LISTEN.
THINK.
SOLVE.®

PharmaSuite®

PHASES OF THE DISPENSE PACKAGE

RELEASE 8.4

TECHNICAL MANUAL

PUBLICATION PSDI-PM005E-EN-E-DECEMBER-2017
Supersedes publication PSDI-PM005D-EN-E

Allen-Bradley • Rockwell Software

Rockwell Automation

Contact Rockwell See contact information provided in your maintenance contract.

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

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

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

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

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

ControlNet is a registered trademark of ControlNet International.

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

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

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

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

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

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

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

Contents

Chapter 1 Introduction 1

Intended Audience 3

Naming Recommendations 3

Typographical Conventions 3

Chapter 2 Administration 5

Configuring Cost Center-related Dispensing 5

Chapter 3 Configuration and Extension 7

Extension Use Case: Adding a Recipe-related Usage Type for Shop Floor-Defined Dispensing 7

Configuring the RecipeUsageType Choice List 8

Adapting Label Layouts 8

Configuring the Usage Type-specific Configuration Keys 9

Assigning Users and Stations 12

Reference Documents 12

Configuration Keys Specific to the D Tare Phase 13

Configuration Keys Specific to the O Tare Phase 13

Chapter 4 Supported Scale Modes 15

Chapter 5 Adapting Phases for Dispense 17

Structure of Dispense Operations 18

Weighing Methods 19

Technical Details of Dispense Phases 23

The Model-View-Controller Concept of the Dispense Phases 23

Context and Phase Data 25

Material Identification 26

Container Integration for Dispense 30
Helper Methods 30
Exceptions 31
Navigator Actions 33
Dispensing Report and Sub-reports 33
Tips and Tricks 33
Extension Use Cases 34
Adapting an Existing Phase 35
Adding a New Phase 35
Extending the Context 36

Chapter 6 Adapting Phases for Inline Weighing 39
Structure of Inline Weighing Operations and Unit Procedure 39
Weighing Methods 40
No Target Sublot Creation for Inline Weighing 41
Container Integration for Inline Weighing 41
Technical Details Specific to Inline Weighing 42

Chapter 7 Adapting Phases for Output Weighing 43
Output Weighing Uses OrderStepOutput 43
Structure of Output Weighing Operations (with Loop) 44
Weighing Methods 45
Different Operation Modes of Output Weighing 46
One-step Output Weighing 46
Two-step Output Weighing 46
Modeling 47
Weighing of Several Outputs 49
Specifics Related to the O Manage Produced Material Phase 50
Yield and Prorate Factor Calculation 52
User-triggered Exceptions 52
System-triggered Exceptions 53
Specifics Related to the O Identify Container Phase 53

Specifics Related to the O Tare Phase 54
Specifics Related to the O Weigh Phase 54
Container Integration for Output Weighing 54
Units of Measure Conversions..... 55
Target Weight Enforcement 56
Access to ScaleEquipment and Scales Objects..... 56

Chapter 8 Adapting Phases for Weighing Support..... 59
Phase Modes of the Get Weight Phase 59
Consistency Checks..... 61

Chapter 9 Reference Documents 63

Chapter 10 Revision History..... 65

Index 69

Figures

Figure 1: Example structure of a Dispense operation 18
Figure 2: MVC class diagram 25
Figure 3: Busy painter 31
Figure 4: Fall-through exception 32
Figure 5: Example of a typical Inline Weighing unit procedure with a loop and transitions 39
Figure 6: Typical Inline Weighing operation with loops and transitions 40
Figure 7: Example structure of an Output Weighing operation 44
Figure 8: Two-step Output Weighing scenario: preparation and weighing of one output material (sequential) 47
Figure 9: Two-step Output Weighing scenario: preparation and weighing of one output material (parallel) 48
Figure 10: Output Weighing scenario: direct weighing of several output materials 49
Figure 11: Yield and prorate factor calculation 52
Figure 12: Class hierarchy of PhaseModeHandler 60

Chapter 1

Introduction

This documentation contains important information how to administer, configure, and adapt phase building blocks of the Dispense Phase Package including information about actions that need to be performed in FactoryTalk® ProductionCentre and PharmaSuite.

The manual is an addition to the "Technical Manual Administration" [A1] (page 63) and the "Technical Manual Configuration and Extension" [A2]-[A6] (page 63).

The information is structured in the following sections:

Administration

In this part you will find tasks for system administrators of PharmaSuite at a customer's site.

- **D Define order**, **D Edit BOM**, **D Dispatch order**, **D Release order** phases
  - Configuring cost center-related dispensing (page 5)

Configuration and Extension

In this part you will find information for system engineers who implement customer-specific configurations and extensions.

- **D Define order**, **D Edit BOM**, **D Dispatch order**, **D Release order** phases
  - Extension use case:
    Adding a recipe-related usage type for shop-floor-defined dispensing (page 7)
- Configuration keys
  - **D Tare** phase (page 13)
  - **O Tare** phase (page 13)

Supported Scale Modes

- **D Tare**, **D Weigh**, **D Release scale**, **O Tare**, **O Weigh**, **O Release scale**, **Get weight** phases
  - Supported scale modes (page 15)

Adapting Phases for Dispense

- **D Identify material**, **Show GHS data**, **D Select scale**, **D Tare**, **D Weigh**, **D Release scale**, **D Print report** phases
  - Structure of Dispense operations (page 18)
  - Weighing methods (page 19)

- Technical details of Dispense phases (page 23)
- Extension use cases (page 34)

ADAPTING PHASES FOR INLINE WEIGHING

- **D Identify material**, **Show GHS data**, **D Select scale**, **D Tare**, **D Weigh**, **D Release scale** phases
  - Structure of Inline Weighing operations and unit procedure (page 39)
  - Weighing methods (page 40)
  - No target sublot creation for Inline Weighing (page 41)
  - Container integration for Inline Weighing (page 41)
  - Technical details specific to Inline Weighing (page 42)

ADAPTING PHASES FOR OUTPUT WEIGHING

- **O Manage produced material**, **Show GHS data**, **O Select scale**, **O Identify container**, **O Tare**, **O Weigh**, **O Release scale** phases
  - Output Weighing uses OrderStepOutput (page 43)
  - Structure of Output Weighing operations (with loop) (page 44)
  - Weighing methods (page 45)
  - Different operation modes of Output Weighing (page 46)
  - Specifics related to the O Manage produced material phase (page 50)
  - Specifics related to the O Identify container phase (page 53)
  - Specifics related to the O Tare phase (page 54)
  - Specifics related to O Weigh phase (page 54)
  - Container integration for Output Weighing (page 54)
  - Units of measure conversions (page 55)
  - Access to ScaleEquipment and Scales objects (page 56)

ADAPTING PHASES FOR WEIGHING SUPPORT

- **Get weight** phase
  - Phase modes of the Get weight phase (page 59)
  - Consistency checks (page 61)

Finally, the Reference Documents (page 63) section provides a list of all the documentation that is referenced in this manual.

Intended Audience

This manual is intended for system engineers, phase developers, and system administrators who maintain phase building blocks of the Dispense Package.

They need to have a thorough working knowledge of FactoryTalk ProductionCentre, PharmaSuite, and PharmaSuite building blocks. Thus we highly recommend to participate in a FactoryTalk ProductionCentre and PharmaSuite training before starting on the tasks described in this manual.
For more information on S88, EBR, and phase building blocks, see also "Technical Manual Developing System Building Blocks" [A7] (page 63).

Naming Recommendations

When naming your artifacts (e.g. objects in Process Designer, classes, interfaces, methods, functions, building blocks) consider the following recommendation:

- Define and make use of a vendor code consisting of up to three uppercase letters as prefix (e.g. MYC for My Company).
- Prefix your artifacts with your vendor code, separated by _.
  X_ and **RS_** are reserved for PharmaSuite.
- Observe the naming conventions defined in "Technical Manual Developing System Building Blocks" [A7] (page 63).

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

Administration

This chapter applies to the **D Define order**, **D Edit BOM**, **D Dispatch order**, and **D Release order** phases. You will learn how to configure PharmaSuite for cost center-related dispensing based on a shop floor-defined order workflow.

Configuring Cost Center-related Dispensing

The following areas are affected:

- Cost center
  To adapt the list of available cost centers, proceed as follows:
  1. In Process Designer, expand the **Lists** node and select the **CostCenters** list.
  2. If version control is enabled, click the **Check out** button.
  3. Edit the list as required.
     Each line defines a cost center with identifier and name. Identifier and name are separated by a space (<cost center identifier> <cost center name>).
     Example: The line `100070 200mg gelcaps` defines the cost center with the identifier **100070** and the name **200mg gelcaps**.
  4. Save and check in the list.
- Product batch
  The default batch prefix is defined with the **CostCenter.BatchPrefix** configuration key.
  For details, please refer to chapter "Configuration Keys of PharmaSuite" in Volume 4 of the "Technical Manual Configuration and Extension" [A3] (page 63).
- Target sublot
  The consumption of the target sublot is controlled with the **CostCenter.AutoConsumeStandaloneTargetSublot** configuration key.
  For details, please refer to chapter "Configuration Keys of PharmaSuite" in Volume 4 of the "Technical Manual Configuration and Extension" [A3] (page 63).
- Report and label settings
  The following settings can be made by specific configuration keys:
  - Report design of the batch report: **CostCenter.Main**

- Label (report) design of the dispensing label: **CostCenter.DispensingLabel**
- Label (report) design of the pallet or container sublot label: **CostCenter.PalletContainerSublotLabel**
- Number of labels to be printed for the target sublot: **CostCenter.LabelCount4TargetSublot**

For details, please refer to chapter "Configuration Keys of PharmaSuite" in Volume 4 of the "Technical Manual Configuration and Extension" [A3] (page 63).

Chapter 3

Configuration and Extension

This chapter applies to the following phases:

- **D Define order**, **D Edit BOM**, **D Dispatch order**, and **D Release order**. You will learn how to adapt PharmaSuite for the execution of orders (recipes) with a specific usage type (here: **Experimental batch**) for shop floor-defined dispensing.
- **D Tare** and **O Tare**. You will learn which phase-specific configuration keys are available.

For general information on configuration keys, see chapter "Managing Configurations" in Volume 4 of the "Technical Manual Configuration and Extension" [A5] (page 63).

TIP

Configuration keys are modified and defined in Process Designer.

Extension Use Case: Adding a Recipe-related Usage Type for Shop Floor-Defined Dispensing

PharmaSuite supports two types of orders: production-related orders that are defined in the Production Management Client and orders that are created at the shop floor level. The latter are characterized by a usage type other than **Production**. The usage type is related to a master recipe and its orders.

The extension use case applies to dispensing of the **Experimental batch** type with MFC-relevant material parameters and a production quantity assigned to the master recipe. The master recipe consists of exactly one unit procedure with one operation. After weighing a position, two labels with a specific layout shall be printed.
At order start, the batch identifier of an experimental batch is automatically created. The prefix shall be **EX**.
After completion of the order, the quantity of the target sublots shall be consumed.
The batch report is identical to the one for production orders.

The description covers the following areas:

- Configure the **RecipeUsageType** choice list (page 8) to make the **Experimental batch** type available for the recipes and orders.
- Adapt the layout of the dispensing and the pallet and container sublot labels (page 8).

- Configure the usage type-specific configuration keys (page 9) to control the system behavior
  For an overview of the available configuration keys and their settings, see section "Configuration Keys for Recipe-related Usage Types" (page 11).
- Optional
  Assign your **Application** object to specific users and stations.

Finally, create a master recipe of the **Experimental batch** usage type and use the workflow phases (**D Define order**, **D Edit BOM**, **D Dispatch order**, **D Release order**) to create the corresponding order. Then the order is available for processing in the Production Execution Client.

Configuring the RecipeUsageType Choice List

To configure the **RecipeUsageType** choice list, proceed as follows:

Step 1

Add the **ExperimentalBatch** usage type as choice list element to the **RecipeUsageType** choice list

- For details, see [UC1] (page 12)
- Example meaning: ExperimentalBatch
- Example value: 25
- Example message ID: RecipeUsageType-ExperimentalBatch

Step 2

Add a message for the new **ExperimentalBatch** usage type to the **MESChoiceListStrings** message pack

- Example message ID: RecipeUsageType-ExperimentalBatch
- Example message: Experimental batch

Adapting Label Layouts

To adapt the dispensing and the pallet and container sublot labels to be used by the **Experimental batch** usage type, proceed as follows:

Step 1

Create the **Ct_XBRelatedSublotLabelRD** and **Ct_XBRelatedSublotLabelPalletRD** report designs

- For details, see [UC3] (page 12)

Configuring the Usage Type-specific Configuration Keys

To configure the configuration keys for the **Experimental batch** usage type, proceed as follows:

Step 1

Add the **LibraryHolder/services-wd-impl.jar/ExperimentalBatchMasterRecipeQuantityCheckApplicable** configuration key to your **Application** object to enable the check specific to the master recipe quantity

- For details, see [UC2] and [UCA4] (page 12)
- Add a property to the **Library** object
- Example value: True

Step 2

Modify the **LibraryHolder/services-wd-impl.jar/ExperimentalBatch.MasterRecipeUsageTypeStructureCheckApplicable** configuration key in your **Application** object to enable the check specific to the structure of the master recipe

- For details, see [UC2] and [UC4] (page 12)
- Example value: True

Step 3

Add the **LibraryHolder/services-wd-impl.jar/ExperimentalBatch.MaterialParameterCountsCheckApplicable** configuration key to your **Application** object to enable the check specific to the material parameters

- For details, see [UC2] and [UC4] (page 12)
- Add a property to the **Library** object
- Example value: True

Step 4

Add the **LibraryHolder/services-inventory-impl.jar/ExperimentalBatch.BatchPrefix** configuration key to your **Application** object to define the batch prefix for automatic batch creation

- For details, see [UC2] and [UC4] (page 12)
- Add a property to the **Library** object
- Example value: EX

Step 5

Add the **LibraryHolder/services-wd-impl.jar/ExperimentalBatch.DispensingLabel** configuration key to your **Application** object to define the report design to be used for dispensing labels

- For details, see [UC2] and [UC4] (page 12)
- Add a property to the **Library** object
- Example value: Ct_XBRelatedSublotLabelRD

Step 6

Add the **LibraryHolder/services-wd-impl.jar/ExperimentalBatch.PalletContainerSublotLabel** configuration key to your **Application** object to define the report design to be used for pallet and container sublot labels

- For details, see [UC2] and [UC4] (page 12)
- Add a property to the **Library** object
- Example value: Ct_XBRelatedSublotLabelPalletRD

Step 7

Add the **LibraryHolder/services-wd-impl.jar/ExperimentalBatch.AutoConsumeStandaloneTargetSublot** configuration key to your **Application** object to define the automatic consumption behavior

- For details, see [UC2] and [UC4] (page 12)
- Add a property to the **Library** object
- Example value: True

Step 8

Add the **LibraryHolder/services-wd-impl.jar/ExperimentalBatch.LabelCount4TargetSublot** configuration key to your **Application** object to define the number of target sublot labels to be printed

- For details, see [UC2] and [UC4] (page 12)
- Add a property to the **Library** object
- Example value: 2

Step 9

Add the **LibraryHolder/services-wd-impl.jar/ExperimentalBatch.Main** configuration key to your **Application** object to define the report design to be used for the batch report

- For details, see [UC2] and [UC4] (page 12)
- Add a property to the **Library** object
- Example value: PS-BatchReport-Main

Configuration Keys for Recipe-related Usage Types

PharmaSuite provides a set of configuration keys for a usage type to control the behavior of Recipe and Workflow Designer and the Production Execution Client depending on the usage type of the respective recipe or order.

