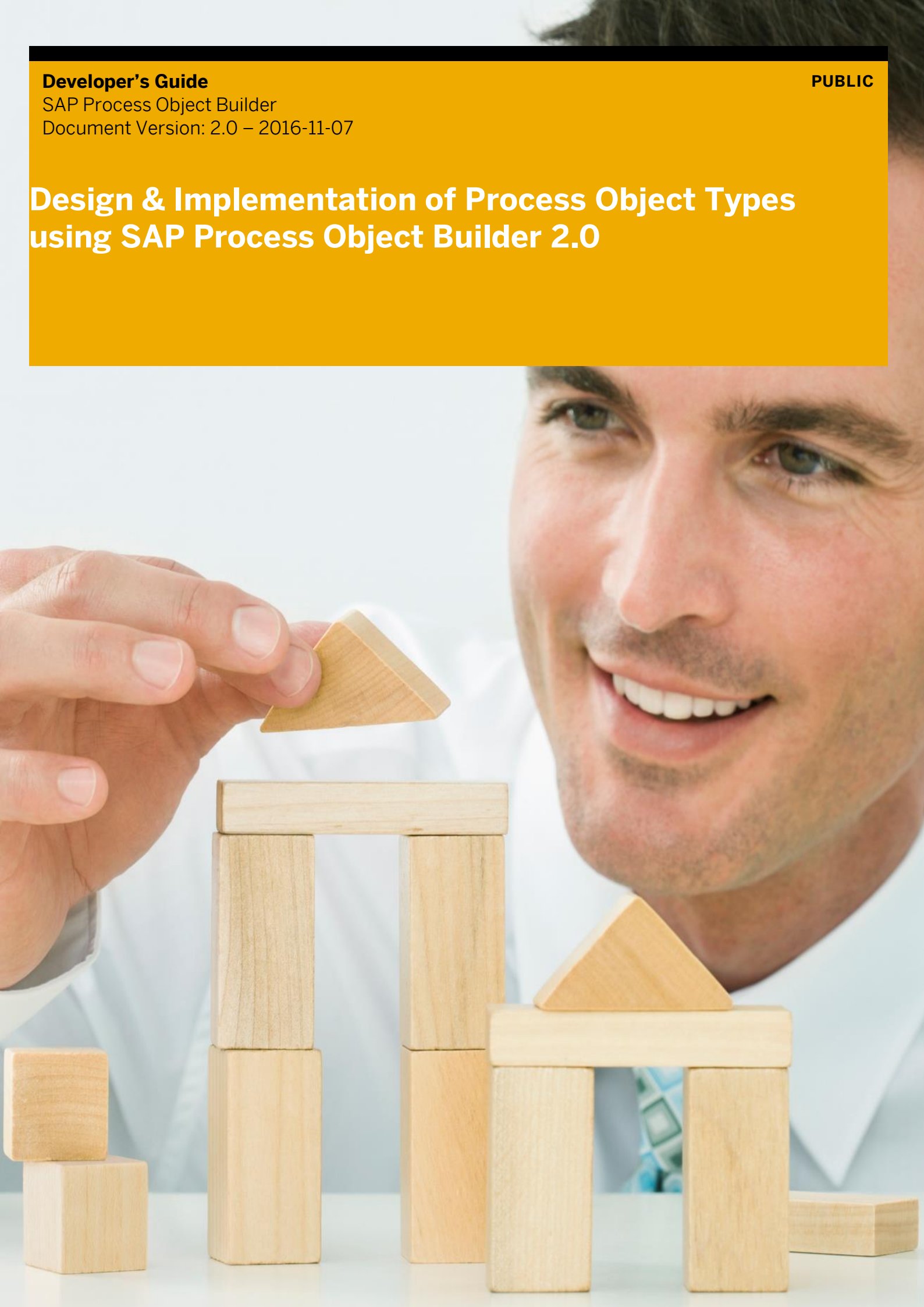
Developer's Guide **PUBLIC**
SAP Process Object Builder
Document Version: 2.0 – 2016-11-07

Design & Implementation of Process Object Types using SAP Process Object Builder 2.0



Typographic Conventions

Type Style	Description
Example	Words or characters quoted from the screen. These include field names, screen titles, pushbuttons labels, menu names, menu paths, and menu options. Textual cross-references to other documents.
Example	Emphasized words or expressions.
`EXAMPLE`	Technical names of system objects. These include report names, program names, transaction codes, table names, and key concepts of a programming language when they are surrounded by body text, for example, SELECT and INCLUDE.
`Example`	Output on the screen. This includes file and directory names and their paths, messages, names of variables and parameters, source text, and names of installation, upgrade and database tools.
`Example`	Exact user entry. These are words or characters that you enter in the system exactly as they appear in the documentation.
`<Example>`	Variable user entry. Angle brackets indicate that you replace these words and characters with appropriate entries to make entries in the system.
`EXAMPLE`	Keys on the keyboard, for example, `F2` or `ENTER`.

Modeling Conventions and Generated Artifact Names

This guide uses diagrams in accordance with several standard modeling techniques in order to explain the various architecture and design concepts.

The following modeling conventions are used in the guide:

SAP SOA Modeling Methodology is used for models in the SOA context.

UML2.0 is pre-dominantly used as the methodology for detailed artifact modeling.
Sometimes SAP Block Diagrams are used to explain high-level concepts. (TAM Standard – a SAP Block Diagram broadly corresponds to Composite Structure Diagram in UML.)

<<Interface>> indicates that **Interface** is a stereotype of an artifact.

<IF_NAME> indicates that **IF_NAME** is a placeholder for the name of an artifact.

<MLB> is the generic placeholder for the abbreviated name of a POT.
During generation of a POT, **<MLB>** is derived with the following concatenation rule: `[/$ABAPNameSpace$/] + [$ABAPArtifactPrefix$] + [$ABAPAbbrev$]`

- `$ABAPNameSpace$ =` ABAP Namespace
- `$ABAPArtifactPrefix$ =` A standardized prefix for ABAP artifacts wherever applicable (for example **CL_** for ABAP class; no prefix for DDIC type and so on)
- `$ABAPAbbrev$ =` the concatenation of the ABAP Abbreviation (Specification Wizard Step *Define ABAP Environment*) and the POT ABAP short name (Specification Wizard Step *Enter ABAP Short Names*) separated by underscore and the POT version.

For the sample scenario of this developer guide, this means:

- `$ABAPNameSpace$` = **/PL9/**
- `$ABAPAbbrev$ =` **GXX_FB01**

Therefore:

`<MLB> =` **/PL9/GXX_FB01** for DDIC type name

`<MLB>` **= /PL9/CL_GXX_FB01** for ABAP Class name and so on

Document History

Version	Date	Change
1.0	2015-09-01	Based on SAP Process Object Builder 1.0 Support Package 5 Initial Version
1.0.1	2016-05-30	Based on SAP Process Object Builder 1.0 Support Package 5 Handling of Process Log corrected
2.0	2016-11-07	Based on SAP Process Object Builder 2.0

Table of Contents

1 Preface **9**
1.1 Target Audience 9
1.2 Prerequisites 9
1.3 References 11
1.4 Disclaimer 11

2 Introduction **12**
2.1 Context 12
2.2 Structure of this Guide 15
2.3 System Prerequisites for POB and POT 15
2.3.1 General Prerequisites 16
2.3.2 Prerequisites for Back-End Services 16
2.4 Deployment Options and POT Shipment 17
2.5 Sample Business Process: Travel Booking 17
2.6 How to Read this Document? 18

3 POT Definition **20**
3.1 Model-Driven Software Development (MDSD) 20
3.1.1 Versioning of a POT 21
3.2 Identification of a POT 21
3.2.1 Process Definition 21
3.2.2 Investigation of Back-End Business Objects and Services 24
3.3 Modeling a POT 24
3.3.1 POT Modeling using SAP SOA Methodology 24
3.3.2 POT Object Model 27
3.3.3 'Beautified View' Concept 28
3.3.4 POT Object Model Template 29
3.3.5 Predefined Nodes 32
3.3.6 POT Object Model in ESR 34
3.3.7 Separation of Modeling and Productive Entities 37

4 POT Programming Model **38**
4.1 Basic Principles 38
4.1.1 Separation of Concerns: Process vs. Business Logic 38
4.1.2 Pattern for Process Flow 38
4.1.3 Multiple Consumption Options 39
4.2 Phase Model 39
4.3 Status Model 40
4.4 Status Transitions 41
4.5 Phase Implementation 43
4.5.1 Create Phase 43
4.5.2 Check Phase 45
4.5.3 Execute Phase 46
4.5.4 Node Dependencies 49
4.5.5 Cross-references 50

5 POT High-Level Design **51**
5.1 Block Diagram 51
5.1.1 ESR Content 52
5.1.2 POT (ABAP Artifacts) 53
5.1.3 Editing UIs 53
5.1.4 BPM Content 53
5.1.5 Back-End Services 53
5.2 Package Structure 54
5.2.1 POT Core 55
5.2.2 POT Editing UI 55
5.2.3 Package Encapsulation 56
5.3 POT Design & Specification Checklist 57

6 POT Detailed Design **62**
6.1 Layer Concept 63
6.2 Application Logic Layer 64
6.2.1 API Design 65
6.2.2 Database Tables 67
6.2.3 API for BPO 68
6.2.4 API for BPON 69
6.2.5 Version API 70
6.2.6 Message Persistence 70
6.2.7 Phase Handling 71
6.3 Communication Layers 81
6.3.1 Inbound Communication 82
6.3.2 Outbound Communication 90
6.4 UI Layer 96
6.4.1 Screen Structure of the Editing UI 98
6.4.2 Actions on the Editing UI 101
6.4.3 Technical Structure of the Editing UI 102
6.4.4 Technical Behavior of the Editing UI 105
6.4.5 Data Buffering in the Model 111
6.5 General Concepts 112
6.5.1 Transaction Control 112
6.5.2 Exceptions and Message Classes 113
6.5.3 Instance Management 114
6.5.4 Status Handling 114
6.5.5 Event Handling 117
6.5.6 Cross-Reference Handling 118
6.5.7 Process Monitoring & Analytics 120
6.5.8 Archiving 122

7 POT Implementation **128**
7.1 Custom Phase Implementation 128
7.1.1 Conventions and Disclaimers 128
7.1.2 Goals of Phase Implementation 129
7.1.3 Basic Principles of Phase Implementation 129
7.1.4 Helpers for Phase Implementation 136
7.1.5 Create phase – Custom Implementation 141
7.1.6 Check Phase – Custom Implementation 148
7.1.7 Execute Phase – Custom Implementation 158
7.2 Implementing Archiving Extensions 172
7.2.1 Additional Selection Criteria for Archiving Write Program 172
7.2.2 Additional Archivability Checks 172

7.2.3 Data Mapping for Archive Search ... 173
7.2.4 Condition Fields for ILM Rules ... 174
7.3 Implementing Additional Authorization Checks ... 174
7.4 Internal Types ... 175
7.5 Usage of WSDL-based backend services ... 177

8 Implementation and Extension of the Editing UI ... 180
8.1 Extending the Search Functionality ... 180
8.1.1 Adding Additional Search Criteria ... 181
8.1.2 Adding Fields to the Search Result ... 182
8.2 Implementing Main and Subpages ... 183
8.2.1 Goals of the Implementation of Main and Subpages ... 183
8.2.2 Phase Model for Processing Main and Subpages ... 184
8.2.3 Basic Principles ... 185
8.2.4 Tools and Helpers ... 188
8.2.5 Custom Feeder Class Implementation – Basics ... 193
8.2.6 Custom Feeder Class Implementation – Advanced Features ... 200
8.2.7 Custom Application Controller Implementation ... 206
8.2.8 Custom Implementation of Navigation and Data Creation ... 210
8.2.9 Adaption of the generated FPM configurations ... 216

Terms and Abbreviations ... 217

Table of Figures ... 218

Table of Tables ... 221

Table of Listings ... 222

1 Preface

This document is primarily intended for developers who want to create process object types (POTs, also known as process objects) using SAP Process Object Builder. These developers are called POT developers. The guide aims to describe the capabilities of SAP Process Object Builder (POB) and all the aspects that should be considered in order to successfully design and develop process objects. It also provides recommendations and how-tos for leveraging the various features of POB.

1.1 Target Audience

We assume that the following software development roles are involved in a POT implementation project:

- Developers
  especially in the area of service implementation and POT implementation
- Development Architects
  especially architects responsible for designing the process object type cut and also for related software logistics like software component cut, package structure, namespace definitions etc.
- Integration Experts / Integration Architects
  responsible for enterprise services governance and all other aspects related to cross-component integration
- Implementation consultants
  with primary experience of ABAP / PI implementations, responsible for POT implementation in partner and customer implementation projects

As the development of process object types involves many aspects related to SAP enterprise services (web services with semantics and governance of SAP), process integration and process orchestration, it is expected and desirable that the roles mentioned above also have a general experience in the areas of process integration (SAP PI) and process orchestration (SAP BPM).

The following section provides a detailed list of knowledge pre-requisites.

1.2 Prerequisites

The target audience described in the previous section should have knowledge in the development-related topics listed in the table below. The table is sorted according to whether knowledge about a given topic is classified as "Mandatory", "Highly Recommended" or "Recommended".

Topic	Know-how Requirement	Comment
ABAP Objects	Mandatory	The ABAP artifacts generated for a POT are primarily in ABAP Objects. Know-how on the topic is required to gain a basic understanding of how the generated coding handles specific requirements.

Topic	Know-how Requirement	Comment
SAP Process Object Builder Product Documentation	Mandatory	The Installation Guide, Administrator's Guide and the application help for SAP POB provide detailed information about system prerequisites and the tools that are used as well as information about POT modeling concepts and the POB wizards for creating a POT.
Floor Plan Manager (FPM) for Web Dynpro ABAP (WDA)	Mandatory	When POTs run into exceptional situations at runtime, editing UIs are required to modify the data of the process object and re-trigger the execution. These editing UIs are also generated by POB based on some basic definitions of which data needs to be part of which screen. The generation of editing UIs is based on FPM and the UIs themselves are generated as Web Dynpro ABAP applications.
SAP Enhancement Concept (BAdIs)	Mandatory	POB generates BAdIs for custom enhancements, if specified to do so.
SAP XI / PI Design time (PI Integration Builder)	Mandatory	POB uses the Enterprise Services Repository (ESR, also known as Integration Builder in SAP NetWeaver™ PI/XI) as the modeling environment at design time. Some steps in the POT generation with POB involve manual tasks in ESR, like copying data types and activating the resulting change lists and so on. POB also generates all remote interfaces of a POT into ESR.
SAP XI / PI Configuration time (PI Integration Directory and SOA Manager)	Mandatory	At configuration time, the consumer proxies for the service interfaces of a POT must be configured against the corresponding back-end provider services. To do so, either a SOA Manager-based configuration or an XI/PI configuration (also known as Integration Directory) can be used.
Testing Environment for SOA services (SPROXY, WS Navigator, etc.)	Mandatory	Enterprise services have a separate test environment. In addition to SAP tools, some external tools (for example soapUI) might also be used. Basic working knowledge with the chosen test tool is required.
Services Test Automation	Highly recommended	Automated service tests facilitate faster testing processes and also a more comprehensive bug identification.
SAP BPM	Highly recommended	The business processes implemented by POTs are often automated using a Business Process Management (BPM) tool is required.
Object-Oriented Design Patterns	Highly recommended	Some of the generated ABAP classes and interfaces also rely on and follow well-known design patterns like Singleton, Façade and Adapter.
SAP SOA Modeling Methodology	Highly recommended	Modeling and implementation of a POT follows the SAP SOA Modeling Methodology, especially business object modeling, service operation design and global data types design.

Topic	Know-how Requirement	Comment
Tools related to service implementation infrastructure at SAP (SIW, ECH, PPO, PSJ, CNS, OAF, Process Observer)	Highly recommended	The generated artifacts of a POT contain a built-in integration to most of these tools. Working knowledge of the tools would be helpful for testing / debugging a POT.
SAP PI – Event Interfaces	Recommended	POB generates service interfaces as event interfaces.
BRFplus	Recommended	A POT can optionally use business rules that are implemented using the SAP BRFplus framework. These rules can be used to check the consistency of data from a business perspective and to control the subsequent execution of a process object. POB can generate the corresponding BRFplus artifacts.

1.3 References

Product Documentation for SAP Process Object Builder

1.4 Disclaimer

The processes described in this guide are used as examples and merely serve to illustrate specific aspects of process objects purely from a learning perspective. No claims are made as to the business validity of these sample business processes. SAP assumes no responsibility for errors or omissions in this document.

2 Introduction

SAP Process Object Builder (POB) is a tool that enables rapid development of software artifacts that are needed for an easier consumption of enterprise services and the execution of real-world business processes in a consistent manner. To do so, you need to be familiar with the POB as a development tool. However, it is also important to understand the motivation for developing a new POT before using the tool itself. POTs belong to a different logical architecture layer than the back-end applications. Therefore, POTs must also address some of the specific challenges that are typical of this layer. You need to be aware of these challenges in order to better understand and leverage the features that POB offers to address them. Therefore, this chapter briefly discusses the background and the need for POTs in general and the motivation for using POB to develop a POT.

