LISTEN.
THINK.
SOLVE.®

PharmaSuite®

DEVELOPING SYSTEM BUILDING BLOCKS

RELEASE 8.4

TECHNICAL MANUAL

PUBLICATION PSBB-PM007E-EN-E-DECEMBER-2017
Supersedes publication PSBB-PM007D-EN-E

Allen-Bradley • Rockwell Software

Rockwell Automation

Contents

Figures

Chapter 1

Introduction

This documentation contains important information about how to create system building blocks for specific use cases. The current version of PharmaSuite only supports the creation of phase building blocks.

The information is structured in the following sections:

General Information

In this part you will find general information about the concepts behind building blocks (page 9) and their execution implemented with PharmaSuite and an introduction to master recipes (page 9), master workflows (page 11), phases (page 12) and their parameter classes (page 13), process parameters (page 14), material parameters (page 16), equipment requirement parameters (page 16), privilege parameters (page 16), capabilities (page 17), exceptions (page 17), and actions (page 18).

Creation and Handling of Phase Building Blocks

In this part you will find information about the tools used for the creation of phase building blocks (e.g. phase generator (page 19), parameter class generator (page 23), phase copy tool (page 25)) and the underlying naming conventions (page 5). You will also learn how to

- create phase building blocks (page 33),
- create UI extensions for phase building blocks (page 91),
- add process parameters to phase building blocks (page 99),
- use material parameters in phase building blocks (page 113),
- use equipment requirement parameters in phase building blocks (page 117),
- handle S88 equipment-related feature (page 119),
- add exception handling for phase-internal checks (page 135) and recurring irregular situations (page 143),
- add actions for completed phases (page 153),
- add phase-specific sub-reports (page 159),
- add pause-aware support (page 161),
- respond to the Change User and Register at Station actions (page 163),
- extend product phases and parameter classes (page 165),

- define a mapping between parameter bean attributes and AT definition columns (page 167),
- create a new version of a phase and their related parameter classes (page 169), and
- adapt the behavior of existing phase building blocks (page 171).

Appendix

In this part you will find

- tips and tricks related to phase development (page 181),
- information related to a sandbox for phases to manage unhandled situations (page 183), and
- sample phase building blocks such as HelloWorld - text output (page 197) and GetValue - store data (page 198).

Finally, the Reference Documents (page 175) section provides a list of all the documentation that is referenced in this manual.

Intended Audience

This manual is intended for system engineers and phase developers who maintain phase building blocks to be used with PharmaSuite.

They need to have a thorough working knowledge of FactoryTalk® ProductionCentre and PharmaSuite. Thus we highly recommend to participate in a FactoryTalk ProductionCentre and PharmaSuite training before starting on the tasks described in this manual.

In order to maintain phase building blocks, you need 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 - window and dialog titles - menu functions - panel, tab, and button names - box labels - object properties and their values (e.g. status).

Italic typeface	Designates technical background information, such as ■ path, folder, and file names ■ methods ■ classes.
CAPITALS	Designate keyboard-related information, such as ■ key names ■ keyboard shortcuts.
`Monospaced typeface`	Designates code examples.

Chapter 2

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 this manual.
- 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 5).

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 175)).
   - 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" [A3] (page 175)).
   - 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 7).

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 5), 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.

What Is a Building Block?

PharmaSuite is based on S88 methodology. S88 (aka ANSI/ISA-88) is an industry standard for batch process control. The standard defines reference models for batch control as used in the process industries and terminology that helps explain the relationships between those models and terms.

According to S88, PharmaSuite provides four hierarchy levels of procedural elements, which are also referred to as building blocks: A **procedure** consists of **unit procedures**; each unit procedure consists of **operations**, and each operation consists of **phases**. Phases are atomic.
The building blocks within procedures, unit procedures, and operations are connected by means of Sequential Function Charts (SFC).

Additionally, on all levels there are parameters available for each building block:

- Process parameters
  They describe process-relevant data like a mixer speed or an allowed temperature.
- Process inputs and outputs
  They describe resources used or produced by one recipe entity. Resources are, for example, materials, equipment, privileges, etc.

Building blocks are stored in the building block library. They are used to build master recipes (page 9) and master workflows (page 11) in Recipe and Workflow Designer. In the Production Execution Client, master recipes are executed as control recipes and master workflows as workflows. All user operations and data collection is performed when phases are executed.

What Is a Master Recipe?

In PharmaSuite, a master recipe is an implementation of an S88-based recipe for the pharmaceutical batch production. It consists of:

- procedural elements according to S88 (procedures, unit procedures, operations and phases),
- a set of parameters (process parameters, material parameters, equipment requirement parameters),
- a set of data describing the material flow within a recipe (MFC data).

TIP

Bills of materials and routes still exist, but are only used internally and are not accessible from the user interface of PharmaSuite.

A master recipe is built by using phases and instantiating them in an operation. The following example of a master recipe

- has three phases in one operation,
- is built from two phases, one reading a value and one reading a signature,
    - with different parameters, and
    - a repeated first phase.

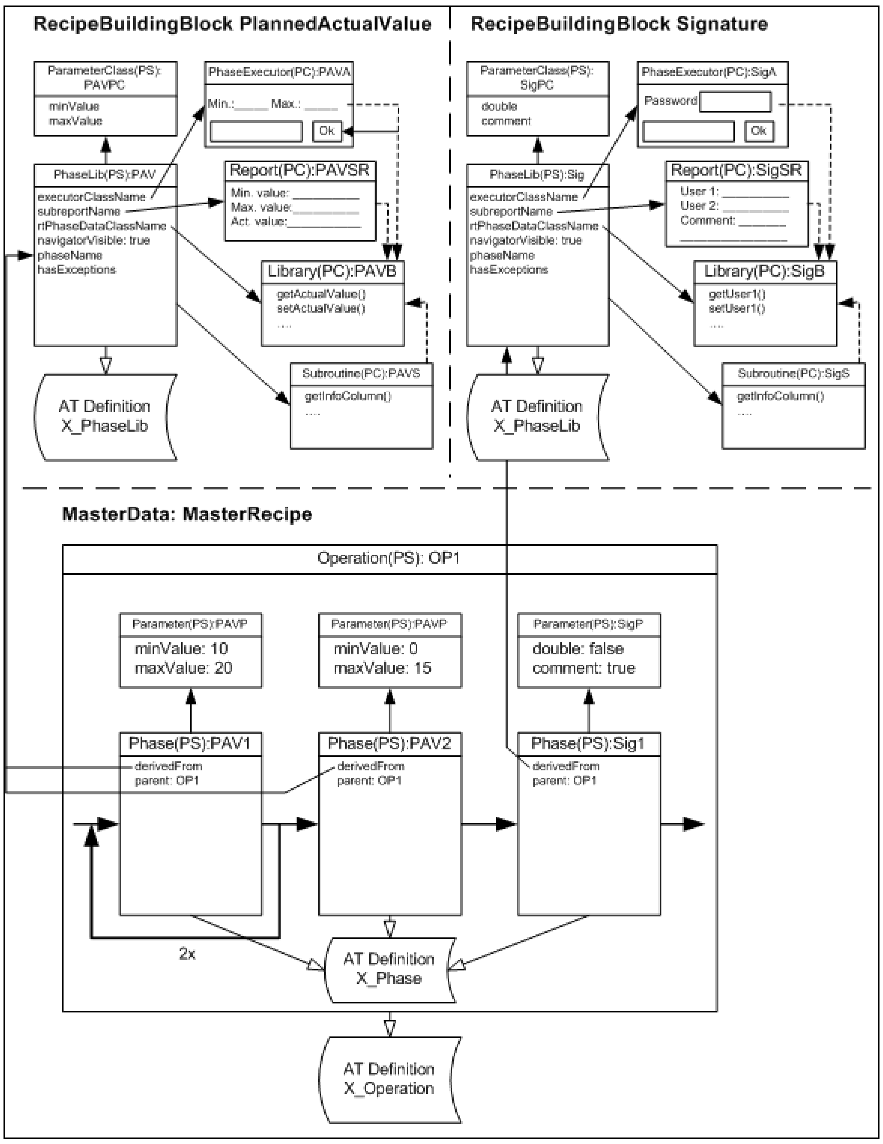


Figure 1: Example of a master recipe

During execution, the system creates a runtime operation for each operation and a runtime phase for each phase of the master recipe. If a phase is repeated in a loop, the system creates a new runtime phase for each repetition. Along with the creation of the runtime phase, the system increases the instance count.

The example shows the situation after the example has been executed completely:

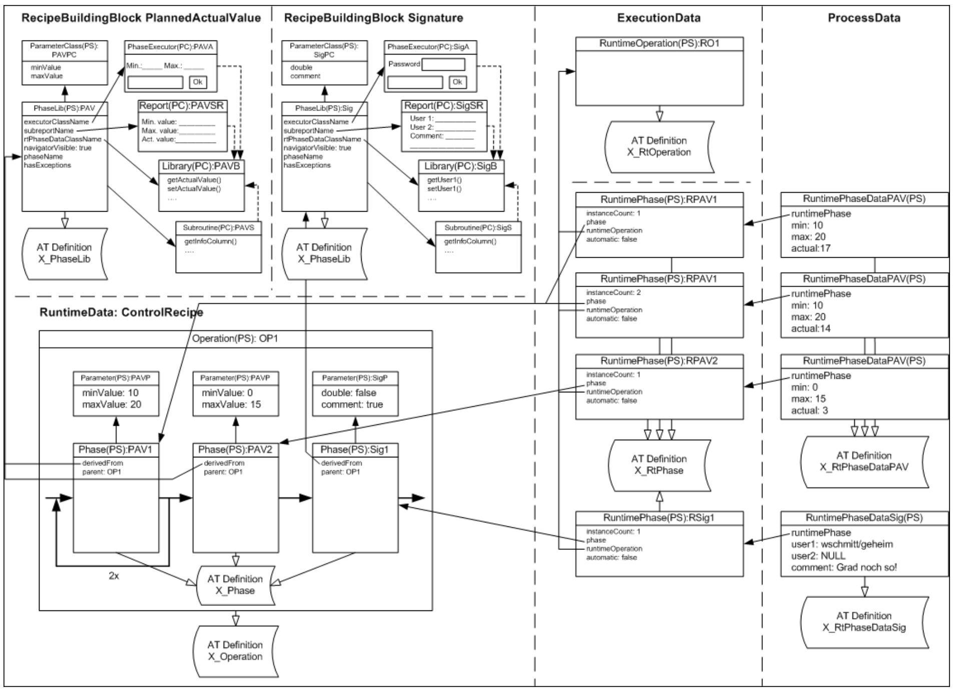


Figure 2: Example of recipe execution

What Is a Master Workflow?

In Workflow Designer of PharmaSuite, a master workflow defines the execution steps of operations, which are often intended for multiple executions and are not intended to produce an end product. Examples are cleaning processes or inventory management operations. It is structurally similar to a master recipe but only allows one element on the unit procedure level. It consists of:

- procedural elements according to S88 (procedures, unit procedures, operations and phases),
- a set of parameters (process parameters, (rarely used) material parameters, equipment requirement parameters),
- a (restricted) set of data describing the material flow within a workflow (MFC data).

A master workflow can have only one procedure and one unit procedure.

A workflow is typically intended for multiple executions and has a non-production purpose such as cleaning.

Phase Building Block-related Objects

The following sections describe phase building blocks and related objects in more detail. All of them are relevant to master recipes and master workflows.

What Is a Phase?

A phase is a reusable building block that defines the behavior of a process-oriented task. Its behavior can be configured during master recipe or master workflow creation, respectively. All phase entries together form the **Library** of components that can be used for phases.

Depending on a phase's properties, it can be usable in the context of master recipes, of master workflows, or in the context of both.

Each phase is constructed from a phase building block that is stored in a library and instantiated in the phases of a master recipe or master workflow. Phases are stored in the **X_PhaseLib** application table.

A phase consists of the following components, which have to be created by a phase designer when adding a new phase to the system:

- RuntimePhaseExecutor
  This class is a piece of Java code and runs during the execution of a phase in the Production Execution Client. The class may also contain a GUI if the phase should be displayed during processing. It can store and retrieve *Runtime phase data* from the *RuntimePhase* and store the *Runtime phase output* usable by other phases.
- Runtime phase data (optional)
  Because of the diverse nature of phases, each phase has its own set of data. The phase-specific runtime data application table and corresponding classes and interfaces are used to store phase-specific data collected at runtime.
  For information on application tables, please refer to the FactoryTalk ProductionCentre documentation. In particular, see section "Application Table (AT) Definitions" in "Process Designer Online Help" [B1] (page 175).
- Runtime phase output (optional)
  Each phase can define its specific output. An output consists of one or more output values. The data is stored in a phase-specific application table (analogous to the runtime data application table). Similar to the runtime phase data, there are Java interfaces and classes to store and access the output values at runtime.

- Subroutine (optional)
  This object provides methods for customer-specific implementations, i.e. for the Navigator's information column string.
- Parameters (optional)
  A definition of parameters that define the interface of a phase (e.g. material or specification of an allowed range for an actual value, a text to be displayed during execution, a signature required for recording exceptions). For details, see sections "What Is a Process Parameter?" (page 14), "What Is a Material Parameter?" (page 16), "What Is an Equipment Requirement Parameter?" (page 16), and "What Is a Privilege Parameter?" (page 16).
- Sub-report (optional)
  The definition of the layout and the displayed information of the phase in the batch report. The sub-report is also used to display detailed information of a phase and is accessible via the Navigator's information column.

Thus an example of a phase that reads a value expected to be within a given range may look like this:

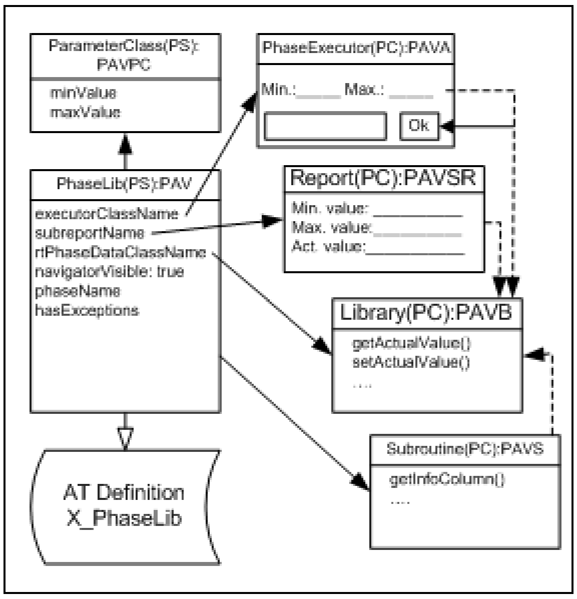


Figure 3: Example of a phase

What Is a Parameter Class?

Parameter classes are used to define the types of parameters a building block expects in its interface. Technically, a parameter class is mainly a tuple that consists of a set of attributes. The attributes can be Java types or any FactoryTalk ProductionCentre objects. All phases (page 12) have to use parameter classes to specify their interface, so a phase holds a set of parameter classes. When a parameter class is used to create a specific parameter instance (see section "What Is a Process Parameter?" (page 14)) it holds the specific parameters used in the master recipe or master workflow. The attributes of the parameter classes have to be filled in Recipe and Workflow Designer once a phase has been instantiated there.

Since some parameter attributes are directly related with each other, e.g. the minimum and maximum value of a range, they are grouped in a parameter class instead of one forming a separate parameter class by itself.

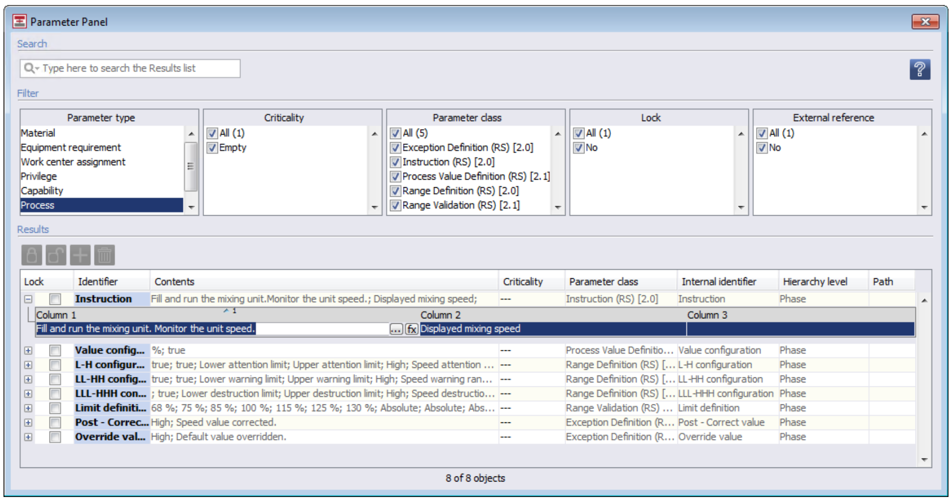
What Is a Process Parameter?

Process parameters can be defined on all levels of a master recipe or master workflow and are used to configure all of their entities by specifying their necessary input.

Examples of process parameters are:

- Instruction text displayed in a phase.
- Pre-defined mixer speed.
- Allowed temperature range for a temperature value used during production.

When a phase is inserted in a master recipe or master workflow, the system instantiates a process parameter (also referred to as process parameter instance) for each parameter defined in the interface of the phase. The process parameter holds the parameter values used in a specific master recipe or master workflow and can be accessed for editing in Recipe and Workflow Designer.



Parameter type	Criticality	Parameter class	Lock	External reference
Material	✓ All (1)	✓ All (5)	✓ All (1)	✓ All (1)
Equipment requirement	✓ Empty	✓ Exception Definition (RS) [2.0]	✓ No	✓ No
Work center assignment		✓ Instruction (RS) [2.0]		
Privilege		✓ Process Value Definition (RS) [2.1]		
Capability		✓ Range Definition (RS) [2.0]		
Process		✓ Range Validation (RS) [2.1]		

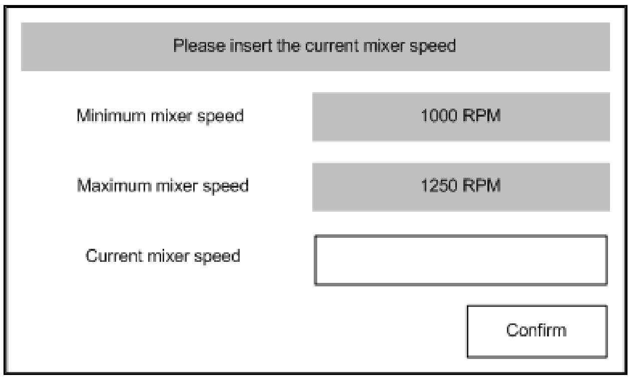
Lock	Identifier	Contents	Criticality	Parameter class	Internal identifier	Hierarchy level	Path
☐	**Instruction**	Fill and run the mixing unit.Monitor the unit speed.; Displayed mixing speed;	---	Instruction (RS) [2.0]	Instruction	Phase	
☐	**Value config...**	%; true	---	Process Value Definitio...	Value configuration	Phase	
☐	**L-H configur...**	true; true; Lower attention limit; Upper attention limit; High; Speed attention ...	---	Range Definition (RS) [...	L-H configuration	Phase	
☐	**LL-HH config...**	true; true; Lower warning limit; Upper warning limit; High; Speed warning ran...	---	Range Definition (RS) [...	LL-HH configuration	Phase	
☐	**LLL-HHH con...**	; true; Lower destruction limit; Upper destruction limit; High; Speed destructio...	---	Range Definition (RS) [...	LLL-HHH configuration	Phase	
☐	**Limit definiti...**	68 %; 75 %; 85 %; 100 %; 115 %; 125 %; 130 %; Absolute; Absolute; Abs...	---	Range Validation (RS) ...	Limit definition	Phase	
☐	**Post - Correc...**	High; Speed value corrected.	---	Exception Definition (R...	Post - Correct value	Phase	
☐	**Override val...**	High; Default value overridden.	---	Exception Definition (R...	Override value	Phase	

Column 1	Column 2	Column 3
Fill and run the mixing unit. Monitor the unit speed.	Displayed mixing speed	

8 of 8 objects

Figure 4: Editing values of a process parameter

In the following example, a phase uses process parameters to display an instruction text and to check if the provided mixer speed is within a given range (boxes with a gray background are filled using process parameters).


Please insert the current mixer speed

Minimum mixer speed 1000 RPM

Maximum mixer speed 1250 RPM

Current mixer speed

Figure 5: Example of a phase with parameters

The library and the associated master recipe look as follows:

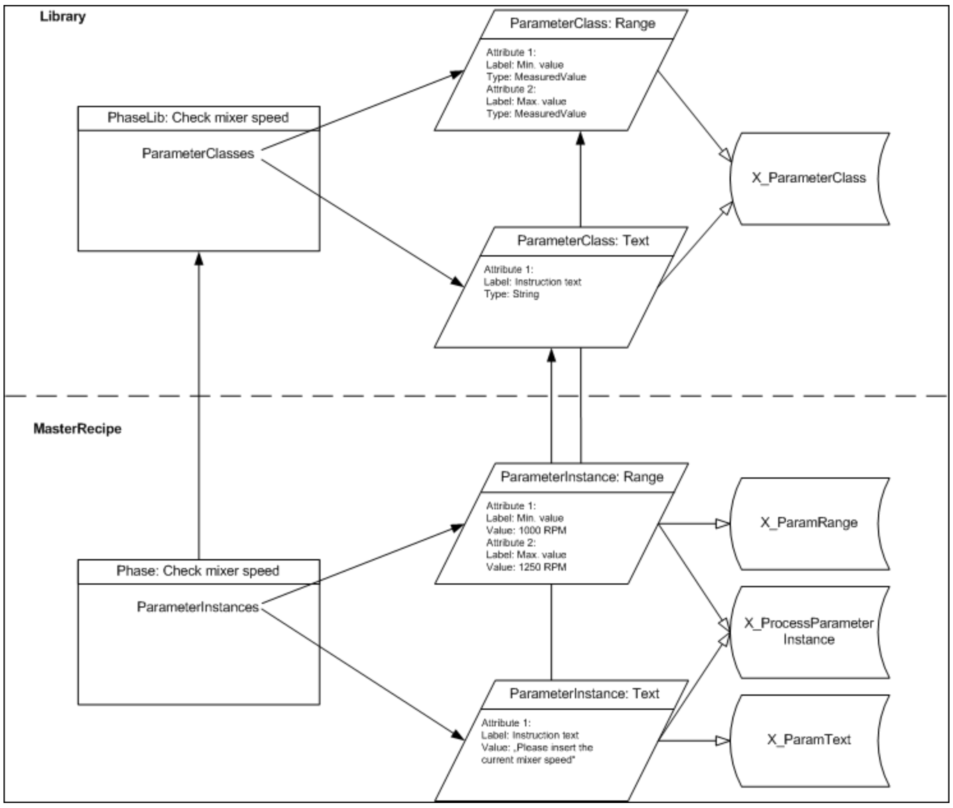


Figure 6: Parameter classes and instances of a phase with two process parameters

TIP

The **Current mixer speed** text box in the example phase is no parameter; it is the value provided by an operator at recipe execution.

To avoid having hundreds of parameter classes, they are reusable, i.e. one parameter class can be used by many phases:

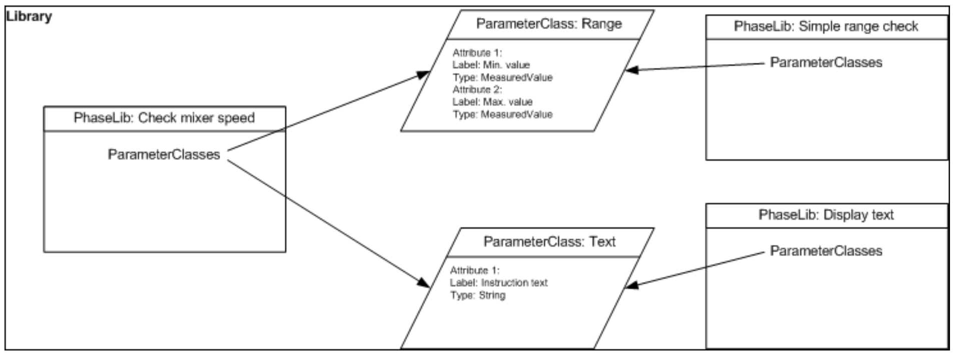


Figure 7: Parameter classes are reusable

What Is a Material Parameter?

A material parameter defines which materials are required for the execution of a phase, as process input and as process output. In Recipe and Workflow Designer, you can add the material parameters as process inputs and process outputs to a phase. However, it is in the responsibility of the phase itself which operations or checks are performed with the specified materials during execution. The phase developer specifies the number of expected material parameters.

What Is an Equipment Requirement Parameter?

An equipment requirement parameter defines the capabilities an equipment entity must provide for the execution. For example, if an equipment class is assigned as an equipment requirement parameter to a phase, an equipment entity used during execution must be a member of the class.

In Recipe and Workflow Designer, you can add equipment requirement parameters to a phase. However, it is in the responsibility of the phase itself to check if used equipment entities fulfill the equipment requirements during execution. The phase developer specifies the number of expected equipment requirement parameters.

What Is a Privilege Parameter?

A privilege parameter is used to define an access privilege representing a signature. During phase execution, a signature will be requested in the following cases:

- Recording a phase-specific exception of a specified risk category (see also section "What Is an Exception?" (page 17)).
- Completing a phase. If defined, a phase is only able to perform its completion function if the signature for phase completion has been given.

TIP

The signature must be added as an access privilege in Process Designer.

What Is a Capability?

Capability parameters define if certain behaviors or capabilities are available during recipe and workflow execution.
Example: If a phase supports pause-aware, it can interpret a pause signal it receives from its unit procedure and is thus suitable for use in a unit procedure that holds the **Pause-enabled** capability.

What Is an Exception?

Exceptions are deviations from the specified process. When executing a phase in a pharmaceutical environment each deviation has to be recorded as an exception for review and risk assessment purposes. Exceptions are an important element of the batch record.

PharmaSuite provides the following exception categories:

- A **user-defined exception** can be recorded by a user at any time for an active or a completed phase. Exceptions of this category are free-text exceptions and cover unexpected exceptions (e.g. someone enters a production facility without GxP-conforming clothing).
  User-defined exceptions do not have to be provided by a phase since they are already supported by the framework.
- A **user-triggered exception** can be recorded by a user for an active phase. Exceptions of this category cover expected, pre-defined exception options that are provided by the phase (e.g. the "Identify material" phase could provide the "manual identification" exception to cover an incident when a scanner is out of order or a label is illegible).
  User-triggered exceptions must be provided by a phase.
- A **system-triggered exception** is based on the data recorded during phase execution. The phase will prompt the user to record the exception. Exceptions of this category cover expected, pre-defined exceptions that are provided by the phase (e.g. a limit violation of a requested value entry during execution).
  System-triggered exceptions must be provided by a phase.
- A **post-completion exception** can be recorded by a user for a completed phase. Exceptions of this category cover expected, pre-defined exceptions that are provided by the phase (e.g. label reprint). See also section "What Is an Action?" (page 18).
  Post-completion exceptions must be provided by a phase.

In PharmaSuite, exceptions are linked to phases. There may be several exceptions per phase. This also applies to operations and unit procedures.

What Is an Action?

Actions are accessible via the action column of the Navigator. They are related to post-completion exceptions (page 17) that can be recorded for completed phases. Thus, actions cover irregular situations that have to be handled after the phase has been completed (e.g. reprint of a label, correction of a recorded value).

Chapter 4

Tools for Creating Phase Building Blocks

PharmaSuite provides several tools that support creating phase building blocks. These are:

- Phase generator (page 19)
- Parameter class generator (page 23)
- Phase Copy Tool (page 25)

What Is the Phase Generator?

This section describes how to create the skeleton of a phase building block with the phase generator (see also "Creating a Phase Building Block with the Phase Generator" (page 33)). Phases based on such a building block can be used in Recipe Designer as a recipe element, in Workflow Designer as a workflow element, and eventually executed in the Production Execution Client or on the Operation Execution server.

The phase generator creates all artifacts that are necessary to implement a phase building block, which leaves only a small amount of further work to be done. The generator provides the following artifacts:

APPLICATION TABLE ENTRY FOR PHASE BUILDING BLOCKS

An entry in the **X_PhaseLib** application table is generated. All usable phase building blocks must be registered in this application table. For any existing entry with the same name, the phase generator updates the existing phase building block artifacts.

PARAMETER CLASS ASSIGNMENTS

All parameter class assignments for the phase building block as described in the corresponding phase building block description are generated. For any existing parameter class assignments, the phase generator updates their assignments.

JAVA FILES

The required Java files are generated:

- *MESRtPhaseData<BasePhaseName>.java*
  - Runtime phase data bean.
  - Can be extended if necessary.
  - Only present if the phase has specific runtime data.

- *MESGeneratedRtPhaseData<BasePhaseName>.java*
  - Base class of *MESRtPhaseData<BasePhaseName>*.
  - Contains all getters and setters for the persistent runtime phase data.
  - Should remain unchanged in case a re-generation of the phase building block is necessary (e.g. if the application table of the runtime phase data was extended by a new column).
  - Only present if the phase has specific runtime data.
- *MESRtPhaseData<BasePhaseName>Filter.java*
  - Runtime phase data bean filter.
  - Can be extended if necessary.
  - Only present if the phase has specific runtime data.
- *MESGeneratedRtPhaseData<BasePhaseName>Filter.java*
  - Base class of *MESRtPhaseData<BasePhaseName>Filter*.
  - Contains the declaration of all standard methods to set filter criteria.
  - Should remain unchanged in case a re-generation of the phase building block is necessary (e.g. if the application table of the runtime phase data was extended by a new column).
  - Only present if the phase has specific runtime data.
- *MESRtPhaseOutput<BasePhaseName>.java*
  - Phase output data bean.
  - Can be extended if necessary.
  - Only present if the phase has specific output.
- *MESGeneratedRtPhaseOutput<BasePhaseName>.java*
  - Base class of *MESRtPhaseOutput<BasePhaseName>*.
  - Contains all getters and setters for the persistent output data.
  - Should remain unchanged in case a re-generation of the phase building block is necessary (e.g. if the application table of the phase output was extended by a new column).
  - Only present if the phase has specific output.
- *MESRtPhaseOutput<BasePhaseName>Filter.java*
  - Phase output data bean filter.
  - Can be extended if necessary.
  - Only present if the phase has specific output.

- *MESGeneratedRtPhaseOutput<BasePhaseName>Filter.java*
  - Base class of *MESRtPhaseOutput<BasePhaseName>Filter*.
  - Contains the declaration of all standard methods to set filter criteria.
  - Should remain unchanged in case a re-generation of the phase building block is necessary (e.g. if the application table of the phase output was extended by a new column).
  - Only present if the phase has specific output.
- *RtPhaseExecutor<BasePhaseName>.java*
  - Skeleton implementation of (client-side or server-side) runtime phase executor.

The default client-side phase building block is a Swing-based phase building block. This means that the GUI of a phase is created using Swing components. The signature of the corresponding methods uses Swing components (JComponents). The phase generator creates a Swing-based runtime phase executor.

If you need a client-side runtime phase executor using **FactoryTalk ProductionCentre controls in the GUI** of the phase, you have to adapt the generated skeleton as follows:

1. Change the base class of your runtime phase executor from *AbstractPhaseExecutorSwing* to *AbstractPhaseExecutor*.
2. Replace the signature of the methods listed below appropriately (see *AbstractPhaseExecutor* base class):

Swing	FactoryTalk ProductionCentre
createPhaseComponent()	*createPhaseCControl()*
createPhaseActionComponent()	*createPhaseActionCControl()*
createPhaseExceptionComponent()	*createPhaseExceptionCControl()*

3. Implement the GUI of these methods using FactoryTalk ProductionCentre controls (see "Using FactoryTalk ProductionCentre Controls instead of Swing Components" (page 67)).

If you need a runtime phase executor, **which runs on the PharmaSuite OE server and therefore has no GUI**, you have to set the *IsServerRunPhase* flag in the XML description of the phase (see section "Step 1: Creating an XML Description of the Phase Building Block" (page 33) in chapter "Creating a Phase Building Block" (page 33)). For more information about server-side phases, please see section "Specific Information on Phases Running on the PharmaSuite OE Server" (page 69) in "Step 5: Creating a Phase Executor (Java Code)" (page 49) in chapter "Creating a Phase Building Block" (page 33).

APPLICATION TABLE FOR RUNTIME PHASE DATA

If necessary, the application table that holds the phase-specific runtime phase data is generated including Java access bean code (see above) in the *ATDefinition* subdirectory of the output directory. If the application table already exists and there are no objections, the phase generator updates the application table. For details, see "Critical Changes" (page 48).

APPLICATION TABLE FOR RUNTIME PHASE OUTPUT DATA

If necessary, the application table that holds the phase-specific output is generated including Java access bean code (see above) in the *ATDefinition* subdirectory of the output directory. If the application table already exists and there are no objections, the phase generator updates the application table. For details, see "Critical Changes" (page 48).

AT ROWS IN XML FORMAT

The following AT rows are generated in XML format in subdirectories of the *<output directory>\AtRow* directory. They are important for deploying a phase. The ATRow entries are stored in files named after the ATRow GUID (e.g. *{78BC6EB4-DF7A-26FE-7E8E-E11BFCAC5F23}.xml*).

- *ATRow\X_PhaseLib* subdirectory
  Contains the application table entry with the main phase building block properties.
- *ATRow\X_BuildingBlock2ParamClass* subdirectory
  Contains the parameter class assignments of the phase, if present.

TIP

The terms **phase** and **phase building block** (i.e. a building block for a phase) are sometimes used synonymously if their particular meaning is clear from the context.

SUB-REPORT ARTIFACTS

If a report design name is specified, an empty report design is generated in the database and exported in two formats:

- as XML file into the *ReportDesign* subdirectory of the output directory and
- as JRXML file into the output directory.
  The file is not necessary as an artifact, since the XML file contains the information. However, it is added for convenience.

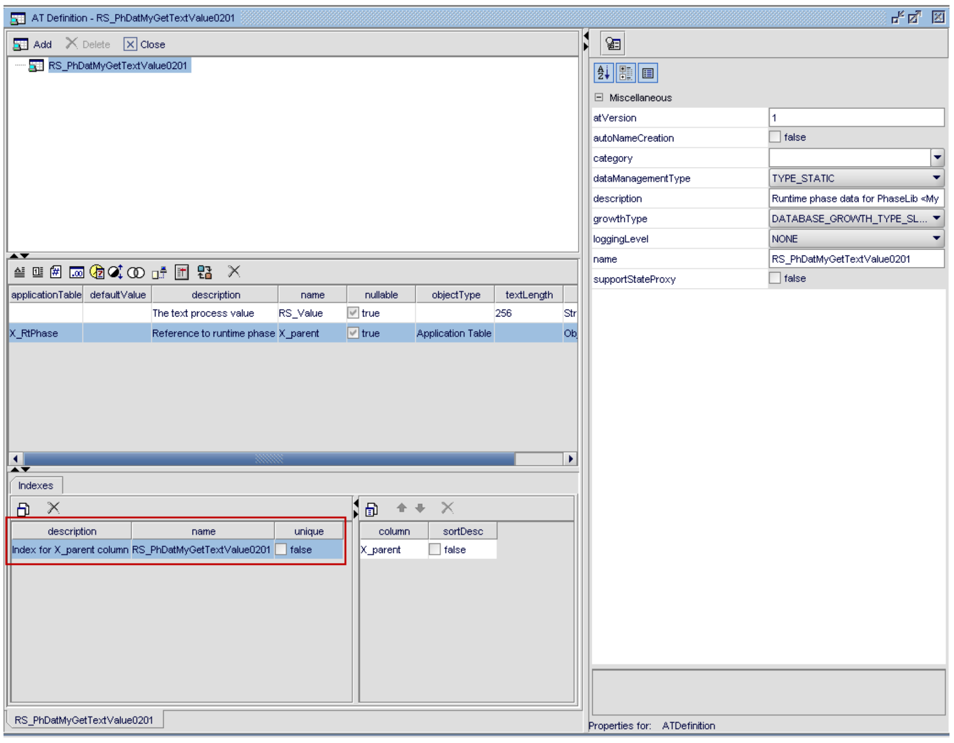
If a report design with the specified name already exists, it is neither overwritten nor exported.

Performance during Execution of Phase Building Blocks

If you experience poor performance during recipe or workflow execution, consider to introduce new indexes to accelerate data retrieval.

If the phase building blocks have not been installed yet, run the phase generator of PharmaSuite again to adapt your phases.

If the phase building blocks have already been installed, adapt the AT definition in Process Designer manually. For both **phase data** and **phase output** AT definitions, add a new non-unique index to the **X_parent** column. The name of the index should be identical to the AT definition itself to avoid migration issues.



AT Definition - RS_PhDatMyGetTextValue0201
Add
Delete
Close
RS_PhDatMyGetTextValue0201
Indexes
RS_PhDatMyGetTextValue0201
Properties for: ATDefinition

applicationTable	defaultValue	description	name	nullable	objectType	textLength	
		The text process value	RS_Value	✓ true		256	Str
X_RtPhase		Reference to runtime phase	X_parent	✓ true	Application Table		Ob

Miscellaneous

atVersion	1
autoNameCreation	false
category	
dataManagementType	TYPE_STATIC
description	Runtime phase data for PhaseLib <My
growthType	DATABASE_GROWTH_TYPE_SL...
loggingLevel	NONE
name	RS_PhDatMyGetTextValue0201
supportStateProxy	false

description	name	unique
Index for X_parent column	RS_PhDatMyGetTextValue0201	false

column	sortDesc
X_parent	false

Figure 8: Adapt phase data AT definition example

What Is the Parameter Class Generator?

This section describes how to create the skeleton of a parameter class with the parameter class generator (see also "Creating and Using Process Parameters" (page 99)).

The parameter class generator creates all artifacts that are necessary to implement a parameter class, which leaves only a small amount of further work to be done. The generator provides the following artifacts:

PARAMETER CLASS ENTRY

The corresponding entry in the **X_ParameterClass** application table is generated. All usable parameter classes must be registered in this application table. For any existing entry with the same name, the parameter class generator updates the entry.

APPLICATION TABLE

The application tables required for the specific parameter class attributes are generated in the *ATDefinition* subdirectory of the output directory. If an application table already exists and there are no objections, the parameter class generator updates the application table. For details, see "Critical Changes" (page 104).
The parameter class generator generates two application tables with identical columns; one table is responsible for the parameters of the master recipe or master workflow at design time, the other can be used for the parameters at runtime.

JAVA FILES

The required Java files are generated:

- *MESParam<ParamClassBaseName>.java*
  - Access bean.
  - Can be extended if necessary.
- *MESGeneratedParam<ParamClassBaseName>.java*
  - Base class of *MESParam<ParamClassBaseName>*.
  - Contains all getters and setters for the parameter instance data.
  - Should remain unchanged in case a re-generation of the parameter class is necessary (e.g. if the application table of the specific parameter class attributes was extended for a new column).
- *MESParam<ParamClassBaseName>Filter.java*
  - Access bean filter.
  - Can be extended if necessary.
- *MESGeneratedParam<ParamClassBaseName>Filter.java*
  - Base class of *MESParam<ParamClassBaseName>Filter*.
  - Contains the declaration of all standard methods to set filter criteria.
  - Should remain unchanged in case a re-generation of the parameter class is necessary (e.g. if the application table of the specific parameter class attributes was extended for a new column).
- *MESRtParam<ParamClassBaseName>.java*
  - Access bean for runtime parameter.
  - Can be extended if necessary.

- *MESRtParam<ParamClassBaseName>Filter.java*
  - Access bean filter for runtime parameter.
  - Can be extended if necessary.

AT ROWS IN XML FORMAT

The following AT rows are generated in XML format in subdirectories of the *<output directory>\AtRow* directory. They are important for deploying a phase. The ATRow entries are stored in files named after the ATRow GUID (e.g. *{78BC6EB4-DF7A-26FE-7E8E-E11BFCAC5F23}.xml*).

- *ATRow\X_ParameterClass* subdirectory
  Contains the parameter class application table entry.

TIP

You can re-use existing parameter classes, i.e. a parameter class can be used by multiple phases, so creating a new parameter class is only necessary if there is none that fits your needs.
If you create a new parameter class with string attributes, make sure to provide a corresponding length restriction in the data dictionary entry for the attribute. Otherwise, Recipe and Workflow Designer cannot enforce these restrictions and thus, may lead to master recipes or master workflows that cannot be saved.

What Is the Phase Copy Tool?

This section describes how to create a new phase based on an already existing one. For this purpose, the artifacts of the existing source phase are used to create a new phase with a user-specified name. The **Phase Copy Tool** serves the following purposes:

- Creation of new phase versions
  This is done by simply renaming a phase by means of the copy tool.
- Creation of new phases if an existing phase serves as a starting point for a new phase

In any case, the new phase will have the same runtime phase data, same output data, same process parameters, same number of material parameters, and the same number of equipment requirement parameters as the original phase.
If you wish to create new versions of the process parameters, phase runtime data, or output data, please refer to "Creating a New Version of a Phase and their Related Parameter Classes" (page 169).

IMPORTANT

Please note that the purpose of the Phase Copy Tool is to generate the artifacts that you need in order to build an installer for the new phase. Some of the artifacts of the phase will be created on your current system (the same artifacts that are created during phase generation), but the JAR file of the new phase and some of its artifacts will not be created on phase copy (e.g. sub-report). So you will not be able to use the new phase directly on your system.

The tool provides the following artifacts:

ARTIFACTS FROM THE PHASE GENERATOR

All artifacts created by the phase generator except for the Java files (see "What Is the Phase Generator?" (page 19)).

JAVA FILES

All Java classes of the source phase are copied and renamed according to the name and/or version of the new phase. The references in the source code and, if needed, the package names are updated.

XML DESCRIPTION FILES

The XML description files of the source phase are copied and the XML description of the phase building block is modified according to the properties of the new phase. For more information, see steps 1 to 3 of "Creating a Phase Building Block" (page 33) (Step 1: Creating an XML Description of the Phase Building Block (page 33), Step 2: Creating an XML Description of the Runtime Phase Data (page 41), Step 3: Creating an XML Description of the Runtime Phase Output (page 45)).

DSX FILES

All disassembled DSX artifacts of the source phase.
This can include report designs, images, data dictionary entries etc. The report design is renamed, if specified. The data dictionary classes and their message packs are also renamed. Artifacts of the phase generator overwrite the artifacts of the original phase.

TIP

Please note that some DSX artifacts need to be post-processed. The phase copy tool provides a list of folders that may need additional processing after the new phase has been created. For example, message packs are not automatically renamed. You may have to rename them or move the DSX artifact for the message packs to another project where both, the source and the new phase can reference it. Otherwise, conflicts may occur if both phases are installed on the same system. Also, if you use Java code in your report (e.g. a scriptlet), you may have to modify the report design manually.