The table shows the configuration keys and their settings for the **Production** and **Cost center** default usage types and the **Experimental batch** usage type of the extension use case.

- For more details, see [A2] (page 12)

Configuration key (Type)	Production	Cost center	Experimental batch
LibraryHolder/services-wd-impl.jar/<usageType>.MasterRecipeQuantityCheckApplicable (**Type**: Boolean)	True	False	True
LibraryHolder/services-wd-impl.jar/Production.MasterRecipeUsageTypeStructureCheckApplicable (**Type**: Boolean)	False	True	True
LibraryHolder/services-wd-impl.jar/<usageType>.MaterialParameterCountsCheckApplicable (**Type**: Boolean)	True	False	True
LibraryHolder/services-inventory-impl.jar/<usageType>.BatchPrefix (**Type**: String)	BX	CX	EX
LibraryHolder/services-wd-impl.jar/<usageType>.DispensingLabel (**Type**: Object - ReportDesign)	MESOrderRelatedSublotLabelRD	MESCostCenterRelatedSublotLabelRD	Ct_XBRelatedSublotLabelRD
LibraryHolder/services-wd-impl.jar/<usageType>.PalletContainerSublotLabel (**Type**: Object - ReportDesign)	MESOrderRelatedSublotLabelPalletRD	MESCostCenterRelatedSublotLabelPalletRD	Ct_XBRelatedSublotLabelPalletRD
LibraryHolder/services-wd-impl.jar/<usageType>.AutoConsumeStandaloneTargetSublot (**Type**: Boolean)	True	False	True
LibraryHolder/services-wd-impl.jar/<usageType>.LabelCount4TargetSublot (**Type**: Long)	1	1	2

Configuration key (Type)	Production	Cost center	Experimental batch
LibraryHolder/services-wd-impl.jar/<usageType> .Main (**Type**: Object - ReportDesign)	PS-BatchReport-Main	PS-BatchReport-Main	PS-BatchReport-Main

Assigning Users and Stations

This step is optional.

To define the users and stations to be used by the **Experimental batch** usage type, proceed as follows:

Step 1

Assign your **Application** object to specific users and stations

- For details, see [UC4] (page 12)

Reference Documents

The following documents provide more details relevant to the implementation of the extension use case and are available from the Rockwell Automation Download Site.

No.	Keyword	Document Title	Part Number
UC1	■ Choice lists	PharmaSuite Technical Manual Configuration & Extension - Volume 2: Adapting and Adding Field Attributes ■ Choice Lists	PSCEV2-GR008E-EN-E
UC2	■ Configuration Keys	PharmaSuite Technical Manual Configuration & Extension - Volume 4: Configuration Keys of PharmaSuite	PSCEV4-GR008E-EN-E
UC3	■ Changing or Adding New Labels	PharmaSuite Technical Manual Configuration & Extension - Volume 2: Changing or Adding New Labels	PSCEV2-GR008E-EN-E
UC4	■ User and Application object	PharmaSuite Technical Manual Configuration & Extension - Volume 4: Managing Configurations	PSCEV4-GR008E-EN-E

TIP

To access the Rockwell Automation Download Site, you need to acquire a user account from Rockwell Automation Sales or Support.

Configuration Keys Specific to the D Tare Phase

This section describes the configuration keys specific to the **D Tare** phase.

Phase/WDTare/allowOverrideReferenceTare

- **Type**: Boolean
- **Value**: False
- **Description**: The configuration applies to the **Failed tare check** system-triggered exception of the **D Tare** phase.
  If the value is set to **true** and the check fails, the operator first has to sign the exception then the phase overwrites the existing reference tare in the container's property of the **Reference Tare (RS)** purpose with the actual tare and updates the logbook accordingly (if maintained).
- **Evaluated**: When the **D Tare** phase is started in the Production Execution Client.
- **Range**: [False, True]

Configuration Keys Specific to the O Tare Phase

This section describes the configuration keys specific to the **O Tare** phase.

Phase/OWTare/allowOverrideReferenceTare

- **Type**: Boolean
- **Value**: False
- **Description**: The configuration applies to the **Failed tare check** system-triggered exception of the **O Tare** phase.
  If the value is set to **true** and the check fails, the operator first has to sign the exception then the phase overwrites the existing reference tare in the container's property of the **Reference Tare (RS)** purpose with the actual tare and updates the logbook accordingly (if maintained).
- **Evaluated**: When the **O Tare** phase is started in the Production Execution Client.
- **Range**: [False, True]

Chapter 4

Supported Scale Modes

This chapter applies to the **D Tare**, **D Weigh**, **D Release scale**, **O Tare**, **O Weigh**, **O Release scale**, and **Get weight** phases. The phases support the following scale modes:

- **Online**
  In the **Online** mode, the scale is physically connected to a PharmaSuite weighing work center. A scale driver is responsible for the communication between PharmaSuite and the scale. This is the standard use case.
- **Manual**
  In the **Manual** mode, the scale is not connected to a PharmaSuite weighing work center. The phases display input boxes whenever a scale value is needed. It is assumed that the operator manually zeros and tares the scale and types the readings of a real scale including its unit of measure. For this mode, the phases are equipped with a pre-defined phase completion signature based on the **WD_ES_MANUAL_SCALE** access privilege. This phase completion signature replaces any other phase completion signature configured in Recipe and Workflow Designer for the phases.
  All exceptions related to manual input and offline scale are disabled for the phases.
  The scale mode is defined in the scale configuration data of the scale equipment entity in Data Manager - Equipment.

Chapter 5

Adapting Phases for Dispense

This chapter applies to the **D Identify material**, **Show GHS data**, **D Select scale**, **D Tare**, **D Weigh**, **D Release scale**, and **D Print report** phases. You will learn how to adapt the phases for Dispense. The phases are executed within a Dispense operation in the Production Execution Client for EBR.
For the **D Select scale** phase, see also section "Access to ScaleEquipment and Scales Objects" (page 56).

Like other EBR operations, the Dispense operation is a structure built of phases. Therefore, both can be adapted in the same way as other EBR building blocks. However, the phases of the Dispense operation have some additional capabilities. The main purpose of the additional capabilities is to provide a smooth integration with the Process Model of FactoryTalk® ProductionCentre.

A Dispense operation describes the course of a dispense scenario. The operation comprises phase building blocks and transitions between them. The behavior of the phases can be affected by exceptions and the used weighing methods.
The result of a Dispense operation is documented in the Dispensing Report and its sub-reports.

For more information on Dispense and Inline Weighing, see also "Functional Requirement Specification Dispense and Inline Weighing" [A8] (page 63) and "User Manual Dispense and Weighing Phases" [A9] (page 63).

Structure of Dispense Operations

The basic structure of a Dispense operation is a straight sequence of several phases in a loop in order to process all of the material inputs that are available for processing.

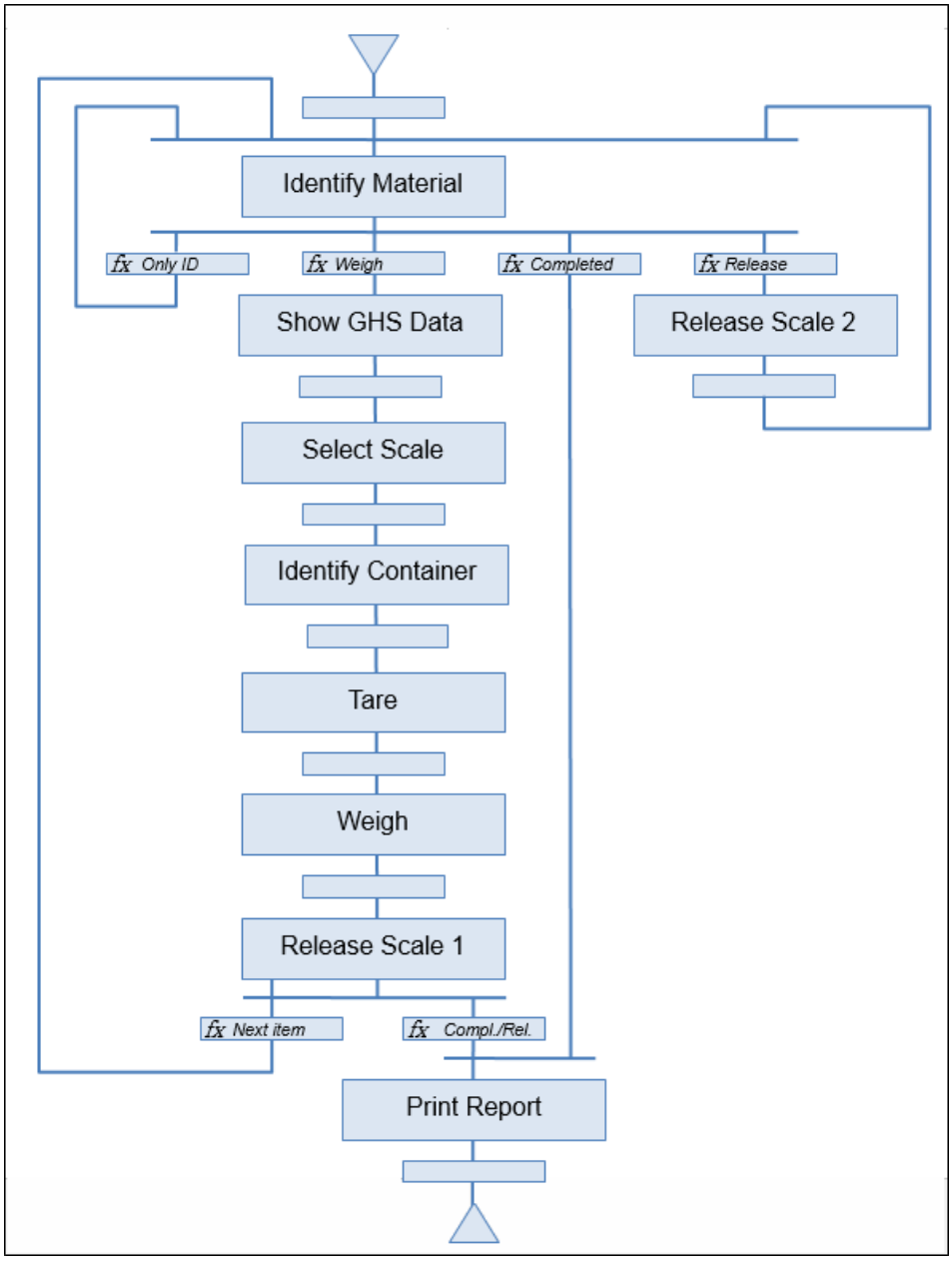

Figure 1: Example structure of a Dispense operation

Usually, the business logic is implemented within the phases.

The usage of the **O Identify container** and **Show GHS data** phases during Dispense is optional.
In order to support displaying the GHS data of a material, the **Show GHS data** phase needs to be added after the **Identify material** phase and after the branching (e.g. prior to

the **Select scale** phase and after the **Weigh** transition or after the **Only ID** transition). The **Show GHS data** phase can also be added right before the **Weigh** phase.

The Dispense phases implement a common system-triggered exception that is used to document an unforeseen resume of a Dispense or Inline Weighing operation. This applies to all phases, except the **D Print report**, **O Manage produce material**, and **O Select scale** phases.

Weighing Methods

When you make use of weighing methods, the structure becomes more complicated.

- Net weighing and Net removal weighing
  In **Net weighing**, first the tare weight of the target vessel is weighed. Then, the input material is filled into the target container and weighed.
  In **Net removal weighing**, first the source container is placed on the scale. Then, material is removed to the target container and the remaining material in the source container is weighed. Since material was removed, the weight is displayed as a negative value.
  The following scenarios may occur:
  - **Underweight** condition in **Net weighing**
    The target container is placed on the scale. If the source container is empty, the target container is kept and remains on the scale. So, there is no need to release, re-select, and re-tare the scale. During cycling the loop, the corresponding phases are skipped. They are not visible on the user interface and are completed automatically. Only the **D Identify material** and **D Weigh** phases are processed.
  - **Underweight** condition in **Net removal weighing**
    The source container is placed on the scale. If the target container is full, the source container remains on the scale and is kept there for further processing. Also in this situation, there is no need to release, re-select, and re-tare the scale. So, the corresponding phases will be skipped. They are not visible on the user interface and are completed automatically. Only the **D Identify material** and **D Weigh** phases are processed.
  - **Source batch changed** in **Net weighing** and **Net removal weighing**
    In the **Underweight** condition of the **D Weigh** phase, the *keep target* option was selected. In the subsequent **D Identify material** phase, material of another source batch is scanned. Then, in **Net weighing** and in **Net removal weighing**, the target has to be finished and a label will be printed. In **Net weighing** the scale needs to be released; this is realized by a connection and a corresponding transition from the **D Identify material** phase to the **D Release scale** phase.

- **Source sublot changed** in **Net removal weighing**
  In the **Underweight** condition of the **D Weigh** phase, the *keep source* option was selected. In the subsequent **D Identify material** phase, a sublot different from the sublot on the scale is scanned. Then, the scale needs to be released; this is achieved by a connection and a corresponding transition from the **D Identify material** phase to the **D Release scale** phase.

Technical details:

- There are no helper classes specific to **Net weighing** and **Net removal weighing**.
- The sublot label design is named MESOrderRelatedSublotLabelRD.

- Gross weighing and Gross removal weighing
  In **Gross weighing**, first the filled source vessel is placed on the scale. Then, the tare of the source vessel is entered manually and the source vessel is weighed.
  In **Gross removal weighing**, first the source container is placed on the scale. Then, the tare of the source container is entered manually, material is removed to another container, and the remaining material in the source container is weighed.
  Both weighing methods are implemented as a single weighing method. The operator decides based on the current situation whether a removal takes place or not.
  The business logic is implemented in the **D Tare** phase. Additionally, the **D Weigh** phase detects a removal of material and sets an output accordingly. This output is not used by PharmaSuite but is available for extension purposes.
  The following scenarios may occur:

  - **Overweight** condition
    The operator removes material from the container, thus using **Gross removal weighing**. If **Gross removal weighing** is not allowed for the current material, the operator may return to the **D Identify material** phase. The remaining quantity needs to be processed with a reasonable weighing method.
  - **Underweight** condition
    The operator finishes the current item by creating a split position related to the target quantity and material.