2.1 Context

Typically, real-world IT landscapes can broadly be classified into two major architectural layers: channel applications and back-end applications (also known as platform applications). In many cases, there is a clear separation between these two architectural layers, mainly driven by the fact that the architecture of the channel applications layer often requires more agility and different innovation cycles. (The typical technology turnaround time is approximately 2-3 years). A classic example of this was the introduction of online banking (that is, using the web) in the mid-1990s. This did not require many new functions or products in the back-end applications area, but existing functions had to be provided with new, unexplored channels, which bore unknown risks and security issues.

A more recent example is mobile banking.

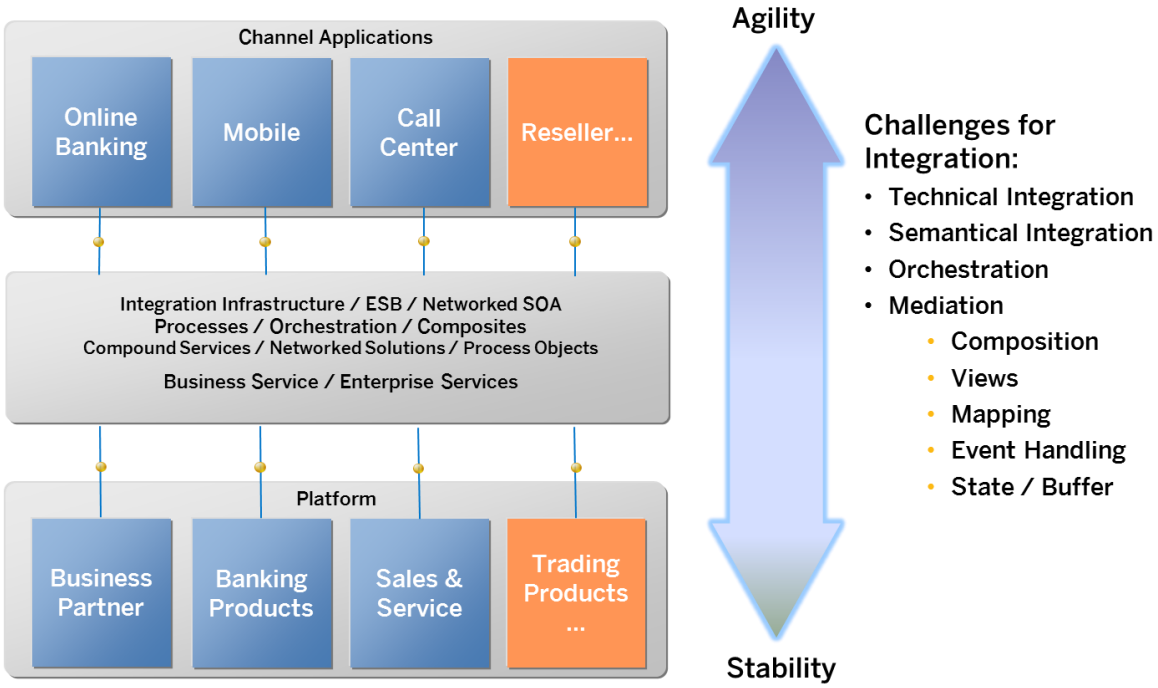


Figure 1: Typical IT Landscapes and Integration Challenges

From Figure 1, it can be seen that for any enterprise, there are two major approaches to differentiate from competition: Adding new business functions to the business objects of the backend platform and providing a user-friendly, rich yet easy-to-use interface for customers, partners and employees.

Residing in between these two architectural layers is the third architecture layer (called integration layer), which contains a multiplicity of components from different vendors built with different technologies that need to be interfaced. The main task of these components is to address challenges in the following topic areas:

Technical Integration: First of all, there are multiple instances involved, which results in a network. Web services have become a commonly accepted standard for communication over a network. Over the years, the means of integration have changed from EAI (Enterprise Application Integration) to SOA principles. Today, there is no big customer that does not need an integration infrastructure - based on web services technology.

Semantic Integration: Web services include a contract. A clarification of meanings is required to understand the content of the contract. The semantics of applications (that is, the meaning of data) must be understood correctly. (For example, an account number in a platform component might mean far more than just a number for the account).

Mediation: Channel-specific requirements and back-end offerings usually do not fit together. It is not realistic to expect that the platform can offer interfaces that completely match the channel-specific requirements. The topics to be addressed in this area could be, for example, the orchestration of back-end services, workflow sequencing in channels, mapping between channel-specific views and back-end objects, buffering, state handling, event processing and so on.

In short, the components in the integration layer are intended to:

- Simplify back-end interfaces by reduction/aggregation (also known as compound services)
- Address generic cross-cutting concerns like mediation
- Provide faster development times for more agility on the channel applications architecture layer.

This is valid enough motivation for a (semi-) automated application development tool like POB.

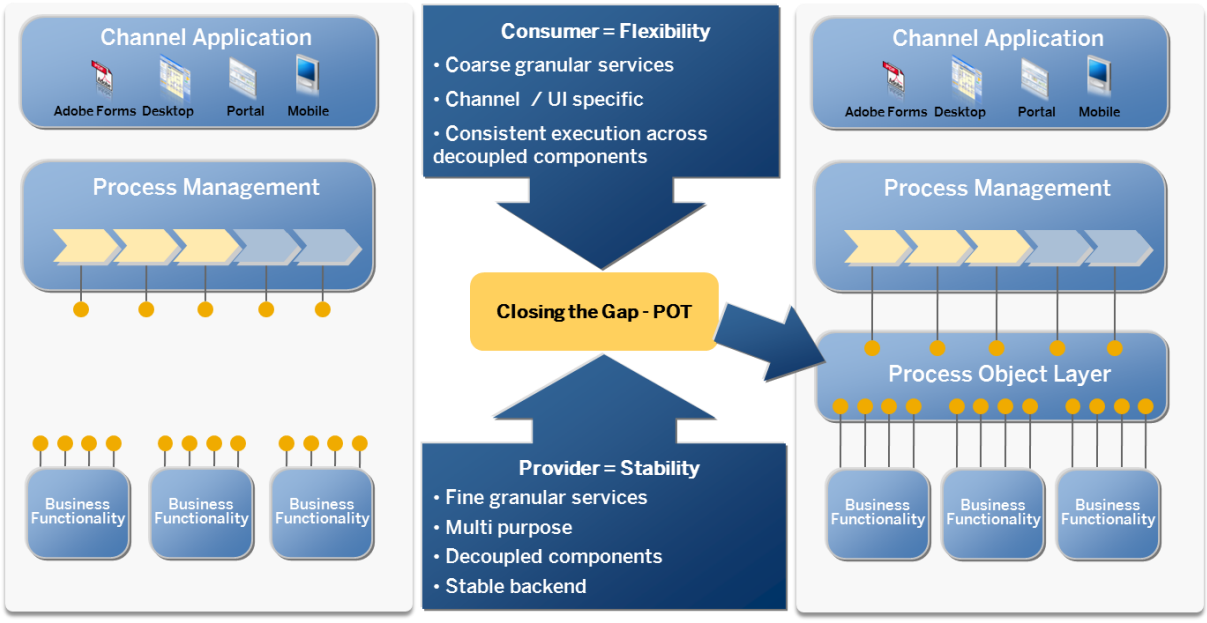


Figure 2: Motivation for Process Object Types

In addition, even though the integration layer is primarily intended to ensure the separation of channel applications from back-end applications, there is no clear standard or guideline available as to how process knowledge and business functionality should ideally be distributed across the three architecture layers. Each software application vendor in the market provides different offerings in that respect.

SAP offers the following features with regard to the three architectural layers:

- **A stable SOA-based application platform** (in the back-end applications architecture layer) with fine-granular business objects and corresponding enterprise services for homogenous provisioning and consumption of the business functionality
- **A business process management tool (SAP BPM)** for an automated execution of web-services (in SOAP protocol). BPM is best suited for the automation of coarse-granular web services (compound services).
- **SAP Process Object Builder** to build process object types (POTs) using a tool-based rapid application development (RAD) approach to close the gap for defining compound services and to address some of the integration and mediation challenges described above.

SAP's strategy is to simplify the consumption of its back-end applications layer components (application platform). This completes the context for the positioning of POTs that can be rapidly developed using POB.

The various features and options offered by POB for generating POTs, which are described in the rest of the document, are best understood with this overall context in mind.

Motivation for Process Object Types

Process Object Types (POTs) address the gap for:

- Compounding fine-granular services of back-ends to a service required for agile development of UI applications
- An implementation that addresses typical integration layer challenges required for a successful realization of cross-component business processes in a consistent way.

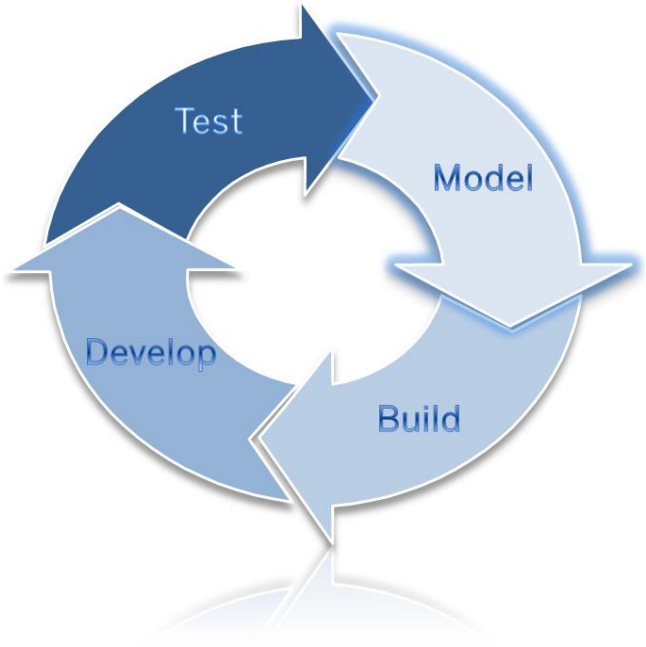
SAP Process Object Builder (POB) is the tool that enables the rapid development of POTs.

2.2 Structure of this Guide

The sequence in which the topics are described in this document follows the life cycle phases of a POT. As the document progresses into the later phases of the POT life cycle, several topics are described in detail and refer to other parts of the document.

The development life cycle of a POT involves several high-level phases like modeling, wizard-based generation, implementation of code slots as well as testing and refactoring: For a better orientation, the main chapters of this guide are tagged with the corresponding phase of this life cycle perspective.

The following figure shows the phases as referred to in this guide:



POT Development Life Cycle

- **Model** – identification and modeling of the POT in accordance with SAP SOA methodology
- **Build** – building the POT in a step-by-step approach, guided by the POB wizards
- **Develop** – developing the code slots that are provided in the generated POT artifacts to complete the implementation of the POT
- **Test** – testing the POT

Testing a POT could result in reworking a POT in one or more phases in order to refine the functionality of the POT.

2.3 System Prerequisites for POB and POT

The prerequisites described in all the sections below apply to all versions of SAP Process Object Builder 2.0.

2.3.1 General Prerequisites

A POT primarily contains ABAP artifacts and runs in ABAP server. The minimum required SAP NetWeaver™ release is SAP NetWeaver™ ABAP 7.50 (minimum SP-Level for NetWeaver™ 7.50 is SP04).

The Business Suite Foundation Layer (SWC BS_FND) is required for POB itself as well as for running a POT.

In In the SIW component, a dedicated landscape for POB must be specified (see the "Tools Used" section in the POB Application Help). SIW is part of the SAP NetWeaver™ Platform.

A POT can run independently of any other application platform component (in particular SWC FSAPPL).

A POT is completely independent from POB and does NOT require the POB at runtime.

ESR requirements:

POB requires an ESR (Enterprise Services Repository) that contains active storage locations for services artifacts (SWCVs, namespaces, folders). More information on the required releases can be found in the POB Installation and Upgrade Information.

The POT SWCV must not be locked.

Services of the back-end application SWCVs must also be present in ESR. On the corresponding ABAP back end system for a POT, a connection to ESR must be set up and a matching landscape definition must exist in SIW customizing.

Tools used:

The following list names all the other tools that are required by a POT.

- Extensible Objects Framework (Package: XO_FRAMEWORK)
- Process Step Journal (Package: FS_PSJ)
- Error and Conflict Handler (Package: FS_ECH)
- Process Observer (Package: BS_POC)
- Floor Plan Manager (Package. APB_FPM)
- Outbound Agent Framework (Package: FS_OAF)
- Change Notification Service (Package: FS_CNS)
- Services Library (Package: FS_SERVICE_LIBRARY)
- Business Rules Framework (Package: SFDT)

For more information on the tools and how they are used with a POT see the "Tools Used" section of the POB application help.

2.3.2 Prerequisites for Back-End Services

Only back-end service operations that meet the following requirements can be used by SAP Process Object Builder:

- The operation is defined in ESR.
- For custom data type enhancements done in ESR, the SPROXY structures must be generated manually (outside POB). (See POT Enhancement Guide for more details)
- The attribute structure message types of the operation should contain a convenient identification for a message instance in order to map asynchronous outbound request messages to the corresponding confirmation messages.

If enterprise services provided by SAP are used as back end services, this requirement is mostly fulfilled because one of the following standard data types is always part of the payload in the signature of any operation:

- `BusinessDocumentMessageHeader`
- `BasicBusinessDocumentMessageHeader`

If these data types are not present in the signatures of the back end operations, you need to implement a code slot to map the asynchronous requests to their corresponding responses. If you have an attribute to identify a message instance, this will ease up the implementation of this code slot.

2.4 Deployment Options and POT Shipment

As already mentioned in section 2.3, POB only generates artifacts of a POT and these artifacts are completely independent of POB itself. Therefore, from a deployment perspective, a POT can run completely independent of POB in a separate instance. A POT consist of the following runtime artifacts:

- ABAP Content
- ESR Content

In addition, the following optional artifacts might be required to operate a POT:

- BPM Content
- WebDynpro ABAP FPM Content
- Portal Content

The same applies for a POT software shipment.

Please note that SAP does not ship any productive POTs along with the POB itself.

> **POT Shipment**
>
> There are no process object types (POTs) included in the shipment of SAP Process Object Builder (POB)!

2.5 Sample Business Process: Travel Booking

To help you to better understand each individual aspect/concept of modeling, building and developing a POT implementation, this document follows one sample business process throughout.

The sample business process is called “Travel Booking for a Business Trip”. Throughout this guide, the phrase ‘sample business process’ always refers to this travel booking process. The explanations and figures that relate to this process cover all the aspects/concepts and artifacts that are explained in the guide. They always follow the more abstract explanation of the corresponding aspect/concept or artifact.

Please note that the process is only intended to explain the concepts described in this guide and therefore is in no way complete.

The following section gives a brief explanation and overview of the process. The details on how to implement this process using a POT (from modeling to testing) are progressively illustrated in the subsequent chapters of this guide.

Process Overview

A company employee needs to go on a business trip. To do so, the employee needs to book a flight ticket and up to two shuttle transfers, one from his/her home address to the airport of departure, and another one from the airport of arrival to the destination address.

The objective of the sample process is to provide the employee with an easy-to-use self-service for booking the flight ticket and the shuttle transfers.

This simplified business process is shown in Figure 3:

Figure 3: Travel Booking

For the sake of simplicity, reservations for hotels and rental cars and so on, are not part of this sample business process.

An instance of the sample business process could be as follows:

An Employee residing near Frankfurt (Germany) needs to go on a business trip to Manhattan, New York (U.S.):

- First, the employee selects a suitable flight from Frankfurt Airport (Home Airport) to JFK Airport, New York (Destination Airport).
- The employee then books a shuttle from his/her home address to Frankfurt Airport and from JFK Airport to Manhattan.

2.6 How to Read this Document?

Part I – High-level Architecture & Design Concepts.

For architects, integration architects/experts - Modeling, high-level architecture, deployment and shipment (with and without BPM), high level design considerations and their implications like SWCV cut, separation of modeling and production, namespaces, package concept, and global data types (GDTs).

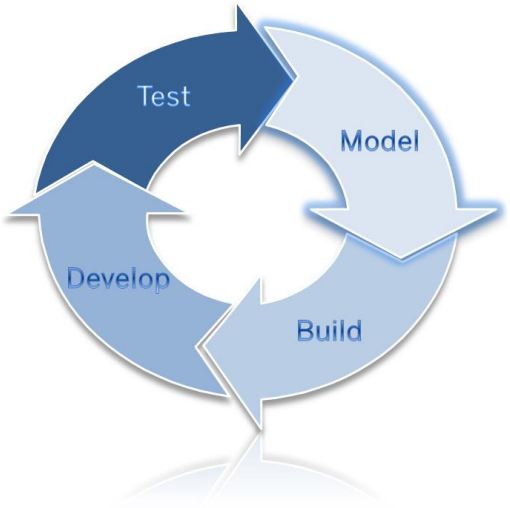
Detailed modeling aspects for process object cut (details nodes, process control constraints, checks) as well as modeling and design of editing UIs.

Part II – Detailed Design & Implementation

For architects / developers - specification steps and their implications, generation of services and consumed back-end services (roughly covering specification and generation wizard, but NOT just step by step explanation), details of generated artifacts (tables, classes, helpers) and detailed information on code slot implementation.