Working with the Phase Copy Tool

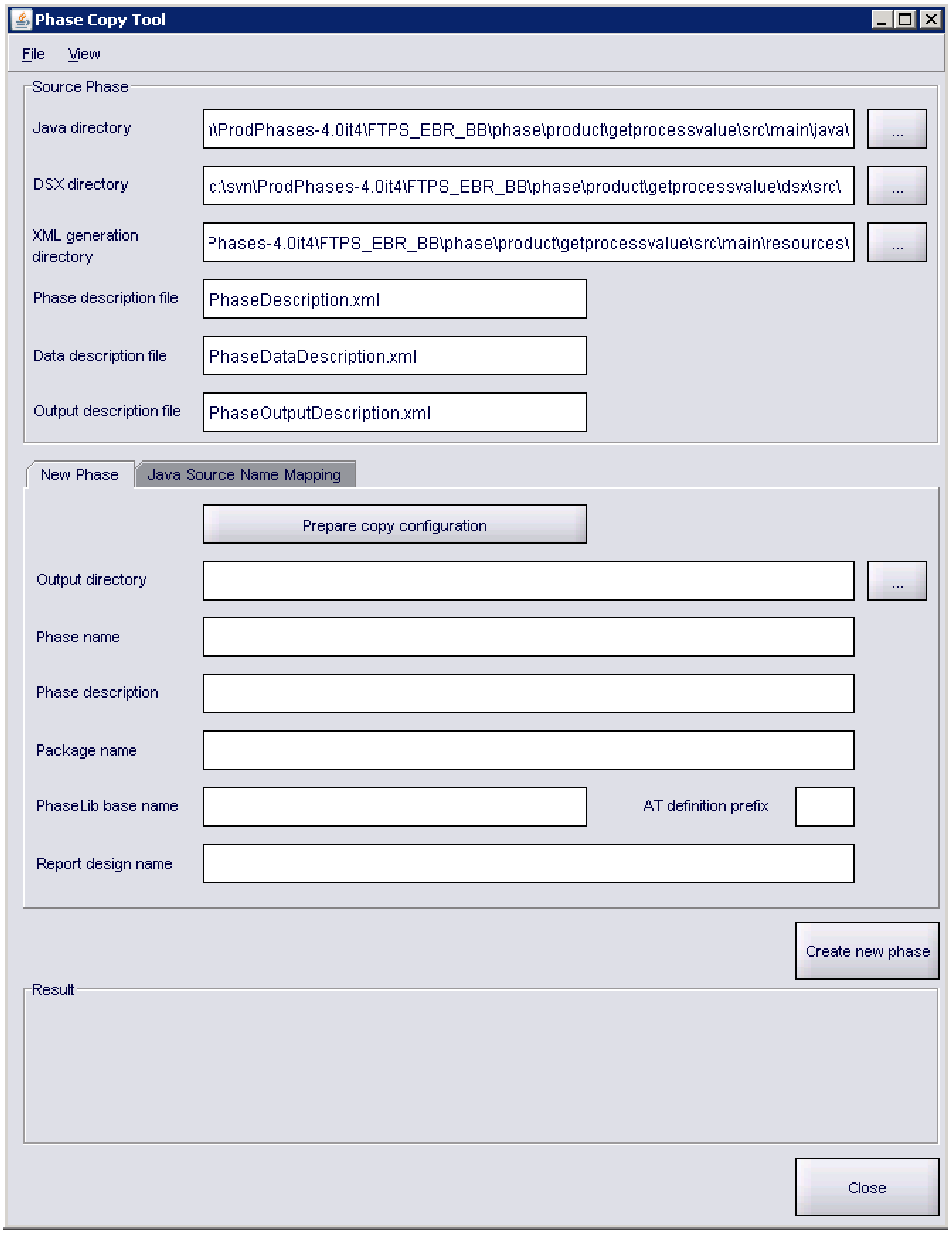
The **Phase Copy Tool** requires a set of configuration parameters specifying the source phase. All process parameters of the phase must be available on the system where you are performing the phase copy, i.e. the process parameters can be generated or the source phase has to be installed. Then, the tool prepares the parameters for the new phase. Some of them must be updated by the user in order to provide the information required for naming the Java source files. Finally, the new phase can be created.

TIP

The tool is not available in PharmaSuite, but only in the PharmaSuite Building Block SDK.

To create a copy of a phase, proceed as follows:

1. In Process Designer, run the **PhaseCopyTool** form to start the tool.
   OR
   In Shop Operations, open the **PhaseCopyTool** form to start the tool.
2. In the *Source Phase* tab, provide the required configuration parameters.
   - Java directory
     Location of all Java classes of the source phase. Specify the directory directly above the uppermost package directory.
     Example: The Java classes are in the *com.mycompany.phase.gettextvalue* package, then specify the directory that contains the *com* directory.
   - DSX directory
     Location of all disassembled DSX artifacts of the source phase.
   - XML generation directory
     Location of the XML description files of the source phase.
   - Phase description file
     Name of the XML description file of the source phase. It should be located in the *XML generation directory*.
   - Data description file
     Name of the XML description file of the runtime phase data of the source phase. It should be located in the *XML generation directory*.
     Only required, if the source phase stores runtime data.
   - Output description file
     Name of the XML description file of the runtime phase output of the source phase. It should be located in the *XML generation directory*.
     Only required, if the source phase stores output data.



Source Phase

Java directory	\ProdPhases-4.0it4\FTPS_EBR_BB\phase\product\getprocessvalue\src\main\java\
DSX directory	c:\svn\ProdPhases-4.0it4\FTPS_EBR_BB\phase\product\getprocessvalue\dsx\src\
XML generation directory	Phases-4.0it4\FTPS_EBR_BB\phase\product\getprocessvalue\src\main\resources\
Phase description file	PhaseDescription.xml
Data description file	PhaseDataDescription.xml
Output description file	PhaseOutputDescription.xml
Output directory	
Phase name	
Phase description	
Package name	
PhaseLib base name	
AT definition prefix	
Report design name	

Result

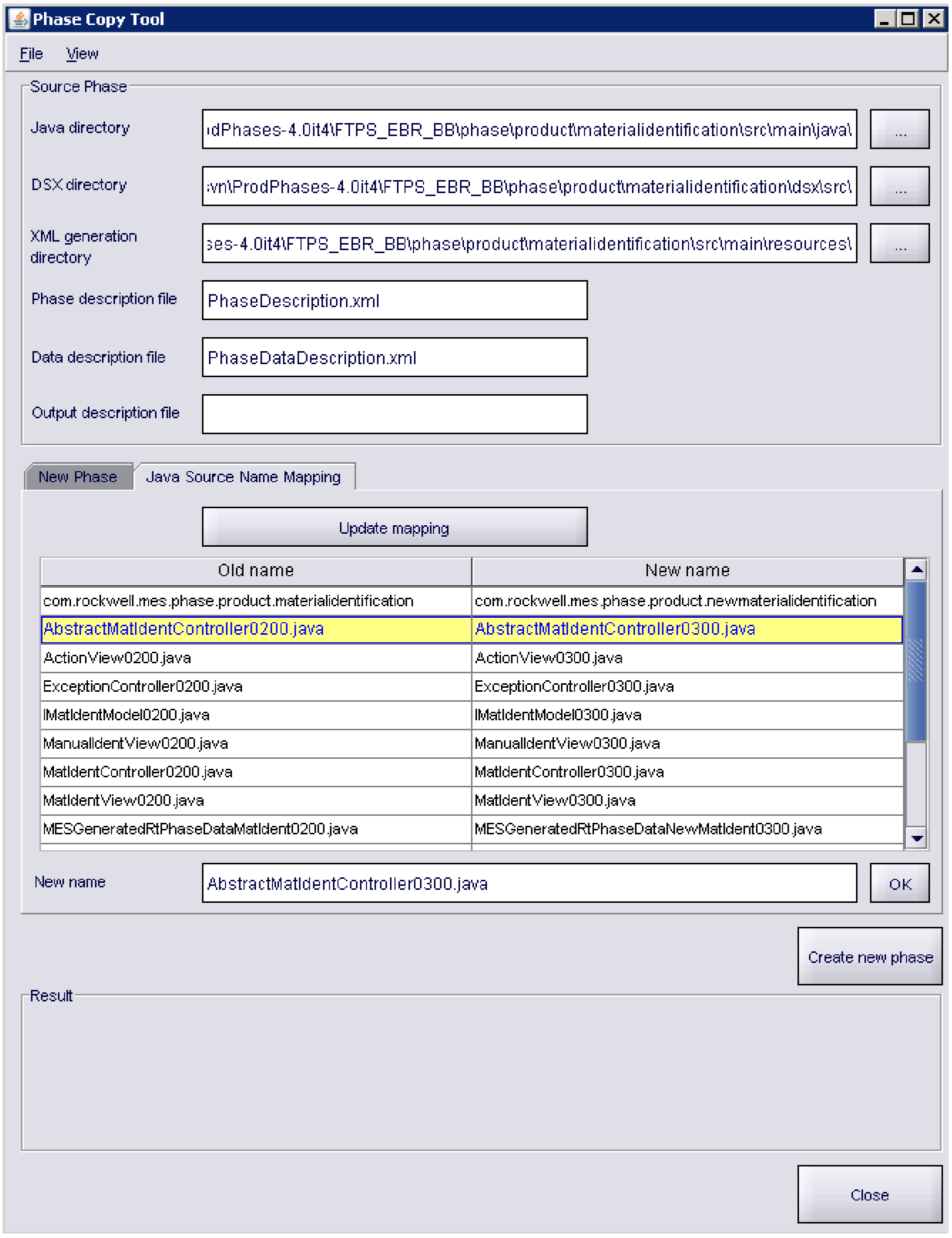
Figure 9: Phase Copy Tool with source phase data

3. In the *New Phase* tab, click the **Prepare copy configuration** button to prepare the configuration parameters required for the new phase.
   The tool reads the phase description file of the source phase and fills the form with the data to be used for the new phase. Review and update the data, if required. It is mandatory to update the *Phase name* and *PhaseLib base name* parameters. For more information, see section "Step 1: Creating an XML Description of the Phase Building Block" (page 33).
4. In the *Java Source Name Mapping* tab, click the **Update mapping** button to define the names of the new Java files.
   The tool lists the Java files and packages of the source phase and their new names.

The files are renamed according to the pattern described in section "What Is the Phase Generator?" (page 19).
If the source phase contains additional classes, the tool suggests new names for them according to the following pattern:

- *PhaseLib base name* of the source phase is replaced by the *PhaseLib base name* of the new phase.
  If the *PhaseLib base name* of the source phase and the new phase contain a version, the version is changed in the names of all Java classes. If the last four characters of a *PhaseLib base name* are digits, they are considered to be the version.
- To change the name of an additional class, select the corresponding row and edit the name in the *New name* text box.

TIP
Please note that in order to avoid hazardous class loading, each Java class of the new phase should have a fully qualified name that differs from the one in the source phase. For this purpose, either change the name of each Java class or its package name.



Source Phase

Java directory	ıdPhases-4.0it4\FTPS_EBR_BB\phase\product\materialidentification\src\main\java\
DSX directory	:vn\ProdPhases-4.0it4\FTPS_EBR_BB\phase\product\materialidentification\dsx\src\
XML generation directory	ses-4.0it4\FTPS_EBR_BB\phase\product\materialidentification\src\main\resources\
Phase description file	PhaseDescription.xml
Data description file	PhaseDataDescription.xml
Output description file	

Old name	New name
com.rockwell.mes.phase.product.materialidentification	com.rockwell.mes.phase.product.newmaterialidentification
AbstractMatIdentController0200.java	AbstractMatIdentController0300.java
ActionView0200.java	ActionView0300.java
ExceptionController0200.java	ExceptionController0300.java
IMatIdentModel0200.java	IMatIdentModel0300.java
ManualIdentView0200.java	ManualIdentView0300.java
MatIdentController0200.java	MatIdentController0300.java
MatIdentView0200.java	MatIdentView0300.java
MESGeneratedRtPhaseDataMatIdent0200.java	MESGeneratedRtPhaseDataNewMatIdent0300.java

New name: AbstractMatIdentController0300.java

Result

Figure 10: Phase Copy Tool with Java source name mapping information

5. Click the **Create new phase** button to trigger the phase creation based on the settings made in the previous steps.
   The tool creates a subfolder in the output directory named after the innermost sub-package of the new phase. If the phase creation is successful, the folder contains the following subfolders:
   - java: contains the Java classes of the new phase,

TIP
Please note that the newly generated Java classes can have some Checkstyle violations caused by refactoring (e.g. longer lines due to longer class names). This has to be fixed manually.

   - resources: contains the XML description files of the new phase, and
   - dsx: contains all disassembled DSX artifacts of the new phase, i.e. the ones generated by the phase generator and the ones copied from the source phase.

6. The tool displays a summary of the operation in the **Result** panel.
7. Close the tool.

Chapter 5

Creating a Phase Building Block

This section describes how to create a phase building block with the phase generator. Phases based on such a building block can be executed in the Production Execution Client.

The examples in the subsequent sections are based on the **MYC_StringValue** phase. The phase allows an operator to enter a string during execution. Phases instantiated from this phase building block will be able to store and display the entered string value.

To create a phase building block from scratch, perform the following steps:

1. Create an XML description of the phase building block (page 33).
2. Create an XML description of the runtime phase data (page 41). This step is only necessary if your phase needs to store specific phase data created during execution.
3. Create an XML description of the runtime phase output (page 45). This step is only necessary if you wish to enable the usage of your phase's output, created during execution, by other phases.
4. Run the phase generator (page 46) with the XML descriptions created in the preceding steps.
5. Implement the runtime phase executor (page 49) so that the phase is rendered and behaves according to your requirements during execution in EBR. This is most probably the step with the biggest effort.
6. Add optional features (page 73) to the Java code to adapt the phase rendering.
7. If the phase shall be visible in the Navigator, implement the *getNavigatorInfoColumn()* method in the runtime phase executor (page 87).
8. Verify the phase building block creation with the phase manager (page 88).

Step 1: Creating an XML Description of the Phase Building Block

The XML schema of a phase building block has the following structure:

XML 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="PhaseLibDescription">
    <xsd:complexType>
```

```
      <xsd:sequence>
        <xsd:element name="OutputDir">
          <xsd:simpleType>
            <xsd:restriction base="xsd:string">
              <xsd:minLength value="1" />
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:element>
        <xsd:element name="Name" type="Name64Type"/>
        <xsd:element name="Description">
          <xsd:simpleType>
            <xsd:restriction base="xsd:string">
              <xsd:minLength value="3" />
              <xsd:maxLength value="255" />
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:element>
        <xsd:element name="ShortDescription" minOccurs="0" maxOccurs="1">
          <xsd:simpleType>
            <xsd:restriction base="xsd:string">
              <xsd:minLength value="1" />
              <xsd:maxLength value="80" />
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:element>
        <xsd:element name="SubroutineName" type="Name64Type" minOccurs="0"
                                           maxOccurs="1"/>
        <xsd:element name="PackageName" type="Name64Type"/>
        <xsd:element name="PhaseLibBaseName">
          <xsd:simpleType>
            <xsd:restriction base="xsd:string">
              <xsd:minLength value="3" />
              <xsd:maxLength value="18" />
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:element>
        <xsd:element name="VisibleInNavigator">
          <xsd:complexType>
            <xsd:attribute name="value" type="xsd:boolean"/>
          </xsd:complexType>
        </xsd:element>
        <xsd:element name="InternalBuildingBlock">
          <xsd:complexType>
            <xsd:attribute name="value" type="xsd:boolean" default="false"/>
          </xsd:complexType>
        </xsd:element>
        <xsd:element name="ReportDesignName" type="Name64Type" minOccurs="0"
                                             maxOccurs="1"/>
        <xsd:element name="SupportExceptions" minOccurs="0" maxOccurs="1">
          <xsd:complexType>
            <xsd:attribute name="value" type="xsd:boolean" default="false"/>
          </xsd:complexType>
        </xsd:element>
        <xsd:element name="MaterialInputMin">
          <xsd:simpleType>
            <xsd:restriction base="xsd:integer">
              <xsd:minInclusive value="0"/>
              <xsd:maxInclusive value="50"/>
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:element>
        <xsd:element name="MaterialInputMax">
```

```
          <xsd:simpleType>
            <xsd:restriction base="xsd:integer">
              <xsd:minInclusive value="0"/>
              <xsd:maxInclusive value="50"/>
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:element>
        <xsd:element name="MaterialOutputMin">
          <xsd:simpleType>
            <xsd:restriction base="xsd:integer">
              <xsd:minInclusive value="0"/>
              <xsd:maxInclusive value="50"/>
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:element>
        <xsd:element name="MaterialOutputMax">
          <xsd:simpleType>
            <xsd:restriction base="xsd:integer">
              <xsd:minInclusive value="0"/>
              <xsd:maxInclusive value="50"/>
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:element>
        <xsd:element name="EquipmentRequirementMin" minOccurs="0" maxOccurs="1">
          <xsd:simpleType>
            <xsd:restriction base="xsd:integer">
              <xsd:minInclusive value="0"/>
              <xsd:maxInclusive value="50"/>
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:element>
        <xsd:element name="EquipmentRequirementMax" minOccurs="0" maxOccurs="1">
          <xsd:simpleType>
            <xsd:restriction base="xsd:integer">
              <xsd:minInclusive value="0"/>
              <xsd:maxInclusive value="50"/>
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:element>
        <xsd:element name="ATDefinitionPrefix">
          <xsd:simpleType>
            <xsd:restriction base="xsd:string">
              <xsd:minLength value="1" />
              <xsd:maxLength value="3" />
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:element>
        <xsd:element name="UsageContext" type="UsageContextType" minOccurs="1"
                                         maxOccurs="unbounded" />
        <xsd:element name="IsETOTriggerPhase" minOccurs="0" maxOccurs="1">
       <xsd:complexType>
      <xsd:attribute name="value" type="xsd:boolean" default="false" />
       </xsd:complexType>
        </xsd:element>
        <xsd:element name="IsServerRunPhase" minOccurs="0" maxOccurs="1">
           <xsd:complexType>
              <xsd:attribute name="value" type="xsd:boolean" default="false" />
           </xsd:complexType>
        </xsd:element>
        <xsd:element name="IsPauseAwarePhase" minOccurs="0" maxOccurs="1">
           <xsd:complexType>
              <xsd:attribute name="value" type="xsd:boolean" default="false" />
           </xsd:complexType>
        </xsd:element>
        <xsd:element name="ParameterClasses" minOccurs="0" maxOccurs="1">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="ParameterClass" type="ParameterClassType" minOccurs="0"
                                                 maxOccurs="unbounded"/>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
        <xsd:element name="DynamicParameterClasses" minOccurs="0" maxOccurs="1">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="MaximumNumber">
                <xsd:simpleType>
                    <xsd:restriction base="xsd:integer">
                        <xsd:minInclusive value="1"/>
                    </xsd:restriction>
                </xsd:simpleType>
              </xsd:element>
              <xsd:element name="DynamicParameterClass" minOccurs="1"
                                                        maxOccurs="unbounded">
                <xsd:complexType>
                  <xsd:sequence>
                    <xsd:element name="MaximumNumber" minOccurs="0" maxOccurs="1">
                      <xsd:simpleType>
                        <xsd:restriction base="xsd:integer">
                          <xsd:minInclusive value="1"/>
                        </xsd:restriction>
                      </xsd:simpleType>
                    </xsd:element>
                    <xsd:element name="Name" type="Name64Type"/>
                    <xsd:element name="UsageIdentifier" type="Name64Type"/>
                    <xsd:element name="DependentParameterClass" type=
                                      "ParameterClassType" minOccurs="0"
                                                           maxOccurs="unbounded"/>
                  </xsd:sequence>
                </xsd:complexType>
              </xsd:element>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

  <xsd:complexType name="ParameterClassType">
    <xsd:sequence>
      <xsd:element name="Name" type="Name64Type"/>
      <xsd:element name="UsageIdentifier" type="Name64Type"/>
      <xsd:element name="SortIndex" minOccurs="0" maxOccurs="1">
        <xsd:simpleType>
          <xsd:restriction base="xsd:integer"/>
        </xsd:simpleType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>

  <xsd:simpleType name="Name64Type">
    <xsd:restriction base="xsd:string">
      <xsd:minLength value="3" />
      <xsd:maxLength value="64" />
```

```
    </xsd:restriction>
  </xsd:simpleType>

  <xsd:simpleType name="UsageContextType">
        <xsd:restriction base="xsd:string">
            <!-- possible values must be choice element meanings of -->
            <!-- choice list "PhaseUsageContext" -->
            <xsd:enumeration value="Recipe" />
            <xsd:enumeration value="Workflow" />
            <xsd:enumeration value="All" />
        </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>
```

This is an example description suitable for our **MYC_StringValue** example phase:

Example: MYC_StringValue

```
<PhaseLibDescription>
  <OutputDir>[absolute path of output directory]</OutputDir>
  <Name>String Value (MYC) [1.0]</Name>
  <Description> Basic phase for Planned/Actual value data type String.</Description>
  <PackageName>com.mycompany.phase.MYCstringvalue</PackageName>
  <PhaseLibBaseName>MYC_StringValue0100</PhaseLibBaseName>
  <VisibleInNavigator value="true"/>
  <InternalBuildingBlock value="false"/>
  <MaterialInputMin>0</MaterialInputMin>
  <MaterialInputMax>0</MaterialInputMax>
  <MaterialOutputMin>0</MaterialOutputMin>
  <MaterialOutputMax>0</MaterialOutputMax>
  <ATDefinitionPrefix>MYC</ATDefinitionPrefix>
  <UsageContext>Recipe</UsageContext>
</PhaseLibDescription>
```

The individual nodes of the XML structure have the following meaning:

- PhaseLibDescription
  Top-level node.
- OutputDir
  Specify the output directory where all the generated files will be located (here: *<source_directory>\apps\testdata\src\com\rockwell\mes\apps\testdata\processdata*).

  TIP
  You can overwrite this specification with the *-outputDir* command line option (see "Step 4: Running the Phase Generator" (page 46)).
- Name
  Name of the phase building block.
  Maximum length is 64 characters.
- Description
  Description of the phase building block.

- PackageName
  Package name of the generated Java artifacts.
- PhaseLibBaseName
  Base name of the phase building block. This name can differ from the actual building block name in order to circumvent name length constraints, especially regarding the names of application tables.
  Maximum length is 18 characters (14 for the name, 4 for the version).

TIP
If the name of your phase building block is **MaterialIdentification0100**, then - according to our naming conventions - the generated application table for the runtime phase data will be called **MYC_PhDataMaterialIdentification0100**. But, since application table names can only be 26 characters long, this would be an invalid name. If you set **PhaseLibBaseName** to **MatIdent**, then the application table name will be **MYC_PhDatMatIdent0100**, which is a valid name.

- VisibleInNavigator
  Specifies whether phases based on this phase building block are visible in the Navigator.
- InternalBuildingBlock
  Specifies whether this phase building block is internal, i.e. not usable from Recipe and Workflow Designer. This flag should always be set to **false** for phase building blocks.
- MaterialInputMin
  Specifies the minimum number of material input parameters that can be assigned to phases based on this phase building block.
- MaterialInputMax
  Specifies the maximum number of material input parameters that can be assigned to phases based on this phase building block.
- MaterialOutputMin
  Specifies the minimum number of material output parameters that can be assigned to phases based on this phase building block.
- MaterialOutputMax
  Specifies the maximum number of material output parameters that can be assigned to phases based on this phase building block.
- ATDefinitionPrefix
  Specifies the prefix of the table and column names of the application table holding the runtime phase data. Use a vendor code consisting of up to three uppercase letters as prefix (e.g. **MYC**).
  X_ and **RS_** are reserved for PharmaSuite and PharmaSuite-specific product building blocks, respectively.

- UsageContext
  Specifies the context in which the phase building block shall be usable. This element can be specified multiple times. Possible values are:
  - Recipe
    Phase can only be used within recipes.
  - Workflow
    Phase can only be used within workflows.
  - All
    Phase can be used in any context.

There are also some optional XML elements, which are not used in the example above:

- ShortDescription
  Short description of the phase building block.
  Maximum length is 80 characters.
- SubroutineName
  Name of a subroutine. Subroutines may provide a customer-specific implementation that determines which phase-related data shall be displayed in the Navigator's information column.
- SupportExceptions
  Specifies that skeleton implementations for the *createPhaseExceptionComponent()*, *exceptionCallbackInSaveTransaction()*, *exceptionSigned()*, and *exceptionCanceled()* methods will be generated in order to support expected exceptions for recurring irregular situations (user-triggered).
  Furthermore, the **hasExceptions** flag of a phase building block is set if **SupportExceptions** is **true**. **hasExceptions** controls whether privilege parameters can be assigned to a phase based on the corresponding phase building block.
- ReportDesignName
  Specifies the name of the phase-specific sub-report used in the batch record. A phase-specific sub-report can have nested sub-reports.
  The sub-report is accessible via the Navigator's information column.
- EquipmentRequirementMin
  Specifies the minimum number of equipment requirement parameters that can be assigned to phases based on this phase building block.
- EquipmentRequirementMax
  Specifies the maximum number of equipment requirement parameters that can be assigned to phases based on this phase building block.
- IsETOTriggerPhase
  Specifies that the phase building block is used to trigger ETO-related runtime operations.

- IsServerRunPhase
  Specifies that the phase building block can only be executed in server-run operations on the Operation Execution server.
- IsPauseAwarePhase
  Specifies that the phase building block reacts on pause and continue events of a pause-enabled unit procedure.
- ParameterClasses
  Pure structuring node to introduce (static) parameter class assignments of the phase building block. Each assignment is represented by a child node called **ParameterClass**. Each **ParameterClass** node has the following children:
  - Name
    Name of the parameter class.
  - UsageIdentifier
    Indicates the specific usage of the parameter class (or, more precisely, the corresponding parameter instance) in the context of the phase building block (keep in mind that a parameter class can be used by several building blocks).
    Example: An identifier can be used within the runtime phase executor to retrieve and evaluate a certain parameter.
  - SortIndex
    Used to sort static process parameters in the user interface.
- DynamicParameterClasses
  Pure structuring node to introduce dynamic parameter class assignments of the phase building block. Each assignment is represented by a child node called **DynamicParameterClass**. A **DynamicParameterClass** node represents a dynamic process parameter and can have further child parameter classes. This implies that a **DynamicParameterClass** represents a whole bundle of dynamic process parameters. In addition, the **DynamicParameterClasses** node has a child node called **MaximumNumber** to specify the overall maximum number of dynamic process parameter bundles of a phase building block. A value of -1 means that the phase can handle an arbitrary number of dynamic process parameter bundles.
  The child node **DynamicParameterClass** itself has the following children:
  - MaximumNumber (optional)
    Can be used to specify the maximum number of this dynamic process parameter bundle a phase building block can handle.
    This option is not evaluated by the PharmaSuite framework.
  - Name
    Name of the parameter class.

- UsageIdentifier
  Indicates the specific usage of the parameter class (or, more precisely, the corresponding parameter instance) in the context of the phase building block. This identifier also serves as a (phase-independent) internal bundle identifier and is displayed in Recipe and Workflow Designer when a dynamic process parameter bundle is added to a phase.
- DependentParameterClass (optional)
  Can be used to define further child parameter class assignments of the dynamic process parameter bundle (i.e. the enclosing DynamicParameterClass node). The number of **DependentParameterClass** nodes is unlimited.
  A DependentParameterClass node itself has the same children as a static parameter class assignment represented by a **ParameterClass** node, i.e. Name, UsageIdentifier, and SortIndex. In contrast to static parameter class assignments, the SortIndex is interpreted as sort index relative to the master (i.e. top-level) parameter class of this dynamic process parameter bundle.

Step 2: Creating an XML Description of the Runtime Phase Data

This step is only necessary if your phase needs to store specific phase data created during execution.

The XML schema of the runtime phase data has the following structure:

XML schema of runtime phase data

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns="http://www.wbf.org/xml/RAPhaseData"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            elementFormDefault="qualified"
            attributeFormDefault="unqualified">
  <xsd:include schemaLocation="PhaseDataType.xsd"></xsd:include>
  <xsd:element name="RtPhaseDataAttributes">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="RtPhaseDataAttribute" maxOccurs="unbounded">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="Name" type="xsd:string"/>
              <xsd:element name="Description" type="xsd:string" minOccurs="0" />
              <xsd:element name="DataType" type="PhaseDataType"/>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

The XML schema of the included *PhaseDataType.xsd* looks as follows:

XML schema of PhaseDataType.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        elementFormDefault="qualified"
        attributeFormDefault="unqualified">
  <complexType name="PhaseDataType">
    <choice>
      <element name="String">
        <complexType>
            <attribute name="length" type="positiveInteger"
                default="256" />
        </complexType>
      </element>
      <element name="Long">
        <complexType />
      </element>
      <element name="Float">
        <complexType />
      </element>
      <!-- used "Bool" instead of "Boolean" to avoid naming conflicts caused by
           Castor -->
      <element name="Bool">
        <complexType />
      </element>
      <element name="Decimal">
        <complexType />
      </element>
      <element name="BigDecimal">
        <complexType />
      </element>
      <element name="DateTime">
        <complexType />
      </element>
      <element name="Duration">
        <complexType />
      </element>
      <element name="MeasuredValue">
        <complexType />
      </element>
      <element name="Binary">
        <complexType />
      </element>
      <element name="Object">
        <complexType>
          <attribute name="class" use="required">
                <simpleType>
                    <restriction base="string">
                        <enumeration value="Part" />
                    </restriction>
                </simpleType>
          </attribute>
        </complexType>
      </element>
      <element name="ApplicationTable">
        <complexType>
          <attribute name="name" type="string" use="required" />
          <attribute name="interfaceClassName" type="string" use="required" />
          <attribute name="implClassName" type="string" use="required" />
        </complexType>
      </element>
    </choice>
  </complexType>
</schema>
```

As you can see in the XML schema, *PhaseDataType.xsd* supports the following data types:

- String
  The default String length is 256 characters. Use the **length** attribute to specify an individual String length.

  TIP
  Please be aware that the maximum length of a String column depends on database constraints. This is a restriction of the FactoryTalk ProductionCentre platform.

- Long
- Float
- Bool (mapped to Boolean; there is a technical reason for this difference)
- Decimal

  TIP
  We do not recommend to use the Decimal data type, since (as the Float data type) it cannot represent all numbers accurately and may result in a loss of scale. Please use the BigDecimal data type instead.

- BigDecimal
- DateTime

  TIP
  Please be aware that the accuracy of the date/time data type differs between Oracle and MS SQL databases. MS SQL provides an accuracy of milliseconds while Oracle only provides an accuracy of seconds. This is a restriction of the FactoryTalk ProductionCentre platform.

- Duration (mapped to *MESDuration* class)
- MeasuredValue
- Binary
- Object
  The object’s class has to be specified by the **class** attribute. The following classes are currently supported:
  - Part: Class of FactoryTalk ProductionCentre

- Application table

  TIP
  The phase generator supports only application tables that are wrapped by an interface implementing *IMESATObject* and by a class that extends *MESATObject*.

  To specify a particular application table as a data type, provide the following attributes:

  - name
    Name of the application table.
  - interfaceClassName
    Full class name (i.e. including package name) of the interface that wraps the application table.
  - implClassName
    Full class name of the class that wraps the application table (and implements the interface listed above).

This is an example description of phase data suitable for our **MYC_StringValue** example phase:

```
<RtPhaseDataAttributes>
  <RtPhaseDataAttribute>
    <Name>actualValue</Name>
    <DataType>
      <String/>
    </DataType>
  </RtPhaseDataAttribute>
</RtPhaseDataAttributes>
```

The individual nodes of the runtime phase data XML structure have the following meaning:

- RtPhaseDataAttributes
  Top-level node containing 1 to N child nodes called **RtPhaseDataAttribute**, each representing one phase data attribute.
- An **RtPhaseDataAttribute** node has the following child elements:
  - Name
    Base name of the attribute. When the corresponding application table is generated, the phase generator automatically adds the prefix configured by the *ATDefinitionPrefix* element to the base name in order to build the column name of the attribute.
    ATDefinitionPrefix is an element of the XML description of the phase building block (page 33).

- DataType
  Data type of the attribute. For details see description of *PhaseDataType.xsd* above.

Step 3: Creating an XML Description of the Runtime Phase Output

This step is only necessary if your phase needs to store specific output data created during execution.

The XML schema of the phase output data has the following structure:

XML schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns="http://www.wbf.org/xml/RAPhaseData"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            elementFormDefault="qualified"
            attributeFormDefault="unqualified">

  <xsd:include schemaLocation="PhaseDataType.xsd"></xsd:include>

  <xsd:element name="PhaseOutputDescription">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="PhaseOutput" maxOccurs="unbounded">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="Name" type="xsd:string"/>
              <xsd:element name="DisplayName" type="xsd:string"/>
              <xsd:element name="Description" type="xsd:string"/>
              <xsd:element name="DataType" type="PhaseDataType"/>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

For a description of the included *PhaseDataType.xsd* see "Step 2: Creating an XML Description of the Runtime Phase Data" (page 41).

The individual nodes of the XML structure have the following meaning:

- PhaseOutputDescription
  Top-level node containing 1 to N child nodes of *PhaseOutput* type, each representing one output attribute.
- A **PhaseOutput** node has the following child elements:
  - Name
    Name of the output. When the corresponding application table is generated, the phase generator automatically adds the prefix configured by the *ATDefinitionPrefix* element to the base name in order to build the column

name of the attribute.
ATDefinitionPrefix is an element of the XML description of the phase building block (page 33).

- DisplayName
Name of the output to be used externally, e.g. in the UI during output selection.
- Description
Description of the output. When the corresponding application table is generated, the phase generator automatically adds this description to the corresponding output column.
- DataType
Data type of the attribute. For details, see description of *PhaseDataType.xsd* in "Step 2: Creating an XML Description of the Runtime Phase Data" (page 41).

TIP
The **Binary** type is only supported in input expressions of parameter attributes, but not in transition conditions.

Step 4: Running the Phase Generator

In order to run the phase generator, you have to call the *mainImpl()* method of the *PhaseLibGenerator* class. The method looks as follows:

```
public void mainImpl(String[] args)
```

The method can take ordinary arguments optionally.

- *-outputDir <output directory name>*
The base directory for the generated output. (Overwrites any specification in the phase building block description.)
The output directory of a particular a phase also depends on the specified package name. For example, if your package name is *com.mycompany.phase.mycstringvalue*, the phase output will be placed into the *<output directory name>\mycstringvalue* directory.
- *-batch <input directory name>*
Runs the generator in batch mode in a given base input directory. We recommend to use this argument, if you wish to generate several phases with a single call of the generator. The generator scans all of the immediate subdirectories of the given directory. If a subdirectory contains a *PhaseLibDescription.xml* file, a corresponding phase will be generated. The same applies to the *PhaseDataDescription.xml* and *PhaseOutputDescription.xml* files, the corresponding data and output will be generated as well.
You can also use the argument to generate a single phase: In this case, the given

directory should contain a *PhaseLibDescription.xml* file, then, the generator will run for this directory only (and will not descend into subdirectories).

Additionally, the generator gets its arguments as system properties (i.e. introduced by the *-D* option). The following arguments are supported:

- *-DphaseLibDescFile*
  The (relative or full) path name of the XML file that contains the description of the phase building block.
  Ignored in batch mode.
- *-DphaseDataDescFile*
  The (relative or full) path name of the XML file that contains the description of the runtime phase data.
  Ignored in batch mode.
- *-DphaseOutputDescFile*
  The (relative or full) path name of the XML file that contains the description of the phase output data.
  Ignored in batch mode.
- *-DexecuteCritical*
  If specified and set to *Y*, the generator performs changes in the form of updates to the generated artifacts that are considered as critical (see "Critical Changes" (page 48)).
- *-DsharedLib*
  If specified and set to *commons-shared-phase-mvc*, the generator creates a skeleton implementation of the runtime phase executor according to the MVC pattern (see "Step 5: Creating a Phase Executor (Java Code)" (page 49)).
- *-DJNP_ADDRESS*
  JNP address of the application server (needed to generate the database-related artifacts).
- *-DHTTP_ADDRESS*
  HTTP address of the application server (needed to generate the database-related artifacts).

You can call this method from Pnuts (e.g. from within a corresponding form implemented by yourself). This is an outline of what the corresponding Pnuts code could look like:

Example Pnuts code

```
import("com.rockwell.mes.tools.phases.impl.PhaseLibGenerator")

function generatePhaseLib() {
  // get the file names of the phase building block description and the
  // runtime phase data description, e.g. from some edit controls named
  // "phaseLibDescPath" and "rtPhaseDataDescPath"
  phaseLibDescFileName = phaseLibDescPath.getText()
  rtPhaseDataDescFileName = rtPhaseDataDescPath.getText()

  // call PhaseLibGenerator
  generator = PhaseLibGenerator()
  args = [] // currently no ordinary program arguments
  System::setProperty("phaseLibDescFile", phaseLibDescFileName)
  System::setProperty("phaseDataDescFile", rtPhaseDataDescFileName)
  generator.mainImpl(args)
}
```

After calling the *PhaseLibGenerator.mainImpl()* method, all the artifacts described in "What Is the Phase Generator?" (page 19) will be available.

In order to run your form containing the above Pnuts code, please make sure the following libraries have been imported into Process Designer:

- *tools-phases-ifc.jar* (located in *<source_directory>\dependencies\FTPS_DSX\addon_jars\*)
- *tools-phases-impl.jar* (located in *<source_directory>\dependencies\FTPS_DSX\addon_jars\*)

Additionally, ensure that the FactoryTalk ProductionCentre *com.rockwell.tools.dsx.DSXDisassembler* class is in your classpath. The class is, for example, contained in *ProductionCentreClient-<build number>_JBossSA.jar*.
If you do not have access to *ProductionCentreClient-<build number>_JBossSA.jar*, you can obtain the class from the *importexport.jar* file of FactoryTalk ProductionCentre. The *importexport.jar* file is located in the *PlantOpsWebStart-JBoss.war* directory of your FactoryTalk ProductionCentre JBoss installation (e.g. *c:\Rockwell\FT_ProductionCentre\jboss\server\datasweepConfig\deploy\DSPlantOperations.ear\PlantOpsWebStart-JBoss.war\importexport.jar*).

Critical Changes

Critical changes are changes that are incompatible with previous versions of a phase and thus incompatible with master recipes, master workflows, and building blocks in which previous versions of this phase have been used. They can only occur if a phase or its associated artifacts, respectively, already exist, i.e. when a phase is created for the first time, there are no critical changes.

The following changes are considered as critical and are therefore only executed if the *executeCritical* system property is set to *Y*.

- Deleting unused parameter class assignments.
- Changing the data type of an application table column.
- Reducing the length of an application table column of the *String* type.

- Increasing the minimum number of material input or output parameters.
- Decreasing the maximum number of material input or output parameters.
- Increasing the minimum number of equipment requirement parameters.
- Decreasing the maximum number of equipment requirement parameters.

If one of the above mentioned critical change scenarios occurs and critical changes are not allowed, the phase generation aborts.

Step 5: Creating a Phase Executor (Java Code)

The phase executor (or runtime phase executor) determines the representation (if any, see "Providing the GUI" (page 51)) and collects phase-specific data during processing. After the business logic has been executed, the phase must be completed (see "Completing a Runtime Phase in the Phase Executor" (page 64)) and if applicable, the output (see "Creating the Runtime Phase Output" (page 67)) and the runtime phase data must be created (see "Creating the Runtime Phase Data" (page 66)).

A phase executor class **with a GUI**, which runs in a Production Execution Client environment, should be derived from *AbstractPhaseExecutorSwing*; whereas a phase executor class **without a GUI**, which runs in a PharmaSuite OE server environment, should be derived from *AbstractServerSidePhaseExecutor*.

Basically, a phase executor will be instantiated for one of the following purposes:

- For preview. In this case, the phase executor instance is in the **Preview** status. The status will never change during the instance's lifetime. These instances have no runtime phase, but a phase.
- For execution. In this case, the phase executor instance is in the **Active** or **Completed** status. These instances have a corresponding runtime phase and an *IPhaseCompleter* object is assigned. This object is available with *getPhaseCompleter()*. A phase completer is an object to be used to perform the completion of the runtime phase.

Corresponding to the purposes of the instantiation of a phase executor, the phase executor has two constructors. The phase executor in the **Preview** status is simply needed to show the preview GUI either during execution or in Recipe and Workflow Designer. The phase executor for the execution must provide the GUI for the phase and provide the behavior during execution. Phases running on a server do not have a GUI at all.

TIP - RTPHASEEXECUTOR AND MVC

For more complex client-side phases, you may wish to make use of the Model-View-Controller pattern.

The phase generator supports the MVC pattern if the following system property argument is specified:

- *-DsharedLib=commons-shared-phase-mvc*

Thus, instead of generating only one file (*RtPhaseExecutor<BasePhaseName>.java*) for the skeleton implementation of the runtime phase executor, the following files are generated:

- *RtPhaseModel<BasePhaseName>.java*
- *RtPhaseView<BasePhaseName>.java*
- *RtPhaseExecutor<BasePhaseName>.java*
- *RtPhaseExceptionView<BasePhaseName>.java*
- *RtPhaseActionView<BasePhaseName>.java*

Consider Abort and Repair

➢ Does not apply to server-side phases running on the OE server.

This section is located at the beginning of the Java coding-related chapter, since for runtime phases that have been terminated as described here, certain assumptions regarding the status of a runtime phase are no longer valid.

Abort and repair are operator-triggered recovery measures which **abnormally** terminate an active runtime phase. This is persisted as *terminateReason* within *IMESRtPhase*. To access *terminateReason* use

```
public PhaseTerminationReason getTerminateReasonEnum()
```

The *PhaseTerminationReason* enumeration consists of the following values:

- COMPLETED, used for regular completion,
- SKIPPED, used if runtime phase was terminated with the **Abort** action,
- ABORTED, used if runtime phase was terminated with the **Repair** action.

If a runtime phase is not terminated yet, *getTerminateReasonEnum()* returns null.

Prior to the introduction of the recovery measure (with PharmaSuite 8.4), a terminated runtime phase was always a COMPLETED phase. Furthermore, the phase executor controlled the transition to the COMPLETED state. As a developer of a phase executor you could assume, that for instance, the *performCompletionCheck()* and *performCompletion()* methods were executed successfully. Along with the status transition, phase data and phase output were set and saved accordingly.

A SKIPPED or ABORTED runtime phase is terminated without invoking *performCompletionCheck()* and *performCompletion()*.

For this reason, the platform suppresses access to certain methods of a SKIPPED or ABORTED phase executor:

- *public CControl createPhaseActionCControl()*
  Post-completion action view is not available.

- *public CControl createPhaseExceptionCControl()*
  User-triggered exceptions are not available.
- *public List<String> getActionButtonTexts()*
  Post-completion action buttons are not available.
- *public String getNavigatorInfoColumn()*
  The information column of the Navigator is provided by the PharmaSuite framework and displays, e.g. **Aborted**.

TIP

This applies for runtime phases with `isCompleted() == true`.

Do not assume that phase data and phase output are available. We recommend to perform a null check before phase data or phase output is referenced. The same is true for other fields within the phase executor that you may have created.

Providing the GUI

➢ Does not apply to server-side phases running on the OE server.

If the phase executor displays a GUI during processing, the GUI has to be created and initialized in the *createPhaseComponent()* method. This method returns a *JComponent*. By default, the parent of this component uses the *PhasePanelUI* class as UI delegate. The UI delegate paints a background gradient if the phase is in the **Active** status.

Since a phase can be in different statuses (**Preview**, **Active**, **Completed**), the phase developer has to make sure to render the phase differently according to these statuses during processing. For instance, it does not make sense to allow data input or button click during the **Preview** or **Completed** statuses. Moreover, saved values should be displayed after the phase has been completed and is in the **Completed** status. The status is available in the phase executor by calling *getStatus()*. The *createPhaseComponent()* method must evaluate the status. It is highly recommended to propagate this status to the controls that you wish to put into the phase component.

The status of a phase component is immutable. That means, whenever a status change happens the phase component will be recreated. The status of a phase executor either corresponds to the status of the runtime phase or is in the **Preview** status.

The following method is used to provide the GUI for a phase:

- *createPhaseComponent()*
  This method must be implemented by each phase executor. It must return the GUI for the phase to be displayed during execution and for the preview in Recipe and Workflow Designer. The returned GUI depends on the status as mentioned above. The method will be called by phase executors in all statuses.

In the **Preview** status, when *createPhaseComponent()* throws a RuntimeException intentionally or due to an error, the framework renders a default preview. The default

preview consists of a two-line string (see figure below). Hence, the default preview can be used as a fallback rendering mechanism in case the regular preview cannot be rendered. This may be the case when the phase executor's GUI depends on the output parameters of the previous phase, which may not yet be completed at that time.
The message displayed by the default preview is stored in the **phasePreviewNotAvailable_ErrorMsg** message of the **pec_ExceptionMessage** message pack.

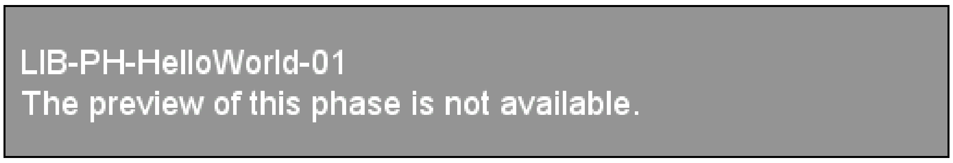

LIB-PH-HelloWorld-01
The preview of this phase is not available.

Figure 11: Default preview of phases

It is not allowed to synchronously complete a phase in *createPhaseComponent()*. If you wish to implement an automatic completion of a phase, you have to call it using *invokeLater*.

Example: Automatic completion of a phase

```
Public JComponent createPhaseComponent() {
...
  // Implementing an auto complete of the phase
  SwingUtilities.invokeLater(new Runnable() {
    public void run() {
      getPhaseCompleter().completePhase();
    }
  });
...
}
```

There are further methods to provide user interfaces of a phase executor for specific purposes:

- Adding exception handling for recurring irregular situations (see "Step 1: Defining the User Interface" (page 143) for the exception options in your phase executor).
- Adding actions for completed phases (see "Step 2: Defining the User Interface of the Actions" (page 155)).

