwell.mes.services.s88.impl.library.MESPhaseLib"
        aliasTableName="phLib"
        whereCondition="X_navigatorVisible_Y = 1 AND X_internal_Y = 0">
        <DatabaseColumns>
            <DBColumn propertyName="key" />
            <DBColumn propertyName="name" />
        </DatabaseColumns>
    </DBTable>

    <FilterCriteria>
        <Criterion identifier="name" />
    </FilterCriteria>

    <GridColumns>
        <GridColumn identifier="name" />
    </GridColumns>

</UniverseObject>

<!-- second example for AT-table -->

```

```

<UniverseObject name="parameterClass">
  <DBTable
    boundClass="com.rockwell.mes.services.s88.impl.library.MESParameterClass">
    <DatabaseColumns>
      <DBColumn propertyName="key" />
      <DBColumn propertyName="name" />
      <DBColumn propertyName="description" />
    </DatabaseColumns>
  </DBTable>

  <FilterCriteria>
    <Criterion identifier="name" />
  </FilterCriteria>

  <GridColumns>
    <GridColumn identifier="name" />
    <GridColumn identifier="description" />
  </GridColumns>
</UniverseObject>

<!-- example for AT-table with status -->
<UniverseObject name="equipmentClass">
  <DBTable
    boundClass="com.rockwell.mes.services.s88equipment.impl.MESS88EquipmentClass"
    orderBy="{identifier}" whereCondition="state_name != 'Archived'"
    aliasTableName="eqCl">
    <DatabaseColumns>
      <DBColumn propertyName="key" />
      <DBColumn propertyName="identifier" />
      <DBColumn propertyName="shortDescription" />
      <DBColumn propertyName="description" />
      <DBColumn propertyName="equipmentLevelAsValue"
        columnName="X_equipmentLevel_I" />
      <DBColumn propertyName="state" isState="true"
        fsmRelationShip="s88EquipmentClassStatus" titleMsgID="state_ShortLabel"
        stateJoinColumnName="X_stateProxy_231" />
    </DatabaseColumns>
  </DBTable>
  <FilterCriteria>
    <Criterion identifier="equipmentLevelAsValue" />
    <Criterion identifier="state" />
  </FilterCriteria>
  <GridColumns>
    <GridColumn identifier="identifier" />
    <GridColumn identifier="shortDescription" />
    <GridColumn identifier="description" />
    <GridColumn identifier="equipmentLevelAsValue" />
    <GridColumn identifier="state" />
  </GridColumns>
</UniverseObject>
</Universe>

```

This more comprehensive example contains 5 different Universe Object Types, one “flat” (twice **material**, once without and once with database information), one “deep” (**ProcessBillOfMaterials**, with join and status), one “very deep” (**MasterRecipe**, with multiple and even nested joins), and two custom object types based on an AT (Application Table) definition (**BasicPhase** and **ParameterClass**). In most cases we omitted the database information, but in a **whereCondition** it is sometimes easier to use the database column names directly.

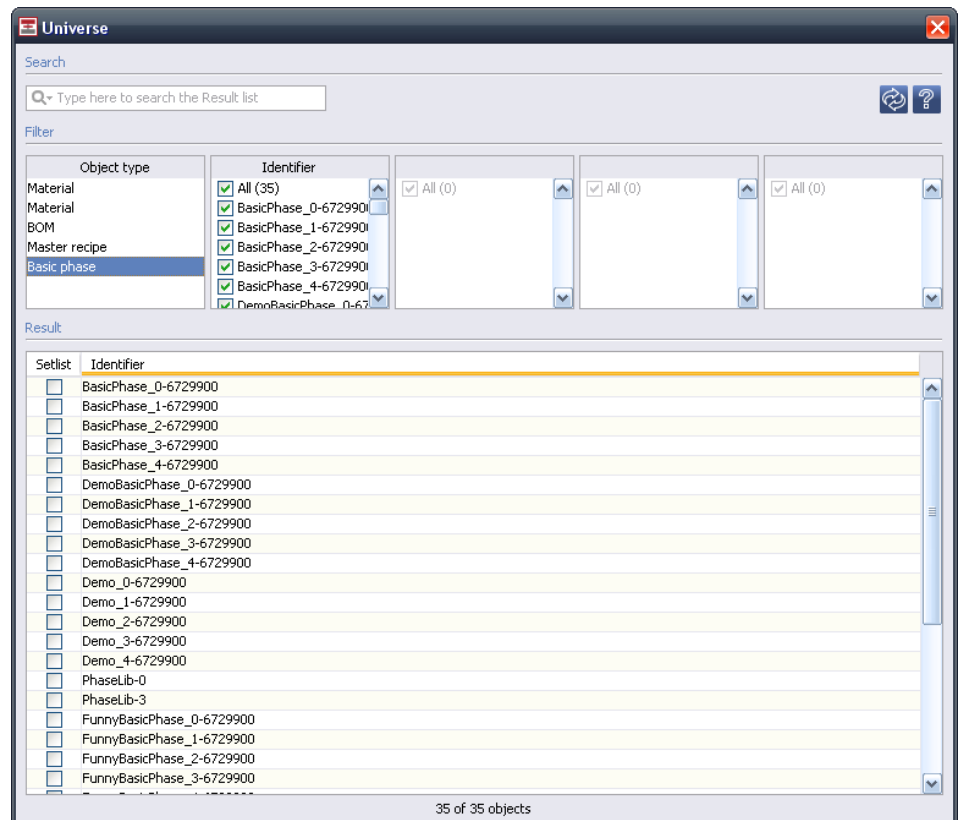


Figure 10: Universe Explorer for a complex configuration containing multiple object types

Elements of the Universe Configuration

The XSD file *universe.xsd*, stored in *PC_MES\apps\recipeeditor\config*, specifies all elements for a Universe Configuration. This is an explanation of the elements and their attributes:

- *Universe*: root element. Consisting of *Universe Objects*.
- *Universe Object*: one for each *Universe Object Type* to be displayed in the *Universe Explorer*. Consisting of
 - *name*: attribute for the unique name
 - *icon*: attribute for the icon and
 - one *DBTable*: mandatory, description of a database table with the data used for searching.
 - one *FilterCriteria*: mandatory, description of filter criteria to be displayed in the filter panel.
 - one *GridColumn*: mandatory, description of grid columns to be displayed in the result grid.

- *DBTable*: configuration for a database table. Consisting of
 - *boundClass*: mandatory, name of the bound Java class with the Data Dictionary.
 - *tableName*: attribute of the database table name.
 - *aliasTableName*: attribute of the database alias table name
 - *udaTableName*: attribute of the database table name for the UDA.
 - *whereCondition*: filter for the data to be read.
 - *orderBy*: order statement for the data.
 - *DatabaseColumns*: mandatory, list of descriptors of database columns to be read.
 - list of *JoinTables*: description of joined table to read from.
- *JoinTable*: similar to *DBTable*, but with an additional attribute:
 - *joinCondition*: mandatory, SQL condition to join the database table to its parent table.
 - *aliasSuffix*: suffix to be used for the alias table name and the UDA table name. This is an alternative to *aliasTableName* for versions of PharmaSuite prior to 8.0.
 - *isInnerJoin*: attribute indicating whether it is an inner or outer join. Default is outer join.
- *DatabaseColumns*: list of descriptors of database columns to be read. Consisting of mandatory *DatabaseColumn*.
- *DatabaseColumn*: descriptors of a database column to be read. Consisting of
 - *propertyName*: mandatory, attribute with the name of the bound property.
 - *columnName*: attribute with the column name of the property.
 - *identifier*: attribute with the key used only in the XML document.
 - *isKey*: attribute indicating that this column is a key column.
 - *isState*: attribute indicating that this column is a status column.
 - *fsmRelationship*: attribute with the FSM relationship name of a status column.
 - *stateJoinColumnName*: attribute with the column name used to build a database join to the status table. It can only be used for application tables (database tables with **AT_** prefix built from an AT definition). The default is **X_stateProxy_231**.
 - *titleMsgID*: message ID to be used for column titles. (Overwrites the internationalization of the Data Dictionary).

- *isCalculation*: attribute indicating that the *columnName* represents a calculation. If true, then no table name is used as prefix in the SQL statement. In this context, calculation means applying SQL functions like ROUND or SIGN instead of mere access to a column's values.
- *forceDataType*: attribute forcing the usage of the configured data type instead of the data type of the property. Allowed values are numeric literals that represent a constant of the *IDataTypes* Java class.
Useful in combination with *isCalculation*, where a data type cannot be determined.
- *isMsgID*: attribute indicating that this value is a message ID. If true and if *isMsgPack* is in an adjacent *DatabaseColumn*, then the value of the message ID will be replaced by the internationalized value.
- *isMsgPack*: attribute indicating that this value is a name of a message pack. If true and if *isMsgID* is in an adjacent *DatabaseColumn*, then the value of the message ID will be replaced by the internationalized value.
- *isHidden*: attribute indicating that the *columnName* will not be read from the database. If true, then the column's values are not included in the result of the SQL statement, but they can be used in join conditions.
- *FilterCriteria*: mandatory, list of descriptors of filter criteria consisting of mandatory *Criterion*.
- *Criterion*: references a *DbColumn* attribute to be used as filter.
- *GridColumns*: list of descriptors of grid columns to be displayed, consisting of mandatory *GridColumn* by the *identifier* attribute.
- *GridColumn*: references *DbColumn*, to be used in grids by the *identifier* attribute.

Parameters in the Universe Configuration

PharmaSuite provides several parameters to simplify the Universe Configuration. Look at the following example:

```
<Universe xsi:noNamespaceSchemaLocation="universe.xsd"
..xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
..<UniverseObject name="material">
....<DBTable boundClass="com.datasweep.compatibility.client.Part"
.....orderBy="{number}"
.....whereCondition="{type} > 0 and {materialUoM} in {discreteUoMs}">
.....<DatabaseColumns>
.....<DBColumn propertyName="key" />
.....<DBColumn propertyName="partNumber" identifier="number" />
.....<DBColumn propertyName="description" />
.....<DBColumn propertyName="UDA_X_shortDescription" />
.....<DBColumn propertyName="UDA_X_materialType" identifier="type" />
.....<DBColumn propertyName="UDA_X_UnitOfMeasure" identifier="materialUoM" />
.....</DatabaseColumns>
....</DBTable>
....<FilterCriteria>
```

```

.....[...]
....</FilterCriteria>
....<GridColumns>
.....[...]
....</GridColumns>
..</UniverseObject>
</Universe>

```

The parameters are enclosed in curly brackets (braces) and colored red. There are two parameter types:

- Local parameters are abbreviations used for local database columns (*DBCColumn*). You can use the identifier of any column of the column list in attributes like **whereCondition**, **joinCondition**, or **orderBy**. In the example above, {type} > 0 is an abbreviation for `UDA_X_materialType > 0`.
- Global parameters are context-specific parameters. The following global parameters are available:
 - *nvlFunction*: The database-dependent function for not-null values.
Oracle: *NVL*, MS SQL Server: *Isnull*
 - *substringFunction*: The database-dependent function to get the portion of a string.
Oracle: *SUBSTR*, MS SQL Server: *Substring*
 - *lengthFunction*: The database-dependent function to get the length of a string.
Oracle: *LENGTH*, MS SQL Server: *LEN*
 - *concatOperator*: The database-dependent operator to concatenate strings.
Oracle: *||*, MS SQL Server: *+*
 - *discreteUoMs*: A comma-separated SQL list of the database keys of all discrete units of measures in the database (each and all convertible units, e.g. Stück in German, pièce in French).
Example: (50, 50074, 49791)
 - *discreteBaseQuantities*: A comma-separated SQL list of the base quantities of all discrete units of measure in the database (each and all convertible units).
Example: ('1 ea', '1 Pc', '1 St')
 - *masterRecipeUsages*: A comma-separated SQL list of choice values of the allowed usage types for phases in master recipes. The values are taken from the **PhaseUsageContext** choice list (see chapter "Adapting and Adding Field Attributes" in Volume 2 of the "Technical Manual Configuration and Extension" [A3] (page 171)).
Example: (1, 3)
 - *workflowUsages*: A comma-separated SQL list of choice values of the allowed usage types for phases in master workflows. The values are taken from the **PhaseUsageContext** choice list (see chapter "Adapting and Adding

Field Attributes" in Volume 2 of the "Technical Manual Configuration and Extension" [A3] (page 171)).

Example: (2, 3)

- *accessPrivilegesForConfidentialObjects*: A comma-separated SQL list of the database keys of all access privileges specific to the confidential objects of the current user.
- *noDBKey*: A numeric constant that represents a non-existing database key. It is useful for *nvlFunction* database function (see above).

Adjusting the Configuration

To adjust the configuration of the Universe Explorer, the **Open** dialog, or the **Select material** dialog, proceed as follows:

1. In Process Designer, expand the **Lists** node in the tree view and open the
 - **red_universeConfiguration** list to adjust the Universe Explorer,
 - **red_openConfiguration** list to adjust the **Open** dialog,
 - **red_materialConfiguration** list to adjust the **Select material** dialog for Recipe Designer - Batch, or
 - **red_materialConfigurationDiscrete** list to adjust the **Select material** dialog for Recipe Designer - Device.

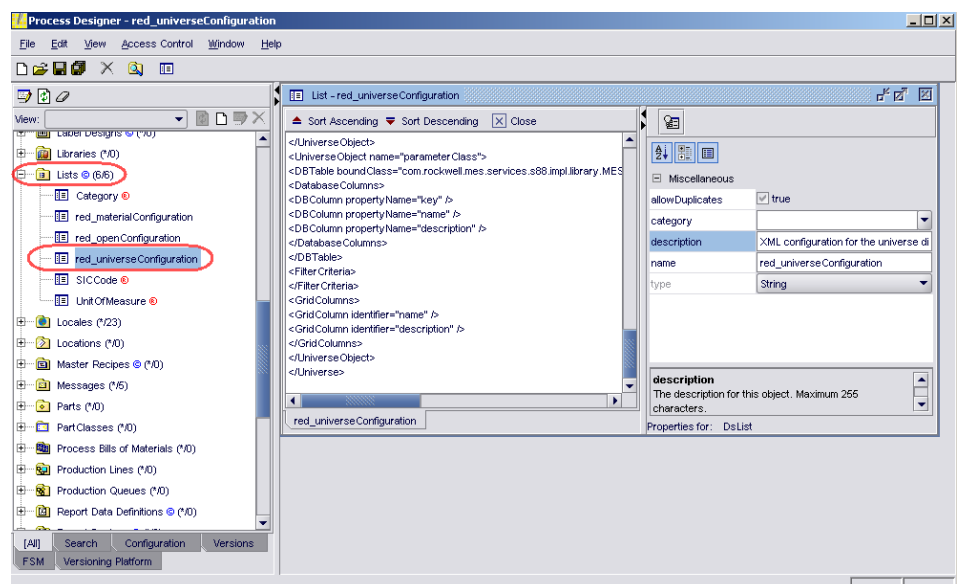


Figure 11: Editing the red_universeConfiguration list

2. Change the XML document as required.
We recommend to use an XML editor and not to edit it in the list editor.
3. Save and close the list.

4. Restart Recipe and Workflow Designer and verify your changes in the
 - Universe Explorer,
 - **Open** dialog, or
 - **Select material** dialog.

Tips and Tricks

There are some configuration keys to ease changing and testing the different Universe Configurations (see chapter "Configuration Keys of PharmaSuite" in Volume 4 of the "Technical Manual Configuration and Extension" [A4] (page 171)). The properties can be set with Process Designer, changing the actual configuration of PharmaSuite (DefaultConfiguration, by default) in the Applications section.

- **Form/Universe/[xmlConfigurationName].readFromFile**
 When doing non-trivial changes, it is easier to use a text or XML editor (using syntax highlighting or code formatting, for example) than working with **List** objects and their default editor in Process Designer. When done, the content can be copied and pasted into the appropriate **List** object. For this purpose, use the **Form/Universe/[xmlConfigurationName].readFromFile** configuration key. If its Boolean value is set to **true**, the XML configuration is not read from the appropriate **List** object, but from a file with the same name and the ".xml" file extension. Like the schema definition, these files are stored in *PC_MES\apps\recipeeditor\config*.
 If **Universe/red_universeConfiguration.readFromFile** is set to **true**, for example, the configuration for the Universe Explorer will be read from *PC_MES\apps\recipeeditor\config\red_universeConfiguration.xml*.

TIP

Please note that the initial content of these files is just an example and does not necessarily match the current Lists' content used for the actual configuration of the dialogs.

- **Form/Universe/[xmlConfigName].refreshAlways**
 The configuration of the Universe dialogs is only read once, when the appropriate dialog is opened for the first time. Although this makes sense for the final application from a performance point of view, it can be quite annoying during development and testing of a configuration because every change applied to the configuration requires a restart of the application to take effect. The configuration key **Form/Universe/[xmlConfigName].refreshAlways** allows you to re-read the configuration even without restarting the application.
 If its Boolean value is set to **true**, the XML configuration is re-read every time the dialog's **Refresh** button is clicked.
 If **Universe/red_openConfiguration.refreshAlways** is set to **true**, for example, the configuration for the **Open master recipe** dialog will be reinitialized with each refresh, avoiding to restart the application to apply configuration changes.

- The initial size and position of the various dialogs is configured in *PC_MES\apps\recipeeditor\config\default.ini*. The prefixes for the configuration of the different dialogs are **universeExplorerWindow**, **openExplorerWindow** and **materialExplorerWindow**.
The dialog's actual dimension and position are then stored separately for each user in *.recipeDesigner.ini* in the user directory.

Configuring the Setlist of Recipe and Workflow Designer

This section contains general information about the Setlist (page 39) and how to adjust its configuration by using standard attributes (page 40) and custom attributes (page 41). You can configure which property shall be visible in the Setlist and in which column of the Setlist's grids a property shall be displayed.

To configure the Setlist, you have to be familiar with

- Process Designer
- Data Dictionary Manager (see chapter "Adapting and Adding Field Attributes" in Volume 2 of the "Technical Manual Configuration and Extension" [A3] (page 171)).
- Recipe and Workflow Designer
- Universe (page 13)

What Is the Setlist?

The Setlist represents a pool of data objects a user can use for building and configuring recipes and workflows in Recipe and Workflow Designer. The Setlist is filled from the Universe (page 13).

The following data objects are available in the Setlist:

- Material
- ERP BOM item (not available in Workflow Designer)
- Equipment class
- Property type
- Work center
- Station
- Signature privilege
- Capability (provided by the system)
- Procedure (current version of PharmaSuite supports custom building-blocks (procedures) and one dummy procedure)
- Unit procedure (current version of PharmaSuite supports custom building blocks (unit procedures) and one dummy unit procedure)

- Operation (current version of PharmaSuite supports custom building blocks (operations) and one dummy operation)
- Phase (current version of PharmaSuite supports standard and custom building blocks (phases) and one dummy phase).

This list is fixed.

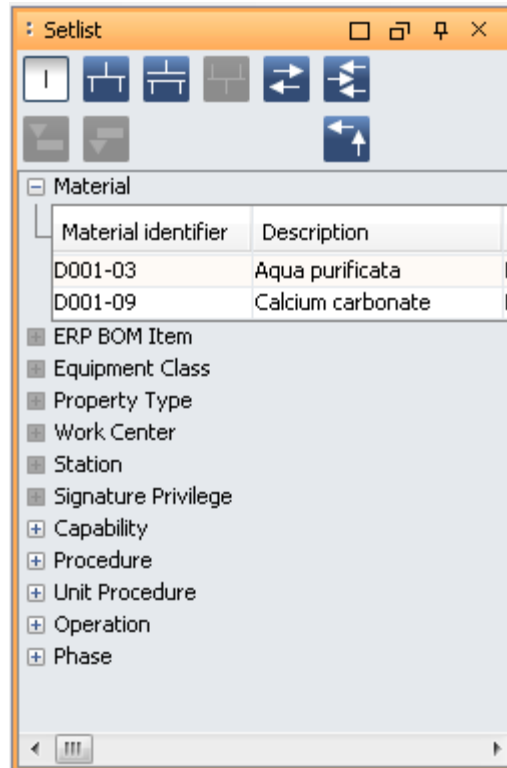


Figure 12: Setlist

Each data object is represented by set of properties such as **Identifier**, **Description**, etc. The set of properties is configurable (page 40).

Adjusting the Configuration

To adjust the configuration of the Setlist, open the Data Dictionary Manager and proceed as follows:

1. Navigate to the **Class management** tab.
 In the **Data Dictionary Classes** panel, in the **Class-related actions** panel, select one of the supported data objects from the drop-down list:
 - Material: *Part* class
 - ERP BOM item: *MESERPomItem* class
 - Equipment class: *MESS88EquipmentClass* class

- Property type: *MESEquipmentPropertyType* class
 - Work center: *WorkCenter* class
 - Station: *Station* class
 - Signature privilege: *AccessPrivilege* class
 - Capability: *SetListCapabilityWrapper* class
 - Phase: *MESPhaseLib* class.
2. In the **Data Dictionary Elements** panel, in the **Element list** panel, select a data dictionary element (e.g. **description**).
 3. Set the following properties according to your needs:
 - **Visible in Setlist**
 - **Grid order**

Adjusting the Configuration of Custom Attributes

As a prerequisite, the application tables of the related recipe entities must be extended by new columns or UDAs, respectively.

- For materials, add as UDA to the **Part** (material) buildtime object
- For ERP BOM items: extend the **X_ERPBomItem** application table.
- For equipment classes: extend the **X_S88EquipmentClass** application table.
- For work centers: add as UDA to the **WorkCenter** buildtime object.
- For stations: add as UDA to the **Station** buildtime object.
- For signature privileges: add as UDA to the **AccessPrivilege** buildtime object.
- For procedures: extend the **X_ProcedureLib** application table.
- For unit procedures: extend the **X_UnitProcedureLib** application table.
- For operations: extend the **X_OperationLib** application table.
- For phases: extend the **X_PhaseLib** application table.

TIP

For capabilities, custom attributes are not supported.

To adjust the configuration of custom attributes (UDAs for runtime/buildtime objects, AT columns for AT tables) of the Setlist, open the Data Dictionary Manager and proceed as follows:

1. Navigate to the **Class management** tab.
In the **Data Dictionary Classes** panel, in the **Class-related actions** panel, select one of the following classes from the drop-down list:
 - For the buildtime objects (with UDAs), select the buildtime object itself:
 - Material: *Part* class
 - Work center: *WorkCenter* class
 - Station: *Station* class
 - Signature privilege: *AccessPrivilege* class.
 - For application tables select one of the custom configuration classes indicated by the **CustomerConfig** appendix:
 - Equipment class: *MESS88EquipmentClassCustomerConfig* class
 - Procedure: *MESProcedureLibCustomerConfig* class
 - Unit procedure: *MESUnitProcedureLibCustomerConfig* class
 - Operation: *MESOperationLibCustomerConfig* class
 - Phase: *MESPhaseLibCustomerConfig* class.
2. In the **Data Dictionary Elements** panel, in the **Element list** panel, select a data dictionary element.
If the custom attribute is not available as data dictionary element, in the **Element-related actions** panel, type the name of the attribute (here: **Y_customerInfo**) in the text box and then click the **Add element** button.

TIP

The attribute name of an UDA starts with **UDA_**, the remainder is identical to the UDA name.

The attribute name of an AT column is identical to its column name.
3. Set the properties according to your needs. The following properties are relevant to the Setlist:
 - **Visible in Setlist**
 - **Grid order**
 - **Category**

Configuring the Property Windows of Recipe and Workflow Designer

This section contains general information about the property windows (page 43) and how to adjust the configuration of standard attributes (page 44) and custom attributes (Header and Element property window (page 45), Source property window (page 49)) as well as packaging level-related attributes (page 47).

You can configure which property shall be visible in the property windows, if a structure type-specific property shall be visible in the property windows, and in which column of the property windows' grids a property shall be displayed.

To configure a property window, you have to be familiar with

- Process Designer
- Data Dictionary Manager (see chapter "Adapting and Adding Field Attributes" in Volume 2 of the "Technical Manual Configuration and Extension" [A3] (page 171)).
- Recipe and Workflow Designer

TIP

This section does not apply to the Header property window of change requests.

What Is a Property Window?

A property window provides access to the properties of the recipe, workflow, or building block headers and elements as well as the source building blocks on which they are based.

The **Header property window** lists the properties of the root components of a recipe, workflow, or custom building block. Root components are master recipes, master workflow, and building blocks (procedure, unit procedure, operation, and phase). A building block is a root component as long as it represents the top level of a structure of procedural elements.

The properties are only editable when the root component has the focus and has not been opened in read-only mode.

The **Element property window** lists the properties of the selected recipe element, workflow element, or custom building block element. Elements are building blocks (procedure, unit procedure, operation, and phase) that belong to a procedural structure of

a root component of a higher level. Element properties are not available for root components and for the "Dummy phase" elements.

The properties are editable by default when the element does not belong to an approved building block structure (i.e. the element of the higher level has not been derived from an approved custom building block).

The **Source property window** lists the properties of the original building block that was used as a template to create the selected recipe, workflow, or custom building block element. The original building block can be a system building block or a custom building block. In case no template was used to create a building block, the **Source property window** is empty.

The properties can only be viewed.

The screenshot shows a window titled "Header" with a standard Windows-style title bar (minimize, maximize, close buttons). The window contains two expandable sections: "Basic Data" and "Status Data".

Basic Data	
Object type	Procedure
Identifier	EW-02
Revision	02
Short description	Demo for Technical Man
Category 1	Fluids
Category 2	Emulsion
Category 3	-
Description	

Status Data	
Status	Draft
Approved on	
Approved by	
Archived on	
Archived by	

At the bottom of the window, there is a tabbed interface with three tabs: "Header" (selected), "Element", and "Source".

Figure 13: Header property window

Adjusting the Configuration

To adjust the configuration of a property window, open the Data Dictionary Manager and proceed as follows:

1. Navigate to the **Class management** tab.
In the **Data Dictionary Classes** panel, in the **Class-related actions** panel, select one of the supported data objects from the drop-down list:
 - Header property window:
 - Master recipe - batch, Master recipe - device, Master workflow:
MESMasterRecipe class

- Procedure: *MESProcedureLib* class
 - Unit procedure: *MESUnitProcedureLib* class
 - Operation: *MESOperationLib* class
 - Phase: *MESPhaseLib* class, see figure below.
 - Element property window:
 - Procedure: *MESProcedure* class
 - Unit procedure: *MESUnitProcedure* class
 - Operation: *MESOperation* class
 - Phase: *MESPhase* class.
 - Source property window:
 - *BuildingBlocksOfRecipeElement* class.
2. In the **Data Dictionary Elements** panel, in the **Element list** panel, select a data dictionary element (e.g. **description**).
 3. Set the properties according to your needs. The following properties are relevant to property windows:
 - **Visible in Setlist**
 - **Visible in structure type**
 - **Grid order**
 - **Category**
 - **Cell-editor class** (optional)

Adjusting the Configuration of Custom Attributes (Header and Element Property Window)

As a prerequisite, the application tables of the related recipe/workflow entities must be extended by new columns or UDAs, respectively.

- For master recipes/workflows: add as UDA.
- For procedures: extend the **X_ProcedureLib** application table.
Additionally, extend the **X_Procedure** application table to make the attribute available in procedures of master recipes/workflows.
- For unit procedures: extend the **X_UnitProcedureLib** application table.
Additionally, extend the **X_UnitProcedure** application table to make the attribute available in unit procedures of master recipes/workflows.

- For operations: extend the **X_OperationLib** application table.
Additionally, extend the **X_Operation** application table to make the attribute available in operations of master recipes/workflows.
- For phases: extend the **X_PhaseLib** application table.
Additionally, extend the **X_Phase** application table to make the attribute available in phases of master recipes/workflows.

To adjust the configuration of custom attributes (UDAs for runtime/buildtime objects, AT columns for AT tables) of a property window, open the Data Dictionary Manager and proceed as follows:

1. Navigate to the **Class management** tab.
In the **Data Dictionary Classes** panel, in the **Class-related actions** panel, from the drop-down list, select one of the custom configuration classes indicated by the **CustomerConfig** appendix:
 - Header property window:
 - Master recipe - batch, Master recipe - device, Master workflow: *MESMasterRecipeCustomerConfig* class
 - Procedure: *MESProcedureLibCustomerConfig* class
 - Unit procedure: *MESUnitProcedureLibCustomerConfig* class
 - Operation: *MESOperationLibCustomerConfig* class
 - Phase: *MESPhaseLibCustomerConfig* class, see figure below.
 - Element property window:
 - Procedure: *MESProcedureCustomerConfig* class
 - Unit procedure: *MESUnitProcedureCustomerConfig* class
 - Operation: *MESOperationCustomerConfig* class
 - Phase: *MESPhaseCustomerConfig* class.
2. In the **Data Dictionary Elements** panel, in the **Element list** panel, select a data dictionary element.
If the custom attribute is not available as data dictionary element, in the **Element-related actions** panel, type the name of the attribute (here: **Y_customerInfo**) in the text box and then click the **Add element** button.

TIP

The attribute name of an UDA starts with **UDA_**, the remainder is identical to the UDA name.
The attribute name of an AT column is identical to its column name.

3. Set the properties according to your needs. The following properties are relevant to property windows:
 - **Visible in Setlist**
 - **Visible in structure type**
 - **Grid order**
 - **Category**
 - **Cell-editor class** (optional)

TIP - CONFIGURING THE SAME UDA

If you configure the same UDA for the *MasterRecipe* and the *MESMasterRecipeCustomerConfig* classes in the Data Dictionary Management tool, then the UDA in the *MESMasterRecipeCustomerConfig* class is ignored. We recommend to configure your system in this way, only if the UDA is to be displayed by different applications (e.g. Production Management Client and Recipe and Workflow Designer) and if you wish to maintain only one Data Dictionary entry. If the UDA is only used in Recipe and Workflow Designer, then configure the UDA for the customer class (*MESMasterRecipeCustomerConfig*).

TIP - CREATING OR USING A CUSTOM BUILDING BLOCK

When you create a custom building block from a recipe/workflow element or insert a custom building block into a recipe/workflow, all customer-specific attributes are copied. The attributes are only copied if the column name of the custom building block attribute matches the column name of the recipe/workflow element attribute. Example: If a custom column with the name **ct_displayOnSummary** exists in the **X_OperationLib** and **X_Operation** application tables, the value is copied.

Adjusting the Packaging Level-related Attributes (Header Property Window)

As a prerequisite for adding packaging level-related attributes, the UDAs of the master recipe type must have been extended.

The next step is to adapt the packaging level bean factory. Since the master recipe holds the higher set of packaging levels (in contrast to the material parameters that carry the lower tier, see "Configuring the Parameter Panel of Recipe and Workflow Designer" (page 53)) the following methods in *PackagingLevelFactory* must be adapted:

1. `List<IPackagingLevelBean>`
`retrievePackagingLevelsOfTierTwoHolder(IPackagingLevelsTierTwoHolder
packagingLevelsTierTwoHolder);`
2. `void updatePackagingLevelOfTierTwoHolder(IPackagingLevelsTierTwoHolder
packagingLevelsTierTwoHolder, IPackagingLevelBean packagingLevel);`
3. `int getNumberOfTierTwoPackagingLevels();`

To adjust the configuration of the packaging levels-related attributes of the **Header property window**, open the Data Dictionary Manager and proceed as follows:

1. Navigate to the **Class management** tab.
In the **Data Dictionary Classes** panel, in the **Class-related actions** panel, from the drop-down list, select one of the custom configuration classes indicated by the **IndexedPropertiesConfig** appendix:
 - Master recipe: *MESMasterRecipeIndexedPropertiesConfig* class
2. In the **Data Dictionary Elements** panel, in the **Element** list panel, the system displays the entries of the standard packaging levels: **packagingLevel_6**, ..., **packagingLevel_9**.
 - To remove an existing level, select the corresponding entry and click the **Remove element** button.
 - To add a new level, e.g. level 10, type **packagingLevel_10** as the name of the new level in the text box and click the **Add element** button.

In any case, in the end the levels must be consecutive and the lowest level must be the highest level of the material parameter plus 1. So, if your lowest level is 6 and your highest level is 10, you must also have all levels in between.
3. When you are adding levels, you must configure the entries in the same way as they are configured for the standard levels. The following properties are relevant to property windows:
 - **Category**
Set the category property to **catPackagingLevels**.
 - **Cell-editor class**
Use the same as the standard levels.
 - **Visible in SetList**
Set to **Yes**.
 - **Visible in structure type**
Select the master recipe types.
 - **Grid order**
Make sure that the new levels have a higher value than the standard ones.

To localize a new level of the master recipe object, open the **DataDictionary_Default_MESMasterRecipe** message pack. In our example, the new level is level 10. Add the **packagingLevel10_Label** message for the label and the **packagingLevel10_Hint** message for additional information.
Now, the **Header property window** displays the messages defined in the message pack.

Adjusting the Configuration of Custom Attributes (Source Property Window)

To adjust the configuration of custom attributes (AT columns for AT tables) of a property window, proceed as follows:

1. In Java, create a subclass of the *AbstractSourceBuildingBlockInformationOfRecipeElement* class and add all custom properties (stored in new AT columns) you want to see. Here, the class is named *MYCBuildingBlocksOfRecipeElement* with MYC for the My Company vendor code.