  Technical details:

  - The aspects of **Gross weighing** and **Gross removal weighing** of the **D Tare** phase are encapsulated in the *GrossWeighingMethodWDTare* helper class.
  - The sublot label design is named MESOrderRelatedSublotLabelRD.

- Pallet weighing
  In **Pallet weighing**, first the loaded pallet is placed on the scale. Then, the tare of the pallet and of one of the vessels it holds are entered manually, along with the number of vessels. Finally, the loaded pallet is weighed.

The weighing method has an impact on the **D Tare** and **D Weigh** phases, the Navigator actions, and the sub-report:

- In the **D Tare** phase, the total tare is calculated from the tare of the pallet, the tare of one of the containers, and the number of the containers. Then, the total tare is sent to the scale in order to determine the pallet's net weight.
- At the completion of the **D Weigh** phase, a split position is created for each container after the first. Target sublot labels are created for each of the positions (containers). The labels contain additional pallet-related information compared to the default target sublot label.
- The **Reprint label** post-completion exception in the Navigator allows the reprint of any of the target sublot labels.
- The **Replace target sublot** post-completion exception in the Navigator allows to replace one or multiple sublot(s).
- The sub-report of the **D Tare** phase contains information about the tare calculation and the sub-report of the **D Weigh** phase contains the list of the target sublots.

The following scenarios may occur:

- **Overweight** condition
  The operator has to remove a container from the pallet, thus requiring a recalculation of the total tare. If a container were removed during the **D Weigh** phase, the operator would have to return to the **D Identify material** phase and repeat the previous steps. To avoid this overhead, the overweight condition is already checked in the **D Tare** phase.
- **Underweight** condition
  The operator closes the target and continues with a new source.

Technical details:

- All aspects of **Pallet weighing** are encapsulated in the *PalletWeighingMethodWDTare* and *PalletWeighingMethodWDWeigh* helper classes.
- The sublot label design is named MESOrderRelatedSublotLabelPalletRD.
- The number of containers and the container tare are passed from the **D Tare** phase to the **D Weigh** phase by means of the operation context.

- Quantity entry
  In **Quantity entry**, no physical scale is used at all. The quantity provided by external means has to be entered manually. The **D Tare** and **D Release scale** phases are skipped in this weighing method. For this weighing method, the **D Weigh** phase is equipped with a pre-defined phase completion signature based on the **WD_ES_QUANTITY_ENTRY_D** access privilege. This phase completion

signature replaces any other phase completion signature configured in Recipe and Workflow Designer for the phase.
All exceptions related to manual input and offline scale are disabled for the **D Weigh** phase. The following scenarios may occur:

- **Underweight** condition in **Quantity entry**
  If the source container is empty, the target container is kept and remains in use. So, it is not allowed to change the weighing method or to select a scale, respectively. During cycling the loop, the corresponding phases are skipped. They are not visible on the user interface and are completed automatically. Only the **D Identify material** and **D Weigh** phases are processed.
- **Source batch changed** in **Quantity entry**
  In the **Underweight** condition of the **D Weigh** phase, the *keep target* option was selected. In the subsequent **D Identify material** phase, material of another source batch is scanned. Then the target has to be finished and a label will be printed. In **Quantity entry** the scale does not need to be released, therefore in the corresponding transition from the **D Identify material** phase to the **D Release scale** phase the **D Release scale** phase is skipped.

Technical details:

- There are no helper classes specific to **Quantity entry**.
- The sublot label design is named MESOrderRelatedSublotLabelRD.

- Only identification
  In **Only identification**, no weighing takes place at all. Sublots that hold a pre-defined material quantity are identified and their known weight is recorded. That means the quantity of the identified source sublot is recorded as quantity of the target sublot.
  The following scenarios may occur:
  - **In tolerance** condition
    The quantity of the identified sublot is within the tolerance band of the nominal quantity. Then, the position is finished, **Only identification** is recorded as weighing method, a new target sublot with the quantity of the identified source sublot is created, and the target sublot label is printed. Finally, **D Identify material** is active again.
  - **Underweight** condition
    The quantity of the identified sublot is below the tolerance band of the nominal quantity. Then, the position is finished, **Only identification** is recorded as weighing method, a new target sublot with the quantity of the identified source sublot is created, and the target sublot label is printed. Additionally, a split position with the remaining quantity is created. Finally, **D Identify material** is active again.

- **Overweight** condition
  The quantity of the identified sublot is above the tolerance band of the nominal quantity. Then, the operator can identify another sublot or continue with the identified sublot. In the latter case, the default weighing method and the **D Select scale** phase becomes active. Now, the operator must select an appropriate weighing method.

Technical details:

- All aspects of **Only identification** are encapsulated in the *IdOnlyWeighingMethodWDMatIdent* helper class.
- The sublot label design is named MESOrderRelatedSublotLabelRD.
- When a sublot has successfully been identified with **Underweight** or **In tolerance**, then a new target sublot is created. The quantity of the source sublot is transferred to the new target sublot and is set to 0 afterwards.

Technical Details of Dispense Phases

When developing phases to be used within a Dispense, Inline Weighing, or Output Weighing operation, you should be familiar with the following technical details:

- Model-View-Controller concept (page 23)
- Context and phase data (page 25)
- Material identification (page 26)
- Helper methods (page 30)
- Exceptions (page 31)
- Navigator actions (page 33)
- Dispensing report and sub-reports (page 33)
- Tips and Tricks (page 33)

The Model-View-Controller Concept of the Dispense Phases

The phases of a Dispense, Inline Weighing, or Output Weighing operation apply the Model-View-Controller pattern (MVC) for better structuring of the code. The model holds the data or knows how to fetch it, respectively. The view only knows the model and how to display it. The controller executes the business logic. Events control the communication between the three areas.

When developing phases to be used within a Dispense, Inline Weighing, or Output Weighing operation, you should reuse the MVC pattern, except for simple phases. The MVC pattern is supported by appropriate base classes.
The *AbstractWeigh** base classes are not used for the **Show GHS data** phase in order to

enable the usage of the **Show GHS data** phase outside of a Dispense, Inline Weighing, or Output Weighing operation.

The *AbstractWeighPhaseExecutor* class is the base class for the controller.

There are three types of weighing-related view events: property change events, barcode scanned events, and the phase confirmed event (see *IWeighViewEventListener* class). The *AbstractWeighPhaseExecutor* is a view event listener and registers itself at the view. The phase executor may add business logic by overriding the event handler methods. Some standard logic is already implemented: the **Confirm** button to complete a phase and the completion of a phase by scanning the barcode of the current scale. Both are implemented in the *AbstractWeighPhaseExecutor* class. You must override the following significant methods:

- *performPhaseCompletionCheck()*: verifies the completion conditions of the phase without saving any data,
- *performPhaseCompletion()*: performs the actual modifications and saves the data,
- *start()*: is called after the UI has been created (in the **Active** mode only). The method is usually used to perform the fall through or initialization tasks in the existing UI.

The *AbstractWeighView* class registers itself as a property change listener at the model. This enables the view to react to model changes. The model is passed on to the class in the constructor. *createUI()* is the most significant method of this class. The method creates the controls and adds them to view itself.

The *exceptionTransactionCallback()* method is used to modify phase data within the transaction that confirms an exception. After this method has been returned, the framework saves both the *RtOperationContext* (page 25) and the phase data.

The *AbstractWeighPhaseView* class implements the support of the **Confirm** button. Thus, the **Confirm** button provided by the subclass will be configured to fire a phase confirm event when the button has been clicked.

The *AbstractWeighModel* class represents the model. The properties, which represent the status of the weighing process of the current execution and which are needed by multiple phases are stored in the *RtOperationContext*.

The *AbstractWeighModel* class provides common properties of all phases. The subclasses of the model may add properties specific to a phase.

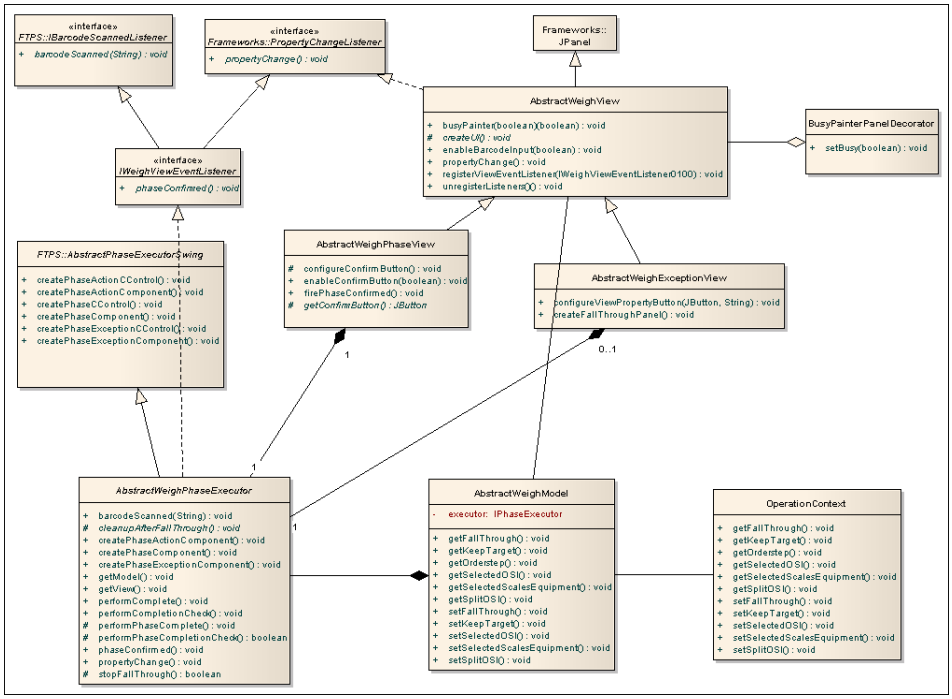

Figure 2: MVC class diagram

Context and Phase Data

The *RtOperationContext* is a singleton per runtime operation and its properties are available for all phases in the **Active** status (except the *Orderstep*). If a phase in the **Completed** status needs properties, it has to store them in the *RtPhaseData* of the phase.

The operation context is fully maintained by the PharmaSuite platform. The *RtOperationContextManager* class creates, saves, loads, and eventually deletes the operation context. The user-visible operation context is exposed through the *RtOperationContext* class.

The *RtOperationContext* class is exposed to the phase developer. The class contains a getter and a setter for named properties.

Within the Dispense phases, the operation context is encapsulated within a specific *OperationContext* class for easier use.

Using the Operation Context

A phase can be in the **Preview**, **Active**, or **Completed** status. The operation context is available only for phases in the **Active** status. The only exception is the *orderStep* property, because it is valid for the complete lifetime of the operation. This property can be retrieved for phases in the **Completed** status too. If this rule is violated, an *IllegalStateException* is thrown.

The content of the operation context is always a snapshot of the operation during processing. During phase completion and in exception transactions, PharmaSuite

automatically saves the operation context. In addition, you can use the *OperationContext.save(IPhaseExecutor)* method to save the operation context at any time. This might be useful to store a value locally to a phase instance to survive a shutdown during phase lifetime. If this is done along with other middle tier calls, be aware that you might need to run the method within a user transaction context.

In case the operation context is used to store a value locally with a phase instance, it might be a good idea to remove this property when completing the phase. This helps to keep the context clean. You can remove a property at any given time with the *removePropertyFromContext()*. *hasProperty()* allows to check for the existence of a property. In addition, you mark context properties as to be removed automatically at phase completion. Then, the Dispense framework will perform the housekeeping tasks automatically.

Usually, the properties provided by the operation context are part of the phase model. Therefore, we recommend to hide context properties inside the related model classes. The model class is also the place where other features like property change support should be implemented if needed.

On the other side, not all model data necessarily has to go into the context. For instance, data that must be evaluated each time an operation starts or resumes (e.g. an "initialized done" flag) is part of the model, but is not required in the context.

By default, each runtime operation starts with a new/empty context.
In some rare case, you may wish to keep an operation context for the next runtime operation instance within a runtime unit procedure. An example would be Inline Weighing with a Net removal loop where the target is charged while the source sublot is kept. In this situation, you need to explicitly set the *context.setPopulateContextIntoNextRtOperationInstance(true)* populate flag.
This flag is not allowed in a Dispense operation, but only for Inline Weighing.

Material Identification

Material identification takes place in the **D Identify material** phase when a sublot or a batch label is scanned.

Internally, PharmaSuite operates with sublots in both cases. However, for identification on batch level, the sublot is marked as temporary. Temporary sublots are automatically removed after weighing (i.e. they will be consumed at target sublot creation). They are not visible on the user interface.

The following facts are essential for material identification:

- In order to uniquely identify a sublot, both sublot identifier and batch identifier are required. Whether batches are unique is controlled by the **LibraryHolder/services-inventory-impl.jar/AreBatchIdentifiersUnique** configuration key. The default setting is **true**. In this case, the batches are supposed to be unique per batch identifier. If the key is set to **false**, then batches are made unique by adding the part identifier to the batch identifier.

For more information, please refer to chapter "Configuration Keys of PharmaSuite" in Volume 4 of the "Technical Manual Configuration and Extension" [A5] (page 63).

- If the identification is performed by scanning, the system first tries to perform the identification on **sublot level**. Only if the scanned barcode does not match the barcode configuration for sublots, tries the system to perform the identification on **batch level**.
- Processing of multiple barcodes is supported for identification on **batch level** only.

The classes and methods involved in the identification process are spread out over the FTPS package (page 27) and the Dispense Building Block package (page 27).

Classes of the FTPS Package

The *IMFCService.getSublotFromSublotOrBatchBarcode* method retrieves a sublot. The method receives a list of barcodes as input. It always returns a sublot; in case of identification on batch level, the sublot is temporary. If no sublot could be retrieved based on the given input, an *MESInvalidBarcodeException* or an *MESMissingBarcodeDataException* exception is thrown.

The *MESInvalidBarcodeException* class has a *getReason()* public method, which returns a *Reason* enumeration with the cause of the exception.
The *MESMissingBarcodeDataException* class has a *getMissingData()* public method, which returns a *MissingData* enumeration with the information what data is missing.
Both classes contain a *getLocalizedMessage()* public method, which returns a localized message for the associated *Reason* or *MissingData* enumeration. The localized messages are stored in the **clientfw_pec.BarcodeSupport** message pack.
The messages used by the *MESInvalidBarcodeException* class use a place holder ({0}), either with the single barcode string or a comma-separated list of barcodes. Message identifiers that exist in singular and plural form are suffixed with _1 or _n, respectively.