TIP - ACCESS TO RUNTIME PHASE

Please note that access to the runtime phase depends on the status of the phase. In order to properly retrieve the runtime phase associated with a phase executor, please use the *getRtPhase()* method. In the **Preview** status, there is no runtime phase. This is why the *getPhase()* method is provided. *getRtPhase()* will return null in this case. When a phase is executed, the runtime phase is saved immediately after it has been created and before *createPhaseComponent()* is called.

There are a few typical places where you need access to the runtime phase:

- *createPhaseComponent()*: Use *getRtPhase()*, unless in the **Preview** status.
- *completePhase()*: At this time, the phase is in the **Active** status. Calling *getRtPhase()* should always return a (non-null) runtime phase.

TIP - BACKGROUND OF PHASE UI

If you layout your phase by means of panels, we recommend to make the panels transparent (opaque=false). Then you will see the standard background of the phase component, which depends on the status of the phase. The background of a phase

- is gray in the **Preview** or **Completed** statuses and
- has a gradient from light blue to dark blue in the **Active** status.

You can adjust all three default colors in the PEC style sheet, 'PhasePanel' section. For information about PharmaSuite style sheets, see chapter "Changing the General Appearance of PharmaSuite" in Volume 1 of the "Technical Manual Configuration and Extension" [A1] (page 175).

TIP - STYLING OF UI COMPONENTS

By default, your Swing components are styled by the style sheet engine. If you need a special styling, you can add the *IEspeciallyStylable* interface to your UI components and overwrite the *applySpecialStyling()* method. This method is called by the style sheet engine **after** the control has been styled by the current style sheet settings. It allows to apply very specific styling rules.

TIP - VERTICAL SCREEN SIZE OF THE PRODUCTION EXECUTION CLIENT

Depending on its configuration, the Production Execution Client runs either with the default size or a specified height of the current screen. For details, see **ProductionExecutionClient/PercentageOfVerticalScreenSize** configuration key in chapter "Configuration Keys of PharmaSuite" in Volume 4 of the "Technical Manual Configuration and Extension" [A3] (page 175).

Available Swing Components

The following Swing components are available:

- *PhaseColumnLayout*
  Layout manager that provides some pre-configured layouts. They support you with creating phases with a common alignment. For details, see section "Using the PhaseColumnLayout Layout Manager" (page 56).
- *PhaseButton*
  Button in the common look and feel of PharmaSuite.
  The example shows how to create a **Save** button:

```
PhaseButton saveButton = PhaseSwingHelper.createPhaseButton("Save", getStatus());
```

- *ConfirmButton*
  Button in the common look and feel of PharmaSuite and with the **Confirm** label.
  The component has the standard name of a **Confirm** button, so the framework is

able to recognize the **Confirm** button. If you use the *PhaseColumnLayout*, the **Confirm** button will be created for you.

- *PhaseToggleLockButton*
  Button in the common look and feel of PharmaSuite. The button toggles between the *Locked* and *Unlocked* statuses. Each status is represented by a graphical symbol on the button. You can register toggle event listeners to react to status changes (e.g. to lock the **Confirm** button dependent on *PhaseToggleLockButton*).

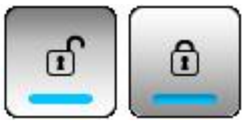


Figure 12: Button to toggle between Unlocked and Locked

```
final PhaseToggleLockButton ptb = new PhaseToggleLockButton(getStatus());
final ConfirmButton confirm = (PhaseColumnLayout)
                              layoutPanel.getLayout()).getConfirmButton()

confirm.setEnabled(!ptb.isLocked());
ptb.addToggleEventListener(new PhaseToggleLockButton.ToggleEventListener() {
  @Override
  public void buttonToggled(boolean locked) {
    confirm.setEnabled(!locked);
  }
});
```

- *PhaseToggleUomButton*
  Button in the common look and feel of PharmaSuite. The button toggles between units of measure.

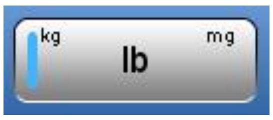


Figure 13: Button to toggle between units of measure

```
PhaseToggleUomButton uomButton = new PhaseToggleUomButton(
                     PCContext.getFunctions().getUnitOfMeasure("g"), getStatus());
uomButton.addToggleObjectListener(new PropertyChangeListener() {
  @Override
  public void propertyChange(PropertyChangeEvent evt) {
    System.out.println("Selected Uom = " + evt.getNewValue());
  }
});
```

- *MESStandardGrid, MESFastGrid*
  Swing grids are available in two versions. *MESStandardGrid* expects the data as a list of objects of a bound class (often created using a filter). Usually, the data for *MESFastGrid* is fetched by an SQL statement. For details, see sections "Creating an MESStandardGrid" (page 57) and "Creating an MESFastGrid" (page 61).

- *PhaseSwingHelper*
  Provides helper methods to create standard components, for example

- Text boxes

```
JTextField textField = PhaseSwingHelper.createJTextField(getStatus());
```

- Labels

```
JLabel label = PhaseSwingHelper.createJLabel(getPhase().getName());
```

- *BarcodeListenerSwingProxy*
  Provides support for barcode listening. You will receive barcode events only if the component passed to the proxy is visible.

```
BarcodeListenerSwingProxy barcodeProxy = new BarcodeListenerSwingProxy(layoutPanel);
barcodeProxy.addBarcodeScannedListener(new IBarcodeScannedListener() {
  @Override
  public void barcodeScanned(String barcode) {
    System.out.println("Scanned barcode = " + barcode);
  }
});
barcodeProxy.register();
```

- *PhaseBarcodeField*
  Text field that supports barcode listening. In the **Active** status of a phase, the field contains the last scanned barcode. For operators, the field is read-only.

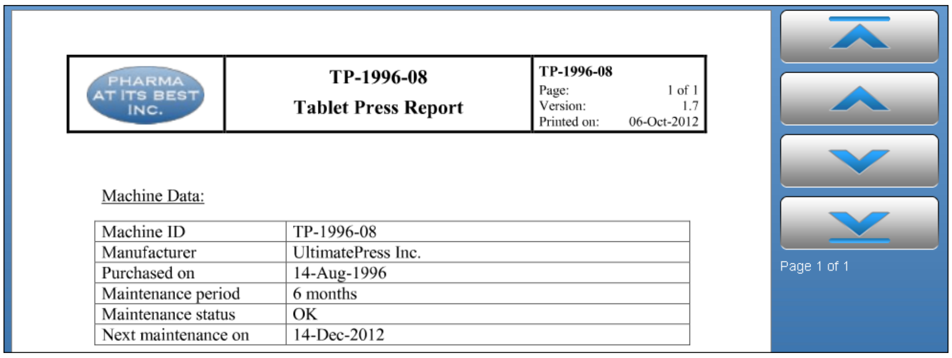
```
PhaseBarcodeField barcodeField = new PhaseBarcodeField(getStatus());
barcodeField.addBarcodeScannedListener(new IBarcodeScannedListener() {
  @Override
  public void barcodeScanned(String barcode) {
    System.out.println("Scanned barcode = " + barcode);
  }
});
```

- *PhaseSeparator*
  Component that can be used to separate multiple exception handling panels so that they have the same look and feel as the phases in the execution view.

```
JPanel separator = new PhaseSeparator();
```

- *PdfRenderer* and *PdfPanel*
  The *PdfRenderer* offers the capability to render a PDF document inside a *JPanel* or *JScrollPane*. It also provides utility methods to convert the PDF to a thumbnail or to a list of page images.
  The *PdfPanel* embeds the *PdfRenderer* and adds navigation buttons to the right.

TIP
The *PdfRender* is not available in PharmaSuite, but only in the PharmaSuite Building Block SDK. If you wish to use the component, you must also take care about its deployment.
Classes can be found in
com.rockwell.mes.shared.viewhelper.PdfRenderer0100 and
com.rockwell.mes.shared.viewhelper.swing.PdfPanel0100.



PHARMA AT ITS BEST INC.

TP-1996-08
Tablet Press Report

TP-1996-08	
Page:	1 of 1
Version:	1.7
Printed on:	06-Oct-2012

Machine Data:

Machine ID	TP-1996-08
Manufacturer	UltimatePress Inc.
Purchased on	14-Aug-1996
Maintenance period	6 months
Maintenance status	OK
Next maintenance on	14-Dec-2012

Figure 14: PdfRenderer and PdfPanel

```
File pdfFile = new File("C:/mypath/mypdf.pdf);
PdfRenderer0100 pdfRenderer = new PdfRenderer0100(pdfFile.getAbsolutePath());
JScrollPane pdfScrollPane = new JScrollPane();
pdfScrollPane.setViewportView(pdfRenderer.getComponent());
```

```
JPanel phasePanel = new JPanel();
PdfPanel0100 pdfRendererPanel = new PdfPanel0100(pdfRenderer, status);
pdfRendererPanel.renderPdf();
phasePanel.add(pdfRendererPanel);
```

Using the PhaseColumnLayout Layout Manager

The *PhaseColumnLayout* layout manager provides four pre-configured layouts.

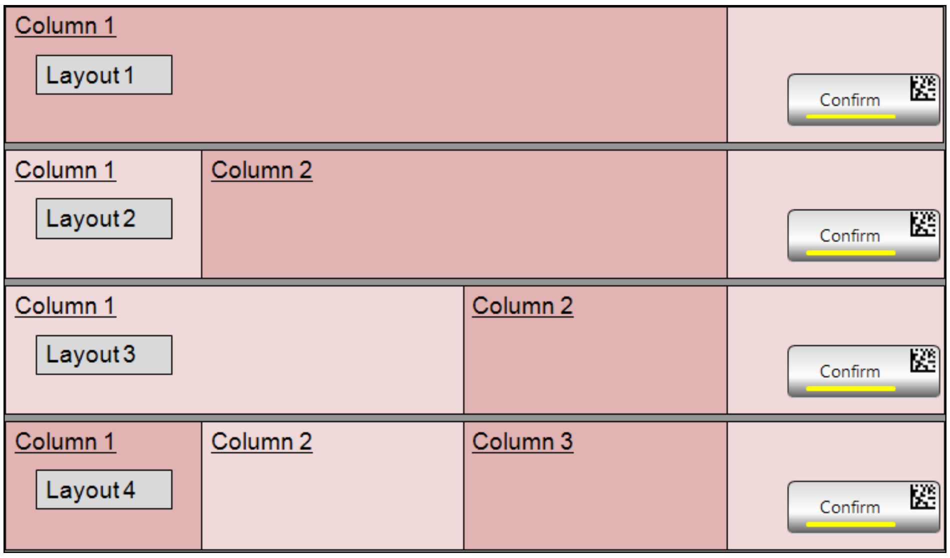


Figure 15: Reference layout templates

In order to create phases with a common alignment, set the *PhaseColumnLayout* layout manager to your panel and supply the desired layout in the constructor of the layout manager. To create a transparent *JPanel* with the *PhaseColumnLayout* layout manager, make use of the *PhaseSwingHelper* class' helper method. Then, you can place your components in the column panels.

Additionally, the column layout provides the **Confirm** button in the rightmost column. Depending on your requirements, you can keep the button or remove it.

The code example below shows how to use the *PhaseColumnLayout* layout manager. The layout manager ensures the default gaps between a UI component within a column and the column border according to the phase UI style guide. So, you can place a UI component within a column without gaps to the border.

In the following example, the first column contains a label with default gaps to the upper and left border of the panel:

```
final JPanel layoutPanel =
    PhaseSwingHelper.createPanel(Layout.LAYOUT_TWO_2ND_WIDER_COLUMN, getStatus());
JLabel label = new JLabel(“Test”);
layoutPanel.add(label, Column.FIRST_COLUMN);
```

This code snippet shows how to add an edit box in the second column aligned with the **Confirm** button:

```
edit = PhaseSwingHelper.createJTextField(getStatus());
layoutPanel.add(edit, Column.SECOND_COLUMN);
```

This code snippet shows how to get the **Confirm** button:

```
final JPanel layoutPanel =
      PhaseSwingHelper.createPanel(Layout.LAYOUT_TWO_2ND_WIDER_COLUMN, getStatus());
...
if (getStatus() == Status.ACTIVE) {
  /** confirm button listener */
  class ConfirmButtonListener implements ActionListener {
    @Override
    public void actionPerformed(ActionEvent event) {
      LOGGER.debug("Button clicked.");
      getPhaseCompleter().completePhase();
    }
  }
  ConfirmButton confirmButton =
             ((PhaseColumnLayout) layoutPanel.getLayout()).getConfirmButton();
 confirmButton.addActionListener(new ConfirmButtonListener());
  }
...
```

Creating an MESStandardGrid

➢ Does not apply to server-side phases running on the OE server.

To create an *MESStandardGrid*, basically, perform the following steps:

1. Create an *MESStandardGrid* object and set some of the grid's properties (e.g. sortable, allow reordering). For configuration options, see *IMESGridConfiguration*.
2. Set the class that is bound to the grid. The bound class is the class of the objects to be shown in the grid. Furthermore, it is the default data dictionary class.
3. Configure the columns to be displayed (e.g. data type, corresponding data dictionary element, alignment).

4. Fill the table with the data by providing the list of objects.

Since the *MESStandardGrid* grid expects the data to be filled in as a list of objects of a class (the bound class), each row of a grid is an object of the bound class of the grid. The list of data can be created using a filter of this class.

We provide some helper methods in the *PhaseSwingHelper* class to support the creation and configuration of *MESStandardGrid*.

- Create an *MESStandardGrid* (make use of *StandardColumnConfig* to configure each column)
  You can configure the *MESStandardGrid* when you provide a *StandardColumnConfig* object for each column. A *StandardColumnConfig* object contains properties for all the settings of a column. This leads to an easily readable and well-structured configuration:

Example code

```
  List<StandardColumnConfig> colDesc = generateColumnsConfiguration();
  MESStandardGrid mesGrid =
                  PhaseSwingHelper.createAndConfigureStandardGrid(getStatus(),
                  UserTestModel.class, colDesc);
  mesGrid.startup(); // Start up grid (after all configuration items are set).
  // set data
  List<UserModel> allUserModels = ... // e.g. get data out of a filter
  mesGrid.setObjects(allUserModels);
...
/**
 * Generate a list containing the info for the grid columns.
  * @return a list
 */
private List<StandardColumnConfig> generateColumnsConfiguration() {
  List<StandardColumnConfig> columns = new ArrayList<StandardColumnConfig>();
  StandardColumnConfig firstNameInfo = new StandardColumnConfig("firstName",
                                       SMALL_COLUMN_WIDTH, LineType.MULTI_LINE);
  firstNameInfo.setAccessCode("user.firstName");
  firstNameInfo.setDataDictClass(User.class.getName());
  columns.add(firstNameInfo);
  StandardColumnConfig lastNameInfo = new StandardColumnConfig("lastName",
                                      BIG_COLUMN_WIDTH, LineType.MULTI_LINE);
  lastNameInfo.setAccessCode("user.lastName");
  lastNameInfo.setDataDictClass(User.class.getName());
  columns.add(lastNameInfo);
  columns.add(new StandardColumnConfig("familyStatus", SMALL_COLUMN_WIDTH,
                                       LineType.SINGLE_LINE));
  columns.add(new StandardColumnConfig("bodyHeight",  MEDIUM_COLUMN_WIDTH,
                                       LineType.MULTI_LINE));
  columns.add(new StandardColumnConfig("birthDay", SMALL_COLUMN_WIDTH,
                                       LineType.SINGLE_LINE, SwingConstants.RIGHT));
  return columns;
}
```

- Display all of the properties of a class that uses the default configuration
  If the properties are not hidden in the corresponding data dictionary and the default configuration of the columns is sufficient for your purposes, use the helper method described below. The default configuration of a column is the reference to the data dictionary and center alignment.
  The *MESStandardGrid*, in the example below, displays all users. The bound class is the *User* class. The data will be fetched with the *UserFilter*.
  The helper method provided by the *PhaseSwingHelper* class creates, completely configures, and fills an *MESStandardGrid*. The helper method uses introspection of the bound class to detect the columns to be displayed and the data dictionary to configure the corresponding properties. The given list of data will be filled into the grid:

Example code

```
final JPanel layoutPanel =
    PhaseSwingHelper.createPanel(Layout.LAYOUT_SINGLE_COLUMN, getStatus());
...
// Get the data to be displayed in the grid:
UserFilter userFilter = new UserFilter(PCContext.getServerImpl());
List<User> allUsers = userFilter.exec();
// Create, configure and fill the grid:
MESStandardGrid mesStdGrid = PhaseSwingHelper.createFilledMESStandardGrid(
                             getStatus(), User.class, allUsers);
// Add the grid to the phase panel:
layoutPanel.add(mesStdGrid, Column.FIRST_COLUMN);
```

- Display some of the properties of a class that uses the default configuration
  If you do not wish to display all properties of the class or want their sequence to be changed, you can provide a list of properties:

Example code

```
final JPanel layoutPanel =
    PhaseSwingHelper.createPanel(Layout.LAYOUT_SINGLE_COLUMN, getStatus());
...
// Get the data to be displayed in the grid:
UserFilter userFilter = new UserFilter(PCContext.getServerImpl());
List<User> allUsers = userFilter.exec();
// The list of properties/columns to be displayed in the list:
List<String> properties = Arrays.asList("firstName", "lastName");
MESStandardGrid mesStdGrid = PhaseSwingHelper.createFilledMESStandardGrid(
                             getStatus(), User.class, allUsers, properties);
// Add the grid to the phase panel:
layoutPanel.add(mesStdGrid, Column.FIRST_COLUMN);
```

- Display properties of a referenced class
  Columns to be displayed in an *MESStandardGrid* need corresponding getter methods on the bound class. To display properties of a referenced class as data, you have to configure the access code of the column descriptor appropriately. For

example, your bound class references a user and has a getter for the user, and *firstName* is the property of the user to be displayed. Then your access code is *User.firstName*.
In the example below, the bound class *UserTestModel* has its own properties in addition to the reference to a user.

Example code

```
public class UserTestModel implements IMESBasePropertyChangeSupport {
  public Long getFamilyStatus() {
    return (Long) getUDA("X_familyStatus");
  }
  public User getUser() {
    return user;
  }
...
}
```

This is an example description for the creation of an *MESStandardGrid* which displays referenced data using the helper methods of the *PhaseSwingHelper* class.

1. The *createMESStandardGrid(Status status, Class boundClass)* method creates an *MESStandardGrid* and sets grid properties for the given class (here: sortable, without scrollbars).
2. The *configureStandardGridWithIntroSpection(IMESStandardGrid grid, Class boundClass, List<String> propertyNames)* method configures the columns by using introspection on the bound class. For each given property, a column descriptor will be created and added to the grid as in the given list, even if the property is not contained in the bound class. The given list defines the sequence of the columns.
   For all properties of the bound class, the corresponding reference to the data dictionary will be set. For the properties not contained in the bound class, a column descriptor (without data dictionary reference) will be added. So, all columns are basically configured and you can change the column descriptors afterwards, if needed.
3. The list of objects of the bound class must be set on the grid.

Example code

```
MESStandardGrid mesGrid = PhaseSwingHelper.createMESStandardGrid(getStatus(),
                                                     UserTestModel.class);
List<String> properties = Arrays.asList("firstName", "familyStatus);
PhaseSwingHelper.configureStandardGridWithIntroSpection(
                 mesGrid, UserTestModel.class, properties);
// Configure referenced columns:
// the JideGridColumnDescriptor uses indexes starting from 1
JideGridColumnDescriptor descr = mesGrid.getGridColumnDescriptor(1);
descr.setAccessCode("user.firstName");
descr.setDataDictClass(User.class.getName());
descr.setDataDictElement("firstName");
mesGrid.startup(); // Start up grid (after all configuration items are set).
```

```
List<UserTestModel> dataList = createStandardGridData()
mesGrid.setObjects(dataList);
```

- Create an *MESStandardGrid* without helper methods
  If the helper methods are not appropriate for your purposes, you can create and configure an *MESStandardGrid* completely by yourself.

Example code

```
MESStandardGrid mesGrid = new MESStandardGrid();
mesGrid.setEnabled(Status.ACTIVE.equals(getStatus()));
mesGrid.setAllowMultiSelection(true);
final int gridRowHeight = 25;
mesGrid.setMinimalRowHeight(gridRowHeight);
mesGrid.setSortable(true);
// disable re-ordering of the columns (using drag and drop)
mesGrid.setReorderingAllowed(false);
mesGrid.setCreateScrollbar(false);
mesGrid.setWithSelectColumn(false);
mesGrid.setUseShortLabels(false);
// bind the grid to the specified
classmesGrid.setBoundClassName(UserModel.class.getName());

// add column descriptors:
int i = 1; // the JideGridColumnDescriptor uses indexes starting from 1
JideGridColumnDescriptor descr = new JideGridColumnDescriptor(i);
descr.setAccessCode("user.firstName");
descr.setDataDictClass(User.class.getName());
descr.setDataDictElement("firstName");
descr.setHorizontalAlignment(SwingConstants.RIGHT + 1);
mesGrid.setGridColumnDescriptor(i, descr);
...
i++;
descr = new JideGridColumnDescriptor(i);
descr.setAccessCode("user.firstName");
descr.setDataDictClass(UserModel.class.getName());
descr.setDataDictElement("familyStatus");
descr.setHorizontalAlignment(SwingConstants.CENTER + 1);
mesGrid.setGridColumnDescriptor(i, descr);
...
mesGrid.startup(); // Start up grid (after all configuration items are set).

// set data
List<UserModel> allUserModels = ... // e.g. get data out of a filter
mesGrid.setObjects(allUserModels);
```

Creating an MESFastGrid

➢ Does not apply to server-side phases running on the OE server.

To create an *MESFastGrid*, basically, perform the following steps:

1. Create an *MESFastGrid* object and set the grid properties you require (e.g. sortable, allow reordering). For configuration options see *IMESGridConfiguration*.

2. Set the class which is bound to the grid. The bound class is used to reference the data dictionary (e.g. the formatting of the property is configured in the data dictionary). Thus, the bound class is the default data dictionary class.
3. Provide the SQL **from** statement.
4. For each row, provide the column of the unique key.
5. For each column, provide a column descriptor consisting of the column name of the database table, the data dictionary reference, the data type, the alignment, etc.
6. Trigger the execution of the SQL statement to fill the grid with data.

We provide some helper methods in the *PhaseSwingHelper* class to support the creation and configuration of an *MESFastGrid*.

- Configure an *MESFastGrid* with introspection of the bound class
  The helper method provided by the *PhaseSwingHelper* class creates an *MESFastGrid*. The column descriptors will be created according to the given list of column names and property names. The property names are related to the bound class. For each given column/property, a column descriptor will be created and added to the grid as in the given list, even if the property is not contained in the bound class. The given list defines the sequence of the columns. For all of the properties of the bound class, the corresponding reference to the data dictionary will be set. The number of columns and properties must be equal.

```
String sql = " FROM APP_USER";
List<String> columnNames = Arrays.asList("APP_USER.first_name", "APP_USER.last_name",
                           "APP_USER.user_name");
List<String> propertyNames = Arrays.asList("firstName", "lastName", "name");
MESFastGrid mesGrid = PhaseSwingHelper.createFilledMESFastGrid(getStatus(), User.class,
                      sql, "APP_USER.user_key", columnNames, propertyNames);
```

- Extend the configuration of an *MESFastGrid* provided by helper methods
  If the configuration done by the helper methods of the *PhaseSwingHelper* class is not completely appropriate for your purposes, you can use them to create the basic configuration and add your own configuration afterwards.
  This code snippet shows how to use the helper methods and change the configuration of the columns to match your needs:

Example code

```
String sql = " FROM APP_USER" //
+ " LEFT OUTER JOIN UDA_User ON " //
+ "APP_USER.user_key = UDA_User.object_key AND " //
+ "APP_USER.site_num = UDA_User.site_num " //
+ "WHERE NOT UDA_User.X_bodyHeight_F IS NULL";

MESFastGrid mesGrid = PhaseSwingHelper.createMESFastGrid(getStatus(), User.class, sql,
                      "APP_USER.user_key");
List<String> columnNames = Arrays.asList("APP_USER.first_name", "APP_USER.last_name",
                           "UDA_User.X_familyStatus_I", "UDA_User.X_bodyHeight_F",
                           "UDA_User.X_birthDay_u");
```

```
List<String> propertyNames = Arrays.asList("firstName", "lastName", "familyStatus",
                            "bodyHeight", "birthDay");
PhaseSwingHelper.configureFastGridWithIntroSpection(mesGrid, User.class, columnNames,
                                                    propertyNames);
// Configure columns, which are not in Data Dictionary or are shall have
// a different configuration:
final int colFamilyStatus = 3;
final int colBirthDay = 5;
FastLaneReaderJideColumnDescriptor desc =
                                   mesGrid.getGridColumnDescriptor(colFamilyStatus);
desc.setHorizontalAlignment(SwingConstants.RIGHT + 1);
desc = mesGrid.getGridColumnDescriptor(colBirthDay);
desc.setDataType(IDataTypes.TYPE_DATETIME);
mesGrid.startup(); // Start up grid (after all configuration items are set).

mesGrid.fillGrid();
```

- Create an *MESFastGrid* without helper methods
  If the helper methods are not appropriate for your purposes, you can create and configure an *MESFastGrid* completely by yourself. The steps you need to perform are described at the beginning of this section.

Example code

```
// Creation of the MESFastGrid object + configuration
MESFastGrid mesGrid = new MESFastGrid();
mesGrid.setEnabled(Status.ACTIVE.equals(getStatus()));
mesGrid.setAllowMultiSelection(true);
final int gridRowHeight = 25;
mesGrid.setMinimalRowHeight(gridRowHeight);
mesGrid.setSortable(false);
// disable re-ordering of the columns (using drag and drop)
mesGrid.setReorderingAllowed(false);
mesGrid.setCreateScrollbar(true);
mesGrid.setWithSelectColumn(false);
mesGrid.setUseShortLabels(true);

// bind the grid to the specified class
mesGrid.setBoundClassName(User.class.getName());
// sql related settings
mesGrid.setKeyName("APP_USER.user_key");
mesGrid.setSQL(" FROM APP_USER" //
+ " LEFT OUTER JOIN UDA_User ON " //
+ "APP_USER.user_key = UDA_User.object_key AND " //
+ "APP_USER.site_num = UDA_User.site_num " //
)
int i = 1;
FastLaneReaderJideColumnDescriptor descr = new FastLaneReaderJideColumnDescriptor(i);
descr.setName("APP_USER.first_name");
descr.setDataType(IDataTypes.TYPE_STRING);
descr.setDataDictClass(User.class.getName());
descr.setDataDictElement("firstName");
descr.setHorizontalAlignment(SwingConstants.RIGHT + 1);
grid.setGridColumnDescriptor(i, descr);
...
i++;
descr = new FastLaneReaderJideColumnDescriptor(i);
descr.setName("UDA_User.X_familyStatus_I");
descr.setDataType(IDataTypes.TYPE_STRING);
descr.setDataDictClass(UserModel.class.getName());
descr.setDataDictElement("familyStatus");
descr.setHorizontalAlignment(SwingConstants.CENTER + 1);
grid.setGridColumnDescriptor(i, descr);

mesGrid.startup(); // Start up grid (after all configuration items are set).

// Execute the sql query and fill the result into the grid
grid.fillGrid();
```

TIP

If no data dictionary class is set on a column descriptor, the bound class is used.

Components for Completion

Typically in the phase GUI, there are one or more components that are used to trigger phase completion (see also "Completing a Runtime Phase in the Phase Executor" (page 64)). The **Confirm** button provided by the *PhaseColumnLayout* layout manager (page 56) is such a component. The *com.rockwell.mes.apps.ebr.ifc.phase.IPhaseExecutorWithCompletionComponents* interface allows you to provide the information which components can be used for completion. The phase-completion-signature UI extension processes the information as follows:

- If the operator signs for the completion of a phase and afterwards modifies anything within the phase (e.g. types a different value), then the password is deleted for security reasons. Additionally, the operator has to sign for the change.
- If the focus moves to a phase completion component, the password is not deleted. If the password were deleted, the operator would never be able to complete the phase.

AbstractPhaseExecutorSwing implements the interface and provides the *addComponentForCompletion* and *removeComponentForCompletion* methods to add or remove components to/from the list of completion components. Use the methods for all components that are available for phase completion. If you do not add any component, a different mechanism for password clearance is used for your phase: the password is deleted several seconds after leaving a signature box. The delay can be configured with the **Station/clearPasswordsDelay** configuration key (see chapter "Configuration Keys of PharmaSuite" in Volume 4 of the "Technical Manual Configuration and Extension" [A3] (page 175)).

Completing a Runtime Phase in the Phase Executor

The implementation of a phase executor has to provide the behavior of the phase during execution. Usually, this will be trigged by the user who operates the GUI provided by the phase.

If the phase executor is ready for completion, it has to call the *getPhaseCompleter().completePhase()* method to initiate the completion of the phase in the system. This is the implementation of *completePhase*:

```
if (phaseExecutor.performCompletionCheck() {
   phaseExecutor.performComplete();
   ...
   // the framework completes the phase in the system
}
```

To perform the completion of the phase, the phase executor has to implement two methods, which will be called by the framework:

- *performCompletionCheck()*
  This method is invoked before a phase has been completed in order to put a veto on phase completion or not. The method has to be implemented by each phase executor. If you do not wish to perform any checks and always allow the phase to complete, just return **true**. For an example on how to use this method in the context of limit checks, please refer to section "Adding Exception Handling for Phase-internal Checks" (page 135).
- *performComplete()*
  This method has to be implemented by each phase executor. The method is invoked on completion of the phase as a consequence of calling the *getPhaseCompleter().completePhase()* method and should be used to write the runtime phase data and the output of the phase, if present.
  Afterwards, the framework saves and completes the runtime phase including its associated phase data and output.

The phase executor has access to the runtime phase, to the runtime phase data, and the runtime phase output. Also, the phase executor is responsible for filling the data values of the runtime phase data and the runtime phase output.
The phase executor can save the data of the runtime phase before its completion. Before a runtime phase is completed, it can be saved multiple times using the *save()* method.
The phase executor must not call the *saveAndComplete()* method. This will be done by the framework. The framework saves the runtime phase including its associated phase data and output data and sets the runtime phase to complete. Afterwards, the data of the runtime phase must not be changed anymore.

TIP

- A phase developer can define several output values for a phase that can be used in expressions, e.g. in transition conditions. When using an output in an expression, the operator has to know the name of the output (to be able to reference it in the expression) and the type of the output (to be able to specify valid expressions). Recipe and Workflow Designer supports the operator to refer only to valid outputs of a phase when entering expressions.

- As a starting point for your implementation, you can, of course, use the skeleton implementation created by the phase generator (see "What Is the Phase Generator?" (page 19)).
- During phase completion, in the *performCompletionCheck()* and *performComplete()* methods, you can access the user name(s) recorded with the phase completion signature if any have been recorded:
  ```
  userName1 =
      getPhaseCompleter().getPerformerOfPhaseCompletionSignature();
  userName2 =
      getPhaseCompleter().getVerifierOfPhaseCompletionSignature();
  ```

It is not allowed to synchronously complete a phase in *createPhaseComponent()*. If you wish to implement an automatic completion of a phase, you have to call it using *invokeLater*.

Example: Automatic completion of a phase

```
Public JComponent createPhaseComponent() {
...
  // Implementing an auto complete of the phase
  SwingUtilities.invokeLater(new Runnable() {
    public void run() {
      getPhaseCompleter().completePhase();
    }
  });
...
}
```

Creating the Runtime Phase Data

A phase can create runtime phase data at any time during execution within its phase executor. Additionally, a phase can create multiple runtime phase data objects (records), since there is a 1:n relation between a runtime phase and its runtime phase data.

This is an example code snippet for the creation of a runtime phase data object:

```
// create a new runtime phase data object (record)
MESRtPhaseDataMYC_StringValue data = addRtPhaseData();
// set the runtime phase data value(s)
data.setActualValue(text);
```

In the phase executor skeleton, the phase generator generates the *addRtPhaseData()* and *getRtPhaseData()* methods to handle runtime phase data.

According to the needs and the nature of your phase you can immediately save your newly created runtime phase data object by calling one of the *save* methods of the runtime phase:

```
try {
  getRtPhase().Save(null, null, PCContext.getDefaultAccessPrivilege());
} catch (DatasweepException e) {
```

```
  throw new MESRuntimeException("Failed to save runtime phase", e);
}
```

If it is not necessary to save your runtime phase data immediately, you can omit the intermediate *save* calls. Saving all of your runtime phase data including output data and setting the runtime phase to complete will be done by the framework during phase completion after *performComplete()*.

Creating the Runtime Phase Output

The *performComplete()* method is also the right place to create the runtime phase output.

This is an example (not related to our **MYC_StringValue** example phase outlined below) for the creation of the phase output. It creates two outputs (each property of the output bean is considered as a separate output), but no runtime phase data.

```
public Response performComplete() {
  // Get the corresponding instance of your output.
  MESRtPhaseOutputMYC_Phase output = getRtPhaseOutput();

  // set your output variables / attributes
  output.setBatchIdentifier("batch");
  output.setSamplingType("Sampling type");

  return new Response();
}
```

In the phase executor skeleton, the phase generator generates the *getRtPhaseOutput()* method for phases with outputs. The method creates (or loads) the phase-specific output record. The record is associated with the runtime phase of the phase executor and must be filled in the *performComplete()* method with the actual output values.

TIP

- If an output is specified, the phase has to set values of the output bean. The framework will check this after phase completion; otherwise an exception will be thrown.
- In contrast to the runtime phase data, we do not need an *addRtPhaseOutput()* method since a runtime phase must have no more than one associated output record.

Using FactoryTalk ProductionCentre Controls instead of Swing Components

➢ Does not apply to server-side phases running on the OE server.

Phases are created by means of Swing components. If you need to use FactoryTalk ProductionCentre controls for your phase, you have to use the *AbstractPhaseExecutor* base class instead of *AbstractPhaseExecutorSwing*.

In this case, you also must implement methods with a different signature-specific method for the user interface. Use *createPhaseCControl* instead of *createPhaseComponent*. The same applies to exceptions and actions.

To support the implementation of a phase control, the following set of helper classes is available in the PharmaSuite Building Block SDK:

- PhaseWorkAreaContainer
  Offers standard layouts for phases.
  The **Confirm** button can be added automatically with *addCompleteButton()*.
  The container contains the following panels:
  - WorkAreaPanel
    The panel provides various column layouts for phases with 1, 2, or 3 columns (see *UIConstants.ProductPhaseLayoutType* below).
  - FillPanel
    The panel is located below the *WorkAreaPanel* panel and can be used to display additional data.
  - BottomPanel
    The panel is located at the bottom of the phase layout and can be used to display additional data (e.g. signatures).
  - PhaseWorkAreaPanel
    A panel which contains several sub-panels for phases with 1, 2, or 3 columns (see *UIConstants.ProductPhaseLayoutType* below).
    Used by *PhaseWorkAreaContainer*.
  - PhasePanel
    Wrapper for the Panel of FactoryTalk ProductionCentre. Simplified usage since it already contains attributes needed each time in the constructor.
    Updates its height (and also the height of the phase) automatically, depending on its content.
  - PhaseLabel
    Wrapper for the FlatLabel of FactoryTalk ProductionCentre. Simplified usage since it already contains attributes needed each time in the constructor.
    Calculates the preferred height depending on the content.
  - PhaseAlphaNumericTouchField, PhaseNumericTouchField
    Wrappers for the TouchFields of PharmaSuite, using a virtual keyboard for user input. Simplified usage since it already contains attributes needed each time in the constructor.
  - PhaseCheckBox
    Wrapper for the CheckBox of FactoryTalk ProductionCentre. Simplified usage since it already contains attributes needed each time in the constructor.

- PhaseButton
  Wrapper for the *GraphicalButtonActivity* in order to create a button for the phase. To create a **Confirm** button, call *addDefaultListener(final IPhaseCompleter phaseCompleter)*.
- UIConstants
  Some constants useful for developing the phase GUI.
- PhaseControlHelper
  Some constants and the *setTransparent* helper method, useful for developing the phase GUI.
- PhaseTableHelper
  Component `createGridControl(List<? extends IMESATObject> gridData, Class beanClass, Status status)`
  creates a grid control. The rows of the grid are given as list of *IMESATObjects*. *beanClass* describes the columns to be displayed.

Instead of using the *PhaseTableHelper*, you can also create your grid manually. For more information, see section "Grids" in chapter "Creating and Adapting Forms" in Volume 1 of the "Technical Manual Configuration and Extension" [A1] (page 175). For this purpose, use the CComponent GridActivity or FastGridActivity to create a grid (which is more comfortable than a JTable) either based on a TableModel or by calling *setObjects()* manually (with MYC for the My Company vendor code):

```
TableModel tableModel = new MYC_TableModel();
GridActivity matTable = new GridActivity(tableModel); // CControl

matTable.setName("MYC_Grid");
matTable.setAllowMultiSelection(false);
matTable.setAllowUserSelection(true);
matTable.setAutoSelectFirstRow(false);
matTable.setEnabled(getStatus() == Status.ACTIVE);

matTable.setDock(ControlDock.FILL);
mainPanel.add(matTable);
```

Specific Information on Phases Running on the PharmaSuite OE Server

In contrast to phases with a GUI, which run in a Production Execution Client environment, when creating phases that run in a PharmaSuite OE server environment (server-run phases) the following information must be considered:

- You have to set the *IsServerRunPhase* flag in the XML description of the phase.
- There is no GUI and no user interaction during execution.
- Their phase executor is inherited from *AbstractServerSidePhaseExecutor* instead of *AbstractPhaseExecutor*.

- The *AbstractServerSidePhaseExecutor* provides a different set of methods of the *AbstractPhaseExecutor*.
  Methods related to GUI and focus handling are not available.
  A method was added to schedule a thread in a managed thread pool provided by PharmaSuite: *AbstractServerSidePhaseExecutor:: <V> ScheduledFuture<V> schedule(final Callable<V> furtherExecution, MESDuration delay)*.
  All server-side phases run within a fixed-size thread pool. That means blocking plugins may prevent processing of other phases. Therefore it is important to observe the following design rules:
  - The phase must not perform long-running processing tasks. If this is absolutely required, do not use the general executor, but create your own.
  - The phase must not block.
- If your phase is used in a parallel processing environment, you have to treat threading aspects with great care. An example for this can be that your phase reacts on messages.
  The first aspect is **concurrent modification**. Please note that **MESATObjects** are not thread-safe by default. If you use more than one thread to perform actions, e.g. on **MESRtPhase**, you have to synchronize the actions in the phase code. However, this might lead to the second aspect which is **deadlocks**. When adding synchronization, you have to be careful not to introduce deadlocks.
- To record a system-triggered exception without user interaction use *PhaseSystemTriggeredExceptionHandler::recordException(IServerSidePhaseExecutor, msg, riskClass)*.
- The *AbstractServerSidePhaseExecutor* provides a method to handle unexpected exceptions. The method allows to send an error message to Production Execution Clients running at the work centers of a unit procedure:
  handleUnexpectedErrorOnServer(final String errorText, final Throwable causeOfTheProblem)
  errorText is extended by the available context information (e.g. runtime phase, order). If a *Throwable* is given, the stack trace is added.
- In this manual, sections that do not apply to server-run phases start with: Does not apply to server-side phases running on the OE server.
- Unless specified otherwise examples in this manual that use *AbstractPhaseExecutor* as base class are also valid for server-run phases using *AbstractServerSidePhaseExecutor* base class instead.
- For details on how to debug the OE server and the phases running on it, see section "Debugging PharmaSuite Event Sheets" in chapter "Monitoring PharmaSuite and Related Components" in "Technical Manual Administration" [A4] (page 175).

Example

The example applies to a phase with a GUI running in a Production Execution Client environment.

In our **MYC_StringValue** example, we will display a label containing the name of the phase, a text box for a string and a button (see *createPhaseComponent()* in the example code below). When an operator clicks the button, the entered value will be stored in a private field for saving, the phase will be completed (by calling *getPhaseCompleter().completePhase()*), and will thus trigger the successor phase. Before completion, the value in the text box will be saved as both runtime phase data and phase output (using the value stored when clicking the button, see *performComplete()* in the example code below). After *performComplete()* has been finished, the status of the runtime phase is **Completed**.

TIP

If we wish to use our **batchIdentifier** output (see example code above (page 67)) in a transition condition, the formula of the transition could look as follows:
`{Name of phase}.{Batch identifier} == "Batch1"`
(provided that **Batch identifier** is the **DisplayName** of the **batchIdentifier** output, see "Step 3: Creating an XML Description of the Runtime Phase Output" (page 45)).

Example

```
import java.awt.GridBagLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JComponent;
import javax.swing.JLabel;
import javax.swing.JTextField;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import com.datasweep.compatibility.client.ActivitySetStep;
import com.datasweep.compatibility.client.Response;
import com.rockwell.mes.apps.ebr.ifc.phase.IPhaseCompleter;
import com.rockwell.mes.apps.ebr.ifc.swing.AbstractPhaseExecutorSwing;
import com.rockwell.mes.apps.ebr.ifc.swing.PhaseColumnLayout;
import com.rockwell.mes.apps.ebr.ifc.swing.PhaseColumnLayout.Column;
import com.rockwell.mes.apps.ebr.ifc.swing.PhaseColumnLayout.Layout;
import com.rockwell.mes.commons.base.ifc.services.PCContext;
import com.rockwell.mes.services.s88.ifc.execution.IMESRtPhase;
import com.rockwell.mes.services.s88.ifc.recipe.IMESPhase;

/**
 * Implementation of simple phase allowing to enter a string. Based on
 * generation.
 */
public class RtPhaseExecutorPAVStr extends AbstractPhaseExecutorSwing {
  /** Logger for logging */
  private static final Log LOGGER = LogFactory.getLog(RtPhaseExecutorPAVStr.class);
  /** the inserted string */
  protected volatile String text = "";
  /** the input field */
  private JTextField edit;

  public RtPhaseExecutorPAVStr(IPhaseCompleter inPhaseCompleter, IMESRtPhase inRtPhase) {
    super(inPhaseCompleter, inRtPhase);
  }

  public RtPhaseExecutorPAVStr(IMESPhase inPhase, ActivitySetStep inStep) {
    super(inPhase, inStep);
  }

  /**
   * REMARK: This method is called when the phase is rendered by the framework.
   * {@inheritDoc}
   *
   * @see com.rockwell.mes.apps.ebr.ifc.swing.AbstractPhaseExecutorSwing#
                                            createPhaseComponent()
   */
  @Override
  public JComponent createPhaseComponent() {
    final JPanel layoutPanel = PhaseSwingHelper.createPanel(
                             Layout.LAYOUT_TWO_1ST_WIDER_COLUMN, getStatus());
    JLabel label = PhaseSwingHelper.createJLabel("Enter 'EXCEPTION' to trigger an
                                                 exception.");
    label.setToolTipText(getPhase().getName());
    label.setPreferredSize(new Dimension(UIConstants.LABEL_PREFERRED_WIDTH_FILL,
                         UIConstants.LABEL_PREFERRED_SIZE.height));
    JPanel column1 = PhaseSwingHelper.createPanel();
    column1.setLayout(new BorderLayout());
    column1.add(label, BorderLayout.NORTH);
    layoutPanel.add(column1, Column.FIRST_COLUMN);
    edit = PhaseSwingHelper.createJTextField(getStatus());
    final JPanel column2 = PhaseSwingHelper.createPanel();
    column2.setLayout(new BorderLayout());
    layoutPanel.add(column2, Column.SECOND_COLUMN);
    column2.add(edit, BorderLayout.NORTH);

    edit.setName("ExecutionValueInputTextField");

    if (getStatus() == Status.ACTIVE) {
      /** confirm button listener */
      class ConfirmButtonListener implements ActionListener {
        @Override
        public void actionPerformed(ActionEvent event) {
          text = edit.getText();
          LOGGER.debug("Button clicked:" + (text == null ? "" : text));
          getPhaseCompleter().completePhase();
        }
      }
      ConfirmButton confirmButton =
                  ((PhaseColumnLayout) layoutPanel.getLayout()).getConfirmButton();
      confirmButton.addActionListener(new ConfirmButtonListener());
      addComponentForCompletion(confirmButton);
    }

    if (getStatus() == Status.COMPLETED) {
      MESRtPhaseDataPAVStr data = getRtPhaseData();
      edit.setText(data.getActualValue());
    }
    return layoutPanel;
  }

  @Override
  public boolean performCompletionCheck() {
    return true;
  }
```

```
    /**
     * REMARK: This method is called when the complete() method has been called
     * by this class. Normally this is when the user has pressed the complete
     * button. In this case the performCompletionCheck() is called. If the check
     * returns true this method is called.
     * {@inheritDoc}
     *
     * @see com.rockwell.mes.apps.ebr.ifc.phase.IPhaseExecutor#performComplete()
     */
    @Override
    public Response performComplete() {
      // Set runtime phase data
      LOGGER.debug("Inserted runtime phase data: " + text);
      MESRtPhaseDataPAVStr data = addRtPhaseData();
      data.setActualValue(text);
      // Set runtime phase output
      MESRtPhaseOutputPAVStr output = getRtPhaseOutput();
      output.setActualValue(text);

      return new Response();
    }

    /**
     * {@inheritDoc}
     *
     * @see com.rockwell.mes.apps.ebr.ifc.phase.AbstractPhaseExecutor#
                                                getNavigatorInfoColumn()
     */
    @Override
    public String getNavigatorInfoColumn() {
        MESRtPhaseDataPAVStr data = getRtPhaseData();
      return data.getActualValue();
    }
  }
```

Step 6: Adding Optional Features (Java Code)

This step describes optional features you can implement when the required Java code is available. The features apply to

- navigation (page 74),
- barcode support (page 74),
- default rendering of the phase background (page 76),
- exception marker of phases with exceptions (page 76),
- default focus component/control for selecting phases (Swing components (page 78), FactoryTalk ProductionCentre controls (page 80)),
- event handling when UI components become visible or invisible (page 80),
- callback methods related to the life cycle of a phase building block (page 81),
- hiding phases after completion (page 82),

- disabling phase UI extensions (page 83),
- customizing phase UI extensions (page 83),
- supporting automatic confirmation of a phase completion signature in the context of external authentication (page 84),
- providing dynamic phase outputs (page 84), and
- error handling (page 87).

in the Execution Window.