```
/**
 * Class which represents the building blocks of a recipe element, i.e. its underlying custom building block (if
 * present) and its underlying system building block (if present).
 * <p>
 *
 * @author System integrator
 */
public class MYCBuildingBlocksOfRecipeElement extends AbstractSourceBuildingBlockInformationOfRecipeElement {

    /**
     * C'tor
     *
     * @param recipeElement Underlying recipe element
     */
    public MYCBuildingBlocksOfRecipeElement(IRecipeElement recipeElement) {
        super(recipeElement);
    }

    /**
     * @return Additional attribute of customized custom building block.
     */
    public String getCustomBBAAdditionalAttribute() {
        if (getCustomBB() == null) {
            return null;
        }
        return (String) getCustomBB().getATRow().getValue("X_additionalAttribute");
    }

    /**
     * @return Additional attribute of customized system building block.
     */
    public String getSystemBBAAdditionalAttribute() {
        return (String) getSystemBB().getATRow().getValue("X_additionalAttribute");
    }
}
```

Figure 14: Example: Customized bean class for the Source Property Window

2. In Java, create a factory class implementing the *ICreatorForSourceBuildingBlockInformationOfRecipeElement* interface, which creates an instance of the newly created *MYCBuildingBlocksOfRecipeElement* class. Here, the factory class is named *MYCBeanCreatorForSourcePropertyPane*.

```
package com.pharmaGiant.mes.apps.recipeeditor.impl.frame.factory;

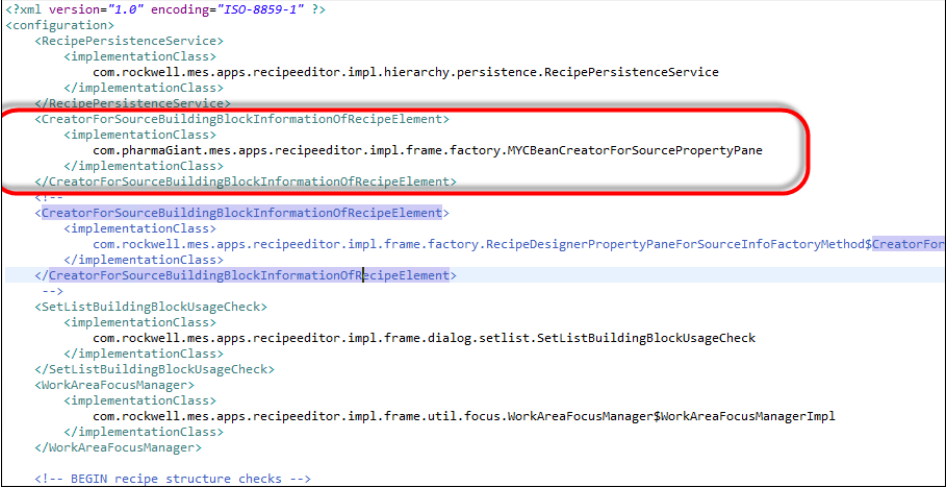
import com.rockwell.mes.apps.recipeeditor.ifc.frame.factory.ICreatorForSourceBuildingBlockInformationOfRecipeElement;

/**
 * Creates the bean that is
 * <ul>
 * <li>displayed in the source property pane and
 * <li>used for comparison.
 * </li>
 * </ul>
 */
public class MYCBeanCreatorForSourcePropertyPane implements ICreatorForSourceBuildingBlockInformationOfRecipeElement {

    @Override
    public ISourceBuildingBlockInformationOfRecipeElement createBeanForSourcePropertyPane(IRecipeElement recipeElement) {
        if (recipeElement == null) {
            return null;
        }
        return new MYCBuildingBlocksOfRecipeElement(recipeElement);
    }
}
```

Figure 15: Example: Customized factory class for the Source Property Window

3. Register the newly created *MYCBeanCreatorForSourcePropertyPane* factory class in the *apps.recipeeditor.config.apps-recipeeditor.xml* configuration with the complete path in the *CreatorForSourceBuildingBlockInformationOfRecipeElement* XML item (see section "Configuring Service Implementations" in chapter "Managing Services" in Volume 2 of the "Technical Manual Configuration and Extension" [A3] (page 171)).



```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<configuration>
  <RecipePersistenceService>
    <implementationClass>
      com.rockwell.mes.apps.recipeeditor.impl.hierarchy.persistence.RecipePersistenceService
    </implementationClass>
  </RecipePersistenceService>
  <CreatorForSourceBuildingBlockInformationOfRecipeElement>
    <implementationClass>
      com.pharmaGiant.mes.apps.recipeeditor.impl.frame.factory.MYCBeanCreatorForSourcePropertyPane
    </implementationClass>
  </CreatorForSourceBuildingBlockInformationOfRecipeElement>
  <!--
  <CreatorForSourceBuildingBlockInformationOfRecipeElement>
    <implementationClass>
      com.rockwell.mes.apps.recipeeditor.impl.frame.factory.RecipeDesignerPropertyPaneForSourceInfoFactoryMethod$CreatorFor
    </implementationClass>
  </CreatorForSourceBuildingBlockInformationOfRecipeElement>
  -->
  <SetListBuildingBlockUsageCheck>
    <implementationClass>
      com.rockwell.mes.apps.recipeeditor.impl.frame.dialog.setlist.SetListBuildingBlockUsageCheck
    </implementationClass>
  </SetListBuildingBlockUsageCheck>
  <WorkAreaFocusManager>
    <implementationClass>
      com.rockwell.mes.apps.recipeeditor.impl.frame.util.focus.WorkAreaFocusManager$WorkAreaFocusManagerImpl
    </implementationClass>
  </WorkAreaFocusManager>
  <!-- BEGIN recipe structure checks -->
```

Figure 16: Example: Registration of a customer factory class for the Source Property Window

4. In Process Designer, run the **mes_DataDictManagerForm** form to start the **Data Dictionary Management** tool.

5. Navigate to the **Class management** tab.

In the **Data Dictionary Classes** panel, in the **Class-related actions** panel, select the newly created class (here: *MYCBuildingBlocksOfRecipeElement*) from the drop-down list. If the class does not exist yet, type the complete name in the class name box and click the **Create/Reload** button to create the required Data Dictionary Class.

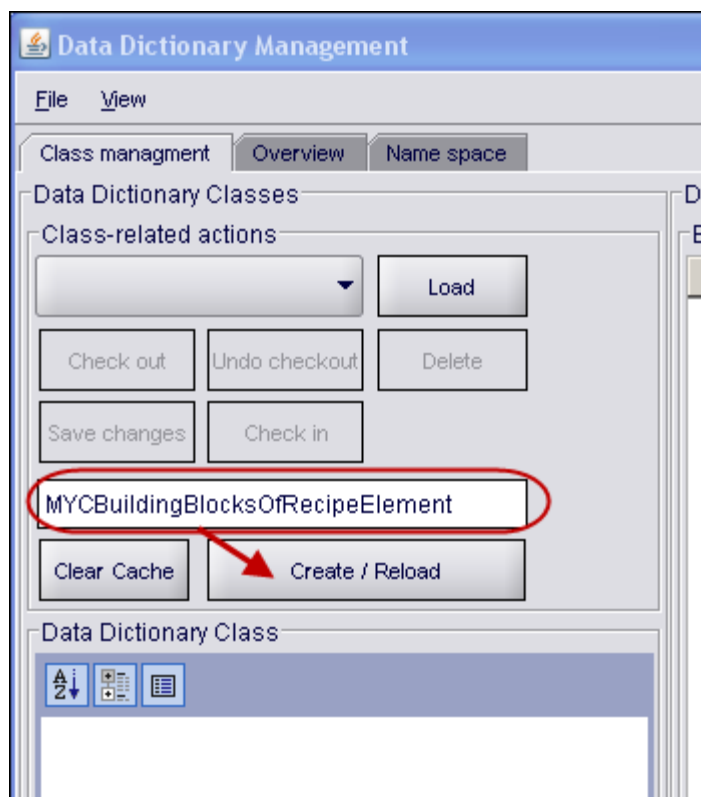


Figure 17: Creating a new Data Dictionary Class for customer bean classes

6. Optional:

To use the data dictionary elements of the standard, just copy them.

Navigate to the **Class management** tab.

In the **Data Dictionary Classes** panel, in the **Class-related actions** panel, click the **Copy elements** button. In the **Copy Data Dictionary Elements** window, select **BuildingBlocksOfRecipeElement** as source data dictionary and click the **Copy** button.

7. In the **Data Dictionary Elements** panel, in the **Element list** panel, select a data dictionary element.

If the custom attribute is not available as data dictionary element, in the **Element-related actions** panel, type the name of the attribute (here: **Y_customerInfo**) in the text box and then click the **Add element** button.

TIP

The attribute name of an AT column is identical to its column name.

8. Set the properties according to your needs. The following properties are relevant to property windows:
 - **Visible in Setlist**
 - **Visible in structure type**
 - **Grid order**
 - **Category**
 - **Cell-editor class** (optional)

Configuring the Parameter Panel of Recipe and Workflow Designer

This section contains general information about the Parameter Panel (page 53) and how to adjust the configuration of standard attributes (page 54) and custom attributes (page 55) as well as packaging level-related attributes (page 56).

You can configure if a structure type-specific property shall be visible in the Parameter Panel and in which column of the Parameter Panel's grid a property shall be displayed.

To configure the Parameter Panel, you have to be familiar with

- Process Designer
- Data Dictionary Manager (see chapter "Adapting and Adding Field Attributes" in Volume 2 of the "Technical Manual Configuration and Extension" [A3] (page 171)).
- Recipe and Workflow Designer

What Is the Parameter Panel?

The Parameter Panel provides access to all parameters and transitions (with an identifier) of a building block.

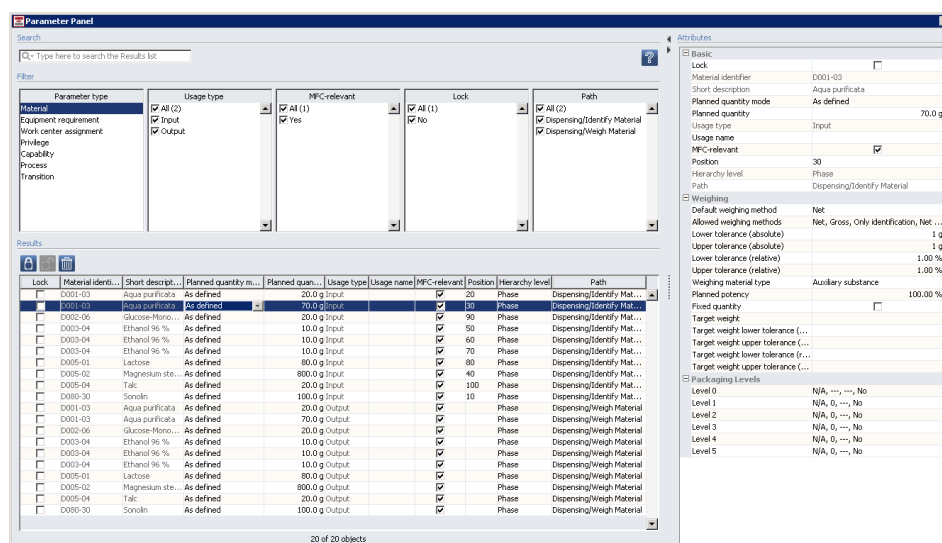


Figure 18: Parameter Panel (material parameters)

The following types of parameters are available:

- Material
- Equipment requirement
- Work center assignment
(applies to work centers on unit procedure level and to stations on operation level)
- Privilege
- Capability
- Process
- Transition

Adjusting the Configuration

To adjust the configuration of the Parameter Panel, open the Data Dictionary Manager and proceed as follows:

1. Navigate to the **Class management** tab.
In the **Data Dictionary Classes** panel, in the **Class-related actions** panel, select one of the supported data objects from the drop-down list:
 - Material parameter: *MESMaterialParameter* class
 - Equipment requirement parameter: *MESEquipmentReqParameter* class
 - Work center assignment parameter: *MESEquipmentParameter* class
 - Privilege parameter: *MESPrivilegeParameter* class
 - Capability: *SetListCapabilityWrapper* class
 - Process parameter: *MESProcessParameterInstance* class
 - Transition: *SFCTransitionTableModelEntry* class.
2. In the **Data Dictionary Elements** panel, in the **Element list** panel, select a data dictionary element (e.g. **description**).
3. Set the properties according to your needs. The following properties are relevant to the Parameter Panel:
 - **Visible in structure type**
 - **Grid order**
 - **Cell-editor class** (optional)

Adjusting the Configuration of Custom Attributes

As a prerequisite, the application tables of the related parameter types must be extended by new columns. For details, see section "Adding Field Attributes to Application Table Objects" in chapter "Adapting and Adding Field Attributes" in Volume 2 of the "Technical Manual Configuration and Extension" [A3] (page 171).

- For material parameters: extend the **X_MaterialParameter** application table.
- For equipment requirement parameters: extend the **X_EquipmentReqParameter** application table.
- For work center assignment parameters: extend the **X_EquipmentParameter** application table.
- For privilege parameters: extend the **X_PrivilegeParameter** application table.
- For process parameters: extend the **X_ProcessParameterInstance** application table.
- For capability parameters and transitions: custom attributes are not possible.

To adjust the configuration of custom attributes (AT columns for AT tables) of the Parameter Panel, open the Data Dictionary Manager and proceed as follows:

1. Navigate to the **Class management** tab.
In the **Data Dictionary Classes** panel, in the **Class-related actions** panel, from the drop-down list, select one of the custom configuration classes indicated by the **CustomerConfig** appendix:
 - Material parameter: *MESMaterialParameterCustomerConfig* class
 - Equipment requirement parameter:
MESEquipmentReqParameterCustomerConfig class
 - Work center assignment parameter:
MESEquipmentParameterCustomerConfig class
 - Privilege parameter: *MESPrivilegeParameterCustomerConfig* class
 - Process parameter: *MESProcessParameterInstanceCustomerConfig* class.
2. In the **Data Dictionary Elements** panel, in the **Element list** panel, select a data dictionary element.
If the custom attribute is not available as data dictionary element, in the **Element-related actions** panel, type the name of the attribute (e.g. **Y_newProperty**) in the text box and then click the **Add element** button.

TIP

The attribute name of an AT column is identical to its column name.

3. Set the properties according to your needs. The following properties are relevant to the Parameter Panel:

- **Visible in structure type**
- **Grid order**
- **Cell-editor class** (optional)
- **Category** (only for material parameters)

To display material parameter properties in the grid (horizontal pane), set the **Category** property to **catParameterBasic**. All other properties are displayed in the configured section of the (vertical) property pane.

TIP

Unless you localize the new column in the corresponding message pack, the Parameter Panel uses the name of the column as label on the user interface in Recipe and Workflow Designer.

To localize the new column, for example, of the material parameter object, open the **DataDictionary_Default_MESMaterialParameterCustomerConfig** message pack (= "**DataDictionary_Default_**" + <class name>). In our example, the column is named **MYC_testString**. Add the **MYC_testString_Label** message for the column's label and the **MYC_testString_Hint** message for additional information. Now, the Parameter Panel displays the messages defined in the message pack.

Adjusting the Packaging Level-related Attributes

As a prerequisite, the application table of the material parameters (**X_MaterialParameter**) must have been extended by new columns. For details, see section "Adding Field Attributes to Application Table Objects" in chapter "Adapting and Adding Field Attributes" in Volume 2 of the "Technical Manual Configuration and Extension" [A3] (page 171).

The next step is to adapt the packaging level bean factory. Since the material parameters hold the lower set of packaging levels (in contrast to the master recipe that carries the higher tier, see "Configuring the Property Windows of Recipe and Workflow Designer" (page 43)), the following methods in *PackagingLevelFactory* must be adapted:

1. *List<IPackagingLevelBean>*
retrievePackagingLevelsOfTierOneHolder(IPackagingLevelsTierOneHolder packagingLevelsTierOneHolder);
2. *void updatePackagingLevelOfTierOneHolder(IPackagingLevelsTierOneHolder packagingLevelsTierOneHolder, IPackagingLevelBean packagingLevel);*
3. *int getNumberOfTierOnePackagingLevels();*

To adjust the configuration of the packaging levels-related attributes of the Parameter Panel, open the Data Dictionary Manager and proceed as follows:

1. Navigate to the **Class management** tab.
In the **Data Dictionary Classes** panel, in the **Class-related actions** panel, from the drop-down list, select one of the custom configuration classes indicated by the **IndexedPropertiesConfig** appendix:
 - Material parameter: *MESMaterialParameterIndexPropertiesConfig* class
2. In the **Data Dictionary Elements** panel, in the **Element** list panel, the system displays the entries of the standard packaging levels: **packagingLevel_0**, ..., **packagingLevel_5**.
 - To remove an existing level, select the corresponding entry and click the **Remove element** button.
 - To add a new level, e.g. level 6, type **packagingLevel_6** as the name of the new level in the text box and click the **Add element** button.

In any case, in the end the levels must be consecutive and the lowest level must be level 0. So, if your lowest level is 0 and your highest level is 6, you must also have all levels in between.
3. When you are adding levels, you must configure the entries in the same way as they are configured for the standard levels. The following properties are relevant to the Parameter Panel:
 - **Category**
Set the category property to **catPackagingLevels**.
 - **Cell-editor class**
Use the same as the standard levels.
 - **Grid order**
Make sure that the new levels have a higher value than the standard ones.

To localize a new level of the material parameter object, open the **DataDictionary_Default_MESMaterialParameter** message pack. In our example, the new level is level 6. Add the **packagingLevel6_Label** message for the label and the **packagingLevel6_Hint** message for additional information. Now, the Parameter Panel displays the messages defined in the message pack.

Managing the Layout of Recipe and Workflow Designer and Data Manager

This section contains general information about layout management (page 59) and how to

- remove/add items (page 60) from/to the menu bar, the toolbar, or the context menus and
- change the layout of floating windows (page 60).

What Is Layout Management?

Layout management is the process of determining the size and position of windows. In Recipe and Workflow Designer and Data Manager the layout of the windows, bars, and the menu items is pre-defined.

The configurations listed below define the initial layout of Recipe and Workflow Designer and Data Manager (see also chapter "Configuration Keys of PharmaSuite" in Volume 4 of the "Technical Manual Configuration and Extension" [A4] (page 171)):

- *red_buttonConfiguration* and *mdt_buttonConfiguration* define in XML which items are visible in the menu bar, the toolbar, and the context menus. By default, Recipe and Workflow Designer and Data Manager read the configuration from the database (**red_buttonConfiguration** list and **mdt_buttonConfiguration** list).
- *red_defaultLayoutSettings* and *mdt_defaultLayoutSettings* define the size and position of floating windows in the format of INI files. By default, Recipe and Workflow Designer and Data Manager read the configuration from the database (**red_defaultLayoutSettings** list and **mdt_defaultLayoutSettings** list).

There is no configuration of the initial layout of the docking frames.

Recipe and Workflow Designer and Data Manager store the user settings in the table of the **X_AppLayoutSettings** AT definition.

Removing/Adding Items from/to the Menu Bar, Toolbar, Context Menu

To remove or add items from/to the menu bar, the toolbar, or the context menus, proceed as follows:

1. In Process Designer, expand the **Lists** node and open the **red_buttonConfiguration** list for Recipe and Workflow Designer or the **mdt_buttonConfiguration** list for Data Manager, respectively.
2. Remove items as required (here: **CloseAll**).
OR
Add new items.

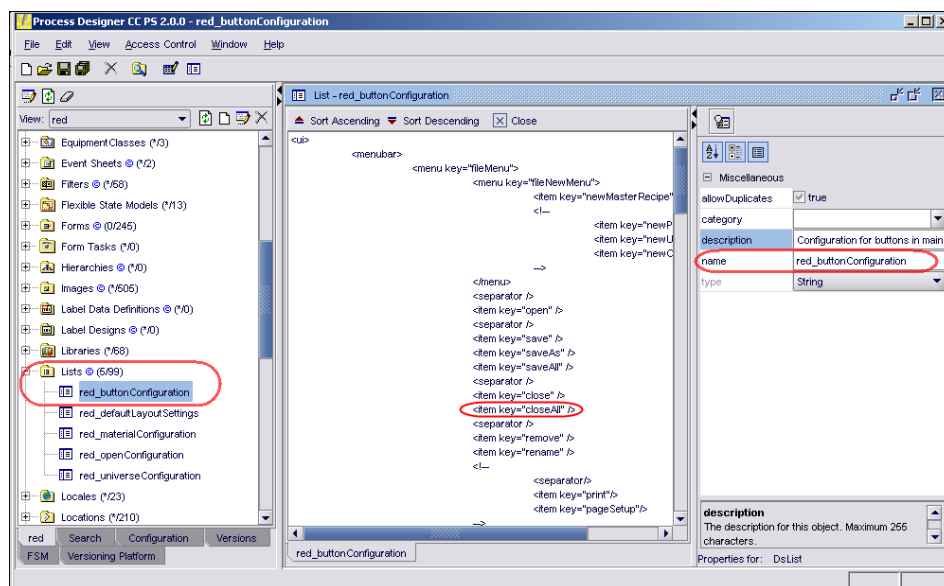


Figure 19: Modifying the red_buttonConfiguration list

3. Save your changes and close the list.
4. Verify your changes in the application.

Changing the Layout of Floating Windows

To change the layout of floating windows, proceed as follows:

1. In Process Designer, expand the **Lists** node and open the **red_defaultLayoutSettings** list for Recipe and Workflow Designer or the **mdt_defaultLayoutSettings** list for Data Manager, respectively.

2. Change the position or the size of the windows as required.
Make sure not to overlap windows that should be completely visible.

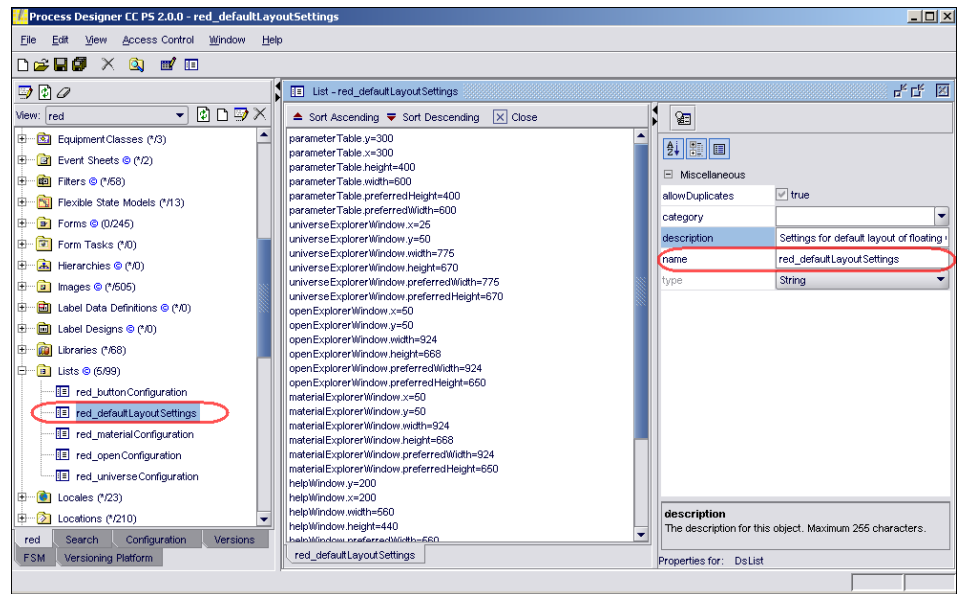


Figure 20: Modifying the red_defaultLayoutSettings list

3. Save your changes and close the list.
4. Verify your changes in the application.

Configuring the Expression Editor of Recipe and Workflow Designer and Data Manager

This section contains general information about the Expression editor (page 63) and how to add user-defined functions to the Expression editor, either a context-related function by configuring the Expression editor (only available for Recipe and Workflow Designer) (page 64) or an arbitrary function by providing Java code (page 68). A new function is listed in the Functions tree panel of the Expression editor.

In Recipe and Workflow Designer, a new function can be used in transition conditions, input expressions, and flexible equipment rules. It is evaluated during EBR execution. In Data Manager, a new function can be used in transition conditions and transition actions of status graphs. It is also evaluated during EBR execution when status graph triggers are fired.

TIP

It is possible to provide the same function for both Recipe and Workflow Designer and Data Manager, but that requires two extensions. The localization of the functions, however, is shared.

To add a user-defined function to the Expression editor, you have to be familiar with

- XML (only required for Recipe and Workflow Designer-related extensions)
- Java (only required for adding an arbitrary function (page 68))
- Process Designer
- Recipe and Workflow Designer
- Data Manager

What Is the Expression Editor?

The Expression editor in Recipe and Workflow Designer supports the operator with defining expressions that provide inputs to process parameter attributes and transition conditions, which are required for graphs containing selection branches or loops.

The Expression editor in Data Manager supports the operator with defining expressions that provide transition conditions and transition actions for status graphs.

Providing a Context-related Function

➤ Applies to Recipe and Workflow Designer only.

To provide a context-related function for the Expression editor, perform the following steps:

1. Configure a new function to be added to the application (page 64).
2. Enable localization and online help (page 67).

Adding a New Function

The **red_expressionContextFunctionConfiguration** list consists of XML code that represents all context-related functions according to the *expressionContextFunctions.xsd* schema:

XSD schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
            attributeFormDefault="unqualified">
  <xsd:element name="ExpressionContextFunctions">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="FunctionDescriptor" minOccurs="0" maxOccurs="unbounded" />
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="FunctionDescriptor">
    <xsd:complexType>
      <xsd:sequence>
        <!-- The name of the virtual function -->
        <xsd:element name="FunctionName" minOccurs="1" maxOccurs="1">
          <xsd:simpleType>
            <xsd:restriction base="xsd:string">
              <xsd:minLength value="1" />
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:element>
        <!-- The mere name of the main class, representing the base object to
              retrieve the property from -->
        <xsd:element name="ClassName" minOccurs="1" maxOccurs="1">
          <xsd:simpleType>
            <xsd:restriction base="xsd:string">
              <xsd:minLength value="1" />
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:element>
        <!-- The path to the property -->
        <xsd:element name="PropertyPath" minOccurs="1" maxOccurs="1">
          <xsd:simpleType>
            <xsd:restriction base="xsd:string">
              <xsd:minLength value="1" />
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:element>
      </xsd:sequence>
      <!-- The identifier of the group the function belongs to -->
      <xsd:attribute name="groupId" type="xsd:string" use="optional" />
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

```

    <!-- The sort index -->
    <xsd:attribute name="sortIndex" type="xsd:integer" use="optional" default="1" />
  </xsd:complexType>
</xsd:element>
</xsd:schema>

```

In order to add a function, add a new **FunctionDescriptor** section to the **red_expressionContextFunctionConfiguration** list according to the schema.

New FunctionDescriptor in XSD schema