The *ISublotParser* class retrieves the identifiers of sublot, batch, and part from the scanned barcodes against given barcode templates. As this class is instantiated via *ServiceFactory.newMESObject()*, you can provide your own implementation to support more sophisticated parsing.

Classes of the Dispense Building Block Package

The *RtPhaseExecutorWDMatIdent* class is the phase executor of the **D Identify material** phase. Identification via scanning is performed within the *barcodeScanned()* listener method while *displayManualIdentificationException()* is responsible for the manual identification within the exception view.
Both methods delegate their processing to a single method, *performIdentification()*. The reason behind this is that from a business point of view there is no difference of how the

identification was performed physically. All of the checks and the service layer functionality should be performed the same way.

The *IdentificationMode* enumeration distinguishes between manual and scanned identification. It encapsulates some detailed aspects that differ between the two identification modes, like risk class or check key for exception recording. The most important method is *fetchIdentifiedSublot()*.
In case of identification by scanner, *fetchIdentifiedSublot()* delegates the data to the *BarcodeProcessor* class, while in case of manual identification the *ManualIdentificationHelper.getSublotByManualIdentificationModel()* helper class method is used.

The *BarcodeProcessor* class is responsible to process scanned barcodes. Each time a barcode is scanned, the class adds the current barcode to a list of barcodes and feeds this list to *IMFCService.getSublotFromSublotOrBatchBarcode()*. Based on the result or an exception, either a sublot or null is returned. In case of an exception, the localized messages are displayed. The internally kept list of seen barcodes is cleared if

- a sublot is returned,
- more than two barcodes are seen, or
- the system recognizes that the operator has scanned something wrong.

BarcodeProcessor deals also with another detail. Since the barcode listener sees all scanned barcodes, it also sees the scans of the 2D barcodes from the buttons on the user interface. Such a 2D barcode leads to an *MESInvalidBarcodeException*, since it does not match the expected sublot or batch format. If the exception is caught and the barcode happens to match the 2D barcodes for buttons, the error dialog is suppressed.

The *ManualIdentificationModel* class is a read-only view of the fields of the user interface within the manual identification exception view. The class provides access to sublot, batch, and part identifiers.

The *ManualIdentificationHelper* class is a collection of static helper methods related to manual identification. It operates similar to *IMFCService* with the exception that it does not operate with barcodes but sublot, batch, and part identifiers, retrieved from *ManualIdentificationModel* instead an *ISublotParser* instance.

Example: If you wish to get rid of the restriction of single barcode recognition for sublots, consider to implement your own *getSublotFromSublotOrBatchBarcode*. Additionally, if you wish to use it only within the context of Dispense, you could implement your own algorithm. The *ManualIdentificationHelper* class provides some ideas how to deal with sublot, batch, and part identifiers. In particular, you can see when you would use which piece of information.

Configuring Barcode Templates

From an operator's point of view there is no difference between identification on sublot or on batch level. Both identification methods expect a scanned identifier. However, PharmaSuite has to distinguish between both methods.

The barcodes for sublot and batch labels must be designed in a way that enables PharmaSuite to distinguish between them. The barcode formats can be configured with the following configuration keys.

- Sublot identifier
  LibraryHolder/services-inventory-impl.jar/DefaultSublotBarcodeTemplate
  LibraryHolder/services-inventory-impl.jar/SublotBarcodeTemplates
- Batch identifier
  LibraryHolder/services-inventory-impl.jar/DefaultBatchBarcodeTemplate
  LibraryHolder/services-inventory-impl.jar/BatchBarcodeTemplates

For more information, please refer to chapter "Configuration Keys of PharmaSuite" in Volume 4 of the "Technical Manual Configuration and Extension" [A5] (page 63).
The default templates (DefaultSublotBarcodeTemplate, DefaultBatchBarcodeTemplate) are used if either the associated template configuration does not exist or the template lists are empty. The latter is the default configuration.

Additionally, the default templates are also used to create the string representation for sublot or batch barcodes using the *ISublotBarcodeParser* methods. This has an impact on the configuration.

> **TIP**
>
> When you configure the templates, you should also add the associated default template to the list. Otherwise, it may happen that you can create barcodes and print labels but you cannot scan them.

For more information about barcode patterns, please refer to chapter "Changing the Number Generation Schemes" in Volume 2 of the "Technical Manual Configuration and Extension" [A3] (page 63).

Even though the templates for sublot and batch barcodes lists are empty, the lists are needed for the following reasons:

- To support different label formats for identification on sublot or batch level.
  Example: For batch identifiers with five and eight digits, you have to configure the following sublot templates:
  $sssssssssbbbbb, $sssssssssbbbbbbbbb
- Some labels may require multiple barcode scans to uniquely identify a sublot or batch, respectively. The **D Identify material** phase supports a sequence of two scans.

If your system is configured for non-unique batches (i.e. a batch is unique only with a material number), your barcode template for the identification on batch level may look as follows:

- bbbbbbbbb, pppppppppppppppp

This example shows that it is not mandatory to use a prefix to characterize the type of a barcode. Based on the different length of the barcodes, PharmaSuite can associate the scans accordingly.

Example: When you scan a 10-character barcode, the system associates the scan with the batch information of a batch label and requests a second scan, since batches are not unique without material information. When you scan an 18-character barcode afterwards, this will be recognized as the part information of the batch label. Then, the system can retrieve a unique batch based on the scanned data. Additionally, a temporary sublot will be created for the batch and exposed to the phase.
It is also possible, to scan the material information first and then the batch information.

Container Integration for Dispense

Dispense can handle source and target containers.

After a successful identification of a sublot, *RtPhaseExecutorWDMatIdent.handleSourceContainer()* checks if the sublot is assigned to a container. If so, the container is bound and fed into the *currentSourceContainer* property of the *OperationContext*. The source container is released after weighing. *IdOnlyWeighingMethodWDMatIdent.handleContainer()* takes care of containers for the **Only identification** weighing method.

If a Dispense operation contains an **O Identify container** phase, an identified container is bound and fed into the *currentTargetContainer* property of the *OperationContext*. In the weighing step, the created target sublot is assigned to the target container. If the new target sublot is closed, the target container is released. In case of a keep target situation, the container remains bound. This behavior is implemented in *RtPhaseExecutorWDWeigh.handleContainers()*.
An identified target container can hold an optional property for a reference tare. This tare value can be checked and modified by the **D Tare** phase.

Helper Methods

The following helper methods are available:

- barcode support and
- busy painter.

Barcode Support

The *AbstractWeighPhaseExecutor* controller base class provides barcode support. The default implementation of *barcodeScanned(String barcode)* completes the phase when

the barcode of the current scale is scanned. If you wish a different behavior, you may override this method.

You may also disable the barcode support by overriding the *AbstractWeighView.isBarcodeSupportEnabled()* method.

BUSY PAINTER

The busy painter provides a visual feedback to the operator when the phase executes business logic. The busy painter is automatically applied during barcode scans and on phase completion with the default **Confirm** button. You may apply the busy painter manually by calling *busyPainter(true)* on the view.

> **TIP**
>
> When you set the busy painter, you have to execute the code following the busy painter activation by using invoke later in order to have the busy painter rendered before the waiting period starts.

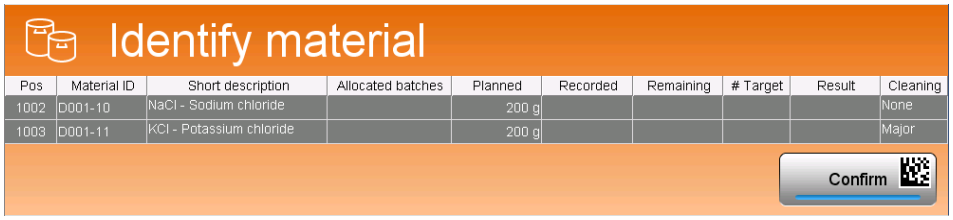

Figure 3: Busy painter

Exceptions

The *AbstractWeighExceptionView* class is the base class for exception views. It provides a listener that supports firing events to the controller in case an exception has been confirmed. For details, see *configureViewPropertyButton(...)* method.

By default, the base class provides the **fall-through** (**Return to material identification**) exception in its *createUI()* method. You have to override this method if you wish to add your own exceptions. If you wish to support fall-through and your own exception, you have to call *super.createUI()* first.

> **TIP**
>
> The **Show GHS data** phase supports fall-through without a specific user-triggered exception.

The handling of **unforeseen resume** exceptions is implemented in the *AbstractWeighPhaseExecutor*. These exceptions are system-triggered exceptions that do not need a specific UI.

FALL-THROUGH

Dispense is implemented as an application in order to avoid the explicit modeling of exceptional cases. Exceptional cases would include back steps. For common use cases, exceptions may be used to fix an issue without the need to undo former steps.

However, unexpected issues can occur that are not detectable by system checks that require the operator to rollback the current work.
If the operator detects such an issue, for example during the **D Tare** phase, he can submit a **fall-through** exception to return to the **D Identify material** phase.

Figure 4: Fall-through exception

The **fall-through** exception completes the current phase, skips the subsequent phases without further action, and starts the **D Identify material** phase. First, the **D Identify material** phase performs cleanup actions on the data model, especially related to the order step input (OSI). Then, the operator can proceed as usual.
An analogous behavior is available for Output Weighing phases with the **Return to material management** exception.

The concept requires that all phases are aware of the fall-through mode. Phases that fall through may render in a special way.

A phase can detect the fall-through mode via the *AbstractWeighModel.getFallThrough* model or via the context.

UNFORESEEN RESUME

When resuming a Dispense, Inline Weighing, or Output Weighing operation after a logout or a system crash, the work environment may be in another status than expected by the resumed operation. Such a situation is critical related to quality aspects; therefore the operator or supervisor has to decide how to proceed. PharmaSuite requests submitting a system-triggered exception, called **Unforeseen resume**, to document this incident. It is not possible to confirm the phase without submitting this exception. The exception is only recorded and has no other consequences. It serves as a hint for QA to review the affected operation in greater detail.

TIP

Please note that this exception can also be triggered by phases that become visible due to an unforeseen resume exception (e.g. **D Tare** phase with keep target). In case of resume, invisible phase become visible, request submitting the exception, and are completed automatically as before. Then, the completed view of the phase is displayed in the Execution Window and the phase is available in the Navigator in order to review and comment the exception.

The behavior of the phases varies slightly between fall-through and resume from a technical point of view. Especially *AbstractWeighPhaseExecutor.start()* will not be called until the exception has been signed off.

Dispense and Weighing phases need to implement the *AbstractWeighingPhaseExecutor.isScaleActive()* abstract method. This method needs to return true, if the phase actively uses a scale when the operation is interrupted.

Navigator Actions

The base class for Navigator actions is the *AbstractWeighActionView* class. You have to override the *createUI(int actionIndex)* method to render the action(s).

For more information, please refer to chapter "Adding Actions for Completed Phases" in the "Technical Manual Developing System Building Blocks" [A7] (page 63).

Dispensing Report and 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. This applies also to the Dispensing Report and its sub-reports.

The values to be displayed in sub-reports must be saved in the phase data of the phase in addition to the process model.
Each phase has a sub-report in which the layout is configured. The labels are filled by a Java scriptlet, e.g. *ReportScriptletWDMatIdent*. The values are retrieved from the data sources using expressions based on the JavaBean properties or directly using the *IBatchProductionRecordDocumentWrapper* helper methods.

For more information on adding phase-specific sub-reports, please refer to chapter "Adding Sub-reports and Phase-specific Sub-reports" in the "Technical Manual Developing System Building Blocks" [A7] (page 63).
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" [A3] (page 63).

Tips and Tricks

In this section, you will find some useful hints how to avoid pitfalls when working with Dispense phases:

- Limitations in the **Preview** status
  Usually, in the **Preview** status, phases do not access the *OperationContext*. Most objects within the *OperationContext* will be in an unknown status. We recommend, not to rely on it.
  Furthermore, there is no runtime phase and hence no runtime phase data.

- Limitations in the **Completed** status
  Objects in the *OperationContext* may be in an unknown status. We recommend, not to use them. Phase-related data may be available as runtime phase data and should be taken from there.
  Getter methods in the model components of a phase should encapsulate the required behavior and access the data from the correct source depending on phase status.
- Dispense phases need to support a fall-through mode.
- Dispense is highly stateful. When you add new functions or change existing behavior, make sure you have thoroughly investigated all effects and potential side effects.
- Always keep in mind that a Dispense phase must be able to run in slightly different contexts with a slightly different behavior depending on circumstances like weighing method, keep target situation, keep source situation, weighing material type. Make sure not to the break the phase's initial functionality when you modify it. We recommend to extensively test the new functionality.
- Do not store references to dependent objects like **OrderStepInput** or **OrderStepOutput** as fields within a class, since dependent objects only exist in the context of their parent object. Sometimes it is necessary to update the selected **OrderStep** object within the context of a Dispense or Inline Weighing operation. In that case, you get new instances of the dependent objects and any reference you hold may point to an outdated instance.
  If you really need to remember dependent objects as fields in your class, store the key or the GUID and fetch it on demand from the model.
  Be careful when you store references as local variables within the scope of a method. However, if somewhere within the method the selected **OrderStep** object is refreshed, you may point to outdated information again.
- Do not refresh the selected **OrderStep** object if it is not necessary in order to avoid a performance-related impact.

  TIP
  If there is a newer instance of the selected **OrderStep** object available than the one stored in the operation context, it is not necessary to refresh the selected **OrderStep** object. Instead, you can update the selected **OrderStep** object with the new instance. We assume this to be much faster.

Extension Use Cases

For more information about adapting Dispense phases, please refer to the following use case descriptions:

- Adapt an existing phase (page 35)

- Add a new phase (page 35)
- Extend the context (page 36)

Adapting an Existing Phase

If you wish to adapt an existing phase, we recommend to copy the folder structure and to rename all of the artifacts appropriately and according to the naming recommendations (page 3).

Additionally, refer to chapter "Extending Product Phases and Parameter Classes" in "Technical Manual Developing System Building Blocks" [A7] (page 63).

Adding a New Phase

Before you can add a new phase to a Dispense, Inline Weighing, or Output Weighing operation, you must create the phase by means of the phase generator. For more information, please refer to chapter "Creating a Phase Building Block" in the "Technical Manual Developing System Building Blocks" [A7] (page 63).

If you wish to use the MVC concept (page 23), then you have to change the base class of the phase executor from *AbstractPhaseExecutorSwing* to *AbstractWeighPhaseExecutor*. The *AbstractWeighPhaseExecutor* phase executor needs the three view classes and model classes as parameterization. You have to create the classes by extending the appropriate base classes.
For details, see chapter "Technical Details of Dispense Phases" (page 23). For reasons of simplification, we recommend to start with the *RtPhaseExecutorWDReleaseScale* phase executor, its view, and model.