Navigation

Usually, a phase has a very local context. That means, all information needed to process and complete the phase is available via the API of the phase executor. However, sometimes it can be necessary to access other data as well. The PharmaSuite API provides methods to navigate within a recipe.

We highly recommend to navigate within the runtime entities first before using the recipe entities. You must not assume that a recipe entity referenced by your runtime phase is unique or local to your control recipe. Instead it is most likely a reference to a master recipe entity and therefore shared by other orders.

Keep this in mind when you implement, for instance, phase-specific SQL statements.

Adding Barcode Support

➢ Does not apply to server-side phases running on the OE server.

Input boxes can consume input typed by an operator or scanned barcodes. There are circumstances when a scanned barcode is preferred to feed an input field or to trigger an action automatically. In this case, you do not necessarily need a UI component to visualize the scan event.

Consider the following aspects when you plan to handle barcode scan events and do not make use of the PharmaSuite platform:

- Different and independent processes and workflows may run in parallel. Only the active one shall receive barcode scan events. The active one is the one that displays an active UI. You must ensure that hidden processes and workflows do not receive barcode events.
- If barcode scanning is disabled by a process or workflow, barcode scans must be swallowed. Otherwise they would appear as keyboard input, which is normally not desired.
- If barcode scanning is enabled by a process or workflow, barcode scans must be swallowed in case a modal dialog is displayed.
- Different types of barcode labels may be used within a process or workflow. Associated with the different types of barcode labels, you may have multiple

consumers for the barcodes. This leads to the question, which consumer shall receive the barcode or how the barcode event shall be dispatched?
Example: You scan a 2D barcode associated with a button click or a sublot barcode label. If a processor responsible for sublot barcodes receives the 2D barcode of an action button, it interprets it as a sublot barcode with missing data and displays a corresponding message.

PharmaSuite populates barcode scan events as a *PropertyChangeEvent* to a *PropertyChangeListener*, which can be registered at a central *BarcodeListener* instance. But, usually an application does not talk directly to the *BarcodeListener*. An implementation of *IBarcodeListenerProxy* is used instead. This proxy is associated with a UI component and enables/disables the registered *PropertyChangeListener* if the component is shown/hidden.

The following implementations for *IBarcodeListenerProxy* are available:

- *BarcodeListenerProxy*, associated with a FactoryTalk ProductionCentre form and
- *BarcodeListenerSwingProxy*, associated with a *JComponent*.

To solve the dispatching issue, register an *IConsumablePropertyChangeListener* instead of a *PropertyChangeListener*. An *IConsumablePropertyChangeListener* is an extension of *PropertyChangeListener* and additionally, offers the *boolean consumablePropertyChange(PropertyChangeEvent evt)* method.

When firing *PropertyChangeEvents*, the *BarcodeListener* will first invoke *consumePropertyChange* to all listeners implementing the *IConsumablePropertyChangeListener*. As soon as a listener consumes the event by returning *true*, the propagation to other listeners stops. If no listener indicates the consumption of the event, the *propertyChange* method will be invoked on all registered listeners.

The *BarcodeListenerSwingProxy* class allows to register the following interfaces:

- *IBarcodeScannedListener* instead of *PropertyChangeListener* and
- *IConsumableBarcodeScannedListener* instead of *IConsumablePropertyChangeListener*.

Both interfaces simply hide the *PropertyChangeEvent* and offer the following methods:

- *void barcodeScanned(String barcode)* or
- *boolean consumableBarcodeScanned(String barcode)*.

The simplified and fragmented example below demonstrates how the *GraphicalButton* class uses the barcode handler classes. In particular, it makes use of *IConsumableBarcodeScannedListener*. The *GraphicalButton* class is used for all action buttons within a phase.

Example

```
package com.rockwell.mes.clientfw.commons.ifc.swing;

public class GraphicalButton extends JButton implements IConsumableBarcodeScannedListener,
             AncestorListener {
  /** The barcode listener */
  private BarcodeListenerSwingProxy barcodeListener;

  public BarcodeListenerSwingProxy(){
    addAncestorListener(this);
  }

  @Override
  public void ancestorAdded(AncestorEvent event) {
    if (BarcodeType.ActiveBarcode.equals(barcodeType)) {
      barcodeListener = new BarcodeListenerSwingProxy(this);
      barcodeListener.register();
      barcodeListener.addBarcodeScannedListener(this);
  } }

  @Override
  public void ancestorRemoved(AncestorEvent event) {
    if (barcodeListener != null) {
      barcodeListener.unregister();
      barcodeListener.removeBarcodeScannedListener(this);
      barcodeListener = null;
  } }

  @Override
  public void barcodeScanned(String barcode) { }

  @Override
  public boolean consumableBarcodeScanned(String barcode) {
    // check if button is active
    if (!isEnabled()) return false;

    // fire CLICK event if button is active and the barcode equals barcodeButtonName
    if ("expected barcode".equals(barcode)) {
      doClick();
      return true;
    }
    return false;
  }
}
```

Changing Background or Exception Rendering

➢ Does not apply to server-side phases running on the OE server.

To change the background or exception rendering of a phase, you have to overwrite *PhasePanelUI* and use it in an **opaque** panel.

TIP

When you implement these changes your phase may no longer match the general look and feel of PharmaSuite.

Please be aware that a customized phase UI may partially or completely cover the default exception marker. In this case, we recommend to adapt the layout.

The figures below display the default rendering and an example of an adapted rendering of phases in the Execution Window.

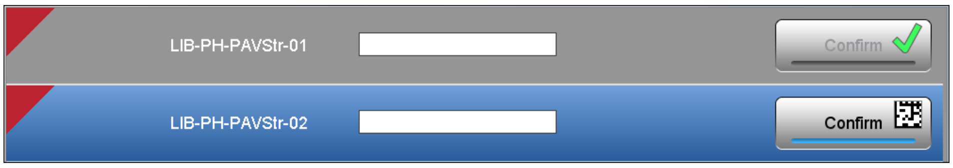


Figure 16: Default rendering of phases in the Execution Window

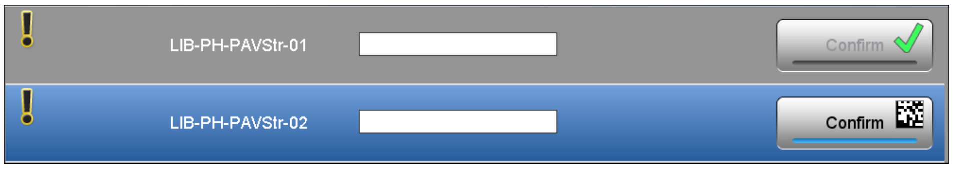


Figure 17: Example of adapted rendering of phases in the Execution Window

TIP

You can change the color of the default exception marker (red upper left corner) by adapting the PEC style sheet, 'PhasePanelUI' section. For information about PharmaSuite style sheets, see chapter "Changing the General Appearance of PharmaSuite" in Volume 1 of the "Technical Manual Configuration and Extension" [A1] (page 175).

Example

As a first step, you need to create a custom *PanelUI* (here: **PhasePanelUIWithIcon**) by deriving from the *PhasePanelUI* class and overwriting the methods you wish to change (e.g. *paintExceptionMarkup(...)*).

```
public class PhasePanelUIWithIcon extends PhasePanelUI {

  @Override
  public void paintExceptionMarkup(Graphics g, JComponent c) {
    ImageIcon exceptionMarkup = PCContext.getFunctions().getImage("message_warning_48");
    g.drawImage(exceptionMarkup.getImage(), 0, 0, null);
  }
}
```

Then, you have to set the new *PanelUI* to the opaque panel created by your phase executor.

```
@Override
public JComponent createPhaseComponent() {
  final JPanel layoutPanel =
    PhaseSwingHelper.createPanel(Layout.LAYOUT_TWO_1ST_WIDER_COLUMN, getStatus());
  layoutPanel.setOpaque(true);
  layoutPanel.setUI(new PhasePanelUIWithIcon(this));
  ...
```

Changing the Focused Component

➢ Does not apply to server-side phases running on the OE server.

By default, the framework provides a focus traversal policy. The default focus traversal policy is a *LayoutFocusTraversalPolicy* which sorts components based on their size, position, and orientation. If your phase component provides a **Confirm** button that is a *ConfirmButton* component (see *ConfirmButton* in "Available Swing Components" (page 53)) or the name of your **Confirm** button has the default name for **Confirm** buttons (see *PhaseSwingHelper* in "Available Swing Components" (page 53)), the default focus is set to the **Confirm** button. Otherwise, the first component receives the focus, since it is the default focus component of the original Swing *LayoutFocusTraversalPolicy*.

You have two options to influence the focus behavior of your phase component:

- Change the default focus component (page 78)
- Create your own focus traversal policy (page 78)

Changing the Default Focus Component

To change the default focus component, you have to set the focus traversal policy of your phase component and set it as *Focus Traversal Policy Provider*. The latter is essential to activate your focus traversal policy. Set the focus traversal policy to an object of the *SwingPhasePanelFocusTraversalPolicy*. The constructor expects the component to get the default focus.

Example

In this example, the default focus component is set to an edit field contained in the phase component. Adapt the following code example to change to focused component.

```
@Override
public JComponent createPhaseComponent() {
  final JPanel layoutPanel =
      PhaseSwingHelper.createPanel(Layout.LAYOUT_SINGLE_COLUMN, getStatus());
...
  layoutPanel.setFocusTraversalPolicy(new SwingPhasePanelFocusTraversalPolicy(edit));
  layoutPanel.setFocusTraversalPolicyProvider(true);
...
}
```

Creating an Own Focus Traversal Policy

If the default focus traversal policy is not appropriate for your phase, you can define an own focus traversal policy and set it on the phase component. Create a subclass of the *FocusTraversalPolicy* abstract class and set it to your phase component. It is important to set your phase component as Focus Traversal Policy Provider in order to become effective.

```
@Override
public JComponent createPhaseComponent() {
JPanel layoutPanel =
  PhaseSwingHelper.createPanel(Layout.LAYOUT_SINGLE_COLUMN, getStatus());
...
layoutPanel.setFocusTraversalPolicy(new MYC_FocusTraversalPolicy());
layoutPanel.setFocusTraversalPolicyProvider(true);
...
}
```

The framework provides a simple focus traversal policy (*ListFocusTraversalPolicy*). The constructor expects a list of components for the focus traversal in the desired order. This focus traversal policy is especially useful, if you need a focus traversal policy for components located in different panels.

Example code

```
@Override
public JComponent createPhaseComponent() {
final JPanel layoutPanel =
      PhaseSwingHelper.createPanel(Layout.LAYOUT_TWO_2ND_WIDER_COLUMN, getStatus());
...
inputField1 = PhaseSwingHelper.createJTextField(getStatus());
inputField2 = PhaseSwingHelper.createJTextField(getStatus());
inputField3 = PhaseSwingHelper.createJTextField(getStatus());
inputField4 = PhaseSwingHelper.createJTextField(getStatus());
JPanel column1 = PhaseSwingHelper.createPanel();
column1.setLayout(new BoxLayout(aPanel, BoxLayout.PAGE_AXIS));
column1.add(textField1);
column1.add(textField2);
JPanel column2 = PhaseSwingHelper.createPanel();
column2.setLayout(new BoxLayout(aPanel, BoxLayout.PAGE_AXIS));
column2.add(textField3);
column2.add(textField4);
layoutPanel.add(column1, Column.FIRST_COLUMN);
layoutPanel.add(column2, Column.SECOND_COLUMN);
...
ListFocusTraversalPolicy ftp = new ListFocusTraversalPolicy(Arrays.asList(inputField1,
                                   inputField2, inputField3, inputField4));
layoutPanel.setFocusTraversalPolicy(ftp);
layoutPanel.setFocusTraversalPolicyProvider(true);
...
}
```

TIP

- If your default focus component is the last component of your tab order, entering a tab will leave your phase. The next time your phase component will get the focus again, the default component will get the focus and not the first component. That is why the components listed prior to the default component (in your tab order) are not accessible by the TAB key. We recommend to have the default focus component as the first component of your tab order.
- *AbstractPhaseExecutor.isFocused()* may be used to check whether the phase is focused in the execution view. This is especially useful in combination with

barcode input.

- If you use an *inputVerifier* on a component, please ensure that the focus can leave the field (see section "Step 2: Triggering the Check during Phase Execution" (page 136) for more details about the use of *inputVerifier* and focus handling) in case the component is not visible on the screen. Otherwise the focus can remain on the field even if the field is not visible because another view (e.g. the Navigator) is displayed.

Changing the Focused Control in a FactoryTalk ProductionCentre-based Phase

➢ Does not apply to server-side phases running on the OE server.

The focus traversal policy of a phase based on FactoryTalk ProductionCentre controls is defined by the tab order on FactoryTalk ProductionCentre controls. If your phase component provides the **Confirm** button out of the *PhaseButtonHelper* class or the name of your **Confirm** button has the default name for **Confirm** buttons (see *PhaseButtonHelper.getConfirmButtonActivityName()*), the default focus is set to the **Confirm** button. Otherwise, the first control defined by your tab order is the focused control.
If you want to define another control as the default focus control, you have to set a specific activity name.

EXAMPLE

Adapt the following code example to change to focused control.

```
public PhaseControl createPhaseCControl() {
  PhaseControl mainPanel = createMainPanel();
  ...
  // Setup the control to be used as default for focusing
  edit.setActivityName(CCPhasePanelFocusTraversalPolicy.NAME_CONTROL_DEFAULT_FOCUS);
```

Event Handling when UI Components Become Visible or Invisible

➢ Does not apply to server-side phases running on the OE server.

In some cases the phase executor may wish to perform actions when UI components, created by the phase executor, become visible or invisible, for example if an operator switches to another view like the Cockpit. For this purpose, you can use the Swing *AncestorListener*.

TIP

If you wish to react to barcode scanner events depending on the visibility, then you should use the *BarcodeListenerSwingProxy* instead.

The difference between the visibility events and the *createPhaseComponent* method is that the events may be invoked multiple times whenever an operator switches the view.

You can also use the same listener approach for the other UI components of the exception or action view.

TIP

Avoid long running middle tier calls in these events to ensure best user experience.

With respect to the phase execution UI, the events are mostly needed for the **Active** status. Therefore, you should register the listener only if the UI is created for the **Active** status.

EXAMPLE

```
@Override
public JComponent createPhaseExceptionComponent() {
  ...
  exceptionsPanel.addAncestorListener(new AncestorListener() {
    @Override
    public void ancestorAdded(AncestorEvent event) {
      LOGGER.debug("Exception UI becomes visible");
    }

    @Override
    public void ancestorRemoved(AncestorEvent event) {
      LOGGER.debug("Exception UI becomes invisible");
    }

    @Override
    public void ancestorMoved(AncestorEvent event) {
      // Ignore
    }
  });
  return exceptionsPanel;
}
```

Callback Methods Related to the Life Cycle of a Phase Building Block

The regular life cycle of a phase (or runtime phase) starts when the phase is transitioned into the **Active** status. After the business logic has been executed, the phase's status is changed to **Completed**. However, beyond its regular life cycle, there are events that may cause a phase to be still in the **Active** status although its execution has been stopped:

- client shutdown,
- detaching an operation, and
- canceling an operation.

The following callback methods related to the life cycle of a phase can be overridden if needed:

- *startPhaseExecution* callback
  The method is called after the phase's status is changed to **Active** and the GUI of

the active phase has been created, if it has a GUI at all (see *createPhaseComponent()* in section "Providing the GUI" (page 51)).

- *holdPhaseExecution* callback
  The method is called when the operation is put on hold while the phase is still in the **Active** status, so on shutdown of the client and on detach of the operation. This means that the instance of the runtime phase may run again later when the operation is resumed.
- *cancelPhaseExecution* callback
  The method is called when the operation is canceled while the phase is still in the **Active** status. This means that the instance of the runtime phase will never run again.

The callback methods are helpful if a phase uses external resources during processing (e.g. to establish a connection to a hardware device when the phase's status is changed to **Active**). The phase normally frees the resources when it is moved to the **Completed** status. However, if the operation is put on hold or canceled while a phase is still active, you may also have to free external resources.

EXAMPLE

```
@Override
public void startPhaseExecution() {
  LOGGER.debug("Start phase. Establishing connection to device.");
  // Here you can create resources, open connections etc.
}

@Override
public void holdPhaseExecution() {
  LOGGER.debug("Hold phase. Closing connection to device.");
  // Here you can free resources, close connections etc.
}

@Override
public void cancelPhaseExecution() {
  LOGGER.debug("Cancel phase. Closing connection to device.");
  // Here you can free resources, close connections etc.
}

@Override
public Response performComplete() {
...
  LOGGER.debug("Complete phase. Closing connection to device.");
  // Here you can free resources, close connections etc.
...
}
```

Hiding a Phase after Completion

➤ Does not apply to server-side phases running on the OE server.

Some phases are skipped during execution. Usually, this happens when the automatic completion feature of a phase is used. In most cases, skipped phases cannot render themselves in the **Completed** status since the data they need for the representation is not available. From a business perspective, they do not have valuable information anyway.

Skipped phases will not be displayed in the following views:

- execution view (as completed phases),
- Navigator (as list item), and
- batch report (as phase sub-report).

To hide a phase after completion, you have to set the *hideAfterCompletion* flag on the runtime phase during the completion of a phase:

```
public Response performComplete() {
  ...
  getRtPhase().setHideAfterCompletion(true);
  ...
}
```

Disabling Phase UI Extensions

➢ Does not apply to server-side phases running on the OE server.

TIP

For more information about UI extensions, please refer to section "Creating UI Extensions for Phase Building Blocks" (page 91).

Sometimes, a phase needs to disable certain UI extensions depending on its internal status. Phase extensions may veto the completion of a phase. In this case, to force an automatic completion, the phase needs to disable blocking phase UI extensions (see section "Step 2: Creating a Vetoable Completion Component" (page 93)). For this purpose, the phase can implement the *isPhaseUIExtensionEnabled* method in the *IPhaseUIExtensionConfiguration* interface.

The example below shows how to disable blocking phase UI extensions:

```
@Override
public boolean isPhaseUIExtensionEnabled(IPhaseUIExtension phaseUIExtension) {
  return AbstractPhaseUIExtension.isBlocking(phaseUIExtension);
}
```

Customizing Phase UI Extensions

➢ Does not apply to server-side phases running on the OE server.

For the phase completion signature and the sequential phase completion signature UI extensions, a custom access privilege can be used. In the *IPhaseExecutor* interface, override the following methods:

```
public AccessPrivilege getAccessPrivilegePhaseCompletion();
public AccessPrivilege getAccessSequentialPrivilegePhaseCompletion();
```

Supporting Automatic Confirmation of a Phase Completion Signature in the Context of External Authentication

➢ Does not apply to server-side phases running on the OE server.

PharmaSuite supports the usage of an external authentication system to populate the related boxes of forms that require the input of user credentials and then simulate the confirmation. For details, see chapter "Supporting External Authentication Systems" in Volume 2 of the "Technical Manual Configuration and Extension" [A2] (page 175).

The external authentication is triggered when the operator clicks with the mouse in or taps the box for the user name in the phase completion panel. To simulate the automatic confirmation of the phase completion signature and the phase when the user name and password boxes are not empty, the system needs to know the **Confirm** button. For this purpose, the *getConfirmButton()* method must be overloaded to return the **Confirm** button of the phase in the **Active** status.

```
...
public class RtPhaseExecutorPAVStr extends AbstractPhaseExecutorSwing {
...
/** Contains in active state the confirm button of the phase, otherwise null */
private JButton confirmButton;
...
public JButton getConfirmButton() {

  return confirmButton;
}
...
```

Providing Dynamic Phase Outputs

A runtime phase supports fixed outputs and dynamic outputs. Fixed outputs are defined during phase creation. The code generated for the outputs configured in "Step 3: Creating an XML Description of the Runtime Phase Output" (page 45) already contains the (static) output descriptors and the setters and getters for the outputs. Dynamic outputs of a phase can dynamically be confined and renamed.

1. Confining outputs means that a phase building block can offer only a subset out of its full number of (static) outputs. Then, only this subset can be used by the information flow within recipes.
2. Renaming outputs means to define another, more suitable display name for the outputs.

In order to provide dynamic outputs, a phase's generated output class has to implement the *IDynamicRtPhaseOutput* interface. *IDynamicRtPhaseOutput* only has the following method:

```
public List<IBuildingBlockOutputDescriptor> getDynamicOutputDescriptors(IMESPhase
                                            phase);
```

getDynamicOutputDescriptors() gets the corresponding phase as parameter, because it is very likely that dynamic outputs depend on the parameterization of the phase within the recipe.

TIP

Outputs can only depend on the following two aspects of the parameterization: which optional process parameter bundles have been assigned and their internal identifiers. They must not depend on dynamic attributes or other parameter types (e.g. material parameters). For more information, please refer to "PharmaSuite-related Java Documentation" [C1] (page 175).

Example

```
public class MESRtPhaseOutputGetValues0100 extends MESGeneratedRtPhaseOutputGetValues0100
             implements IDynamicRtPhaseOutput {
  // ...
  private static final String OUTPUT_PROPERTY_NAME_PREFIX = "attr";

  @Override
  public List<IBuildingBlockOutputDescriptor> getDynamicOutputDescriptors(IMESPhase
             phase) {
    List<IBuildingBlockOutputDescriptor> result = new ArrayList<>();
    List<IS88ProcessParameterBundle> parameterBundles =
        phase.getDynamicProcessParameterBundlesList();

    int bundleNumber = 0;
    for (IS88ProcessParameterBundle bundle : parameterBundles) {
      bundleNumber++;
      List<IBuildingBlockOutputDescriptor> descriptors =
           createOutputDescriptorFromProcessParameterBundle(bundle, bundleNumber);
      result.addAll(descriptors);
    }

    // add phase data reference to (dynamic) outputs
    result.add(createOutputDescriptorForPhaseDataReference("phaseDataReference"));
    return result;
  }

  private List<IBuildingBlockOutputDescriptor>
      createOutputDescriptorFromProcessParameterBundle(IS88ProcessParameterBundle
                                                  bundle,int bundleNumber) {
    String displayName = bundle.getUserDefinedIdentifier();
    List<IBuildingBlockOutputDescriptor> descriptorList = new
         ArrayList<IBuildingBlockOutputDescriptor>();
    switch (RtPhaseModelGetValues0100.ParameterBundleType.fromIdentifier(bundle.
            getInternalIdentifier())) {
    case MEASURED_VALUE_TYPE:
      descriptorList = createMVOutputDescriptorFromProcessParameterBundle(displayName,
                       bundleNumber);
      break;
    case BOOLEAN_TYPE:
      descriptorList = createBooleanOutputDescriptorFromProcessParameterBundle(
                       displayName, bundleNumber);
      break;
    case OPTION_TYPE:
      descriptorList = createOptionOutputDescriptorFromProcessParameterBundle(
                       displayName, bundleNumber);
      break;
    default:
      // must be a programming error
      throw new MESRuntimeException("unexpected internal bundle identifier " +
                                    bundle.getInternalIdentifier());
    }
    return descriptorList;
  }

  private List<IBuildingBlockOutputDescriptor>
          createMVOutputDescriptorFromProcessParameterBundle(String displayName, int
          bundleNumber) {
    List<IBuildingBlockOutputDescriptor> descriptorList = new
         ArrayList<IBuildingBlockOutputDescriptor>();
    String outputPropertyName = null;
    outputPropertyName = OUTPUT_PROPERTY_NAME_PREFIX + bundleNumber + "MVValue";
    IBuildingBlockOutputDescriptor descriptor = getOutputDescriptor(outputPropertyName).
          createCopyWithDisplayName(displayName + " Value");
    descriptorList.add(descriptor);
    outputPropertyName = OUTPUT_PROPERTY_NAME_PREFIX + bundleNumber + "MVUoM";
    descriptor = getOutputDescriptor(outputPropertyName).createCopyWithDisplayName(
                 displayName + " Unit of measure");
    descriptorList.add(descriptor);
    return descriptorList;

  }

  private List<IBuildingBlockOutputDescriptor>
          createBooleanOutputDescriptorFromProcessParameterBundle(String displayName,
          int bundleNumber) {
    List<IBuildingBlockOutputDescriptor> descriptorList = new
         ArrayList<IBuildingBlockOutputDescriptor>();
    String outputPropertyName = null;
    outputPropertyName = OUTPUT_PROPERTY_NAME_PREFIX + bundleNumber + "BooleanValue";
    IBuildingBlockOutputDescriptor descriptor = getOutputDescriptor(outputPropertyName).
          createCopyWithDisplayName(displayName + " Option key");
    descriptorList.add(descriptor);
    outputPropertyName = OUTPUT_PROPERTY_NAME_PREFIX + bundleNumber + "BooleanDValue";
    descriptor = getOutputDescriptor(outputPropertyName).createCopyWithDisplayName(
                 displayName + " Option text");
    descriptorList.add(descriptor);
    return descriptorList;
  }

  private List<IBuildingBlockOutputDescriptor>
          createOptionOutputDescriptorFromProcessParameterBundle(String displayName, int
          bundleNumber) {
    List<IBuildingBlockOutputDescriptor> descriptorList = new
         ArrayList<IBuildingBlockOutputDescriptor>();
    String outputPropertyName = null;
    outputPropertyName = OUTPUT_PROPERTY_NAME_PREFIX + bundleNumber + "ChoiceValue";
    IBuildingBlockOutputDescriptor descriptor = getOutputDescriptor(outputPropertyName).
          createCopyWithDisplayName(displayName + " Option key");
    descriptorList.add(descriptor);
    outputPropertyName = OUTPUT_PROPERTY_NAME_PREFIX + bundleNumber + "ChoiceDValue";
    descriptor = getOutputDescriptor(outputPropertyName).createCopyWithDisplayName(
                 displayName + " Option text");
    descriptorList.add(descriptor);
    return descriptorList;
  }
}
```

In the above example, the *IBuildingBlockOutputDescriptor* provides a new method called *createCopyWithDisplayName()*:

```
public IBuildingBlockOutputDescriptor createCopyWithDisplayName(String newDisplayName);
```

The method can be used to dynamically define another display name for an output.

Error Handling

Phases can detect an erroneous situation, log it, create/record a system-triggered exception, and throw a runtime exception later, which will be handled by the underlying sandbox (page 183) depending on where the phase is running:

- Phase runs in a Production Execution Client environment:
  Sandbox logs the erroneous situation and displays an appropriate message.
- Phase runs in a PharmaSuite OE server environment:
  Sandbox logs the erroneous situation and sends a JMS message to the Production Execution Clients running at the work centers of a unit procedure. The system displays an appropriate message at the Production Execution Clients.
  Additionally, a phase can call *handleUnexpectedErrorOnServer()* to send a JMS message to the Production Execution Clients in case of an exceptional situation (see section "Specific Information on Phases Running on the PharmaSuite OE Server" (page 69) in "Step 5: Creating a Phase Executor (Java Code)" (page 49)).

Step 7: Filling the Navigator's Information Column

➢ Does not apply to server-side phases running on the OE server.

The default implementation of the *getNavigatorInfoColumn()* method located in the *AbstractPhaseExecutor* base class only returns the phase name, i.e. the phase name will be displayed in the Navigator's information column. If you wish to display another piece of information in the information column, you have to override *getNavigatorInfoColumn()* in your phase executor.

Example

For our **MYC_StringValue** example, suppose you wish to display the operator-entered string value in the Navigator's information column. Then, simply implement *getNavigatorInfoColumn()* in the *RtPhaseExecutorMYCStringValue* class.

```
@Override
public String getNavigatorInfoColumn() {
  MESRtPhaseDataPAVStr data = getRtPhaseData();
  return data.getActualValue();
}
```

Alternatively, you can also adapt the Navigator's information column by means of a Pnuts subroutine (see section "Creating a Subroutine for Filling the Navigator's Information Column" (page 171)).

TIP

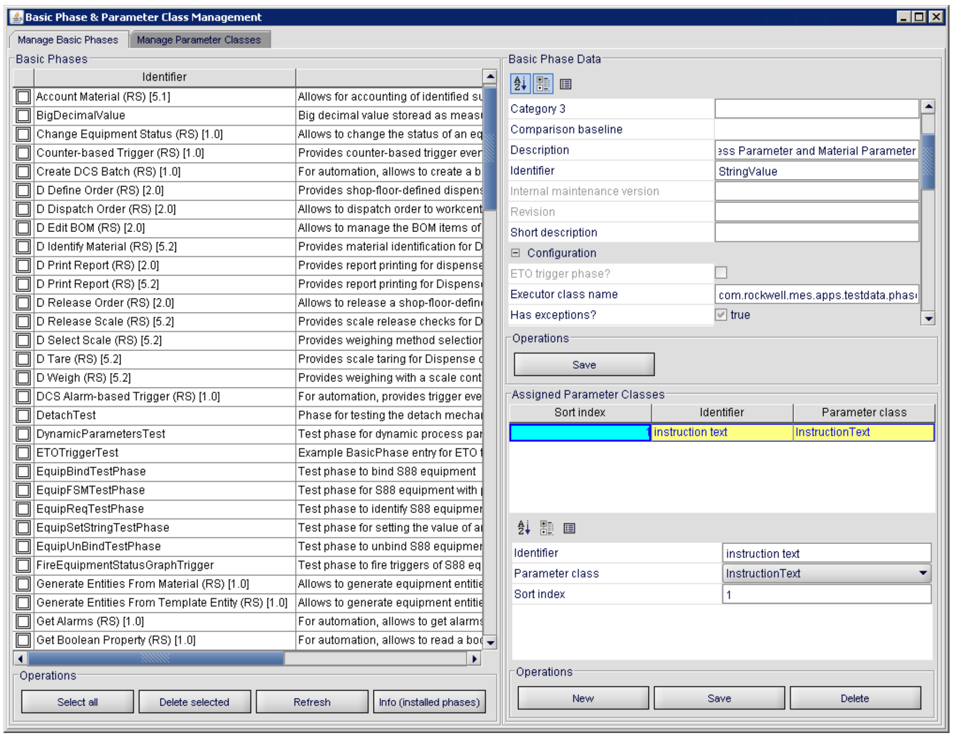
If a subroutine is defined, the *getNavigatorInfoColumn()* method will not be called.

Step 8: Verifying the Phase Building Block Creation with the Phase Manager

Since the phase generator has already created the phase building block artifacts for you there is usually no need to modify the non-Java artifacts by means of the phase manager. However, you can (and should) verify whether the phase building block data looks as expected.

For this purpose, proceed as follows:

- In Process Designer, run the **mes_PhaseLibManager** form to start the phase manager.



Basic Phases

	Identifier	
☐	Account Material (RS) [5.1]	Allows for accounting of identified su
☐	BigDecimalValue	Big decimal value storead as measu
☐	Change Equipment Status (RS) [1.0]	Allows to change the status of an eq
☐	Counter-based Trigger (RS) [1.0]	Provides counter-based trigger ever
☐	Create DCS Batch (RS) [1.0]	For automation, allows to create a b
☐	D Define Order (RS) [2.0]	Provides shop-floor-defined dispens
☐	D Dispatch Order (RS) [2.0]	Allows to dispatch order to workcent
☐	D Edit BOM (RS) [2.0]	Allows to manage the BOM items of
☐	D Identify Material (RS) [5.2]	Provides material identification for D
☐	D Print Report (RS) [2.0]	Provides report printing for dispense
☐	D Print Report (RS) [5.2]	Provides report printing for Dispens
☐	D Release Order (RS) [2.0]	Allows to release a shop-floor-defin
☐	D Release Scale (RS) [5.2]	Provides scale release checks for D
☐	D Select Scale (RS) [5.2]	Provides weighing method selectior
☐	D Tare (RS) [5.2]	Provides scale taring for Dispense c
☐	D Weigh (RS) [5.2]	Provides weighing with a scale cont
☐	DCS Alarm-based Trigger (RS) [1.0]	For automation, provides trigger eve
☐	DetachTest	Phase for testing the detach mechai
☐	DynamicParametersTest	Test phase for dynamic process par
☐	ETOTriggerTest	Example BasicPhase entry for ETO t
☐	EquipBindTestPhase	Test phase to bind S88 equipment
☐	EquipFSMTestPhase	Test phase for S88 equipment with
☐	EquipReqTestPhase	Test phase to identify S88 equipmer
☐	EquipSetStringTestPhase	Test phase for setting the value of ar
☐	EquipUnBindTestPhase	Test phase to unbind S88 equipmer
☐	FireEquipmentStatusGraphTrigger	Test phase to fire triggers of S88 eq
☐	Generate Entities From Material (RS) [1.0]	Allows to generate equipment entitie
☐	Generate Entities From Template Entity (RS) [1.0]	Allows to generate equipment entitie
☐	Get Alarms (RS) [1.0]	For automation, allows to get alarms
☐	Get Boolean Property (RS) [1.0]	For automation, allows to read a boo

Basic Phase Data

Category 3	
Comparison baseline	
Description	ɜss Parameter and Material Parameter
Identifier	StringValue
Internal maintenance version	
Revision	
Short description	
⊟ Configuration	
ETO trigger phase?	☐
Executor class name	com.rockwell.mes.apps.testdata.phas
Has exceptions?	✓ true

Assigned Parameter Classes

Sort index	Identifier	Parameter class
1	instruction text	InstructionText

Identifier	instruction text
Parameter class	InstructionText
Sort index	1

Figure 18: Phase manager

Though it is not recommended, it is possible to change the phase building block data of a generated phase building block by means of the phase manager. You can change all relevant properties as well as the material parameter configuration, the equipment requirement parameter configuration, and the assignment of the used parameter classes.

IMPORTANT

If you modify a generated phase building block generated by means of the phase manager, the phase building block data is not in sync with the XML description of the phase building block (created in Step 1 (page 33)). So if you will have to re-create your phase building block via phase generator in the future, do not modify it by means of

the phase manager.
Instead, modify the XML descriptions of the phase building block, the runtime phase data, and the phase output, if necessary, and re-create your phase building block via phase generator.

When your phase building block looks correct in the phase manager, you can use the phase in Recipe and Workflow Designer to instantiate phases and design your master recipe, master workflow, or building block, respectively.

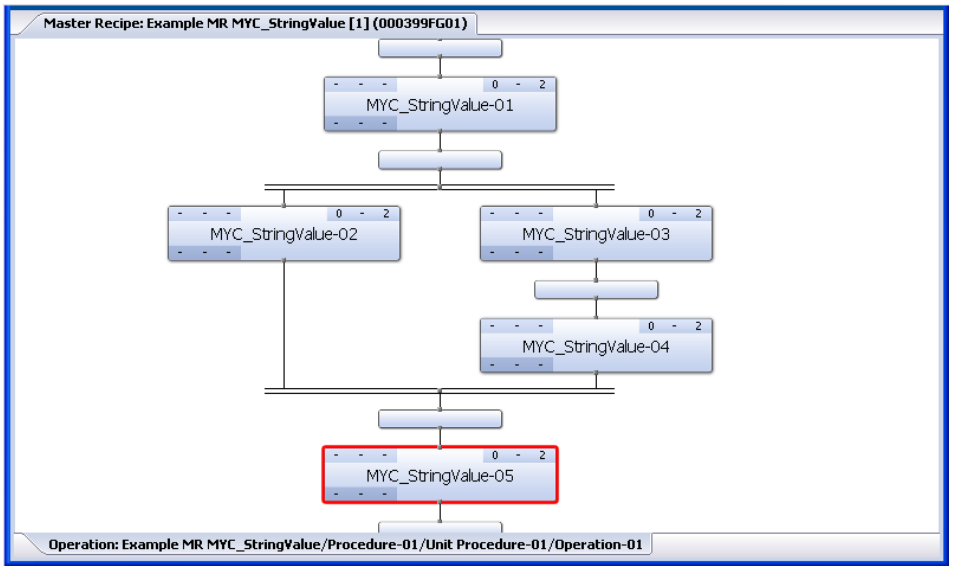
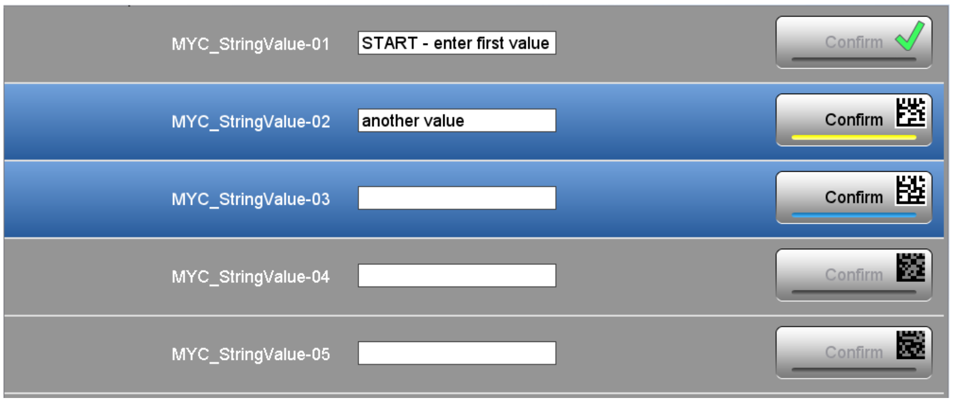

Master Recipe: Example MR MYC_StringValue [1] (000399FG01)
MYC_StringValue-01
MYC_StringValue-02
MYC_StringValue-03
MYC_StringValue-04
MYC_StringValue-05
Operation: Example MR MYC_StringValue/Procedure-01/Unit Procedure-01/Operation-01

Figure 19: MYC_StringValue phase in Recipe Designer



MYC_StringValue-01	START - enter first value	Confirm
MYC_StringValue-02	another value	Confirm
MYC_StringValue-03		Confirm
MYC_StringValue-04		Confirm
MYC_StringValue-05		Confirm

Figure 20: MYC_StringValue phase during execution

Creating UI Extensions for Phase Building Blocks

➢ Does not apply to server-side phases running on the OE server.

This section describes how to create UI extensions for phase building blocks. These UI extensions can be used in any phase building block.

To create a UI extension for phase building blocks, perform the following steps:

1. Create a UI extension for phase building blocks (page 91).
2. Create a vetoable completion component (page 93), if there are active UI elements requiring actions to be performed at phase completion.
   This step is optional.
3. Register the UI extension in the framework (page 96).

Step 1: Creating a UI Extension for Phase Building Blocks

To create a UI extension for phase building blocks, you have to implement the *IPhaseUIExtension* interface by deriving from the *AbstractPhaseUIExtension* abstract base class and overriding one of its draw methods:
The *drawAbove()* and *drawBelow()* methods render the extension of a phase that are outside the UI. Therefore they will work with all phases. In contrast, the *drawInside()* method can modify the UI of a phase itself. Although the latter extension provides more flexibility, you have to thoroughly check whether the structure of the phase UI matches the expectation of the UI extension.

Normally, a UI extension should only implement one of the three methods, but you can combine them. The *drawAbove()* and *drawBelow()* methods are easier to use since they do not have to deal with the inner structure of the phase's UI.

Example

```
public interface IPhaseUIExtension {
  /**
   * Add the UI components for this extension to the GUI of the phase
   *
   * @param phaseExecutor the corresponding phase executor, for which the
   *              extensions shall be drawn.
   */
  public void drawInside(IPhaseExecutor phaseExecutor);

  /**
   * Add the UI components for this extension above the GUI of the phase
   *
   * @param phaseExecutor the corresponding phase executor, for which the
```

```
    *             extensions shall be drawn.
    * @return The UI extension to be drawn above the phase
    */
   public JComponent drawAbove(IPhaseExecutor phaseExecutor);

   /**
    * Add the UI components for this extension below the GUI of the phase
    *
    * @param phaseExecutor the corresponding phase executor, for which the
    *             extensions shall be drawn.
    * @return The UI extension to be drawn below the phase
    */
   public JComponent drawBelow(IPhaseExecutor phaseExecutor);
```

Each *draw* method gets the phase executor as parameter. Thus, the UI extension has complete access to the information needed to implement the UI part of the extension (e.g. status of the phase executor, the *IMESPhase*, *IMESRtPhase*, all parameters of the phase or the phase's UI control). This approach enables the UI extension to create the UI elements depending on their status. The UI extension can decide whether the UI of the phase has to be extended or not. This can depend on the input parameters of the phase or given phase executor.

Please be aware of the fact that *IMESRtPhase* is not available if the UI extension is rendered during preview. We recommend to always use the *getParameters* methods of the executor since the methods always return the correct parameters from the phase during preview and from the runtime phase during execution.

This is an example implementation of a phase UI extension (with MYC for the My Company vendor code).

Example implementation

```
public class MYCUIExtension extends AbstractPhaseUIExtension {

  @Override
  public JComponent drawAbove(IPhaseExecutor phaseExecutor) {
    List<IMESProcessParameterInstance> parameters = phaseExecutor.getParameters();
    // Check process parameter and decide if UI shall be extended
    if (checkParameters(parameters)  {
      JLabel label = new JLabel("Additional Label from MYCUIExtension");
      label.setPreferredSize(UIConstants.LABEL_PREFERRED_SIZE);
      return label;
    } else {
      return null;
    }
  }
```

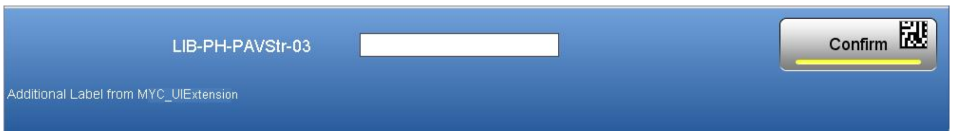


Figure 21: Example implementation rendered in the Execution Window

The phase UI extension can decide whether it will be rendered in any phase or only in some specific cases. In the latter case, for example, the phase UI extension will be rendered depending on specific parameters or only for specific phases.

TIP

- An instance of a phase UI extension should be a singleton.
- A phase UI extension can decide whether it will be rendered or not.
- A phase UI extension can decide whether it will be rendered in different ways depending on the status of the phase.

Step 2: Creating a Vetoable Completion Component

This step is optional.
It is not needed if the UI extension for phase building blocks contains only UI elements that show information.

Perform this step if the UI extension for phase building blocks contains active UI elements that may prevent the phase completion (e.g. mandatory input fields) or that require actions to be performed at phase completion. In this case, you have to create a vetoable completion component to make sure the phase can only be completed if the mandatory input has been provided by the operator. A UI extension that creates and registers a vetoable completion component is called a **blocking phase UI extension**.

A vetoable completion component is a class implementing the following interface.