Part I – High-level Architecture & Design Concepts

Before a POT can be generated using POB, there are several aspects that need to be addressed and decided by development architects to complete the high level architecture of the POT. Such decisions are often considered as simple input parameters on the POB wizard screens. However, these parameters must be considered carefully with complete understanding of their implications. This part of the developer's guide briefly describes the context and the activities that must typically be performed to identify and cut a process object. It also discusses in detail the options for and the impact of several high-level architecture-relevant parameters that are specified as settings for the POT in the Builder.



3 POT Definition

"You can use an eraser on the drafting table or a sledge hammer on the construction site."

- Frank Lloyd Wright

SAP Process Object Builder extensively incorporates a Model-Driven Software Development (MDSD) approach. We therefore strongly recommend that you follow a robust modeling-based approach to define a POT before starting with the POB and generating the POT artifacts using the POB wizards. Detailed modeling and a constant review of the models avoids frequent re-work on the actual POT.

3.1 Model-Driven Software Development (MDSD)

In the POB context, the MDSD approach means that the model of a POT is the heart of the whole software development life cycle: First of all, the model is based on a well-defined metamodel that is defined by SAP. POB not only offers wizards to construct the model of a POT but also uses this model to generate the corresponding software artifacts. It maintains all the metadata related to a POT in exact conformance to the POT model. The generated software artifacts, implementation code and monitors also comply with a standard architecture and programming model.

Any changes to the model result in regeneration and/or deletion of the POT software artifacts, according to the changed metadata and processing instructions. The regeneration steps are robustly built so as not to touch any non-POT-specific artifacts and custom coding in the code slots that are generated for this purpose. The generation and regeneration also takes care of special artifacts like database tables.

POB also supports POT versioning in order to manage changes across different releases of a POT.

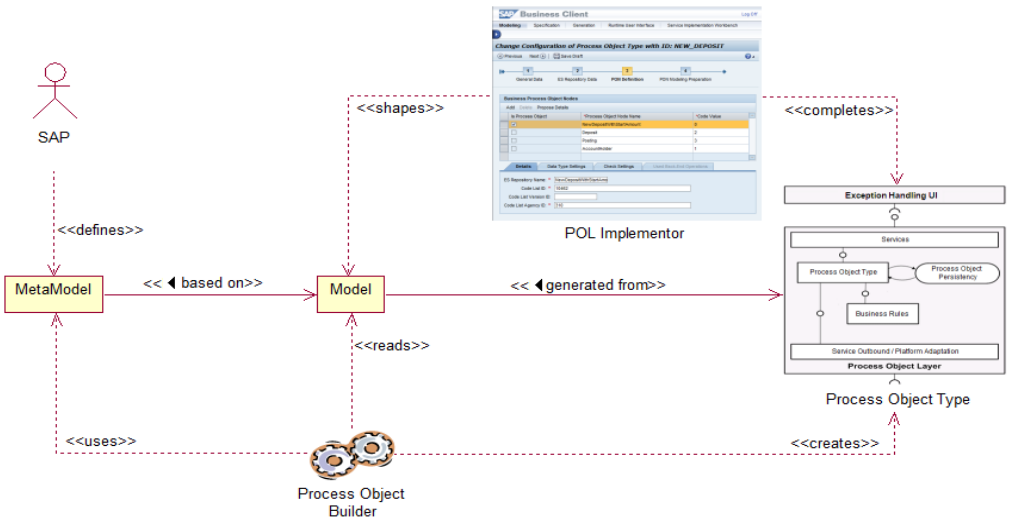


Figure 4: Model-Driven Software Development in the POB/POT Context

Model Driven Software Development (MDSD) of a POT

A POT model is much more than just a documentation of software. It is, in fact, the central artifact that drives the entire software development life cycle of the POT.

3.1.1 Versioning of a POT

As already mentioned in the previous section, POB maintains all the metadata related to a POT. This metadata is called the POT configuration. The version concept defines two types of versions: The metadata version and the configuration version:

Metadata Version

This version reflects the support package (SP) level of the POB that has been used to create the POT configuration. The metadata version is required to support the migration of POT configurations that have been created with a lower POB SP to a higher POB SP.

Configuration Version

Each POT configuration is assigned a configuration version, starting with version number 01. This enables you to create multiple versions of the same POT configuration. The version number is included in most of the generated artifact names.
This versioning of POT configurations is useful in cases where an existing POT evolves over the time and requires new features: Instead of changing and re-generating the existing POT to reflect the new features, a new configuration version can be drawn. This leads to a completely new set of artifacts, which are generated based on the next higher version number. The existing POT remains unchanged.
Instead of drawing a new version for a POT configuration, you can also copy an existing configuration. This leads to a new POT configuration, again starting with version number 01. In this case, the existing POT also remains unchanged.

3.2 Identification of a POT

The detailed process of identifying a POT can vary from one software development project to the other and is therefore not part of this document. However, the sections give you an overview of the activities that should always be part of the POT definition process.

3.2.1 Process Definition

A valid business process driven by an outside-in approach must clearly identify the requirements for a POT in the given business context (see section 2.1 above). This means that the business process must require functionality that spans across several business objects. The requirement definition can also specify whether the business process (or parts thereof) should be executed in an automated way using BPM.

Process Definition: Travel Booking

The process requirement definition for the sample business process (see section 2.5 above) contains the following requirements:

- Basic information related to traveler and travel dates must be sufficient to trigger the process.
- The process steps should be automated by the system as far as possible.
- Shuttle bookings must be consistent with the corresponding flights bookings.
- Travel expenses are limited to a specific amount per business trip.

  However, after the first two travels in a calendar year a traveler is eligible for a double expense limit. The eligibility limit can also be relaxed in special cases by the traveler's manager.

These requirements meet the demands for a process object in the following aspects:

- The process steps require smaller beautified views of the actual back-end business objects involved in the process (traveler, flight, flight booking and shuttle booking) that suite the travel booking process.
- There is a requirement for automated process step execution with minimum user interaction.
- There is a sequence of process steps that should execute services of different application components in a consistent manner: The flight booking application will most differ from the shuttle booking application. However, the times for which shuttles are booked must be consistent with the times of the corresponding flights.
- Finally, there are requirements for business rules at process level. The eligibility for double expense limit and its possible relaxation is a check that relates to the process itself and does not necessarily relate to any of the back-end business objects.

The sample business process therefore qualifies for the introduction of a POT to implement this process.

Figure 5: shows a BPMN model of the requirements listed above together with a broad scope of the corresponding POT:

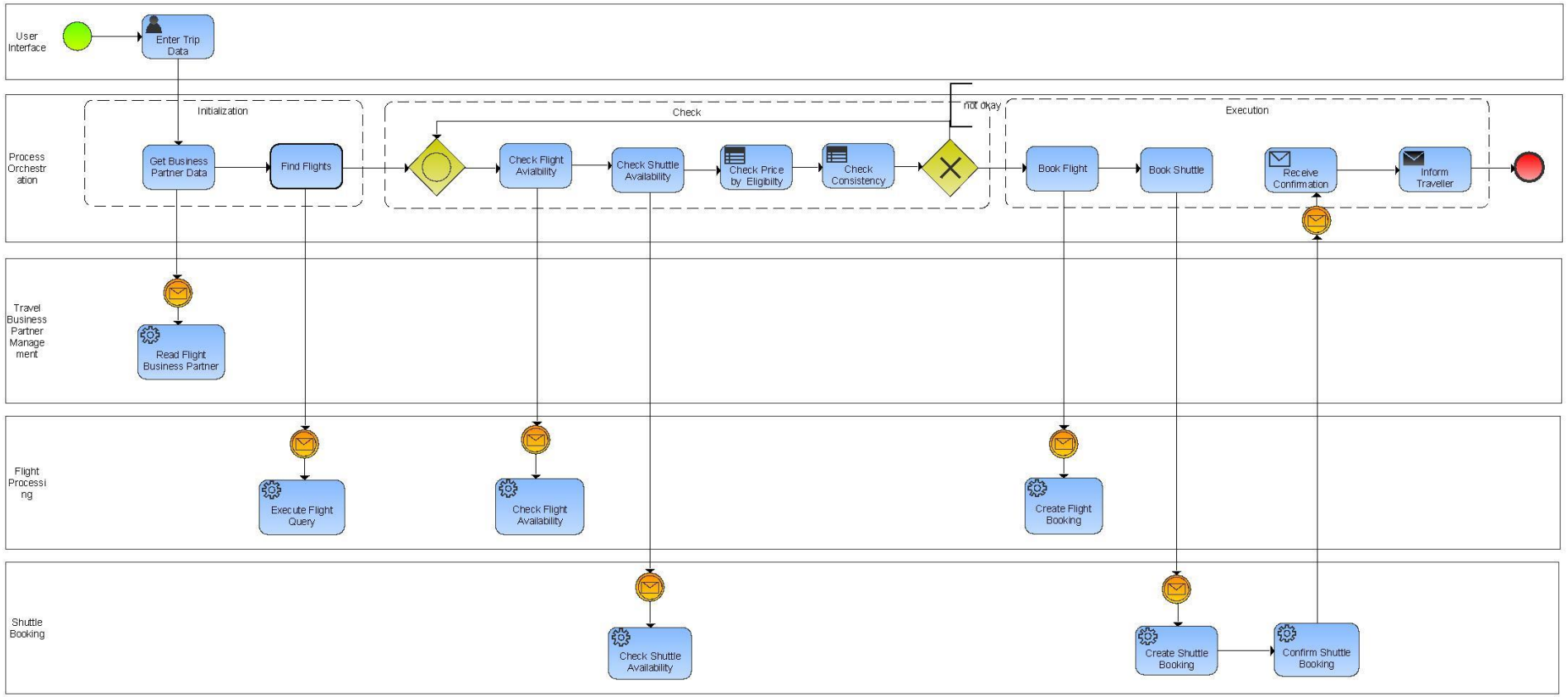


Figure 5: BPMN Model of the Travel Booking Process

3.2.2 Investigation of Back-End Business Objects and Services

The next activity for identifying a POT is to identify all the fine-granular back-end services that the business process requires and to test these services for all required variants (input combinations). The general prerequisites for building a POT must also be tested. Any gap identified with respect to back-end services must be addressed in the back end (for example, by creating new services or new service versions that meet the requirements of the identified POT) before proceeding with the POT definition. The feasibility of compounding fine-granular services for a specific process can also be tested during this activity using well-prepared test suites in a web services testing tool (like SoapUI).

3.3 Modeling a POT

Once the requirements and pre-requisites have been clarified, the POT itself should be modeled at high level by defining the process component to which it belongs, the deployment unit and software component etc. Then the POT object model should be modeled in detail with the process object root node and all the process object nodes.

The overall methodology that is recommended for POT modeling is SAP SOA Modeling Methodology (see section 1.2, SAP SOA Modeling Methodology). All methodology and meta-model entities referred to in the following sections use the terminology and methodology described in this document.

3.3.1 POT Modeling using SAP SOA Methodology

SAP SOA modeling methodology follows a harmonized enterprise model approach that reduces redundancies of SOA services by assigning services to a business object. The business object itself is assigned to exactly one process component and process components are grouped into deployment units. A deployment unit is a separately deployable entity in a shippable software component.

Figure 6 below shows a meta-model for SAP SOA modeling methodology:

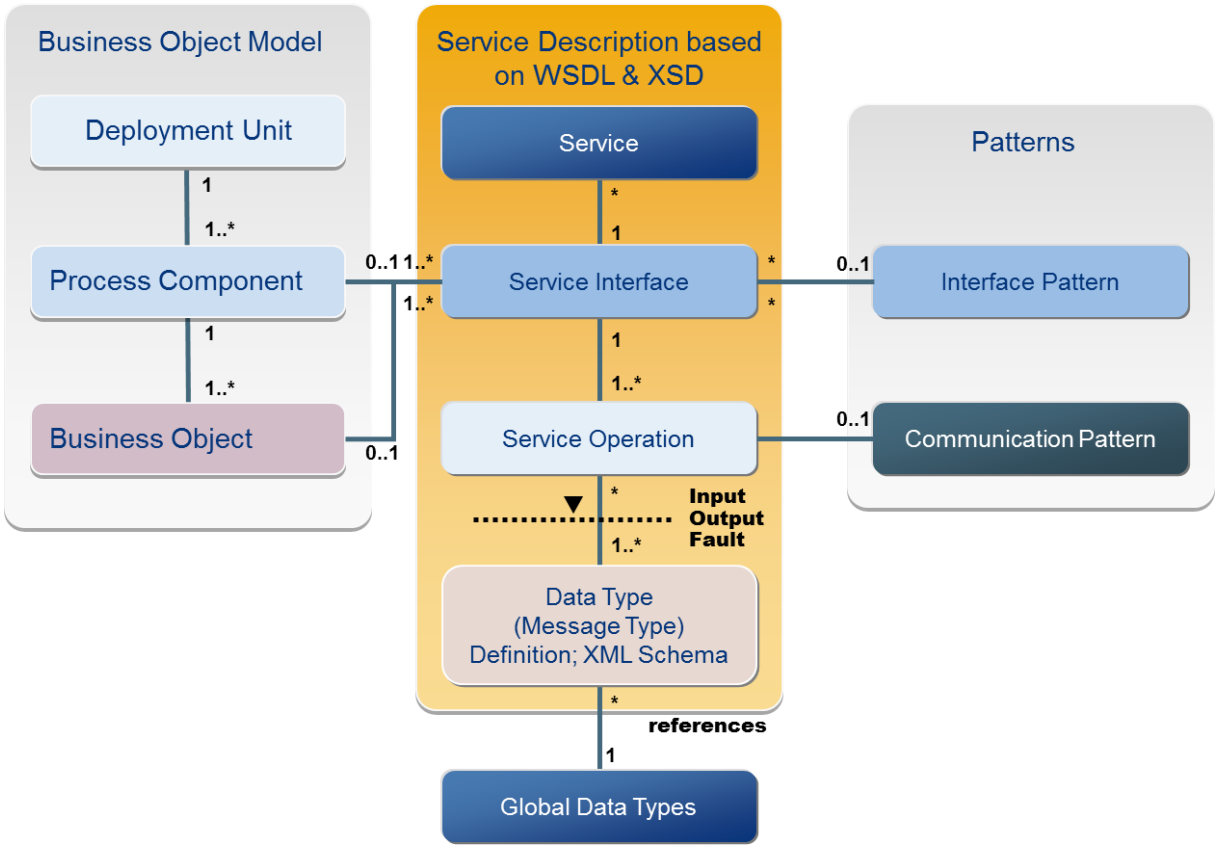


Figure 6: SAP SOA Modeling Methodology Metamodel

POB requires the **process component** that the POT is assigned to as an input parameter at design time. The process component name must be unique for each POT. This means that there is a 1:1 relationship between a POT and the meta-model entity process component in SAP SOA Modeling Methodology. Although, theoretically, the methodology allows for multiple business objects (in this case POTs) to be present in a single process component, a design decision has been made for POB to limit this to a 1:1 relationship. (An exception is made for the different POT versions, which means that different versions of a POT can still be in one process component). The restriction is due to the fact that a process component in business suite applications is only a metamodel entity and has no corresponding shippable entity in ESR. As the POB metamodel heavily relies on ESR entities, it is easier to handle generated artifacts that relate to concrete ESR entities.

A **deployment unit** is another metamodel entity in SAP SOA Modeling Methodology that primarily does not affect anything that POB does at design time (during POT generation). However, assigning process components (and also POTs, due to the 1:1 relationship) to a deployment unit is something that development and integration architects need to consider right at the beginning of the POT definition process.

The farthest boundary of a deployment unit is a **software component (SWC)** that belongs to a unit of shipment. Therefore, common factors such as shipment cycles, landscape optimization and deployment options that influence the cut of software component versions (SWCVs) must be considered when determining the right SWC for a POT (and, consequently, the appropriate modeling entities).

Figure 7 shows an overview of the various SAP SOA entities for the sample process:

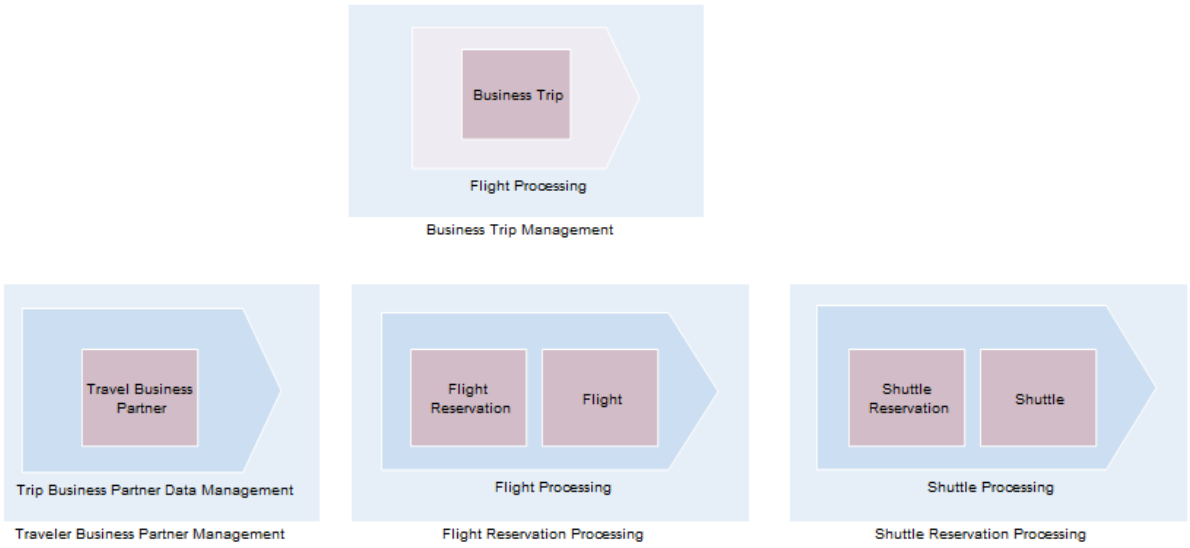


Figure 7: Business Object Map for Sample Process (SAP SOA Modeling Methodology)

Other models that integration architects should define include the **integration scenario model** and the **process component interaction models**. These models are necessary to provide a complete understanding of the enterprise services that must be provided and used for the POT.