If you do not use the MVC concept, you have to ensure that the new phase can deal with the fall-through flag. This means that the phase must automatically complete in case the flag is set and the view becomes invisible.

TIP

Please keep in mind that in case of fall-through, the context and the process model may not contain data that is available during normal processing.

Integrating a Phase into the Structure

In general, you can add new phases at any position in the operation.

When you reference outputs from other phases, make sure, the phases have been processed before your phase. Otherwise the outputs are not available.

Do not modify data with your phase that is expected by succeeding phases.

Extending the Context

The *OperationContext* class is a shared class of the Dispense and Inline Weighing phases. For Output Weighing, a similar class is available: *OWOperationContext*. Both classes provide typed access to individual properties maintained by the context. For each property, a getter (page 37) and setter (page 38) method exists. Only the sets of properties of the *OperationContext* and *OWOperationContext* classes are different.

All methods of the context are static, therefore inheritance is not supported. If you wish to add properties to the context, we recommend to modify the existing *OperationContext* or *OWOperationContext* classes.

Getter and setter methods receive an *IPhaseExecutor* argument in order to identify the operation-specific context.

The PharmaSuite building block development environment provides general getter and setter methods for *boolean* and *sublot* properties.

Defining a Context Property

The *OperationContext* and *OWOperationContext* classes contain a *Property* enumeration. *Property* contains all properties that are available within the context. The following code shows the definition part of this enumeration. It contains all currently used properties. At the end of the example we provide some space where you can add your own properties.

```
/** Contains all properties, declared by {@link OperationContext0100}. */
enum Property {
  /** The selected OrderStepInput. */
  SELECTED_OSI,
  /** The target sublot */
  TARGET_SUBLOT,
  /** The target sublot will be kept and not closed */
  KEEP_TARGET,
  /** The source sublot will be kept (relevant for net removal weighing) */
  KEEP_SOURCE,
  /** The selected {@link IMESScaleEquipment}. */
  SELECTED_SCALE_EQUIPMENT,
  /** The fall through flag to cancel a position. */
  FALL_THROUGH,
  /** The split OrderStepInput. */
  SPLIT_OSI,
  /**
   * The weight used for release scale. It is persistent, because the
   * weight must be read only once to prevent misuse by the operator.
   * Used by release scale phase only.
   */
  RELEASE_SCALE_WEIGHT(Feature.RemoveAfterPhaseCompletion),
  /**
   * <code>true</code>, if the last scales release has failed.
   * Default is <code>true</code>.
   */
  LAST_RELEASE_CHECK_FAILED,
  /**
   * The scale weight (saved in case that it has been entered manually by user.
   */
  SCALE_WEIGHT(Feature.RemoveAfterPhaseCompletion),
  /** Phase is in manual weight input mode. */
```

```
  MANUAL_WEIGHT_MODE(Feature.RemoveAfterPhaseCompletion),
  /** The use-by date the user has entered. */
  USE_BY_DATE(Feature.RemoveAfterPhaseCompletion),
  /**
   * The initial weight value (gross weight) of the source container in
   * case of removal weighing. The value is persistent to be able to check
   * scales restriction in later phases.
   */
  INITIAL_SOURCE_WEIGHT_REMOVAL_NET,
  /** The number of a containers on the pallet */
  NUMBER_OF_CONTAINERS_ON_PALLET,
  /** The tare of a container in case of pallet weighing */
  CONTAINER_ON_PALLET_TARE,
  /**
   * Persistent flag, indicates that the user triggered exception
   * "print label and release scale" was signed.
   * Used by material identification phase only.
   */
  PRINT_LABEL_RELEASE_SCALE(Feature.RemoveAfterPhaseCompletion);
  /** Add here new properties and write setter and getter for it. */
```

THE GETTER METHOD OF A PROPERTY

The main responsibility of the getter is to fetch a context property from the *RtOperationContextManager* class. In addition, the getter method must deserialize the context property from a raw property, which comes from the persistent storage, into the desired target property. The following example shows a getter for *sublot* properties.

```
/**
 * Getter for a Sublot property.
 * @param executor the {@link IPhaseExecutor} providing access to the context
 * @param contextProperty the {@link Property} of interest
 * @return the {@link Sublot} value represented by the property
 */
private static Sublot getSublotProperty(IPhaseExecutor executor,
               Property contextProperty) {
  RtOperationContext context = getContext(executor);
  String propertyName = contextProperty.toString();
  synchronized (context) {
    if (!context.hasProperty(propertyName)) {
      Object rawProperty = context.getRawProperty(propertyName);
      if (rawProperty != null) {
        Sublot property = ServiceFactory.getService(ISublotService.class)
                          .loadSublot(((Long) rawProperty).longValue());
        context.setProperty(propertyName, property);
      }
    }
    return context.getProperty(propertyName);
  }
}
```

The first code line fetches the *RtOperationContext* from the PharmaSuite platform. The string representation of the enumeration serves as property name.

If the desired property has been fetched before, it can simply be returned. Otherwise it must be translated from a raw property (here: *context.getRawProperty()*) into the real one. In the example of fetching sublots, the raw property is just the key of the sublot. The sublot itself is one-time fetched by means of the sublot service.

The synchronized block ensures that only one phase executor changes the context at any given time.

The Setter Method of a Property

The responsibility of the setter is to write a property into the context. This implies a serialization of the property into the raw property. The following example shows a setter for *sublot* properties.

```
/**
 * Setter for a {@link Sublot} property.
 * @param executor the {@link IPhaseExecutor} providing access to the context
 * @param contextProperty the {@link Property} of interest
 * @param sublot the {@link Sublot} to set
 */
private static void setSublotProperty(IPhaseExecutor executor, Property contextProperty,
                                      Sublot sublot) {
  RtOperationContext context = getContext(executor);
  synchronized (context) {
    String propertyName = contextProperty.toString();
    context.setRawProperty(propertyName, sublot != null ? //
        Long.valueOf(sublot.getKey()) : null);
    context.setProperty(propertyName, sublot);
  }
}
```

The onPhaseComplete Method

The *onPhaseComplete* method is called at the end of the phase completion. It removes all properties marked with *Feature.RemoveAtPhaseCompletion* from the context. This is useful for properties, which exist during the lifetime of a single phase only. Obviously, properties marked with this feature cannot be used for inter-phase communication.

Chapter 6

Adapting Phases for Inline Weighing

This chapter applies to the **D Identify material**, **Show GHS data**, **D Select scale**, **D Tare**, **D Weigh**, and **D Release scale** phases. The implementation of the phases for Inline Weighing is almost identical to the implementation of the phases for Dispense. For details, see section "Adapting Phases for Dispense" (page 17).
This section describes the areas in which the phases for Inline Weighing differ from the phases for Dispense.

Structure of Inline Weighing Operations and Unit Procedure

A typical structure of an Inline Weighing unit procedure contains an Inline Weighing operation followed by one or more operations for charging the weighed materials into a vessel for further processing.

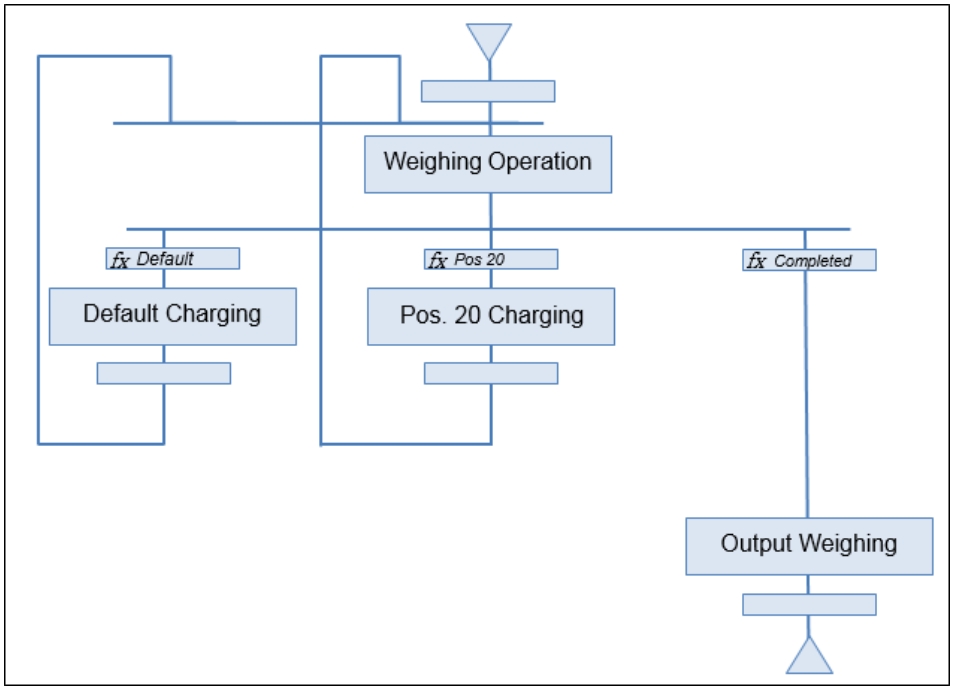

Figure 5: Example of a typical Inline Weighing unit procedure with a loop and transitions

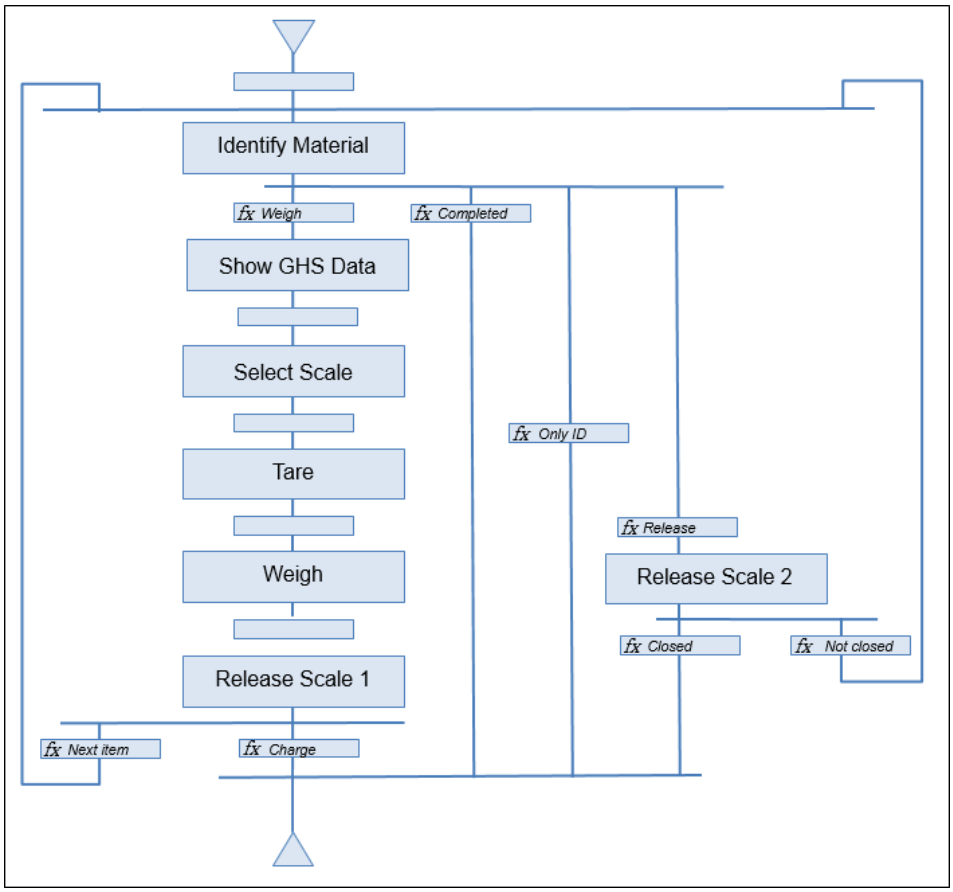

Figure 6: Typical Inline Weighing operation with loops and transitions

The usage of the **Show GHS data** phase during Inline Weighing is optional.
In order to support displaying the GHS data of a material, the **Show GHS data** phase needs to be added after the **Identify material** phase and after the branching (e.g. prior to the **Select scale** phase and after the **Weigh** transition or after the **Only ID** transition).
The **Show GHS data** phase can also be added right before the **Weigh** phase.

Weighing Methods

An Inline Weighing operation can be run with various weighing methods to cover different physical processing conditions, such as source sublot containers that need to be used in subsequent processing steps or source sublots with certified material quantities that only require identification.
The following weighing methods are available:

- Net weighing and Net removal weighing
  In **Net weighing**, first the tare weight of the charging vessel is weighed. Then, the input material is filled into the charging vessel and weighed.
  In **Net removal weighing**, first the source container is placed on the scale. Then, material is removed to the charging vessel and the remaining material in the

source container is weighed. Since material was removed, the weight is displayed as a negative value.

- Gross weighing and Gross removal weighing
  In **Gross weighing**, first the filled source vessel is placed on the scale. Then, the tare of the source vessel is entered manually and the source vessel is weighed.
  In **Gross removal weighing**, first the source container is placed on the scale. Then, the tare of the source container is entered manually, material is removed to another container, and the remaining material in the source container is weighed.
  There is no difference to Dispense.
- Quantity entry
  In **Quantity entry**, no physical scale is used at all. The quantity provided by external means has to be entered manually. The **D Tare** and **D Release scale** phases are skipped in this weighing method. For this weighing method, the **D Weigh** phase is equipped with a pre-defined phase completion signature based on the **WD_ES_QUANTITY_ENTRY_D** access privilege. This phase completion signature replaces any other phase completion signature configured in Recipe and Workflow Designer for the phase.
  All exceptions related to manual input and offline scale are disabled for the D Weigh phase.
- Only identification
  In **Only identification**, no weighing takes place at all. Sublots that hold a pre-defined material quantity are identified and their known weight is recorded.
  There is no difference to Dispense.

No Target Sublot Creation for Inline Weighing

For Inline Weighing, no target sublots are created. This implies that no target sublot labels are printed, the reprint function is not available, and target sublots cannot be replaced.

Container Integration for Inline Weighing

When a source sublot assigned to a container is identified, the container is bound. After weighing, the container is released again.

Due to the nature of Inline Weighing target containers are not supported.

For more details, see "Container Integration for Dispense" (page 30).

Technical Details Specific to Inline Weighing

The *WeighingOperationType* enumeration contains one element for Inline Weighing and another one for Dispense. The methods within the enumeration encapsulate the different handlings for Inline Weighing and Dispense.

The framework also supports phase-specific enumerations of the weighing operation type (e.g. *WeighingOperationTypeMatIdent*) to enable a phase-specific handling of a given phase.