Vetoable completion component

```
public interface IPhaseExecutorVetoableCompletionComponent {
  /**
   * Perform all phase specific checks in order to check if the phase can be
   * completed or if there is a veto.
   *
   * @return true if there is no veto on complete (all checks passed
   *         successfully and phase can be completed)
   */
  public abstract boolean performCompletionCheck();
  /**
   * Perform the complete. This method will be called if performCompletionCheck()
   * returns with true. Currently the Response will not be evaluated. If an error
   * Response will be returned, an error dialog will be created.
   *
   * @return a response object, which contains the no errors or
   *         technical errors, which are occurred.
   */
  public abstract Response performComplete();
```

A vetoable completion component will be called, when a phase is going to complete. The *performCompletionCheck()* method will be called before the completion will be performed. This allows the component to veto the completion. If

performCompletionCheck returns false, the completion will not performed. In addition, the UI extension can display a message to inform the operator why the completion has been vetoed.
Example: The phase UI extension implements an electronic signature. The phase can only continue with its completion if the given user name and password are valid. *performCompletionCheck* verifies the signature (user name, password), returns false to prevent the completion in case of an invalid signature, and displays a corresponding message box.

The *performComplete()* method will be called, when all of the completion-related checks of the registered components and the phase itself have been passed successfully. In our example, the *performComplete()* method must save the signature data.

This is an example implementation of the *IPhaseExecutorVetoableCompletionComponent* interface.

Example implementation

```
import javax.swing.JTextField;
import com.datasweep.compatibility.client.Response;
import com.rockwell.mes.apps.ebr.ifc.phase.IPhaseExecutor;
import com.rockwell.mes.apps.ebr.ifc.phase.IPhaseExecutorVetoableCompletionComponent;

public class PhaseUIExtensionVetoableCompletionComponent implements
IPhaseExecutorVetoableCompletionComponent {
  /** The corresponding phase executor */
  private IPhaseExecutor phaseExecutor;
  /** the ui fields for the signature */
  JTextField editUser;
  /** the ui fields for the signature */
  JTextField editPassword;

  /**
   * @param inPhaseExecutor the corresponding phase executor
   * @param inEditUser        the input field for the user
   * @param inEditPassword    the input field for the password
   */
  PhaseUIExtensionVetoableCompletionComponent(IPhaseExecutor inPhaseExecutor,
                        JTextField inEditUser, JTextField inEditPassword) {
    editUser = inEditUser;
    editPassword = inEditPassword;
    phaseExecutor = inPhaseExecutor;
  }
  @Override
  public boolean performCompletionCheck() {
    return checkUserAndPassword();
  }
  private boolean checkUserAndPassword() {
    // check editUser and editPassword
    // and return true if signature is ok, otherwise false
  }
  @Override
  public Response performComplete() {
    if (checkUserAndPassword()) {
      saveSignature();
    }
    return new Response();
  }
  private void saveSignature() {
    // Save the signature
  }
}
```

TIP

To implement a check related to the signature at field exit, the component can set an input verifier on the edit fields. For details, see section "Step 2: Triggering the Check during Phase Execution" (page 136).

The vetoable completion component has to be registered at the phase completer of the phase executor. For this purpose, the *IPhaseCompleter* interface provides a corresponding method.

Method

```
public interface IPhaseCompleter {
  ...
  /**
   * Adds a component, which shall perform the completion too.
   *
   * @param component component, which is able to perform the completion check
   *                  and do the completion on the component
   *
   */
  public abstract void addVetoableCompletionComponent(
    IPhaseExecutorVetoableCompletionComponent component);
}
```

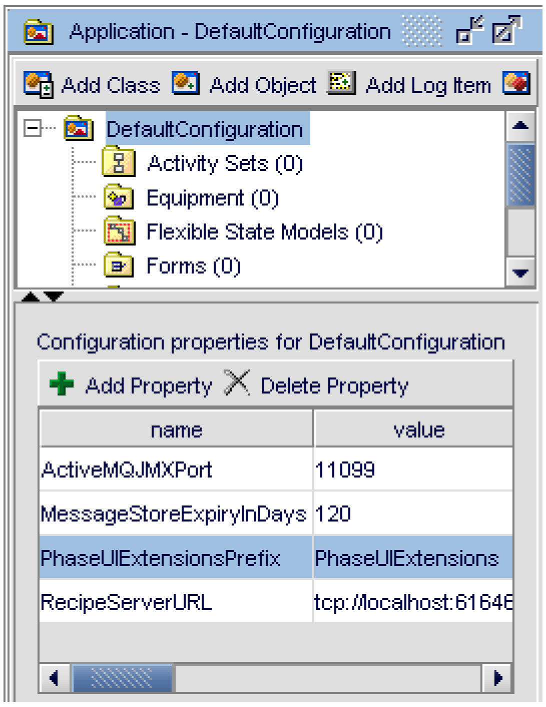
This is an example of a UI extension for phase building blocks with active components. It demonstrates the creation and registration of a vetoable completion component.
Usually, only active phases have active UI elements. Thus, the creation and registration will only be done for phases in the **Active** status.
A blocking phase UI extension must override the *isBlocking()* method and return true.
This feature supports phases to disable blocking phase UI extensions (see section "Disabling Phase UI Extensions" (page 83)).

Example

```
public class MYCActiveUIExtension {
  @Override
  public void drawInside(IPhaseExecutor phaseExecutor) {
    // create UI elements editUser and editPassword
    // add the elements to the phase UI
    ...
    if (phaseExecutor.getStatus() == Status.ACTIVE) {
      PhaseUIExtensionVetoableCompletionComponent activeComponent =
        new PhaseCompletionSignatureHandler(phaseExecutor, editUser, editPassword);
      phaseExecutor.getPhaseCompleter().
        addVetoableCompletionComponent(activeComponent);
    }
  }
  @Override
  public boolean isBlocking() {
    return true;
  }
}
```

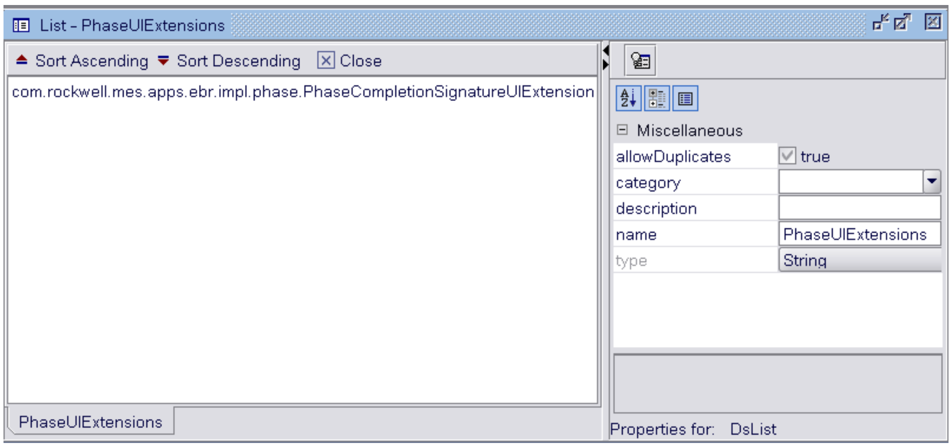
Step 3: Registering a UI Extension for Phase Building Blocks

To make a UI extension for phase building blocks available for use, you must register the full class name in a **List** object in Process Designer with a name starting with **PhaseUIExtensions**. The **PhaseUIExtensionsPrefix** property of the **DefaultConfiguration** application object refers to the prefix used for lists that contain UI extensions. The default **PhaseUIExtensions** list contains the UI extensions provided with PharmaSuite. To make your extensions available, enter the extensions in new lines to an existing list that fits the name pattern.



name	value
ActiveMQJMXPort	11099
MessageStoreExpiryInDays	120
PhaseUIExtensionsPrefix	PhaseUIExtensions
RecipeServerURL	tcp://localhost:61646

Figure 22: PhaseUIExtensionsPrefix as property of the DefaultConfiguration

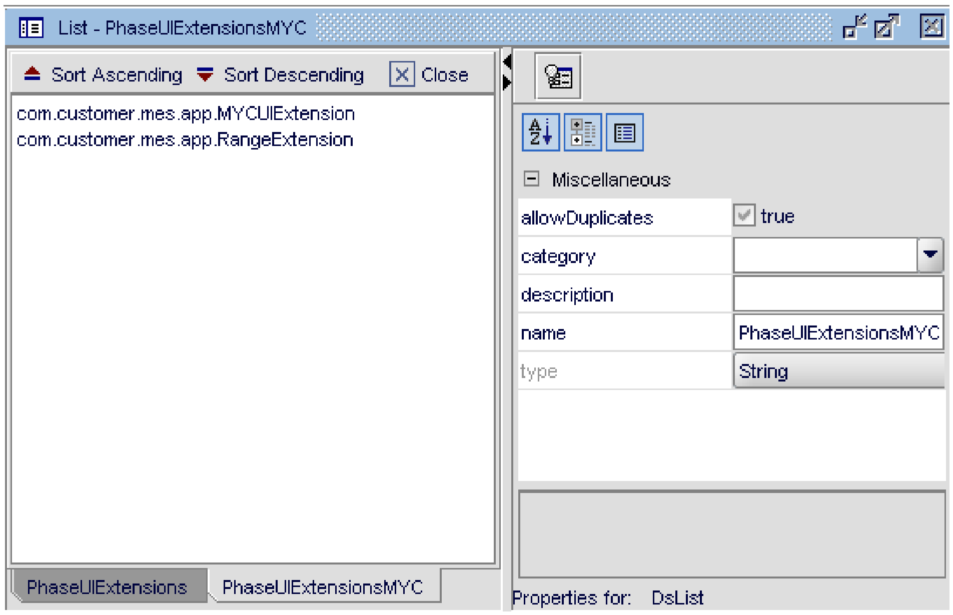


com.rockwell.mes.apps.ebr.impl.phase.PhaseCompletionSignatureUIExtension

allowDuplicates	✓ true
category	
description	
name	PhaseUIExtensions
type	String

Figure 23: PhaseUIExtensions as FactoryTalk ProductionCentre list object

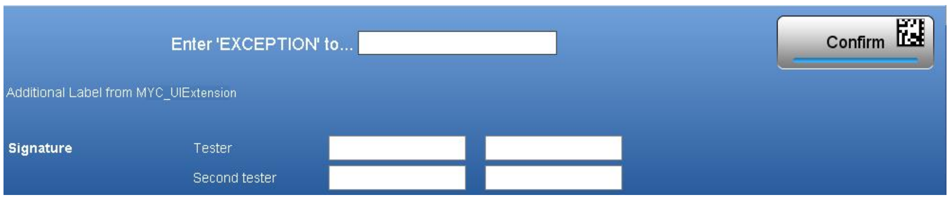
During startup of the Production Execution Client and Recipe and Workflow Designer, the defined entries from all lists with the **PhaseUIExtensions** prefix are registered automatically.



com.customer.mes.app.MYCUIExtension
com.customer.mes.app.RangeExtension

allowDuplicates	✓ true
category	
description	
name	PhaseUIExtensionsMYC
type	String

Figure 24: Custom list object



Enter 'EXCEPTION' to...

Additional Label from MYC_UIExtension

Signature Tester
Second tester

Figure 25: Example with 2 UI extensions rendered in the Execution Window

TIP

The UI extensions from the default **PhaseUIExtensions** list object will be registered and later be drawn during execution in the same order as they are listed. UI extensions from the other list objects will be drawn above them. Thus, if you wish the UI extensions to be drawn in a specific order, we recommend to merge all lists starting with **PhaseUIExtensions** manually into the default list and to sort the entries.

Chapter 7

Creating and Using Process Parameters

This section describes how to create a parameter class with the parameter class generator. In addition, it outlines how to use them within Recipe and Workflow Designer and during the execution of the phase.

At the end of the section we will have extended our **MYC_StringValue** phase created in section "Creating a Phase Building Block" (page 33) to display a string read from a parameter called **Instruction text**.

To create and use a parameter class, perform the following steps:

1. Create an XML description of the parameter class (page 99).
2. Create an XML description for the specific attributes of your parameter class (page 101).
3. Run the parameter class generator (page 103) with the XML descriptions created in the preceding steps.
4. Assign the parameter classes (page 104) to your phase.
5. Optional:
   Assign dynamic parameter classes (page 105) to your phase.
6. Configure the user interface of the parameter class (page 108) in order to define which attributes are displayed in which order when editing the attributes in Recipe and Workflow Designer.
7. Assign values to the parameters (page 110) in Recipe and Workflow Designer in order to set phase-specific values for the parameter attributes.
8. Use the parameters in your phase implementation (page 111) to actually read the parameters (defined in Recipe and Workflow Designer) during execution and react to the provided values accordingly.

Step 1: Creating an XML Description of the Parameter Class

The XML schema of a parameter class has the following structure:

XML schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
                    attributeFormDefault="unqualified">
  <xs:element name="ParameterClassDescription">
    <xs:complexType>
```

```
      <xs:sequence>
        <xs:element name="OutputDir">
          <xs:simpleType>
            <xs:restriction base="xs:string">
              <xs:minLength value="1" />
            </xs:restriction>
          </xs:simpleType>
        </xs:element>
        <xs:element name="Name">
          <xs:simpleType>
            <xs:restriction base="xs:string">
              <xs:minLength value="2" />
            </xs:restriction>
          </xs:simpleType>
        </xs:element>
        <xs:element name="Description">
          <xs:simpleType>
            <xs:restriction base="xs:string">
              <xs:minLength value="3" />
            </xs:restriction>
          </xs:simpleType>
        </xs:element>
        <xs:element name="PackageName">
          <xs:simpleType>
            <xs:restriction base="xs:string">
              <xs:minLength value="17" />
            </xs:restriction>
          </xs:simpleType>
        </xs:element>
        <xs:element name="ParamClassBaseName">
          <xs:simpleType>
            <xs:restriction base="xs:string">
              <xs:minLength value="1" />
            </xs:restriction>
          </xs:simpleType>
        </xs:element>
        <xs:element name="ATDefinitionPrefix">
          <xs:simpleType>
            <xs:restriction base="xs:string"/>
          </xs:simpleType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

For our **Instruction text** parameter, the parameter class description could look as follows:

```
<ParameterClassDescription>
    <OutputDir>[absolute path of output directory]</OutputDir>
    <Name>Instruction Text (MYC) [2.1]</Name>
    <Description>Parameter class for Instruction text with only one text property for the
                 instruction text.</Description>
    <ParamClassBaseName>InstructText0201</ParamClassBaseName>
    <PackageName>com.mycompany.parameter.instructiontext</PackageName>
    <ATDefinitionPrefix>MYC</ATDefinitionPrefix>
</ParameterClassDescription>
```

The individual nodes of the XML structure have the following meaning:

- ParameterClassDescription
  Top-level node.
- OutputDir
  Specify the output directory where all the generated files will be located (here: *<source_directory>\apps\testdata\src\com\rockwell\mes\apps\testdata\recipe*).
- Name
  Name of the parameter class.
  Maximum length is 64 characters.
- Description
  Description of the parameter class.
- ParamClassBaseName
  Base name of the parameter class. This name can differ from the actual building block name in order to circumvent name length constraints, especially regarding the names of application tables.
  Maximum length is 18 characters (14 for the name, 4 for the version).
- PackageName
  Package name of the generated Java artifacts.
- ATDefinitionPrefix
  Specifies the prefix of the table and column names of the parameter instance application table. Use a vendor code consisting of up to three uppercase letters as prefix (e.g. **MYC**).
  X_ and **RS_** are reserved for PharmaSuite and PharmaSuite-specific product building blocks, respectively.

Step 2: Creating an XML Description for the Specific Parameter Class Attributes

The XML schema of the parameter class attributes has the following structure:

XML schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns="http://www.wbf.org/xml/RAPhaseData"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            elementFormDefault="qualified" attributeFormDefault="unqualified">

  <xsd:include schemaLocation="PhaseDataType.xsd"></xsd:include>
  <xsd:element name="ParameterAttributes">
    <xsd:complexType>
      <xsd:sequence>
      <xsd:element name="ParameterAttribute" maxOccurs="unbounded">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="Name">
              <xsd:simpleType>
                <xsd:restriction base="xsd:string">
```

```
                   <xsd:minLength value="1" />
                 </xsd:restriction>
               </xsd:simpleType>
              </xsd:element>
            <xsd:element name="Description" type="xsd:string" minOccurs="0" />
            <xsd:element name="DataType" type="PhaseDataType"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      </xsd:sequence>
     </xsd:complexType>
   </xsd:element>
</xsd:schema>
```

For a description of the included *PhaseDataType.xsd* see "Step 2: Creating an XML Description of the Runtime Phase Data" (page 41).

For our **Instruction text** parameter, the parameter attributes description looks as follows:

```
<ParameterAttributes>
  <ParameterAttribute>
    <Name>Text</Name>
    <Description>The text of the Instruction Text</Description>
    <DataType>
      < String length="64"/>
    </DataType>
  </ParameterAttribute>
</ParameterAttributes>
```

The individual nodes of the XML structure have the following meaning:

- ParameterAttributes
  Top-level node containing 1 to N child nodes called **ParameterAttribute**, each representing one parameter attribute. A **ParameterAttribute** node has the following children:
  - Name
    Base name of the attribute. When the corresponding application table is generated, the parameter class generator automatically adds the prefix configured by the *ATDefinitionPrefix* element to the base name in order to build the column name of the attribute.
    ATDefinitionPrefix is an element of the XML description of the parameter class (page 99).
  - Description
    Description of the attribute.
  - DataType
    Data type of the attribute. For details, see description of *PhaseDataType.xsd* in "Step 2: Creating an XML Description of the Runtime Phase Data" (page 41).

Step 3: Running the Parameter Class Generator

In order to run the parameter class generator, you have to call the *mainImpl()* method of the Java *ParamClassGenerator* class. The method looks as follows:

```
public void mainImpl(String[] args)
```

The method expects no ordinary arguments, i.e. the *args* parameter is not used. Hence, the generator gets its arguments as system properties (i.e. introduced by the *-D* option). The following arguments are supported:

- *-DparamClassDescFile*
  The (relative or full) path name of the XML file that contains the description of the parameter class.
- *-DparamClassDataDescFile*
  The (relative or full) path name of the XML file that contains the description of the specific parameter attributes.
- *-DexecuteCritical*
  If specified and set to *Y*, the generator performs changes in the form of updates to the generated artifacts that are considered as critical (see "Critical Changes" (page 104)).

You can call this method from Pnuts (e.g. from within a corresponding form implemented by yourself). This is an outline of what the corresponding Pnuts code could look like:

Example Pnuts code

```
import("com.rockwell.mes.tools.phases.impl.ParamClassGenerator")

function generateParameterClass() {
  // get the file names of the parameter class description and the
  // parameter attributes description, e.g. from some edit controls
  // "paramClassDescPath" and "paramAttribsDescPath"
  paramClassDescFileName = paramClassDescPath.getText()
  paramAttribsDescFileName = paramAttribsDescPath.getText()

  // call ParamClassGenerator
  generator = ParamClassGenerator()
  args = [] // currently no ordinary program arguments
  System::setProperty("paramClassDescFile", paramClassDescFileName)
  System::setProperty("paramClassDataDescFile", paramAttribsDescFileName)
  generator.mainImpl(args)
}
```

After calling the *ParamClassGenerator.mainImpl()* method, all the artifacts described in "What Is the Parameter Class Generator?" (page 23) will be available.

In order to run your form containing the above Pnuts code, please make sure the following libraries have been imported into Process Designer:

- *tools-phases-ifc.jar* (located in *<source_directory>\dependencies\FTPS_DSX\addon_jars\*)
- *tools-phases-impl.jar* (located in *<source_directory>\dependencies\FTPS_DSX\addon_jars\*)

Additionally, ensure that the FactoryTalk ProductionCentre *com.rockwell.tools.dsx.DSXDisassembler* class is in your classpath. The class is, for example, contained in *ProductionCentreClient-<build number>_JBossSA.jar*.
If you do not have access to *ProductionCentreClient-<build number>_JBossSA.jar*, you can obtain the class from the *importexport.jar* file of FactoryTalk ProductionCentre. The *importexport.jar* file is located in the *PlantOpsWebStart-JBoss.war* directory of your FactoryTalk ProductionCentre JBoss installation (e.g. *c:\Rockwell\FT_ProductionCentre\jboss\server\datasweepConfig\deploy\DSPlantOperations.ear\PlantOpsWebStart-JBoss.war\importexport.jar*).

Critical Changes

Critical changes are changes that are incompatible with previous versions of a parameter class and thus incompatible with master recipes, master workflows, and building blocks in which previous versions of this parameter class have been used. They can only occur if a parameter class or its associated artifacts, respectively, already exist, i.e. when a parameter class is created for the first time, there are no critical changes.

The following changes are considered as critical and are therefore only executed if the *executeCritical* system property is set to *Y*.

- Changing the data type of a parameter instance's application table column.
- Reducing the length of a parameter instance's application table column of the *String* type.

If one of the above mentioned critical change scenarios occurs and critical changes are not allowed, the parameter class generation aborts.

Step 4: Assigning Parameter Classes to Phases

As already outlined in Step 1 (page 33) of "Creating a Phase Building Block" (page 33), the assignment of (static) parameter classes to a phase can be described in the corresponding XML description of the phase building block.

Example

In order to assign the **InstructionText** parameter class to our **MYC_StringValue** phase, we must extend the XML description of the corresponding phase building block as follows (relevant are the red lines):

```
<PhaseLibDescription>
  <OutputDir>[absolute path of output directory]</OutputDir>
  <Name>String Value (MYC) [1.0]</Name>
  <Description> Basic phase for Planned/Actual value data type String.</Description>
  <PackageName>com.mycompany.phase.mycstringvalue</PackageName>
  <PhaseLibBaseName>MYC_StringValue0100</PhaseLibBaseName>
  <VisibleInNavigator value="true"/>
  <InternalBuildingBlock value="false"/>
  <MaterialInputMin>0</MaterialInputMin>
  <MaterialInputMax>0</MaterialInputMax>
  <MaterialOutputMin>0</MaterialOutputMin>
  <MaterialOutputMax>0</MaterialOutputMax>
  <ATDefinitionPrefix>MYC</ATDefinitionPrefix>
  <ParameterClasses>
    <ParameterClass>
      <Name>Instruction Text (MYC) [2.1]</Name>
      <UsageIdentifier>instruction text</UsageIdentifier>
      <SortIndex>1</SortIndex>
    </ParameterClass>
  </ParameterClasses>
</PhaseLibDescription>
```

Then, re-create your **MYC_StringValue** phase with the phase generator.

Now, you can access the corresponding parameter instance during execution by means of its usage identifier (see section "Step 8: Using Parameters during Phase Execution" (page 111)).

Step 5: Assigning Dynamic Parameter Classes to Phases

Analogous to static parameter classes (page 104), the assignment of dynamic parameter classes to a phase can be described in the corresponding XML description of the phase building block, see Step 1 (page 33) of "Creating a Phase Building Block" (page 33).

Example

The following XML excerpt assigns three dynamic parameter class bundles to a phase building block. Each bundle represents a specific configuration:

- MeasuredValue-related configuration with **Measured Value** as internal bundle identifier (**UsageIdentifier** of master/top-level parameter).
- Option list-related configuration with **Option Value** as internal bundle identifier.
- Boolean-related configuration with **Boolean Value** as internal bundle identifier.

The phase building block can handle an overall maximum number of 5 bundles of the above types.

XML excerpt

```
<DynamicParameterClasses>
  <MaximumNumber>5</MaximumNumber>

  <!-- MeasuredValue parameter class bundle -->
  <DynamicParameterClass>
    <Name>Parameter Bundle Attributes (RS) [1.0]</Name>
    <UsageIdentifier>Measured Value</UsageIdentifier>
    <DependentParameterClass>
      <Name>UoM Definition (RS) [1.0]</Name>
      <UsageIdentifier>Unit of measure</UsageIdentifier>
      <SortIndex>1</SortIndex>
    </DependentParameterClass>
    <DependentParameterClass>
      <Name>Expected Value Configuration (RS) [2.0]</Name>
      <UsageIdentifier>Limit configuration</UsageIdentifier>
      <SortIndex>2</SortIndex>
    </DependentParameterClass>
    <DependentParameterClass>
      <Name>MeasuredValue Validation (RS) [1.0]</Name>
      <UsageIdentifier>Limit definition</UsageIdentifier>
      <SortIndex>3</SortIndex>
    </DependentParameterClass>
    <DependentParameterClass>
      <Name>Exception Definition (RS) [3.0]</Name>
      <UsageIdentifier>Override value</UsageIdentifier>
      <SortIndex>4</SortIndex>
    </DependentParameterClass>
    <DependentParameterClass>
      <Name>Exception Definition (RS) [3.0]</Name>
      <UsageIdentifier>Correct value</UsageIdentifier>
      <SortIndex>5</SortIndex>
    </DependentParameterClass>
  </DynamicParameterClass>

  <!-- Option value parameter class bundle -->
  <DynamicParameterClass>
    <Name>Parameter Bundle Attributes (RS) [1.0]</Name>
    <UsageIdentifier>Option Value</UsageIdentifier>
    <DependentParameterClass>
      <Name>ListOfChoices (RS) [2.0]</Name>
      <UsageIdentifier>List of options</UsageIdentifier>
      <SortIndex>1</SortIndex>
    </DependentParameterClass>
    <DependentParameterClass>
      <Name>Expected Value Configuration (RS) [2.0]</Name>
      <UsageIdentifier>Expected value configuration</UsageIdentifier>
      <SortIndex>2</SortIndex>
    </DependentParameterClass>
    <DependentParameterClass>
      <Name>Text Validation (RS) [3.0]</Name>
      <UsageIdentifier>Expected value definition</UsageIdentifier>
      <SortIndex>3</SortIndex>
    </DependentParameterClass>
    <DependentParameterClass>
      <Name>Exception Definition (RS) [3.0]</Name>
      <UsageIdentifier>Override value</UsageIdentifier>
      <SortIndex>4</SortIndex>
    </DependentParameterClass>
    <DependentParameterClass>
      <Name>Exception Definition (RS) [3.0]</Name>
      <UsageIdentifier>Correct value</UsageIdentifier>
      <SortIndex>5</SortIndex>
    </DependentParameterClass>
  </DynamicParameterClass>

  <!-- Boolean parameter class bundle -->
  <DynamicParameterClass>
    <Name>Parameter Bundle Attributes (RS) [1.0]</Name>
    <UsageIdentifier>Boolean Value</UsageIdentifier>
    <DependentParameterClass>
      <Name>Boolean Definition (RS) [1.0]</Name>
      <UsageIdentifier>Boolean options</UsageIdentifier>
      <SortIndex>1</SortIndex>
    </DependentParameterClass>
    <DependentParameterClass>
      <Name>Expected Value Configuration (RS) [2.0]</Name>
      <UsageIdentifier>Expected value configuration</UsageIdentifier>
      <SortIndex>2</SortIndex>
    </DependentParameterClass>
    <DependentParameterClass>
      <Name>Boolean Validation (RS) [1.0]</Name>
      <UsageIdentifier>Expected value definition</UsageIdentifier>
      <SortIndex>3</SortIndex>
    </DependentParameterClass>
    <DependentParameterClass>
      <Name>Exception Definition (RS) [3.0]</Name>
      <UsageIdentifier>Override value</UsageIdentifier>
      <SortIndex>4</SortIndex>
    </DependentParameterClass>
    <DependentParameterClass>
      <Name>Exception Definition (RS) [3.0]</Name>
      <UsageIdentifier>Correct value</UsageIdentifier>
      <SortIndex>5</SortIndex>
    </DependentParameterClass>
  </DynamicParameterClass>

</DynamicParameterClasses>
```

In Recipe and Workflow Designer, a bundle is added to a recipe with the **Add parameter bundle** button in the Parameter Panel.

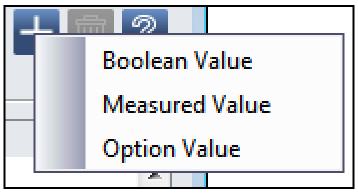
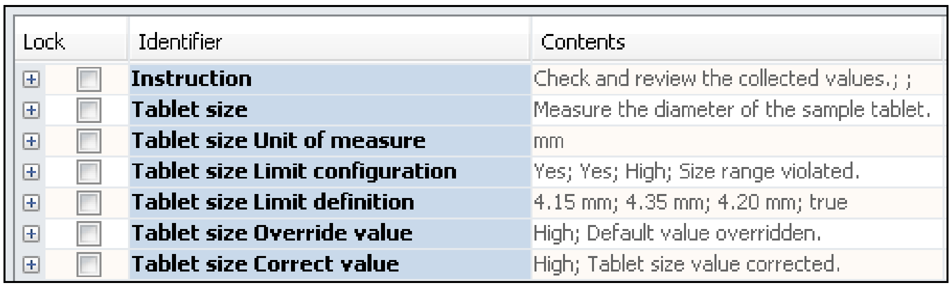


Figure 26: Add dynamic process parameter

In the Parameter Panel, a **Measured Value** bundle with **Tablet size** as user-defined bundle identifier looks as follows:



Lock	Identifier	Contents
☐	**Instruction**	Check and review the collected values.; ;
☐	**Tablet size**	Measure the diameter of the sample tablet.
☐	**Tablet size Unit of measure**	mm
☐	**Tablet size Limit configuration**	Yes; Yes; High; Size range violated.
☐	**Tablet size Limit definition**	4.15 mm; 4.35 mm; 4.20 mm; true
☐	**Tablet size Override value**	High; Default value overridden.
☐	**Tablet size Correct value**	High; Tablet size value corrected.

Figure 27: Measured Value bundle (Tablet size)

The internal identifiers of process parameters have to be unique in the context of a phase. We cannot simply use their usage identifiers configured in the XML description above, because in case the recipe author has added two bundles of the same type, the uniqueness is violated. Therefore the assignment of internal identifiers of parameters of a bundle needs a special handling: In order to determine the internal identifier of a process parameter of a bundle, use the *getInternalIdentifierOfProcessParameter()* dedicated method in *IS88ProcessParameterBundle*:

```
public String getInternalIdentifierOfProcessParameter(String
                                    usageIdentifierFromPhaseDefinition)
```

Once the phase developer has the internal identifier of such a parameter, the handling is identical to the one used for static process parameters.

Step 6: Configuring the UI of Parameter Classes

In Recipe and Workflow Designer, the assignment of values to parameters is done in the **Parameter Panel** (see section "Assigning Values to Parameters (Recipe and Workflow Designer)" (page 110)). The parameter class-specific labels of the **Parameter Panel** are defined in the parameter class-specific message pack **DataDictionary_Default_<ParameterClassName>**. This message pack can be used to configure labels of the specific parameter attributes:

- <Property>_Label

The **<ParameterClassName>** Data Dictionary can be used for configuring parameter classes, e.g. if the properties are editable and visible, to change the order in which the parameter classes are displayed in the grid, etc.

Please replace **<ParameterClassName>** with the name of your Java class (e.g. **MESParamInstructionText**) corresponding to the application table of your parameter class and replace **<Property>** with the identifier of the name of the parameter class.

TIP

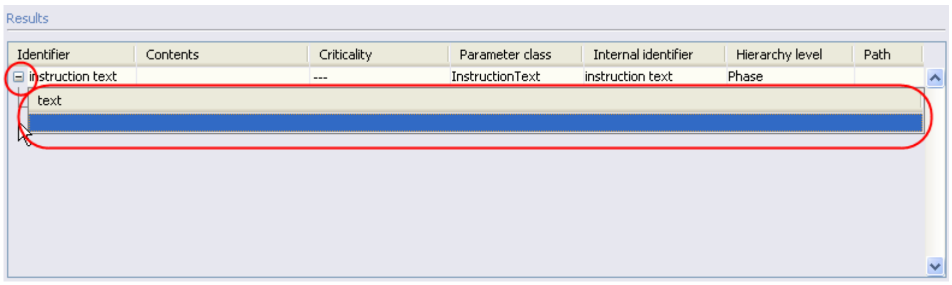
The parameter class-specific message pack (and its entries) as well as the corresponding Data Dictionary is initially created automatically when you open the parameter assignment for a building block containing the parameter class in Recipe and Workflow Designer.
Thus you do not need to create the entries from scratch, it is sufficient to configure them.

Example

To configure the UI of the **Parameter Panel** of Recipe and Workflow Designer for your phase, proceed as follows:

1. In Recipe and Workflow Designer, create a master recipe or master workflow with at least one phase, respectively. Replace it with your phase (here: **MYC_StringValue** type).

2. Click the parameter button of the phase to open the **Parameter Panel** and expand the parameter instance properties.



Results

Identifier	Contents	Criticality	Parameter class	Internal identifier	Hierarchy level	Path
instruction text		---	InstructionText	instruction text	Phase	

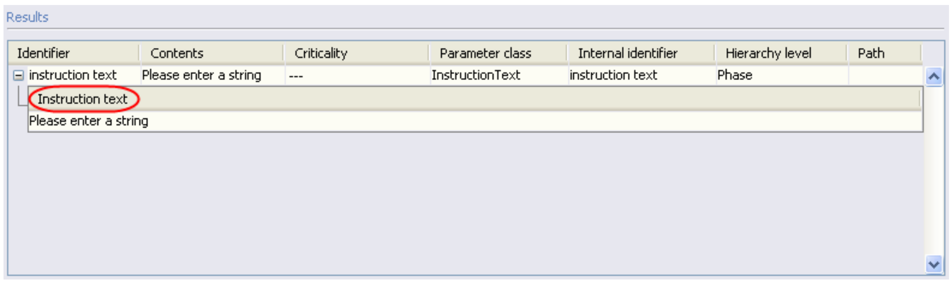
text

Figure 28: Parameter instance properties in the Parameter Panel

3. Close the **Parameter Panel**.
4. The system creates the message pack and the Data Dictionary for the property of the parameter class.

TIP
The same applies to dynamic parameter classes of a phase building block.

5. In Process Designer, expand the **Messages** node and open the **DataDictionary_Default_MESParamInstructionText** message pack.
6. Define the following message:
   - Set **text_Label** to **Instruction text**.
7. Save and close the **DataDictionary_Default_MESParamInstructionText** message pack.
8. Close Process Designer to clear the messages (and Data Dictionary) cache.
9. At the next start of Recipe and Workflow Designer you will see the localized label of your parameter instances property in the **Parameter Panel**.



Results

Identifier	Contents	Criticality	Parameter class	Internal identifier	Hierarchy level	Path
instruction text	Please enter a string	---	InstructionText	instruction text	Phase	

Instruction text
Please enter a string

Figure 29: Localized property label in the Parameter Panel

TIP

By default, you do not need to adjust the Data Dictionary since only the properties of your parameter instance are shown in the **Parameter Panel**. However, if you wish to adjust the Data Dictionary in order to change the display sequence of your properties in the property pane, for example, proceed as follows:

- In Process Designer, run the **mes_DataDictManagerForm** form to start the **Data Dictionary Management** tool.
  You can use the **Data Dictionary Manager** to view and manage contents of the Data Dictionary.
- Navigate to the **Class management** tab.
  In the **Data Dictionary Classes** panel, in the **Class-related actions** panel, select the **MESParamInstructionText** DD class.
- Click the **Load** button. If version control is enabled, click the **Check out** button.
- Set the **Grid order** or any other attribute like **Hidden** or **Edit?**.
- Click the **Save changes** button. If version control is enabled, click the **Check in** button.
- Close the **Data Dictionary Management** tool, then close the form.

Step 7: Assigning Values to Parameters (Recipe and Workflow Designer)

Now, you can assign values to your parameters during master recipe or master workflow creation in Recipe and Workflow Designer. This is done in the **Parameter Panel**. To open the **Parameter Panel**, click the parameter button which is displayed in the upper right corner of your phase building block.
The **Parameter Panel** displays a list of all parameters with their properties. Within this list you can edit the values of the parameters of the phase.

TIP

Please note that the parameter button is only available when a phase uses parameters. The button will then display the number of parameters assigned to your phase.

Example

To assign values to the parameters of a phase in Recipe and Workflow Designer, proceed as follows:

1. In Recipe and Workflow Designer, create a master recipe or master workflow with more than one phase.
2. Replace one of the phases with your **MYC_StringValue** phase.
3. Now, the system displays **1** in the upper right corner of the phase building block. The number indicates the available parameters of the phase.
4. Click the parameter button (here: **1**) in the upper right corner of the phase building block to open the **Parameter Panel**.
5. As **Instruction text** type the following value: **Please enter a string**.
6. Save the phase.

7. Close Recipe and Workflow Designer.

TIP

The same applies to dynamic parameter classes once they have been added to a phase.

Step 8: Using Parameters during Phase Execution

When executing a recipe or workflow in the Production Execution Client, your phase executor can access the values of the phase parameters. How this is done depends on the process parameter type:

- For **static** process parameters, the values are accessed via the **usage identifier** that was defined when you assigned your parameter class to your phase (see section "Step 4: Assigning Parameter Classes to Phases" (page 104)).
- For **dynamic** process parameters, the values are accessed via the **internal identifier** returned by the call of *IS88ProcessParameterBundle.getInternalIdentifierOfProcessParameter(usageIdentifier)* (see section "Step 5: Assigning Dynamic Parameter Classes to Phases" (page 105)).

There is a difference between the parameters defined in a master recipe or master workflow during design time and the parameters used during execution (at runtime). Parameters can contain references to output data of other phases or expressions, the runtime parameters contain the evaluated values of the corresponding runtime phase. Thereby each instance of a runtime phase has its own runtime parameter instances, which can differ from each other.

In the **Preview** status, there is no runtime phase. That is why only the parameters defined during design time are available in the **Preview** status. If a parameter contains references or expressions, their values are not available.

In order to get the correct parameters for each status, the *AbstractPhaseExecutor* class provides the *getProcessParameterData(Class<T> type, String name)* method. The *Type* parameter is the class of your parameter class. Depending on the phase's status (**Preview**, **Active**, **Completed**), the method returns the correct parameters. In the **Preview** status, this will be the design time parameters; in the **Active** and **Completed** statuses, the runtime parameters will be returned. Use *getProcessParameterData(Class<T> type, String name)* to ensure the usage of the correct parameters. Please keep in mind that the corresponding methods of *IMESPhase* will return the design time parameters and not the runtime parameters.

During implementation of your phase executor**,** you know which parameter you expect. Thus the specific Java access bean for the parameter instance can be used.

getProcessParameterData() returns the parameter class-specific data of the parameter instance (see "Example" (page 112)).

TIP

- We recommend to use the *AbstractPhaseExecutor.getProcessParameterData(Class<T> type, String name)* method to get the correct parameter at runtime in any cases.
- Do not use *getPhase().getParameter().getAssociatedProcessParameterData()*, *getPhase().getParameter(String name).getAssociatedProcessParameterData()*, or *getPhase().getParameters()* to get the parameter at runtime. In any case, this method returns the design time parameters and not the runtime parameters.

Example

Add a new method to your *RtPhaseExecutorMYCStringValue* phase executor.

TIP

Do not forget to add the required imports:
import com.rockwell.mes.apps.testdata.recipe.IMESParamInstructionText;

Example

```
/**
 * Reads the string value of the parameter and displays it in the edit field
 *
 * @param label the Label field to set the text
 */
private void setInstructionTextByParameter(JLabel label) {
  MESParamInstructionText paramInstructionTextData =
    getProcessParameterData(MESParamInstructionText.class, "instruction text");
  label.setText(paramInstructionTextData.getText() + " (" + getPhase().getName() + ")");
}
```

You can call your new method in the *createPhaseComponent()* method after the label has been created:

- Replace `label.setText(getPhase().getName());` with `setInstructionTextByParameter(label);`
- And add the necessary imports:
  import com.rockwell.mes.parameter.test.instructiontext.MESParamInstructionText;

Chapter 8

Using Material Parameters

This section contains general information about using material parameters in phases. This includes defining the constraints for the phase, assigning material parameters to the phase, and using them during execution.

At the end of the section we will have extended our **MYC_StringValue** phase created in section "Creating a Phase Building Block" (page 33) to display information of a material parameter called **Required input**.

To add a material parameter to a phase, perform the following steps:

1. Define the material constraints (page 113) for the phase in order to define the required/allowed number of input and output material parameters.
2. Use the material parameters during phase execution (page 114) in order to actually work with the assigned materials.
3. Use the material parameters when creating a recipe/workflow (page 114) by assigning input and output materials to an instance of your phase in Recipe and Workflow Designer.

Step 1: Defining Material Constraints of a Phase

Material constraints of a phase are defined in the XML description of the phase building block (see section "Step 1: Creating an XML Description of the Phase Building Block" (page 33)).

Example

To define material constraints of the **MYC_StringValue** phase, adjust the XML description of "Step 1: Creating an XML Description of the Phase Building Block" (page 33).

In order to define that your **MYC_StringValue** phase has exactly one material input parameter, modify the XML description as listed below (only the relevant lines are shown):

```
<MaterialInputMin>1</MaterialInputMin>
<MaterialInputMax>1</MaterialInputMax>
<MaterialOutputMin>0</MaterialOutputMin>
<MaterialOutputMax>0</MaterialOutputMax>
<ATDefinitionPrefix>MYC</ATDefinitionPrefix>
```

Then, re-create your **MYC_StringValue** phase with the phase generator.

Step 2: Using Material Parameters during Phase Execution

When executing a recipe or workflow in the Production Execution Client, you can access the material parameters of a phase in your phase executor by means of the *getMaterialParameters()* method of *IMESPhase*.

Example

Set the material parameters list to a text box.