Once process component and deployment unit have been defined and a decision on the SWC cut has been made, POB only requires the SWC to be assigned to an appropriate application component in SAP Application Component Hierarchy. The SWCV must also be suitably reflected in ESR (via SLD) and an ESR Namespace must be chosen and available in ESR.

The choice of the **ESR namespace** is also significant for POT development. There are different conventions for defining the ESR namespace. One such convention could be, for example, to choose a namespace that has a 1:1 relationship to the deployment unit of the POT. Following this convention, the ESR namespace for the travel booking process, could be as follows:

- http://test.sap.com/xi/BusinessTripManagement

This is a generic way of naming the ESR Namespace, because in the future there could be other POTs in the same SWC (in our example, this could include rental car booking, hotel booking and so on) that can share this ESR namespace. Choosing a single ESR Namespace for multiple POTs of the same SWC will enable you to reuse back-end proxy objects like GDTs, which are generated only once per ESR namespace in an ABAP back end. However, choosing one ESR Namespace also means that all the ESR objects of multiple POTs exist in the same ESR Namespace. In this case, the ESR objects of several POTs can be separated logically by using ESR folders.

Another way of cutting ESR namespaces could be to have specific names that follow the scope of a POT, like

- http://test.sap.com/xi/FlightAndShuttleBookingProcessing
- http://test.sap.com/xi/CarBookingProcessing
- http://test.sap.com/xi/HotelBookingProcessing

This way of cutting ESR Namespaces would explicitly rule out any reuse of common SPROXY artifacts. However, it also means that only ESR objects related to one POT exist in one specific ESR Namespace.

Based on these aspects, an informed decision must be made on whether to create multiple ESR namespaces in one SWC for different POTs as this decision, for all practical purposes, leads to a one-way path for all subsequent development of POTs within that SWC.

> **Architectural Decisions**
>
> Decisions on the process component, software component and the ESR namespace of a POT have long-lasting consequences. It is not recommended to change them again during the POT development process.

3.3.2 POT Object Model

The next activity in modeling a POT is the modeling of the POT object model itself, which means that the POT is modeled in detail together with its node structure. This also includes node cardinalities and cross-references (if any) between the process object nodes. This model of a POT is later on entered in POB and completed in ESR (to enable POB to work with the model). However, as ESR is not a tool intended for process modeling, we strongly recommend to model a POT upfront in another modeling tool with a suitable metamodel/method definition (SAP Sybase Power Designer, MS Visio, ARIS or others) to be able to review and finalize the model before you start modeling in POB.

Figure 8 shows the metamodel for a POT model:

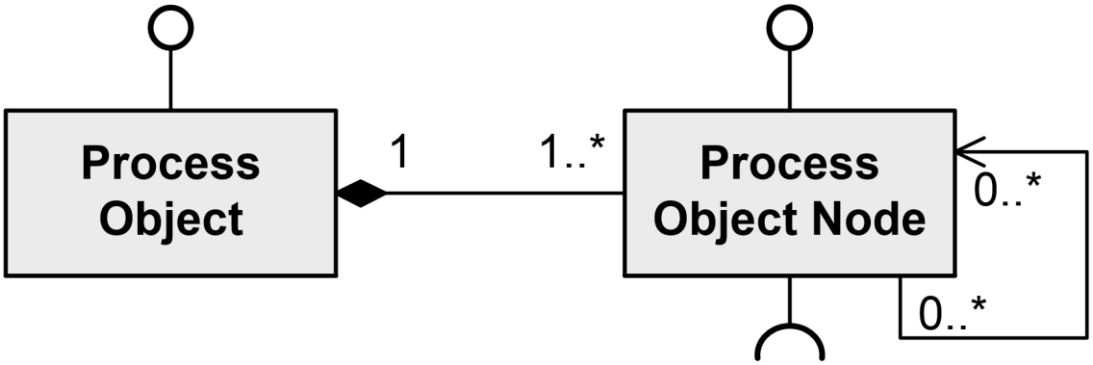


Figure 8: Metamodel of a POT

A POT consists of a root node with the same name as the POT itself. This root node is also generally referred to as the BPO (business process object) or PO (process object).

Under the root node, the POT contains one or more subnodes in a composition relationship. Each subnode is also referred to as BPON (business process object node) or simply PON (process object node). BPONs can have references to each other within a BPO. These references are also called cross-references. The relationship type for cross-references is *association*. (In SAP Business Object modeling methodology, the relationship type is further qualified as *association for navigation*, as the associations are used to navigate from one BPON to another for accessing data).

Following this metamodel, Figure 9 shows the POT object model for the sample process (M^{-1} Model).

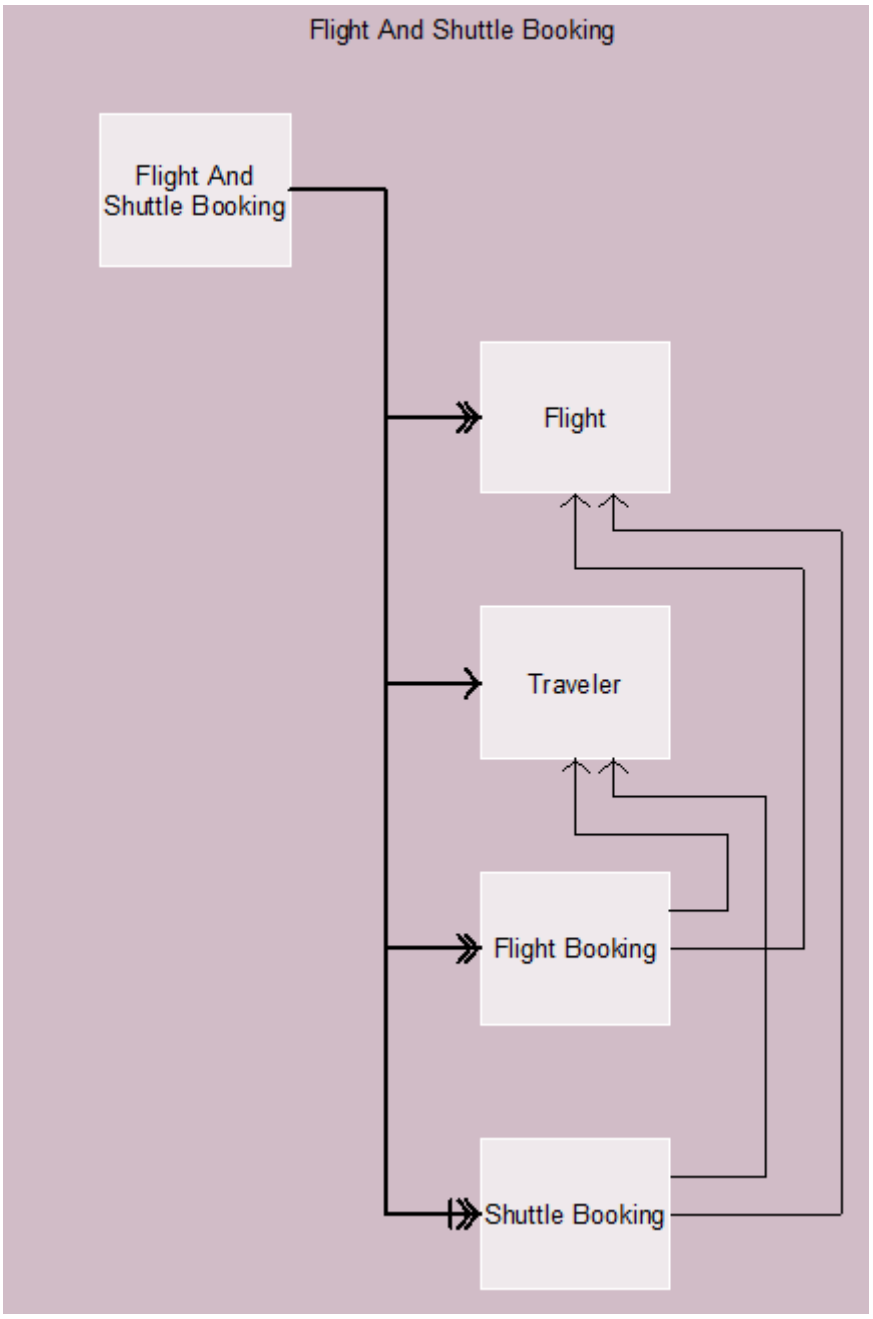


Figure 9: Preliminary Object Model for `FlightAndShuttleBooking` POT

3.3.3 'Beautified View' Concept

One objective of a POT is to compound fine-granular services of the back-end applications into coarse-granular services that are cut according to the business process and the UIs of the process that consume these services. It is therefore very important to consider the structural details of the BPO and BPON nodes. Typically, the structures of the services that a back-end business object exposes for its manipulation are process-agnostic. As a consequence, these structures are mostly huge as they must meet the requirements of several business processes and also expose the full functional capabilities of the business object. On the other hand, a business process that a specific POT is designed for has specific requirements that relate to certain parts of the whole functionality of the business object. Therefore, a POT can build its own view of the back-end

business object with a reduced structure that best suits the business process. This reduced view is also called the ‘beautified view’.

Typically, we recommend to have a beautified view on a single back-end business object in the corresponding BPON of the POT. (The reasons for such a 1:1 relationship will be introduced in later sections.)

In addition to the BPONs, the root node itself can also have specific attributes that are relevant for processing multiple BPONs.

As a rule, a beautified view must be designed to incorporate as few attributes as possible (those which are required for the process as user input/ for maintenance). In addition to this basic rule, the attribute structure of a BPON or BPO will also influence and is influenced by the following criteria:

- Attributes that are required for other BPONs in case associations between several BPONs exist in the model (also known as cross-references)
- Attributes used in checks/business rules that control the execution of the business process (called process control constraints (PCC) and private details of a BPON)
- Attributes that can be maintained by a well-defined user role that handles exceptions of the process (also known as public details of a BPON)

Any attributes of the back-end service operation signature that can be defaulted or derived from other attributes of the BPON should be left out of the BPON structures. In order to structure the attributes described above, POB inserts several predefined nodes into a BPON model that will be described in the next section.

Data structures (message intermediate data types) that are used to model BPON details can be flattened and re-structured for simplicity and need not necessarily reflect their original structure as in the corresponding back-end business object model.

Beautified View

The node structure of a BPO/BPON should have as few attributes as necessary for the process and as many as required for the successful execution of a POT.

3.3.4 POT Object Model Template

POB follows a standardized business object model template for completing the POT model. While the designer of a POT decides on the structure of BPO / BPONs with the attributes that are required from a business process perspective, POB in addition adds some standardized nodes that later on address the generic POT processing features. The attributes that are related to the back-end business object and relevant for a POT must always to be added in a predefined standard node called `Details`, which is available for each node of the POT.

The following nodes are added by POB for all nodes of a POT in a defined hierarchy:

- `Administrative Data` (Always added by POB)
- `Process Control Constraints` (optional, to be specified in POB if required)

Figure 10 shows the abstract POT Model Template:

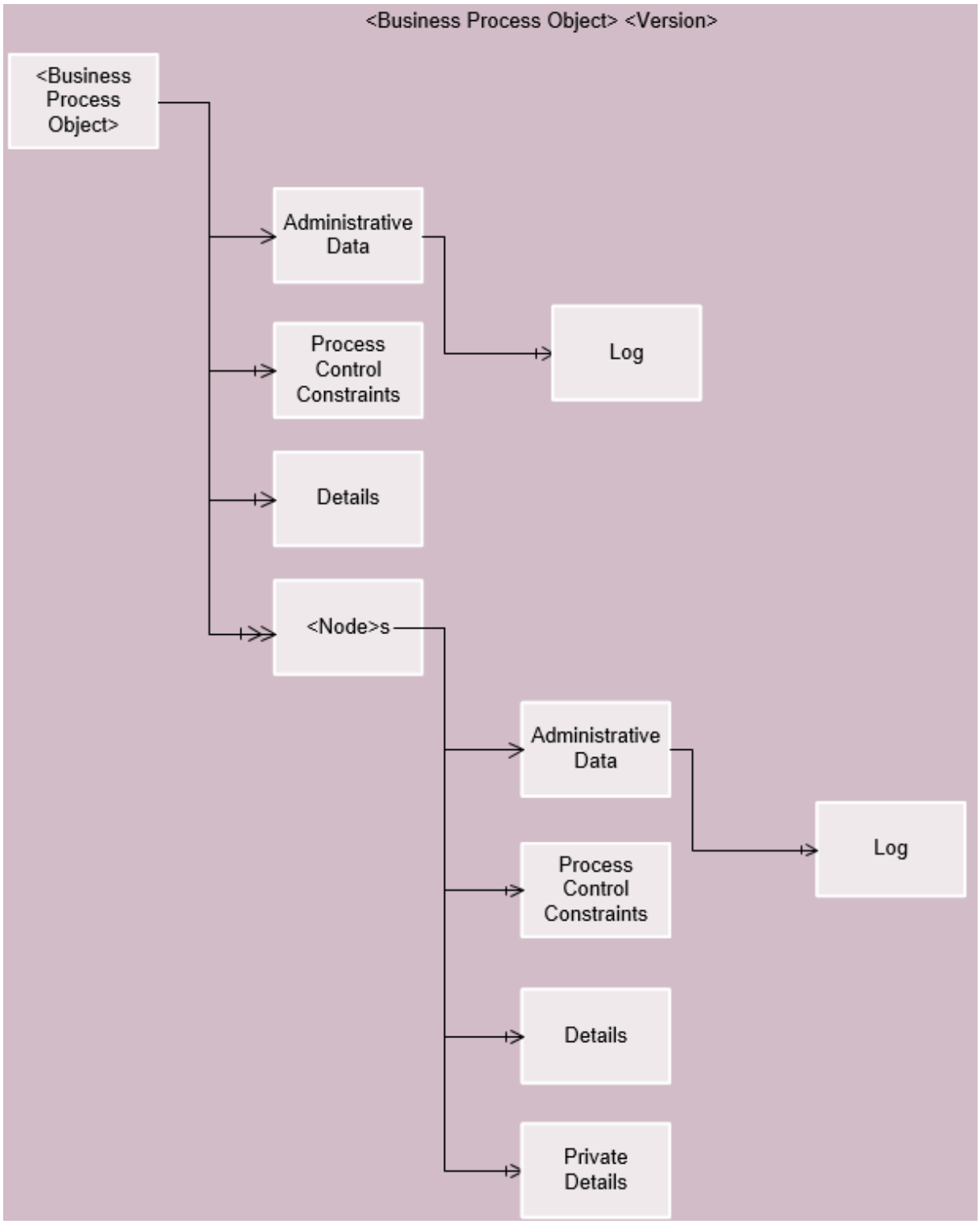


Figure 10: Abstract POT Model Template

The names for `<Business Process Object>` and `<Node>` can be chosen following the guidelines and naming conventions as specified by SAP Modeling Methodology (see section 1.2, SOA Methodology @ SAP). `<Node>s` represents the BPON types (there can be one or more in a BPO) and the indicated cardinality is the cardinality of the BPON instance.