```

<?xml version="1.0" encoding="UTF-8"?>
<ExpressionContextFunctions xsi:noNamespaceSchemaLocation=
    "expressionContextFunctions.xsd"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  [...]
  <FunctionDescriptor groupId="100_CustomFunctionsGroup" sortIndex="10">
    <FunctionName>customContextFunction</FunctionName>
    <ClassName>Station</ClassName>
    <PropertyPath>WorkCenter.Location.Description</PropertyPath>
  </FunctionDescriptor>
  [...]
</ExpressionContextFunctions>

```

FunctionName (here: **customContextFunction**) is the name by which the function will be accessible in the expression in the Expression editor.

ClassName and **PropertyPath** are used to access a certain context-related property.

The class name property defines an alias for the root object that serves as the entry point to retrieve a property from the current execution context. Valid values are:

- **ControlRecipeCommon**: for the **ControlRecipe** object whose properties are available for use in master recipes, master workflows, and building blocks;
- **ControlRecipe**: for the **ControlRecipe** object whose properties are available for use in master recipes and building blocks;
- **ControlRecipeForWorkflow**: for the **ControlRecipe** object whose properties are available for use in master workflows and building blocks;
- **OrderStepMfcItem**: for the common properties of the **OrderStepInput** and **OrderStepOutput** objects;
- **OrderStepOutputItem**: for the specific properties of the **OrderStepOutput** object (that do not exist for the **OrderStepInput** object);
- **BomItem**: for the **ProcessBomItem** object (not used in the standard system, but available for extensions);
- **Station**: for the **Station** object;
- **Room**: for the **IMESRoomEquipment** object.

The set of root objects is fixed and cannot be extended by means of configuration.

TIP

Most context-related properties will be available using the given root objects. If a property of interest cannot be retrieved this way, you can still implement an arbitrary user-defined function to retrieve that property by using custom code instead of adapting the configuration. For details, see "Providing an Arbitrary Function" (page 68).

The property path property defines the object property to be accessed according to the PharmaSuite object model. The property allows to access properties across multiple object hierarchy levels. For example, the **WorkCenter.Location.Description** property path applied to the **Station** object retrieves the description of the current location (i.e. storage area). It is possible to access more complex properties like UDAs, choice elements, and flexible state models:

- UDAs can be accessed similarly to standard properties, just add UDA_<property name>.

Example: **ControlRecipe:ProcessOrderItem.Part.UDA_X_shortDescription**

- To access choice elements append (CE) to the property name. Then, any property of the **IMESChoiceElement** object, like **Meaning** (represents the internal identifier/meaning) or **LocalizedName** (represents the localized name/item), can be used.

Example:

ControlRecipe:ProcessOrderItem.Part.UDA_X_materialType(CE).Meaning

- To access states of flexible state models add **FSM[]** and the FSM relationship name. Then, any property of the **ObjectState** object can be used.

Example:

ControlRecipe:ProcessOrderItem.UDA_X_batch.FSM[BatchQuality].Name

TIP

The availability of context-related functions is not part of their configuration but depends on the root object from which the property is retrieved.

However, as mentioned above there are different alias configurations for the **ControlRecipe** object which can be used to define for which **RecipeStructureType** the property is available.

Properties retrieved from the **ControlRecipe** object are available on all hierarchy levels, for example, whereas properties retrieved via the **IMESRoomEquipment** (room) object are only available on phase level. On the other hand, properties retrieved from the **IMESRoomEquipment** (room) object are available for all recipe structure types, but properties retrieved from the **ControlRecipe** object (configured via the alias **ControlRecipe**) are not available for workflows.

The optional **groupId** attribute (here: **100_CustomFunctionsGroup**) is used to define the group in the Functions tree to which the function is added. The groups itself is sorted by the group identifier. A default sorting criterion is available with a three-letter integer number as group identifier prefix. PharmaSuite comes with the **010_Structure**, **020_Measure**, **030_Aggregation**, **040_Conversion**, **050_OrderContext**, **060_OSMFCItemContext**, **070_Equipment**, and **080_LocationContext** group identifiers. Hence, the **100_CustomFunctionsGroup**, starting with the prefix of 100, is appended at the end of the Functions tree. Functions without a group message ID are added at the bottom of the tree.

The **sortIndex** attribute is an integer that sorts the functions within a group in the Functions tree.

Enabling Localization and Online Help

The final step is to enable localization and online help.

To enable localization of the function name, in the **ui_ExpressionEditor** message pack, define a message for the ID of the new function

(**Language_Function_[functionName]_Label**, here:

Language_Function_customContextFunction_Label). The localized function name determines the first component of a node label in the Functions tree panel.

If a custom **groupId** has been defined, localization of the function group identifier is also enabled when you define a message for the identifier of the new group in the

ui_ExpressionEditor message pack

(**LanguageTree_Functions_Grp_[groupId]_Label**, here:

LanguageTree_Functions_Grp_100_CustomFunctionsGroup_Label).

The tooltip is automatically generated by the system, there is no need to configure it.

Optionally, you can define a specific help page for the new function. By default, the help page for the Functions tree panel is shown. To link a specific help page, in the

OnlineHelpLinkage_ui_RecipeDesigner message pack, enter its URL in the message for the function ID (**ALL_ExpressionEditor_[functionName]**, here:

ALL_ExpressionEditor_customContextFunction). See also "Adapting User Documentation" (page 165).

TIP

The **ALL** prefix indicates that the help link is valid for all modes of Recipe and Workflow Designer.

Providing an Arbitrary Function

To provide an arbitrary function for the Expression editor, perform the following steps:

1. In Java, create a function evaluation class that holds the implementation of the function (page 68).
2. In Java, implement a type inference class for the function (page 69).
3. Export the function via an annotation (page 70).
4. Configure the application to use the function evaluation class (page 71).
5. Enable localization and online help (page 72).

In the subsequent sections, the steps are described with a complete example. We will add an **if-then-else** function to the Expression editor. The function expects three arguments, a **boolean condition** value, a **then value**, and an **else value**. The semantics are as follows: If the condition is **true**, the function returns the **then value**. If the condition is **false**, the function returns the **else value**. If the condition is **null** (i.e. undefined), the result is also **null**.

Creating a Function Evaluation Class

The default function evaluation class of Recipe and Workflow Designer is *ExpressionEval*, for Data Manager it is *StatusGraphExpressionEvaluation*. A custom function evaluation class must be derived from the default class.

In your custom class, add a public method that implements the new function. The name that you choose for the method will be the function name that has to be used in the Expression editor later (here: **ifThenElse**).

Example of the function evaluation class for Recipe and Workflow Designer

```
/**
 * Custom function evaluation class for the expression editor of the recipe designer.
 */
public class ExpressionEvalEx extends ExpressionEval {
    /**
     * Implements an if-then-else function.
     * @param condition condition
     * @param thenValue value to return if condition is fulfilled
     * @param elseValue value to return if condition is not fulfilled
     * @param <T> value type
     * @return <code>thenValue</code> if <code>condition</code> is
     *         <code>true</code>, <code>elseValue</code> if
     *         <code>condition</code> is <code>>false</code>, and
     *         <code>null</code> if <code>condition</code> is <code>null</code>
     */
    public <T> T ifThenElse(Boolean condition, T thenValue, T elseValue) {
        if (condition == null) {
            return null;
        }
        return condition ? thenValue : elseValue;
    }
}
```

TIP

Your function implementation must not have any side-effects, i.e. it must not perform any write operations like changing the database.

Implementing a Type Inference Class

A type inference class serves two purposes:

- Calculates a type descriptor for the result type of the function (based on the types of the arguments).
- Checks that the actual arguments match the function signature, i.e. the expected arguments.

In our **if-then-else** example, we have to check that the first argument is a **boolean** value. Additionally, we must ensure that the second and the third argument have the same type. (However, the type itself is arbitrary.) The result type of the function is the common type of the two arguments.

A function type inference class must implement the *IExpressionFunctionTypeInferencer* interface. The interface consists of the *inferResultTypeForFunctionCall* single method. This method receives a support instance facilitating the implementation of the method.

The result type descriptor does not only state the data type of the result value (i.e. the Java class), it also indicates whether the function can return a null value (i.e. undefined). In our example, we make the following inferences: If the condition can be null, the result can also be null, no matter which values the second and the third argument can assume. If the condition cannot be null, we inspect the second and the third argument. If any of them can be null, so can be the result. (This explains the initialization of the *nullableCalcStrategy* variable.)

Example of the type inference class

```
/**
 * Type inferencer for the if-then-else function.
 */
public final class FunctionTypeInferencerIfThenElse implements
IExpressionFunctionTypeInferencer {

    /** index of the condition argument */
    private static final int ARG_IDX_CONDITION = 1;

    /** index of the then argument */
    private static final int ARG_IDX_THEN = 2;

    /** index of the else argument */
    private static final int ARG_IDX_ELSE = 3;

    @Override
    public IExpressionTypeDescriptor
        inferResultTypeForFunctionCall(IExpressionFunctionTypeInferenceSupport support) {
```

```
// Make sure that the first argument is a boolean value:
IExpressionTypeDescriptor conditionType =
    support.checkArgumentHasType(ARG_IDX_CONDITION, Boolean.class);

// The 2nd and the 3rd argument can have an arbitrary type but both must
// have the same type.
// The result type is the same type as the type of the 2nd and 3rd argument.
// The result is nullable if at least one of the three arguments can be null.
NullableCalcStrategy nullableCalcStrategy =
    conditionType.isNullable() ? NullableCalcStrategy.ALWAYS :
    NullableCalcStrategy.ANY;
IExpressionTypeDescriptor resultType =
    support.checkArgumentsInRangeHaveSameType(ARG_IDX_THEN, ARG_IDX_ELSE,
        nullableCalcStrategy);
return resultType;
}
}
```

Exporting the New Function via Annotation

In order to export the new function, add an annotation to its implementation in the function evaluation class. The annotation also connects the function to its type inference class.

```
public class ExpressionEvalEx extends ExpressionEval {

    @ExpressionFunction(typeInferencerClass = FunctionTypeInferencerIfThenElse.class,
        groupId = "100_CustomFunctionsGroup")
    public <T> T ifThenElse(Boolean condition, T thenValue, T elseValue) {
        ...
    }
}
```

Besides the type inference class, the annotation has no mandatory attribute. In the example, we use the optional *groupId* attribute. For details, see section "Enabling Localization and Online Help" (page 72).

TIP

For further optional attributes, please refer to the "PharmaSuite-related Java Documentation" [C1] (page 171).

Some functions may not be applicable in all modes of Recipe and Workflow Designer. For example, some order context functions do not make sense for master workflows. Hence they are deactivated in Workflow Designer. This deactivation can be done with the optional *@RDEExpressionFunctionScope* annotation. (If the annotation is omitted, then there is no scope restriction for the respective function.) The scope for Recipe and Workflow Designer can be limited in the following areas:

- Structure type: master recipe, master workflow, building block.
- Structure subtype (for master recipes): batch or discrete (device).
- Hierarchy level: procedure, unit procedure, operation, phase.
- Usage: input expression, transition condition, flexible equipment rules.

For details, please refer to the "PharmaSuite-related Java Documentation" [C1] (page 171).
For Data Manager, the scope cannot be limited.

CONSIDER THE EQUIPMENT-SPECIFIC USAGE LIST

If the new function has a constant equipment class identifier as parameter and the equipment-specific usage list shall consider the new function, add the optional `@ExpressionFunctionArgumentS88EquipmentClassIdentifier` annotation to mark the function. For more information on the equipment-specific usage list, see "Data Manager User Documentation" [C3] (page 171).

```
public class ExpressionEvalEx extends ExpressionEval {
    @ExpressionFunction(typeInferencerClass =
        FunctionTypeInferencerEquipmentIsMemberOfClass.class, groupId = GRP_EQUIPMENT,
        sortIndex = EQUIPMENT_IS_MEMBER_OF_CLASS_SORTINDEX)
    @ExpressionFunctionArgumentS88EquipmentClassIdentifier(argumentIndex = 2)
    @SuppressFBWarnings(value = "NP_BOOLEAN_RETURN_NULL", justification =
        "necessary to distinguish between valid and invalid evaluation result")
    public Boolean equipmentIsMemberOfClass(final IMESS88Equipment equipmentEntity,
        final String equipmentClassIdentifier)
    { ... }
}
```

Configuring PharmaSuite

Now that the preparatory steps are completed, you can configure PharmaSuite to use the new function evaluation class instead of the default one.

The actions to take here are basically the same as described in section "Configuring Service Implementations" in chapter "Managing Services" in Volume 2 of the "Technical Manual Configuration and Extension" [A3] (page 171). We repeat them here for completeness.

Proceed as follows (with myc for the My Company vendor code):

1. Create the *myc-expr-functions.xml* file. In the *implementationClass* section, specify the fully qualified name of your function evaluation class:
ExpressionEvalEx for Recipe and Workflow Designer and
StatusGraphExpressionEvaluationEx for Data Manager.

```
<configuration>
  <ExpressionEval>
    <implementationClass>
      com.rockwell.mes.apps.ebr.ifc.ExpressionEvalEx
    </implementationClass>
  </ExpressionEval>
</configuration>
```

2. Create the *myc-config.xml* file. It is a copy of the default *config.xml* file extended by a reference to *myc-expr-functions.xml*. (It is important that this reference is

right at the beginning to make sure that later definitions are overridden.)

Example: *myc-config.xml* with reference to *myc-expr-functions.xml*.

myc-expr-functions.xml

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!-- Common config.xml including all service definitions -->
<configuration>
  <hierarchicalXml fileName="myc-expr-functions.xml" />
  <hierarchicalXml fileName="apps-wd.xml" />
  <hierarchicalXml fileName="apps-wip.xml" />
  <hierarchicalXml fileName="apps-recipeeditor.xml" />
  <hierarchicalXml fileName="clientfw-commons.xml" />
  <hierarchicalXml fileName="clientfw-grapheditor.xml" />
  <hierarchicalXml fileName="clientfw-pec.xml" />
  <hierarchicalXml fileName="clientfw-pmc.xml" />
  <hierarchicalXml fileName="commons-base.xml" />
  <hierarchicalXml fileName="commons-deviation.xml" />
  <hierarchicalXml fileName="commons-versioning.xml" />
  <hierarchicalXml fileName="commons-security.xml" />
  <hierarchicalXml fileName="commons-audittrail.xml" />
  <hierarchicalXml fileName="services-egm.xml" />
  <hierarchicalXml fileName="services-inventory.xml" />
  <hierarchicalXml fileName="services-recipe.xml" />
  <hierarchicalXml fileName="services-s88.xml" />
  <hierarchicalXml fileName="services-wd.xml" />
  <hierarchicalXml fileName="services-wip.xml" />
  <hierarchicalXml fileName="services-order.xml" />
</configuration>
```

3. In Process Designer, open your application configuration and set the value of the **LibraryHolder/commons-base-ifc.jar/ServicesConfigFile** configuration key to *myc-config.xml*.

TIP

Both XML configuration files and the two Java classes (function evaluation and type inference class) must be contained in one or more Jar files defined as Library/Libraries in Process Designer.

Enabling Localization and Online Help

The final step is to enable localization and online help.

To enable localization of the function name, in the **ui_ExpressionEditor** message pack, define a message for the ID of the new function (**if-then-else**, here:

Language_Function_ifThenElse_Label). The localized function name determines the first component of a node label in the Functions tree panel (here: **If-Then-Else**).

In the annotation (page 70), we defined the **100_CustomFunctionsGroup** group message ID. Now, we have to specify a message for this group identifier (here:

LanguageTree_FunctionsTree_Grp_100_CustomFunctionsGroup_Label) also in the **ui_ExpressionEditor** message pack (here: **Custom functions**). Then, a group node appears in the Functions tree panel located between the root node and the node for our

new function.

If you define the same group message ID for several functions, they will be grouped underneath the group node in the tree. The group nodes are sorted alphabetically by their group identifier. A default sorting criterion is available with a three-letter integer number as group identifier prefix. PharmaSuite comes with the **010_Structure**, **020_Measure**, **030_Aggregation**, **040_Conversion**, **050_OrderContext**, **060_OSMFCItemContext**, **070_Equipment**, and **080_LocationContext** group identifiers. Hence, the **100_CustomFunctionsGroup**, starting with the prefix of 100, is appended at the end of the Functions tree. Functions without a group message ID are added at the bottom of the tree.

The tooltip is automatically generated by the system, there is no need to configure it.

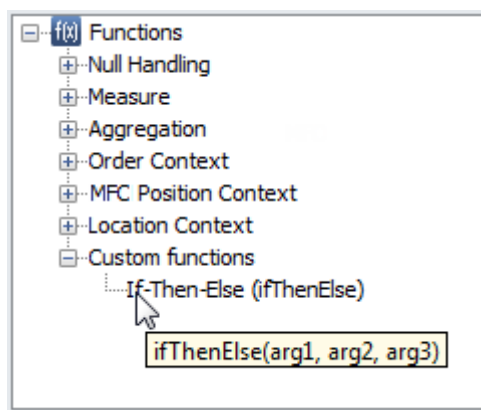


Figure 21: Custom functions in the Functions tree panel

Optionally, you can define a specific help page for the new function. By default, the help page for the Functions tree panel is shown. To link a specific help page, in the **OnlineHelpLinkage_ui_RecipeDesigner** message pack, enter its URL in the message for the function ID (here: **ALL_ExpressionEditor_ifThenElse**). For Data Manager, use the **OnlineHelpLinkage_ui_MasterDataEditor** message pack. See also "Adapting User Documentation" (page 165).

TIP

The **ALL** prefix indicates that the help link is valid for all modes of Recipe and Workflow Designer or Data Manager, respectively.

Using ERP BOMs in Recipe and Workflow Designer

This section describes how to enable the usage of ERP BOMs in Recipe Designer (page 40) in case your recipes are based on BOMs that are stored in a superordinate ERP system.

The following prerequisites apply:

- ERP BOMs are stored in the PharmaSuite database.
- Materials that are referenced by ERP BOMs exist in the PharmaSuite database.

In PharmaSuite, ERP BOMs are read-only.

When you create a new recipe in Recipe Designer, ERP BOMs can be referenced. Thus, the system supports the recipe author with creating recipes that correspond to given ERP BOMs. Specific checks enforce the correspondence between the master recipe's material parameter attributes and the given ERP BOM. See ERP BOM-related checks of the **LibraryHolder/apps-recipeeditor-impl.jar/RecipeStructureChecks** configuration key in chapter "Configuration Keys of PharmaSuite" in Volume 4 of the "Technical Manual Configuration and Extension" [A4] (page 171).

Technical Details

The *IS88RecipeService* interface (*com.rockwell.mes.services.s88.ifc*) provides methods to create and read ERP BOM data. For further details, please refer to the "PharmaSuite-related Java Documentation" [C1] (page 171).

The interfaces of the objects (*IMESERPomHeader*, *IMESERPomItem*) and filters (*IMESERPomHeaderFilter*, *IMESERPomItemFilter*) can be found in *com.rockwell.mes.services.s88.ifc.recipe*.

The associated application tables are **X_ERPBomHeader** und **X_ERPBomItem**.

The referenced materials must be stored in the FactoryTalk ProductionCentre **Part** table.

Configuring the Initialization of Material Parameters in Recipe and Workflow Designer

This section describes how to configure the initialization of custom attributes added to the material parameters.

When a material parameter object is created in Recipe and Workflow Designer, the system initializes the weighing attributes (lower and upper tolerance, potency, etc.) with the corresponding values of the **Part** (material) object. Then, the system performs two copy operations: first each named UDA of the **Part** (material) object and secondly each column of the **X_ERPBomItem** application table to the columns of the **X_MaterialParameter** application table with the same name. This applies only to material input parameters. Furthermore, the system copies each column of the **Part** (material) object with the same name as named UDA to the **ProcessBomItem** object. The system behavior applicable to custom attributes is contained in a class implementing the *IMaterialParameterCustomAttributesHandler* interface.

If you wish to replace the system behavior regarding custom attributes, write your own class or extend the *MESMaterialParameterCustomAttributesHandler* class implementing the *IMaterialParameterCustomAttributesHandler* interface and configure your class according to your requirements, for example, initialize material output parameters or define a default value.

Using the *IMaterialParameterCustomAttributesHandler* Interface

The *IMaterialParameterCustomAttributesHandler* interface defines methods to copy custom attributes of the **Part** (material) object to the material parameter object and from the material parameter object to the **ProcessBomItem** object. Additionally, custom attributes can be initialized.

```
public interface IMaterialParameterCustomAttributesHandler {

    /**
     * Initialize custom attributes of a newly created Material Parameter e.g.
     * by copying custom attributes of the part.
     *
     * The standard implementation copies all named UDAs of the Part
     * (<code>parameter.getMaterial()</code>)
     * with same name to the corresponding column of the MaterialParameter's
     * ApplicationTable for all input material parameters
     * (<code>(parameter.getType() == TYPE.INPUT) </code>)
     *
     * Only touch custom attributes of Material Parameter, otherwise
     */
}
```

```

    * this may lead to unexpected system behavior.
    *
    * @param material the corresponding material of the material parameter
    * @param parameter the material parameter object
    */
public void copyCustomAttributes(Part material, IMESMaterialParameter parameter);

/**
 * Copy custom attributes of the material parameter object to the ProcessBomItem.
 *
 * The standard implementation copies all named UDAs of the material parameter object
 * with the same name to the corresponding named UDA of the ProcessBomItem.
 *
 * @param parameter the material parameter object
 * @param bomItem the target ProcessBomItem
 */
public void copyCustomAttributes(IMESMaterialParameter parameter,
                                ProcessBomItem bomItem);

/**
 * Copy custom attributes of the ERP BOM item to the material parameter.
 *
 * The standard implementation copies all AT-rows of the ERP BOM item with the same
 * name to the corresponding named AT-rows of the material parameter object.
 *
 * @param erpBomItem the source ERP BOM item
 * @param parameter the material parameter object
 * @param copyNullValues If true, copy null values, otherwise skip null values
 */
void copyCustomAttributes(IMESERPItem erpBomItem,
                          IMESMaterialParameter parameter, boolean copyNullValues);
}

```

TIP

In the current version of PharmaSuite, **ProcessBom** and **ProcessBomItem** objects are not visible, however they are still components of master recipes. The **ProcessBomItem** objects provide the data of the order step inputs. That is why the *copyCustomAttributes* handler is important to transfer data from material parameters (master data) to order step inputs (processing data).

Configuring the Custom Attributes Handler for Material Parameters

The implementation class for the custom attributes handler for material parameters can be configured by means of a configuration key in the **Application** object (Default configuration) in Process Designer. For details, see

LibraryHolder/services-s88-ifc.jar/MaterialParameterCustomAttributesHandler configuration key in Volume 4 of the "Technical Manual Configuration and Extension" [A4] (page 171).

Example Implementation

The example initializes the **MYC_example** custom attribute with the corresponding UDA of the **Part** (material) object. If no value is given in the **Part** (material) object, **MYC_example** is initialized with a default value.

```
public class MyCustomerMaterialParameterInitializer extends
    MESMaterialParameterCustomAttributesHandler {

    @Override
    public void copyCustomAttributes(Part material, IMESMaterialParameter parameter) {
        if (parameter.getType() == TYPE.INPUT) {
            if (material.getUDA("MYC_example") == null) {
                parameter.getATRow().setValue("MYC_example", defaultValue);
            } else {
                parameter.getATRow().setValue("MYC_example", material.getUDA("MYC_example"));
            }
        }
    }
}
```

TIP

The standard implementation copies all of the matching UDAs. If you configure your own class, you replace the standard implementation. Make sure your implementation copies all of your custom attributes. We recommend to extend the *MESMaterialParameterCustomAttributesHandler* class used by the system. Calling methods of the superclass ensures this copy behavior and allows you to extend it.

Configuring Menu and Toolbar of Recipe and Workflow Designer

This section contains general information about Recipe and Workflow Designer actions (page 81), implementing action classes (page 82), and the configuration of the menus and toolbars (page 84).

To replace an existing action implementation, perform the following steps:

1. Implement a new action class (page 82) containing the adapted behavior of an existing action.
2. Modify the configuration (page 82) in order to use the new action class implementation.
3. Run the application to verify the changes in Recipe and Workflow Designer and its actions.

To add a new action, perform the following steps:

1. Implement a new action class (page 82).
2. Add the configuration for the new action class (page 82).
3. Run the application to verify the changes in Recipe and Workflow Designer and its actions.

What Are Actions in Recipe and Workflow Designer?

In Recipe and Workflow Designer, most functions are called by actions, located in the menus or the toolbars.

An example for an action is the **Exit** action in the **File** menu or the **New master recipe** action in the toolbar.

The functionality and business logic of an action is encapsulated in a component, here a Java class.

Depending on your requirements, you either may implement a new action class containing the adapted behavior of an existing action or implement a new action class.

Implementing an Action Class

An action class has to extend the *RDAbstractExternalAction* class. The implementation of the functionality of the action is located in the *actionPerformed* method. Each time the context changes (e.g. a master recipe was loaded, the user switched from one opened master recipe to another), the update method is called. Here, you can implement the logic to react to those events.

Example: The *MYCMenuAction* action class simply executes some kind of business logic with MYC for the My Company vendor code. The action is only sensitive when a master recipe is loaded and not modified.

```
package com.rockwell.mes.cust.impl;
import java.awt.event.ActionEvent;
import com.rockwell.mes.apps.recipeeditor.ifc.frame.action.RDAbstractExternalAction;
import com.rockwell.mes.apps.recipeeditor.ifc.frame.model.IS88File;
import com.rockwell.mes.clientfw.docking.ifc.editor.MESAbstractEditorInstance;
/**
 * Customer menu/toolbar action.
 */
public class MYCMenuAction extends RDAbstractExternalAction {
    /**
     * Comment for <code>serialVersionUID</code>
     */
    private static final long serialVersionUID = 1L;
    /**
     * Constructor.
     *
     * @param name action name
     * @param editor editor instance
     */
    public MYCMenuAction(String name, MESAbstractEditorInstance editor) {
        super(name, editor);
    }
    @Override
    public void actionPerformed(ActionEvent e) {
        System.out.println("I am a customer action.");
    }
    @Override
    public void update() {
        final IS88File s88File = getFocusedS88File();
        final boolean isS88FileFocused = (s88File != null) && (!s88File.wasDestructed());
        final boolean isS88FileUnmodified = isS88FileFocused && !s88File.isSaveable();
        setEnabled(isS88FileUnmodified && s88File.isMasterRecipe());
    }
}
```

Configuring Action Classes

Before you can use an implementation class of an action, the class must be configured. The configuration of implementation classes of Recipe and Workflow Designer actions is held in the **red_buttonConfiguration** list object.

To add a new action class or to replace an existing action class, a configuration entry is required. The XML file below contains such an entry in the *<item>* XML element with a

unique key and the qualified name of the action itself (here: *com.rockwell.mes.cust.impl.MYCMenuAction*).

To replace an existing action class, change the value of the implementation class to the implementation class of your action. In order to add a new action class, add a new XML element that refers to the new action.

You can also add a new sub-menu to the main menu or a new toolbar to the set of toolbars and place your action there.

New sub-menu:

```
<menu key="mycMenu">
<item key="mycMenuAction" action="com.rockwell.mes.cust.impl.MYCMenuAction"/>
</menu>
```

New toolbar:

```
<toolbar key="mycToolbar ">
<item key="mycToolbarAction" action="com.rockwell.mes.cust.impl.MYCToolbarAction"/>
</toolbar>
```

Since Recipe and Workflow Designer supports different modes (Recipe Designer - Batch, Recipe Designer - Device, and Workflow Designer), it is possible to specify an action to be only available in one mode. By default, an action is available in all modes.

```
<item key="newMasterRecipe" perspective="RECIPE,DISCRETE_RECIPE" />
<item key="newWorkflow" perspective="WORKFLOW" />
```

Configuring Actions

In order to use an action in Recipe and Workflow Designer, some details related to the action have to be defined. The details depend on the usage of the action (menu or toolbar).

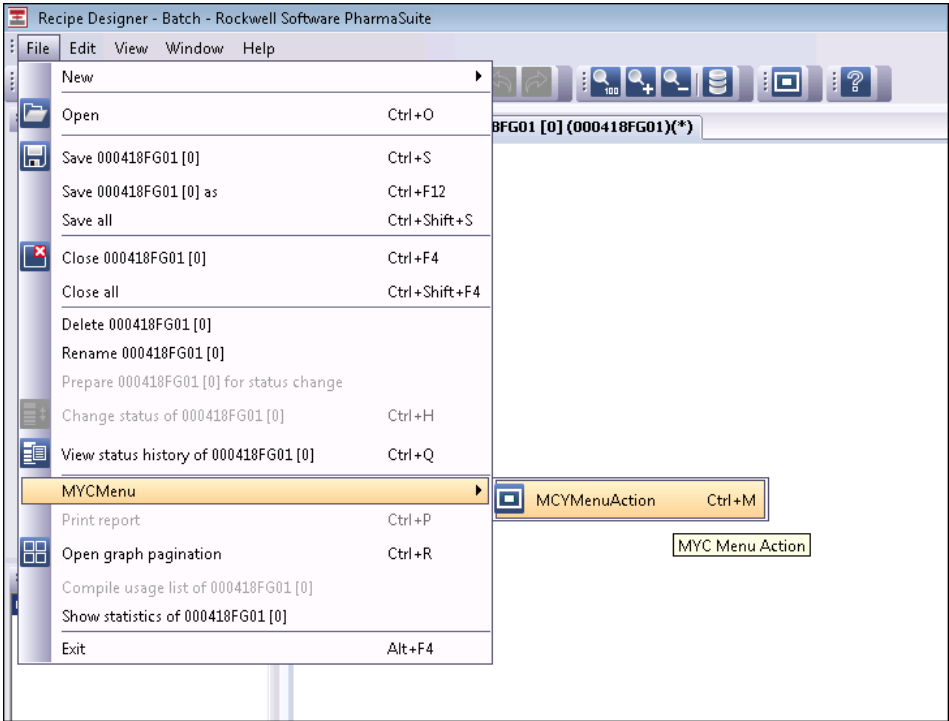


Figure 22: MYCMenuAction action in the MYCMenu sub-menu

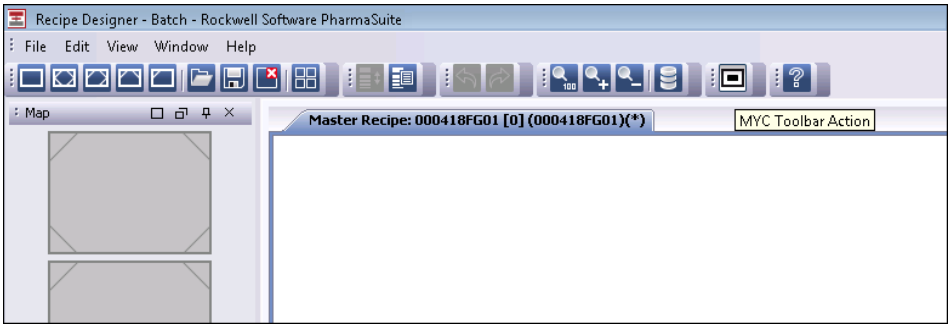


Figure 23: MYCToolbarAction action toolbar MYCToolbar toolbar

Configuring an Action for the Menu Bar

The following properties must be configured to define a menu action:

Property	Value
Label (sub-menu)	Message pack: ui_RecipeDesigner Entry: <key> + "_Menu"

Property	Value
Label (action)	Message pack: ui_RecipeDesigner Entry: <key> + "_Menu"
Tooltip	Message pack: ui_RecipeDesigner Entry: <key> + "_Hint"
Accelerator	Message pack: ui_RecipeDesigner Entry: <key> + "_Accelerator"
Icon	Image: "red_" + <key>
Icon (toggled)	Image: "red_" + <key> + "_toggled"

Configuring an Action for the Toolbar

The following properties must be configured to define a toolbar action:

Property	Value
Label (only visible if undocked)	Message pack: ui_RecipeDesigner Entry: <key> + "_Toolbar"
Icon	Image: "red_" + <key>
Icon (toggled)	Image: "red_" + <key> + "_toggled"
Tooltip	Message pack: ui_RecipeDesigner Entry: <key> + "_Hint"

Adding a Context to an Action

For some actions it is useful to visualize context information with the label of a menu item or a toolbar button. For example, the name of a currently selected master recipe can be used as context information.