```
@Override
public JComponent createPhaseComponent() {
  ...
  List<IMESMaterialParameter> matLis = getPhase().getMaterialParameters();
  textField.setText(matLis.toString());
  ...
}
```

Step 3: Assigning Material Parameters (Recipe and Workflow Designer)

Now, you can assign material parameters to your phase during recipe or workflow creation in Recipe and Workflow Designer. This is done using the Setlist.
The material parameters assigned to a phase can be edited in the **Parameter Panel**. To open the **Parameter Panel**, click the material buttons which will be displayed in the upper left corner (for input parameters) and in the lower left corner (for output parameters) of your phase building block.

TIP

Please note that the material buttons are only available if the phase uses material parameters. The buttons will then display the numbers of the inputs and outputs assigned to your phase.

Example

To assign material parameters to a phase in Recipe and Workflow Designer, proceed as follows:

1. In Recipe and Workflow Designer, create a master recipe or master workflow with more than one phase.
2. Replace one of the phases with your **MYC_StringValue** phase.
3. Now, the system displays **0** in the upper left corner of the phase building block. The number indicates the number of assigned material parameters (input) of the phase.
4. Use the Setlist to assign a material input parameter to the phase.
5. Click the material button in the upper left corner of the phase building block to open the **Parameter Panel**.

6. You can optionally supply a planned quantity.
7. Save the phase.
8. Close Recipe and Workflow Designer.

Chapter 9

Using Equipment Requirement Parameters

This section contains general information about using equipment requirement parameters in phases. This includes defining the constraints for the phase, assigning equipment requirement parameters to the phase, and using them during execution.

To add an equipment requirement parameter to a phase, perform the following steps:

1. Define the equipment constraints (page 117) for the phase in order to define the required/allowed number of equipment requirement parameters.
2. Use the equipment requirement parameters during phase execution (page 117).
3. Use the equipment requirement parameters when creating a recipe/workflow (page 118) by assigning equipment requirement parameters to an instance of your phase in Recipe and Workflow Designer.

Step 1: Defining Equipment Constraints of a Phase

Equipment constraints of a phase are defined in the XML description of the phase building block (see section "Step 1: Creating an XML Description of the Phase Building Block" (page 33)).

Example

To define equipment constraints, adjust the XML description of "Step 1: Creating an XML Description of the Phase Building Block" (page 33).

To define that your phase needs exactly one equipment requirement parameter, modify the XML description as listed below (only the relevant lines are shown):

```
<EquipmentRequirementMin>1</EquipmentRequirementMin>
<EquipmentRequirementMax>1</EquipmentRequirementMax>
```

Then, re-create your phase with the phase generator.

Step 2: Using Equipment Requirement Parameters during Phase Execution

When executing a recipe or workflow in the Production Execution Client, you can easily check an equipment entity you wish to use against the configured equipment requirement parameters in your phase executor by means of the *checkRequirements()* method of the *IS88EquipmentExecutionService*. For this purpose, pass the equipment entity and the current runtime phase to this method. Its return value indicates whether the entity is suitable or not.

Example

Check the equipment requirement parameters and display an error message if the entity is not suitable

```
private boolean checkRequirements(IMESS88Equipment equipment) {
  IS88EquipmentExecutionService eqmS88ExecutionService =
                       ServiceFactory.getService(IS88EquipmentExecutionService.class);

  EquipmentReqValidationErrorResult result =
            eqmS88ExecutionService.checkRequirements(equipment, getRtPhase());

  if (result != null) {
    // error case
    showErrorDialog(result.getI18nErrorMessage());
  }

  return result == null;
}
```

Step 3: Assigning Equipment Requirement Parameters (Recipe and Workflow Designer)

Now you can assign equipment requirement parameters to your phase during recipe or workflow creation in Recipe and Workflow Designer. This is done using the Setlist. PharmaSuite only supports equipment classes as equipment requirements. However, a class requirement can be extended by adding property types. They can be assumed as further subordinate requirements.
The equipment requirement parameters assigned to a phase can be edited in the **Parameter Panel**. To open the **Parameter Panel**, click the equipment button which will be displayed in the upper left corner of your phase building block.

Handling S88 Equipment-related Features

This section describes how phases handle features related to S88 equipment such as creating logbook entries for S88 equipment (page 119), modifying the values of runtime properties of S88 equipment (page 121), handling of equipment entity groups (page 121), and handling of generated equipment entities (page 121).
For printing equipment entity labels, see also section "Printing Equipment Entity Labels During Execution" in chapter "Changing or Adding New Labels" in Volume 2 of the "Technical Manual Configuration and Extension" [A2] (page 175).

Adding S88 Equipment-related Features

This section describes how phases can create logbook entries related for S88 equipment. The entries are visible in the **Logbook** tab of equipment entities in Data Manager.

The API of *IS88EquipmentLogbookService* provides several methods to create logbook entries for S88 equipment. The following methods are intended to be used directly by a phase:

writeLogbookEntryOfCategoryPhase

```
/**
 * Create a logbook entry with {@code Category#PHASE} category and information string.
 * The logbook entry will be written only if equipment is logbook enabled.
 * The logbook entry is saved by this method.
 *
 * @param equipment the given S88 equipment entity
 * @param rtPhase the {@link IMESRtPhase} that create the logbook entry
 * @param information the information string to be written into the logbook
 * @param signature the {@link IMESSignature} attached to the logbook entry
 * @return the created logbook entry or {@code null} if no logbook entry was created
 */
public IMESS88EquipmentLogbook writeLogbookEntryOfCategoryPhase(
          IMESS88Equipment equipment, IMESRtPhase rtPhase, String information,
          IMESSignature signature);
```

writeLogbookEntryOfCategoryManual

```
/**
 * Create a logbook entry with {@code Category#MANUAL} category and information string.
 * The logbook entry will be written only if equipment is logbook enabled.
 * The logbook entry is saved by this method.
 *
 * @param equipment the given S88 equipment entity
 * @param rtPhase the {@link IMESRtPhase} that create the logbook entry
 * @param information the information string to be written into the logbook.
 * @param signature the {@link IMESSignature} attached to the logbook entry
 * @return the created logbook entry or {@code null} if no logbook entry was created
 */
public IMESS88EquipmentLogbook writeLogbookEntryOfCategoryManual(
          IMESS88Equipment equipment, IMESRtPhase rtPhase, String information,
          IMESSignature signature);
```

Both methods are very similar as they have the same set of arguments. They create a logbook entry for as S88 equipment in the context of a runtime phase. The methods differ in the category of logbook entry being created.

- The *writeLogbookEntryOfCategoryPhase* method is a general purpose method, it can be used by a phase at any time.
- The *writeLogbookEntryOfCategoryManual* method is intended for logbook entries created directly by a user.

If a not-null signature is passed as argument to one of the methods, the signature is attached to the logbook entry.

The example below demonstrates the usage of the phase completion signature to sign a phase in which a S88 equipment was used. The following assumptions apply: a phase completion signature is configured and the phase has a text box for an equipment identifier. The phase completion signature is created by *PhaseCompletionSignatureUIExtension*, which runs prior to the phase's *performComplete*.

Usage of phase completion signature

```
@Override
public Response performComplete() {
  createEquLogbookEntryForSignature();
  return new Response();
}
private void createEquLogbookEntryForSignature() {
  final String equipmentIdentifier = eqmIdentifier.getText();
  // An equipment identifier is needed
  if (StringUtils.isNotEmpty(equipmentIdentifier)) {
    final IMESS88Equipment equipment = loadEquipment(equipmentIdentifier);
    // An equipment is required
    if (equipment != null) {
      // Fetch completion signature
      final IS88ExecutionService executionService =
            ServiceFactory.getService(IS88ExecutionService.class);
      final IMESSignature phaseCompletionSignature =
            executionService.getCompletionSignatureForRtPhase(getRtPhase());
      final String informationText = phaseCompletionSignature != null
            ? "Complete with signature" : "Complete without signature";
      // Write logbook entry
      final IS88EquipmentLogbookService logbookService =
            ServiceFactory.getService(IS88EquipmentLogbookService.class);
      logbookService.writeLogbookEntryOfCategoryPhase(
            equipment, getRtPhase(), informationText, phaseCompletionSignature);
    }
  }
}
```

The information string being created is just an example. If you just wish to record a signature event, you can use null as value.

Modifying Runtime Properties

This section describes how a phase can modify the values of runtime properties of S88 equipment.

Please consider the following items when you modify a runtime property:

- A property can be of the **Runtime, Specification**, **Automation**, or **Historian** usage type. A phase may only modify runtime properties.
- Make sure that the equipment is bound to the phase before the phase modifies a runtime property. It is not sufficient that the equipment is loaded or identified.
- Make sure to save the property when a runtime property has been modified. Save only the property; saving the equipment entity would trigger an update of UI properties for unchanged properties.

To modify a runtime property of S88 equipment use the *setAttributeValue* method of the *IMESEquipmentProperty<T>* API.

Handling of Equipment Entity Groups

Phase building blocks can process groups of equipment entities. Since the group membership can be modified outside of a phase (e.g. with a repair use case in Data Manager - Equipment), it is essential that the phase works on the current group members.

Call *IMESS88Equipment.refreshAssignments()* in order to refresh the list of assigned descendants (i.e. all child entities). This will clear the cache of the children and reload them upon the next call of e.g. *getDescendants()*.
Additionally, a phase can call *IMESS88Equipment.refreshAncestorReferences()* to also refresh all parent properties (i.e. reload the ATRow) upwards the hierarchy.

Handling of Generated Equipment Entities

This section describes how equipment entities can be generated during execution with a phase building block. The generation is based on a template equipment entity. Generated equipment entities can later be identified and used for further processing.

Equipment entities of the **Hybrid** equipment type are related to materials. Prior to their generation and identification, a phase can execute material-/sublot-related checks specified within the phase. The results of some checks can be overridden with an operator's signature if they fail during phase execution. For details, see "Overriding Check Results with a Signature" (page 131).

PharmaSuite provides specific service calls to

- generate equipment entities based on a template equipment entity only (page 122),
- generate equipment entities based on a template equipment entity and a sublot as a **Hybrid** equipment entity (page 123), and
- identify generated **Hybrid** equipment entities (page 125).

Generating Equipment Entities Based on a Template Equipment Entity

In this case, a phase uses a template equipment entity directly to generate equipment entities.

The API of *IS88EquipmentExecutionService* provides a method to generate a specified number of equipment entities from a template equipment entity:

- generateEntitiesFromTemplate

The method is intended to be used directly by a phase. For details, see "PharmaSuite-related Java Documentation" [C1] (page 175).

generateEntitiesFromTemplate

```
/**
 * Generates the given number of entities using the specified template entity.
 */
public List<IMESS88Equipment> generateEntitiesFromTemplate(
       IS88TemplateEntity templateEntity, long numberOfGeneratedEntities,
       boolean forceTargetStatus)
       throws MESS88TemplateEntityArchivedException;
```

The method has the following parameters:

- **templateEntity** specifies the template entity to be used.
- **numberOfGeneratedEntities** specifies the number of entities to be generated.
- **forceTargetStatus** specifies if the status of the generated entities shall be the same as the status of the template equipment entity.

The method has the following return value:

- List of generated equipment entities.
  In some cases, e.g. if an identifier of an equipment entity to be generated is already used, the generation terminates and the result contains the equipment entities that have been generated so far.

If the template equipment entity is in the **Archived** status, the method throws an *MESS88TemplateEntityArchivedException* exception and returns an empty list.

Generating Equipment Entities Based on a Sublot and a Template Equipment Entity

In this case, a phase uses a sublot and the template equipment entity assigned to the sublot's material to generate equipment entities. The equipment entity can be generated for two different reasons:

- The sublot is **consumed immediately**. The sublot identifier of the identified sublot is stored in the property of the **Base Sublot (RS)** purpose of all generated equipment entities.
- The sublot is **split** according to the specified number of generated equipment entities, i.e. one new sublot is generated per generated equipment entity. Again, the sublot identifier of the identified sublot is stored in the property of the **Base Sublot (RS)** purpose of all generated equipment entities.
  Additionally, for each split sublot, its identifier is stored in the property of the **Current Sublot (RS)** purpose of the newly generated equipment entity.

The API of *IS88EquipmentExecutionService* provides two methods to generate a specified number of equipment entities from a sublot and the related template equipment entity:

- generateEntitiesFromMaterialWithImmediateConsumptionOfBaseSublot
- generateEntitiesFromMaterialWithSplitOfBaseSublot

Before the sublot is processed, configurable and fixed checks can be executed. The checks are grouped in the *IIdentifactionCheckSuite* check suite and provided to the service call for being executed. In case of failed checks, they are returned as a result of the service call and the generation terminates. If at least one overridable check has failed in a first call and the operator has signed for the failed check, the check can be skipped in a second call. For details, see "Overriding Check Results with a Signature" (page 131).

The methods are intended to be used directly by a phase. For details, see "PharmaSuite-related Java Documentation" [C1] (page 175).

generateEntitiesFromMaterialWithImmediateConsumptionOfBaseSublot

```
/**
 * Generates (hybrid) entities from the given base sublot and performs its immediate
 * consumption according to the given parameters.
 * In case the base sublot's quantity is zero in the end it will be (logically) deleted.
 */
public IEquipmentEntityGenerationFromMaterialResult
       generateEntitiesFromMaterialWithImmediateConsumptionOfBaseSublot(
       Sublot baseSublot, long numberOfGeneratedEntities,
       MeasuredValue quantityToConsumePerEntity,
       IIdentificationCheckSuite checkSuite, IESignatureExecutor signaturesException,
       TransactionSubtype transactionSubtype, boolean correctInventory,
       IMESRtPhase rtPhase)
       throws MESIncompatibleUoMException, MESQuantityMustNotBeNegativeException,
       MESSublotOperationNotAllowedException, MESS88TemplateEntityArchivedException;
```

generateEntitiesFromMaterialWithSplitOfBaseSublot

```
/**
 * Generates (hybrid) entities from the given base sublot and performs a split
 * of the base sublot according to the given parameters.
 * In case the base sublot's quantity is zero in the end it will be (logically) deleted.
 */
public IEquipmentEntityGenerationFromMaterialResult
       generateEntitiesFromMaterialWithSplitOfBaseSublot (
       Sublot baseSublot, long numberOfGeneratedEntities,
       MeasuredValue quantityToConsumePerEntity,
       IIdentificationCheckSuite checkSuite, IESignatureExecutor signaturesException,
       TransactionSubtype transactionSubtype, boolean correctInventory,
       IMESRtPhase rtPhase)
       throws MESSublotOperationNotAllowedException, MESIncompatibleUoMException,
       MESQuantityMustNotBeNegativeException, MESSublotSplitPreconditionException,
       MESQuantityExceedsAvailableQuantityException,
       MESS88TemplateEntityArchivedException;
```

Both methods share the following set of parameters:

- **baseSublot** specifies the identifier of the sublot to be used to generate equipment entities. A template equipment entity must be assigned to the sublot's material (part).
- **numberOfGeneratedEntities** specifies the number of entities to be generated.
- **quantityToConsumePerEntity** specifies the quantity to be consumed from the base sublot per generate equipment entity.
- **checkSuite** specifies a configurable check suite to be executed before the sublot is identified. The check suite can consist of checks configured by process parameters and/or fixed checks. The *IIdentificationCheckSuite* check suite contains the required checks. For details, see "Creating a Sublot-specific Check Suite" (page 128).
- **signaturesException** specifies the signature executor required to skip failed checks.
- **transactionSubtype** specifies a transaction subtype of the **TransactionSubtype** choice list. This parameter is optional for the generation of equipment entities with a split of the base sublot and mandatory for the generation of equipment entities with immediate consumption.
- **correctInventory** specifies whether the inventoried quantity is corrected if required.
  If set to **true** and the quantity of the base sublot is not sufficient to generate all equipment entities, the inventory is corrected automatically upfront to the necessary value in order to ensure the successful generation of the equipment entities.
  If set to **false**, an error occurs and no equipment entities are generated.

- **rtPhase** specifies the runtime phase needed to create appropriate entries in the equipment entity logbook.

Both methods have the following return value:

- Instance of *IEquipmentEntityGenerationFromMaterialResult*, which consists of three parts:
  1. An **IssueType**:
     - CHECKS_NOT_SUCCESSFUL: at least one check of the check suite failed.
     - TEMPLATE_ENTITY_NOT_FOUND: no template equipment entity could be found for the base sublot.
     - NOT_ENOUGH_QUANTITY_AVAILABLE: the quantity of the base sublot is insufficient.
     - NOT_ALL_ENTITIES_GENERATED: the specified number of equipment entities could not be generated.
     - NOT_ALL_CHECK_SUITE_EXCEPTIONS_SIGNED: an operator has already signed for check suite exceptions that occurred during a previous check, but there are new exceptions that require to be signed.
  2. A list of generated equipment entities.
  3. The check suite with the executed material-/sublot-related checks, their results, and the overall check result.

Both methods throw a set of possible exceptions, mainly triggered by internally used inventory service calls. For details, see "PharmaSuite-related Java Documentation" [C1] (page 175).

Identifying a Generated Hybrid Equipment Entity

Generated **Hybrid** equipment entities are material- and sublot-related. The material and sublot information is stored in the properties of the **Base Sublot (RS)** and **Current Sublot (RS)** (if applicable) purposes of the generated equipment entities. The phase can identify single generated equipment entities or equipment entity groups with **Hybrid** and non-hybrid generated equipment entities.

The API of *IMaterialEquipmentService* provides three methods to perform a complete identification of equipment entities:

- identifyMaterialForEquipmentGroup
- consumeIdentifiedSublots
- undoIdentifiedMaterial

The methods are intended to be used directly by a phase. For details, see "PharmaSuite-related Java Documentation" [C1] (page 175).

identifyMaterialForEquipmentGroup

```
public List<EquipmentWithMaterialIdentificationResult>
       identifyMaterialForEquipmentGroup(//
       IMESS88Equipment equipment, //
       IidentificationCheckSuitePhaseCustomized checkSuiteBaseSublot, //
       IidentificationCheckSuitePhaseCustomized checkSuiteCurrentSublot, //
       IidentificationCheckSuitePhaseCustomized checkSuiteAllCurrentSublots, //
       Map<OrderStepInput, List<Batch>> osiAllocBatchesMap //
       IESignatureExecutor signatureExecutor)
       throws MESTransitionFailedException, MESException, DatasweepException;
```

The *identifyMaterialForEquipmentGroup* method identifies the material of a **Hybrid** equipment entity (single or group member). All passed check suites are executed. For each equipment entity, the results are stored in the **EquipmentWithMaterialIdentificationResult** object and aggregated in a list of such objects.
If all checks have passed successfully, the phase can consume the identified sublots of the generated equipment entities with the *consumeIdentifiedSublots* method and use the list of **EquipmentWithMaterialIdentificationResult** objects as input.
If at least one overridable check has failed in a first call, not all sublots of the generated equipment entities are identified. In case the operator has signed for the failed check or cancels the identification process, the phase can revoke the identification of the already identified sublots with the *undoIdentifiedMaterial* method and can call the *identifyMaterialForEquipmentGroup* method for a second time, now with a signature executor.

The method has the following parameters:

- **equipment** specifies an already identified equipment entity or equipment entity group.
- **checkSuiteBaseSublot** specifies a configurable check suite of material- and sublot-related checks. The check suite can consist of checks configured by process parameters and/or fixed checks.
  Each check is executed on the sublot stored in the property of the **Base Sublot (RS)** purpose of the generated equipment entity.
  If no checks are required, the check suite is null.
  For details, see "Creating a Sublot-specific Check Suite" (page 128).
- **checkSuiteCurrentSublot** specifies a configurable check suite of material-and/sublot-related checks. The check suite can consist of checks configured by process parameters and/or fixed checks.
  Each check is executed on the sublot stored in the property of the **Current Sublot (RS)** purpose of the generated equipment entity.

If no checks are required, the check suite is null.
For details, see "Creating a Sublot-specific Check Suite" (page 128).

- **checkSuiteAllCurrentSublots** specifies a configurable check suite of sublot-related checks. The check suite can consist of checks configured by process parameters and/or fixed checks.
  Each check is executed on all sublots stored in the properties of the **Current Sublot (RS)** purpose of the generated equipment entities.
  If no checks are required, the check suite is null.
  This check suite and the *PhaseCurrentSublotQuantityCheck* check allow to check if the planned quantity of an order step input matches the sum of the quantities of the related current sublots. The *PhaseCurrentSublotQuantityCheck* class is located in the Commons folder for phase development and can therefore be accessed from any other building block. The check is a wrapper class for the original check with the *InventoryServiceProvider.CHECK_ALL_CURRENT_SUBLOT_QUANTITY_BEAN* bean.
  For details, see "Creating a Sublot-specific Check Suite" (page 128).
- **osiAllocBatchesMap** consists of the main order step input objects and a list of allocated batches. The map is passed to the check suites. The phase must provide this information.
- **signatureExecutor** specifies the signature executor required to skip failed checks.

The method has the following return value:

- List of instances of *EquipmentWithMaterialIdentificationResult*, which is used to transfer identification results from the phase to the service calls and vice versa. The list contains the check suite results, references to identified sublots, and references to related order step input objects. Two scenarios can occur:
  - Check(s) failed: each object of the list provides information about the executed check suite results (error, exception, or warning) per equipment entity. The phase can retrieve the information with appropriate methods.
  - Checks passed: each object of the list provides references to the identified current sublot and its related order step input per equipment entity. The information can later be used to consume the sublots during equipment binding in the phase with the *consumeIdentifiedSublots* method.

The method throws a set of possible exceptions, mainly triggered by internally used inventory service calls. For details, see "PharmaSuite-related Java Documentation" [C1] (page 175).

consumeIdentifiedSublots

```
public void consumeIdentifiedSublots
      (List<EquipmentWithMaterialIdentificationResult> resultList) throws MESException;
```

The *consumeIdentifiedSublots* method consumes the quantities of all successfully identified current sublots. The sublots can be retrieved from the list of **EquipmentWithMaterialIdentificationResult** objects. The sublots' quantity is set to zero and the sublots are logically deleted. They are no longer visible in the Production Management Client.
The method is transactional over all referenced sublots.

The method has the following parameter:

- **resultList** is a list of **EquipmentWithMaterialIdentificationResult** objects. The list is the result of a preceding call of *identifyMaterialForEquipmentGroup()*.

The method throws an *MESException* exception, which contains an exception mainly triggered by internally used inventory service calls. For details, see "PharmaSuite-related Java Documentation" [C1] (page 175).

undoIdentifiedMaterial

```
public void undoIdentifiedMaterial(List<EquipmentWithMaterialIdentificationResult>
        resultList);
```

The *undoIdentifiedMaterial* method revokes the identification of the current sublots referenced in the list of **EquipmentWithMaterialIdentificationResult** objects. Afterwards, the sublots can be identified again. Use this method in the following situation: a first call of *identifyMaterialForEquipmentGroup()* returns at least one failed check and some sublots of the generated equipment entities are already identified, Then, the phase displays the failed check(s) in a dialog and the operator can sign the exceptions and repeat the identification.
The method is transactional over all referenced sublots.

The method has the following parameter:

- **resultList** is a list of **EquipmentWithMaterialIdentificationResult** objects. The list is the result of a preceding call of *identifyMaterialForEquipmentGroup()*.

Creating a Sublot-specific Check Suite

For the generation of equipment entities and their identification, a phase can define sublot-specific checks in a check suite. A check suite can consist of checks configured by process parameters and/or fixed checks.

Checks used by the check suite for generating equipment entities based on a sublot and a template equipment entity (page 123):

- Fixed checks
  - InventoryServiceProvider.SUBLOT_CHECK_SUBLOT_NOT_IN_CONTAINER

  - InventoryServiceProvider.SUBLOT_CHECK_SUBLOT_BLOCKED_BY_PROCESSING_BEAN
  - IWDOrderStepInputService.SUBLOT_CHECK_DELETED_BEAN
  - InventoryServiceProvider.CHECK_QUANTITIES_FOR_ENTITY_GENERATION_BEAN.
    Checks that the sublot quantity and the given quantity is valid: quantity > 0, UoM is defined, not %, and convertible to UoM of sublot quantity.
- Configurable checks:
  - PhaseBatchStatusIdentificationCheck<version>
  - PhaseBatchExpiryDateIdentificationCheck<version>
  - PhaseBatchRetestDateIdentificationCheck<version>

Checks used by the check suites for identifying a generated **Hybrid** equipment entity (page 125):

- Fixed checks (of the *checkSuiteCurrentSublot* check suite):
  - InventoryServiceProvider.SUBLOT_CHECK_BOM_SPECIFIC_MATERIAL_BEAN
  - InventoryServiceProvider.SUBLOT_CHECK_SUBLOT_PRODUCED_OTHER_OS_BEAN
  - InventoryServiceProvider.SUBLOT_CHECK_SUBLOT_BLOCKED_BY_PROCESSING_BEAN
  - InventoryServiceProvider.SUBLOT_CHECK_MATERIAL_INTERMEDIATE_OSI_BEAN
  - IWDOrderStepInputService.SUBLOT_CHECK_DELETED_BEAN
  - IWDOrderStepInputService.SUBLOT_CHECK_MATERIAL_INTERMEDIATE_NOT_REPLACED_BEAN
- Configurable checks (of the *checkSuiteBaseSublot*, *checkSuiteCurrentSublot*, and *checkSuiteAllCurrentSublots* check suites):
  - PhaseBatchStatusIdentificationCheck<version>
  - PhaseBatchExpiryDateIdentificationCheck<version>
  - PhaseBatchRetestDateIdentificationCheck<version>
  - PhaseAllocationIdentificationCheck<version>
  - PhaseCurrentSublotQuantityCheck<version> (only in the *checkSuiteAllCurrentSublots* check suite)

To support the creation and usage of check suites, PharmaSuite provides some (helper) classes in the Equipment Tracking phase package, especially the following two classes:

- AbstractPhaseIdentificationCheckSuiteSublot<version>
- AbstractPhaseIdentificationCheckSuiteSublotList<version>

For more details about the classes and their usage, see the following example of a check suite:

MyCheckSuite

```
public class MyCheckSuite extends AbstractPhaseIdentificationCheckSuiteSublot0600 {
  public static final String MSG_PACK = "PhaseEqmEqGenFromMat0100";

  public GenerateFromMaterialCheckSuite0100(List<? extends IS88Parameter> config,
                                            Map<String, Object> context) {
    super(config, context);
  }

  @Override
  protected String getMessagePack() {
    return MSG_PACK;
  }

  @Override
  protected void addFixedChecks(Map<String, Object> context) {
    addIdentificationCheckDAO(CheckType.ERROR,
       InventoryServiceProvider.SUBLOT_CHECK_SUBLOT_NOT_IN_CONTAINER, null, null,
       context);
    ...
  }
  private void addIdentificationCheckDAO(CheckType type, String beanName,
               String argCheckKey, List<Object> params, Map<String, Object> context) {
    getIdentChecks().add(new PhaseIdentificationCheckSublot0600(type, beanName,
                         argCheckKey, null, params, context));
  }
  @Override
  protected void addConfigurableChecks(List<? extends IS88Parameter> config,
                                       Map<String, Object> context) {
    MESParamBatchCheckDef0600 paramValues = getIdentCheckParam(config);
    for (IS88Parameter param : config) {
      if (param instanceof IMESProcessParameterInstance) {
        IMESProcessParameterInstance paramInstance =
                            (IMESProcessParameterInstance) param;
        IMESProcessParameterData data =
                            paramInstance.getAssociatedProcessParameterData();
        if (data instanceof MESParamExcpEnableNDef0100 &&
                            isEnabled((MESParamExcpEnableNDef0100) data)) {
          MESParamExcpEnableNDef0100 validationDef =
                                    (MESParamExcpEnableNDef0100) data;
          PhaseCheckConfiguration0600 checkConfig = new
                    PhaseCheckConfiguration0600(getMessagePack(), validationDef);
            String paramInstanceName = paramInstance.getIdentifier();
          if (paramInstanceName.equals
                   (MaterialCheckUIConstants0600.PARAMETER_BATCH_STATUS) &&
                            paramValues.getMinBatchStatus() != null) {
            List<Object> params = null;
            params = new ArrayList<Object>();
            params.add(paramValues.getMinBatchStatus());
            getIdentChecks().add(new
               PhaseBatchStatusIdentificationCheck0600(CheckType.EXCEPTION,
                         paramInstanceName, checkConfig, params, context));
      ...
    }

  private MESParamBatchCheckDef0600 getIdentCheckParam(
                                    List<? extends IS88Parameter> config) {
    for (IS88Parameter param : config) {
      if (param instanceof IMESProcessParameterInstance) {
         IMESProcessParameterInstance paramInstance =
                    (IMESProcessParameterInstance) param;
         IMESProcessParameterData data =
                    paramInstance.getAssociatedProcessParameterData();
      if (data instanceof MESParamBatchCheckDef0600) {
        return (MESParamBatchCheckDef0600) data;
      }
  }
    }
    return null;
  }
}
```

Overriding Check Results with a Signature

An operator can override failed checks with a signature. Examples are an expiry date or retest date in the past. Checks are performed when **Hybrid** equipment entities are generated and identified. The mechanism works basically in the same way for both generation and identification:

- Call the appropriate service (*IS88EquipmentExecutionService*, *IMaterialEquipmentService*) with the signature executor-specific parameter set to null. The defined checks are performed and unless there are errors or exceptions, the appropriate operation is executed.
- If at least one overridable check failed, the service call is aborted and the check suite with the failed check is returned.
- The phase displays the failed checks in a dialog and the operator can sign the exception.
  The phase constructs an instance of *IESignatureExecutor* (normally in the *exceptionTransactionCallback()*) with the signature data, provides the necessary information of the check(s), and performs the service call a second time with the signature executor set. The service call repeats the checks but skips the ones identified by the signature executor and saves the signature executor. If there are no other errors, the appropriate operation is executed.

Example

The example shows how to use a service call to provide a check suite and (in a second call) a signature executor. A phase can use the service calls in the following way:

RtPhaseExcutorMyPhase

```
...
private IPhaseIdentificationCheckSuite0600 sublotCheckSuite;
...
@Override
public JComponent createPhaseComponent() {
  if (Status.ACTIVE.equals(getStatus())) {
    sublotCheckSuite = new MyCheckSuite(getParameters(),
                         Collections.EMPTY_MAP);
    }
  return super.createPhaseComponent();
}
...
private void setupSignatureExecutor() {
  for (int i = 0; i < sublotCheckSuite.getCheckSuite().size(); i++) {
    IIdentificationCheck sublotCheck = sublotCheckSuite.getCheckSuite().get(i);
      if (!sublotCheck.isOk()) {
        signatureExecutor.addInfo(
          new ArrayList<String>(sublotCheck.getExceptionList()));
      }
  }
}

private boolean doGenerateEntitiesFromSublot() {
  IIdentificationCheckSuite checkSuite;
  ...
  if (signatureExecutor != null) {
    // Second call with exception signed
    setupSignatureExecutor();
    checkSuite = sublotCheckSuite.getCheckSuite();
  } else {
    // First call
    checkSuite = sublotCheckSuite.createCheckSuite(Collections.EMPTY_LIST);
  }
  IEquipmentEntityGenerationFromMaterialResult result;
  if (getModel().getParamSplitSublot()) {
    try {
      result = eqmExecService.generateEntitiesFromMaterialWithSplitOfBaseSublot(
               sublot, numberOfGeneratedEntities, quantityPerEntity, checkSuite,
               signatureExecutor, transactionSubtype, correctInventory);
    } catch (MESSublotOperationNotAllowedException | MESIncompatibleUoMException |
             MESQuantityMustNotBeNegativeException |
             MESQuantityExceedsAvailableQuantityException |
             MESSublotSplitPreconditionException |
             MESS88TemplateEntityArchivedException e) {
      throw new MESRuntimeException(e);
    }
  }
  if (!checkResult(result)) {
    return false;
  }
  List<IMESS88Equipment> generatedEntities = result.getGeneratedEntities();
  ...
  return true;
}
```

For the equipment entity identification, you can use the *setupSignatureExecutorForIdentification()* helper method available in *IMaterialEquipmentService*.

setupSignatureExecutorForIdentification

```
public void setupSignatureExecutorForIdentification(
       IESignatureExecutor signatureExecutor,
       List<EquipmentWithMaterialIdentificationResult> resultList);
```

The helper method allows to set up the signature executor with the information of the (signed) exceptions extracted from the result list of the *identifyMaterialForEquipmentGroup* method. The signature executor is not saved in this method, only the necessary information is added.

The method has the following parameters:

- **signatureExecutor** specifies the signature executor constructed in the phase.
- **resultList** specifies the result list returned by the first service call.

Chapter 11

Adding Exception Handling for Phase-internal Checks

➢ Does not apply to server-side phases running on the OE server.
For server-side phases, please refer to "Specific Information on Phases Running on the PharmaSuite OE Server" (page 69).

This section contains general information about exception handling for phase-internal checks. Phase-internal checks lead to system-triggered exceptions. Exception handling includes checking limits during phase execution, displaying phase-specific exception messages or error messages, recording exceptions, and determining whether an exception has been signed before phase execution can continue.

At the end of the section we will have extended our **MYCStringValue** phase executor created in section "Creating a Phase Building Block" (page 33) to perform a limit check for the user input upon phase completion. The check allows an operator to sign limit violations and prevents the completion of the phase unless the operator has signed in case the limit check has failed.

To add exception handling for phase-internal checks to a phase, perform the following steps:

1. Implement a check (page 135) for the phase-specific exception in your phase executor.
2. Trigger the check during phase execution (page 136) in order to determine whether an exception has occurred.
3. Display an exception message (page 138) in case an exception has occurred.
4. Determine whether an exception has been signed (page 140) in order to allow the completion of the phase.

Step 1: Implementing a Phase-specific Check

Field validation is controlled by the phase itself, thus the check has to be implemented in the corresponding phase executor (see section "Step 5: Creating a Phase Executor (Java Code)" (page 49)).

TIP

We recommend to use process parameters in order to allow the recipe/workflow author to define the limits for your phase-specific check (see section "Creating and Using Process Parameters" (page 99)).

Example

In our example, we will check for an expected value that is hard-coded within the phase. If the operator provides a value that differs from the expected value (here: **check passed**), the check returns the **ERROR_EXCEPTION_TYPE** exception type. In case of a successful check, the return value is **NO_EXCEPTION**.

Add the constant for the expected value and the new method for the check to your *RtPhaseExecutorMYCStringValue* phase executor.

Example

```
/** the expected value */
private static final String EXPECTED_STR = "check passed";

/**
 * perform limit check - create an error on the current value and show
 * error dialog in case of limit violation
 *
 * @return the result of the limit check
 */
private ExceptionType performLimitCheck() {
  ExceptionType response = ExceptionType.NO_EXCEPTION;

  text = inputField.getText();
  if (!text.equals(EXPECTED_STR)) {
    // control limit violation
    response = ExceptionType.ERROR_EXCEPTION_TYPE;
  }

  return response;
}
```

Step 2: Triggering the Check during Phase Execution

It is the responsibility of the phase developer to trigger the checks at certain points, e.g. when the operator types his input value or before completing the phase. In case of an error, an exception dialog is displayed. Now, the operator can trigger an exception (see section "Step 3: Displaying an Exception Message" (page 138)). It must not be possible to complete the phase unless the operator has signed the exception (see section "Step 4: Determining whether an Exception Has Been Signed" (page 140)).

A check that takes place before the phase is completed can be implemented by overriding the *performCompletionCheck()* method.

TIP

Frequently, you may wish to trigger the check when leaving the text box in order to have immediate feedback. If a touch screen is used, we recommend to use *InputVerifier*. Thus, you do not have to take care of the different events needed to trigger the check depending on the fact whether the virtual keyboard is used or not.

Example

In our example, we check the operator input (against the expected value) before the phase will be confirmed. Thus, we prevent the phase from completing if an unexpected value has been provided.

To trigger the checks during execution of your *RtPhaseExecutorMYCStringValue* phase executor, proceed as follows:

1. In order to perform the check when the **Complete** button is executed, change the overridden *performCompletionCheck()* method. Now, the operator cannot confirm the phase unless the correct value has been entered. In step 4 (page 140), the behavior will be enhanced to take care of signed exceptions.

Example

```
/**
 * {@inheritDoc}
 *
 * @see com.rockwell.mes.apps.ebr.ifc.phase.IPhaseExecutor#performCompletionCheck()
 */
@Override
public boolean performCompletionCheck() {
  ExceptionType currentException = performLimitCheck();
  if (currentException != ExceptionType.NO_EXCEPTION) {
    // limit check failed
    showExceptionDialog();
    return false;
  }
  // limit check passed successfully
  return true;
}
```

2. In order to retrigger the check after the input value has been changed, add an *InputVerifier* to your text box and execute the check when the input changes.

Example

```
public JComponent createPhaseComponent() {
  ...
  if (getStatus() == Status.ACTIVE) {
    inputField.setInputVerifier(new ValidateFieldInputVerifier());
  ...
}

/**
 * InputVerifier to validate the field input in order to check for limit
 * violations (and trigger exception dialog)
 */
private class ValidateFieldInputVerifier extends InputVerifier {
  @Override
  public boolean verify(JComponent input) {
    if (!input.isShowing()) {
      // Don't trigger an exception if component is not visible
      return true;
    } else {
      ExceptionType response = performLimitCheck();
```

```
        if (response != ExceptionType.NO_EXCEPTION) {
          showExceptionDialog(response);
        }
      }
      // always allow to leave field
      return true;
    }
  }
```

In the example above, the *verify* method of *inputVerifier* checks whether a component is visible or not. If the component is not visible on the screen, another view is visible. Here, the *inputVerifier* is performed on an edit field in the phase panel of the execution.

Let us assume that an operator has entered a value and this field has the focus. Then, the operator clicks the **Navigator** button. The change to the Navigator triggers the *inputVerifier* method because the Navigator requests the focus. If the *verify* method of the *inputVerifier* returns false, the focus cannot be changed. So, no component in the Navigator gets the focus. The invisible field in the execution window still has the focus although the Navigator is visible. In this specific situation, requesting exceptions is not allowed. So, we recommend not to trigger an exception record if the component is not visible on the screen. The framework prevents the pop up of a dialog and also prevents requesting exception records without dialog in a condition in which it is no possible. *AbstractPhaseExecutor.displayException* returns false if it is not possible to request an exception record and *PhaseExceptionDialog.recordException* returns *IExceptionHandler.CLOSED_OPTION*.

TIP

We recommend to return true in the *verify* method of the *InputVerifier*, at least if the affected component is not visible on the screen (`input.isShowing() == false`).

Step 3: Displaying an Exception Message

The static *PhaseSystemTriggeredExceptionHandler.recordException()* method displays a dialog that forces an operator to record an exception. When the operator has tapped the **Exception** button, the method switches to the exception view, populates a message and risk class into the exception record, and waits for the operator to sign. The waiting is done by setting the *checkKey* marker. The marker indicates that there is an exception that has not yet been recorded or signed (see section "Step 4: Determining whether an Exception Has Been Signed" (page 140)).

TIP

If the operator decides not to sign the requested exception, he is forced to sign an "Exception canceled" exception related to the requested exception. With this exception, the system tracks the cancelation of the original exception and adds it to the batch report.

The **Cancel** button of the exception dialog can be enabled with the

LibraryHolder/apps-ebr-ifc.jar/CancelButtonEnabledOfExceptionDialog configuration key (see chapter "Configuration Keys of PharmaSuite" in Volume 4 of the "Technical Manual Configuration and Extension" [A3] (page 175)). Thus no exception is enforced at that time.

The *PhaseSystemTriggeredExceptionHandler* class provides only one style of dialogs. To display information dialogs, use *PhaseInfoDialog.showDialog()*; to display error dialogs, use *PhaseErrorDialog.showDialog()*.

Both dialog types provide only one button, which allows the operator to simply confirm the acknowledgment of the content. The dialogs differ only in style. However, we recommend that after an information dialog processing continues normally, while the error dialog stops normal processing. In this case, it is assumed that something severe has happened that cannot even be fixed by recording an exception.

Sometimes you may wish to ask the operator a question. For this purpose, use *PhaseQuestionDialog.showDialog()*.

TIP

The *PhaseSystemTriggeredExceptionHandler* sets a marker for unrecorded exceptions (please make sure to assign a unique key to each check) and supports different types.

You can also display an error dialog that does not display the **Exception** button to handle error situations that cannot be signed. For this purpose, use *PhaseErrorDialog.showDialog()*. Since this dialog does not set exception markers, make sure to prevent phase completion in your phase implementation.

Example

The method that displays the exception dialog is called when the **Confirm** button of the phase is executed (see section "Step 2: Triggering the Check during Phase Execution" (page 136)). In our case we set the risk category to **High**. The access privilege required for the exception signature is automatically retrieved from the privilege parameters assigned to the phase, according to the risk category (see also "What Is a Privilege Parameter?" (page 16)).

Example

```
/** The message pack */
private static final String MESSAGE_PACK = "ebr_RtPhaseExecutorMYCStringValue";

/** msg id for expected value violation */
private static final String MSG_ID_EXPECTED_VALUE_LIMIT_VIOLATION =
                     "expectedLimitViolation_Error";

/** key of the check key in order to mark exceptions as recorded */
private static final String EXPECTED_VALUE_CHECK_KEY = "checkKey";

/**
 * generating the corresponding message for the exception type and show the
 * exception dialog for the exception type with the specified message and key
 *
 * @return an integer indicating the option (@see ExceptionHandler) indicating
 * if the dialog was displayed or not due to improper system state or user has
 * clicked the cancel button of the dialog.


 */
private int showExceptionDialog() {
  // control limit violation, show dialog
  String msg = I18nMessageUtility.getLocalizedMessage(MESSAGE_PACK,
                MSG_ID_EXPECTED_VALUE_LIMIT_VIOLATION,
                new String[] { EXPECTED_STR, text });

  return PhaseSystemTriggeredExceptionHandler.recordException(this, dialogString, msg,
         IMESExceptionRecord.RiskClass.high, EXPECTED_VALUE_CHECK_KEY);
}
```

TIP

Please keep in mind that the **Cancel** button of the exception dialog can be enabled (see "Step 3: Displaying an Exception Message (page 138)"). The *recordException* method may also return the cancel option (*CANCEL_OPTION*).

To add the message displayed in the exception dialog, in Process Designer, create a new message (here: **ebr_RtPhaseExecutorMYCStringValue**) with its ID (here: **expectedLimitViolation_Error**) and text (here: **Expected value control limit has been violated: Expected ''{0}''.\nActual value: {1}.**).

TIP

Keep in mind to clear the messages (and Data Dictionary) cache of Process Designer.

Step 4: Determining whether an Exception Has Been Signed

Since it must not be possible to complete a phase if there are unsigned exceptions, we have to modify *performCompletionCheck()* accordingly (see section "Step 2: Triggering the Check during Phase Execution" (page 136)).

In order to check whether an exception has been signed, use the *isExceptionSigned(String checkKey)* method.

TIP

Even if the check was triggered by the **Confirm** button, you have to retrigger the completion of the phase after signing for the exception.

In addition, you can use the following methods to access all check keys used by a phase:

- *Set<String> getAllCheckKeys()*
- *Set<String> getCheckKeysSigned()*

- *Set<String> getCheckKeysNotSigned()*

Example

The exception dialog should not be displayed if an exception has already been signed and the value has not been modified since the last signature. Thus we use *isExceptionSigned()* and the "dirty flag" to ensure that a phase cannot be confirmed again after changing an already signed value to another invalid one.

Example

```
/**
 * {@inheritDoc}
 *
 * @see com.rockwell.mes.apps.ebr.ifc.phase.IPhaseExecutor#performCompletionCheck()
 */
@Override
public boolean performCompletionCheck() {
  ExceptionType currentException = performLimitCheck();
  if (currentException != ExceptionType.NO_EXCEPTION
                       && !isExceptionSigned(EXPECTED_VALUE_CHECK_KEY) {
    // limit check failed
    showExceptionDialog();
    return false;
  }
  // limit check passed successfully
  return true;
}
```

Chapter 12

Adding Exception Handling for Recurring Irregular Situations

➢ Does not apply to server-side phases running on the OE server.

This section contains general information about providing specific exception options to cover recurring irregular situations. The exception options lead to user-triggered exceptions. Each option represents a variant of the current phase. The applied option must be confirmed by an electronic signature and added as an exception. The options are available, in conjunction with each other, for processing in the Exception Window.

A phase developer can support exception options for recurring irregular situations in the phase executor for the operator's convenience, but he is not forced to provide them.

At the end of the section we will have a phase that provides three exception options to cover recurring irregular situations.

To add exception handling for recurring irregular situations to a phase, perform the following steps:

1. Define the user interface (page 143) for the exception options in your phase executor.
2. Add business logic to the exception option (page 146) in order to allow executing the specific business logic for exception option of the phase.
3. Add checks to the exception option (page 151) in order to log failed check results by extending the description of the recorded exception.
   This step is optional.

Step 1: Defining the User Interface

TIP

The *SupportExceptions* attribute in the XML description of the phase building block specifies whether your phase supports exceptions (see section "Step 1: Creating an XML Description of the Phase Building Block" (page 33) in chapter "Creating a Phase Building Block" (page 33)).

The user interface of the exception options for recurring irregular situations can be defined by overwriting the *AbstractPhaseExecutorSwing.createPhaseExceptionComponent()* method. By default, the method returns null.
If you want to support multiple exception options, you can nest them as panels.

Example

In our **RtPhaseExecutorInstructTxt** example, we define three exception options as three separate sub-panels: **DoubleCheck**, **SingleCheck**, and **NoCheck**. The sub-panels are added to the parent panel.

Example

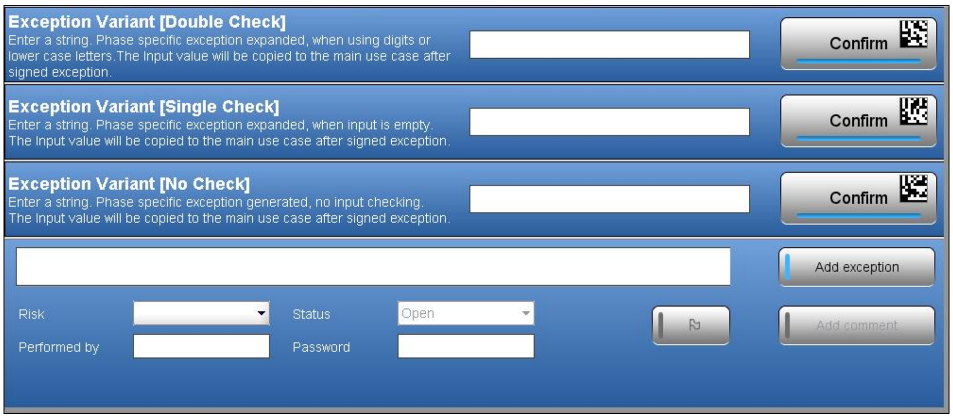
```
@Override
public JComponent createPhaseExceptionComponent() {
  threeExceptionsPanel = new JPanel(new GridBagLayout());
  GridBagConstraints constraints = new GridBagConstraints();
  // Fill horizontal because the exception panel is little bigger than the
  // normal phase column layout.
  constraints.fill = GridBagConstraints.HORIZONTAL;
  constraints.weightx = 1;
  // Leave some space between the panels
  final int vGap = 3;
  constraints.insets = new Insets(0, 0, vGap, 0);
  threeExceptionsPanel.add(createExceptionPanelDoubleCheck(), constraints);
  constraints.gridy = 1;
  threeExceptionsPanel.add(createExceptionPanelSingleCheck(), constraints);
  constraints.gridy = 2;
  threeExceptionsPanel.add(createExceptionPanelNoCheck(), constraints);
  return threeExceptionsPanel;
}
```

We use helper methods (e.g. *createExceptionPanelDoubleCheck()*) to generate the sub-panels. All of the sub-panels have to contain a **Confirm** button to call the *displayException()* method at the end (see "Step 2: Adding Business Logic to the Exception Option" (page 146)).

Example

```
@Override
private JPanel createExceptionPanelDoubleCheck() {
  final JPanel exceptionPanel =
        PhaseSwingHelper.createExceptionPanel(Layout.LAYOUT_TWO_1ST_WIDER_COLUMN);

  JLabel label = new JLabel(EXCEPTION_LABEL_TEXT_DC);
  label.setPreferredSize(EXCEPTION_LABEL_DIM);
  exceptionPanel.add(label, Column.FIRST_COLUMN);

  JPanel editPanel = PhaseSwingHelper.createPanel();
  exceptionTextFieldDC = PhaseSwingHelper.createJTextField(getStatus());
  editPanel.add(exceptionTextFieldDC);
  exceptionPanel.add(editPanel, Column.SECOND_COLUMN);

  /** confirm exception listener */
  class ConfirmExceptionListener implements ActionListener {
    @Override
    public void actionPerformed(ActionEvent event) {
      // Clear other variants
      exceptionTextFieldSC.setText("");
      exceptionTextFieldNC.setText("");
      doConfirmActionDC(exceptionTextFieldDC.getText());
    }
```

```
   }
   ConfirmButton confirmButton =
     ((PhaseColumnLayout) exceptionPanel.getLayout()).getConfirmButton();
   confirmButton.addActionListener(new ConfirmExceptionListener());
   styleSubPanel(exceptionPanel);
   return exceptionPanel;

 }
```


Exception Variant [Double Check]
Enter a string. Phase specific exception expanded, when using digits or lower case letters.The Input value will be copied to the main use case after signed exception.

Exception Variant [Single Check]
Enter a string. Phase specific exception expanded, when input is empty. The Input value will be copied to the main use case after signed exception.

Exception Variant [No Check]
Enter a string. Phase specific exception generated, no input checking. The Input value will be copied to the main use case after signed exception.

Risk
Status Open
Performed by
Password

Figure 30: Exception Window with three exception options

TIP

There are two different methods to define the background gradient color scheme of exception options:

- If you wish to define a separate background color gradient for sub-panels as in the example above, call
  `subPanel.setUI(new PhaseExceptionPanelUI(exceptionPanel));`
  `subPanel.setOpaque(true);`
  for each sub-panel.
  Now, each sub-panel uses the same gradient color scheme as defined for the exception frame panel (see pec_StyleSheet, 'PhaseExceptionPanel' section) and section "Adapting Style Sheets" in chapter "Changing the General Appearance of PharmaSuite" in Volume 1 of the "Technical Manual Configuration and Extension" [A1] (page 175). Additionally, it adjusts the background coloring according to *enable* status of the sub-panel.

- If you wish to define only one background color gradient for all exception options (the default setting), provide the information that your sub-panels are transparent. In this case, for each sub-panel call `subPanel.setOpaque(true);`. Also set the parent panel holding the sub-panels to transparent, then, the gradient color scheme of the exception frame panel is visible.

Step 2: Adding Business Logic to the Exception Option

When the **Confirm** button of the **exception option** has been executed, the phase must call the *AbstractPhaseExecutor::displayException()* method. The idea behind this is that the business logic, which is typically executed along with the **Confirm** button, will be delayed until the signature has successfully been accepted for the exception option. If the signature has not been provided or accepted, no business logic will be executed. The phase developer can define the unique check key for the exception, the risk category, and the exception description. The access privilege required for the exception signature is automatically retrieved from the privilege parameters assigned to the phase, according to the risk category (see also "What Is a Privilege Parameter?" (page 16)).

The *displayException* method fills the exception recording panel with the given exception information.

After the phase executor has requested an exception to be recorded, the framework disables the panel of the exception including its **Confirm** button. This prevents that the data of an exception that was confirmed but not signed may be lost after the **Confirm** button has been clicked a second time. Additionally, the gradient in the background will be replaced by a gray background.

When the operator has signed the exception, the framework does not enable the panel again. This is to avoid confusing situations for the operator and to prevent misuse. If the operator wishes to submit another exception, he has to leave the exception window and return to it.
To change this default behavior, you may enable the panel again in the *isExceptionSigned()* callback method.

AbstractPhaseExecutor::displayException() method

```
/**
  * display the exception form allowing to sign for an exception
  * @param checkKey the unique key to identify that the exception was
  *            really entered when the exception window was closed
  * @param riskClass the riskClass of the exception record
  * @param freeText The text to be stored in the exception
  *
  * @return true, if the exception form with the requested exception is displayed
  *         false, if the state of the framework doesn't allow this operation.
  *                This is the case e.g. when the Navigator or Cockpit is displayed
  *                or when the user already has leaved the exception view.