Following this abstract POT model, Figure 11 shows the object model of POT for the sample business process:

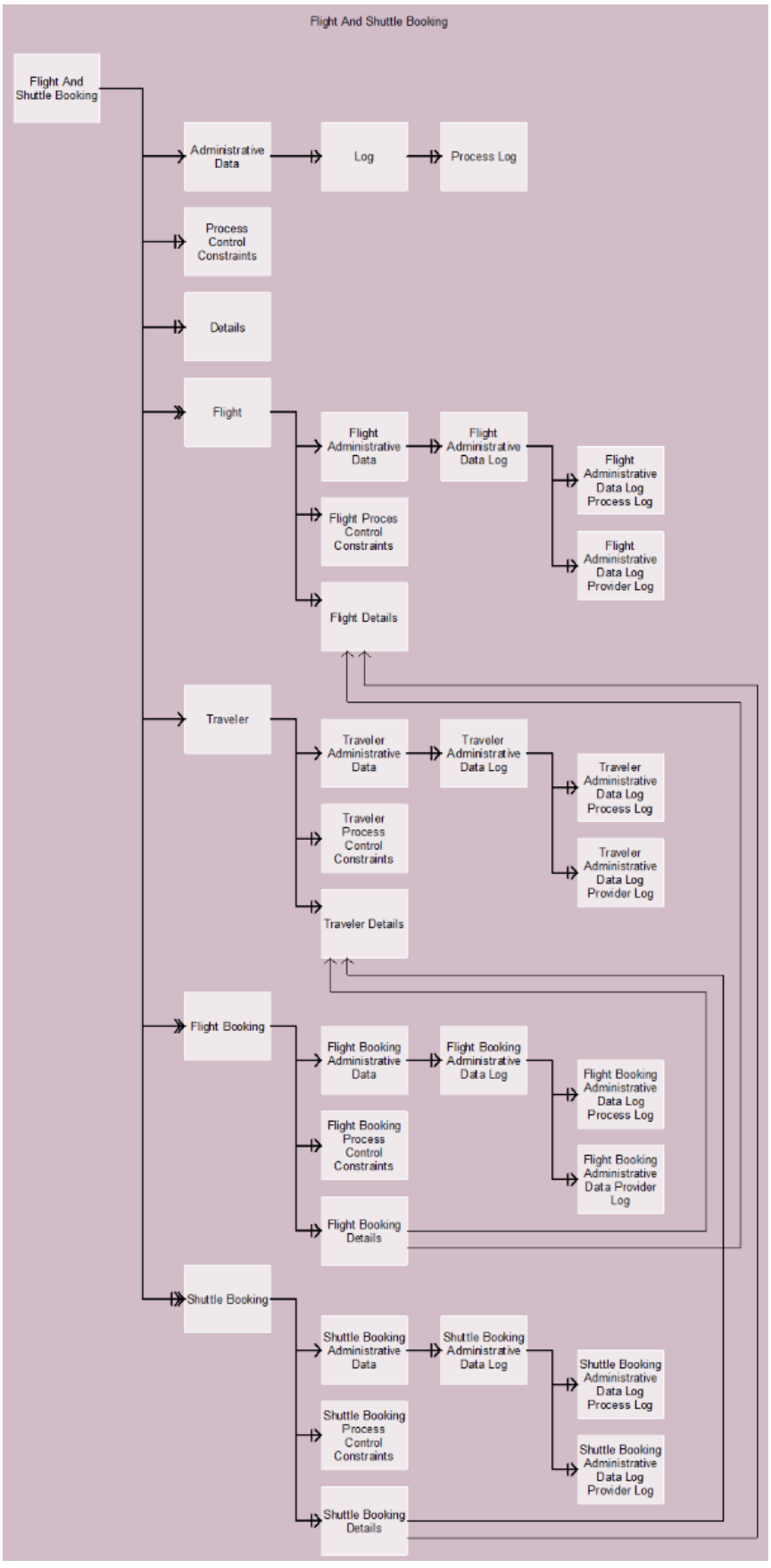


Figure 11: Detailed Object Model for sample POT `FlightAndShuttleBooking`

3.3.5 Predefined Nodes

This section describes in detail the attribute structure of the predefined nodes that are introduced into a POT model by POB.

In general, POB introduces the following elements into the service signatures of all the service operations that are generated for POT services:

- `MessageHeader`
  typed with AGDT `BusinessDocumentMessageHeader`
- `Log`
  typed with AGDT `Log_V1`

Each node of a POT (including the root node) has a unique identifier:

- Element `UUID`
  typed with GDT `UUID`

For the detailed description of the GDTs and Aggregated GDTs, please refer to the SAP GDT Catalog.

3.3.5.1 Administrative Data

The node `Administrative Data` of the root (BPO) has the following structure:

Element Name: (A)GDT / (M)IDT	Remark
`ReferenceID`: `BusinessTransactionDocumentID`	Externally assigned ID as an alternative identification for the BPO instance. The ReferenceID is not required to be unique.
`BusinessProcessChainAssignment`: `OPTIONAL_BusProcChnAssgmtFSElmnts`	BPCA: Contains elements for unique identification of the process chain that the POT instance is assigned to together with a `BankingBusinessTransactionTypeCode` (the coded representation of the process type). At runtime, if the element `UUID` is not assigned by the consumer of a POT service, it is automatically assigned by the POT implementation. Once a BPCA is assigned to a BPO instance, the BPCA cannot be changed anymore.
`ChangeStateID`: `ChangeStateID`	Represents the latest change state of the POT instance. This element is used to implement the optimistic locking feature in the stateless enterprise services of a POT.

Element Name: (A)GDT / (M)IDT	Remark
`ObjectNodeTypeCode: ObjectNodeTypeCode`	Coded representation of each BPON including the BPO (root node). Once the POT design and governance process has been finalized, the integration architects assign the `ObjectNodeTypeCode` and also ensure its uniqueness. The context of `ObjectTypeCode` (coded representation of the business objects) is encoded in the `ObjectNodeTypeCode`. The `ObjectNodeTypeCode` values that are assigned to the BPONs in POB (at design time) should not be changed afterwards. A change requires regeneration.
`BusinessProcessObjectNodeStatusCode: BusinessProcessObjectNodeStatusCode`	Each BPON has a status, which can be changed by executing a POT service operation. The overall status of the BPO is calculated taking the status of the root node itself and the BPON statuses into account. (More details follow in the Status Model section).
`Log: <Business Process Object><Application>_BPO_Log`	For the BPO (root node), this element is further qualified by a subnode called `ProcessLog`: It contains those messages related to the processing of the POT instance (*Check* phase, see the Phase Model section for details) that are relevant for the root node. The logs are cleared when the status of the process object is set to *Unchecked*.

The node `Administrative Data` as used in the individual BPONs has a slightly different structure:

Element Name: (A)GDT / (M)IDT	Remark
`ReferenceID: BusinessTransactionDocumentID`	Externally assigned ID as an alternative identification for the BPON instance. The ReferenceID is not required to be unique.
`CreationInstructionCode: RelatedObjectExistenceAssumption-NodeCreationInstructionCode`	This element is only available in the input signature of the BPON `Create` operation. It instructs the POT on how to fill BPON data when a service call is made to the back end in the *Create* phase. As each BPON represents the beautified view of one back-end object, several options can be specified (for example to create the BPON with the data provided by the service or to overwrite the data provided in the POT service with the data retrieved by back-end service call during the *Create* phase). For details, see the Phase Model section.

Element Name: (A)GDT / (M)IDT	Remark
`ProviderID:` `<GDT to be assigned during modeling>`	Identifier of the related back-end business object for the BPON. The GDT of the back-end business object identifier has to be assigned during POT modeling in POB. The element must be filled on successful confirmation of a back-end service call in the *Execute* phase using a custom code slot implementation. This enables the POT to react accordingly in the status transitions.
`ObjectNodeTypeCode:` `ObjectNodeTypeCode`	Same as for BPO element `ObjectNodeTypeCode` (see previous table)
`BusinessProcessObjectNodeStatusCode:` `BusinessProcessObjectNodeStatusCode`	Same as for BPO element `BusinessProcessObjectNodeStatusCode` (see previous table)

`Log:`
`<Business Process Object><Application>_BPON_Log`
	At BPON level, this element has two subnodes:
	`ProcessLog`
	Contains those messages related to the processing of the POT instance (*Check* phase, see the Phase Model section for details) that are relevant for the BPON.
	`ProviderLog`
	Contains messages from the back-end system for backend calls made by the POT during the *Create*, *Check* or *Execute* phases.

3.3.6 POT Object Model in ESR

POB uses ESR as the modeling environment to capture the complete object model of a POT with all its elements and attributes. There is no graphical modeling environment in ESR. Therefore, all the relevant metadata of a POT are captured using an existing entity in ESR, which is the data type.

ESR is used as the POT modeling tool because during POT generation POB works extensively with ESR and can programmatically access the model of a POT just as it would access other data types. Furthermore, having the model as well as the implemented data types/services of a POT in a single tool enables the MDSD approach.

In order to capture the POT object model in ESR, POB also generates the following special data types that define the metadata of a POT object model:

3.3.6.1 Cardinalities

`M_<Business Process Object>_BPO_BPON_Cardinalities`

In this data type, the cardinalities of each BPON in a BPO must be specified using the occurrence property. The specified occurrences are used by the generated POT artifacts to check the cardinalities of a specific POT instance at runtime and also in the *Check* phase.

Details of the POT runtime status, phase and programming model follow in the next chapter.

3.3.6.2 ProviderIDs

`M_<Business Process Object>_BPO_BPON_ProviderIDs`

POB proposes the generic GDT `BusinessTransactionDocumentID` for the element `ProviderID` (see section 3.3.5.1 above). This data type can be used to specify the actual identifiers of the related back-end business objects.

3.3.6.3 Details

`M_<Business Process Object>_<Business Process Object Node>_Details`

The `Details` node is optional for all nodes (BPO and BPON). In this node, the beautified views of related back end business objects are modeled. The elements of the node are freely definable. However, at BPON level, most of the elements of this node can typically be found in the back-end business operations that are associated with the BPON.

In accordance with the concepts discussed in section 3.3.3 above, the following modeling options are also possible:

- For the sake of beautification, the deep structures of the back-end operations can be flattened out to define the element structures.
- Elements of this node can also be deep structured using table-like nodes.

In addition to the beautified view, the relationships to other BPONs of the POT (if any) must also be included in the `Details` node using the data type prescribed by POB for cross-references. Cross-references can be modeled in the `Details` node or in the `ProcessControlConstraints` node (see section 3.3.6.5). The data type for cross-references is `M_<Business Process Object>_CrossReference`. The cardinalities of the modeled cross-references are checked at runtime by the generated POT coding artifacts in the *Check* phase of the POT. (Details of the POT runtime status, phase and programming model will follow in the next chapter.) This data type has the following elements:

- `UUID:UUID`

  The unique identifier of the BPON to which the currently modeled BPON has a cross-reference (will be assigned by POT implementation).

- `Reference: BusinessTransactionDocumentID`

  The ReferenceID of the BPON to which the currently modeled BPON has a cross-reference (the ReferenceID might not be unique).

  The identifier of the related back-end business object for the BPON to which the currently modeled BPON has a cross-reference.

- `TypeCode: ObjectNodeTypeCode`

  A coded representation of the referenced BPON type.

3.3.6.4 Private Details

`M_<Business Process Object>_<Business Process Object Node>_PrivateDetails`

The `PrivateDetails` node is optional for all nodes (BPONs). The elements of this node are freely definable. In contrast to the `Details` node, the `PrivateDetails` node will not be exposed by the POT service interfaces.

Private Details should only be modeled if a BPON requires additional attributes which will only be used for internal POT processing and should not be exposed to the consumer of the POT.

In ESR, the elements of this node are modeled in the same way as those of the `Details` node. These elements will then be available for POT-internal processing during all phases of a POT.

3.3.6.5 Process Control Constraints (PCCs)

`M_<Business Process Object>_<Business Process Object Node>_PCC`

The node `ProcessControlConstraints` is optional for all nodes (BPO and BPON).

This node is used to model additional attributes that are only needed during process execution by the POT itself, but the attributes are not represented in or are irrelevant for the related back-end business object of the BPON.

The elements of this node can be chosen freely as long as the naming conventions follow the service modeling guidelines and GDTs are used as data types.

For example, in the sample POT, `ProcessControlConstraints` are modeled for the Flight BPON. This PCC structure contains an element that captures the date of the planned flight for the business trip and which does not belong to the beautified views of any of the back-end business objects:

- PlannedFlightDate: `Date`

The following basic principles apply for the specification of process control constraints:

- The specified attributes must be relevant and required for processing the POT and they must not belong to any of the back-end business objects.
- The PCC must be sparingly used and they must not contain any information that is already contained in the `Details` node of a BPON.

3.3.6.6 Datatypes used for Modeling

During modeling of Details, Private Details, and PPCs it is important to consider how the data types that are used for modeling are handled when it comes to generation of the runtime ESR artifacts. The following types can be used:

- Use data type with prefix `M_<Business Process Object>_)` as prescribed by POB during modeling. During generation of the (runtime) ESR artifacts, this data type will be copied to a new type with a POT specific name. The cardinality of all elements of this type will be set to optional. This is necessary to support the work in progress handling of "incomplete" data within a POT. Consequently modeling types with the prefix M_<Business Process Object>_ are POT specific and cannot be reused across different POT models.
- Use a GDTs or other data type that does not follow the naming convention `M_<Business Process Object>.` During generation of the (runtime) ESR artifacts, this data type will be used within the ESR runtime artifacts as is. In contrast to the modeling type with prefix `M_<Business Process Object>_` these types can be used across different POT models.

Private Details are not exposed in any POT service interfaces. However, corresponding runtime ESR artifacts for Private Details are generated in order to have a corresponding SPROXY type available which is the basis for the generation of DDIC types.

The elements of Details and PCCs are also used later to build the editing UIs of the POT using WDA FPM. We highly recommend to keep the node structure as simple as possible and as detailed as absolutely required for the consumer of the POT.

3.3.7 Separation of Modeling and Productive Entities

The ESR entities that are used to define the model of a POT are not required for the productive environment of a POT. They are only required again by POB if and when the POT model changes. In this case, they are used to regenerate the remaining affected artifacts of a POT. The model data in ESR for a POT object model is not required for the runtime of the POT. Therefore, POB (early on in the Modeling Wizard) separates the modeling and productive environment of a POT.

There are several separation options:

- Same SWC and SWCV and ESR namespace for modeling and productive environments but separate folders

  In this case, there is no way to separate the delivery of modeling and productive content in ESR as folders are just logical constructs. This means that the model data of a POT is also delivered and available within the productive landscape of the POT.
  This option is mostly used for POT prototyping in order to avoid multiple namespaces in ESR.
- Same SWC and SWCV but different namespaces for modeling and productive environments

  In this case, there is a separation between modeling and productive content in ESR based on the namespaces. This option means that the modeling data of a POT is also delivered and available in the productive landscape of the POT.
- Different SWC and SWCV (and therefore different namespaces) for modeling and productive environments

  Separate delivery is straightforward. Modeling content can explicitly be excluded from delivery.
  This option is the most flexible one but it requires the creation of SWC/SWCV for modeling only (at least in ESR), and during POT generation a small additional step is required from the POT developer to ensure that the required data types (prerequisite data types like `BusinessDocumentMessageHeader` or BOCTs used in the `Details` nodes) are present in the ESR namespaces of both SWCVs.

> **Architectural Decisions**
>
> The way in which modeling and productive environments are separated must suit the delivery requirements of a POT.

> **Architectural Recommendation**
>
> It is strongly recommended to separate modeling and productive environments into different Software Component Versions (SWCVs) for a productive POT.

4 POT Programming Model

A POT that is generated by the POB is based on MDSD. The MDSD approach shortens implementation times by generating a lot of implementation code upfront and providing only minor parts of the implementation to be completed in code slots. Before implementing those code slots, it is very important that you understand the underlying predefined programming model of a POT. The detailed description of the programming model can be found in later parts of this guide. However, this understanding is also important for POT modeling, for example to verify the capabilities of a POT against the requirements of a process and to decide on the BPON cut as well as the number and type of back-end services to be assigned to a BPON in the different life cycle phases of a POT). Therefore, this section provides a high-level overview of the programming model that a POT is based on.

4.1 Basic Principles

The programming model of a POT is based on following basic principles:

4.1.1 Separation of Concerns: Process vs. Business Logic

In SAP SOA architecture methodology, the granularity at which business logic is encapsulated is the business object. A business object resides completely in one software component (deployed on a single physical back-end instance). It is responsible for the consistency of its own data, for its own life cycle phases, the phase transitions and all associated validation and consistency checks.

A process object on the other hand is the orchestrator of several back-end business objects, represented by the corresponding BPONs in the PO model. Multiple POTs could be defined for distinct processes that are required to orchestrate an overlapping set of back-end business objects. Therefore, a POT implementation must never, for example, duplicate the business object level validations/checks but only deal with process orchestration activities like sequencing the back-end objects calls, managing checks and rules (those that do not belong to the business logic of any back-end business object but to the process itself) and handling responses from individual back-end service calls.

The programming model is based on this principle and, as a consequence, the generated POT implementations are designed exclusively for dealing with the process logic.

4.1.2 Pattern for Process Flow

The current POT programming model complies with a use case in which a POT executes a process in the following pattern:

1. Initialize the process object with the data provided by the consumer and other relevant data retrieved from the back-ends
2. Check the process object for readiness to execute the process (process relevant checks only, see previous section 4.1.1)
3. Execute the process object.

The process object then triggers all the defined back-end service calls in order to complete the data changes to the respective back-end business objects.

This process flow pattern also forms the basis for the phase model of a POT (explained in later sections). In addition to the basic flow of the pattern described here, the programming model also foresees errors / conflicts during the *Execute* phase and incorporates a robust status model that provides flexibility for:

- Maintaining POT data (for example process control constraints) and reprocessing the POT. Editing UIs (to maintain the POT) are also generated as WD ABAP applications.
- Compensating individual back-end changes during cancellation of a POT, in case that some BPONs of a POT are completed successfully, but the overall process object execution fails.

4.1.3 Multiple Consumption Options

A POT can be consumed in multiple ways in a system landscape. Depending on the consumption option, different interfaces could be required from a POT.

The following are the main consumption options:

- A 'start UI' captures the most relevant data of the process, and, once UI-level data validation (formats, allowed values of individual fields etc.) is completed, a BPM process instance is created and the rest of the flow is automated using BPM.
- An 'interactive build and process UI' is used to build the POT in a step-by-step fashion (for example at BPON level) and to directly trigger the execution of the POT without using BPM in between.