Chapter 7

Adapting Phases for Output Weighing

This chapter applies to the **O Manage produced material**, **Show GHS data**, **O Select scale**, **O Identify container**, **O Tare**, **O Weigh**, and **O Release scale** phases. The implementation of the phases for Output Weighing resemble the implementation of the phases for Dispense and Inline Weighing. For details, see section "Adapting Phases for Dispense" (page 17). The Output Weighing phases use the same MVC concept and the same base classes as the Dispense and Inline Weighing phases. Additionally, the Output Weighing phases use the runtime operation context to persist life-cycle status information.
This section describes the areas in which the phases for Output Weighing differ from the phases for Dispense.

For more information on Output Weighing, see also "Functional Requirement Specification Output Weighing" [A10] (page 63) and "User Manual Dispense and Weighing Phases" [A9] (page 63).

Output Weighing Uses OrderStepOutput

Dispense and Inline Weighing operations work on input material (**OrderStepInput**) while Output Weighing operations work on output material (**OrderStepOutput**). This is the most important differentiator of Output Weighing and the main reason to create a separate set of phases for supporting the Output Weighing use cases.

Common business functionality for Dispense, Inline Weighing, and Output Weighing is available in *IWeighingService*. Business functionality for Output Weighing is bundled in *IOutputWeighingService*. Additional functionality related to **OrderStepOutput** is available in *IOrderStepOutputService*.

Structure of Output Weighing Operations (with Loop)

The basic structure of an Output Weighing operation is a straight sequence of several phases with a loop.

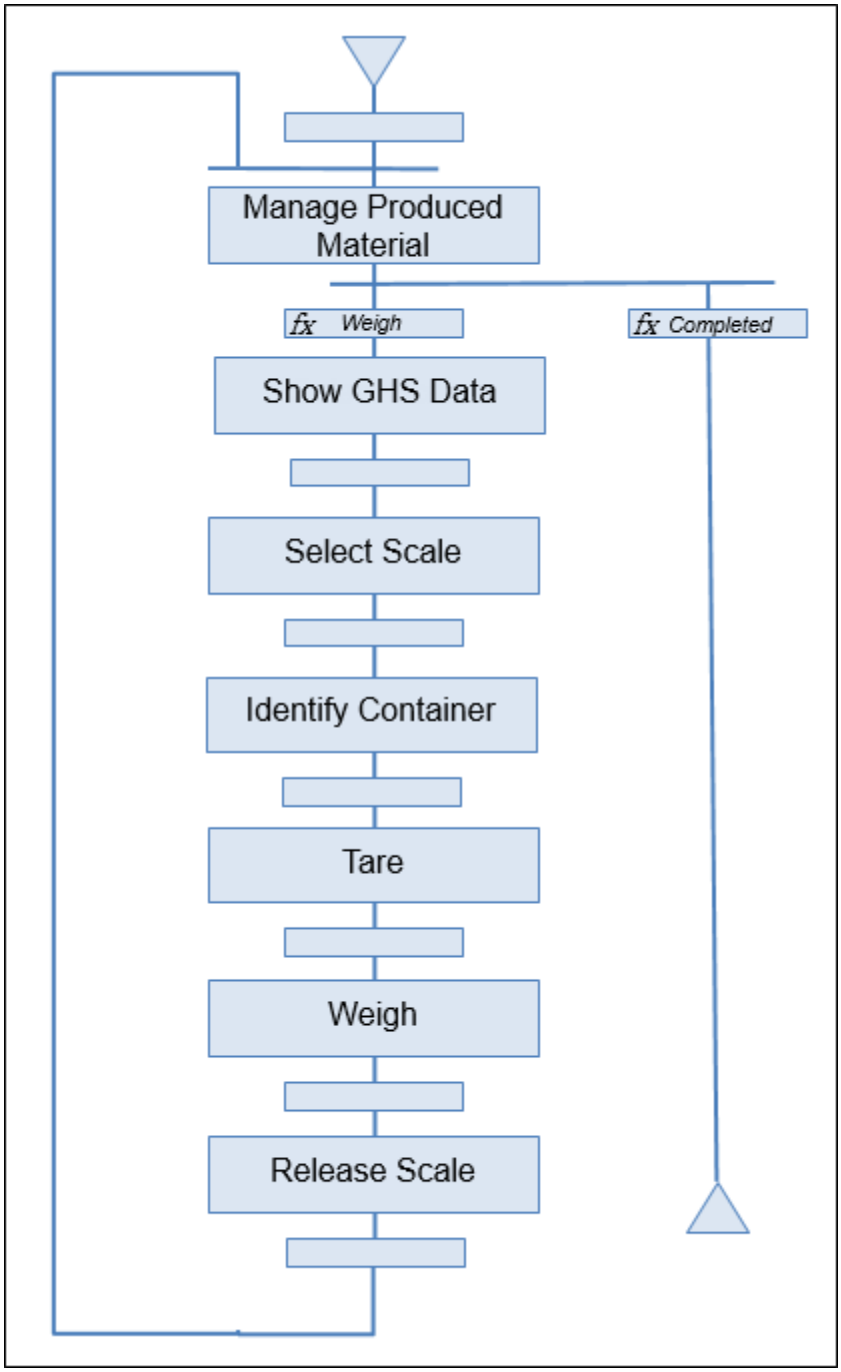

Figure 7: Example structure of an Output Weighing operation

The following characteristics apply to the Output Weighing phases:

- There is no weighing report for Output Weighing.

- Optional phases:
  - **O Release scale**
  - **O Identify container**
  - **Show GHS data**
- There are no material input parameters. The material to be weighed is defined as material output parameter of the **O Weigh** phase.
- The **O Manage produced material** phase controls the flow of the Output Weighing loop. Within each loop a single container/sublot of the produced material is tared and weighed. A sublot label for the container is printed upon completion of the **O Weigh** phase.

Weighing Methods

An Output Weighing operation can be run with one or more weighing methods to cover its physical processing conditions and usage scenario.
The following weighing methods are available:

- Net weighing
  In **Net weighing**, first the tare weight of the target vessel is weighed. Then, the produced material is filled into the target vessel and weighed.
  Net weighing is available for Output Weighing operations run in the **Prepare only** mode.
- Gross weighing
  In **Gross weighing**, first the filled source vessel is placed on the scale. Then, the tare of the source vessel is entered manually and the source vessel is weighed.
  Gross weighing is neither available for Output Weighing operations run in the **Prepare only** mode nor for sublot preparation in the **Flexible** operation mode. Identification of target containers is not available for **Gross weighing**.
- Pallet weighing
  In **Pallet weighing**, first the loaded pallet is placed on the scale. Then, the tare of the pallet and of one of the vessels it holds are entered manually, along with the number of vessels. Finally, the loaded pallet is weighed.
  Pallet weighing is neither available for Output Weighing operations run in the **Prepare only** mode nor for sublot preparation in the **Flexible** operation mode. Identification of target containers is not available for **Pallet weighing**.
- Quantity entry
  In **Quantity entry**, no physical scale is used at all. The quantity provided by external means has to be entered manually. The **O Tare** and **O Release scale** phases are skipped in this weighing method. For this weighing method, the **O Weigh** phase is equipped with a pre-defined phase completion signature based on

the **WD_ES_QUANTITY_ENTRY_O** access privilege. This phase completion signature replaces any other phase completion signature configured in Recipe and Workflow Designer for the phase.
All exceptions related to manual input and offline scale are disabled for the **O Weigh** phase.
Quantity entry is available for Output Weighing operations run in the **Prepare only** mode.

Different Operation Modes of Output Weighing

Output Weighing can be processed in **one step** or **two steps**. The different operation modes are controlled by the *OperationMode* enumeration. The operation mode is part of the runtime operation context. The following values are defined:

- *CONTINUOUS*: used for one-step Output Weighing
- *PREPARE*: used for two-step Output Weighing
- *WEIGH*: used for two-step Output Weighing

One-step Output Weighing

This is the simplest form of Output Weighing. Within each loop of the operation a new container/sublot with outgoing material is produced. In **Net** weighing, each container/sublot is tared, then weighed, and finally labeled. In the **Quantity entry** weighing method, taring is skipped. Other weighing methods like **Gross** or **Pallet** weighing are also supported.

Tolerances as known from Dispense and Inline Weighing are not used. However if a target weight with its tolerances has been defined for the material, the net weight is checked for each sublot that is created. In case the weight is outside the tolerances, a system-triggered exception is created.

Two-step Output Weighing

The process of producing a sublot of the output material is split in two steps:

- *PREPARE*: A new target container/sublot is created and tared. In the **Quantity entry** weighing method, taring is skipped. A label with zero net weight is printed, i.e. the container/sublot remains empty.
- *WEIGH*: The labeled and tared container/sublot is identified with the **O Manage produced material** phase.
  In case the container/sublot does not have a known tare since the **Quantity entry** weighing method was used in the *PREPARE* step, in the **O Select scale** phase, the **Quantity entry** weighing method is automatically selected and the phase is automatically completed. Additionally, the **O Tare** phase is skipped.
  Otherwise, if the container/sublot is already tared, in the **O Select scale** phase, the

Net and **Quantity entry** weighing methods are available for selection. In **Net** weighing, the phase sends the tare value to the scale (without taring). Afterwards the container/sublot is filled with the output material and can be weighed. If a target weight with its tolerances has been defined for the material, the net weight is checked against the target weight before a sublot is created. In case the weight is outside the tolerances, a system-triggered exception is created. A new label is printed.

Two-step Output Weighing allows only **Net** weighing and **Quantity entry**.

Modeling

Both, one-step Output Weighing and two-step Output Weighing can be modeled with the basic Output Weighing operation shown in "Structure of Output Weighing Operations (with Loop)" (page 44). You can produce the first container/sublot with *OperationMode.CONTINUOUS*. Then, the second container/sublot can be produced with the two-step approach, i.e. first *OperationMode.PREPARE* and afterwards *OperationMode.WEIGH*.

However, you can also model two Output Weighing operations either in sequence or in parallel.

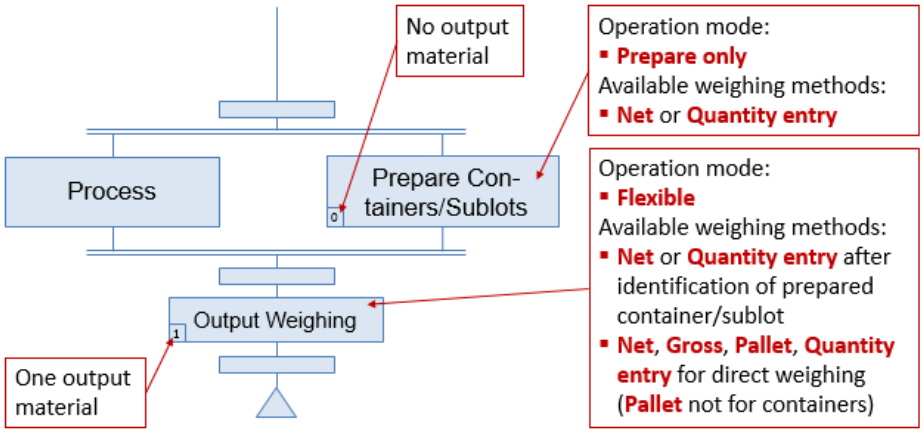

Figure 8: Two-step Output Weighing scenario: preparation and weighing of one output material (sequential)

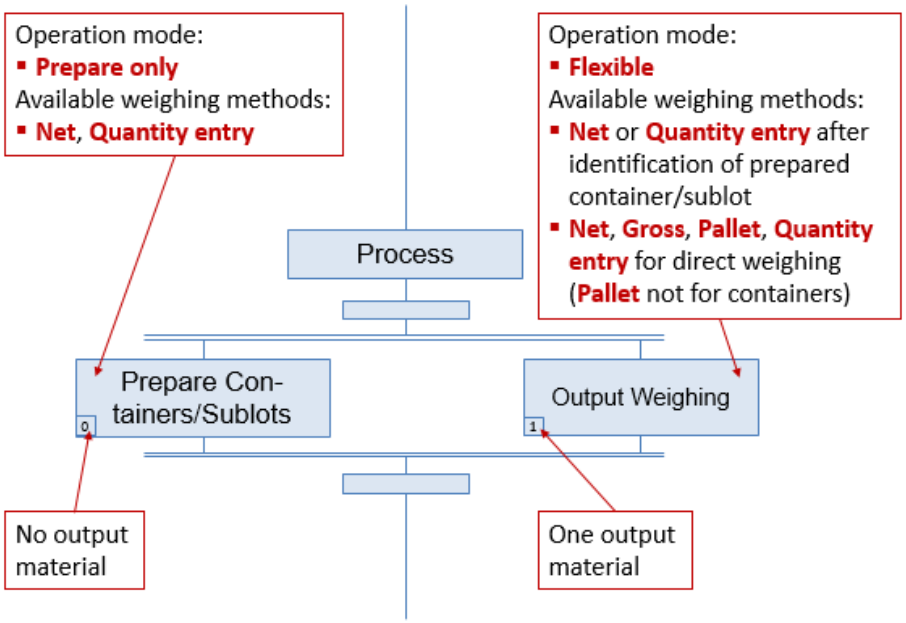

Figure 9: Two-step Output Weighing scenario: preparation and weighing of one output material (parallel)

Both operations can have a material output parameter. If so, both of them must match the same material. But only the parameter of the **Output Weighing** operation must be flagged as MFC-relevant.

The **Prepare Containers/Sublots** operation is responsible for preparing containers/sublots, while the **Output Weighing** operation identifies and weighs the prepared containers/sublots. The behavior of both operations is controlled by the **Operation mode** process parameter of their respective **Manage produced material** phases. The **Operation mode** process parameter provides the following modes according to the *OWOperationMode* choice list:

- PREPARE_ONLY
  - **O Manage produced material** phase:
    - Sets *OWOperationMode.PREPARE*.
    - Disables **Done**-related check.
    - No yield and prorate factor calculation.
  - **O Weigh** phase:
    - Simplified user interface.
    - Performs an automatic completion of the phase after label printing.
    - No weighing is performed since the scale was tared recently (milliseconds before).

- FLEXIBLE
  - **O Manage produced material** phase:
    - Sets *OWOperationMode.CONTINUOUS*.

The only use of the **Operation mode** process parameter is during initialization of the operation mode within the **Manage produced material** phase. Later on, only the operation mode (stored in the model) is used instead of the process parameter.

Weighing of Several Outputs

In another scenario with parallel Output Weighing operations, several one-step Output Weighing operations produce several output materials. In this case, all material output parameters must be flagged as MFC-relevant.