  */
 public void displayException(String checkKey, IMESExceptionRecord.RiskClass riskClass,
                              String freeText,)
```

Callback methods:

- There are also several callback methods which should be overwritten to implement phase-specific business logic for the exception options.

All of these methods are part of the *IExceptionRecordingCallbackConsumer* interface.

- *AbstractPhaseExecutor::exceptionSigned(String checkKey)*
  It will be called after an operator has successfully added an exception to a phase.

TIP
The *exceptionSigned(String checkKey)* method is intended for UI updates only as it runs after the transaction. If you need to modify data, use the *exceptionCallbackInSaveTransaction* callback.

- *AbstractPhaseExecutor::exceptionCanceled(String checkKey)*
- *AbstractPhaseExecutor::exceptionCallbackInSaveTransaction(String checkKey, IMESExceptionRecord exceptionRecord, IESignatureExecutor sigExecutor)*

exceptionSigned(String checkKey) method

```
/**
 * This method is called after a requested exception with 'checkKey' was successfully
 * signed.
 * This callback is intended for updating the UI accordingly.
 * Note: Do not modify data (phase data and process model) in this callback
 * because it is not inside the transaction used to record the exception.
 * For this purpose use the {@link #exceptionCallbackInSaveTransaction} callback.
 *
 * @param checkKey unique key to identify the exception
 */
void exceptionSigned(String checkKey);
```

exceptionCanceled(String checkKey) will be called after an operator has canceled adding an exception to a phase.

exceptionCanceled(String checkKey) method

```
/**
 * This method is called after a requested exception with 'checkKey' was not signed,
 * but canceled.
 * This callback is intended for updating the UI accordingly.
 * Please note, that model changes must not be reversed, since no transaction
 * to commit model changes was made by {@link #exceptionCallbackInSaveTransaction}.
 *
 * @param checkKey unique key to identify the exception
 */
void exceptionCanceled(String checkKey);
```

- A third callback may be overwritten, if there is a need to save modified data along with the exception and signature:

- *AbstractPhaseExecutor::exceptionCallbackInSaveTransaction(String checkKey, IMESExceptionRecord exceptionRecord, IESignatureExecutor sigExecutor)*

exceptionCallbackInSaveTransaction(...) is called after an exception record has been saved and the operator has signed for the exception, but before the signature is saved.

exceptionCallbackInSaveTransaction(...) method

```
/**
 * This method is called after the exception record has been saved, but before
 * the signature executor is saved and the transaction has ended.
 * It can be overwritten to make changes that need to be saved within the
 * transaction.
 * Note: Any code in the overwritten method is executed within the transaction.
 * Refrain from any actions that would break the transaction as user interaction,
 * etc.
 * @param checkKey unique key to identify the exception
 * @param exceptionRecord the already saved exception record
 * @param sigExecutor the created, but not yet saved signature executor
 */
public void exceptionCallbackInSaveTransaction(String checkKey,
    IMESExceptionRecord exceptionRecord, IESignatureExecutor sigExecutor) {
}
```

The "save the exception record", "execute the callback", and "save the signature" operations are executed by one transaction. Either all of the operations succeed and are saved or at least one of them fails, then none of them are saved.

At this point, the exception record has already been saved, but the signature executor has not yet saved the signature. Saving the signature within the callback is acceptable, but not necessary. This flexibility is needed since some service calls, which you might want to call in such a callback, require the signature executor and will save the signature within the service call (see e.g. *com.rockwell.mes.services.inventory.ifc.IMFCService.unidentifySublot Sublot sublot, IESignatureExecutor sigExecutor*).

If the signature is not saved within the callback, it will be saved afterwards by the "save the signature" operation.

TIP
Transactions are subject to timeouts. That is why you should refrain from any time-consuming or interruptible operations (e.g. prompting for user input) within the *exceptionCallbackInSaveTransaction(...)* callback.

TIP
Contrary to the other two callbacks the third callback is called whenever an exception is recorded for the current phase, not only if the phase executor has initiated the exception. If the exception has not been initiated by the phase executor, the *checkKey* parameter will be *null*. So it is good practice to verify

whether the *checkKey* parameter is not null and matches the *checkKey* you have passed to *displayException(String checkKey, ...)*.

Example

In our **RtPhaseExecutorInstructTxt** example, we update the contents of the text box of the current phase (in the Execution Window) only if the signature has successfully been accepted for the exception option.

Example

```
/**
 * update the field text in the main control
 *
 * @param checkKey which references the exception variant
 */
@Override
public void exceptionSigned(String checkKey) {
  String newValue = null;
  if (checkKey.equals(EXCEPTION_CHECKKEY_DC)) {
    newValue = exceptionTouchFieldDC.getInputField().getText();
    ...
  } else if (checkKey.equals(EXCEPTION_CHECKKEY_SC)) {
    newValue = exceptionTouchFieldSC.getInputField().getText();
    ...
  } else if (checkKey.equals(EXCEPTION_CHECKKEY_NC)) {
    newValue = exceptionTouchFieldNC.getInputField().getText();
    ...
  } else {
    throw new MESRuntimeException("undefined case for checkKey:" + checkKey);
  }
  text = newValue;
  touchFieldForText.getInputField().setText(newValue);
}
```

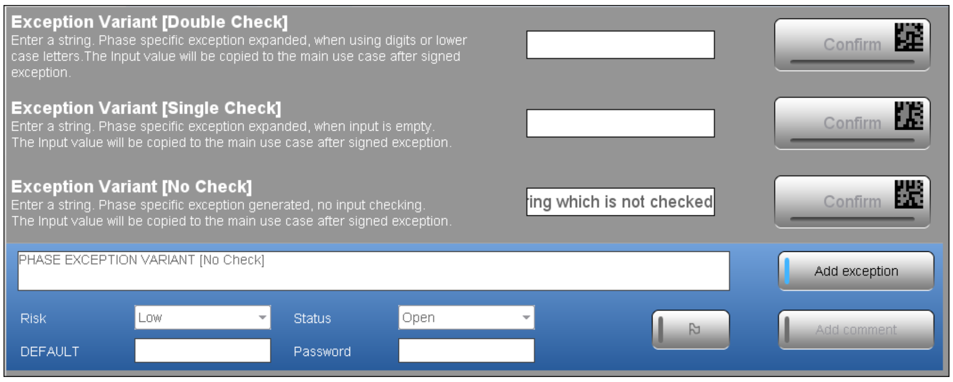
In this case, there is no need for special logic in case the operator has not provided a signature. However, it would also be possible to do some cleanup here.

```
@Override
public void exceptionCanceled(String checkKey) {
  LOGGER.debug("exceptionCanceled:" + checkKey);
}
```

The following example modifies the phase data in the *exceptionCallbackInSaveTransaction(...)* callback. The modified data is only saved, if the exception record and the signature have successfully been recorded and saved. If the callback fails and throws an exception, neither exception record nor signature will be saved.

Example

```
@Override
public void exceptionCallbackInSaveTransaction(String checkKey,
    IMESExceptionRecord exceptionRecord, IESignatureExecutor sigExecutor) {
    // callback can come any time, not only when initiated from phase executor
    // here, only react when phase is active and exception recording was
    // initiated by us (the phase executor), i.e. checkKey is not null
    IMESRtPhase rtPhase = getRtPhase();
    if (checkKey != null && textField != null && rtPhase != null) {
      String newValue = textField.getText()
        + "(SaveCallback at " + Calendar.getInstance().getTime() + ")";
      textField.setText(newValue);
      // runtime phase may be new and phase data not yet created.
      MESRtPhaseDataInstructTxt data = getRtPhaseData();
      if (data == null) {
        data = addRtPhaseData();
      }
      data.setActualValue(text);
      Response response = rtPhase.save(null, null,
        PCContext.getDefaultAccessPrivilege());
      if (response.isError()) {
        // will cause rollback!
        throw new MESRuntimeException("Could not save phase data");
      }
    }
}
```


Exception Variant [Double Check]
Enter a string. Phase specific exception expanded, when using digits or lower case letters. The Input value will be copied to the main use case after signed exception.

Exception Variant [Single Check]
Enter a string. Phase specific exception expanded, when input is empty. The Input value will be copied to the main use case after signed exception.

Exception Variant [No Check]
Enter a string. Phase specific exception generated, no input checking. The Input value will be copied to the main use case after signed exception.

ing which is not checked

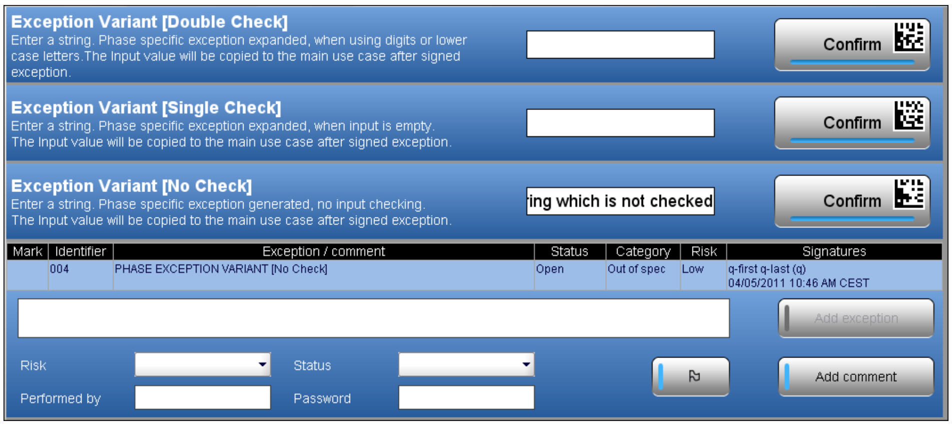
PHASE EXCEPTION VARIANT [No Check]

Risk: Low
Status: Open
DEFAULT
Password

Figure 31: Exception option [No Check] has been confirmed

If the operator does not add the exception option (**Add exception** button) and returns to the Execution Window (**Back** button) instead, the system displays a question dialog provided by the framework, which allows the operator either to return to the Execution Window or to stay in the Exception Window.

If the operator has successfully signed the exception option, the exception will be displayed in the exception table and the business logic will be executed.



Exception Variant [Double Check]
Enter a string. Phase specific exception expanded, when using digits or lower case letters.The Input value will be copied to the main use case after signed exception.

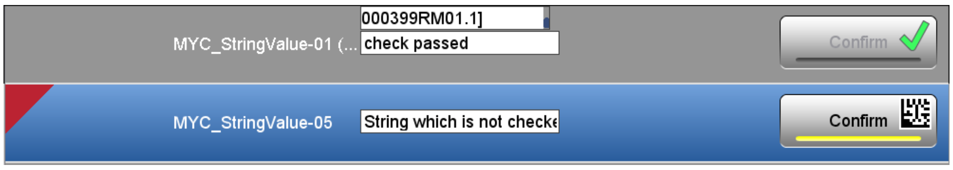
Exception Variant [Single Check]
Enter a string. Phase specific exception expanded, when input is empty.
The Input value will be copied to the main use case after signed exception.

Exception Variant [No Check]
Enter a string. Phase specific exception generated, no input checking.
The Input value will be copied to the main use case after signed exception.

ing which is not checked

Mark	Identifier	Exception / comment	Status	Category	Risk	Signatures
	004	PHASE EXCEPTION VARIANT [No Check]	Open	Out of spec	Low	q-first q-last (q) 04/05/2011 10:46 AM CEST

Figure 32: Exception option [No Check] has been confirmed and added



MYC_StringValue-01 (... 000399RM01.1] check passed

MYC_StringValue-05 String which is not checke

Figure 33: Result of exception option [No Check] in Execution Window

Step 3: Adding Checks to Exception Options

This step is optional.
Depending on the exception option, the system can also perform additional checks (e.g. status checks or rule checks) **before** the description of the exception option will be displayed. The *PhaseWarningDialog.showDialog()* method can be used to display an exception-related string of a failed check. The description of the corresponding exception option can be extended by the exception-related string of a failed check. That means, when the exception is signed and recorded later, the results of the additional checks will automatically be recorded, since they are part of the description of the exception.

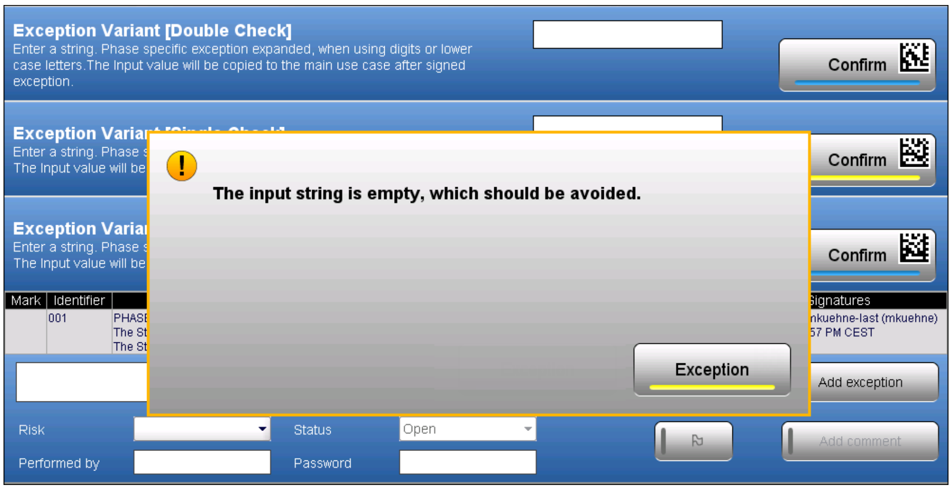
Example

In our **RtPhaseExecutorInstructTxt** example, we check whether no input was provided and we display a violation using *PhaseWarningDialog.showDialog()*. The exception-related string of the failed check is appended to the description of the exception option. Both, the original exception description and the extension will be recorded. If an operator cancels the check-related dialog box, the confirmation of the exception option is canceled.

Example

```
private boolean doConfirmActionSC(final String inputString) {
  StringBuffer freeText = new StringBuffer("PHASE EXCEPTION VARIANT [Single Check]");
  int result = IExceptionHandler.OK_OPTION;
  if (StringUtils.isEmpty(inputString)) {
    PhaseWarningDialog dialog = new PhaseWarningDialog();
    String dialogText = "The input string is empty, which should be avoided.";
```

```
      result = dialog.showDialog(dialogText);
      if (result != IExceptionHandler.OK_OPTION) {
        return false;
      }
      freeText.append("\n").append(dialogText);
    }
    if (result == IExceptionHandler.OK_OPTION) {
      displayException(EXCEPTION_CHECKKEY_SC, IMESExceptionRecord.RiskClass.medium,
                       freeText.toString());
      return true;
    }
    return false;
  }
```


Exception Variant [Double Check]
Enter a string. Phase specific exception expanded, when using digits or lower case letters.The Input value will be copied to the main use case after signed exception.

Exception Variant [Single Check]
Enter a string. Phase s
The Input value will be

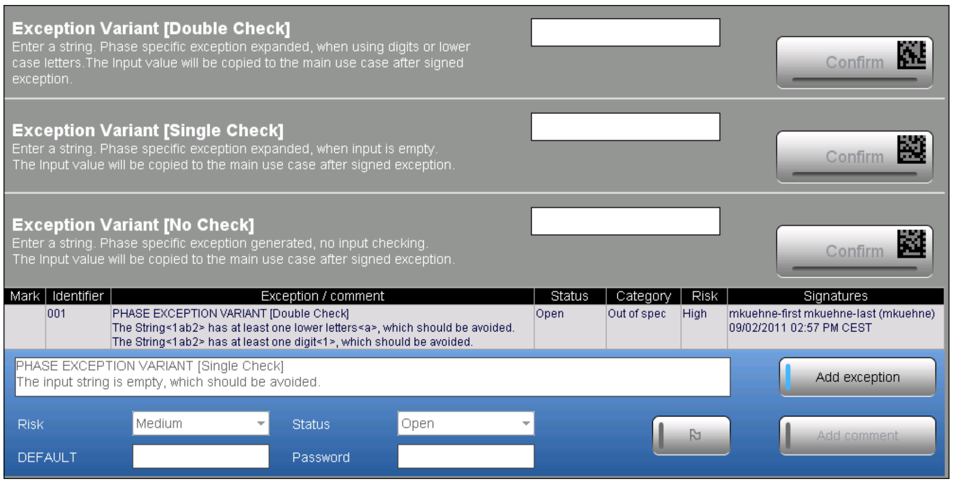
Exception Varia
Enter a string. Phase s
The Input value will be

The input string is empty, which should be avoided.

Mark	Identifier		Signatures
	001	PHASE The St The St	nkuehne-last (mkuehne) 57 PM CEST

Risk
Status: Open
Performed by
Password

Figure 34: Exception option [Single Check] with "check" dialog box


Exception Variant [Double Check]
Enter a string. Phase specific exception expanded, when using digits or lower case letters.The Input value will be copied to the main use case after signed exception.

Exception Variant [Single Check]
Enter a string. Phase specific exception expanded, when input is empty.
The Input value will be copied to the main use case after signed exception.

Exception Variant [No Check]
Enter a string. Phase specific exception generated, no input checking.
The Input value will be copied to the main use case after signed exception.

Mark	Identifier	Exception / comment	Status	Category	Risk	Signatures
	001	PHASE EXCEPTION VARIANT [Double Check] The String<1ab2> has at least one lower letters<a>, which should be avoided. The String<1ab2> has at least one digit<1>, which should be avoided.	Open	Out of spec	High	mkuehne-first mkuehne-last (mkuehne) 09/02/2011 02:57 PM CEST

PHASE EXCEPTION VARIANT [Single Check]
The input string is empty, which should be avoided.

Risk: Medium
Status: Open
DEFAULT
Password

Figure 35: Exception option [Single Check] with extended exception description

Chapter 13

Adding Actions for Completed Phases in the Navigator

➢ Does not apply to server-side phases running on the OE server.

This section contains general information about actions that can be performed on completed phases via the Navigator. The purpose of the actions is to cover irregular situations that have to be handled after a phase has been completed. Examples of such actions are the reprint of a label or the correction of a recorded value.

You can add up to three post-completion actions to your phase. Each action will be represented by a button in the action column of the Navigator.

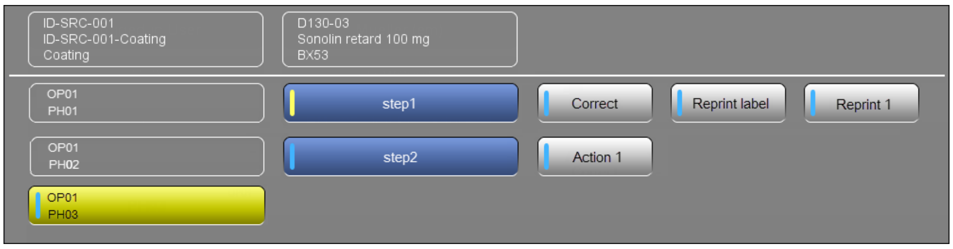
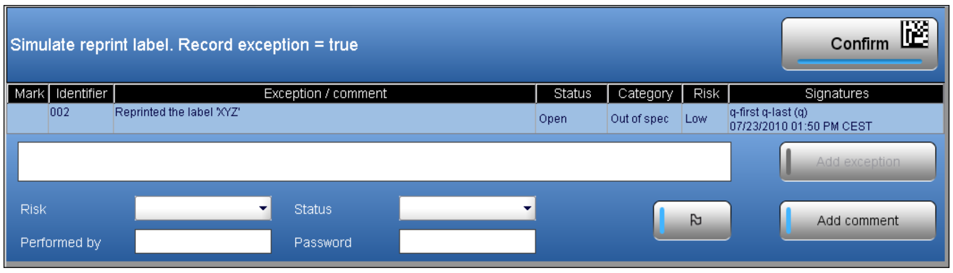


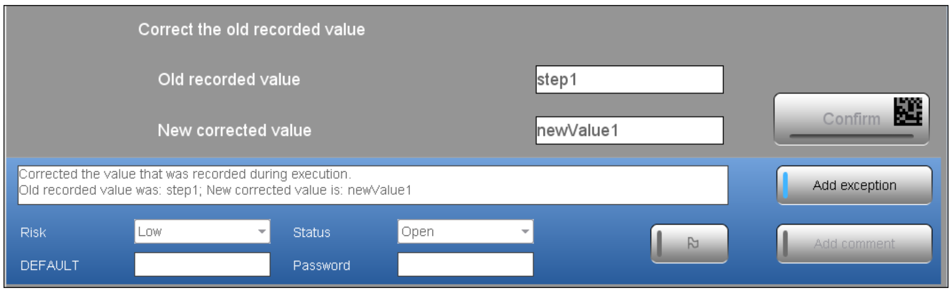
Figure 36: Example: Actions for completed phases in the Navigator

When a user clicks an action button, the system displays the Exception Window with the action-specific GUI. Depending on the phase implementation, the user can either record an exception and execute business functionality, or only record an exception, or only execute business functionality.


Simulate reprint label. Record exception = true

Mark	Identifier	Exception / comment	Status	Category	Risk	Signatures
	002	Reprinted the label 'XYZ'	Open	Out of spec	Low	q-first q-last (q) 07/23/2010 01:50 PM CEST

Figure 37: Example: Simple action-specific GUI



Corrected the value that was recorded during execution.
Old recorded value was: step1; New corrected value is: newValue1

Figure 38: Example: Action-specific GUI with input box

A phase developer may support Navigator actions in the phase executor for the operator's convenience, but he is not forced to provide them.

To add Navigator actions to a phase, perform the following steps:

1. Define the text to be displayed on the action buttons (page 154).
2. Define the user interfaces of the actions (page 155).
3. Add the business logic to each action (page 156).

Step 1: Defining the Names of the Action Buttons

To define the names of the action buttons, overwrite the *AbstractPhaseExecutor.getActionButtonTexts()* method. By default, the method returns null.

TIP

If you need more than three actions, you have to use the third action button as entry point for a GUI hosting multiple actions. For example, name the button **More...**.

Example

In our example, we define three navigator actions: **Correct**, **Reprint label**, and **Reprint 1**. **Reprint 1** a label reprints without recording an exception.

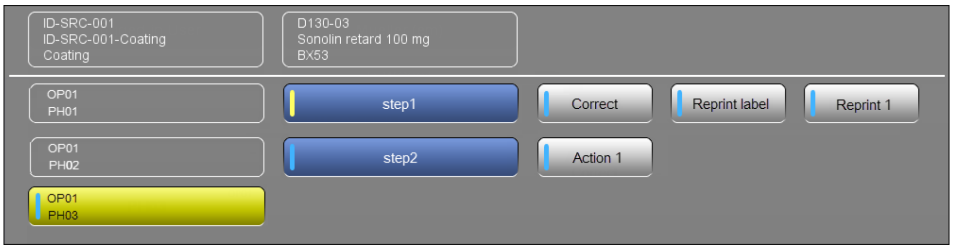


Figure 39: Example: Actions for completed phases in the Navigator

Example: getActionButtonTexts()

```
@Override
public List<String> getActionButtonTexts() {
  List<String> buttonTexts = new Vector(MAXIMAL_NUMBER_ACTIONS);
  buttonTexts.add("Correct");
  buttonTexts.add("Reprint label");
  buttonTexts.add("Reprint 1");
  return buttonTexts;
}
```

TIP

Do not forget to use internationalization; this is not covered in the example.

Step 2: Defining the User Interfaces of the Actions

To define the action-specific UI, you have to overwrite the *AbstractPhaseExecutorSwing.createPhaseActionComponent(int actionIndex)* method, which returns a *JComponent*. By default, the method returns null.

TIP

The action index is the index of the action button text in the list defined in Step 1 (page 154). Therefore the first action has the index 0.

We recommend to create individual methods per action GUI rather than implementing all of the GUIs in this method.

Example

Example: Action buttons

```
@Override
public JComponent createPhaseActionComponent(int actionIndex) {
  switch (actionIndex) {
  case 0:
    return createActionPanelCorrectValue();
  case 1:
    return createActionPanelReprint(true);
  case 2:
    return createActionPanelReprint(false);
  default:
    throw new MESRuntimeException("Unsupported actionIndex " + actionIndex);
  }
}
```

In the individual method for the action GUI, you have to create all of the necessary controls, e.g. input boxes (if required), and the **Confirm** button to trigger the execution of the action’s business logic.

Example: Exception view

```
private JPanel createActionPanelReprint(final boolean recordException){
  final JPanel exceptionPanel =
               PhaseSwingHelper.createExceptionPanel(Layout.LAYOUT_SINGLE_COLUMN);

  JPanel column1 = PhaseSwingHelper.createPanel();
  exceptionPanel.add(column1, Column.FIRST_COLUMN);
  JLabel label = new JLabel("Simulate reprint label. Record exception = " + recordException);
  column1.setLayout(new BorderLayout());
    column1.add(label, BorderLayout.NORTH);
    /** confirm exception listener */
    class ConfirmReprintActionListener implements ActionListener {
      @Override
      public void actionPerformed(ActionEvent event) {
        if (recordException) {
          displayException(ACTION_CHECKKEY_REPRINT_LABEL,
                           IMESExceptionRecord.RiskClass.low,
                           "Reprinted the label 'XYZ'");
        } else {
          doReprintLabel();
        }
      }
    }
    ConfirmButton confirmButton = ((PhaseColumnLayout)
                                  exceptionPanel.getLayout()).getConfirmButton();
    confirmButton.addActionListener(new ConfirmReprintActionListener());
    return exceptionPanel;
```

Step 3: Adding the Business Logic to the Actions

The business logic is called when the **Confirm** button is executed. We recommend to encapsulate the business logic in an individual method.
Thus, you need one method to trigger the exception and one method to perform the business logic when the signature has successfully been accepted for the exception.

TIP

You may record an exception and then execute your business logic. You can also just record an exception or call your business logic without recording an exception.

After the phase executor has requested an exception to be recorded, the framework disables the panel of the action including its **Confirm** button. This prevents that the data of an exception that was confirmed but not signed may be lost after the **Confirm** button has been clicked a second time. Additionally, the gradient in the background will be replaced by a gray background.

When the operator has signed the exception, the framework does not enable the panel again. This is to avoid confusing situations for the operator and to prevent misuse. If the operator wishes to submit another exception, he has to leave the exception window and return to it.
To change this default behavior, you may enable the panel again in the *isExceptionSigned()* callback method.

Example

Example: triggerReprintLabel

```
private void triggerReprintLabel(final boolean recordException) {
  if (recordException) {
    String freeText = "Reprinted the label 'XYZ'";
    displayException(EXCEPTION_CHECKKEY_REPRINT_LABEL,
                     IMESExceptionRecord.RiskClass.low, freeText);
  } else {
    doReprintLabel();
  }
}
...
@Override
public void exceptionSigned(String checkKey) {
  if (EXCEPTION_CHECKKEY_REPRINT_LABEL.equals(checkKey)) {
    doReprintLabel();
  }
}
```

Example (Changing Runtime Phase Data)

You can change the runtime phase data and/or runtime phase output data when the post-completion action is executed. One way to achieve this is to use a **Correct value** action.

After changing the data, the runtime phase must be saved.

Example: Changing runtime phase data

```
private Response correctValue() {
  // Set runtime phase data: the field text should be retrieved from the Action UI
  MESRtPhaseDataPAVStr data = getRtPhaseData();
  data.setActualValue(text);

  // Save the phase
  Response res = getRtPhase().save(null, null, PCContext.getDefaultAccessPrivilege());
}
```

Adding Phase-specific Sub-reports

From a technical point of view, a batch report in PharmaSuite consists of one main report and several sub-reports. Sub-reports divide a report into several components. Each sub-report has its individual report design. Thus, each sub-report can be designed and filled separately.

Furthermore there is another partition of the batch report: a static part and a dynamic part. The static part comprises the main report(s) with all sub-reports, which represents the reports and sub-reports of the batch report framework. The dynamic part holds the phase-specific sub-reports. They are not known when the generation of the batch report starts, but they are created on the fly, based on the phases that are executed in the Production Execution Client.

The batch report is designed to receive the batch production record (short: batch record) as a JavaBean-based data source. The data source of the main report is an object of the *IBatchProductionRecordDocumentWrapper* type. It wraps a *BatchProductionRecordDocument*, - a B2MML-based object containing the batch record data, - providing additional methods to improve the performance (B2MML = Business To Manufacturing Markup Language). Each sub-report has its own data source that is either the same as for the main report or a B2MML-based object contained in the *BatchProductionRecordDocument*. The data displayed in the batch report is retrieved from the data sources either with expressions based on the JavaBean properties or by directly using the *IBatchProductionRecordDocumentWrapper* helper methods.

For more details about reports in PharmaSuite and how to add phase-specific sub-reports to a batch report, see chapter "Changing or Adding New Reports" in Volume 2 of the "Technical Manual Configuration and Extension" [A2] (page 175).

Additionally to the batch report, the phase-specific sub-report is accessible via the Navigator's information column.

Chapter 15

Adding Pause-aware Support

A phase that shall react on pause and continue events of a pause-enabled unit procedure requires to support **pause-aware**. Typical use cases are: interrupt phase execution, prevent phase completion, or only visualize the **Pause** status in the UI of the Production Execution Client.

The pause-aware support defines two callbacks in addition to the normal life cycle as described in "Callback Methods Related to the Life Cycle of a Phase Building Block" (page 81).

Additionally, the phase can query the **Pause** status while it is active.

To enable and use the pause-aware support, perform the following steps:

1. Set the pause-aware flag of a phase (page 161).
2. Use the pause callback API during phase execution (page 161).

Step 1: Setting the Pause-aware Flag of a Phase

The Pause-aware flag of a phase is defined in the XML description of the phase building block. For details, see section "Step 1: Creating an XML Description of the Phase Building Block" (page 33).

Example

To make a phase pause-aware, add the following line to the XML description listed in "Step 1: Creating an XML Description of the Phase Building Block" (page 33):

```
<IsPauseAwarePhase value="true"/>
```

Then, re-create your phase with the phase generator.

Step 2: Using the Pause Callback API during Phase Execution

In our example, we only want to visualize the current **Pause** status in the UI.
For this purpose, we use the *pausePhaseExecution* and *continuePhaseExecutionAfterPause* callbacks to modify the text displayed with a *JLabel* component.

However, if the unit procedure is already paused when the phase is started, the *pausePhaseExecution* callback will not occur.
Therefore we also have to check on *startPhaseExecution* if the pause is already active.

Example

Example: Using the pause callback API

```
@Override
public void pausePhaseExecution() {
  setUpPausedLabel(true);
}

@Override
public void continuePhaseExecutionAfterPause() {
  setUpPausedLabel(false);
}

@Override
public void startPhaseExecution() {
  // Evaluate the pause state initially because it might already be set before
  setUpPausedLabel(isPaused());
}

private void setUpPausedLabel(final boolean pausedState) {
  unitProcedurePausedLabel.setText("UP paused=" + pausedState);
}
```

Chapter 16

Responding to the Change User and Register at Station Actions

➢ Does not apply to server-side phases running on the OE server.

This section describes how a phase can respond to the **Change User** and **Register at Station** actions provided in the Cockpit of the Production Execution Client.

- A change of the logged-in user does not interrupt started operations and workflows. They continue to run with the current user until another user with sufficient access rights successfully takes over, and after this take-over, they continue to run with the new user.
- A station change holds the operations and workflows running on the current station, since different operations and workflows are available on the new station.

We recommend to use a context function to retrieve the logged-in user or current station if the phase does not require to cache the data for functional reasons:

```
PCContext.getFunctions().getCurrentUser()
PCContext.getFunctions().getStation()
```

If the rendering of your phase or its status depends on the logged-in user or the current station, register a listener at the *IChangeUserOrStationService* service:

```
IUserOrStationChangeListener cl = new MyUserChangeListener();
// Register to change events
ServiceFactory.getService(IChangeUserOrStationService.class).addChangeListener(cl);
// Do not forget to remove the listener to avoid memory leaks
ServiceFactory.getService(IChangeUserOrStationService.class).removeChangeListener(cl);
```

For convenience, PharmaSuite provides an adapter that allows to implement only one of the two change-related events:

```
private class MyUserChangeListener extends
        IChangeUserOrStationService.UserOrStationChangeAdapter {
  @Override
  public void userChanged(String oldUserName, String newUserName) {
    // implement functionality
    System.out.println("User changed. Old=" + oldUserName + " new=" + newUserName);
  }
}
```

Chapter 17

Extending Product Phases and Parameter Classes

This section describes the general extension mechanism for PharmaSuite product phases and parameter classes. The task is to adapt the provided source code of the product phases and their related parameter classes.

Please ensure that you observe the guidelines listed below.

From the point of view of a PharmaSuite system, an extended phase is a new phase. PharmaSuite supports the installation and usage of the original phase and the new (=extended) phase in parallel in the same system. The installation of the original product phase must not interfere with the installation of the new (=extended) phase and vice versa.

The same also applies for new versions of already extended phases or new versions of customer-created phases. Please refer to the chapters "Create a Phase Building Block" (page 33) and "Creating and Using Process Parameters" (page 99).

Consider to track all changes applied to the new phase to avoid any migration issues. Migration or update issues may occur in two cases:

1. PharmaSuite provides product phases in a new version
   The artifacts of the new version of the product phase are renamed according to the naming conventions (page 5). Therefore you should consider to create a new version of the extended phases. For this purpose, proceed as described above.
2. PharmaSuite provides a Maintenance Release for product phases
   The artifacts of the updated product phases are **not** renamed. Therefore it may not be necessary to create a new version of the extended phases.

Chapter 18

Mapping between Parameter Bean Attributes and AT Definition Columns

This section describes how to create and use parameter attributes that are not directly supported by a column of the parameter class.

Assume you have created a parameter attribute of the **Long** type, whose values are provided by a choice list. Within the Data Dictionary of the parameter class, you can associate a choice list with the attribute. The Parameter Panel of Recipe and Workflow Designer displays a choice list editor for the attribute.
If you intend to feed the attribute value via information flow based on an input expression, the following options are available:

- Change the data type of the attribute from **Long** to **String**.
  Starting with PharmaSuite 8.1, it is possible to assign the choice list editor not only to the **Long** type, but also to the **String** type. In the latter case, the choice list editor will edit the choice element meaning instead of the **Long** value.
- If changing the data type is no option, create a new attribute of the **String** type in your parameter class, which has a 1:1 relationship with the **Long** attribute.

The subsequent code snippet shows a concrete example. The **Long** attribute *method* represents a choice list element encoding a weighing method. Within Recipe and Workflow Designer, however, a **String** attribute *methodMeaning* shall be edited instead. This can be supplied by the choice list meaning as a **String** value via input expressions. Within the Data Dictionary, make the attribute *method* invisible and *methodMeaning* visible.
In order to store the data within the AT Row (containing only the **Long** column *method*), it is necessary to translate and transfer *methodMeaning* to *method*. This is done by the *getMethodMeaning* and *setMethodMeaning* methods. We recommend to also override the *setMethod()* method in order to receive correct *PropertyChangeEvents* if either the **Long** or the **String** values have changed. In particular, the repair recovery measure of the Production Execution Client (page 50) relies on the setter.

Code example

```
public class MESParamGWeighMethod0100 extends
       MESGeneratedParamGWeighMethod0100 implements IMESProcessParameterData {
  /**
   * @return the method as meaning
   */
  public String getMethodMeaning() {
    final Long weighingMethod = getMethod();
    if (weighingMethod == null) {
      return null;
    }
    return MESChoiceListHelper.getChoiceElement(
           GetWeightWeighingMethodChoicelistEnum0100.CLNAME,
           weighingMethod).getMeaning();
  }

  /**
   * @param meaning the meaning to set
   */
  public void setMethodMeaning(final String meaning) {
    if (StringUtils.isEmpty(meaning)) {
      setMethod(null);
    } else {
      setMethod(MESChoiceListHelper.getChoiceElement(
               GetWeightWeighingMethodChoicelistEnum0100.CLNAME, meaning).getValue());
    }
  }
  @Override
  public void setMethod(Long value) {
    final String oldMeaning = getMethodMeaning();
    super.setMethod(value);
    final String newMeaning = getMethodMeaning();
    pcs.firePropertyChange("methodMeaning", oldMeaning, newMeaning);
  }
```

In addition, you have to implement and register an *ATColumnNameMapper* as shown in the code snippet below. Otherwise the **PharmaSuite Export/Import Utility tool** for master recipes, master workflows, and building blocks will not be able to map a process parameter input expression for *methodMeaning*. That means input expressions will be lost during the export.

Code example

```
private static final IATColumnNameMapper GWEIGHT_METHOD_MAPPER =
        new DefaultATColumnNameMapper() {

  // Overridden to map hidden property "method" to visible property "methodMeaning"
  @Override
  public String getAttributeNameForAtColumnName(String atColumnName, String prefix) {
    final String attributeName =
          super.getAttributeNameForAtColumnName(atColumnName, prefix);
    if ("method".equals(attributeName)) {
      return "methodMeaning";
    }
    return attributeName;
  }
}
// Register the mapper
static {
  MESAtColumnNameManager.registerColumnMapper(MESParamGWeighMethod0100.class,
                         GWEIGHT_METHOD_MAPPER);
}
```

Chapter 19

Creating a New Version of a Phase and their Related Parameter Classes

This section describes how to create a new version of a phase and their related parameter classes.

PharmaSuite supports the installation and usage of different versions of a phase in the same system. In order to create a new version the artifacts of the original phase must be copied and renamed according to the naming conventions (page 5).

Depending on whether you wish to change the version of the parameters of a phase, you have the following options:

- Changing only the phase version
  If you wish to keep the version of the related process parameters, make use of the phase copy tool (page 25) to easily change the version of the phase.
- Changing the phase version and extending the phase runtime data or phase output data
  If you wish to keep the version of the related process parameters and to extend the phase runtime or output data, first copy the phase by means of the phase copy tool (page 25). Then, extend the XML description of the runtime phase or output data and run the phase generator with the changed XML files (see "Creating a Phase Building Block" (page 33)).
- Changing the phase version and the parameters versions
  (This cannot be done by means of the phase copy tool.)
  If you wish to create a new version of the phase and of the process parameters as well or, if you wish to change the phase runtime data or output data, perform the following steps for the phase and their related parameter classes:

  1. Define your vendor code and the version number to be used.
  2. Copy the whole original phase folder structure.
  3. Rename the folders according to the naming conventions, e.g. reflecting new Java package names.
  4. Update the *PhaseLibDescription* node in the XML schema of a phase (see chapter "Creating a Phase Building Block" (page 33)) and the *ParameterClassDescription* node in the XML schema of a parameter class (see chapter "Creating and Using Process Parameters" (page 99)), respectively.

5. Run the phase generator and the parameter class generator, respectively.
6. Adjust the DSX data in the XML files.
7. Refactor the Java sources to match the new version and adjust the constant values.

Finally, you have a new version of the phase working in the same way as the original phase. Now, any functional changes may be applied to the new phase.

Chapter 20

Adapting an Existing Phase Building Block

This section describes how to adapt the behavior of an existing phase building block.

TIP

If you have created your phase building block by means of the phase generator (see section "Creating a Phase Building Block" (page 33)) and there is a chance that you must re-create your phase building block, you should adapt your phase building block by means of adjusting the corresponding XML descriptions (see sections "Step 1: Creating an XML Description of the Phase Building Block" (page 33), "Step 2: Creating an XML Description of the Runtime Phase Data" (page 41), and "Step 3: Creating an XML Description of the Runtime Phase Output" (page 45)).

Adapting your phase building block via the phase manager is not recommended in this case, so the usage of the phase manager in the remainder of this section assumes that there is no need to re-create your phase building block.

At the end of the section our **MYC_StringValue** example phase will display a customer-specific string in the Navigator's information column (see section "What Is a Phase?" (page 12)).

To change the configuration of a phase, perform the following steps:

1. Create a subroutine (page 171) that overwrites the default implementation of the Navigator's information column.
2. Configure the phase with the phase manager (page 172).

Step 1: Creating a Subroutine for Filling the Navigator's Information Column

The runtime phase executor's default implementation for the string to be displayed in the Navigator's information column can also be overwritten using Pnuts code in a subroutine. The function will get the runtime phase as parameter.

TIP

Please note that when you define multiple methods in your subroutine, always the last method in the subroutine is called.

Example

To create a subroutine that delivers a string which will be displayed in the Navigator's information column, proceed as follows:

1. In Process Designer, right-click the **Subroutines** node and select the **New Subroutine** function.
2. Type a name (here: **infoColumn_MYC_StringValue**) for the new subroutine.
3. Use the following Pnuts code for the subroutine.

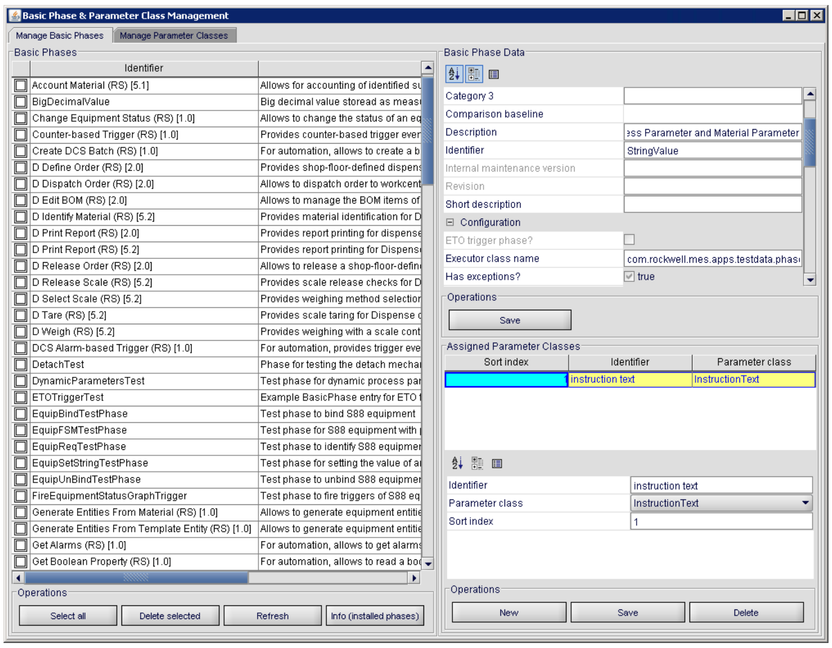
Subroutine