In addition, error situations which are handled by the generated editing UIs (which are also are decoupled from the POT implementation itself) also require a set of interfaces to interact with the POT.

The interface cut offered by a POT is based on the above consumption options. A POT is enabled for consumption using one of the main options described above. These consumption options are also the reason why the POT does not offer a single 'all-in-one' service interface that creates, checks and completes the process in an end-to-end way .

4.2 Phase Model

Following the process flow pattern (4.1.2), the POT phase model consists of three phases:

Create phase

- Initializes the process object (PO) and its process object nodes (PONs) according to the given data
- nothing is changed in the back ends as yet, except fetching relevant data

Check (modify & check) phase

- The PO and its PONs can be modified: update, cancel and (for nodes only) create
- The PO and its PONs can be checked against predefined business rules.

Execute phase

- Successfully checked process objects can be executed.

  During execution, the corresponding back-end objects are called using services.

Figure 12 shows a high-level overview of the phases and phase transitions of a POT:

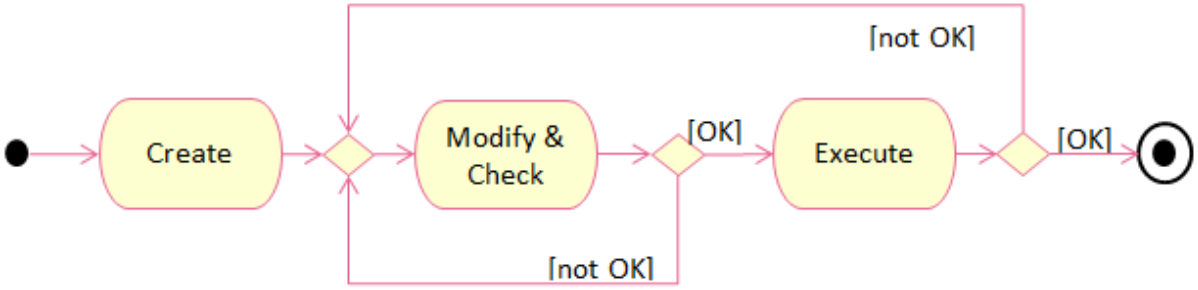


Figure 12: High Level Phase Model of a POT

Figure 12 shows only a simplified phase model, related to the overall life cycle of a POT. Although the figure shows overall error situations at a high level, errors can also occur during creation (failed or erroneous back-end calls), during modification and subsequent checks, and so on.

A more comprehensive overview including all the allowed status transitions follows in section 4.4 below.

4.3 Status Model

For each phase of a POT, there is a set of predefined statuses and status transitions.

The following basic principles apply to the POT status model:

- The status of a BPON can be changed by executing a service operation (each status change operation ,except `Cancel`, is part of the action service interface of a POT)
- The overall status of the BPO is calculated taking into account the status of the BPO and the statuses of all its BPONs
- The status of the BPO/BPON determines, which service operations can be executed on the BPO/BPON

The statuses (BPO / BPON) are listed as code values of the GDT `BusinessProcessObjectNodeStatusCode`:

Code	Name	Description
1	*Unchecked*	BPON is not checked.
2	*Checked*	BPON checked successfully.
3	*Erroneous*	BPON has erroneous data.
4	*In Maintenance*	BPON is in maintenance for (manual) post processing.
5	*Waiting for Execution*	BPON is waiting for execution.
6	*In Execution*	BPON is currently being executed.
7	*Failed*	The execution of the BPON failed.
8	*Confirmed*	BPON successfully executed and execution has been confirmed.
9	*Canceled*	BPON is canceled.
10	*In Compensation*	BPON execution has failed and, in order to cancel this node, compensation is needed.

Table 4-1: Values of `<BPON>StatusCode`

Figure 13 explains at a high level, how POT statuses can be manipulated by the corresponding services of the POT action interface and also shows an example of how the overall (aggregated) BPO status is calculated from the statuses of the individual BPONs:

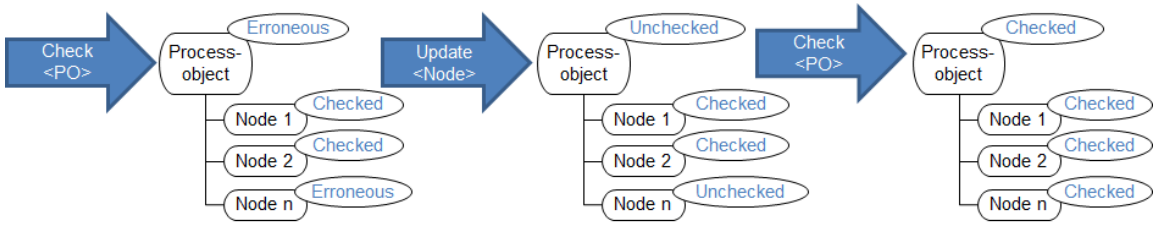


Figure 13: Status Transitions and Calculation of Aggregated POT Status

The blue arrows indicate a service call. The process object and its nodes indicate the status on node level as well as the aggregated status on BPO level after the service operation is executed. After the execution of the operation Check<PO>, the process object gets the status *Erroneous* since at least one of its BPONs (Node n) is *Erroneous*. After the Node n is updated (to correct its data) it get the status *Unchecked*, so the overall status of the process object aggregates to *Unchecked*. After the subsequent execution of the operation Check<PO> all nodes are *Checked* which means that also the overall status of the process object is *Checked*.

Details on overall status calculation follow in Part II of this guide.

4.4 Status Transitions

The POT programming model predefines the allowed status transitions depending on the status itself and the phase in which the POT currently is. The high-level phases Create, Check and Execute can be classified into the following status groups:

- *In Processing* Phase

  This includes the Initial creation of the POT and its BPONs (including error handling) and maintenance (update) of BPONs after a failed execution during reprocessing.

  This broadly corresponds to the phases Create and Modify & Check in Figure 12.

- *In Execution* Phase:

  If the checks were successful, execution is triggered. The POT executes the assigned back-end calls of each BPON in a predefined manner. It receives and processes back-end confirmations.

  This broadly corresponds to the success path of the *Execute* phase in Figure 12.

- *In Cancelation* Phase:

  The BPO is in the process of cancelation, the individual BPONs are checked to determine whether compensations might be required (some back-end changes might already be successfully completed, while others are not).

  This broadly corresponds to the unsuccessful path that is not completely shown in Figure 12.

Figure 14 gives a detailed overview of the allowed status transitions in the context of the above classification of the POT phases:

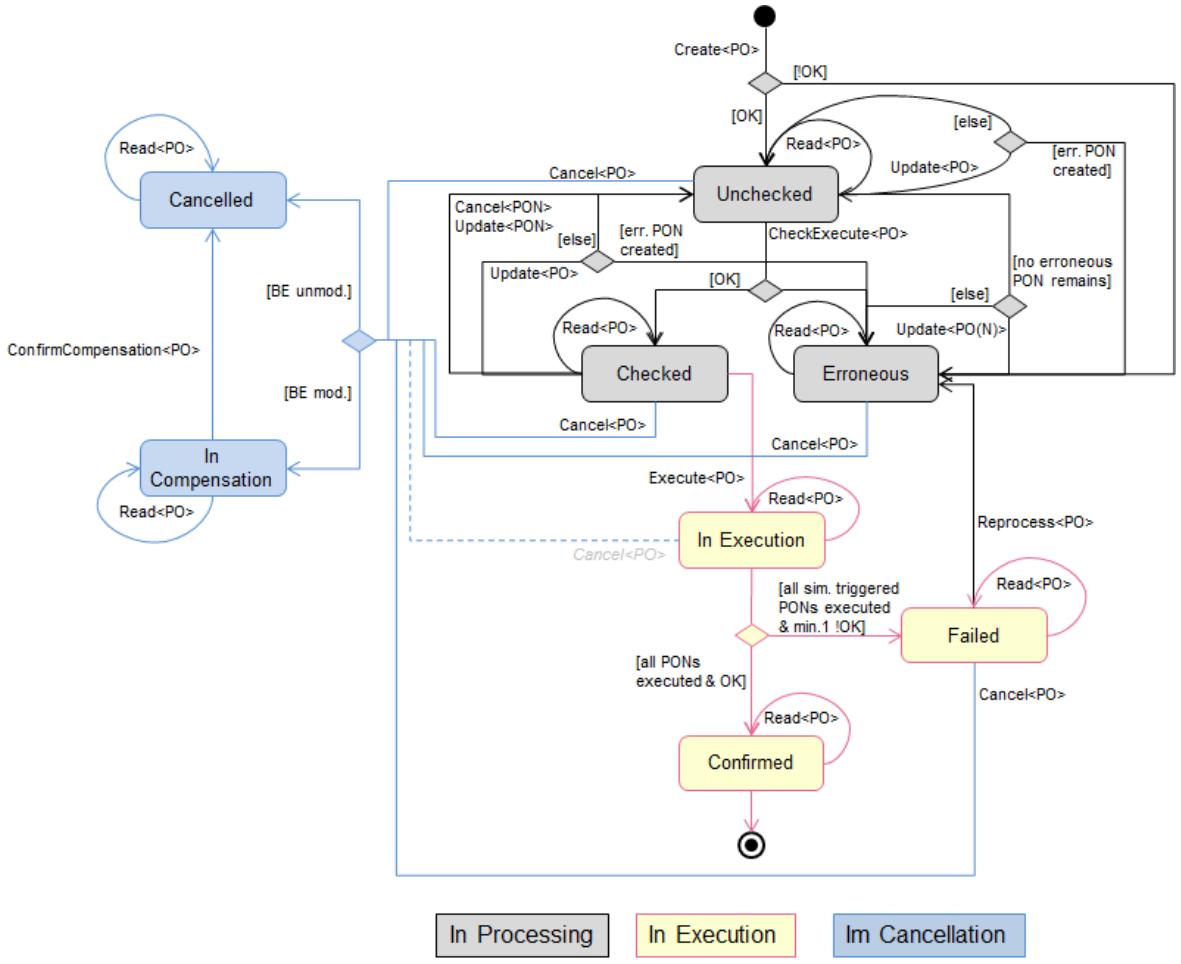


Figure 14: POT Status Transitions Overview

Here are some key takeaways:

- It is always possible to read a POT, irrespective of the BPO status or that of its BPONs
- The status *Erroneous* can occur either due to a failed check operation or due to errors in create/update operations
- A successful creation or update of a BPON always results in the status *Unchecked* of this BPON
- When POT cancelation is triggered, the status calculation automatically determines whether or not a BPON is to be externally compensated.

  However, the completion of such an external compensation must explicitly be made known to the POT by the consumer of the POT using the service operation `ConfirmCompensation<BPON>`.

A detailed description of how statuses are handled by the POT and by the custom implementation code slots follows in Part II of this guide.

4.5 Phase Implementation

Almost all the custom code slots that must be implemented by a POT developer to complete the POT implementation are related to and classified according to the phase model of the POT.

The following sections describe the key principles that you should keep in mind while implementing the phases of a POT.

Figure 15 provides a legend for the sequence diagrams in all the following sections:

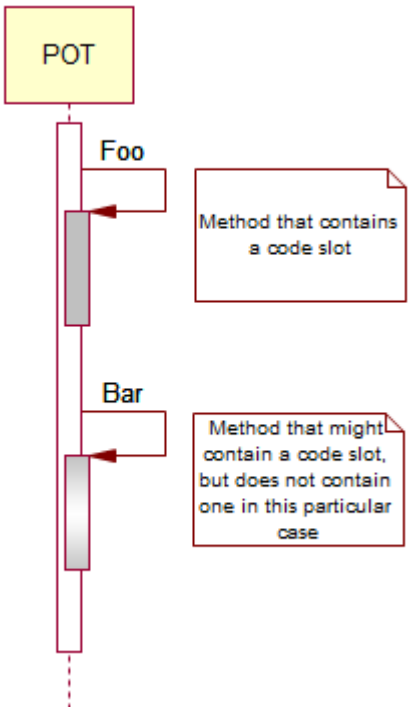


Figure 15: Legend for Sequence Diagrams

Methods that contain a code slot are depicted as a solid grey bar. Methods that can contain a code slot but do not have one in this particular case are depicted as a grey bar with gradient.

4.5.1 Create Phase

In the *Create* phase, the data of a POT is populated at creation time according to the following basic principles:

- The back-end services that have been assigned to the BPON during POT modeling in POB will be called to retrieve the data of back-end business objects that is relevant for the beautified view of the BPON (POT service call `<BPON>Create`, and `<BPO>Create`).
- When a BPON is updated (POT service call `<BPON>Update`), no back-end service calls will be executed.
- Back-end service calls will only be executed if a BPON instance is newly created during the update of the whole BPO (POT service call `<BPO>Update` with `actionCode = 01 -Create-` for new BPON instance).
- The custom implementation must take care of the `CreationInstructionCode` and populate the BPON data accordingly.

  The `CreationInstructionCode` values contain instructions on whether to keep, overwrite or merge the data supplied via POT service call with the data retrieved from the back-end service call during the

creation of a BPON. A detailed description on how to handle the `CreationInstructionCode` element will follow in Part II of this guide (section 7.1.3.6).

In the `FlightAndShuttleBooking` sample POT, the following back-end services can be modeled for calls in the *Create* phase: (The actual service interfaces and operations will be detailed in Part II of this guide.)

- BPON: `BusinessTraveller`

  Service operation for reading `BusinessTraveller` information (from the back-end business object "Business Partner") that is used to fill the `Details` node of the BPON.
- BPON: `Flight`
  - Service operation for reading flight information for the specified `PlannedFlightDate` in the `ProcessControlConstraints` node of the BPON and filling the BPON instance `Details` for each flight that is available for the specified `PlannedFlightDate`.

Figure 16 shows a sequence diagram of the *Create* phase, including the *Create* phases of all the BPONs and the back-end services involved:

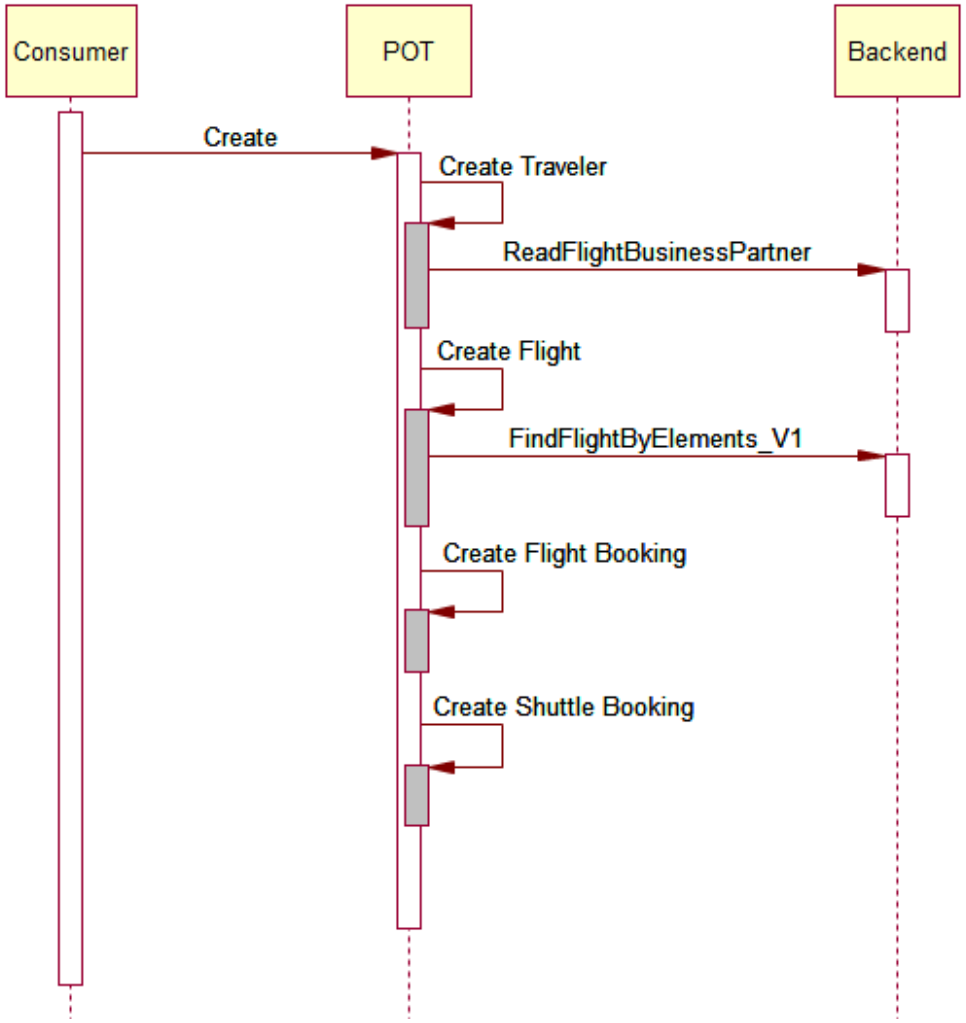


Figure 16: *Create* phase – Dynamic View

The BPONs "Traveler" and "Flight" each call a back-end service during the *Create* phase. The BPONs "Flight Booking" and "Shuttle Booking" do not use back-end services in this phase.

In the *Create* phase, all the BPONs always have a code slot.