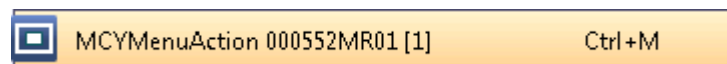


Figure 24: MYCMenuAction with context information

To add the context to the action label, first enhance your action:

```
package com.rockwell.mes.cust.impl;
import java.awt.event.ActionEvent;
import com.rockwell.mes.apps.recipeeditor.ifc.frame.action.RDAbstractExternalAction;
import com.rockwell.mes.apps.recipeeditor.ifc.frame.model.IS88File;
import com.rockwell.mes.clientfw.docking.ifc.editor.MESAbstractEditorInstance;
/**
 * Customer menu/toolbar action.
 */
public class MYCMenuAction extends RDAbstractExternalAction {
    /**
     * Comment for <code>serialVersionUID</code>
     */
    private static final long serialVersionUID = 1L;
```

```

/**
 * File name of the currently focused recipe
 */
private String currentFileName;
/**
 * Constructor.
 *
 * @param name action name
 * @param editor editor instance
 */
public MYCMenuAction(String name, MESAbstractEditorInstance editor) {
    super(name, editor);
}
@Override
public void actionPerformed(ActionEvent e) {
    System.out.println("I am a customer action.");
}
@Override
public void update() {
    final IS88File s88File = getFocusedS88File();
    final boolean isS88FileFocused = (s88File != null) && (!s88File.wasDestructed());
    final boolean isS88FileUnmodified = isS88FileFocused && !s88File.isSaveable();
    setEnabled(isS88FileUnmodified && s88File.isMasterRecipe());
    setCurrentFile(s88File);
}
@Override
public List<String> getLabelContext() {
    if (currentFileName == null) {
        return null;
    }
    return Collections.singletonList(currentFileName);
}
/**
 * @param newValue The last recent file.
 */
protected final void setCurrentFile(IS88File newValue) {
    final List<String> oldLabelContext = getLabelContext();
    currentFileName = newValue != null ? newValue.getFileName() : null;
    final List<String> newLabelContext = getLabelContext();
    firePropertyChange(PROPERTY_LABEL_CONTEXT, oldLabelContext, newLabelContext);
}
}

```

Now, you only have to add a placeholder for the context ("{0}") to the corresponding message pack entries ("_Menu" or "_Hint").

Example: MYC Menu Action {0}

Configuring the Details Window of Data Manager - Work Center

This section contains general information about the Details window (page [87](#)) and how to adjust the configuration of the **Basic** tab by using standard attributes (page [88](#)) and custom attributes (page [89](#)).

You can configure if a specific property shall be visible in the **Basic** tab of the Details window and in which row of the Details window's grid a property shall be displayed.

To configure the Details Window, you have to be familiar with

- Process Designer
- Data Dictionary Manager (see chapter "Adapting and Adding Field Attributes" in Volume 2 of the "Technical Manual Configuration and Extension" [A3] (page [171](#))).
- Data Manager

What Is the Details Window?

The **Basic** tab of the Details window provides access to the key properties of a master data object.

The following objects are available in Data Manager - Work Center:

- Work center
- Station

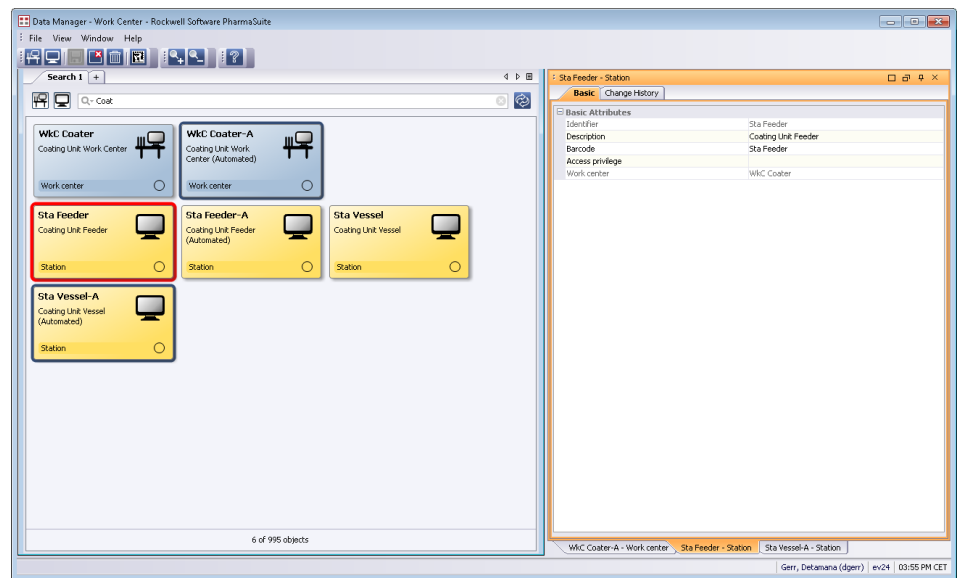


Figure 25: Basic tab of Details window for stations

Adjusting the Configuration

To adjust the configuration of the **Basic** tab of the Details window, open the Data Dictionary Manager and proceed as follows:

1. Navigate to the **Class management** tab.
In the **Data Dictionary Classes** panel, in the **Class-related actions** panel, select one of the supported data objects from the drop-down list:
 - Work center (*MESWorkCenter* class)
 - Station (*MESStation* class).
2. In the **Data Dictionary Elements** panel, in the **Element list** panel, select a data dictionary element (e.g. **description**).
3. Set the properties according to your needs. The following properties are relevant to the Details window:
 - **Hidden**
 - **Category**
 - **Grid order**
 - **Cell-editor factory** (optional)
 - **Choice list** (optional, only for choice elements)
 - **Length** (optional, only for strings)

Adjusting the Configuration of Custom Attributes

As a prerequisite, the UDA list of the related types must be extended by new UDAs. For details, see section "Adding Field Attributes by Means of UDAs" in chapter "Adapting and Adding Field Attributes" in Volume 2 of the "Technical Manual Configuration and Extension" [A3] (page 171).

- For work centers: extend the UDA list of the **WorkCenter** object.
- For stations: extend the UDA list of the **Station** object.

To adjust the configuration of UDA custom attributes of the Details window, open the Data Dictionary Manager and proceed as follows:

1. Navigate to the **Class management** tab.
In the **Data Dictionary Classes** panel, in the **Class-related actions** panel, from the drop-down list, select one of the custom configuration classes indicated by the **CustomerConfig** appendix:

- Work center (*MESWorkCenterCustomerConfig* class)
- Station (*MESStationCustomerConfig* class).

2. In the **Data Dictionary Elements** panel, in the **Element list** panel, select a data dictionary element.
If the custom attribute is not available as data dictionary element, in the **Element-related actions** panel, type the name of the attribute (e.g. **UDA_Y_newProperty**) in the text box and then click the **Add element** button.

TIP

The attribute name of a UDA is prefixed "UDA_" plus the UDA name.

3. Set the properties according to your needs. The following properties are relevant to the Details window:

- **Hidden**
- **Category**
- **Grid order**
- **Cell-editor factory** (optional)
- **Choice list** (optional, only for choice elements)
- **Length** (optional, only for strings)

TIP

Unless you localize the new UDA in the corresponding message pack, the Details window uses the name of the UDA as label on the user interface in Data Manager.

To localize the new UDA, for example, of the station object, open the **DataDictionary_Default_StationCustomerConfig** message pack (= **"DataDictionary_Default_"**+<class name>). In our example, the UDA is named **MYC_testString**). Add the **UDA_MYC_testString_Label** message for the UDA's label and the **UDA_MYC_testString_Hint** message for additional information.

Now, the Details window displays the messages defined in the message pack.

Configuring the Change History of Data Manager - Work Center

This section contains general information about the change history (page 91) of work center objects (work center, station). It covers how to adjust the configuration by using standard attributes (page 92) and custom attributes (page 92). Additionally, it provides details about monitoring work center objects in non-PharmaSuite applications (page 94).

To configure the change history, you have to be familiar with

- Process Designer
- Data Dictionary Manager (see chapter "Adapting and Adding Field Attributes" in Volume 2 of the "Technical Manual Configuration and Extension" [A3] (page 171)).
- Data Manager

Which Events Are Tracked by the Change History?

The change history tracks the following events:

Event	Work center	Station
Created	Yes	Yes
Deleted	Yes	Yes
Station assigned/removed	Yes	N/A
Work center assigned/removed	N/A	Yes
Attribute updated (Editable and not Hidden attributes only)	Yes	Yes
Property assigned/removed	Yes	Yes
Property value updated	Yes	Yes
Storage area changed	Yes	N/A
Access privilege changed	N/A	Yes

Tracking applies to all attributes that are editable and visible in the **Details** window (property pane) of the affected object types in Data Manager - Work Center. In this case visible means that the **Hidden** property is set to **No** in the Data Dictionary Manager for the *MESWorkCenter* and *MESStation* classes. Therefore, the settings of the **Hidden**

property made in the Data Dictionary Manager define the events tracked in the change history.

Adjusting the Configuration

To adjust the configuration of the change history, open the Data Dictionary Manager and proceed as follows:

1. Navigate to the **Class management** tab.
In the **Data Dictionary Classes** panel, in the **Class-related actions** panel, select one of the supported data objects from the drop-down list:
 - Work center (*MESWorkCenter* class)
 - Station (*MESStation* class).
2. In the **Data Dictionary Elements** panel, in the **Element list** panel, select a data dictionary element (e.g. **description**).
3. Set the properties according to your needs. The following properties are relevant to the change history:
 - **Hidden**
 - **MES choice list name**

TIP

To support internationalization of a choice list used in the change history, configure the name of the choice list in the corresponding Data Dictionary element. You must configure the name of the choice list at the Data Dictionary element for *<attribute name>AsValue*.

Adjusting the Configuration of Custom Attributes

If the Data Dictionary of a customer configuration contains custom attributes, the attributes will only be tracked if they are visible (**Hidden** property is set to **No**). Custom prefixes (e.g. *MYC_*) of column names are supported. Example: *MYC_myAttribute* is the database column name and there is a Data Dictionary element for *myAttribute*. Then the system expects this element as the configuration of the column.

As a prerequisite, the UDA list of the related parameter types must be extended by new UDAs. For details, see section "Adding Field Attributes by Means of UDAs" in chapter "Adapting and Adding Field Attributes" in Volume 2 of the "Technical Manual Configuration and Extension" [A3] (page 171).

- For work centers: extend the UDA list of the **WorkCenter** object.
- For stations: extend the UDA list of the **Station** object.

IMPORTANT

UDAs with list support are not supported by the change history.

To adjust the configuration of UDA custom attributes of the change history, open the Data Dictionary Manager and proceed as follows:

1. Navigate to the **Class management** tab.
In the **Data Dictionary Classes** panel, in the **Class-related actions** panel, from the drop-down list, select one of the custom configuration classes indicated by the **CustomerConfig** appendix:
 - Work center (*MESWorkCenterCustomerConfig* class)
 - Station (*MESStationCustomerConfig* class).
2. In the **Data Dictionary Elements** panel, in the **Element list** panel, select a data dictionary element.
If the custom attribute is not available as data dictionary element, in the **Element-related actions** panel, type the name of the attribute (e.g. **myAttribute** for AT columns or **UDA_myAttribute** for UDAs) in the text box and then click the **Add element** button.
3. Set the properties according to your needs. The following properties are relevant to the change history:
 - **Hidden**
 - **MES choice list name**

TIP

To support internationalization of a choice list used in the change history, configure the name of the choice list in the corresponding Data Dictionary element. You must configure the name of the choice list at the Data Dictionary element for *<attribute name>AsValue*.

To localize the new attribute, for example, of the station object, open the **DataDictionary_Default_StationCustomerConfig** message pack (= "DataDictionary_Default_" + <class name>). In our example, the attribute is named **MYC_testString**. Add the **MYC_testString_Label** message for the attribute's label.

Now, the change history tracks the changes related to the attribute using the localized name of the messages defined in the message pack.

Monitoring Work Center Objects in Non-PharmaSuite Applications

When you implement a new application and you wish to apply the change history for work centers and stations, consider the following fact: Before changing an existing persistent object, tracking of the corresponding object has to be enabled. Otherwise no changes are tracked in the change history. For newly created objects, monitoring is enabled by default.

When editing of the object is finished, monitoring should be disabled. This has to be done for newly created objects, too. Deletion of an object automatically disables its monitoring.

Tracked changes will become persistent when the corresponding object has been saved or deleted. To implement an undo of changes, it is possible to remove all tracked changes. Obviously, this has to be done before calling the save operation for the object.

IMESChangeHistoryOwner provides an API for this purpose.

```
/**
 * Start to monitor the station to create change history events.
 */
void startMonitorChangeHistoryEvents();

/**
 * Finish monitoring of the station. No change history events will be created anymore.
 * Not yet saved history events will be cleared.
 */
void finishMonitorChangeHistoryEvents();

/**
 * Clear all not yet saved change history events of the history owner.
 * This function is to be used if changes on the owner object will be restored.
 */
public void clearMonitoredChangeHistoryEvents();
```

Configuring FSMs for Equipment Properties

This section describes how to configure a flexible state model (FSM) in Process Designer in order to use it to create equipment property types and equipment properties (in Data Manager). It also describes how to configure semantic properties and assign them to FSM states, so they can be used in equipment property requirements (in Recipe and Workflow Designer).

Both FSMs and semantic properties can be configured to use a default filter or a custom filter to define sets of FSMs and semantic properties available for selection.

To configure FSMs and semantic properties, you have to be familiar with the

- **Flexible State Model** object of Process Designer [B1] (page [171](#))
- **Semantic Property** object of Process Designer [B1] (page [171](#))

The following requirements apply:

- The FSM which shall be made available in Data Manager already exists.

We highly recommend to use equipment-related graphs in Data Manager as preferred approach. FSMs can still be used, however, this approach is no longer recommended.

PharmaSuite uses the **S88EquipmentBinding** FSM internally to support the **Identify**, **Bind**, and **Release** actions on S88-specific equipment.

Configuring an FSM

To configure the FSM for Data Manager with the **s88_eqm_availableFSMsForPropertyType** default filter, proceed as follows in Process Designer:

1. Make the FSM available in Data Manager:
 1. Open the FSM.
 2. Set the **Category** property to **S88_eqm**.
 3. Save the FSM.
2. Localize the FSM **Name** and **Description** properties:
 1. Open the message pack associated with the FSM (see **messagePackName** property).

If there is no message pack name specified yet, create a new message pack and associate it with the FSM. The name of the message pack is the FSM name prefixed with **fsm_** (e.g. message pack of the **myc_MyModel** FSM is **fsm_myc_MyModel**).

2. In the message pack, create a message with **fsm_name** as message ID and provide the localized FSM name as message text.
3. Create a message with **fsm_description** as message ID and provide the localized FSM description as message text.
4. Save the message pack.

TIP

Unless you provide the **fsm_name** or **fsm_description** messages, Data Manager will use the non-localized name and description specified in the FSM itself as fall-back.

If you wish to configure another filter criterion for the FSM, proceed as follows in Process Designer:

1. Configure your own selection criteria in the **s88_eqm_availableFSMsForPropertyType** default filter.
2. Save the filter.

Configuring Semantic Properties

In the context of equipment property requirements, we use semantic properties (attached to FSM states) in order to specify in which state an equipment must be to fulfill the requirement.

In our example, the **myc_Initialization** FSM describes the initialization state of some equipment entities. The FSM has been configured according to section "Configuring an FSM" (page 95). Then, in Data Manager, create the **initialization** FSM property type based on the FSM. In Recipe and Workflow Designer, a recipe author can create an equipment property requirement based on the **initialization** property type for his master recipe. In the next step, the recipe author specifies that an initialized equipment entity is required for the execution of a phase. It may well be that the concept of an initialized equipment entity corresponds to more than one FSM states: **initialized-minor** and **initialized-major**. Hence, a recipe author needs to specify a state group instead of a single state in a property requirement. For this purpose, state groups can be used with support of semantic properties.

To provide and configure a new state group with the **s88_eqm_availableSemanticPropertiesForFSMPropertyType** default filter, proceed as follows in Process Designer:

1. Create a new **Semantic property** object and specify a name (here: **myc_Initialization**).
2. Set the **Category** property to **S88_eqm**.
3. Set the **Semantic type** property to **TYPE_PERMISSION**.
Keep **USED_FOR_STATE** as **Used for** property.
4. Save the semantic property.
5. Add the semantic property to an existing or a new semantic property set and save the set.
(This is necessary since you cannot add single semantic properties to an FSM.)
6. Open the FSM (here: **myc_Initialization**).
7. From the toolbar, choose the **Add Semantic Property Set to FSM** function and add your semantic property set to the FSM (if not already present).
8. Add a semantic property to a state:
 1. Select a state (here: **initialized-minor**).
 2. From the toolbar, choose the **Add Semantic Property to State or Transition Instance**.
 3. Expand the node corresponding to your semantic property set, select the semantic property (here: **myc_Initialization**), and click **OK**.
9. Repeat the previous step to add the semantic property to all states that shall belong to the property group.
10. Localize the semantic property **Name** property:
 1. Open the **fsm_SemanticProperties** message pack.
 2. Create a new message whose identifier is the non-localized name of the semantic property suffixed with **_name** (here: **myc_Initialization_name**) and provide the localized name as message text.

TIP

It is important to set the correct semantic type for a semantic property since Data Manager only accepts permission properties whose category contains **S88_eqm**.

If you wish to configure another filter criterion for the semantic properties, proceed as follows in Process Designer:

1. Configure your own selection criteria in the **s88_eqm_availableSemanticPropertiesForFSMPropertyType** default filter.
2. Save the filter.

Providing Images for Equipment Classes in Data Manager

This section describes how to provide images for equipment classes to be displayed in Data Manager by means of the **Image** object of Process Designer. The image will be displayed as an icon on the equipment classes' **Style** tab.

Images can be configured to use a default filter or a custom filter to define sets of images available for selection.

To provide an image for equipment classes, you have to be familiar with the

- **Image** object of Process Designer [B1] (page 171)

The following requirements apply to the image:

- It must be stored on the local file system.
- The file format must be JPG, GIF, or PNG.
- The recommended size is 48x48 pixels.

Providing an Image

To provide an image with the **s88_eqm_availableImagesForEquipmentClass** default filter, proceed as follows in Process Designer:

1. Create a new **Image** object.
2. Click the **Open** button to select the image file stored on the local file system.
3. Set the **Category** property to **S88_eqm**.
4. We recommend to use a vendor-specific prefix when specifying the name (e.g. **myc_reactor**).

In order to avoid displaying identical prefixes for all images, the longest common prefix of the image names that ends with an underscore is hidden on the user interface.

If you wish to configure another filter criterion for the images, proceed as follows in Process Designer:

1. Configure your own selection criteria in the **eqm_availableImagesForEquipmentClass** default filter.
2. Save the filter.

Providing Report Designs for (Template) Equipment Entities in Data Manager

This section describes how to provide report designs (used as label layouts) for (template) equipment entities to be selectable in Data Manager by means of the **ReportDesign** object of Process Designer. The report design will be used to print equipment entity labels during execution.

To provide a report design for (template) equipment entities, you have to be familiar with the

- **ReportDesign** object of Process Designer [B1] (page [171](#))
- Label layouts (see chapter "Changing or Adding New Labels" in Volume 2 of the "Technical Manual Configuration and Extension" [A3] (page [171](#))).
- Data Dictionary Manager (see chapter "Adapting and Adding Field Attributes" in Volume 2 of the "Technical Manual Configuration and Extension" [A3] (page [171](#))).

Providing a Report Design

To provide a report design, proceed as follows in Process Designer:

1. Create a new **ReportDesign** object.
2. Click the **Export** button to create the layout file on the local file system, edit the layout file with Jaspersoft Studio, import it from the local file system, and save and check in the report design.
3. Set the **Category** property to **S88_eqm**.
4. We recommend to use a vendor-specific prefix when specifying the name (e.g. **myc_layout**).

If you wish to configure another category for the report designs, proceed as follows in Process Designer:

1. Expand the **Lists** node and select the **Category** list.
2. Remove the read-only flag.
3. Add your new category (e.g.: **myc_layout**).

4. Save the list and set it to read-only again.
5. Run the **mes_DataDictManagerForm** form to start the **Data Dictionary Management** tool.
6. Navigate to the **Class management** tab.
In the **Data Dictionary Classes** panel, in the **Class-related actions** panel, select the **MESS88Equipment** DD class.
7. Click the **Load** button. If version control is enabled, click the **Check out** button.
8. In the **Data Dictionary Elements** panel, in the **Element list** panel, select the **labelReportDesign** DD element.
9. In the **Data Dictionary Element** panel, set the validation configuration to your new category (e.g.: **myc_layout**).
10. Click the **Save changes** button. If version control is enabled, click the **Check in** button.
11. Repeat steps 6 through 10 for the **S88TemplateEntity** DD class.
12. Close the **Data Dictionary Management** tool, then close the form.

Adding an Equipment Logbook Category Accessible by Phase Building Blocks

This section describes how to add an equipment logbook category and make this category accessible by phase building blocks executed in the Production Execution Client. With this extension a phase building block can store its own events in the equipment logbook using the added category. Subsequently you can filter the equipment logbook for entries related to the added category.

To add an equipment logbook category, you have to be familiar with

- Choice List Manager (see chapter "Adapting and Adding Field Attributes" in Volume 2 of the "Technical Manual Configuration and Extension" [A3] (page 171)).
- Process Designer

Extending the Category Choice List

To extend the choice list of the equipment logbook category, proceed as follows:

1. In Choice List Manager, add new categories (here: CUST-1, CUST-2) to your version of the **S88EquipmentLogbookCategory** choice list.

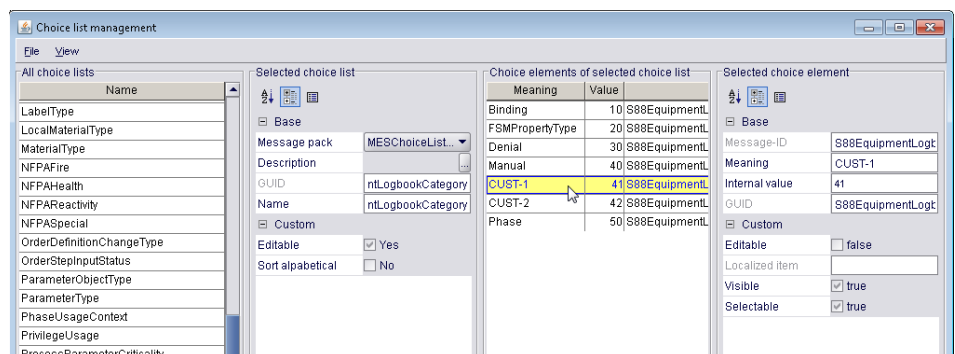


Figure 26: Adding a choice element in Choice List Manager

2. In Process Designer, adapt your version of the **MESChoiceListStrings** message pack to support your required locales.

The figure below shows the localized categories.

Timestamp	Category	Action	Old status	New status	Property t...	Information
07/17/2013 03:32:06 PM ...	Customer-2					log by category
07/17/2013 03:31:49 PM ...	Customer-1					log by category
07/17/2013 03:31:19 PM ...	Manual					manual log
07/17/2013 03:31:11 PM ...	FSM	[Forced]	Clean	To be cleaned	CW-ENTITY-...	
07/17/2013 02:10:14 PM ...	Customer-1					entered via category CUST-1
07/17/2013 02:10:12 PM ...	Manual					manual log entry
07/17/2013 02:10:08 PM ...	FSM	[Forced]	To be cleaned	Clean	CW-ENTITY-...	

Figure 27: Additional categories in equipment logbook of Data Manager

Writing Equipment Logbook Entries Using New Categories

To write equipment logbook entries using new categories in the context of a phase, use the *IMESS88EquipmentLogbook* method of the *IS88EquipmentLogbookService* class:

```
public IMESS88EquipmentLogbook writeLogbookEntryOfExtendedCategory(IMESS88Equipment
equipment, IMESRtPhase rtPhase, String categoryMeaning, String information, IMESSSignature
signature);
```

The third parameter (String categoryMeaning) contains the meaning of the choice list elements (e.g.CUST-1).

TIP

If the meaning of an already existing PharmaSuite category is passed (e.g. *Binding*), this method throws an *IllegalArgumentException* because the access is reserved for internal use. The service provides specific methods for the internal categories.

Example to write out of a phase

```
try {
    IS88EquipmentLogbookService service =
        ServiceFactory.getService(IS88EquipmentLogbookService.class);
    String categoryMeaning = "CUST-1";
    String information = "My information";
    service.writeLogbookEntryOfExtendedCategory(equipment, rtPhaseId,
        categoryMeaning,information);
} catch (RuntimeException x) {
    showErrorDialog(x.getLocalizedMessage());
}
```

Accessing Equipment Logbook Entries

To access equipment logbook entries, use the standard PharmaSuite filter mechanism. The *IMESS88EquipmentLogbookFilter* method of the *IS88EquipmentLogbookService* class provides means to create any filter on the *IMESS88EquipmentLogbook* objects.

```
public IMESS88EquipmentLogbookFilter createLogbookFilter();
```


Retrieving Specific Information from the Equipment Logbook

This section describes how to retrieve information about the last product being used on a given equipment entity and how this information can be made available for recipe authors, in terms of functions being available in the Expression editor of Recipe and Workflow Designer.

Retrieving the last product identifier (**Last Product ID** function) is available by default. But there are similar queries like the last product batch that can be derived from the available implementation.

What Is a Last Product?

The subsequent figure shows a simplified equipment logbook after processing a few orders and workflows. The oldest entries are at the bottom and the newest ones at the top. This logbook is used to explain what the term **last product** means at any given point in time.

Between	08/01/2013 12:00:00 AM CEST					
And	08/01/2013 11:59:59 PM CEST					
Timestamp	Category	Action	Workflow	Order	Product batch	Product ma...
08/01/2013 01:19:09 PM ...	Binding	Release		418-03	BX111	000418FG01
08/01/2013 01:18:22 PM ...	Denial			419-02	BX112	000419FG01
08/01/2013 01:17:09 PM ...	Binding	Identify		418-03	BX111	000418FG01
08/01/2013 01:16:15 PM ...	Binding	Release	WF13000013			
08/01/2013 01:16:13 PM ...	Binding	Bind	WF13000013			
08/01/2013 01:16:13 PM ...	Binding	Identify	WF13000013			
08/01/2013 01:15:57 PM ...	Binding	Release		419-01	BX110	000419FG01.1
08/01/2013 01:15:56 PM ...	Binding	Bind		419-01	BX110	000419FG01
08/01/2013 01:15:55 PM ...	Binding	Identify		419-01	BX110	000419FG01
08/01/2013 01:15:31 PM ...	Binding	Release		418-02	BX109	000418FG01.1
08/01/2013 01:15:31 PM ...	Binding	Bind		418-02	BX109	000418FG01
08/01/2013 01:15:30 PM ...	Binding	Identify		418-02	BX109	000418FG01
08/01/2013 01:14:41 PM ...	Binding	Release		418-01	BX108	000418FG01.1
08/01/2013 01:14:40 PM ...	Binding	Bind		418-01	BX108	000418FG01
08/01/2013 01:14:22 PM ...	Binding	Identify		418-01	BX108	000418FG01

Figure 28: Example of equipment logbook (binding)

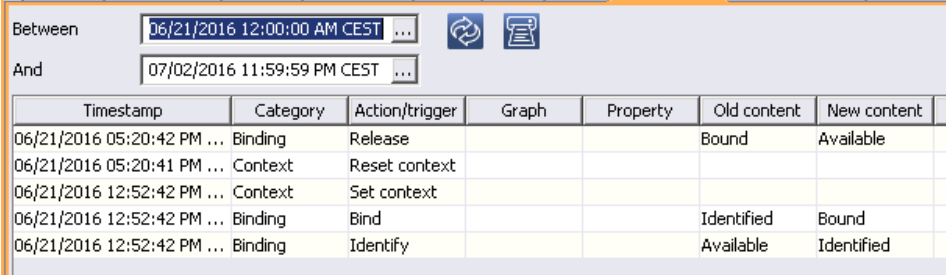
The association between an equipment entity and its exposure to processing is done through the binding process. A typical use case is like this:

- An operator identifies an equipment entity by scanning its barcode (**Identify** action in the logbook). This can fail, for instance if the equipment entity is

attached to a different processing context. The **Denial** entry for the **419-02** order was created, because at that time the equipment entity was already identified for the **418-03** order.