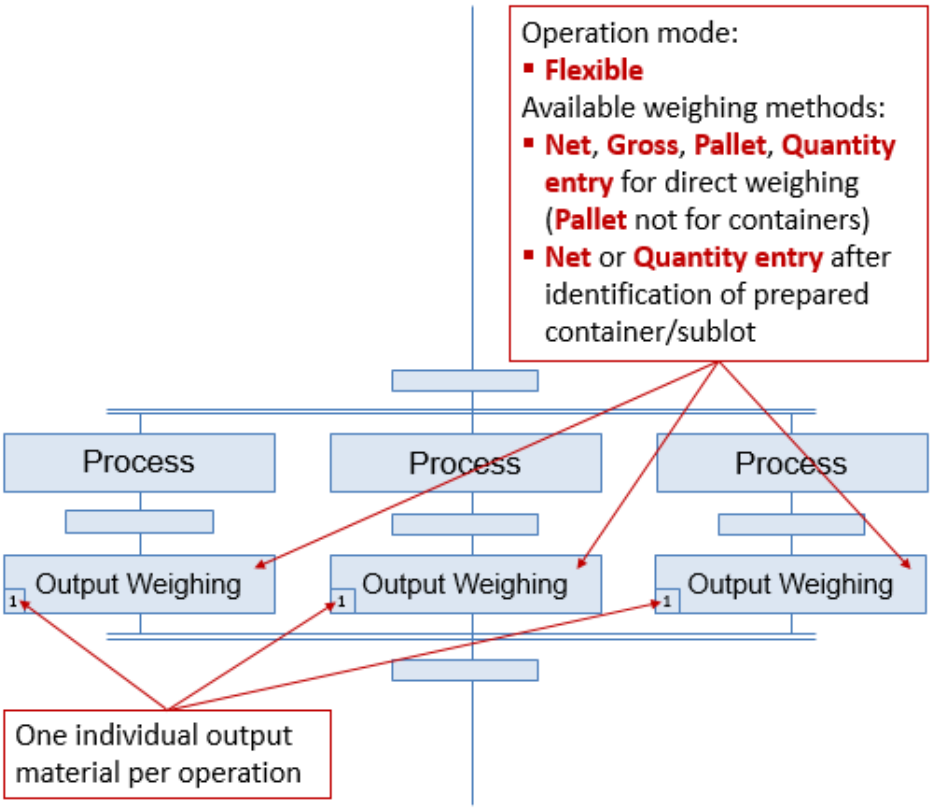

Figure 10: Output Weighing scenario: direct weighing of several output materials

Based on a search algorithm, a **Manage produced material** phase can decide which **OrderStepOutput** must be used: The algorithm searches for phases with material output parameters within the current operation. There must be only one material output parameter flagged as MFC-relevant. This material parameter determines the **OrderStepOutput**.

Specifics Related to the O Manage Produced Material Phase

The **O Manage produced material** phase plays a similar role as the **D Identify material** phase for Dispense and Inline Weighing. The phase controls the Output Weighing operation. The *OWManProdMatModel* model class of the phase maintains:

- a list of the containers/sublots that have been produced for the associated *OrderStepOutput*,
- calculated yield and prorate factor, and
- total position status of the *OrderStepOutput*.

Container/sublot identification within the **O Manage produced material** phase uses the same mechanisms as the **D Identify material** phase. The main difference is that the **O Manage produced material** phase does not support the identification of batches. Therefore, no batch-specific input boxes are available, for example for manual identification.

Multiple operations can work in parallel on the very same **OrderStepOutput**. That means the phase must be aware that other phases can create or modify sublots concurrently. This fact must be taken into consideration.

All phase-specific information must be stored within the phase data or the phase output. Keep in mind that **OrderStepOutput** data as well as sublot data can be changed by other clients at any time.

The list of produced containers/sublots is dynamic. The latest list of containers/sublots can always be fetched from the database. However, the latest list of containers/sublots is not always desired. The typical scenario is a user transaction you wish to support. During the transaction you may wish to fetch the list of containers/sublots at the beginning and work with the list throughout the transaction. In other words, you take a snapshot at the beginning of the transaction and keep it frozen for some time.
In order to support this behavior, the *OWManProdMatModel* model maintains an internal list of containers/sublots. The list is loaded lazily, when accessed by one of the following methods, exposed by the model:

- *getAllProducedSublots()*,
- *getPreparedItems()*, and
- *getFilteredItem(SublotOutputStatus)*.

The filter methods perform a client-side filtering of the containers/sublots stored in the model. If you wish to refresh the list, you can invoke the *refreshProducedSublots()* method. The method tells the model to reload the list of containers/sublots upon next access.

The *getProducedQty()* method uses the internal list of containers/sublots to sum up the quantities of the sublots.

An explicit refresh of the model and the view can be performed by invoking *OWManProdMatView.refreshView()*. The explicit refresh is needed, when the **Done**-related check is performed. Additionally, there is a *RefreshViewListener* inside the *OWManProdMatView* view class. The *RefreshViewListener* ensures that each time the view becomes visible, both view and model are refreshed.

The **Done**-related check (*performDoneCheck()*) is executed during the phase completion check (*performPhaseCompletionCheck()*) if the operator has selected the **Done** option before. Based on refreshed sublot data, yield and prorate factor are calculated.

Due to the model refresh after the phase has been confirmed with the **Confirm** button, it may happen that the calculated model data that is populated into phase data and phase output differs from the data displayed before phase completion. In order to avoid displaying different data, a comparison of the old versus the refreshed model is performed. If the model has changed, *performPhaseCompletionCheck()* returns **false** and displays an appropriate error message. It is important that you observe the following rules when extending the model:

- Only update the model during *phaseCompletionCheck()*.
- Populate model data into phase data, phase output, and process model (e.g. **OrderStep** and **OrderStepOutput**) during *phaseCompletion()*.
- Before updating **OrderStep** and **OrderStepOutput** perform *OrderStep.refresh()*. **OrderStep** data can be changed by other clients at any time.
- Extend the model compare algorithm as needed. Currently, yield, prorate factor, and the produced quantity are calculated data based on the list of containers/sublots.

Yield and Prorate Factor Calculation

The planned quantity of a unit procedure is affected by the prorate factor of its predecessor. For details, see figure below.

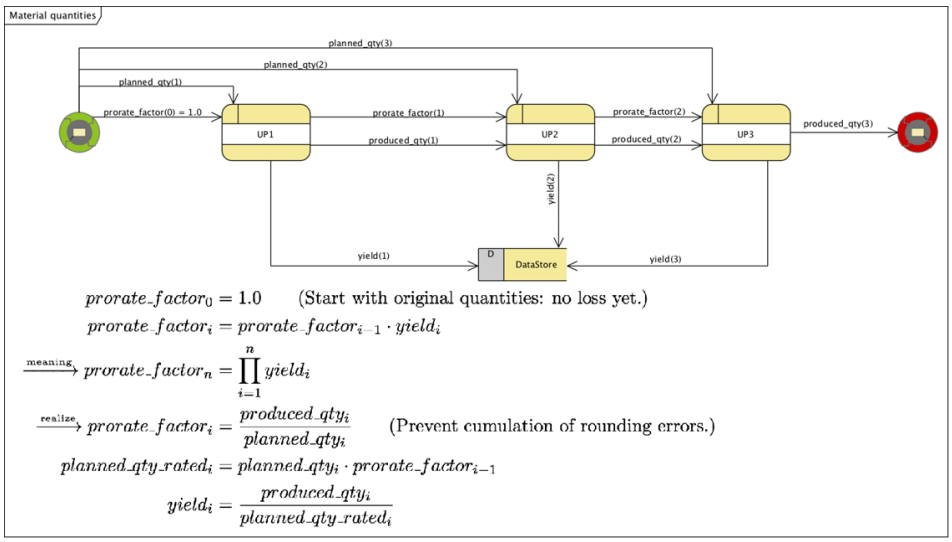

Figure 11: Yield and prorate factor calculation

In short:
Prorate factor = Actual produced output quantity / Original planned output quantity
Yield = Actual produced output quantity / Prorated planned output quantity

User-triggered Exceptions

Each user-triggered exception view has its own view class, derived from *AbstractUsertriggeredExceptionView*.

If you wish to add a new user-triggered exception view, proceed as follows:

- Create a new view class.
- Implement a static factory method:
  - *AbstractUsertriggeredExceptionView createUserTriggeredExceptionView(final OWManProdMatExceptionView exceptionView)*
- Invoke the new factory method within the method:
  - *OWManProdMatExceptionView.createView()*

```
protected void createUI() {
  setLayout(new BoxLayout(this, BoxLayout.Y_AXIS));
  usertriggeredExceptionViews.clear();
  usertriggeredExceptionViews.add(OWManualIdentificationExceptionView.
                              createUserTriggeredExceptionView(this));
  usertriggeredExceptionViews.add(OWAnnulSublotExceptionView.
                              createUserTriggeredExceptionView(this));
  usertriggeredExceptionViews.add(OWReplaceSublotExceptionView.
                              createUserTriggeredExceptionView(this));
```

```
  usertriggeredExceptionViews.add(OWOverrideProrateExceptionView.
                                  createUserTriggeredExceptionView(this));
  populateUserTriggeredExceptionViewsIntoExceptionView();
}
```

Most user-triggered exception views of the **O Manage produced material** phase perform some kind of container/sublot identification. The base class for the views is *AbstractIdentificationExceptionView*. The class provides an input box for the container/sublot identifier.
If a grid with containers/sublots shall be displayed, *AbstractIdentificationWithGridExceptionView* is used as base class.

BARCODE SCANNING WITHIN USER-TRIGGERED EXCEPTION VIEWS

If a user-triggered exception view class overrides the *OWManProdMatIdentificationContext getIdentificationContext()* method and returns a not-null *OWManProdMatIdentificationContext*, the user-triggered exception view receives barcode scan events, if the view or any component within the view have the focus. *OWManProdMatIdentificationContext* determines how the barcode is processed.

System-triggered Exceptions

Conditions for system-triggered exceptions are calculated within *performDoneCheck()*. The check may detect multiple violations. However, they are all signed with one signature.

A system-triggered exception is characterized by: *checkKey*, *riskClass*, *exceptionText*, and *additionalInfo*.

SystemTriggeredExceptions maintains a collection of system-triggered exceptions. The *showExceptionDialog()* method shows the system-triggered exception dialog if the collection contains at least one system-triggered exception and records a single system-triggered exception.

The *OWManProdMatSystemtriggeredExceptions* class contains the *SystemtriggeredExceptionEnum* enumeration. Each member of the enumeration represents a specific system-triggered exception. The enumeration provides methods to check the exception condition and to create the exception text for each exception. Extend the enumeration when you add a system-triggered exception.

Specifics Related to the O Identify Container Phase

The **O Identify container** phase is a specialization of the **Identify equipment** phase. It allows to identify S88 containers only. Furthermore, it can be used in Dispense, Inline Weighing, and Output Weighing operations, i.e. it supports fall-through (page 32) and *OperationContext* (page 25). After a container has been successfully identified, it is bound and a **CONT_ID** trigger is fired. Then, the container is fed into the *TARGET_CONTAINER* property of the *OperationContext* for usage in other phases.

Specifics Related to the O Tare Phase

The **O Tare** phase determines the tare weight of the target container for the output material.

The following processing sequence applies if the operation mode in the operation runtime context is *WEIGH* and the weighing method is not **Quantity entry**. The container/sublot has been created and tared during a *PREPARE* loop earlier. Now, the **O Tare** phase reads the tare value from the identified container/sublot, sends the tare value to the scale, and performs an automatic completion of itself.
In the **Quantity entry** weighing method, the **O Tare** phase is skipped.

Specifics Related to the O Weigh Phase

The **O Weigh** phase weighs the target container or sublot of the produced output material and prints a label for it.
In the **Quantity entry** weighing method, the net weight value and the unit of measure have to be entered manually. The entered unit of measure must be convertible to the unit of measure of the planned quantity as defined in Recipe and Workflow Designer. If no planned quantity is available, the unit of measure of the batch is used. If the batch does not exist, the material's unit of measure is used.

The following processing sequence applies if the operation mode in the runtime context is *WEIGH*. The net weight is weighed (or entered manually) and defined as initial weight of the sublot. If a target weight with its tolerances has been defined for the material and the net weight is less than the lower tolerance or greater than the upper tolerance, the phase creates the **Out of tolerance** system-triggered exception.
For an operation with the **Prepare only** operation mode, the phase does not weigh. In case a target container has been identified, the phase completes automatically. If no target container has been identified, the phase prints a label for the sublot, which has been tared unless the **Quantity entry** weighing method is used, and then completes automatically.

For an operation with the **Flexible** operation mode and if the operation mode in the runtime context is *PREPARE*, the phase weighs or requires a manual weight input to ensure that the prepared container/sublot is really empty.

For an operation with the **Flexible** operation mode and if the operation mode in the runtime context is *CONTINUOUS*, the operator can switch between the **Weigh** and **Prepare** options.

Container Integration for Output Weighing

Not only sublots but also containers can be prepared for Output Weighing. The prepared containers are stored as list of X_S88Equipment references in the X_containers UDA at the *OrderStepOutput* associated with the Output Weighing operation. The list is maintained via *IMESContainerEquipmentService* with *addContainerToOSO()*,

removeContainerFromOrderStepOutput(), *setContainersToOSO()*, *getContainersFromOSO()*, and *hasOSOContainersAssigned()* methods.

The *IdentifiedItems* class was introduced to abstract from sublots and containers. Some methods names reflect this, e.g. *OWManProdMatModel.getPreparedItems()* which provides all prepared sublots and containers as a list of *IIdentifiedItem* objects. The abstraction is heavily used in the **O Manage produced material** phase.

Prepared containers are assigned to the *OrderStepOutput* when the **O Weigh** phase is completed.

In case a prepared container was identified (in the *WEIGH* operation mode), a target sublot is created and assigned to the prepared container. Then, the container is removed from the list of prepared containers. Loaded containers are tracked using their assigned sublots.

This behavior of the **O Weigh** phase is implemented in the *IOperationModeHandler.handleTargetContainer()* method for each operation mode.

An identified target container can hold an optional property for a reference tare. This tare value can be checked and modified by the **O Tare** phase.

Units of Measure Conversions

This section is important for the **O Manage produced material** phase as this phase displays a lot of quantities. But it has an impact on the other phases as well.

In the Production Management Client of PharmaSuite, you can configure a material-specific conversion for an output material; in Recipe and Workflow Designer you can use any unit of measure for planned quantity and the absolute lower and upper tolerances for which there is a conversion in the system. However, you cannot use different units of measure for planned quantity and the tolerance.

IMeasuredValueConverter IOWCommonWeighModel.getConverterForOrderStepOutput() is the material-specific unit of measure converter. Use the converter to convert container/sublot quantities, tolerances, or weighed quantities.
To retrieve the main target unit of measure of the conversion, use *IUnitOfMeasure IOWCommonWeighModel.getUnitOfMeasureOfOrderStepOutput()*.
The returned unit of measure is the unit of measure of the planned quantity as defined in Recipe and Workflow Designer. If no planned quantity is available, the unit of measure of the batch is used. If the batch does not exist, the material's unit of measure is used.