4.5.2 Check Phase

During the *Check* phase, the following checks are performed in the order specified below:

1. **Standard Validation**

   The standard check is basically intended to check cardinalities (only for the BPO) and cross-references (only for BPONs).

   This check is completely generated, whereas the following checking step are intended for customer-specific business rules.

2. **Check BAdI**

   The Check BAdI is used to perform business checks based on ABAP coding. For each BPON/BPO a separate BAdI method is available (if selected during modeling) which has the complete public type of the BPO/BPON as importing parameter.

3. **BRFplus rule-based checks**

   BRFplus rule-based checks can be defined for the complete BPO and for each of the BPONs separately. The rules must be maintained in rule sets, which are attached to the corresponding function. For each BPON/BPO a separate BRFplus function is available (if selected during modeling). BRF functions are the "single point of entry" for BRFplus-based checks on the BPO and the BPONs.

   In this context, BRFplus is used for data validation, not for the derivation of values.

4. **Check services**

   During the *Check* phase, a code slot is provided where those services can be called, that are assigned to the *Check* phase of the corresponding BPON type. If no services are defined for this phase, there will be no code slot available.

The most important principle for this phase is that the data of a BPON cannot be modified.

In the sample POT, the following back-end services can be modeled for calls in the *Check* phase (the actual service interfaces and operations will be detailed in Part II of this guide):

- BPON: `FlightBooking`

  Service Operation to check the availability of seats in the selected flight. The check operation is provided by the back-end business object for flight booking.

- BPON: `ShuttleBooking`

  Service Operation to check the availability of seats in the shuttle connection for the selected flight. The check operation is provided by the back-end business object for booking shuttle connections.

In addition to the services mentioned above, a BRFplus check has been modeled at the root node of the sample POT to check the price of the selected flight and raise an error message in case the price exceeds a predefined limit (for example: 1000 €). When this rule is called by the *Check* phase implementation and the supplied POT instance contains a selected flight with a price that is higher than the predefined limit, the phase implementation sets the status of the BPON "Flight" (and, subsequently, also the calculated overall status of the BPO) to *Erroneous*.

Figure 17 shows a sequence diagram for the *Check* phase of the sample POT:

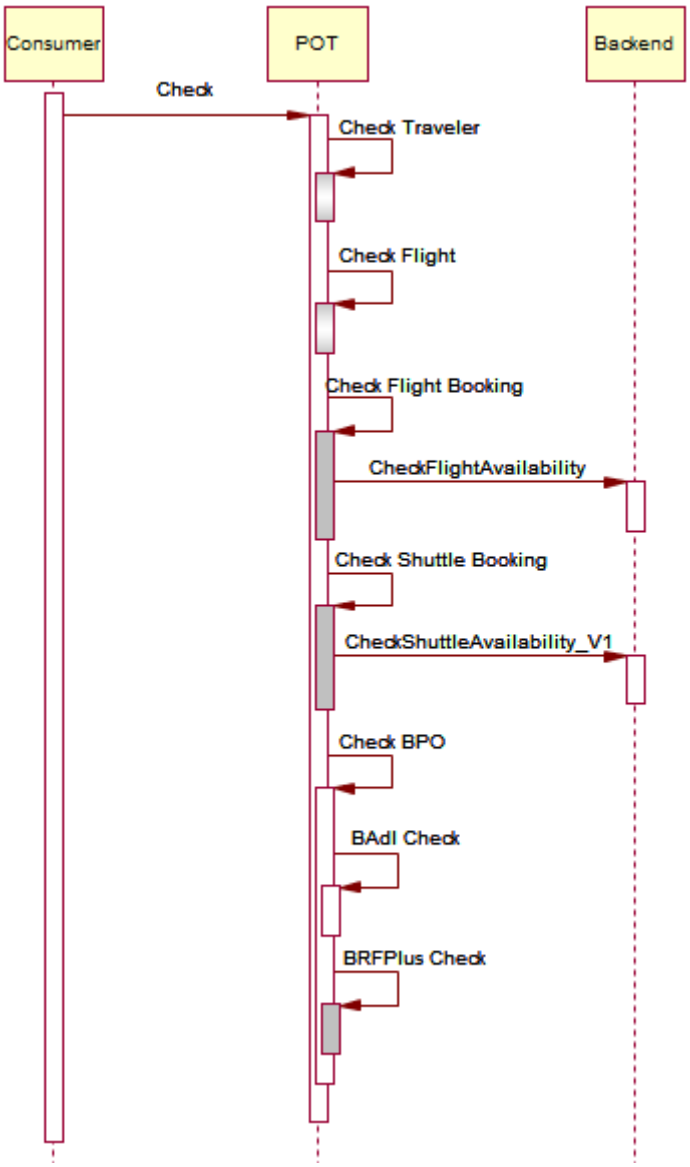


Figure 17: *Check* Phase – Dynamic View

First, all BPONs are checked. The BPONs "Flight Booking" and "Shuttle Booking" call services from the back end. In the sample POT,no BAdI or BRFplus checks are executed at BPON level.
After that the BPO is checked, starting with the standard checks. Then the Check BAdI is called followed by the checks implemented in BRFplus. Services cannot be modeled on BPO level.

In the *Check* phase, only BPONs with services have code slots. The code slot for the BRFplus check (in this case on BPO level) is used for mapping POT and BRFplus data representations.

4.5.3 Execute Phase

In this phase, the back-end services that have been specified for the individual BPONs during POT modeling in POB must be called. The custom implementation must make the following settings:

- Set the identifier of the back-end object as the `ProviderID` of the BPON (if the back-end call was successful)
- Set the parameter `backend_modification_state` (`modified/unmodified`)

- Set the appropriate status

There are two possibilities to do this:

- Call synchronous services and set the parameters based the immediate response
- Call asynchronous services and set the parameters in a different code slot that processes the confirmation message

The most important principle for this phase is that the `Details` node of a BPON cannot be modified. Only the `AdministrativeData` like `ProviderID`, `StatusCode` and `backend_modification_state` can be set based on the confirmations received from the back end.

In the `FlightAndShuttleBooking` sample POT, the following back-end services can be modeled for calls in the *Execute* phase (the actual service interface and service operations will be detailed in Part II of this guide):

- BPON: `BusinessTraveller`

  No service calls to back ends, as the BPON is only intended to query data required for POT processing
- BPON: `Flight`

  No service calls to back ends, as the BPON is only intended to query data required for POT processing
- BPON: `FlightBooking`

  Synchronous service operation to create the actual flight booking. The create operation is provided by the back-end business object `FlightBooking`.
- BPON: `ShuttleBooking`

  Asynchronous service operation to create a shuttle booking. The create operation is provided by the back-end business object `ShuttleBooking`.

Figure 18 shows a sequence diagram for the *Execute* phase of the sample POT:

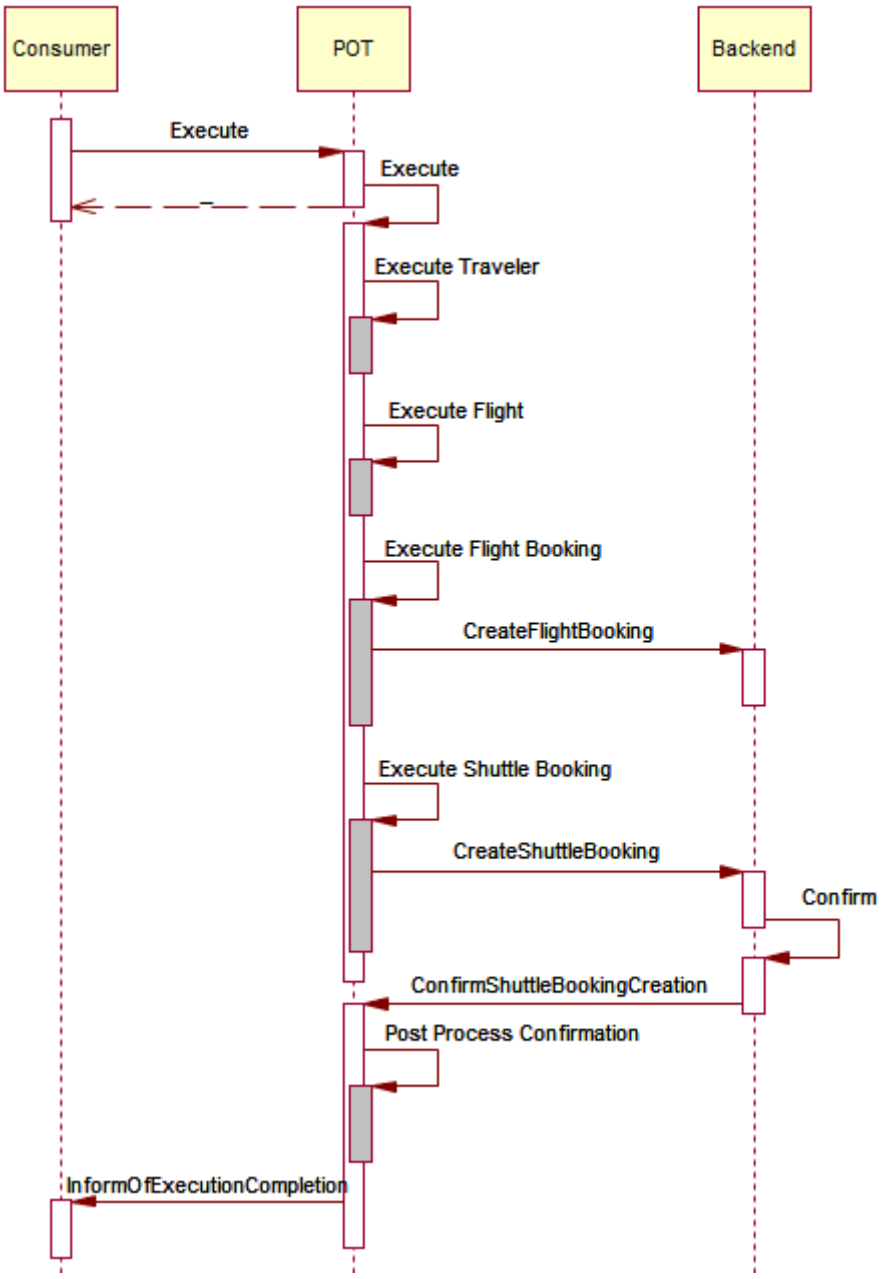


Figure 18: *Execute* Phase – Dynamic View

Like `Create` and `Check`, the `Execute` service operation is a synchronous operation. However, the actual execution is done asynchronously: Both BPONs, "Flight Booking" and "Shuttle Booking", call services from the back end. The `CreateShuttleBooking` operation is an asynchronous operation, which means that the *Execute* phase is completed as soon as the asynchronous message has been sent.
The response of the asynchronous service (`ConfirmShuttleBookingCreation`) triggers the post processing of the asynchronous confirmations. Once the POT instance has been executed completely, the consumer is informed by means of the asynchronous `InformOfExecutionCompletion` operation.

As with the *Create* phase, all BPONs always have a code slot for the *Execute* phase. If asynchronous back-end services are involved, there are also code slots for post-processing the corresponding confirmation messages.

4.5.4 Node Dependencies

In general, the POT programming model treats each BPON of a POT independently from other BPONs during the creation or execution of a POT (since each BPON represents an independent business object of the corresponding back end). That means that a BPON, in general, can neither access nor modify details of another BPON. However, during the *Create* phase and the *Execute* phases of a POT, it is often required to call the back-end services in a certain sequence to ensure an appropriate creation of the BPONs or the execution of the BPONs in a consistent manner. In order to implement such a sequencing of the back-end service calls, POB offers the so-called "node dependencies", which can be specified for a POT at design time in the POB Specification Wizard.

In our sample business scenario, the BPON "Flight" must be executed before the BPON "ShuttleBooking" is executed, as the flight arrival and departure times are required to successfully perform the shuttle booking.

The specification of node dependencies for the EXECUTE phase of sample business process looks as follows:

Execute Phase	
Prior Process Object Node (BPON)	Subsequent Process Object Node (BPON)
`BusinessTraveller`	`FlightBooking`
`Flight`	`FlightBooking`
`BusinessTraveller`	`ShuttleBooking`
`Flight`	`ShuttleBooking`

If node dependencies have been specified, the POT implementation permits access to the details of the corresponding BPON (Prior Process Object Node) by a dependent BPON (Subsequent Process Object Node) via cross-references (see section 4.5.5 below for an explanation of the cross-reference concept).

These are the most important principles for the specification of node dependencies:

- Node dependencies can be specified independently for the *Create* and for the *Execute* phase.
- Access from a subsequent BPON to a prior BPON is restricted to the corresponding phase for which the dependency has been specified at design time. That means that accessing a prior BPON from a subsequent BPON during the *Create* phase is only possible if the corresponding dependency has been specified for the *Create* phase. It is not possible to access a prior BPON in the *Create* phase if the corresponding dependency has been declared for the *Execute* phase.

4.5.5 Cross-references

As already mentioned in the previous section, a POT implementation treats each BPON implementation independently from other BPONs. In case node dependencies between two BPONs are specified for a certain phase of POT, it is required to navigate to a prior node from a subsequent node in order to access the data. This navigation is enabled by a construct called "cross-reference". The cross-references concept for BPONs is similar to the foreign key concept in database tables.

A cross-reference is the specification of a key of the prior BPON in subsequent BPONs that is used for navigation by the implementation of the BPON. A cross-reference is a generic data type that is predefined by POB and made available at design time in the Modeling Wizard. You should include this data type in the `Details` node of each BPON that you want to specify as a subsequent BPON. The name of the prior BPON, to which this cross-reference refers to in the subsequent BPON, must be identified using the description field of the data type in ESR.

Cross-references have the following generic structure:

Generic Element Structure for Cross-references: **`M_<BPO>_CrossReference`**

Element Name	Type	Comment
`UUID`	`UUID`	`UUID` of the prior BPON Exclusively identifies the prior BPON
`Reference`	`BusinessTransactionDocumentID`	`ReferenceID` of the prior BPON Part of the composite key for identification of the prior BPON
`TypeCode`	`ObjectNodeTypeCode`	`TypeCode` of the Prior BPON Part of the composite key of the prior BPON

For the implementation, either `UUID` or the combination of `Reference` and `TypeCode` can be used as the key for navigation in order to access the details of the prior BPON.

In the sample business scenario, the following cross-references are specified in the BPON FlightBooking, according to the required dependencies:

Element structure for BPON: **`FlightBooking-Details`**

Element Name	Type	Comment
`BusinessTraveller`	`M_BusinessTrip_CrossReference`	Cross-reference to BPON `BusinessTraveller` from BPON `FlightBooking`
`Flight`	`M_BusinessTrip_CrossReference`	Cross-reference to BPON `Flight` from BPON `FlightBooking`

These are the most important principles for the specification of cross-references:

- Cross-references should be specified in accordance with the required node dependencies for the BPON.
- Cross-references must be specified in the modeling phase of a BPON using the predefined data type generated by POB.
- Cross-references must only be used to create links to the `Details` node of a prior BPON, not to the `ProcessControlConstraints` node of the prior BPON.

5 POT High-Level Design

Now that you've got an understanding of the programming model of a POT, it is useful to look at the high-level design of a POT in terms of its static view.

This chapter therefore introduces the high-level design of a POT, including a general overview of both, the generated POT artifacts and the static relationships at design time and runtime. The ABAP package overview for a POT is also introduced with high level dependencies. The last section of this chapter gives an overview of the key information required from a development architect's point of view (in the form a checklist).

5.1 Block Diagram

The complete design time and runtime environment of a POT can be classified into the following parts:

- Design-time artifacts in ESR for the service operations that are provided and consumed by a POT
- The generated ABAP artifacts for the implementation of the POT (including the generated editing UIs for the POT, if any)
- Optional process definitions and configuration of processes in SAP NW BPM

In addition to the parts listed above, which are relevant for POT shipment, the POB design-time modeling artifacts that are used for generating the POT artifacts also reside in ESR.