- Certain checks are performed to ensure that the identified equipment entity is appropriate for use in the given processing context. If all checks are passed successfully, the equipment entity is bound to the processing context. An equipment entity bound to an order context is part of the production of the product material of that order.
- In addition to equipment binding, the system also maintains specific execution context information for each equipment entity. The main differences from the binding context are:
 1. Monitoring of actual and previous usage in terms of related objects (e.g. Involved material, Order step) and
 2. possibility to make GxP-relevant decisions during execution (e.g. calculate the cleaning demand for room equipment entities).

Any modification of the execution context (set and reset) is intended to be done after the equipment entity has been bound and before it is released.



Timestamp	Category	Action/trigger	Graph	Property	Old content	New content
06/21/2016 05:20:42 PM ...	Binding	Release			Bound	Available
06/21/2016 05:20:41 PM ...	Context	Reset context				
06/21/2016 12:52:42 PM ...	Context	Set context				
06/21/2016 12:52:42 PM ...	Binding	Bind			Identified	Bound
06/21/2016 12:52:42 PM ...	Binding	Identify			Available	Identified

Figure 29: Example of equipment logbook (context)

- At some point in the future, the equipment entity is **released** from this processing context. From that time on the equipment entity is no longer involved in the processing of the order and the formerly bound product becomes the **last product**.

The example logbook also shows the following events:

- At the very top, an equipment entity is **released** after an **Identify** action, there is no **Bind** action in between. This was caused by performing the **Cancel order** function before the **Bind** action had taken place.
- There are three entries related to binding for a workflow. As this workflow did not record a product material, the **Identify » Bind » Release** sequence has no impact on the **last product**.

A last product is defined by the following algorithm:

1. Look for the most recent **Release** action with a non-empty product material.

2. Starting from that logbook entry, look for the most recent **Bind** action with a non-empty product material.
3. Take the product material from that **Bind** action as result.

Retrieving the Last Product Information

In order to provide a **Last Product** function within the Expression editor of Recipe and Workflow Designer, the following tasks must be performed:

- Implement an access method that returns the desired logbook entry (page 109).
- Implement a function that makes the **Last Product** function available for the Expression editor of Recipe and Workflow Designer (page 111).
- Implement a *type inference* class that guides the Expression editor to perform semantic checks while editing the expression (page 111).

Retrieving the Last Product Logbook Entry

IS88EquipmentLogbookService provides methods to access the S88 equipment logbook. However, if you wish to add a project-specific function, you can implement it within a project-specific service or helper class instead of extending *IS88EquipmentLogbookService*.

The signature of the method looks like this:

```
public IMESS88EquipmentLogbook getEntryForLastBindTransitionWithProduct(IMESS88Equipment
equipment);
```

The method returns the logbook entry rather than the identifier of the last product.

To implement the method for the last product information, you can use one of the following strategies:

- SQL statement to query the database. The last product method makes use of this approach.
This is an efficient and straightforward strategy.
- FastLaneReader or FastBeanReader.
FactoryTalk ProductionCentre filter to query the logbook table for the desired data.
Both strategies would require a high effort to emulate the join and where clauses.

The following SQL statement retrieves the key of the logbook entry holding the last product information for a given equipment entity. The statement has two sections:

- Retrieve the Cartesian product of two logbook entries. The Cartesian product is a mathematical operation on sets (here: logbook entries) that returns the set of all ordered pairs of the members of two given sets.

The two given sets are release transition (REL) and the bind transition (BIND). It is important that BIND is created before REL and that both REL and BIND have a non-empty product part.

- Retrieve (REL, BIND) with the latest BIND creation timestamp. This is implemented with a sub-select statement and the same query as used for the first step.

```
SELECT DISTINCT BIND.ctr key FROM AT_X_S88EquipmentLogbook REL,
AT_X_S88EquipmentLogbook BIND
WHERE REL.X_s88equipment_64=975343 AND REL.X_category_I=10 AND
REL.X_transition_S='release' AND REL.X_productPart_21 IS NOT NULL
AND BIND.X_s88equipment_64=975343 AND BIND.X_category_I= 10 AND
BIND.X_transition_S='bind' AND BIND.X_productPart_21 IS NOT NULL
AND BIND.creation_time <= REL.creation_time
AND BIND.creation_time=
(SELECT MAX(BIND1.creation_time) FROM AT_X_S88EquipmentLogbook REL1,
AT_X_S88EquipmentLogbook BIND1
WHERE REL1.X_s88equipment_64=975343 AND REL1.X_category_I= 10 AND
REL1.X_transition_S='release' AND REL1.X_productPart_21 IS NOT NULL
AND BIND1.X_s88equipment_64=975343 AND BIND1.X_category_I= 10 AND
BIND1.X_transition_S='bind' AND BIND1.X_productPart_21 IS NOT NULL
AND BIND1.creation_time <= REL1.creation_time)
```

TIP

The product parts of REL and BIND need not be identical.

We recommend to use the column names from *IS88EquipmentLogbook* to create the SQL statement as string. Each column name is a constant starting with *IS88EquipmentLogbook.SQL_COL_NAME*.

We assume that the string is available in the *sqlString* variable. The *getLogbookEntryFromSQLResult()* method extracts the logbook key from the SQL result and instantiates the *IMESS88EquipmentLogbook* entry. Make sure to use *MESATObjectManager.getInstance()*. This ensures that only a single wrapper instance associated the underlying AT row is hold by the system.

```
try {
    final List<String[]> readerResult =
        PCCContext.getFunctions().getArrayDataFromActive(sqlString);
    return getLogbookEntryFromSQLResult(readerResult);
} catch (RuntimeException e) {
    throw new MESRuntimeException(e);
}

private IMESS88EquipmentLogbook getLogbookEntryFromSQLResult(
    final List<String[]> readerResult) {
    if (readerResult != null && !readerResult.isEmpty()) {
        final String keyString = readerResult.get(0)[0];
        final IMESS88EquipmentLogbook entry =
            MESATObjectManager.getInstance(IMESS88EquipmentLogbook.class,
                Long.parseLong(keyString));
        return entry;
    }
}
```

```
return null;
}
```

Implementing the Expression Editor Function

For the general concept of how to implement a new Expression editor function, see chapter "Configuring the Expression Editor of Recipe and Workflow Designer and Data Manager" (page 63).

To implement the **Last Product ID** function for the Expression editor, extend the *ExpressionEvalFunction* class and add a method as follows:

```
@ExpressionFunction(typeInferencerClass = FunctionTypeInferencerLastProduct.class,
    minArgCount = 1, maxArgCount = 1, groupId = "055Equipment", sortIndex = 30)
public String lastProduct(IMESS88Equipment equipment) {
    if (equipment == null) {
        return null;
    }
    final IMESS88EquipmentLogbook logbookEntry =
        ServiceFactory.getService(IS88EquipmentLogbookService.class)
            .getEntryForLastBindTransitionWithProduct(equipment);
    final String result =
        logbookEntry != null ? logbookEntry.getProductPartIdentifier() : null;
    return result;
}
```

The annotation of this method defines that the function:

- has precisely one argument,
 - appears within the equipment-related function, and
 - uses the *FunctionTypeInferencerLastProduct* type inference class.
- You must implement this class as well.

The function fetches the logbook entry using the access method defined in the last section, extracts the product part identifier, and returns it. If no logbook entry was found or no equipment entity was given, the function returns null. Null is the default value for unknown expression results.

Implementing the Type Inference Class

The *FunctionTypeInferencerLastProduct* type inference class completes the implementation. For more information, see section "Implementing a Type Inference Class" (page 69) in chapter "Configuring the Expression Editor of Recipe and Workflow Designer and Data Manager" (page 63).

FunctionTypeInferencerLastProduct has a single argument (note the starting index of 1), which must be an equipment entity. The result is a String, which can be null.

```
public class FunctionTypeInferencerLastProduct implements
    IExpressionFunctionTypeInferencer {
```

```
/** Argument index for all arguments. */
private static final int ARG_INDEX_EQUIPMENT = 1;

@Override
public IExpressionTypeDescriptor inferResultTypeForFunctionCall(
    IExpressionFunctionTypeInferenceSupport support) {
    support.checkArgumentHasType(ARG_INDEX_EQUIPMENT, IMESS88Equipment.class);
    final boolean resultIsNullable = true;
    return support.createTypeDescriptor(String.class, resultIsNullable);
}
}
```

Configuring Additional Purposes for the Equipment Type Technical Property Type

This section describes how to add additional purposes to the **Equipment type** technical property type and how to extend the validation check to be executed when a property type is saved. The validation check verifies the consistency of purpose, usage type, and data type of a property type.

To add additional purposes, you have to be familiar with

- Extending a choice list (see section "Choice Lists", chapter "Adapting and Adding Field Attributes" in Volume 2 of the "Technical Manual Configuration and Extension" [A3] (page 171)).
- Registering custom classes (see section "Configuring Service Implementations", chapter "Managing Services" in Volume 2 of the "Technical Manual Configuration and Extension" [A3] (page 171)).
- Creation of custom JAR files (see section "Libraries" in "Process Designer Online Help" [B1] (page 171)).

Add a Purpose and Extend the Validation Check

To add a purpose and extend the validation check, proceed as follows:

1. Add a choice element for the additional purpose to the **S88EquipmentPropertyTypePurpose** choice list and provide localized message pack entries.
Example: Meaning=CurrentWeight, Value=77
2. The *EquipmentPropertyTypePurposeValidator* class contains the standard purpose-specific validation check.
Create a subclass of the *EquipmentPropertyTypePurposeValidator* class, extend the constructor of your subclass by calling the *addValidPurposeConfiguration(meaningOfPurpose, technicalPropertyType, usage)* method. The method is documented in the *IEquipmentPropertyTypePurposeValidator* interface.

Example code

```
package com.rockwell.custmes.services
import ...
```

```
public class CustEquipmentTypePurposeValidator extends
    EquipmentPropertyTypePurposeValidator {

    public CustEquipmentPurposeValidator() {
        // add your additional valid (purpose, technical property type, usage) combination
        // all parameters are using the meaning of the choicelist element
        addValidPurposeConfiguration("CurrentWeight",
            EnumEqmPropertyTechnicalType.MEASURED_VALUE,
            EnumEqmPropertyTypeUsage.RUNTIME);
    }
}
```

3. Register the subclass in an XML configuration file similar to a service.
Replace the path formatted in red with the complete path to your class.

Example code

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<configuration>
  <EquipmentPropertyTypePurposeValidator>
    <implementationClass>
      com.rockwell.custmes.services.CustEquipmentPropertyTypePurposeValidator
    </implementationClass>
  </EquipmentPropertyTypePurposeValidator>
</configuration>
```

4. Create a custom JAR file containing your class and your XML file.
5. In Process Designer, create a library and import your JAR file.
6. Start Data Manager, create a new property type and verify that
 - for the **Purpose** attribute, the new choice element is available and
 - the property type with a valid combination and the new choice element can be saved successfully.
 - We recommend to also check that for invalid combinations, the system displays an error message and the property type cannot be saved.

Adding a Workflow Type to Workflow Designer and the Production Execution Client

This section describes how to add a workflow type to Workflow Designer and make this type available as a group in the Production Execution Client.

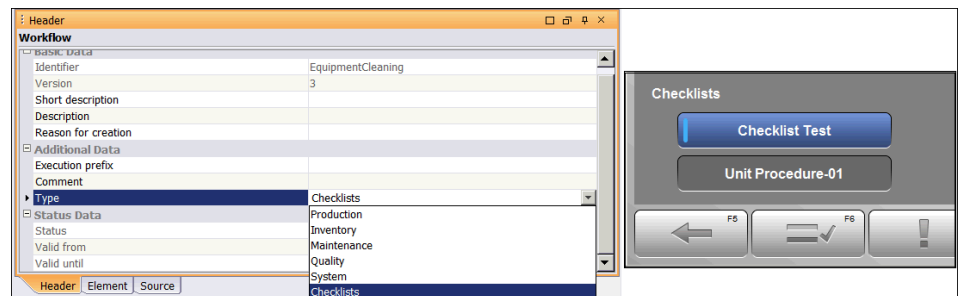


Figure 30: New workflow type in Workflow Designer and the Production Execution Client

To add a workflow type, you have to be familiar with

- Choice List Manager (see chapter "Adapting and Adding Field Attributes" in Volume 2 of the "Technical Manual Configuration and Extension" [A3] (page 171)).
- Process Designer
- Workflow Designer

Creating a Workflow Type

To add a workflow type to Workflow Designer, proceed as follows:

1. In Choice List Manager, add the new workflow type (e.g. Checklists) to your version of the **WorkflowType** choice list.

2. In Process Designer, adapt your version of the **MESChoiceListStrings** message pack to support your required locales.

The screenshot shows the 'Choice List Manager' interface with three main panels:

- Selected choice list:**
 - Message pack: choiceListStrings
 - Description:
 - GUID: WorkflowType
 - Name: WorkflowType
 - Custom:
 - Editable: ☒ Yes
 - Sort alphabetical: ☐ No
- Choice elements of selected choice list:**

Meaning	Value	Msg. ID
Production	10	WorkflowType-Produ
Inventory	20	WorkflowType-Invent
Maintenance	30	WorkflowType-Mainte
Quality	40	WorkflowType-Qualit
System	50	WorkflowType-Syster
Checklists	60	WorkflowType-Check
- Selected choice element:**
 - Message-ID: lowType-Checklists
 - Meaning: Checklists
 - Internal value: 60
 - GUID: 6F-FF72AFB724B4}
 - Custom:
 - Editable: ☐ false
 - Localized item: Checklists
 - Visible: ☒ true
 - Selectable: ☒ true

Figure 31: Adding a choice element in Choice List Manager

Adding a Group to the Production Execution Client

To add a group for the new workflow type to the Production Execution Client, proceed as follows:

1. In Process Designer, create a new **List** object for the new group (e.g. StartableItemNamesChecklists).
2. Add the new group to your version of the **StartableItemGroups** list object.
3. Adapt your version of the **StartableItemGroups** message pack to support your required locales.

Managing Cockpit Actions of the Production Execution Client

This section contains general information about Cockpit actions (page 117), implementing action classes (page 118), the configuration of action classes (page 123), the configuration of actions (page 124), actions with restricted access rights (page 125), filtering of actions (page 126), enable scrolling in forms (page 127), and details about optional Cockpit actions (page 128), invisible Cockpit actions (page 129), scanner actions in the Cockpit (page 130), and configuring the running process of the Cockpit (page 133).

To modify the access privilege settings of an existing action, perform the following steps:

1. Modify the access privilege settings (page 126) of the action.
2. Run the application to verify the changes in the Cockpit and its actions.

To replace an existing action implementation, perform the following steps:

1. Implement a new action class (page 118) containing the adapted behavior of an existing action.
2. Modify the configuration (page 123) in order to use the new action class implementation.
3. Run the application to verify the changes in the Cockpit and its actions.

To add a new action, perform the following steps:

1. Implement a new action class (page 118).
2. Add the configuration for the new action class (page 123).
3. Configure the action (page 124) to be displayed as startable by the Cockpit.
4. Define an access privilege (page 126) for the action.
5. Optional: If the new action displays a form, enable the scrolling of the form's content (page 127).
6. Run the application to verify the changes in the Cockpit and its actions.

What Are Cockpit Actions?

In PharmaSuite, the recipe and workflow execution is controlled via the Cockpit. The Cockpit represents an operator's entry point to the execution process. Specific to each station and user, the Cockpit provides a list of all startable, resumable, and running

processes, from which the operator can select unit procedures and operations to start or switch to. The running processes also include those operations of the same unit procedure that are running at another station. This also applies to actions.

Thus, the Cockpit is able to manage workflows and actions. As opposed to a pre-defined workflow with several steps and alternative flows, an action represents a single step without alternatives (exit, label reprint, etc.). An action may have a user interface, but it is not required. We recommend to implement less complex functions as actions (e.g. preview of a report) and to implement complex functions as workflows (e.g. goods receipt).

An example for an action is the **Exit** action. The **Exit** action triggers the display of a user dialog and logs out the user after confirmation. An action may also display a form to perform the functionality.

The functionality and business logic of an action is encapsulated in a component, here a Java class.

Depending on your requirements, you either may implement a new action class containing the adapted behavior of an existing action or implement a new action class.

Implementing an Action Class

An action class has to extend the *AbstractAction* class. The implementation of the functionality of the action is located in the *actionPerformed* method. Actions vary with regard to their user interface and the business logic behind the action:

- actions without a user interface (page 118),
- actions displaying a dialog (page 119),
- actions displaying a form (page 119), and
- actions displaying a panel (page 120).

Actions without a User Interface

An action without a user interface is an action for which no further user interaction is required after the execution of the action has been triggered through the action button in the Cockpit. In this case, the *actionPerformed* method must contain the business logic of the action.

Example: The *MYAction* action class simply executes some kind of business logic with MYC for the My Company vendor code.

```
javax.swing.AbstractAction;  
  
public class MYAction extends AbstractAction {  
    public MYAction() {  
        super("i18nNameOfMYAction");  
    }  
    public void actionPerformed(ActionEvent e) {
```

```
// Implementation of the business logic of the Action
}
}
```

Actions Displaying a Dialog

An action can display a dialog after the action has been triggered through the action button in the Cockpit. For example, the **Exit** action displays a question dialog. In this case, the *actionPerformed* method must create the dialog and execute the business logic dependent on the operator's choice.

Example: The *MYAction* action class displays a question dialog and executes some kind of business logic if the operator has selected the **Yes** option with MYC for the My Company vendor code.

```
javax.swing.AbstractAction;

public class MYAction extends AbstractAction {
    public MYAction() {
        super("i18nNameOfMYAction");
    }
    public void actionPerformed(ActionEvent e) {
        IExceptionHandler exService = ServiceFactory.getService(IExceptionHandler.class);
        int answer = exService.handleQuestion("My Action Question", "My Caption", null,
            JOptionPane.YES_NO_OPTION, null);
        if (JOptionPane.YES_OPTION == answer) {
            // call business logic
        }
    }
}
```

Actions Displaying a Form

An action can display a form after the action has been triggered through the action button in the Cockpit. For example, the **Material Processing** action displays a grid containing the order steps available for processing. The execution of the business logic is triggered when the operator selects materials (i.e. rows in the grid) and then taps the **Process** button. For this purpose, the Production Execution Client provides an embedded form within the Cockpit UI frame in which the action can display its form. In this case, the *actionPerformed* method triggers the display of the form in the dedicated embedded form. The business logic itself is implemented within the form.

To display a form in the Cockpit, call the *displayCockpitEmbeddedForm* method on the Production Execution Client (see example below). The first argument expects the name of an existing form. The second argument is a string, which will be displayed in the status bar when displaying the form. In this case, the *actionPerformed* method triggers the display of the form in the Production Execution Client. The form has to provide the business logic of the action.

The framework does not close the form automatically, since it depends on the implementation of the action when the form must be closed. The following options are available for leaving an action:

- operator-triggered
The framework closes the form when the operator has clicked the **Back** button.
- form-triggered
The code of the form can explicitly close the form of the action by calling the *closeCockpitEmbeddedForm* method on the Production Execution Client (see example below). For example, the operator has triggered the business logic behind the action and the action shall be left.

Example: The *MYAction* action class displays the *mycActionFormName* form with MYC for the My Company vendor code.

MYAction

```
import javax.swing.AbstractAction;
public class MYAction extends AbstractAction {
    public MYAction() {
        super("i18nNameOfMYAction");
    }
    public void actionPerformed(ActionEvent e) {
        IProductionExecutionClient pec =
            ServiceFactory.getService(IProductionExecutionClient.class);
        pec.displayCockpitEmbeddedForm("mycActionFormName", "MYCStatusBarTitle");
    }
}
```

To finish an action, the *mycActionFormName* form must call the following Pnuts code to close the form in the Cockpit:

Example code: finish an action

```
import ("com.rockwell.mes.clientfw.pec.ifc.controller.IProductionExecutionClient")
import ("com.rockwell.mes.commons.base.ifc.services.ServiceFactory")

serviceFactory = ServiceFactory::getInstance();
pec = serviceFactory.getServicePNuts(IProductionExecutionClient,
                                    "ProductionExecutionClient")
pec.closeCockpitEmbeddedForm()
```

Actions Displaying a Panel

According to "Actions Displaying a Form" (page 119), the Production Execution Client provides an embedded form within the Cockpit UI frame in which an action can display its form. Additionally, the embedded form can display a Swing panel (*JPanel* class). For example, the **Batch Processing** action displays a Swing panel with a grid containing the order steps available for processing and a Quick Filter component. The execution of the business logic is triggered within the panel. In this case, the *actionPerformed* method

triggers the display of the panel in the dedicated embedded form. The business logic itself is implemented within the panel.

To display a panel in the Cockpit, call the *displayCockpitEmbeddedForm* method on the Production Execution Client (see example below). The argument expects a class implementing the *ICockpitActionJPanel* interface. The panel (class) implementing *ICockpitActionJPanel* has to provide the business logic of the action.

The framework does not close the panel automatically, since it depends on the implementation of the action when the panel must be closed. The following options are available for leaving an action:

- operator-triggered
The framework closes the form when the operator has clicked the **Back** button.
- action-triggered
The code of the panel can explicitly close the form of the action by calling the *closeCockpitEmbeddedForm* method on the Production Execution Client (see example below). For example, the operator has triggered the business logic behind the action and the action shall be left.

Example: The *MYCCockpitActionJPanel* action class displays a panel created within the *MYCCockpitActionJPanel* panel with MYC with MYC for the My Company vendor code.

MYCAction

```
import javax.swing.AbstractAction;
public class MYCAction extends AbstractAction {
    public MYCAction() {
        super("i18nNameOfMYCAction");
    }
    public void actionPerformed(ActionEvent e) {
        IProductionExecutionClient pec =
            ServiceFactory.getService(IProductionExecutionClient.class);
        pec.displayCockpitEmbeddedForm(new MYCCockpitActionJPanel());
    }
}
```

The *MYCCockpitActionJPanel* class must implement the *ICockpitActionJPanel* interface:

MYCCockpitActionJPanel

```
package com.rockwell.mes.apps.ebr.ifc.cockpitaction;
public interface ICockpitActionJPanel {
    /**
     * build the ui of the panel.
     */
    public void buildUI();
    /**
     * @return the jPanel to be shown
     */
    public JPanel getJPanel();
}
```

```

    * @return null or a scroll pane if a scroll pane within the jPanel shall react on
    *       pageup/down cockpit buttons
    */
    public JScrollPane getScrollPane();
    /**
    *   return part of the name for the help URL. The complete URL is prepended with
    *   + "cockpitForm_"
    */
    public String getHelpURL();
    /**
    *   @return title to be displayed in the status bar
    */
    public String getStatusBarTitle();
}

```

The *CockpitActionJPanel* abstract base class provides events for opening and closing the panel of the action. To enable barcode scanner support for the action, just return an *IBarcodeScannedListener*.

CockpitActionJPanel

```

public class CockpitActionJPanel extends JPanel implements ICockpitActionJPanel {
    /**
    *   build the ui of the panel.
    */
    public void buildUI() {
        buildPanelContent();
        // ...
    }
    /**
    *   @return the jPanel to be shown
    */
    public JPanel getJPanel() {
        Return this;
    }
    /**
    *   Is called when action jPanel get opened initially. If subclass provides a barcode
    *   listener it is registered.
    *   Could be overloaded to extend.
    */
    protected void onOpenActionJPanel() {
        // ...
    }
    /**
    *   Is called if action jPanel get closed. Registered Barcode listeners are deregistered.
    *   Could be overloaded to extend.
    */
    protected void onCloseActionJPanel() {
        // ...
    }
    /**
    *   @return an IBarcodeScannedListener if scanner support shall be installed.
    */
    protected abstract IBarcodeScannedListener getBarcodeScannedListener();
}

```


To finish an action, the *MYCCockpitActionJPanel* panel must call *closeCockpitEmbeddedForm()*:

Example code: finish an action

```
import com.rockwell.mes.clientfw.pec.ifc.controller.IProductionExecutionClient;
import com.rockwell.mes.commons.base.ifc.services.ServiceFactory;
import com.rockwell.mes.apps.ebr.ifc.cockpitaction.CockpitActionJPanel;

public class MYCCockpitActionJPanel extends CockpitActionJPanel {
    protected void buildPanelContent () {
        // build the content of this JPanel
    }
    private void onBusinessSpecificUIEvent() {
        // do business functions
        // close Action:
        IProductionExecutionClient pec =
            ServiceFactory.getService(IProductionExecutionClient.class);
        pec.closeCockpitEmbeddedForm();
    }
    public String getStatusBarTitle() {
        return "My status bar title";
    }
}
public String getHelpURL() {
    return "MYHelpURL";
}
}
public JScrollPane getScrollPane () {
    return myScrollPane;
}
}
public IBarcodeScannedListener getBarcodeScannedListener() {
    return new MYCBarcodeScannedListener();
}
}
...
}
```

Configuring Action Classes

Before you can use an implementation class of an action, the class must be configured. The configuration of implementation classes of Cockpit actions is held in *clientfw-pec.xml*.

To add a new action class or to replace an existing action class, a configuration entry is required. The XML file below contains such an entry in the *<configuration>* XML element. The action name (here: *MYCAction* with MYC for the My Company vendor code) is used as the XML element name: *<MYCAction>...</MYCAction>*. In this element, the implementation class (here: *com.rockwell.mes.cust.impl.MYCAction*) is given as the content of the *<implementationClass>* XML element.

To replace an existing action class, change the value of the implementation class to the implementation class of your action. In order to add a new action class, add a new XML element that refers to the new action.

```
<configuration>
  <MYCAction>
```

```
<implementationClass>com.rockwell.mes.cust.impl.SpecialAction
</implementationClass>
</MYCAction>
</configuration>
```

TIP

The name of an action (here: *MYCAction*) has to end with the string *Action*. This is important for the configuration of the Cockpit later on (page 124).

Configuring Actions

In order to display an action in the Cockpit of the Production Execution Client, the action has to be added to the **List** object of startable items of the corresponding group. For details, see section "Displaying an Activity Set as Startable" in chapter "Adapting the Workflows of the Production Execution Client" in Volume 1 of the "Technical Manual Configuration and Extension" [A2] (page 171).

Example: Adding the *MYCAction* action to *StartableItemsNamesMaintenance* with MYC for the My Company vendor code.

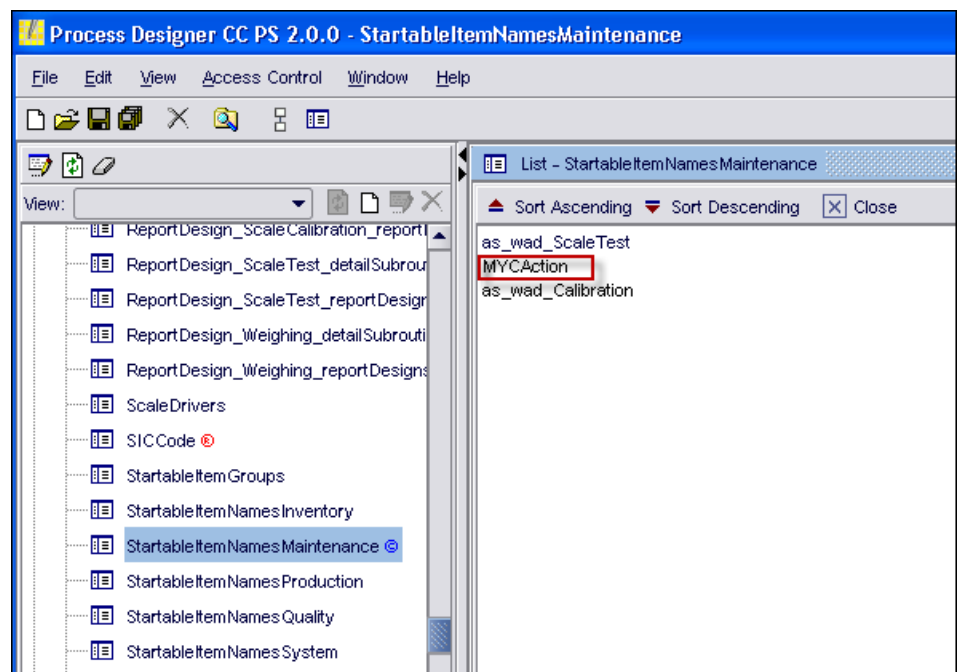


Figure 32: Configuring the MYCAction action for the Cockpit

After restarting the Production Execution Client, the **MYCAction** action is listed in the configured group of the Cockpit.

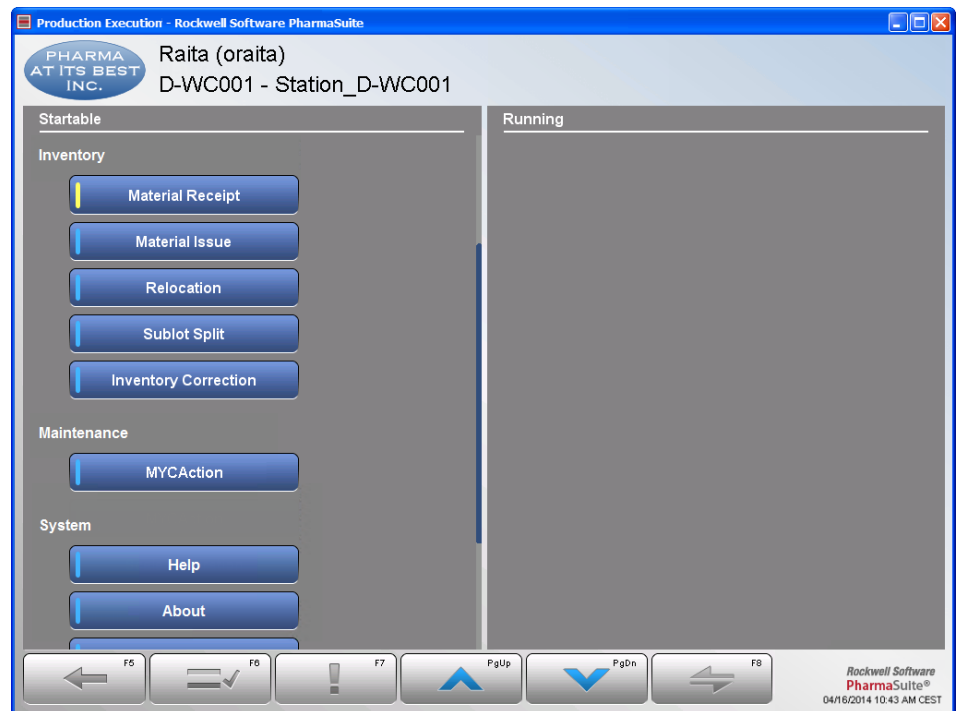


Figure 33: MYCAction action in the Cockpit

TIP

Do not forget to assign an access privilege to your new action (page 126).

Actions with Restricted Access Rights

PharmaSuite allows to have Cockpit actions with restricted access rights to handle the following specific situations:

An operator can log in to Production Execution Client and

- no station is assigned to the current device or
- the operator does not have the access privilege assigned to the current station.

In both situations the operator is not allowed to process any workflows. The default configuration only provides the actions of the **System** group to operators with restricted access rights.

To apply the restricted mode to an action, add the `@NoStationOrStationPrivilegeRequired` Java annotation. This code snippet shows an implementation of the **About** action in the Cockpit.