TIP

A produced sublot must have the unit of measure of its batch, according to FactoryTalk ProductionCentre.
Please keep in mind that PharmaSuite supports different kinds of batches (batch for intra material, batch for finished goods).

Target Weight Enforcement

The functionality related to the initialization and updating of a target weight for Output Weighing is bundled in *IOutputWeighingService*:

- *calcTargetWeightAndTolerances*
  The method initializes (calculates and sets) the nominal target weight and tolerances based on the definitions made in the master recipe.
- *setNominalTargetWeightAndTolerances*
  The method can be used to dynamically set or update the target weight and its tolerances during execution.

Access to ScaleEquipment and Scales Objects

This section is important for the **D Select scale** and **O Select scale** phases. But it has an impact on the other phases as well.

When working with scales, two main objects are important:

- **IMESScaleEquipment**
  The **ScaleEquipment** object that can be bound to an **OrderStepInput** or **OrderStepOutput**. In addition, the **ScaleEquipment** object keeps track of scale testing or calibration and maintains a logbook.
- **Scales**
  A plain Java object that provides access to a physically connected scale device unless the **ScaleEquipment** object has been configured as **Manual scale** in Data Manager - Equipment. Due to current constraints, each PharmaSuite client supports only one active scale at a time. In order to maintain this behavior properly, PharmaSuite uses an exclusive *ScalesResource* to ensure the singleton constraint. *ScalesResource* can be linked to a *RuntimeActivitySet*, an *IMESRtOperation*, or *IMESRtUnitProcedure*.
  For Dispense and Inline Weighing, *ScalesResource* is linked to the unit procedure, but Output Weighing links it to the runtime operation instead.
  The latter is important to protect *ScalesResource* from being taken over by another runtime operation, which can happen when two-step Output Weighing operations run in parallel.

IWeighBaseModel provides methods to access the **Scales** and **ScaleEquipment** objects:

- *IMESScaleEquipment getSelectedScalesEquipment*
  Loads the **ScaleEquipment** object from phase data for completed phases or from the runtime operation context for active phases.
- *Scales getAndConnectToSelectedScalesEquipment()*
  In addition to the first method, the second method establishes a physical connection to the **Scales** object and links it to the *ScalesResource* object. If the **ScaleEquipment** object has been configured as **Manual scale** in Data Manager -

Equipment, no connection is established and the method returns null. Before you call the method for non-manual scales, make sure that either the current runtime operation owns the *ScalesResource* singleton or that the associated *IMESScaleEquipment* object can be bound to the **OrderStepInput** or **OrderStepOutput**.

To following scenario illustrates the situation: A unit procedure contains an Inline Weighing operation followed by an Output Weighing operation looping back into the Inline Weighing operation. The Inline Weighing operation performs a **Net removal** and keeps the scale connected, while it passes the control to the Output Weighing operation. If the Output Weighing operation runs on the same device as the Inline Weighing operation, the Output Weighing operation tries to use a different scale. The connection of the scale used within Inline Weighing is interrupted and the communication corrupted. This is prevented by the measure described above.

Chapter 8

Adapting Phases for Weighing Support

This chapter applies to the **Get weight** phase. The phase allows to model weighing operations outside of Dispense operations.

The **Get weight** phase differs from the Dispense, Inline Weighing, and Output Weighing phases in the following areas

- It does not use the *RtOperationContext* (page 25). Instead it uses process parameters to configure the phase and the *RtPhaseContext* to maintain its state.
- It does not support fall-through (page 32). Therefore this phase must not be used within a Dispense, Inline Weighing, or Output Weighing loop.

For more information on Weighing support, see also "Functional Requirement Specification Dispense and Inline Weighing" [A8] (page 63) and "User Manual Dispense and Weighing Phases" [A9] (page 63).

Phase Modes of the Get Weight Phase

The behavior of the **Get weight** phase is mainly driven by two process parameters, which are supported by choice lists:

- Operation mode: GWOperationMode<version>
- Weighing method: GWWeighingMethod<version>

The phase must implement phase executors for the relevant elements of the Cartesian product of the two process parameters. This is represented by phase mode handlers. All phase mode handler-related classes implement the *IPhaseModeHandler<version>* interface. The interface contains some of the life-cycle methods of a phase executor like *startPhaseExecution()*, *performPhaseCompletionCheck()*, and *performPhaseCompletion()*.
The phase executor of the **Get weight** has an *IPhaseModeHandler<version>* instance as member field and uses that field as delegate.

The *IPhaseModeHandler<version>* interface contains the following methods:

Methods	Description
start()	It is called by *AbstractWeighPhaseBaseExecutor<version>* when the *createPhaseComponent()* method is executed. The *start()* method exists for a long time, much longer than *startPhaseExecution()* since it is called

Methods	Description
	while creating the UI. The UI is not displayed yet and the *start()* method must not do anything that relies on the UI being visible.
startPhaseExecution()	It is called by the PharmaSuite framework after the phase UI has been added to the execution view. The phase mode handler uses the method to display the **Confirm scale load** message if needed.
performZeroCompletionCheck()	They are directly linked to the **Zero** or **Tare** buttons of the phase. The phase implements a sequence of three simple phases within a single phase: **Zero**, **Tare**, and **Weigh**. The *perform<Zero,Tare>CompletionCheck()* and *perform<Zero,Tare>Completion()* methods are implementing the associated functionality of the **Zero** and **Tare** phases.
performZeroCompletion()	
performTareCompletionCheck()	
performTareCompletion()	
performPhaseCompletionCheck()	---
performPhaseCompletion()	---
getButtonConfigurationList()	---
isTareSupported()	---
isScaleEquipmentUsedByHandler()	---
isScaleActive()	---
markSelectedScalesOffline()	---

The following figure shows the hierarchy of phase mode handler classes.

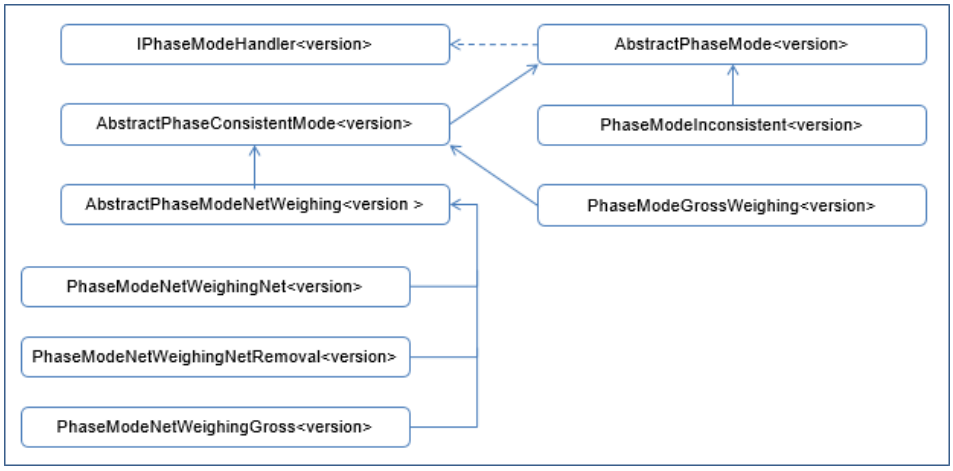

Figure 12: Class hierarchy of PhaseModeHandler

Consistency Checks

The configuration of the **Get weight** phase should be done thoroughly due to the various features the phase provides. Since almost everything can be defined via input expression, a consistency check is required during the startup of the phase. In fact, the check is performed when the phase mode handler of the phase is determined. *GetWeightConsistencyChecker<version>* performs the check itself. If the check fails, the specific *PhaseModeInconsistent<version>* phase mode handler is used. The phase mode handler triggers a system-triggered exception. Most other methods throw *UnsupportedOperationException* to make sure that no business logic is executed.

The check suite itself is hard-coded in *GetWeightConsistencyChecker<version>*. It performs the following checks:

- Issues with equipment objects (only if the weighing method is not **Quantity entry**)
  - Equipment entity parameter must not be null.
  - Equipment entity parameter must represent an **IMESScaleEquipment** object.
  - *IMESScaleEquipment* must have a valid configuration.
  - *ScalesResource* must be available for the current runtime operation.
- Issues with **Operation mode** and **Weighing method** parameters
  - Operation mode must be defined.
  - If the operation mode is **Net weighing**, the weighing method must be defined too.
- Issues with planned quantity/tolerances if they have been defined
  - Units of measure must not be null.
  - Units of measure must be suitable for weighing.
  - Nominal value and tolerances must be possible with the given scale.

Reference Documents

The following documents are available from the Rockwell Automation Download Site.

No.	Document Title	Part Number
A1	PharmaSuite Technical Manual Administration	PSAD-RM008E-EN-E
A2	PharmaSuite Technical Manual Configuration & Extension - Volume 1	PSCEV1-GR008E-EN-E
A3	PharmaSuite Technical Manual Configuration & Extension - Volume 2	PSCEV2-GR008E-EN-E
A4	PharmaSuite Technical Manual Configuration & Extension - Volume 3	PSCEV3-GR008E-EN-E
A5	PharmaSuite Technical Manual Configuration & Extension - Volume 4	PSCEV4-GR008E-EN-E
A6	PharmaSuite Technical Manual Configuration & Extension - Volume 5	PSCEV5-GR005E-EN-E
A7	PharmaSuite Technical Manual Developing System Building Blocks	PSBB-PM007E-EN-E
A8	PharmaSuite Functional Requirement Specification Dispense and Inline Weighing	PSFRSDI-RM006E-EN-E
A9	PharmaSuite User Manual Dispense and Weighing Phases	PSPD-UM005F-EN-E
A10	PharmaSuite Functional Requirement Specification Output Weighing	PSFRSOW-RM002E-EN-E

TIP

To access the Rockwell Automation Download Site, you need to acquire a user account from Rockwell Automation Sales or Support.

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
Introduction (page 1)	Document also applies to the **Show GHS data** phase. Configuration keys of the **D Tare** and **O Tare** phases added.	1.0

Changes related to "Administration" (page 5):

Object	Description	Document
---	---	---

Changes related to "Configuration and Extension" (page 7):

Object	Description	Document
Configuration Keys Specific to the D Tare Phase (page 13)	New section.	1.0
Configuration Keys Specific to the O Tare Phase (page 13)	New section.	1.0

Changes related to "Supported Scale Modes" (page 15):

Object	Description	Document
---	---	---

Changes related to "Adapting Phases for Dispense" (page 17):

Object	Description	Document
Adapting Phases for Dispense (page 17)	Document also applies to the **Show GHS data** phase.	1.0
Structure of Dispense Operations (page 18)	Document also applies to the **Show GHS data** phase.	1.0
Weighing Methods (page 19)	WD_ES_QUANTITY_ENTRY_D access privilege replaces WD_ES_QUANTITY_ENTRY access privilege.	1.0
The Model-View-Controller Concept of the Dispense Phases (page 23)	The *AbstractWeigh** base classes are not used for the **Show GHS data** phase in order to enable the usage of the **Show GHS data** phase outside of a Dispense, Inline Weighing, or Output Weighing operation.	1.0
Container Integration for Dispense (page 30)	The **D Tare** phase supports an optional property for a reference tare.	1.0
Exceptions (page 31)	The **Show GHS data** phase supports fall-through without a specific user-triggered exception.	1.0

Changes related to "Adapting Phases for Inline Weighing" (page 39):

Object	Description	Document
Adapting Phases for Inline Weighing (page 39)	Document also applies to the **Show GHS data** phase.	1.0
Structure of Inline Weighing Operations and Unit Procedure (page 39)	Document also applies to the **Show GHS data** phase.	1.0
Weighing Methods (page 40)	WD_ES_QUANTITY_ENTRY_D access privilege replaces WD_ES_QUANTITY_ENTRY access privilege.	1.0

Changes related to "Adapting Phases for Output Weighing" (page 43):

Object	Description	Document
Adapting Phases for Output Weighing (page 43)	Document also applies to the **Show GHS data** phase.	1.0
Structure of Output Weighing Operations (with Loop) (page 44)	Document also applies to the **Show GHS data** phase.	1.0
Weighing Methods (page 45)	WD_ES_QUANTITY_ENTRY_O access privilege replaces WD_ES_QUANTITY_ENTRY access privilege.	1.0
One-step Output Weighing (page 46)	Behavior in case a target weight has been defined for a material output parameter.	1.0

Object	Description	Document
Two-step Output Weighing (page 46)	Behavior in case a target weight has been defined for a material output parameter.	1.0
Specifics Related to the O Weigh Phase (page 54)	Behavior in case a target weight has been defined for a material output parameter.	1.0
Container Integration for Output Weighing (page 54)	The **O Tare** phase supports an optional property for a reference tare.	1.0
Target Weight Enforcement (page 56)	Behavior in case a target weight has been defined for a material output parameter.	1.0

Changes related to "Adapting Phases for Weighing Support" (page 59):

Object	Description	Document
---	---	---

Index

A

allowOverrideReferenceTare (D Tare phase) (configuration key) • 13
allowOverrideReferenceTare (O Tare phase) (configuration key) • 13
Audience • 3

B

Barcode support • 30
Barcode template • 29
Busy painter • 31

C

Configuration key (D Tare phase) allowOverrideReferenceTare • 13
Configuration key (O Tare phase) allowOverrideReferenceTare • 13
Context • 25
Extending • 36
Conventions (typographical) • 3
Cost center-related dispensing • 5

D

Dispense • 17
Container • 30
D Identify material • 26
Operation • 18
Report • 33
Weighing method • 19
Download Site • 63

E

Exception • 31
Fall-through • 32
Unforeseen resume • 32
Experimental batch • 7
Extension • 34

F

Fall-through • 32

H

Helper method • 30
Barcode support • 30
Busy painter • 31

I

Inline Weighing • 39
Container • 41
No target sublot creation • 41
Operation • 39
Unit procedure • 39
Weighing method • 40
Intended audience • 3

M

Model-View-Controller • 23
MVC • 23

N

Naming • 3
Navigator action • 33

O

OperationContext • 25
Output Weighing • 43
Container • 54
O Identify container phase • 54
O Manage produced material phase • 50
O Tare phase • 54
O Weigh phase • 54
One-step Output Weighing • 46
Operation • 44
OrderStepOutput • 43
Prorate factor calculation • 52
Scale • 56

Several outputs • 49
Target weight enforcement • 56
Two-step Output Weighing • 46
Unit of measure conversions • 55
Weighing method • 45
Yield calculation • 52

P

Phase data • 25
Adapting • 35
Adding • 35

R

Recipe-related usage type • 7
Reference documents • 63
Rockwell Automation Download Site • 63

S

Scale mode • 15
Shop floor-defined dispensing • 7
Sub-report • 33

U

Unforeseen resume • 32

W

Weighing support • 59
Consistency checks of Get weight phase • 61
Get weight phase • 59
Modes of Get weight phase • 59