```
/*
 * Subroutine holding customization hooks (info column) for PhaseLib of type MYC_StringValue
 */

/*
 * get the column info for the runtime phase of this kind
 * PARAMETER
 *   rtPhase
 *     reference to the corresponding runtime phase
 * RETURN
 *   column info for the corresponding Lib
 */
function getInfoColumnStr(rtPhase) {
  ret = "SUBROUTINE: " + rtPhase.getPhase().getName()
  return ret
}
```

4. Save and close the new subroutine.

Step 2: Working with the Phase Manager

The phase manager provides several functions that support the creation, editing, and deletion of phase building blocks.



	Identifier	
☐	Account Material (RS) [5.1]	Allows for accounting of identified su
☐	BigDecimalValue	Big decimal value storead as measu
☐	Change Equipment Status (RS) [1.0]	Allows to change the status of an eq
☐	Counter-based Trigger (RS) [1.0]	Provides counter-based trigger ever
☐	Create DCS Batch (RS) [1.0]	For automation, allows to create a b
☐	D Define Order (RS) [2.0]	Provides shop-floor-defined dispens
☐	D Dispatch Order (RS) [2.0]	Allows to dispatch order to workcent
☐	D Edit BOM (RS) [2.0]	Allows to manage the BOM items of
☐	D Identify Material (RS) [5.2]	Provides material identification for D
☐	D Print Report (RS) [2.0]	Provides report printing for dispense
☐	D Print Report (RS) [5.2]	Provides report printing for Dispens
☐	D Release Order (RS) [2.0]	Allows to release a shop-floor-defin
☐	D Release Scale (RS) [5.2]	Provides scale release checks for D
☐	D Select Scale (RS) [5.2]	Provides weighing method selectio
☐	D Tare (RS) [5.2]	Provides scale taring for Dispense c
☐	D Weigh (RS) [5.2]	Provides weighing with a scale cont
☐	DCS Alarm-based Trigger (RS) [1.0]	For automation, provides trigger eve
☐	DetachTest	Phase for testing the detach mecha
☐	DynamicParametersTest	Test phase for dynamic process par
☐	ETOTriggerTest	Example BasicPhase entry for ETO t
☐	EquipBindTestPhase	Test phase to bind S88 equipment
☐	EquipFSMTestPhase	Test phase for S88 equipment with
☐	EquipReqTestPhase	Test phase to identify S88 equipmer
☐	EquipSetStringTestPhase	Test phase for setting the value of a
☐	EquipUnBindTestPhase	Test phase to unbind S88 equipmer
☐	FireEquipmentStatusGraphTrigger	Test phase to fire triggers of S88 eq
☐	Generate Entities From Material (RS) [1.0]	Allows to generate equipment entitie
☐	Generate Entities From Template Entity (RS) [1.0]	Allows to generate equipment entitie
☐	Get Alarms (RS) [1.0]	For automation, allows to get alarms
☐	Get Boolean Property (RS) [1.0]	For automation, allows to read a boo

Category 3	
Comparison baseline	
Description	ess Parameter and Material Parameter
Identifier	StringValue
Internal maintenance version	
Revision	
Short description	
⊟ Configuration	
ETO trigger phase?	☐
Executor class name	com.rockwell.mes.apps.testdata.phas
Has exceptions?	✓ true

Sort index	Identifier	Parameter class
1	instruction text	InstructionText

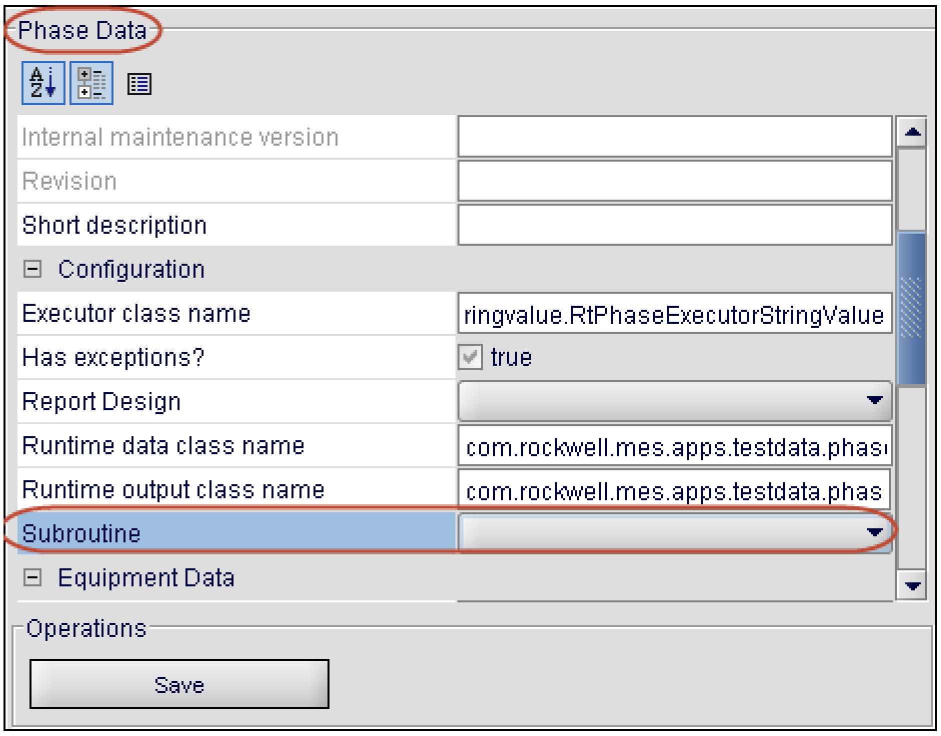
Identifier	instruction text
Parameter class	InstructionText
Sort index	1

Figure 40: Phase manager

Example

To configure a phase, proceed as follows:

1. In Process Designer, run the **mes_PhaseLibManager** form to start the phase manager.
2. Navigate to the **Manage Basic Phases** tab.
   In the **Basic Phases** panel, select the **MYC_StringValue** phase.
3. From the **Subroutine** drop-down list, select your subroutine (here: **infoColumn_MYC_StringValue**).


Phase Data

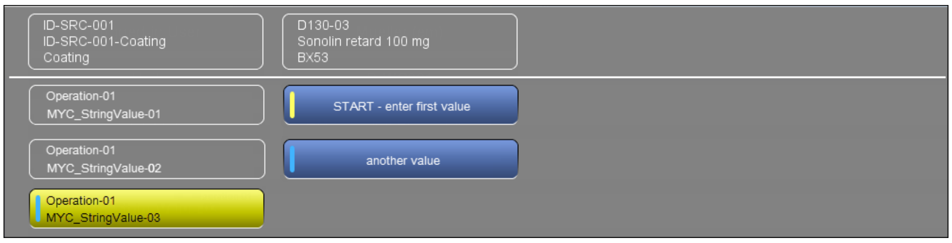
Internal maintenance version	
Revision	
Short description	
⊟ Configuration	
Executor class name	ringvalue.RtPhaseExecutorStringValue
Has exceptions?	✓ true
Report Design	
Runtime data class name	com.rockwell.mes.apps.testdata.phas
Runtime output class name	com.rockwell.mes.apps.testdata.phas
Subroutine	
⊟ Equipment Data	

Operations

Figure 41: Subroutine drop-down list of Phase Manager

4. Click the **Save** button to save the adapted phase in the database.
5. Close the phase manager, then close the form.

Now, the adapted phase can be used for execution.


ID-SRC-001
ID-SRC-001-Coating
Coating

D130-03
Sonolin retard 100 mg
Bx53

Operation-01
MYC_StringValue-01

START - enter first value

Operation-01
MYC_StringValue-02

another value

Operation-01
MYC_StringValue-03

Figure 42: MYC_StringValue phase with subroutine in the Navigator

TIP

If you have created your **MYC_StringValue** phase by means of the phase generator, then you would have achieved the same result by adding a corresponding **SubroutineName** node to the XML description of your **MYC_StringValue** phase (see section "Step 1: Creating an XML Description of the Phase Building Block" (page 33)).

Chapter 21

Reference Documents

The following documents are available from the Rockwell Automation Download Site.

No.	Document Title	Part Number
A1	PharmaSuite Technical Manual Configuration & Extension - Volume 1	PSCEV1-GR008E-EN-E
A2	PharmaSuite Technical Manual Configuration & Extension - Volume 2	PSCEV2-GR008E-EN-E
A3	PharmaSuite Technical Manual Configuration & Extension - Volume 4	PSCEV4-GR008E-EN-E
A4	PharmaSuite Technical Manual Administration	PSAD-RM008E-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

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.

Appendix A

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 5):

Object	Description	Document
---	---	---

Changes related to "What Is a Building Block?" (page 9):

Object	Description	Document
---	---	---

Changes related to "Tools for Creating Phase Building Blocks" (page 19):

Object	Description	Document
Performance during Execution of Phase Building Blocks (page 23)	Section moved from release notes.	1.0

Changes related to "Creating a Phase Building Block" (page 33):

Object	Description	Document
Step 5: Creating a Phase Executor (Java Code) Consider Abort and Repair (page 50)	New section.	1.0

Changes related to "Creating UI Extensions for Phase Building Blocks" (page 91):

Object	Description	Document
---	---	---

Changes related to "Creating and Using Process Parameters" (page 99):

Object	Description	Document
---	---	---

Changes related to "Using Material Parameters" (page 113):

Object	Description	Document
---	---	---

Changes related to "Using Equipment Requirement Parameters" (page 117):

Object	Description	Document
---	---	---

Changes related to "Handling S88 Equipment-related Features" (page 119):

Object	Description	Document
---	---	---

Changes related to "Adding Exception Handling for Phase-internal Checks" (page 135):

Object	Description	Document
Step 3: Displaying an Exception Message (page 138)	The system forces an operator to add an exception. Tip related to the *PhaseSystemTriggeredExceptionHandler* (was: *PhaseExceptionDialog*) updated.	1.0

Object	Description	Document
Example (page 139)	Code example updated.	1.0

Changes related to "Adding Exception Handling for Recurring Irregular Situations" (page 143):

Object	Description	Document
---	---	---

Changes related to "Adding Actions for Completed Phases" (page 153):

Object	Description	Document
---	---	---

Changes related to "Adding Phase-specific Sub-reports" (page 159):

Object	Description	Document
---	---	---

Changes related to "Adding Pause-aware Support" (page 161):

Object	Description	Document
---	---	---

Changes related to "Responding to the Change User and Register at Station Actions" (page 163):

Object	Description	Document
---	---	---

Changes related to "Extending Product Phases and Parameter Classes" (page 165):

Object	Description	Document
---	---	---

Changes related to "Mapping between Parameter Bean Attributes and AT Definition Columns" (page 167):

Object	Description	Document
Mapping between Parameter Bean Attributes and AT Definition Columns (page 167)	Setter to support *PropertyChangeEvents* added.	1.0

Changes related to "Creating a New Version of a Phase and their Related Parameter Classes" (page 169):

Object	Description	Document
---	---	---

Changes related to "Adapting an Existing Phase Building Block" (page 171):

Object	Description	Document
---	---	---

Changes related to "Appendix - Tips and Tricks" (page 181):

Object	Description	Document
---	---	---

Changes related to "Appendix - Sandbox for Phases to Manage Unhandled Situations" (page 183):

Object	Description	Document
---	---	---

Changes related to "Appendix - Sample Phase Building Blocks" (page 195):

Object	Description	Document
---	---	---

Appendix - Tips and Tricks

In this section, you will find some useful hints how to avoid pitfalls when creating system building blocks.

- Make sure to observe the naming conventions (page 5) that apply to building blocks.
- Parameters for drawing phases in the **Preview** status
  The entry point for creating your phase-related GUI is the *createPhaseComponent()* method. Please be aware that a runtime phase only exists for the **Active** or **Completed** statuses. So do not try to access the runtime phase (*IMESRtPhase getRtPhase()*) when drawing the preview (*Status.PREVIEW*); since you will get a Null Pointer Exception.
  For details see "Step 5: Creating a Phase Executor (Java Code)" (page 49).
- Persistent types in the context of phase output and transition conditions
  Phase outputs are automatically stored as activity set variables in order to be usable within transition conditions. Only activity set variables of the following data types are persistent (unless a custom persistence class is specified for the variable): simple types (*String*, *Long*, *Integer*, *Short*, *Byte*, *Float*, *Double*, *Character*, *Boolean*), *IMeasuredValue*, *BigDecimal*, *Time*, and *Keyed*. This means that in order to survive a restart of the Production Execution Client, transition conditions should only refer to outputs of a persistent data type.
  If your outputs are defined by means of the phase generator (page 19), the generator makes sure that all variables generated for the supported data types are persistent. Nevertheless, you should keep this restriction in mind.
- Methods called without an active/initialized phase
  (Does not apply to server-side phases running on the OE server.)
  Typically, the first method called for a phase is the *createPhaseComponent()* method and most of the phase initialization (e.g. initializing internal fields) takes place there. However, if your order has parallel operations, you can access the Navigator from one operation and there you see the completed phases of all other operations. Additionally, you can execute post-completion actions of all phases and add user-defined exceptions to all phases. This means that the methods listed below can be called for a phase that is not initialized by calling the *createPhaseComponent()* method:
  - *createPhaseActionComponent*
  - *getNavigatorInfoColumn*

- *exceptionCallbackInSaveTransaction*
- *exceptionSigned*
- *exceptionCanceled*

When you are using these methods, keep in mind that objects initialized in *createPhaseComponent()* can be null.

Appendix - Sandbox for Phases to Manage Unhandled Situations

This section provides information on how to manage unhandled situations that can occur during phase execution. Throughout a phase's life cycle, the PharmaSuite framework calls most methods of a phase executor. This is why the framework provides a sandbox for phases to manage unhandled situations. When you develop a phase, there can be some error situations, which you do not want to handle and expect to be handled by the framework, for example, if the network is down or the connection to the middle tier is lost.

First, the sandbox is described, followed by a detailed description how methods react to unhandled situations. Finally, some helper methods supporting sandbox development are provided.

The General Strategy of the Sandbox

When an unforeseen situation occurs during phase execution, a Java *RuntimeException* is thrown somewhere in the execution stack of the phase executor. In order to deal with the *RuntimeException*, each method of the *IPhaseExecutor*'s API runs in a sandbox environment similar to the following example of the *IPhaseExecutor.performCompletionCheck()* method.

Example code: sandbox environment

```
@Override
public boolean performCompletionCheck() {
  checkIsEDTThread();
  try {
    return phaseExecutor.performCompletionCheck();
  } catch (RuntimeException exc) {
    throw sandBoxRuntimeException(exc, "performCompletionCheck");
  }
}
```

The *phaseExecutor* variable is an instance of the phase executor created by the phase developer. If a runtime exception is thrown while executing the method, the exception is handled by the *sandBoxRuntimeException* method. Depending on the environment, i.e. the sandboxed method, a specific phase method runs the continuation after returning from the *sandBoxRuntimeException*. Usually, the sandbox leaves a phase in a safe and consistent state, if the phase has been in such a state before.

TIP

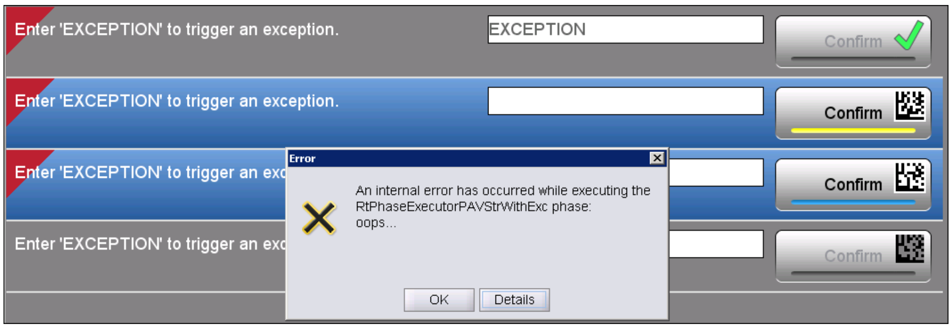
PharmaSuite provides a similar sandbox for server-run phases, which also catches exceptions and only logs them in the log files. No message is displayed.

The *sandBoxRuntimeException* method itself creates an error log and displays an internal error dialog. The error log can be found in the PharmaSuite-specific log file.

Assume, your *performCompletionCheck* looks as follows:

```
@Override
public boolean performCompletionCheck() {
  if (this != null) {
    throw new RuntimeException("oops...");
  }
  return true;
}
```

When the phase has been confirmed, the system displays the following internal error dialog.


Enter 'EXCEPTION' to trigger an exception.
EXCEPTION
Confirm
Enter 'EXCEPTION' to trigger an exception.
Confirm
Enter 'EXCEPTION' to trigger an exc
Confirm
Enter 'EXCEPTION' to trigger an exc
Confirm
Error
OK
Details

An internal error has occurred while executing the RtPhaseExecutorPAVStrWithExc phase:
oops...

Figure 43: Internal error dialog

The **Details** button displays the stack trace.
The PharmaSuite log file contains an entry similar to the following example.

Example: PharmaSuite log file

```
2012-01-27 11:23:58,202 ERROR ExceptionHandler:141 - oops...
java.lang.RuntimeException: oops...
  at
com.rockwell.mes.apps.testdata.phase.pavstringexc.RtPhaseExecutorPAVStrWithExc.performC
ompletionCheck(RtPhaseExecutorPAVStrWithExc.java:468)
  at
com.rockwell.mes.apps.ebr.ifc.phase.SandBoxPhaseExecutor.performCompletionCheck(SandBox
PhaseExecutor.java:338)
  at
com.rockwell.mes.apps.ebr.ifc.activities.PhaseExecutorActivity.completePhase(PhaseExecu
torActivity.java:332)
  at
com.rockwell.mes.apps.testdata.phase.pavstringexc.RtPhaseExecutorPAVStrWithExc$ConfirmB
uttonListener.actionPerformed(RtPhaseExecutorPAVStrWithExc.java:181)
  at javax.swing.AbstractButton.fireActionPerformed(AbstractButton.java:1995)
  at javax.swing.AbstractButton$Handler.actionPerformed(AbstractButton.java:2318)
  at javax.swing.DefaultButtonModel.fireActionPerformed(DefaultButtonModel.java:387)
  at javax.swing.DefaultButtonModel.setPressed(DefaultButtonModel.java:242)
  at
javax.swing.plaf.basic.BasicButtonListener.mouseReleased(BasicButtonListener.java:236)
  at java.awt.Component.processMouseEvent(Component.java:6263)
  at javax.swing.JComponent.processMouseEvent(JComponent.java:3267)
  at java.awt.Component.processEvent(Component.java:6028)
  at java.awt.Container.processEvent(Container.java:2041)
  at java.awt.Component.dispatchEventImpl(Component.java:4630)
  at java.awt.Container.dispatchEventImpl(Container.java:2099)
  at java.awt.Component.dispatchEvent(Component.java:4460)
  at java.awt.LightweightDispatcher.retargetMouseEvent(Container.java:4574)
  at java.awt.LightweightDispatcher.processMouseEvent(Container.java:4238)
  at java.awt.LightweightDispatcher.dispatchEvent(Container.java:4168)
  at java.awt.Container.dispatchEventImpl(Container.java:2085)
  at java.awt.Window.dispatchEventImpl(Window.java:2475)
  at java.awt.Component.dispatchEvent(Component.java:4460)
  at java.awt.EventQueue.dispatchEvent(EventQueue.java:599)
  at java.awt.EventDispatchThread.pumpOneEventForFilters(EventDispatchThread.java:269)
  at java.awt.EventDispatchThread.pumpEventsForFilter(EventDispatchThread.java:184)
  at java.awt.EventDispatchThread.pumpEventsForHierarchy(EventDispatchThread.java:174)
  at java.awt.EventDispatchThread.pumpEvents(EventDispatchThread.java:169)
  at java.awt.EventDispatchThread.pumpEvents(EventDispatchThread.java:161)
  at java.awt.EventDispatchThread.run(EventDispatchThread.java:122)
```

If the operator exits the Production Execution Client, repairs the cause of the runtime exception, and restarts the Production Execution Client, whenever possible, the operator can continue to process the operation.

Invoking Methods from the IPhaseExecutor Interface

The sandbox is active for methods directly invoked by the PharmaSuite platform. Runtime exceptions occurring in methods called by your implementation are handled by the sandbox calling your code.

For all methods of your phase implementation triggered by other events, your code must handle runtime exceptions. No sandbox is effective then. The subsequent section describes how to realize this sandbox behavior.

TIP

We do not recommend that your implementation **invokes overridden** methods from the phase executor's API. The methods are supposed to be invoked by the PharmaSuite framework in order to provide the appropriate sandbox.
The main reason for the recommendation is that the methods you override play a well-defined role within the phase's life cycle and are not intended to be called randomly.

What the Sandbox Solves

Although the sandbox handles runtime exceptions of the *IPhaseExecutor*'s API and significantly reduces the complexity related to error handling for developing phases, the sandbox does not solve all of the potential issues. In this regard, the following areas are important:

- additional action buttons (page 186),
- background execution (page 187), and
- interaction with external devices (page 187).

Additional Action Buttons

➢ Does not apply to server-side phases running on the OE server.

Usually, a phase in the **Active** status comes with a **Confirm** button. After implementing *performCompletionCheck* and *performComplete* there are no further activities required from the phase developer.

However, the phase developer can add action buttons to a phase and thus introduce persistent sub-statuses within the **Active** status of a phase.
Then, it is the responsibility of the phase developer to ensure that the phase is able to resume from each of the sub-statuses since an operator can exit the Production Execution Client at any given time. Resuming is not related to unforeseen runtime exceptions.

TIP

If the action behind an additional button performs middle tier operations, consider to put them in a transactional environment.

Provide a sandbox environment for the additional action (see *performCompletionCheck* method in "The General Strategy of the Sandbox" (page 183)).

The following code snippet demonstrates how to use the *TransactionInterceptor* for transactions. Keep in mind that the *TransactionInterceptor* forwards checked exceptions as defined by the *Callable*.

Example code: usage of TransactionInterceptor

```
@Override
public void actionPerformed(ActionEvent e) {
  boolean success = true;
  try {
    final class PerformActionInTransaction implements Callable<Void> {
      @SuppressWarnings("PMD.SignatureDeclareThrowsException")
      @Override
      public final Void call() throws Exception {
        // Do your stuff
        return null;
      }
    }
    TransactionInterceptor.callInTransactionImpl(new PerformActionInTransaction());
  } catch (Exception exc) {
    success = false;
    if (exc instanceof RuntimeException) {

      PhaseExecutorHelper.sandBoxRuntimeException((RuntimeException) exc,
                         "PerformActionInTransaction", "call", true);
    } else {
      // handle checked exceptions
    }
  }
  if (!success) {
    // Do something
  }
}
```

Background Execution

There are situations where phase processing is not fully user interface-driven although it always starts with the user interface creation.

Assume, a phase pushes certain activities into background tasks, for example with the *Future<T> ExecutorService.submit(Callable<T>)* method. An unhandled exception that occurs during the execution of the *Callable<T>* is captured within the *Future<T>* return value and it is up to the submitter to extract it from there.

If the background task performs transactional middle tier calls, keep in mind that the task is typically executed in a different thread than the submitter and that transactions are thread-local, i.e. do not expect that their submitter is able to rollback actions performed by the background.

Sometimes a long-running task is submitted to background processing for performance reasons and its submitter never processes the return value from the background task. In this case, the occurrence of a runtime exception that happens within the background task can easily get lost.

TIP

Handle runtime exceptions that happen within a background task inside the background task.

Interaction with External Devices

Due to the variety of external devices a general guidance cannot be given for this area. PharmaSuite provides transaction handling, which allows to rollback a middle tier transaction, but that does not undo actions triggered on external devices like scales, OPC connectors, or messaging.

Additionally, interactions with external devices can be long-running, for example, an internal calibration of a scale can take a few minutes. We recommend to avoid running such tasks within a user transaction.

How the Framework Reacts to RuntimeExceptions

This section applies to the methods that can be overridden by a phase.

It describes how the PharmaSuite framework reacts to runtime exceptions thrown by the methods. It is assumed that each phase implementation extends the *AbstractPhaseExecutor* or *AbstractServerSidePhaseExecutor*, respectively.

TIP

You can expect that all non-final public methods of the *AbstractPhaseExecutor* or the *AbstractServerSidePhaseExecutor* are subject to overriding. However, it is highly recommended that you override only the methods listed below.
If you override other methods, you may break existing functionality through unforeseen side effects.
Some of these unlisted methods are not final in order to ensure backward compatibility.

For each method, details about the sandbox behavior and its usage within the PharmaSuite framework are listed.

Methods related to user interface creation	
createPhaseCControl (not available for AbstractServerSidePhaseExecutor)	
Framework usage	Is invoked by the phase panel activity when rendering the operation.
Sandbox behavior	*RuntimeException* is written to the PharmaSuite log file. Displays an internal error dialog if the phase is in the **Active** or **Completed** status. In the **Preview** status, runtime exceptions are accepted due to possibly missing input parameters. Renders a default control that contains a single localizable label. No buttons are available. A phase in the **Active** status cannot be processed since there is no **Confirm** button. TIP Server-run phases only log the errors and do not display a dialog. *RuntimeException* will be absorbed by the sandbox.

createPhaseActionCControl **createPhaseExceptionCControl (not available for AbstractServerSidePhaseExecutor)**	
Framework usage	Is invoked by the phase panel activity when rendering the operation.
Sandbox behavior	*RuntimeException* is written to the PharmaSuite log file. Displays an internal error dialog. Renders a default control that contains a single localizable label. No buttons are available. It is not possible to perform a user-triggered or a post-completion exception. *RuntimeException* will be absorbed by the sandbox.
Methods related to phase completion	
enableConfirmButton (not available for AbstractServerSidePhaseExecutor)	
Framework usage	PharmaSuite invokes this method when rendering UI extensions for phases in the **Active** status. This is handled within the same sandbox as *createPhaseCControl*. Therefore the same default control is shown and the *RuntimeException* will be absorbed by this sandbox.
Sandbox behavior	*RuntimeException* is written to the PharmaSuite log file. Displays an internal error dialog. *RuntimeException* will be re-thrown as *MESSandBoxException*.
performComplete	
Framework usage	PharmaSuite invokes this method if the **Confirm** button is executed (*IPhaseCompleter.completePhase()* implemented by *PhaseExecutorActivity*). The re-thrown *RuntimeException* interrupts the user transaction, which causes a rollback of that transaction. The re-thrown *RuntimeException* is finally handled by the Event Dispatching Thread. TIP Server-run phases must explicitly call *performComplete()*.
Sandbox behavior	*RuntimeException* is written to the PharmaSuite log file. Displays an internal error dialog. Phase remains in the **Active** status. TIP Server-run phases only log the errors and do not display a dialog. *RuntimeException* will be re-thrown as *MESSandBoxException*.

performCompletionCheck	
Framework usage	PharmaSuite invokes this method if the **Confirm** button is executed (*IPhaseCompleter.completePhase()* implemented by *PhaseExecutorActivity*). The re-thrown *RuntimeException* is finally handled by the Event Dispatching Thread. TIP Server-run phases must explicitly call *performComplete()*, which internally calls *performCompletionCheck()*.
Sandbox behavior	*RuntimeException* is written to the PharmaSuite log file. Displays an internal error dialog. Phase remains in the **Active** status. TIP Server-run phases only log the errors and do not display a dialog. *RuntimeException* will be re-thrown as *MESSandBoxException*.
Methods related to exception recording	
getExceptionRecordingCallbackConsumer (not available for AbstractServerSidePhaseExecutor)	
Framework usage	Is invoked by the exception view controller twice. Therefore, if this method throws a runtime exception, it occurs twice, leading to two subsequent internal error dialogs. Is first invoked by the exception view controller within the transaction that records a user- or a system-triggered exception. The re-thrown *RuntimeException* breaks this transaction and causes a rollback - at a very early stage of the transaction. The exception view controller absorbs the re-thrown *RuntimeException*. Is invoked a second time by the exception view controller when notifying the phase about the success (or failure) of recording a user- or a/system-triggered exception. The exception view controller absorbs the re-thrown *RuntimeException*.
Sandbox behavior	*RuntimeException* is written to the PharmaSuite log file. Displays an internal error dialog. *RuntimeException* will be re-thrown as *MESSandBoxException*. The *AbstractPhaseExecutor* provides an overridable implementation that returns this.

exceptionCallbackInSaveTransaction (not available for AbstractServerSidePhaseExecutor)	
Framework usage	Is invoked by the exception view controller within the transaction that records a user- or a system-triggered exception. The *RuntimeException* breaks this transaction and causes a rollback. The exception view controller provides the sandbox for this method. *RuntimeException* is written to the PharmaSuite log file. Displays an internal error dialog. The exception view controller absorbs the *RuntimeException*.
Sandbox behavior	Is not part of the sandbox, because it is not a member of the *IPhaseExecutor* interface. An empty, overridable implementation is provided by the *AbstractPhaseExecutor*.
exceptionCanceled (not available for AbstractServerSidePhaseExecutor)	
Framework usage	Is invoked by the exception view controller after recording a system-triggered exception has been canceled by the operator (with the **Back** and **Yes** buttons). The exception view controller provides the sandbox for this method. *RuntimeException* is written to the PharmaSuite log file. Displays an internal error dialog. The exception view controller absorbs the *RuntimeException*.
Sandbox behavior	Is not part of the sandbox, because it is not a member of the *IPhaseExecutor* interface. An empty, overridable implementation is provided by the *AbstractPhaseExecutor*.
exceptionSigned (not available for AbstractServerSidePhaseExecutor)	
Framework usage	Is invoked by the exception view controller after recording a system-triggered exception by the middle tier. It is not part of the user transaction being responsible for recording the exception. The exception view controller provides the sandbox for this method. *RuntimeException* is written to the PharmaSuite log file. Displays an internal error dialog. The exception view controller absorbs the *RuntimeException*.
Sandbox behavior	Is not part of the sandbox, because it is not a member of the *IPhaseExecutor* interface. An empty, overridable implementation is provided by the *AbstractPhaseExecutor*.

Methods related to the Navigator	
getActionButtonTexts (not available for AbstractServerSidePhaseExecutor)	
Framework usage	Is invoked by *NavigatorViewActivity* to render the action buttons for phases in the **Completed** status. The *RuntimeException* is completely handled by the sandbox. No specific handling within the Navigator.
Sandbox behavior	*RuntimeException* is written to the PharmaSuite log file. Displays an internal error dialog. A single action button with a localizable message is rendered. The message should indicate the error situation. *RuntimeException* will be absorbed by the sandbox.
getNavigatorInfoColumn (not available for AbstractServerSidePhaseExecutor)	
Framework usage	Is invoked by the *NavigatorViewActivity* only if the phase library entry associated with the phase does not declare a Pnuts subroutine being responsible for the Navigator's information column. The *NavigatorViewActivity* creates its own sandbox for the entire task. It catches runtime exceptions thrown by either the subroutine processing or the phase executor’s method (*getNavigatorInfoColumn*). The *NavigatorViewActivity* renders a default, localizable message. The message should indicate the error situation. The *NavigatorViewActivity* absorbs the *RuntimeException*.
Sandbox behavior	*RuntimeException* is written to the PharmaSuite log file. Displays an internal error dialog. *RuntimeException* will be re-thrown as *MESSandBoxException*.
Other methods	
holdPhaseExecution	
Framework usage	The sandbox method is not invoked by PharmaSuite. Only the non-sandboxed method is invoked. This happens when the operation executor puts the operation on hold. TIP Provide your own sandbox, if you wish to implement this method.
Sandbox behavior	*RuntimeException* is written to the PharmaSuite log file. Displays an internal error dialog. *RuntimeException* will be re-thrown as *MESSandBoxException*. The *AbstractPhaseExecutor* provides an overridable, empty default implementation.

Helper Methods

The *PhaseExecutorHelper* class provides some static helper methods. They support the phase developer when implementing his own sandbox behavior.

- *handleExceptionWithDialog* (*not available for AbstractServerSidePhaseExecutor*)
  This method is used to document the exception in the PharmaSuite error log. It also displays the internal error dialog.
  Its argument indicates whether the dialog shall be drawn immediately or through *SwingUtilities.invokeLater*. The latter is important if you invoke the dialog not from the Event Dispatcher Thread, e.g. from background tasks.
- *sandBoxRuntimeException* (*not available for AbstractServerSidePhaseExecutor*)
  This method checks if the *RuntimeException* is an *MESSandBoxRuntimeException*. If this is not the case, the *handleExceptionWithDialog* will be invoked and the *RuntimeException* will be packed into the *MESSandBoxRuntimeException* and returned.
  If this method is already invoked with an *MESSandBoxRuntimeException*, it is assumed that the exception has already been logged and shown to the operator. Then, the exception is returned as is. No additional logging or dialogs will be created.

Appendix - Sample Phase Building Blocks

This section provides sample phase building blocks that can be used as templates or within other phase building blocks.

General Information

The following characteristics apply to each sample phase building block. The included files and classes depend on the type of phase building block (e.g. whether data is stored or not). Some of the included classes remain as created; they will only be modified if required.
<PhaseBB> is the name of the sample building block in upper camel case, <phasebb> is the name of the sample building block in lower camel case, <installation directory> is the directory of your PharmaSuite installation.

- Java package
  - *com.rockwell.mes.phase.example.<phasebb>*
    Package containing the sample phase building block.
- Included files
  Input files related to data description and output description are optional.
  - *<installation directory>/config/phase/<phasebb>/PhaseDescription.xml*
    Input file used to create the source files of the sample phase building block by the phase generator.
  - *<installation directory>/config/phase/<phasebb>/PhaseDataDescription.xml*
    Input file used to create the data source files of the sample phase data building block by the phase generator.
  - *<installation directory>/config/phase/<phasebb>/PhaseOutputDescription.xml*
    Input file used to create the source files of the sample runtime phase output by the phase generator.
- Included classes
  Classes related to phase data and phase output are optional.
  - GUI, created by the phase generator.
    com.rockwell.mes.phase.example.<phasebb>.RtPhaseExecutor<PhaseBB>

- Runtime phase data bean, created by the phase generator.
  com.rockwell.mes.phase.example.<phasebb>.MESRtPhaseData<PhaseBB>
- Base class of *MESRtPhaseData<PhaseBB>*, created by the phase generator.
  com.rockwell.mes.phase.example.<phasebb>.MESGeneratedRtPhaseData<PhaseBB>
- Runtime phase data bean filter, created by the phase generator.
  com.rockwell.mes.phase.example.<phasebb>.MESRtPhaseData<PhaseBB>Filter
- Base class of *MESRtPhaseData<PhaseBB>Filter*, created by the phase generator.
  com.rockwell.mes.phase.example.<phasebb>.MESGeneratedRtPhaseData<PhaseBB>Filter
- Runtime phase output bean, created by the phase generator.
  com.rockwell.mes.phase.example.<phasebb>.MESRtPhaseOutput<PhaseBB>
- Base class of *MESRtPhaseOutput<PhaseBB>*, created by the phase generator.
  com.rockwell.mes.phase.example.<phasebb>.MESGeneratedRtPhaseOutput<PhaseBB>
- Runtime phase output bean filter, created by the phase generator.
  com.rockwell.mes.phase.example.<phasebb>.MESRtPhaseOutput<PhaseBB>Filter
- Base class of *MESRtPhaseOutput<PhaseBB>Filter*, created by the phase generator.
  com.rockwell.mes.phase.example.<phasebb>.MESGeneratedRtPhaseOutput<PhaseBB>Filter

Samples

The following sample phase building blocks are available. Starting with a simple text output, the sample phase building blocks increase in complexity:

- HelloWorld (page 197)
- GetValue (page 198)

Sample: HelloWorld

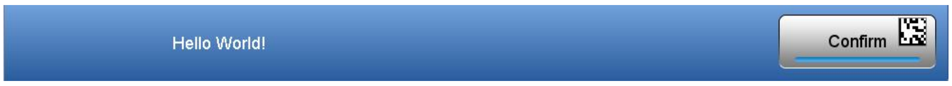


Figure 44: HelloWord in Active status

- **Name**: HelloWorld (RS) [1.0]
- **Purpose**: Text output.
- **Input parameter**: N/A
- **Output parameter**: N/A
- **Action**: N/A
- **Exception**: N/A
- **Report**: N/A
- **Java project**: *apps.example*
- **Java package**: *com.rockwell.mes.phase.example.helloworld*
- **Included files**: *config/phase/helloworld/PhaseDescription.xml*
- **Included classes**:
    - *com.rockwell.mes.phase.example.helloworld.RtPhaseExecutorHelloWorld0100*
- **Code snippet**:

Code example of the user interface

```
/**
 * REMARK: This method is called when the phase is rendered by the
 * framework.
 * {@inheritDoc}
 *
 * @see
com.rockwell.mes.apps.ebr.ifc.swing.AbstractPhaseExecutorSwing#createPhaseComponent()
 */
@Override
public JComponent createPhaseComponent() {
  JPanel layoutPanel =
         PhaseSwingHelper.createPanel(Layout.LAYOUT_SINGLE_COLUMN, getStatus());

  JLabel label = PhaseSwingHelper.createJLabel(LABEL_TEXT);
  // label shall be directly on top after insets
  label.setVerticalAlignment(SwingConstants.TOP);
  layoutPanel.add(label, Column.FIRST_COLUMN);
  if (getStatus() == Status.ACTIVE) {
    /** confirm button listener */
    class ConfirmButtonListener implements ActionListener {
      @Override
      public void actionPerformed(ActionEvent event) {
        LOGGER.debug("Button clicked.");
        getPhaseCompleter().completePhase();
      }
    }
    ConfirmButton confirmButton = ((PhaseColumnLayout)
                                  layoutPanel.getLayout()).getConfirmButton();
    confirmButton.addActionListener(new ConfirmButtonListener());
  }
  return layoutPanel;
}
```

Code example of phase completion with confirm

```
/**
 * REMARK: This method is called when the complete() method has been called
 * by this class. Normally this is when the user has pressed the complete
 * button. In this case the performCompletionCheck() is called. If the check
 * returns true this method is called.
 * {@inheritDoc}
 *
 * @see com.rockwell.mes.apps.ebr.ifc.phase.IPhaseExecutor#performComplete()
 */
@Override
public Response performComplete() {
  return new Response();
}
```

Sample: GetValue

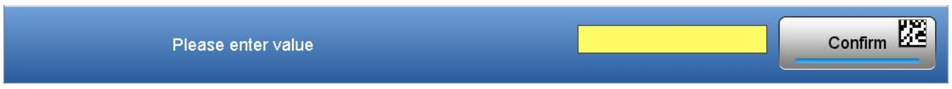


Figure 45: GetValue in Active status

- **Name**: GetValue (RS) [1.0]
- **Purpose**: Stores phase-specific data, which was provided by an operator. Overwrites the *getNavigatorInfoColumn()* method to display phase-specific data (*SomeStringValue* variable) in the Navigator instead of the phase name (default).
- **Input parameter**: N/A
- **Output parameter**: N/A
- **Action**: N/A
- **Exception**: N/A
- **Report**: N/A
- **Java project**: *apps.example*
- **Java package**: *com.rockwell.mes.phase.example.getvalue*
- **Included files**:
  - *config/phase/getvalue/Phasedescription.xml*

- *config/phase/getvalue/PhaseDataDescription.xml*
- **Included classes**:
  - *com.rockwell.mes.phase.example.getvalue.RtPhaseExecutorGetValue0100*
  - *com.rockwell.mes.phase.example.getvalue.MESRtPhaseDataGetValue0100*
  - *com.rockwell.mes.phase.example.getvalue.MESGeneratedRtPhaseDataGetValue0100*
  - *com.rockwell.mes.phase.example.getvalue.MESRtPhaseDataGetValue0100Filter*
  - *com.rockwell.mes.phase.example.getvalue.MESGeneratedRtPhaseDataGetValue0100Filter*
  - *com.rockwell.mes.phase.example.getvalue.MESRtPhaseOutputGetValue0100*
  - *com.rockwell.mes.phase.example.getvalue.MESGeneratedRtPhaseOutputGetValue0100*
  - *com.rockwell.mes.phase.example.getvalue.MESRtPhaseOutputGetValue0100Filter*
  - *com.rockwell.mes.phase.example.getvalue.MESGeneratedRtPhaseOutputGetValue0100Filter*
- **Code snippet**:

Code example of the user interface

```
/**
 * REMARK: This method is called when the phase is rendered by the
 * framework.
 * {@inheritDoc}
 *
 * @see
com.rockwell.mes.apps.ebr.ifc.swing.AbstractPhaseExecutorSwing#createPhaseComponent()
 */
@Override
public JComponent createPhaseComponent() {
  final JPanel layoutPanel =
        PhaseSwingHelper.createPanel(Layout.LAYOUT_TWO_1ST_WIDER_COLUMN, getStatus());
  JLabel label = PhaseSwingHelper.createJLabel(getPhase().getName());
  layoutPanel.add(label, Column.FIRST_COLUMN);

  edit = PhaseSwingHelper.createJTextField(getStatus());
  layoutPanel.add(edit, Column.SECOND_COLUMN);

  if (getStatus() == Status.ACTIVE) {
    /** confirm button listener */
    class ConfirmButtonListener implements ActionListener {
      @Override
      public void actionPerformed(ActionEvent event) {
        text = edit.getText();
        LOGGER.debug("Button clicked:" + (text == null ? "" : text));
        getPhaseCompleter().completePhase();
      }
    }
    ConfirmButton confirmButton =
                   ((PhaseColumnLayout) layoutPanel.getLayout()).getConfirmButton();
    confirmButton.addActionListener(new ConfirmButtonListener());
  }

  if (getStatus() == Status.COMPLETED) {
    MESRtPhaseDataGetValue0100data = getRtPhaseData;
    edit.setText(data.getActualValue());
  }
  return layoutPanel;
}
```

Code example of phase completion with confirm

```
/**
 * REMARK: This method is called when the complete() method has been called
 * by this class. Normally this is when the user has pressed the complete
 * button. In this case the performCompletionCheck() is called. If the check
 * returns true this method is called.
 * {@inheritDoc}
 *
 * @see com.rockwell.mes.apps.ebr.ifc.phase.IPhaseExecutor#performComplete()
 */
@Override
public Response performComplete() {
  // set runtime phase data
  LOGGER.debug("Inserted runtime phase data: " + text);
  MESRtPhaseDataGetValue0100 data = addRtPhaseData();
  data.setActualValue(text);
  // get output bean and set the output
  MESRtPhaseOutputGetValue0100 output = getRtPhaseOutput ();
  output.setActualValue(text);

  return new Response();
}
```

Index