```
@NoStationOrStationPrivilegeRequired
public class CockpitHelpAction extends AbstractPECHelpAction {

    private static final long serialVersionUID = 1L;

    /**
```

```
* constructor.
*/
public CockpitHelpAction() {
    super (I18NHelper.getMessage (MessageIDConstants.BUTTON_TASK_COCKPIT_HELP),
        ServiceFactory.getService (IProductionExecutionClient.class));
}
}
```

Filtering Actions by Means of Access Privileges

To filter actions, the Cockpit makes use of access privileges. To filter specific actions for a specific user or user group, configure the access privilege of the action. The names of the access privileges for actions are defined according to the following syntax:

PEC_AccessRight_ + <name of the action>.

Action	Access privilege
AboutAction (About)	PEC_AccessRight_AboutAction
ChangeStationAction (Register at Station)	PEC_AccessRight_ChangeStationAction
ChangeStationScannerAction (Register at Station by Scan)	PEC_AccessRight_ChangeStationScannerAction
ChangeUserAction (Change User)	PEC_AccessRight_ChangeUserAction
CockpitHelpAction (Help)	PEC_AccessRight_CockpitHelpAction
DeviceProcessingAction (Device Processing)	PEC_AccessRight_DeviceProcessingAction
LogoutAction (Exit)	PEC_AccessRight_LogoutAction
StartCampaignWeighingAction (Material Processing)	PEC_AccessRight_StartCampaignWeighingAction
StartProcessingAction (Start Batch Processing)	PEC_AccessRight_StartProcessingAction
StartBatchProcessingScannerAction (Start Batch Processing by Scan)	PEC_AccessRight_StartBatchProcessingScannerAction
StartDeviceProcessingScanner Action (Start Device Processing by Scan)	PEC_AccessRight_StartDeviceProcessingScannerAction
WorkflowOrderProcessingAction (Workflow Processing)	PEC_AccessRight_WorkflowOrderProcessingAction

To assign an access privilege to a new action, proceed as follows:

- Create a new access privilege and assign a name to it according to the following syntax:

PEC_AccessRight_ + <name of your action>



Figure 34: Access privilege for new action

For more information on access privileges, please refer to chapter "Managing Electronic Signatures and Access Rights" in Volume 2 of the "Technical Manual Configuration and Extension" [A3] (page 171).

Adapting the ChangeStationAction (Register at Station) Action

The **ChangeStationAction** (Register at Station) action allows an operator to register a device at another station. The system provides all the stations to which the logged-in operator has access. The access to a station is defined with the **X_accessPrivilege** UDA of the station.

To restrict the list of available stations, the action provides an extension hook. Proceed as follows:

1. Create a new class that extends the *ChangeStationAction* class, e.g. *MyChangeStationAction*.
2. Override the *applyFilter* method by returning a list of allowed stations.
3. Configure your action class according to section "Configuring Action Classes" (page 123).
4. Replace the configuration of the *ChangeStationAction* action with your *MyChangeStationAction* action according to section "Configuring Actions (page 124)".

Example code to restrict the list of available stations

```
/**
 * Allows customized filtering the list of allowedStationNames by overloading
 *
 * @param allowedStationNames list of allowed station names for the current user
 * @return the list of allowed station names, which can be restricted by this method
 */
protected List<String> applyFilter(List<String> allowedStationNames) {
    return allowedStationNames;
}
```

Enabling of Scrolling in Forms

Actions can display forms. Depending on the form, it is useful to enable paging through the content of the form with the **Page Up** and **Page Down** buttons.

For this purpose, register your *scroll bar* object at the *scroll support* object. Then, tapping the **Page Up** or **Page Down** button triggers the registered *scroll bar* object to perform the scrolling.

The *scroll support* object is available as the *IScrollSupport:FORM_PROPERTY_SCROLL_SUPPORT* form property:

```
public interface IScrollSupport {

    /** The form property to provide the scroll support object */
    public static final String FORM_PROPERTY_SCROLL_SUPPORT = "ScrollSupport";

    /**
     * Initialize the scrolling, meaning that the vertical scroll bar will be registered
     * for scrolling by page up/down buttons and supports the scrolling with the mouse
     * @param scrollPane the scroll pane to be used for scrolling and
     *                  containing the vertical scroll bar
     */
    public void initializeScrolling(JScrollPane scrollPane);

    /**
     * Deinitialize the scrolling of the vertical scroll bar by page up/down
     * buttons and the scrolling with the mouse
     * @param scrollPane the scroll pane to be used for scrolling
     *                  and containing the vertical scroll bar
     */
    public void deinitializeScrolling(JScrollPane scrollPane);
}
```

Example: The Pnuts code of a form that will be displayed by a Cockpit action. Here, *activityControl1* is a *FastLaneReaderGridActivity* that offers a method to get the vertical scroll bar.

```
import ("com.rockwell.mes.clientfw.pec.ifc.view.IScrollSupport")

function initForm() {
    initializeScrolling()
}

function initializeScrolling() {
    scrollSupport = getFormProperty(IScrollSupport::FORM_PROPERTY_SCROLL_SUPPORT)
    if (scrollSupport != null)
        scrollSupport.initializeScrolling(activityControl1.getScrollPane())
}

function deinitializeScrolling() {
    scrollSupport = getFormProperty(IScrollSupport::FORM_PROPERTY_SCROLL_SUPPORT)
    if (scrollSupport != null)
        scrollSupport.deinitializeScrolling(activityControl1.getScrollPane())
}

function closeForm() {
    deinitializeScrolling()
}
```

Optional Cockpit Actions

Apart from the default Cockpit actions which are enabled by default, there are optional Cockpit actions which are not enabled by default. To enable an optional Cockpit action, add the action to the list of startable items of the corresponding group (see "Configuring Actions" (page 124)).

TIP

Please note that the action listed below already has an assigned access privilege.

- RefreshStartableOperationsAction

Sends a request to retrieve all startable operations available for the current station to refresh the Cockpit. We recommend to enable the action if the network connectivity is bad which can result in a loss of broadcast messages.

Example: After the completion of an operation the Cockpit does not display the subsequent operation of the unit procedure. In other words, the Production Execution Client did not receive the *Operation Startable* broadcast.

Invisible Cockpit Actions

The Cockpit can handle invisible actions. Even when they are active, they are not visible in the Cockpit. Typically, invisible actions are used in the context of scanner actions in the Cockpit (page 130).

In order to make an invisible action available in the Cockpit, the action (class) has to be added to the **List** object of startable invisible items (**StartableItemNamesInvisible**). The list must be a member of the **StartableItemGroups** list.

The string **Invisible** in the group name indicates that this group contains invisible items. Such a group will not be displayed in the Cockpit, but the actions defined for the group are active, nevertheless.

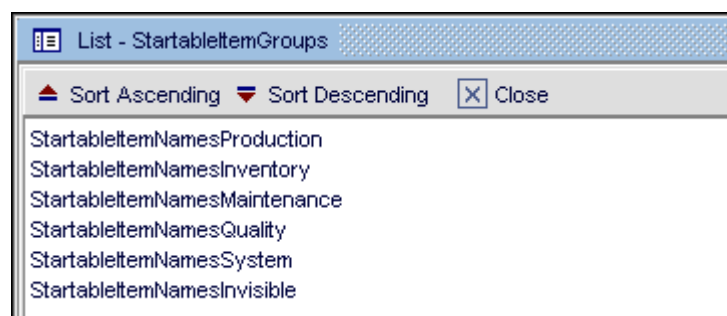


Figure 35: Group of invisible startable items in group of startable items

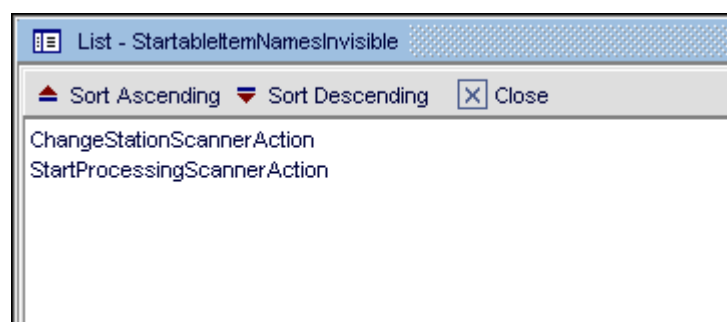


Figure 36: Invisible startable item in group of invisible startable items

The constructor of the actions in the **StartableItemNamesInvisible** list is called during initialization of the Cockpit configuration (e.g. at startup, user change, register at station).

Scanner Actions in the Cockpit

The abstract *ScannerAction* base class allows to implement scanner support in the Cockpit.

ScannerAction

```
public abstract class ScannerAction extends AbstractAction implements
    IBarcodeScannedListener {
    public ScannerAction() {
        super();
        // register barcode listener to the cockpit controller
        final IProductionExecutionClient pec =
            ServiceFactory.getService(IProductionExecutionClient.class);
        final ICockpit controller = (ICockpit) pec.getCurrentCockpit();
        controller.addBarcodeListener(this);
        // deregister is automatically done when cockpit is newly initialized
        // after changeUser() or changeStation()
    }
    @Override
    public void actionPerformed(ActionEvent e) {
        throw new MESRuntimeException("actionPerformed. Not supported for this action.");
    }
    @Override
    public abstract void barcodeScanned(String barcode);
}
```

The base class already provides the following functionality:

- Registers a barcode listener to the Cockpit controller in the *ScannerAction::ScannerAction()* constructor
The registered listener is active while the Cockpit view is visible and becomes automatically inactive when the Cockpit view is not displayed.
This means that the barcode listener is inactive when another view is displayed (e.g. Navigator, exception, execution).
- Calls a *ScannerAction::barcodeScanned(barcode)* method to pass the barcode
This method must be overloaded in the specific class that implements the business logic to react to the barcode.

A specific *ScannerAction* class, e.g. *MyScannerAction*, can inherit from *ScannerAction* and only needs to overwrite the *ScannerAction::barcodeScanned(String barcode)* method.

MyScannerAction

```
public class MyScannerAction extends ScannerAction {
    public MyScannerAction () {
        super();
        // specific code, if required
    }
}
```



```

    }
    @Override
    public void barcodeScanned(String barcode) {
        // specific business code to react to the passed barcode
    }
}

```

The following rules and recommendations apply:

- Each registered scanner action should use its own barcode format. Usually, the format is detected by a unique barcode prefix.
- The registered scanner action should be polite, i.e. it should only react to a barcode starting with a given prefix. Barcodes with other prefixes should be ignored.
- Multiple different non-conflicting scanner actions are possible, if they are polite.
- PharmaSuite uses the mechanism of invisible actions (page 129) to integrate the scanner support for the **Register at Station** action in the Cockpit.
- For more barcode details, see chapter "Changing Number Generation Schemes" in Volume 2 of the "Technical Manual Configuration and Extension" [A3] (page 171).

EXAMPLE: STATION-SPECIFIC CONFIGURATION

The application configuration concept allows to define a station-specific configuration of the startable items.

For details of application configurations, see chapter "Managing Configurations" in Volume 4 of the "Technical Manual Configuration and Extension" [A4] (page 171).

The example illustrates the following configuration:

- **Station_D-WC001** uses the **StartableItemNamesInvisible** list.
The list contains the **ChangeStationScannerAction** item which is enabled in the Cockpit.
- **Station_D-WC002** does not use the **StartableItemNamesInvisible** list and no invisible actions are available in the Cockpit.

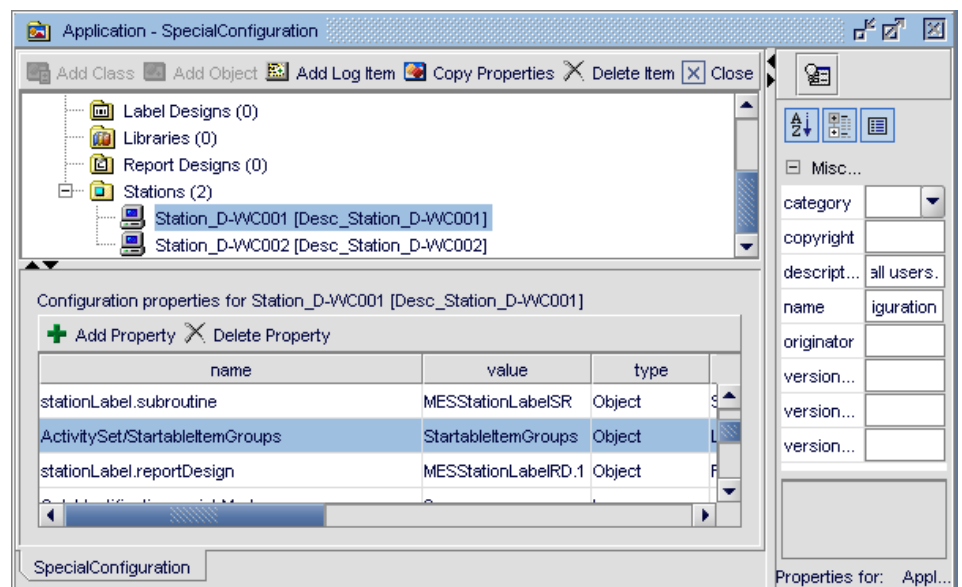


Figure 37: Station-specific configuration with ActivitySet/SetStartableItemGroups=StartableItemsGroup

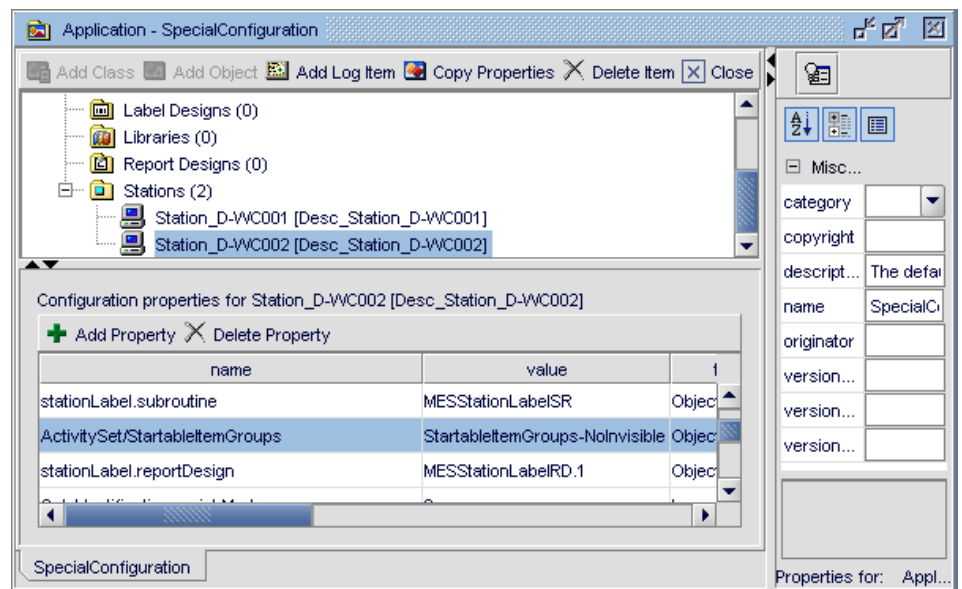


Figure 38: Station-specific configuration with ActivitySet/SetStartableItemGroups=StartableItemsGroup-Nolnvisible

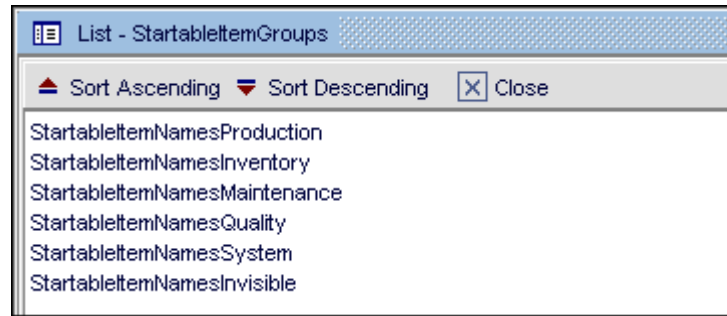


Figure 39: Group of invisible startable items in group of startable items

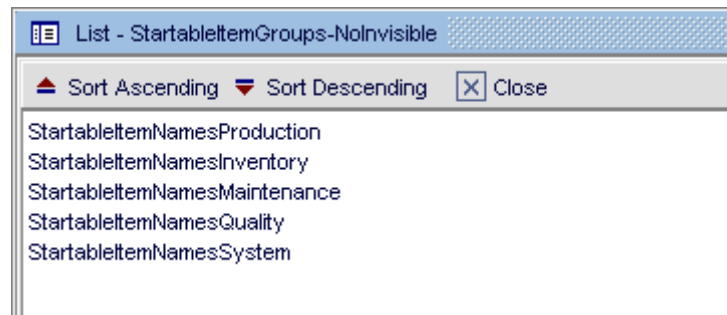


Figure 40: No group of invisible startable items in group of startable items

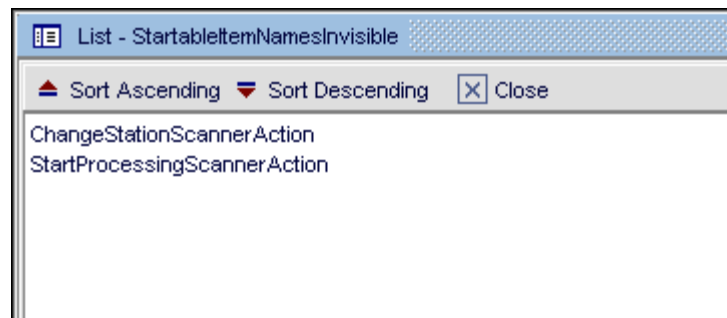


Figure 41: Invisible startable item in group of invisible startable items

Running Processes in the Cockpit

The Cockpit displays the running processes in the right part of its work area. The list below represents the order in which the sorting criteria are applied and the default sorting within an application area (e.g. Production):

1. Top level
 - Name
 - Order/workflow identifier
 - Localized workflow name (for pre-defined workflows based on activity sets)

- Type
 - Batch- and device-specific orders
 - Workflow-specific orders
 - Pre-defined workflows based on activity sets

2. Second level

- Procedure
 - Identifier of the procedure
- Unit procedure
 - Sequence count of an activity set step of the parent activity set
- Operation
 - Sequence count of the related activity set step of the parent activity set
- Template identifier
 - Template identifier of a runtime operation in case of ETO templates
- Instance count
 - Instance count of a runtime operation in case of ETO instances

TIP

A sequence count is defined in Recipe and Workflow Designer when a master recipe or master workflow is prepared for status change.

ADAPTING THE SORTING ORDER

To adapt the sorting of running processes in the Cockpit, proceed as follows:

1. Create a new java class that implements *com.rockwell.mes.apps.ebr.ifc.cockpit.view.ICockpitItemComparator* (or extends *CockpitItemComparator*).

Example code to sort the top level items by the starting date of ProcessOrderItem

```

public class CockpitItemComparatorCustom extends CockpitItemComparator {

    private static final int CMP_EQUAL = MESComparableToComparatorAdapter.EQUAL;

    @Override
    public int compareTopLevelItem(final CockpitTopLevelItem topLevelItem1,
                                   final CockpitTopLevelItem topLevelItem2) {
        int result = super.compareTopLevelItem(topLevelItem1, topLevelItem2);
        if (result != CMP_EQUAL) {
            // compare start date only for different top level items
            final Time date1 = getTopLevelDate(topLevelItem1);
            final Time date2 = getTopLevelDate(topLevelItem2);
            if (date1 != null && date2 != null) {
                int resultDateCompare = compareTopLevelDate(date1, date2);
            }
        }
    }

```

```

        if (resultDateCompare != CMP_EQUAL) {
            return resultDateCompare;
        }
    }
    return result;
}
private Time getTopLevelDate(final CockpitTopLevelItem cockpitItem) {
    Time topLevelStartDate = null;
    if (cockpitItem.getOrderStep() != null) {
        topLevelStartDate = getPOIStartDate(cockpitItem.getOrderStep());
    }
    return topLevelStartDate;
}
private Time getPOIStartDate(OrderStep os) {
    final ControlRecipe controlRecipe = os.getControlRecipe();
    final ProcessOrderItem poi = controlRecipe.getProcessOrderItem();
    Time startDate = MESNamedUDAProcessOrderItem.getActualStartDate(poi);
    return startDate;
}
private int compareTopLevelDate(final Time date1, final Time date2) {
    return date1.compareTo(date2);
}
}

```

2. Adapt *apps-ebr.xml*.

Example code of adapted *apps-ebr.xml*

```

<CockpitItemComparator>
  <implementationClass>
    com.rockwell.mes.apps.ebr.impl.cockpit.view.CockpitItemComparatorCustom
  </implementationClass>
</CockpitItemComparator>

```

TIP

The order in which the sorting criteria (i.e. top level element, procedure, unit procedure, operation, ETO template, ETO instance) are applied is fix and cannot be modified. To adapt the sorting of individual Cockpit items, implement *ICockpitItemComparator*.

Configuring Menu and Toolbar of the Production Response Client

This section contains general information about Production Response Client actions (page 137), implementing action classes (page 138), and the configuration of the main menu and toolbar (page 139).

To replace an existing action implementation, perform the following steps:

1. Implement a new action class (page 138) containing the adapted behavior of an existing action.
2. Modify the configuration (page 138) in order to use the new action class implementation.
3. Run the application to verify the changes in the Production Response Client and its actions.

To add a new action, perform the following steps:

1. Implement a new action class (page 138).
2. Add the configuration for the new action class (page 138).
3. Run the application to verify the changes in the Production Response Client and its actions.

What Are Actions in the Production Response Client?

In the Production Response Client, most functions are called by actions, located in the main menu or the toolbar.

An example for an action is the **Exit** action in the main menu or the **Print batch report** action in the toolbar.

The functionality and business logic of an action is encapsulated in a component, here a Java class.

Depending on your requirements, you either may implement a new action class containing the adapted behavior of an existing action or implement a new action class.

Implementing an Action Class

An action class has to extend the *AbstractAction* class. The implementation of the functionality of the action is located in the *actionPerformed* method.

Example: The *MYAction* action class simply executes some kind of business logic with MYC for the My Company vendor code.

```
javax.swing.AbstractAction;  
  
public class MYAction extends AbstractAction {  
    /** Comment for <code>serialVersionUID</code> */  
    private static final long serialVersionUID = 1L;  
    public void actionPerformed(ActionEvent e) {  
        // Implementation of the business logic of the Action  
    }  
}
```

Configuring Action Classes

Before you can use an implementation class of an action, the class must be configured. The configuration of implementation classes of Production Response Client actions is held in the **edb_uiConfiguration** list object.

To add a new action class or to replace an existing action class, a configuration entry is required. The XML file below contains such an entry in the *<item>* XML element with a unique key and the qualified name of the action itself (here: *com.rockwell.mes.cust.impl.MYAction*).

To replace an existing action class, change the value of the implementation class to the implementation class of your action. In order to add a new action class, add a new XML element that refers to the new action.

```
<item key = "ToolMyAction">  
    <action="com.rockwell.mes.cust.impl.MYAction" />  
</item>
```


Configuring Actions

In order to use an action in the Production Response Client, some details related to the action have to be defined. The details depend on the usage of the action (menu or toolbar).

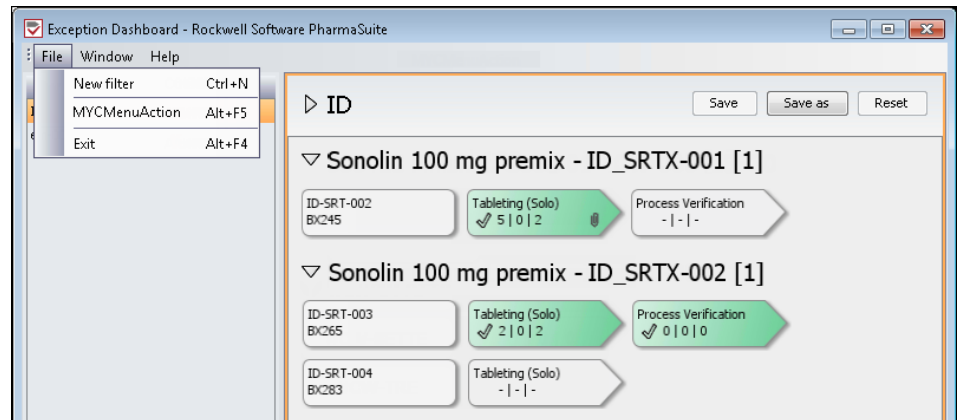


Figure 42: MYCMenuAction action in the menu

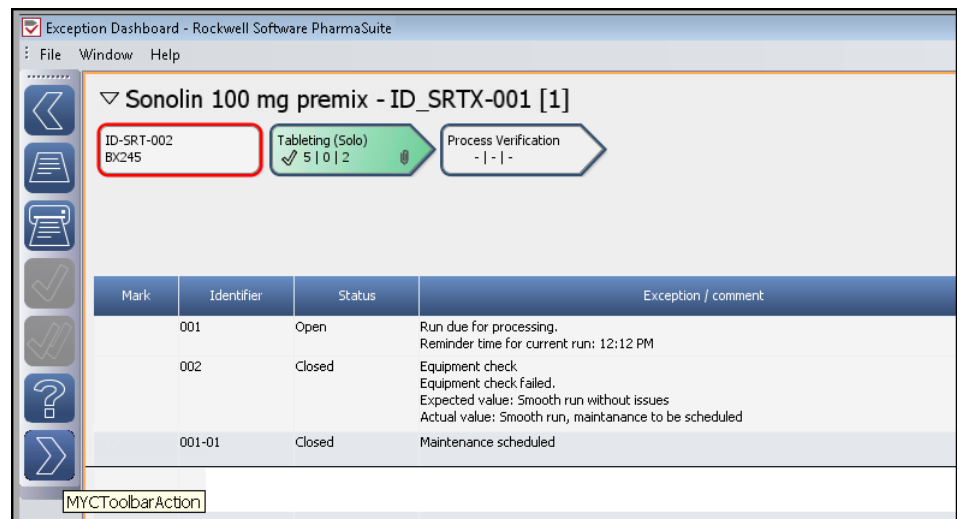


Figure 43: MYCToolbarAction action in the toolbar

Configuring an Action for the Menu Bar

The following properties must be configured to define a menu action:

Property	Value
Label	Message pack: ui_ExceptionDashboard Entry: <key> + "_Menu"
Accelerator	Message pack: ui_ExceptionDashboard Entry: <key> + "_Accelerator"
Icon	Image: "edb_" + <key>

Property	Value
Icon (toggled)	Image: "edb_" + <key> + "_toggled"

Configuring an Action for the Toolbar

The following properties must be configured to define a toolbar action:

Property	Value
Icon	Image: "edb_" + <key>
Icon (toggled)	Image: "edb_" + <key> + "_toggled"
Tooltip	Message pack: ui_ExceptionDashboard Entry: <key> + "_Tooltip"

Adding Filter Attributes to the Exception Dashboard of the Production Response Client

This section contains general information about Production Response Client filter attributes (page 141) and the implementation of custom filter attributes (page 147).

To create a custom filter attribute, perform the following steps:

1. Implement a new attribute filter class (page 147) for the desired filter attribute.
2. Modify the filter attribute configuration (page 162) in order to use the new filter attribute class implementation either as a standard or additional filter attribute.
3. Run the application to verify the changes in the Production Response Client and its filter attributes.

IMPORTANT

We highly recommend not to remove an added filter attribute after it has been used in the Production Response Client. If the removal is required anyhow, delete all user-defined filters that use this attribute before deleting the added filter attribute.

In general, filtering in the Production Response Client works by obtaining the result for each filter in the following way:

- Multiple independent SQL SELECT statements are executed.
- The results of the SELECT statements are combined, i.e. in order to determine the objects that have to be displayed according to the defined filter, an intersection of the results of the SELECT statements is built.

SELECT statements belong to the following filter areas: process order items, runtime unit procedures, and exceptions (see "Supported Database Objects" (page 142)).

What Are Filter Attributes in the Production Response Client?

In the Production Response Client, the visible list of orders that match the filter can be limited by defining filter criteria for the filter attributes. The supported filter attributes are displayed in the Filter Definition panel, which is located above the Overview panel.

A filter attribute is either **fixed (standard)** and always visible or **optional (additional)** and only visible if selected with the **More filter criteria** button.



Figure 44: Filter Definition panel with filter attributes

Example of filter attributes are: **Processing type**, **Status**, **Order ERP start**, and **Master recipe/workflow identifier**.

The functionality and business logic of a filter attribute is encapsulated in a component, here a Java class.

You can implement an additional filter attribute for supported database objects (page 142) and data types (page 144).

Supported Database Objects

In order to define a filter attribute, you have to know the column name(s) to be used in the WHERE condition of the SQL statement for the filter attribute you wish to use as filter criterion and its filter type.

- A column name consists of the table name (or its alias, respectively) and the SQL column name itself. The aliases of supported tables are available as constants, the SQL column names can be found in e.g. IDBSchema or the generated application table/UDA classes.
- A filter type indicates to which filter area a filter attribute belongs:
 - PROCESS_ORDER_ITEM: process order items
 - RT_UNIT_PROCEDURE: runtime unit procedures
 - UNIT_PROCEDURE_STATISTICS: exceptions

The value typed in the filter attribute-specific editor is added as filter criterion for the column(s) to the SQL statement that queries the visible objects.

Depending on the filter type, the following database tables are supported:

PROCESS_ORDER_ITEM

Table	SQL alias
PROCESS_ORDER	IProcessOrderItemModel.PROCESS_ORDER_ALIAS
PROCESS_ORDER_ITEM	IProcessOrderItemModel.PROCESS_ORDER_ITEM_ALIAS
UDA_ProcessOrderItem	IProcessOrderItemModel.PROCESS_ORDER_ITEM_UDA_ALIAS
BATCH	IProcessOrderItemModel.BATCH_ALIAS
UDA_Batch	IProcessOrderItemModel.BATCH_UDA_ALIAS
CONTROL_RECIPES	IProcessOrderItemModel.CONTROL_RECIPES_ALIAS

Table	SQL alias
MASTER_RECIPE	IProcessOrderItemModel.MASTER_RECIPE_ALIAS
UDA_MasterRecipe	IProcessOrderItemModel.MASTER_RECIPE_UDA_ALIAS
PART	IProcessOrderItemModel.PART_ALIAS
UDA_Part	IProcessOrderItemModel.PART_UDA_ALIAS
AT_X_RtProcedure	IProcessOrderItemModel.RTPROCEDURE_ALIAS
AT_X_DiscreteDevice	IProcessOrderItemModel.DEVICE_ALIAS
OBJECT_STATE	IProcessOrderItemModel.OBJECT_STATE_ALIAS
STATE	IProcessOrderItemModel.STATE_ALIAS
FSM_CONFIG_ITEM	IProcessOrderItemModel.FSM_CONFIG_ITEM_ALIAS

RT_UNIT_PROCEDURE

Table	SQL alias
AT_X_RtUnitProcedure	IRtUnitProcedureModel.RT_UNIT_PROCEDURE_ALIAS
AT_X_UnitProcedure	IRtUnitProcedureModel.UNIT_PROCEDURE_ALIAS
AT_X_RtProcedure	IRtUnitProcedureModel.RTPROCEDURE_ALIAS
WORK_CENTER	IRtUnitProcedureModel.WORK_CENTER_ALIAS
UDA_WorkCenter	IRtUnitProcedureModel.WORK_CENTER_UDA_ALIAS
CONTROL_RECIPE	IRtUnitProcedureModel.CONTROL_RECIPE_ALIAS
UDA_ProcessOrderItem	IRtUnitProcedureModel.PROCESS_ORDER_ITEM_UDA_ALIAS
AT_X_OrderToRtUnitProcedure	IRtUnitProcedureModel.ORDER2RTUNITPROCEDURE_ALIAS
ORDER_STEP	IRtUnitProcedureModel.ORDER_STEP_ALIAS
OBJECT_STATE	IRtUnitProcedureModel.OBJECT_STATE_ALIAS
STATE	IRtUnitProcedureModel.STATE_ALIAS
FSM_CONFIG_ITEM	IRtUnitProcedureModel.FSM_CONFIG_ITEM_ALIAS

UNIT_PROCEDURE_STATISTICS

Table	SQL alias
AT_X_ExceptionRecord	IUnitProcedureStatistics.EXCEPTION_RECORD_ALIAS
AT_X_ExceptionObjectRelation	IUnitProcedureStatistics.EXCEPTION_OBJECT_RELATION_ALIAS
AT_X_RtUnitProcedure	IUnitProcedureStatistics.RT_UNIT_PROCEDURE_ALIAS

Data Types Supported as Filter Attributes

The following data types are supported as filter attributes. PharmaSuite provides base classes for the data types that can be used if you wish to add your own filter attributes. Depending on the data type a corresponding UI editor is automatically generated by PharmaSuite.

■ String data type

An editor to enter text as filter criterion.

Allows to specify a default filter text, if the search is case-sensitive, and where to search for matches (`TextPositionMatchMode.FROM_START`, `TextPositionMatchMode.EXACTLY`, `TextPositionMatchMode.ANYWHERE`). Use *TextBasedFilterAttributeConfiguration* to specify your own default values or use *getDefaultConfiguration()* to get pre-defined default values (no default text, case-insensitive, match anywhere (contains)).

Base class: *AbstractTextBasedFilterAttribute*



Figure 45: Editor for text-based attributes

■ String list data type

An editor to select pre-defined text as filter criterion. Mostly used to filter on a potentially large set of strings, which is not fixed, but likely varies only from time to time. Offers a quick search field to limit the available (visible) options and allows to select all values.

Use *OptionListFilterAttributeConfiguration* to specify your own pre-selected values or use *AbstractOptionListFilterAttribute.<String>getDefaultConfiguration()* to have no values pre-selected.

Base class: *AbstractStringListFilterAttribute*

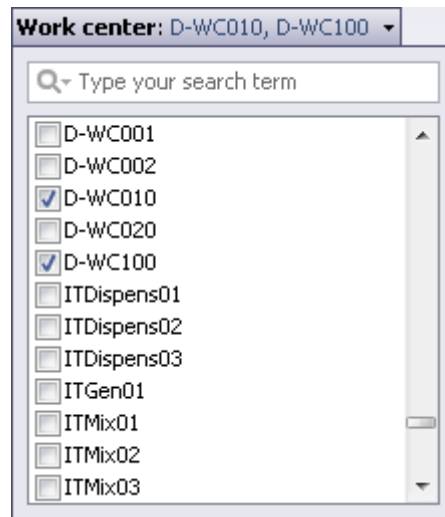


Figure 46: Editor for string list attributes

■ Option list data type

An editor to select pre-defined options (e.g. status values) as filter criterion. Additionally, all options can be selected.

Use *OptionListFilterAttributeConfiguration* to specify your own pre-selected values or use *getDefaultConfiguration()* to have no values pre-selected.

Base class: *AbstractOptionListFilterAttribute*

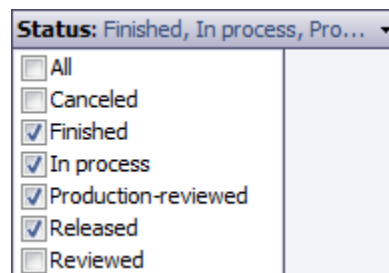


Figure 47: Editor for option list attributes

■ Choice list data type

An editor to select choice list values as filter criterion. Allows to specify if a value is mandatory or if null is allowed. Additionally, all options can be selected.

Use *OptionListFilterAttributeConfiguration* to specify your own pre-selected values or use *AbstractOptionListFilterAttribute.<Long> getDefaultConfiguration()* to have no values pre-selected.

Base class: *AbstractChoiceListFilterAttribute*

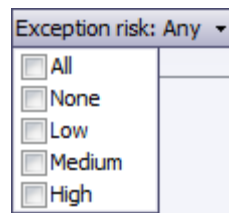


Figure 48: Editor for choice list attributes

■ **Boolean** data type

An editor to select **True** or **False** as filter criterion. Additionally, all options can be selected.

Use *OptionListFilterAttributeConfiguration* to specify your own pre-selected values or use *AbstractOptionListFilterAttribute.<Boolean>getDefaultConfiguration()* to have no values pre-selected.

Base class: *AbstractBooleanFilterAttribute*

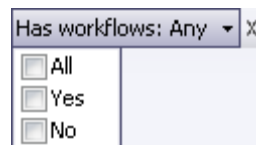


Figure 49: Editor for boolean attributes

■ **Date** data type

An editor to select durations (Between .. and .., Within the last, More than .. ago) as filter criterion.

Allows to specify the pre-selected filter mode

(*DateFilterMode.BETWEEN_DATE_AND_DATE*,

DateFilterMode.WITHIN_LAST_DURATION,

DateFilterMode.MORE_THAN_DURATION_AGO), default values for date/time and duration, and if a date contains date and time or date only.

Use *DateFilterAttributeConfiguration* to specify your own default values or use *getDefaultConfiguration()* to get pre-defined default values (between date and date pre-selected, no default dates and duration).

Base class. *AbstractDateFilterAttribute*

Figure 50: Editor for date attributes (Date)

Figure 51: Editor for date attributes (Duration)

Implementing a Filter Attribute

Before you implement a new filter attribute, you have to know its data type. Depending on the data type, the implementation looks different because your new filter attribute must subclass the corresponding data type-specific base class (see "Data Types Supported for Filter Attributes" (page 144)). However, some methods are common to all data types.

For details how to configure the filter attributes, see "Configuring Filter Attributes" (page 162).

TIP

In all code examples the package declaration and the imports are omitted for the sake of clarity.

The name of a filter attribute is also the name of its implementing Java class.

Methods Common to All Data Types

The following methods have to be implemented for each filter attribute, regardless of its data type:

- A default constructor, which calls the corresponding specific constructor with appropriate default values.
- *getDisplayName()*
Returns the UI text that will be displayed on the corresponding UI filter control. We recommend to use a message pack to enable localization.
- *getFilterType()*
Returns the filter type of the attribute (i.e. the filter area to which this filter attribute belongs).
- *createFilterAttribute(newConfiguration)*
Creates and returns a new filter attribute of this type by calling the data type-specific constructor.
Remember that filter attributes are immutable and this method acts as a factory when an attribute needs to be updated.

TIP

Filter attributes are serializable, therefore your new filter attribute must declare a `serialVersionUID` (see code examples in the data type-specific implementation description).

Filter attributes are immutable, which means that they should not have any setter methods.

String Data Type

The **String** data type results in a text-based filter attribute. It must inherit from the *AbstractTextBasedFilterAttribute* base class. As an example, we provide a filter attribute that filters for master recipe comments. We call our filter attribute *MasterRecipeCommentFilterAttribute*.

```
/**
 * Filter attribute for master recipe comments
 */
public class MasterRecipeCommentFilterAttribute
    extends AbstractTextBasedFilterAttribute {

    // Use/generate a real "serialVersionUID", because filter attributes are
    // serialized to make them persistent
```

```

private static final long serialVersionUID = -4321637190142090572L;

// The column names used in the WHERE condition of the attribute.
// Here we only have the corresponding column for the master recipe comment, which is
// in the UDA table of the master recipe.
private static final String[] COLUMN_NAMES =
    { IProcessOrderItemModel.MASTER_RECIPE_UDA_ALIAS + "." +
      MESNamedUDAMasterRecipe.SQL_UDA_COMMENT };

/**
 * Constructor
 *
 * @param configuration attribute configuration
 */
public MasterRecipeCommentFilterAttribute
    (final TextBasedFilterAttributeConfiguration configuration) {
    super(configuration);
}

/**
 * Default constructor (necessary)
 */
public MasterRecipeCommentFilterAttribute() {
    this(getDefaultConfiguration());
}

@Override
public String getDisplayName() {
    // TASK: il8n
    return "Master recipe comment";
}

@Override
public FilterType getFilterType() {
    // The UDA table for master recipes is part of the process order item
    // filter area (i.e. it is joined with the process order items).
    // Therefore the filter type must be PROCESS_ORDER_ITEM.
    return FilterType.PROCESS_ORDER_ITEM;
}

@Override
protected String[] getColumnNames() {
    return COLUMN_NAMES;
}

@Override
public AbstractTextBasedFilterAttribute createFilterAttribute
    (TextBasedFilterAttributeConfiguration configuration) {
    return new MasterRecipeCommentFilterAttribute(configuration);
}
}

```

Besides the common methods (page 148), you have to implement the following additional ones:

- A specific constructor, which calls the appropriate constructor of the base class.
- *getColumnNames()*
Returns the column names that are used in the WHERE condition of the filter attribute (see "Supported Database Objects" (page 142)). The construction of the

WHERE condition itself is generic and done by the framework. If you add multiple column names, the result will contain objects with the filter criteria in any given column (OR).

The Filter Definition panel with the new attribute may look like this, depending on the actual configuration of the attributes:

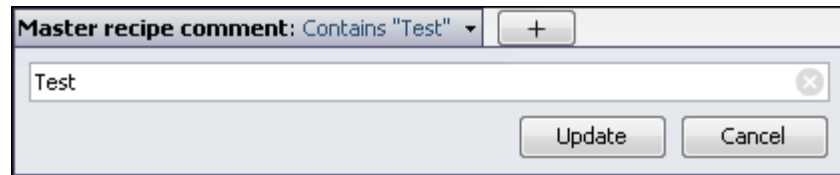


Figure 52: MasterRecipeCommentFilterAttribute in the Filter Definition panel

String List Data Type

The **String list** data type is suitable to filter on a potentially large set of pure strings, which is not fixed, but may vary from time to time (e.g. the names of all work centers, materials, or master recipes available in the database). Usually, pure strings are names or identifiers that do not have to be localized. Thus the **String list** data type is a simplified version of the **Option list** data type (page 152). As an example, we provide a filter attribute that filters for master recipe names. We call our filter attribute *MasterRecipeIdentifiersFilterAttribute*.

```
/**
 * Filter attribute for the master recipe identifier (as string list,
 * as opposed to mere string of standard filter attribute)
 */
public class MasterRecipeIdentifiersFilterAttribute
    extends AbstractStringListFilterAttribute {

    // use/generate a real "serialVersionUID", because filter attributes
    // are serialized to make them persistent
    private static final long serialVersionUID = -3222742307790352483L;

    // WHERE condition of attribute.
    // Do not forget to use the placeholder named "VAR_NAME_FOR_IN_CLAUSE"
    // in the IN clause of the statement so that the framework can replace
    // it with the actual master recipe names the user selected
    private static final String WHERE_CONDITION =
        IProcessOrderItemModel.MASTER_RECIPE_ALIAS + "."
        + IDBSchema.MASTER_RECIPE_MASTER_RECIPE_NAME
        + " IN ( {" + VAR_NAME_FOR_IN_CLAUSE + " } )";

    /**
     * Constructor
     *
     * @param theConfiguration configuration
     */
    public MasterRecipeIdentifiersFilterAttribute(final
        OptionListFilterAttributeConfiguration<String> theConfiguration) {
        super(theConfiguration);
    }

    /**
```

```

    * Default constructor.
    */
    public MasterRecipeIdentifiersFilterAttribute() {
        this(AbstractOptionListFilterAttribute.<String> getDefaultConfiguration());
    }

    @Override
    public String getWhereConditionConstant() {
        return WHERE_CONDITION;
    }

    @Override
    public FilterType getFilterType() {
        // The master recipe is part of the process order item filter area.
        // Therefore the filter type must be PROCESS_ORDER_ITEM
        return FilterType.PROCESS_ORDER_ITEM;
    }

    @Override
    public String getDisplayName() {
        // TASK: i18n
        return "Master recipe";
    }

    @Override
    public List<String> getAllLocalizedOptions() {
        final List<String> allMasterRecipeNames = getAllMasterRecipeNames();
        return allMasterRecipeNames;
    }

    private static List<String> getAllMasterRecipeNames() {
        final SqlStatement sqlStatement = new SqlStatement(IDBSchema.
            MASTER_RECIPE_TABLE_NAME, "mr", null);
        int mrNameColumnIndex = sqlStatement.addColumn(IDBSchema.
            MASTER_RECIPE_MASTER_RECIPE_NAME, "mr", "distinct");
        sqlStatement.addOrderByColumn(mrNameColumnIndex);

        final List<String[]> result = sqlStatement.execute();

        final List<String> mrNames = new ArrayList<>();
        for (final String[] res : result) {
            mrNames.add(res[mrNameColumnIndex]);
        }

        return mrNames;
    }

    @Override
    public AbstractOptionListFilterAttribute createFilterAttribute(
        OptionListFilterAttributeConfiguration<String> theConfiguration) {
        return new MasterRecipeIdentifiersFilterAttribute(theConfiguration);
    }
}

```

Besides the common methods (page 148), you have to implement the following additional ones:

- A specific constructor, which calls the appropriate constructor of the base class.
- *getWhereConditionConstant()*
Returns the WHERE condition constant of the filter attribute, i.e. the WHERE

condition that contains the corresponding placeholder for the actual filter values (see "Supported Database Objects" (page 142)).

- *getAllLocalizedOptions ()*
Returns the list of all localized string values to filter for.

TIP

For details of the UI rendering of a specific data type, see section "Data Types Supported as Filter Attributes" (page 144).

Option List Data Type

The **Option list** data type is appropriate for filter attributes that filter for a set of options that is not based on a choice list (for this purpose use the **Choice list** data type) and that is also not a pre-defined and fixed set of strings (for this purpose the **String list** data type). A use case for the **Option List** data type is the order status, which is already a standard filter attribute of the system. As an example, we provide a filter attribute that filters for the batch status. We call our filter attribute *BatchStatusFilterAttribute*.

```
/**
 * Filter attribute for the batch status
 */
public class BatchStatusFilterAttribute
    extends AbstractOptionListFilterAttribute<String> {

    // Use/generate a real "serialVersionUID", because filter attributes are
    // serialized to make them persistent
    private static final long serialVersionUID = 8995762695437036708L;

    // WHERE condition of attribute. Don't forget to use the placeholder
    // named "VAR_NAME_FOR_IN_CLAUSE" in the IN clause of the statement so that
    // the framework can replace it with the actual batch states the user selected
    private static final String WHERE_CONDITION =
        IProcessOrderItemModel.BATCH_ALIAS + "." + IDBSchema.BATCH_BATCH_KEY //
        + " IN ( SELECT OBJECT_STATE.object_key " //
        + "     FROM OBJECT_STATE " //
        + "     WHERE OBJECT_STATE.object_type = " + IObjectTypes.TYPE_BATCH //
        + "     AND OBJECT_STATE.state_key IN " //
        + "         ( SELECT STATE.state_key " //
        + "         FROM STATE " //
        + "         WHERE STATE.state_name IN ( {" + VAR_NAME_FOR_IN_CLAUSE + "} ) " //
        + "         AND STATE.fsm_key IN " //
        + "             ( SELECT FSM_CONFIG_ITEM.fsm_key " //
        + "             FROM FSM_CONFIG_ITEM " //
        + "             WHERE FSM_CONFIG_ITEM.fsm_relationship_name=N'BatchQuality' " //
        + "             ))";

    private static final List<String> BATCH_STATES_LOCALIZED_NAMES = new ArrayList<>();

    private static Map<String, String> batchStatusLocalizedName2InternalNameMap =
        new HashMap<>();

    private static Map<String, String> batchStatusInternalName2LocalizedNameMap =
        new HashMap<>();

    static {
```

```

final String fsmRelationship = "BatchQuality";
// AbstractBatchQualityTransitionEventListener.FSM_REL_SHIP;
final List<State> orderStates = FSMHelper.getFsmStatesByObjectType(
    IOObjectTypes.TYPE_BATCH, fsmRelationship);

for (final State state : orderStates) {
    final String internalValue = state.getName();
    final String localizedValue = state.getLocalizedName();

    BATCH_STATES_LOCALIZED_NAMES.add(localizedValue);

    batchStatusLocalizedName2InternalNameMap.put(localizedValue, internalValue);
    batchStatusInternalName2LocalizedNameMap.put(internalValue, localizedValue);
}
}

/**
 * Constructor
 *
 * @param theConfiguration attribute configuration
 */
public BatchStatusFilterAttribute
    (OptionListFilterAttributeConfiguration<String> theConfiguration) {
    super(theConfiguration);
}

/**
 * Default constructor (necessary)
 */
public BatchStatusFilterAttribute() {
    this(AbstractOptionListFilterAttribute.<String> getDefaultConfiguration());
}

@Override
public String getDisplayName() {
    // TASK: il8n
    return "Batch status";
}

@Override
public FilterType getFilterType() {
    // Batch is part of the process order item filter area (i.e. it is joined with
    // the process order items). Therefore the filter type must be PROCESS_ORDER_ITEM
    return FilterType.PROCESS_ORDER_ITEM;
}

@Override
public String getWhereConditionConstant() {
    return WHERE_CONDITION;
}

@Override
public List<String> getAllLocalizedOptions() {
    // return list of localized batch status names
    return BATCH_STATES_LOCALIZED_NAMES;
}

@Override
public AbstractOptionListFilterAttribute createFilterAttribute
    (OptionListFilterAttributeConfiguration<String> theConfiguration) {
    return new BatchStatusFilterAttribute(theConfiguration);
}
}

```

```

@Override
public String getLocalizedStringForInternalValue(String internalValue) {
    // get localized status name from given internal status name
    return batchStatusInternalName2LocalizedNameMap.get(internalValue);
}

@Override
public String getInternalValueForLocalizedString(String localizedValue) {
    // get internal status name for given localized status name
    return batchStatusLocalizedName2InternalNameMap.get(localizedValue);
}
}

```

As another example, we provide a filter attribute that filters for the units of measure of all materials. We call our filter attribute *MaterialUoMFilterAttribute*.

```

/**
 * Filter attribute for the unit of measure of the produced material
 */
public class MaterialUoMFilterAttribute
    extends AbstractOptionListFilterAttribute<Long> {

    // Use/generate a real "serialVersionUID", because filter attributes are
    // serialized to make them persistent
    private static final long serialVersionUID = 2068946700204348588L;

    // WHERE condition of the attribute. The material UoM is stored in a part UDA.
    // The IN clause construct "IN ( { " + VAR_NAME_FOR_IN_CLAUSE + " } )" is
    // evaluated by the framework, where VAR_NAME_FOR_IN_CLAUSE is a placeholder
    // for the actually selected UoMs
    private static final String WHERE_CONDITION =
        IProcessOrderItemModel.PART_UDA_ALIAS + "."
        + MESNamedUDAPart.SQL_UDA_UNITOFMEASURE + " IN ( { "
        + VAR_NAME_FOR_IN_CLAUSE + " } )";

    private static Map<String, Long> uomLocalizedName2Key = new HashMap<>();

    private static Map<Long, String> uomKey2LocalizedName = new HashMap<>();

    static {
        final UnitOfMeasure[] allUoMs = PCContext.getFunctions().getUnitOfMeasures();

        for (final IUnitOfMeasure uom : allUoMs) {
            final long key = ((UnitOfMeasure) uom).getKey();
            final String symbol = uom.getSymbol();

            uomLocalizedName2Key.put(symbol, key);
            uomKey2LocalizedName.put(key, symbol);
        }
    }

    /**
     * Constructor
     *
     * @param theConfiguration attribute configuration
     */
    public MaterialUoMFilterAttribute
        (OptionListFilterAttributeConfiguration<Long> theConfiguration) {
        super(theConfiguration);
    }
}

```



```

/**
 * Default constructor (necessary)
 */
public MaterialUoMFilterAttribute() {
    this(AbstractOptionListFilterAttribute.<Long> getDefaultConfiguration());
}

@Override
public String getDisplayName() {
    // TASK: il8n
    return "Material UoM";
}

@Override
public FilterType getFilterType() {
    // The UDA table for parts is part of the process order item filter area
    // (i.e. it is joined with the process order items).
    // Therefore the filter type must be PROCESS_ORDER_ITEM
    return FilterType.PROCESS_ORDER_ITEM;
}

@Override
public String getWhereConditionConstant() {
    return WHERE_CONDITION;
}

@Override
public List<String> getAllLocalizedOptions() {
    final List<String> allMaterialUoMs = getAllMaterialUoMs();
    return allMaterialUoMs;
}

private List<String> getAllMaterialUoMs() {
    final SqlStatement sqlStatement = new SqlStatement("UDA Part");
    int uomColumnIndex = sqlStatement.addColumn(MESNamedUDAPart.SQL_UDA_UNITOFMEASURE,
        "UDA_Part", "distinct");

    final List<String[]> result = sqlStatement.execute();

    final List<String> allUoMs = new ArrayList<>();
    for (final String[] res : result) {
        final String uomKeyString = res[uomColumnIndex];
        if (!StringUtils.isEmpty(uomKeyString)) {
            final Long uomKey = Long.parseLong(uomKeyString);
            allUoMs.add(getLocalizedStringForInternalValue(uomKey));
        }
    }

    return allUoMs;
}

@Override
public String getLocalizedStringForInternalValue(final Long internalValue) {
    return uomKey2LocalizedString.get(internalValue);
}

@Override
public Long getInternalValueForLocalizedString(String localizedValue) {
    return uomLocalizedString2Key.get(localizedValue);
}

@Override

```

```
public AbstractOptionListFilterAttribute createFilterAttribute
    (OptionListFilterAttributeConfiguration<Long> theConfiguration) {
    return new MaterialUoMFilterAttribute(theConfiguration);
}
}
```

Besides the common methods (page 148), you have to implement the following additional ones:

- A specific constructor, which calls the appropriate constructor of the base class.
- *getWhereConditionConstant()*
Returns the WHERE condition constant of the filter attribute, i.e. the WHERE condition that contains the corresponding placeholder for the actual filter values (see "Supported Database Objects" (page 142)).
- *getAllLocalizedOptions()*
Returns the list of all localized (if necessary) string values to filter for.
- *getLocalizedStringForInternalValue()*
Returns the localized string representation for the given internal value (in the above example for the given internal status name).
- *getInternalValueForLocalizedString()*
Returns the internal value (in the above example the corresponding internal status name) for the given localized string representation of the value.

TIP

For details of the UI rendering of a specific data type, see section "Data Types Supported as Filter Attributes" (page 144).

Choice List Data Type

Filter attributes of the **Choice list** data type filter for choice list values. They must inherit from the *AbstractChoiceListFilterAttribute* base class. As an example, we provide a filter attribute that filters for the material type of the produced material. We call our filter attribute *MaterialTypeFilterAttribute*. The choice list of the material types is called *MaterialType*.

```
/**
 * Filter attribute for the type of the produced material
 */
public class MaterialTypeFilterAttribute extends AbstractChoiceListFilterAttribute {

    // Use/generate a real "serialVersionUID", because filter attributes are
    // serialized to make them persistent
    private static final long serialVersionUID = -7901204891888441551L;

    // WHERE condition of the attribute. The material type is stored in a part UDA.
    // The IN clause construct "IN ( {" + VAR_NAME_FOR_IN_CLAUSE + " } )" is evaluated
    // by the framework, whereas VAR_NAME_FOR_IN_CLAUSE is a placeholder for the
    // actually selected material types
    private static final String WHERE_CONDITION = IProcessOrderItemModel.PART_UDA_ALIAS
```

```

        + "." + MESNamedUDAPart.SQL_UDA_MATERIALTYPE
        + " IN ( {" + VAR_NAME_FOR_IN_CLAUSE + " } ) ";

/**
 * Constructor
 *
 * @param theConfiguration attribute configuration
 */
public MaterialTypeFilterAttribute
    (OptionListFilterAttributeConfiguration<Long> theConfiguration) {
    // "true" means that a choice is mandatory
    super(theConfiguration);
}

/**
 * Default constructor (necessary)
 */
public MaterialTypeFilterAttribute() {
    this(AbstractOptionListFilterAttribute.<Long> getDefaultConfiguration());
}

@Override
public String getDisplayName() {
    // TASK: i18n
    return "Material type";
}

@Override
public FilterType getFilterType() {
    // The UDA table for parts is part of the process order item filter area (i.e.
    // it is joined with the process order items). Therefore the filter type
    // must be PROCESS_ORDER_ITEM
    return FilterType.PROCESS_ORDER_ITEM;
}

@Override
public String getWhereConditionConstant() {
    return WHERE_CONDITION;
}

@Override
protected ChoiceListCache createChoiceListCache() {
    // "true" means that a choice is mandatory (null values not allowed)
    return new ChoiceListCache("MaterialType", true);
}

@Override
public AbstractChoiceListFilterAttribute createFilterAttribute
    (OptionListFilterAttributeConfiguration<Long> theConfiguration) {
    return new MaterialTypeFilterAttribute(theConfiguration);
}
}

```

Besides the common methods (page 148), you have to implement the following additional ones:

- A specific constructor, which calls the appropriate constructor of the base class.
- *getWhereConditionConstant()*
Returns the WHERE condition constant of the filter attribute, i.e. the WHERE

condition that contains the corresponding placeholder for the actual filter values (see "Supported Database Objects" (page 142)).

■ *createChoiceListCache()*

Defines which choice list (or which choice elements, if only a subset of the choice list is used) is referenced by the attribute. Also defines if the selection of a choice value is mandatory or if null is allowed.

The Filter Definition panel with the new attribute may look like this, depending on the actual configuration of the attributes:

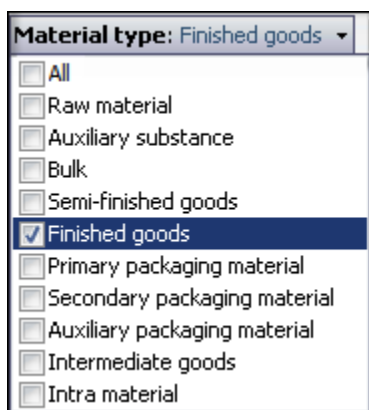


Figure 53: *MaterialTypeFilterAttribute* in the Filter Definition panel

Boolean Data Type

As an example for a filter attribute of **Boolean** data type, we provide a filter attribute that filters for Dispense unit procedures. We call our filter attribute *WeighAndDispenseUnitProcedureFilterAttribute*.

```
/**
 * Filter attribute which filters if a unit procedure is a Dispense unit procedure or not
 */
public class WeighAndDispenseUnitProcedureFilterAttribute
    extends AbstractBooleanFilterAttribute {

    // Use/generate a real "serialVersionUID", because filter attributes are
    // serialized to make them persistent
    private static final long serialVersionUID = 5089532800878987259L;

    // WHERE condition of filter attribute. The W&D flag is stored in
    // the unit procedure AT object.
    // The IN clause construct "IN ( {" + VAR_NAME_FOR_IN_CLAUSE + " } )" is
    // evaluated by the framework, whereas VAR_NAME_FOR_IN_CLAUSE is a placeholder
    // for the actually selected boolean values
    private static final String WHERE_CONDITION =
        IRTUnitProcedureModel.UNIT_PROCEDURE_ALIAS + "."
        + IMESUnitProcedure.SQL_COL_NAME_ISWEIGHANDDISPENSE + "
        IN ( {" + VAR_NAME_FOR_IN_CLAUSE + " } )";

    /**
     * Constructor
     *
     * @param theConfiguration attribute configuration
     */
}
```

```

    */
    public WeighAndDispenseUnitProcedureFilterAttribute
        (OptionListFilterAttributeConfiguration<Boolean> theConfiguration) {
        super(theConfiguration);
    }

    /**
     * Default constructor (necessary)
     */
    public WeighAndDispenseUnitProcedureFilterAttribute() {
        this(AbstractOptionListFilterAttribute.<Boolean> getDefaultConfiguration());
    }

    @Override
    public String getDisplayName() {
        // TASK: il8n
        return "Is W&D unit procedure";
    }

    @Override
    public FilterType getFilterType() {
        // The unit procedure AT table is part of the unit rt unit procedure filter area.
        // Therefore the filter type must be RT_UNIT_PROCEDURE.
        return FilterType.RT_UNIT_PROCEDURE;
    }

    @Override
    public String getWhereConditionConstant() {
        return WHERE_CONDITION;
    }

    @Override
    public AbstractBooleanFilterAttribute createFilterAttribute
        (OptionListFilterAttributeConfiguration<Boolean> theConfiguration) {
        return new WeighAndDispenseUnitProcedureFilterAttribute(theConfiguration);
    }
}

```

Besides the common methods (page 148), you have to implement the following additional ones:

- A specific constructor, which calls the appropriate constructor of the base class.
- *getWhereConditionConstant()*
Returns the WHERE condition constant of the filter attribute, i.e. the WHERE condition that contains the corresponding placeholder for the actual filter values (see "Supported Database Objects" (page 142)).

The Filter Definition panel with the new attribute may look like this, depending on the actual configuration of the attributes:

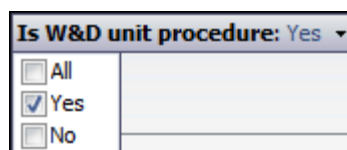


Figure 54: *WeighAndDispenseUnitProcedureFilterAttribute* in the Filter Definition panel

Date Data Type

Filter attributes of the **Date** data type support multiple filter modes (see section "Data Types Supported as Filter Attributes" (page 144)). As an example, we provide a filter attribute that filters for the expiry date of an order-related batch. We call our filter attribute *BatchExpiryDateFilterAttribute*.

```
/**
 * Filter attribute for the batch expiry date
 */
public class BatchExpiryDateFilterAttribute extends AbstractDateFilterAttribute {

    // Use/generate a real "serialVersionUID", because filter attributes are
    // serialized to make them persistent
    private static final long serialVersionUID = 6060945156698906617L;

    // Date column to filter for
    private static final String SQL_DATE_COLUMN = IProcessOrderItemModel.BATCH_ALIAS
        + "." + IDBSchema.BATCH_EXPIRATION_TIME;

    // Since date filter attributes support multiple filter modes we also need several
    // WHERE conditions which are stored in a map
    private static final Map<SqlDateFilterMode, String> WHERE_CONDITIONS =
        new EnumMap<>(SqlDateFilterMode.class);

    static {
        // Initialize the mapping of the placeholder that is used in the generic
        // WHERE conditions of the base class to the actual database column.
        // "DATE_ATTRIBUTE_VAR_NAME_IN_WHERE_CONDITION" is defined in the base class
        final Map<String, String> variablesMap = new HashMap<>();
        variablesMap.put(DATE_ATTRIBUTE_VAR_NAME_IN_WHERE_CONDITION, SQL_DATE_COLUMN);

        // Define the necessary WHERE conditions.
        // "SqlDateFilterMode", "WHERE_CONDITION_FROM_TO", "WHERE_CONDITION_AFTER"
        // and "WHERE_CONDITION_BEFORE" are defined in the base class
        WHERE_CONDITIONS.put(SqlDateFilterMode.FROM_TO_DATE,
            DataBaseUtility.resolvePlaceHolders(WHERE_CONDITION_FROM_TO,
                variablesMap, true));
        WHERE_CONDITIONS.put(SqlDateFilterMode.AFTER_DATE,
            DataBaseUtility.resolvePlaceHolders(WHERE_CONDITION_AFTER,
                variablesMap, true));
        WHERE_CONDITIONS.put(SqlDateFilterMode.BEFORE_DATE,
            DataBaseUtility.resolvePlaceHolders(WHERE_CONDITION_BEFORE,
                variablesMap, true));
    }

    /**
     * Constructor
     *
     * @param theConfiguration attribute configuration
     */
    public BatchExpiryDateFilterAttribute
        (DateFilterAttributeConfiguration theConfiguration) {
        super(theConfiguration);
    }

    /**
     * Default constructor (necessary)
     */
    public BatchExpiryDateFilterAttribute() {
        this(getDefaultConfiguration());
    }
}
```

```

    }

    @Override
    public String getDisplayName() {
        // TASK: i18n
        return "Batch expiry date";
    }

    @Override
    public FilterType getFilterType() {
        // Batch is part of the process order item filter area (i.e. it is joined with the
        // process order items). Therefore the filter type must be PROCESS_ORDER_ITEM
        return FilterType.PROCESS_ORDER_ITEM;
    }

    @Override
    public String getWhereConditionConstant(SqlDateFilterMode mode) {
        return WHERE_CONDITIONS.get(mode);
    }

    @Override
    public AbstractDateFilterAttribute createFilterAttribute
        (DateFilterAttributeConfiguration theConfiguration) {
        return new BatchExpiryDateFilterAttribute(theConfiguration);
    }
}

```

Besides the common methods (page 148), you have to implement the following additional ones:

- A specific constructor, which calls the appropriate constructor of the base class.
- *getWhereConditionConstant()*
 The *getWhereConditionConstant()* method of **Date** data type has a parameter of the *SqlDateFilterMode* type. The framework passes the user-activated filter mode to the parameter. Depending on the filter mode, the method returns the appropriate WHERE condition.
 The WHERE conditions for the different filter modes are prepared in a separate static block (see example code above), whereas the replacement of the column placeholders is done immediately.

By default, dates for this filter attribute allow to define the time in addition to the date. If only the date shall be specified without time, the *containsTime()* method has to be overridden to return **false**.

The Filter Definition panel with the new attribute may look like this, depending on the actual configuration of the attributes:

Figure 55: BatchExpiryDateFilterAttribute in the Filter Definition panel

Configuring Filter Attributes

Before a filter attribute is visible in the Production Response Client, the configuration has to be adapted. You can configure your filter attribute to be always visible as a **fixed (standard)** attribute or to be only visible as an **optional (additional)** attribute if selected with the **More filter criteria** button.

The configuration of the respective attribute type is read from the **prc_Standard_Filter_Attributes** and the **prc_Additional_Filter_Attributes** list objects. The lists to be used are configured with the **Form/ExceptionDashboard/prc_Standard_Filter_Attributes.customerName** and **Form/ExceptionDashboard/prc_Additional_Filter_Attributes.customerName** configuration keys (see chapter "Configuration Keys of PharmaSuite" in Volume 4 of the "Technical Manual Configuration and Extension" [A4] (page 171)).

To make your new filter attribute visible, add the fully qualified class name to one of the lists.

Material Handling for DCS

This section describes the implementations that are provided by PharmaSuite in the context of the DCS Adapter (see "Technical Manual DCS Adapter" [A5] (page 171)).

PharmaSuite provides implementations for integration touchpoints of a DCS that are intended to be sent by a DCS to an MES. The following requests are handled by PharmaSuite:

- **Process Consumed Material**
Receive information about material consumptions (aka goods receipts) performed on the DCS as part of a specific order.
PharmaSuite applies the corresponding changes to the inventory and documents the consumed materials in the MES batch report.
- **Process Produced Material**
Receive information about material production (aka goods issues) performed on the DCS as part of a specific order.
PharmaSuite applies the corresponding changes to the inventory and documents the produced materials in the MES batch report.
- **Get Batch Information**
Read all relevant information of a batch from the MES.
This is typically done from the DCS to retrieve the batch status and other information relevant to an automatic execution.

The services listed below implement the business functionality of the requests:

- *com.rockwell.mes.services.wip.ifc.dcs.IDCSConsumeMaterialService*
- *com.rockwell.mes.services.wip.ifc.dcs.IDCSProduceMaterialService*
- *com.rockwell.mes.services.wip.ifc.dcs.IDCSInformationService*

To adapt the implementation of the requests, adapt the behavior of the correspondent services with the following steps:

- Create new java classes that extend the existing services.
- Register custom classes (see section "Configuring Service Implementations", chapter "Managing Services" in Volume 2 of the "Technical Manual Configuration and Extension" [A3] (page 171)).
- Create custom JAR files (see section "Libraries" in "Process Designer Online Help" [B1] (page 171)).

TIP

If there is a need to transfer further data that is not contained in the standard, it is possible to add a list of key/value pairs for both directions. The values of the request are available as input parameters of the service method. Values to be sent on the reply message can be added to the response.

Adapting User Documentation

This section contains general information about the authoring platform (page 165) used to compile the PharmaSuite documentation, the generation of help URLs (page 166) in PharmaSuite, and how to adapt the help path (page 169).

Accessing and Configuring the Author-it Database

The PharmaSuite user documentation is compiled with the authoring platform Author-it (library version: 6.35 (build 6.4.2.36311)) from Author-it Software Corporation [D1] (page 172).

For viewing the WebHelp output it must be located on a web server.

TIP

The folder structure described here helps to ensure correct publishing. If you wish to apply a different structure, observe the dependencies between the folders and make sure you have NO BLANKS in your paths, otherwise the after-publish macros for HTML will not function correctly.

To access the Author-it database proceed as follows:

1. Copy the database to C:\AIT.
2. Copy the **Templates**, **Macros**, and the **images_*** folders to C:\AIT.
3. Log in as **System Integrator** without password.

TIP

We recommend to create individual user accounts for all Author-it users. The accounts should have the same access privileges as the **System Integrator** account. You can create user names and passwords according to your policies. Make sure to run the **Update License Wizard** from **Author-it Administrator** to license the database for your use.

Some of the user options refer to the folder structure. They are defined in the **Author-it options**. This applies to the following settings:

- **File Locations** tab, **Templates**: C:\AIT\Templates
- **Publishing** tab, **Publish**: C:\AIT_publishing

The **File** objects are based on corresponding templates. They refer to the folder structure. The following templates for **File** objects are available:

- RA_TemplateIcons, **Directory**: Templates\RA_Customized_Images
- RA_TemplateLinkedPNG, **Directory**: images_PS

Publishing Profiles define the output and refer to templates and macros located on the file system. The following **Publishing Profiles** are available:

- RA_MSWord, **Theme** tab, **Word template**:
Templates\RA_Word\PharmaSuite.dotm
- RA_WebHelp_standalone to publish stand-alone Web Help, **Actions** tab, **After Publishing**: adjusts tables, browser compatibility
 - **Command**: C:\AIT\Macros\RA_HTML\<macro_name>.bat
 - **Argument**: C:\AIT_publishing\<Publishing Profile name>

Actions tab, **After Publishing**: delete *.bak files, move files to web server location
- RA_WebHelp to publish Web Help to be called from the application (with collapsed Contents frame and without header and footer areas), **Actions** tab, **After Publishing**: adjusts tables, adjusts help, browser compatibility
 - **Command**: C:\AIT\Macros\RA_HTML\<macro_name>.bat
 - **Argument**: C:\AIT_publishing\<Publishing Profile name>

Actions tab, **After Publishing**: delete *.bak files, move files to web server location

TIP

Java (JRE 1.5 at least) is required for running the after-publish macros.

In **Author-it Administrator**, the **Archive Folder** is set to C:\AIT\Archive.

Generating Help URLs in PharmaSuite

A help URL consists of three components:

<server>/[<application path>]/<help path>

- <server>
Server to which the PharmaSuite client is connected (e.g. http://localhost:8080, http://<MES-PS-HOST>.8080, where <MES-PS-HOST> is the name of your PharmaSuite server.)

- <application path>
 - For the Production Execution Client, the <application path> is read from the **Form/ApplicationStart_ProductionExecutionClient/onlineHelpPrefix** configuration key. The default value is **/PharmaSuite/documentationandhelp/**.
 - For the Production Management Client, the <application path> is read from the **Form/ApplicationStart_ProductionManagementClient/onlineHelpPrefix** configuration key. The default value is **/PharmaSuite/documentationandhelp/**.
 - For details on configuration keys, see chapter "Configuration Keys of PharmaSuite" in Volume 4 of the "Technical Manual Configuration and Extension" [A4] (page 171).
 - For the Production Response Client, Recipe and Workflow Designer, and Data Manager, the default value (**/PharmaSuite/documentationandhelp/**) is used.
- <help path>
 - For the Production Execution Client, the <help path> is read from the **OnlineHelpLinkage_<activity set name> FactoryTalk ProductionCentre Message** object. The **Message ID** parameters correspond to the activity set steps. The **Message** contains the <help path> itself.
Example: Activity set = **GoodsIssue**, activity set step = **asSelectSublots**,
Message = **pec/index.htm#2844.htm**

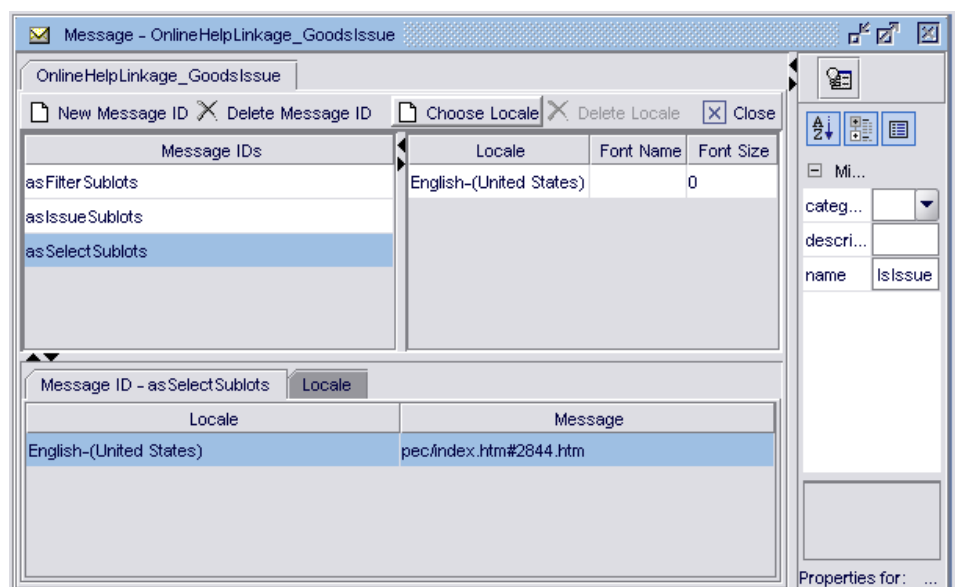


Figure 56: Help path for Production Execution Client

- For the Production Management Client, the <help path> is read from the **OnlineHelpLinkage_ucOrder** FactoryTalk ProductionCentre **Message** object. The **Message ID** parameters correspond to the template identifiers of the selected node. The **Message** contains the <help path> itself. Example: Use case = **ucOrder**, node = **dtUseOrderFilter**, Message = **pmc/index.htm#1739.htm**

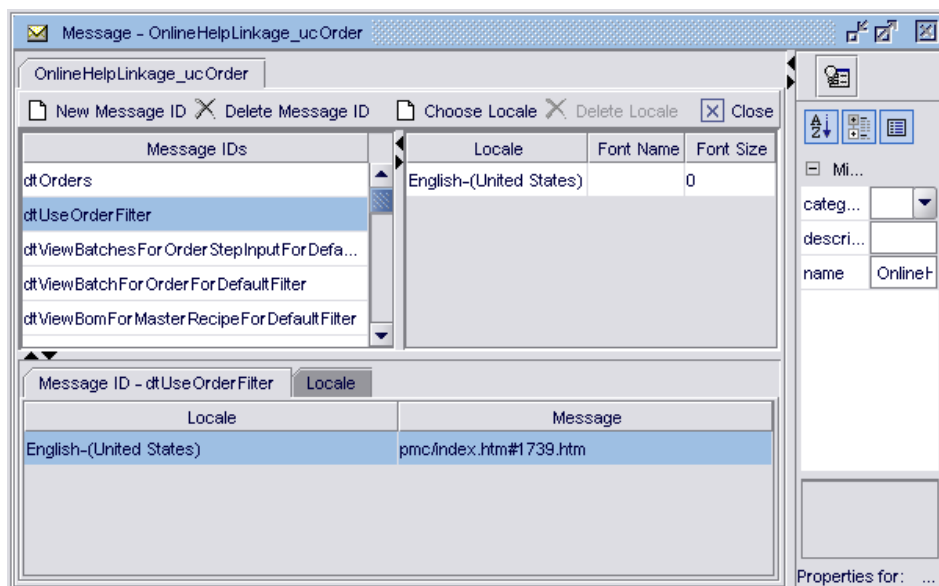


Figure 57: Help path for Production Management Client

- For the Production Response Client, the <help path> is read from the **OnlineHelpLinkage_ui_ExceptionDashboard** FactoryTalk ProductionCentre **Message** object. The **Message ID** parameters correspond to user interface elements. The **Message** contains the <help path> itself. Example: Filter panel, Message = **prc/index.htm#13175.htm**
- For Recipe and Workflow Designer, the <help path> is read from the **OnlineHelpLinkage_ui_RecipeDesinger** FactoryTalk ProductionCentre **Message** object. The **Message ID** parameters correspond to user interface elements and are prefixed with the perspective to which they apply:
 - RD_ for Recipe Designer - Batch
 - DR_ for Recipe Designer - Device
 - WF_ for Workflow Designer
 - ALL_ for common links

The **Message** contains the <help path> itself.

Example: Map, Message = **rwd/index.htm#9005.htm**

- For Data Manager, the <help path> is read from the **OnlineHelpLinkage_ui_MasterDataEditor** FactoryTalk ProductionCentre

Message object. The **Message ID** parameters correspond to user interface elements and are prefixed with the perspective to which they apply:

- EQ_ for Data Manager - Equipment - Smart Search
- EM_ for Data Manager - Equipment - Basic Search
- WC_ for Data Manager - Work Center
- ALL_ for common links

The **Message** contains the <help path> itself.

Example: Search window, Message = **dm/index.htm#24423.htm**

■ Examples:

<http://localhost:8080/PharmaSuite/documentationandhelp/pec/index.htm#2844.htm>

<http://localhost:8080/PharmaSuite/documentationandhelp/pmc/index.htm#1739.htm>

<http://localhost:8080/PharmaSuite/documentationandhelp/prc/index.htm#13175.htm>

<http://localhost:8080/PharmaSuite/documentationandhelp/rwd/index.htm#9005.htm>

<http://localhost:8080/PharmaSuite/documentationandhelp/dm/index.htm#24423.htm>

Adapting the Help Path

The help path must be adapted if a different html file, activity set step/node (in Production Execution Client), or activity set/use case (in Production Management Client) should be referenced.

To adapt the reference to the **html file** in the help path, proceed as follows:

1. In Process Designer, expand the **Messages** node and open the **Message** object to be adapted.
2. Navigate to the **Message ID** parameter of the affected activity set step or node.
3. Adapt the **Message** parameter.
4. Save your changes.

To adapt the reference to the **activity set step** or **node** in the help path of the Production Execution Client, proceed as follows:

1. In Process Designer, expand the **Messages** node and open the **Message** object to be adapted.
2. Add a new **Message ID** parameter referencing the affected activity set step or node (e.g. **x_asSelectSublots**, **x_dtUseOrderFilter**).

3. Add the reference to the html file as **Message** parameter.
4. Save your changes.

To adapt the reference to the **activity set** or **use case** in the help path of the Production Management Client, proceed as follows:

1. In Process Designer, right-click the **Messages** node and select the **New Message** function to add a new **Message** object referencing the affected activity set or use case (e.g. **OnlineHelpLinkage_x_GoodsIssue**, **OnlineHelpLinkage_x_ucOrder**).
2. Add new **Message ID** parameters referencing the activity set steps or nodes.
3. Add the references to the html files as **Message** parameters.
4. Save your changes.

TIP

In Production Execution Client: If you copy and modify an activity set, copy the corresponding OnlineHelpLinkage_<activity set name> message (and modify it if you have renamed activity set steps).

In Production Management Client: If you copy and modify a use case, copy the corresponding OnlineHelpLinkage_<use case name> message (and modify it if you have renamed template identifiers of nodes).

Reference Documents

The following documents are available from the Rockwell Automation Download Site.

No.	Document Title	Part Number
A1	PharmaSuite Technical Manual Developing System Building Blocks	PSBB-PM007E-EN-E
A2	PharmaSuite Technical Manual Configuration & Extension - Volume 1	PSCEV1-GR008E-EN-E
A3	PharmaSuite Technical Manual Configuration & Extension - Volume 2	PSCEV2-GR008E-EN-E
A4	PharmaSuite Technical Manual Configuration & Extension - Volume 4	PSCEV4-GR008E-EN-E
A5	Technical Manual DCS Adapter	DCTMAD-GR001C-EN-E

TIP

To access the Rockwell Automation Download Site, you need to acquire a user account from Rockwell Automation Sales or Support.

The following documents are distributed with the FactoryTalk ProductionCentre installation.

No.	Document Title / Section
B1	Process Designer Online Help

TIP

To access the "Process Designer Online Help", use the following syntax:
<http://<MES-PS-HOST>:8081/PlantOpsDownloads/docs/help/pd/index.htm>, where
 <MES-PS-HOST> is the name of your PharmaSuite server. To view the online help, the Apache Tomcat of the FactoryTalk ProductionCentre installation must be running.

The following documents are distributed with the PharmaSuite installation.

No.	Document Title / Section
C1	PharmaSuite-related Java Documentation: Interfaces of PharmaSuite
C2	Recipe and Workflow Designer User Documentation

No.	Document Title / Section
C3	Data Manager User Documentation

TIP

To access the "PharmaSuite-related Java Documentation", use the following syntax: *http://<MES-PS-HOST>:8080/PharmaSuite/javadoc/*, where <MES-PS-HOST> is the name of your PharmaSuite server.

To access the "Recipe and Workflow Designer User Documentation", use the following syntax: *http://<MES-PS-HOST>:8080/PharmaSuite/documentationandhelp/index.htm*, where <MES-PS-HOST> is the name of your PharmaSuite server.

To access the "Data Manager User Documentation", use the following syntax: *http://<MES-PS-HOST>:8080/PharmaSuite/documentationandhelp/index.htm*, where <MES-PS-HOST> is the name of your PharmaSuite server.

The following third-party documentation is available online as reference:

No.	Document Title / Web Site
D1	Author-it (http://www.author-it.com/)

Revision History

The following table describes the history of this document.

Changes related to the document:

Object	Description	Document
---	---	---

Changes related to "Introduction" (page [1](#)):

Object	Description	Document
---	---	---

Changes related to "Extension and Naming Conventions" (page [7](#)):

Object	Description	Document
---	---	---

Changes related to "Adding Numeric Fields to the Database" (page [11](#)):

Object	Description	Document
---	---	---

Changes related to "Configuring the Universe of Recipe and Workflow Designer" (page [13](#)):

Object	Description	Document
Parameters in the Universe Configuration (page 33)	<i>accessPrivilegesForConfidentialObjects</i> and <i>noDBKey</i> parameters added.	1.0

Changes related to "Configuring the Setlist of Recipe and Workflow Designer" (page [39](#)):

Object	Description	Document
---	---	---

Changes related to "Configuring the Property Windows of Recipe and Workflow Designer" (page [43](#)):

Object	Description	Document
Adjusting the Packaging Level-related Attributes (Header Property Window) (page 47)	New section.	1.0

Changes related to "Configuring the Parameter Panel of Recipe and Workflow Designer" (page [53](#)):

Object	Description	Document
Adjusting the Configuration of Custom Attributes (page 55)	Description for localization: UDA replaced by column and removed from message name. No change of code.	1.0
Adjusting the Packaging Level-related Attributes (page 56)	New section.	1.0

Changes related to "Managing the Layout of Recipe and Workflow Designer and Data Manager" (page [59](#)):

Object	Description	Document
What Is Layout Management? (page 59)	Recipe and Workflow Designer and Data Manager store the user settings in the table of the X_AppLayoutSettings AT definition.	1.0
Where Are the Initial Layouts of Recipe and Workflow Designer and Data Manager Configured?	Section moved to "What Is Layout Management?".	1.0
Where Are the User Preferences of Recipe and Workflow Designer and Data Manager Stored?	Section removed.	1.0
Resetting User-specific Layouts	Section removed.	1.0

Changes related to "Configuring the Expression Editor of Recipe and Workflow Designer and Data Manager" (page 63):

Object	Description	Document
Consider the Equipment-specific Usage List (page 71)	New section.	1.0

Changes related to "Using ERP BOMs in Recipe and Workflow Designer" (page 75):

Object	Description	Document
---	---	---

Changes related to "Configuring the Initialization of Material Parameters in Recipe and Workflow Designer" (page 77):

Object	Description	Document
---	---	---

Changes related to "Configuring Menu and Toolbar of Recipe and Workflow Designer" (page 81):

Object	Description	Document
---	---	---

Changes related to "Configuring the Details Window of Data Manager - Work Center" (page 87):

Object	Description	Document
---	---	---

Changes related to "Configuring the Change History of Data Manager - Work Center" (page 91):

Object	Description	Document
---	---	---

Changes related to "Configuring FSMs for Equipment Properties" (page [95](#)):

Object	Description	Document
---	---	---

Changes related to "Providing Images for Equipment Classes in Data Manager" (page [99](#)):

Object	Description	Document
---	---	---

Changes related to "Providing Report Designs for (Template) Equipment Entities in Data Manager" (page [101](#)):

Object	Description	Document
---	---	---

Changes related to "Adding an Equipment Logbook Category Accessible by Phase Building Blocks" (page [103](#)):

Object	Description	Document
---	---	---

Changes related to "Retrieving Specific Information from the Equipment Logbook" (page [107](#)):

Object	Description	Document
---	---	---

Changes related to "Configuring Additional Purposes for the Equipment Type Technical Property Type" (page [113](#)):

Object	Description	Document
---	---	---

Changes related to "Adding a Workflow Type to Workflow Designer and the Production Execution Client" (page [115](#)):

Object	Description	Document
---	---	---

Changes related to "Managing Cockpit Actions of the Production Execution Client" (page [117](#)):

Object	Description	Document
Running Processes in the Cockpit	Name section of top level updated.	1.0

Changes related to "Configuring Menu and Toolbar of the Production Response Client" (page [137](#)):

Object	Description	Document
---	---	---

Changes related to Adding Filter Attributes to the Exception Dashboard of the Production Response Client (page [141](#)):

Object	Description	Document
---	---	---

Changes related to "Material Handling for DCS" (page [163](#)):

Object	Description	Document
Material Handling for DCS (page 163)	New chapter.	1.0

Changes related to "Adapting User Documentation" (page [165](#)):

Object	Description	Document
---	---	---

A

- Action class • 117
 - Configuring (Production Execution Client) • 123
 - Configuring (Production Response Client) • 138
 - Configuring (Recipe and Workflow Designer) • 82
 - Implementing (Production Execution Client) • 118
 - Implementing (Production Response Client) • 138
 - Implementing (Recipe and Workflow Designer) • 82
- Adapting
 - Help path • 169
 - User documentation • 165
- Audience • 4
- Author-it • 165

C

- Change history (Data Manager - Work Center)
 - Adjusting configuration • 92
 - Adjusting configuration of custom attributes • 92
 - Tracked events • 91
- Cockpit action
 - Access privilege • 126
 - ChangeStationAction • 128
 - Configuring • 124
 - Displaying a dialog • 119
 - Displaying a form • 119
 - Displaying a panel • 120
 - Filtering • 126
 - Invisible action • 129
 - Managing • 117
 - Optional action • 128
 - RefreshStartableOperationsAction • 128
 - Restricted access rights • 125
 - Running processes • 133
 - Scanner action • 129
 - Scrolling • 124
 - Without user interface • 118
- Context-related function (Expression editor) • 64

- Conventions • 7
- Conventions (typographical) • 5

D

- Data Manager • 59
 - Context menu • 60
 - Floating window • 60
 - Layout • 59
 - Layout management • 59
 - mdt_buttonConfiguration • 59
 - mdt_defaultLayoutSettings • 59
 - Menu bar • 60
 - Toolbar • 60
- DCS • 163
- Details window (Data Manager - Work Center)
 - Adjusting configuration • 88
 - Adjusting configuration of custom attributes • 89
- Download Site • 171

E

- Equipment class (Data Manager) • 99
 - Image • 99
- Equipment entity (Data Manager) • 101
 - Report design • 101
- Equipment logbook (specific information) • 107
- Equipment logbook category • 103
- Equipment property
 - FSM • 95
 - Semantic properties • 96
- Equipment type • 113
 - Purpose • 113
- ERP BOM • 75
- Expression editor
 - Configuring • 63
 - Context-related function (in Recipe and Workflow Designer) • 64
 - Function (arbitrary) • 68
- Extension conventions • 7

F

Function (arbitrary, Expression editor) • 68

H

Help URL • 166

I

IMaterialParameterCustomAttributesHandler • 77

Initialize material parameter • 77

Intended audience • 4

L

Layout management • 59

M

Material handling for DCS • 163

Material parameter (initialize) • 77

N

Naming conventions • 7

Numeric fields • 11

O

Open dialog (Universe) • 17

P

Parameter Panel (Recipe and Workflow Designer)

Adjusting configuration • 54

Adjusting configuration of custom attributes • 55

Adjusting packaging level-related attributes • 56

Configuring • 53

Production Response Client action

Configuring • 139

Production Response Client filter attribute

Adding • 141

Boolean data type • 158

Choice list data type • 156

Common methods • 148

Configuring • 162

Data type • 144

Database object • 142

Date data type • 160

Implementing • 147

Option list data type • 152

String data type • 148

String list data type • 150

Property window (Recipe and Workflow Designer)

Adjusting configuration • 44

Adjusting configuration of custom attributes (header, element) • 45

Adjusting configuration of custom attributes (source) • 49

Adjusting packaging level-related attributes (header) • 47

Configuring • 43

Purpose (equipment type) • 113

R

Recipe and Workflow Designer • 59

Context menu • 60

Floating window • 60

Layout • 59

Layout management • 59

Menu bar • 60

red_buttonConfiguration • 59

red_defaultLayoutSettings • 59

Toolbar • 60

Recipe and Workflow Designer action

Configuring • 84

Context • 85

Reference documents • 171

Rockwell Automation Download Site • 171

S

Select material dialog (Universe) • 17

Setlist (Recipe and Workflow Designer)

Adjusting configuration • 40

Adjusting configuration of custom attributes • 41

Configuring • 39

T

Template equipment entity (Data Manager) • 101

Report design • 101

U

Universe (Recipe and Workflow Designer)

Adjusting configuration • 35

Configuring • 13

Explorer • 14

Object • 13

Object type • 13

Tips and tricks • 36

User documentation • 165

W

Workflow

Type • 115