Figure 19: provides the overview of the static structure:

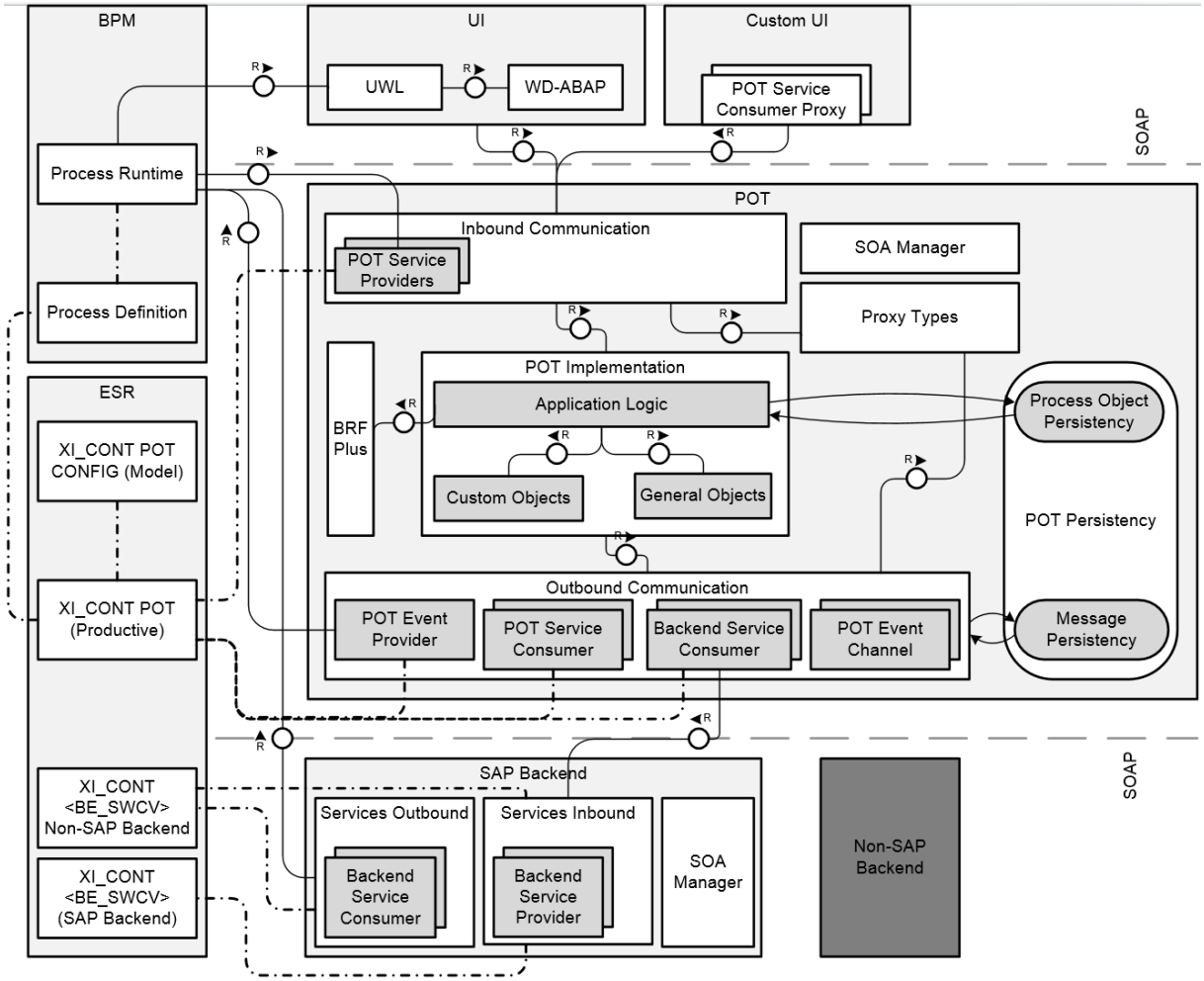


Figure 19: POT Static Structure Overview

The dashed lines represent POT design time relationships.

The diagram also shows configuration parts for the case that P2P (SOAMANAGER) is used. This is only one option of configuring the services that is used for indicative purposes. SAP NetWeaver™ PI Integration Builder or any other 3rd Party middleware could be used for configuring service connections.

The following sections describe the static structure in more detail.

5.1.1 ESR Content

POB generates all the services required by a POT into ESR:

- Synchronous and asynchronous inbound services to access the POT
- Asynchronous outbound services which confirm the execution of the asynchronous POT inbound services
- Synchronous outbound services corresponding to the synchronous POT inbound services that are used by the editing UIs
- Outbound services (synchronous as well as asynchronous) that are required to call the back-end inbound services that have been modeled at BPON level of a POT
- Asynchronous inbound services that receive the replies from the back-end service calls

In addition to the services mentioned above, the ESR content that is used for POT modeling is also present depending on the choice of SWCV and ESR namespace (see section 3.3.7).

5.1.2 POT (ABAP Artifacts)

The POT itself is made up of all the ABAP artifacts that are generated by POB.

POB generates ABAP artifacts according to the package and software layer concept, which is standardized.

The layer concept of a POT can briefly be summarized as follows:

- Inbound Communication Layer – ABAP implementation artifacts for all the inbound service interfaces of a POT
- Application Layer – all the artifacts that implement the ABAP APIs, phase and status handling, checks, persistence and integration to tools like CNS, OAF and Process Observer for a POT
- Outbound Communication Layer – ABAP implementation for all the outbound service interfaces and ABAP Channels of a POT

5.1.3 Editing UIs

POB also generates the editing UIs as WebDynpro ABAP (WDA) applications, along with the application configurations that are required to view and process a POT.

Editing UIs are optional, they are only generated if the POB Editing UI Wizard is run. CHIPs can also be generated by the Editing UI Wizard. These CHIPs can be assigned to human interaction steps of a BPM process.

5.1.4 BPM Content

In contrast to the other POT artifacts, POB does not generate any BPM content. If you want a BPM process to handle the automation of a business process using a POT, you need to manually create and configure the process definition in a suitable BPM tool. SAP offers SAP NetWeaver™ BPM for this purpose.

5.1.5 Back-End Services

In general, POB does not touch any existing back-end service interfaces. The service definitions of the back-end services that are called by a POT must exist in the same ESR system that is used by POB for modeling and generation of the POT. The message types and global data types that are used by the back-end services are copied (in a manual step) into the ESR namespace of the POT.

Nevertheless, the shipment of SAP Process Object Builder also includes SIW templates that help you to implement the service interfaces in the back end. You can use these templates to generate service implementations that follow a standard programming model, which will accelerate the implementation of back-end services with features like ECH, PSJ and so on, depending on the service type. However, please note that the shipped SIW templates in POB are only intended to help with service implementation. You cannot generate any ESR content with these templates.

A developer's guide "Implementation of Enterprise Services Using SAP Process Objects Builder" can be found in the Product Documentation for SAP Process Object Builder.

5.2 Package Structure

At the highest level, a POT implementation (with all its ABAP artifacts) is generated into a main package. The super package for this main package must be specified in the POB Specification Wizard (Step 1 – Define ABAP Environment).

There are three packages that define the ABAP environment of a POT:

- The root package (or <SUPER> in Figure 20 below) can be, for example, the structure package of the software component.

  ABAP namespaces are also supported. If an ABAP namespace is used, the package name should also start with this ABAP namespace.
- A POT can exist together with other POTs in this super package, as long as they use the same productive ESR namespace. In this case,
  - the SPROXY types for all ESR data types will be in the package for common proxy types (GDTs)
  - the internal DDIC types that are used across different POTs are located in the package for common internal types

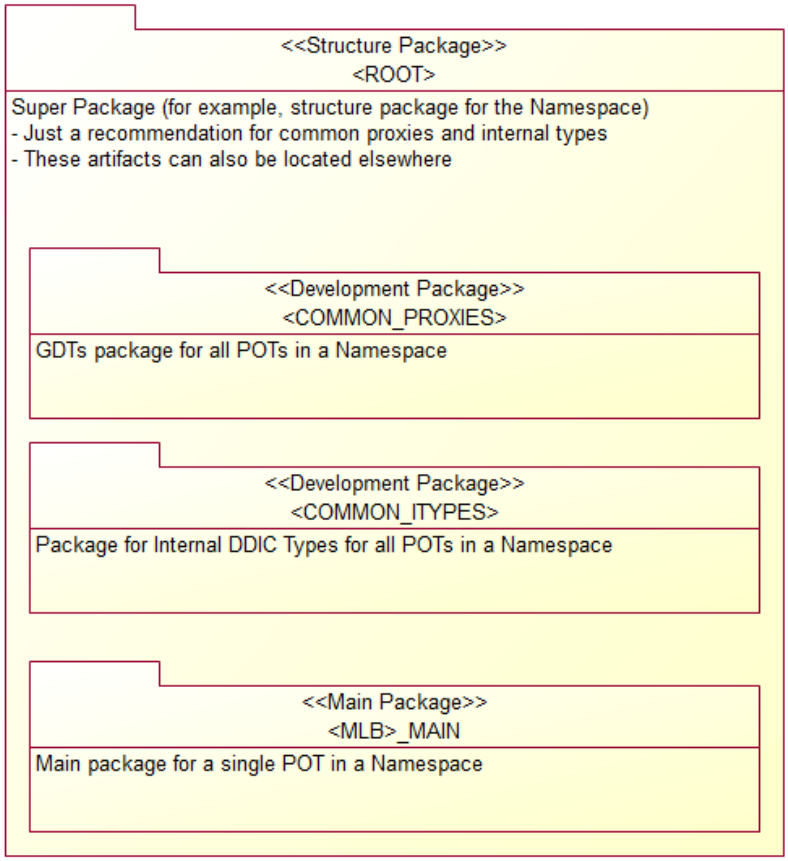


Figure 20: ABAP Environment of a POT

Once the ABAP environment of a POT is defined, all the ABAP artifacts of the POT itself are generated into the corresponding subpackages (see subsequent sections).

5.2.1 POT Core

Figure 21 shows the package structure (subpackages under the main package of a POT). Dedicated packages are foreseen for internal types as well as inbound and outbound service layers according to a standard software layer concept.

The application layer of a POT consists of dedicated packages for general objects (reused across several POT artifacts), the core API of a POT, and a separate package is used for custom objects (ABAP artifacts that are intended for code slot implementation).

Please note that a POT developer (with sufficient development coordination authorization in the ABAP system) must create these packages manually according to the instructions in the corresponding Generation wizard Step.

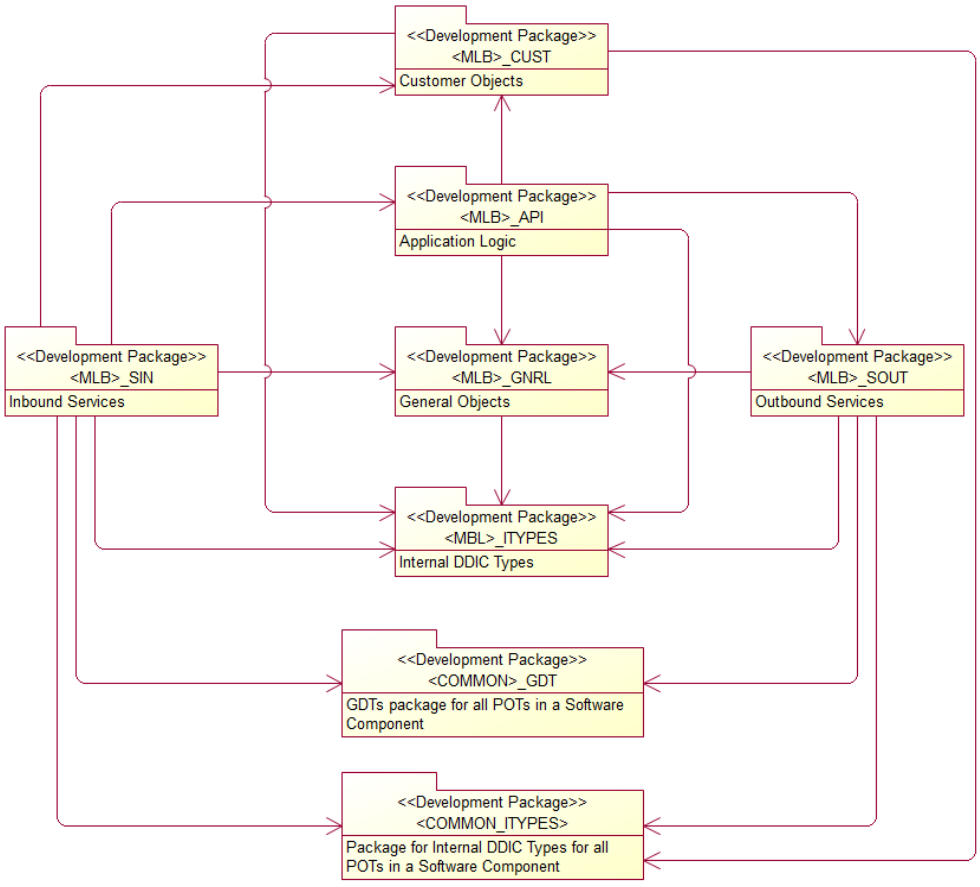


Figure 21: POT Core Package Structure

5.2.2 POT Editing UI

Figure 22 shows the package structure with the focus on the editing UIs of a POT. All the POB-generated artifacts that do not require custom code slot implementation are generated into the `<MLB>_UI` package, whereas those artifacts that require custom code slot implementation are generated into the package `<MLB>_CUST`.

Please note that the generated editing UIs of a POT also use enterprise services to communicate with the POT itself (loose coupling of POT editing UI with the POT core implementation).

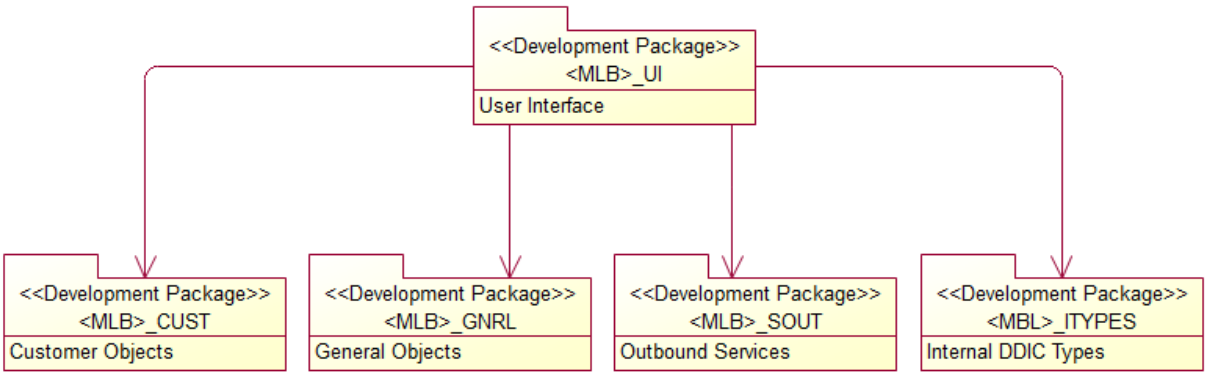


Figure 22: POT Package Structure – Editing UI View

5.2.3 Package Encapsulation

It is a well-proven best practice in architecture to encapsulate functionality and use package interfaces and use accesses to expose only the required software artifacts. This ensures a better handling of dependencies between software artifacts. In the context of a POT implementation, the most important objective of encapsulating artifacts is to minimize the dependency of custom code slot implementation on the other generated POT artifacts. To follow this practice, you should ensure that the custom code slots only use generated objects that are sufficiently exposed. Doing this, in turn, provides the advantage that the custom code slots will rely on stable contracts between generated and custom coding and also ensures that there will be no impact on custom coding in case some internals of the generated (not exposed) code change due to a feature enhancement in POB.

POB also validates that the package interfaces and corresponding use accesses are maintained sufficiently and appropriately (no less than necessary, no more than required). However, the maintenance of package interfaces and use cases itself is a manual activity that is performed in one of the last Generation Wizard steps.

The following three types of package interfaces are used within a POT:

- Interfaces for customer-specific implementations

  These interfaces only expose those ABAP artifacts that are intended for customer usage and therefore offer the required stability. These interfaces are added as use accesses to the customer package.
- POT-internal interfaces

  POT-internal interfaces are added as use accesses for packages that contain generated ABAP artifacts only.
- POT-internal interfaces for usage in UI Package

  The interface for usage in the UI package is currently only needed in the custom package. It contains all the objects of the custom package that are generated in the Editing UI Wizard and only needed in the UI package.

5.3 POT Design & Specification Checklist

POB requires several input data that is used to derive the names of the generated POT artifacts like ESR content and the ABAP implementation objects. This input data is requested by POB in various steps of the Modeling and the Specification Wizard.

Although most of these values seem trivial to specify in the wizard steps, in a real POT development scenario some of these values must be chosen with care by development architects. For others, we recommend that the architects specify naming conventions for the development teams. This section provides an overview of this type of input data that is required by the POB wizards together with some rationale/motivation and suggestions for naming conventions.

The following table only gives you an overview of the input data that is important to consider from a development architect's point of view. We strongly recommend POT developers to always refer to the quick help that is provided with each POB wizard step in order to get the most relevant information on the values to be specified, applicable prerequisites for a step and so on.

In the "Wizard" column, the following abbreviations are used:

"M" - Modeling Wizard

"S" - Specification Wizard

"G" - Generation Wizard

Input / Activity	Wizard	Step	Comment / Motivation / Suggestion
POT Models	-	-	It is a general recommendation that the overall context and software architecture environment of a POT should be modeled using standard SAP or other modeling tools of choice. For the POTs shipped by SAP, the SAP SOA Modeling Methodology standards and tools are adhered to.
Documenting a POT design	-	-	The design document of a POT is one of the few things that the POB does not generate (as yet). We strongly recommend to sufficiently document important design and architecture decisions in a dedicated design document.
Configuration ID	M	*Enter General Data*	Must be unique for a POT.
Process Component Name	M	*Enter General Data*	Must be unique for a POT. For SAP-internal development, this name must be identical to the process component name approved in the PIC governance process (PIC 0).
SIW Landscape ID	M	*Enter General Data*	Must correspond to a working SIW Landscape (refer to SIW customizing)

Input / Activity	Wizard	Step	Comment / Motivation / Suggestion
Application Component	M	*Enter General Data*	Appropriate name that is assigned to the POT in the SAP Application Component Hierarchy. This is an important decision that must be made by architects. The value is used (amongst others) in the artifacts related to the following aspects: • Archiving • BRFplus • Packages • ECH / PPO Later changes to this value are incompatible and require regeneration of the POT.
Software Component	M	*Enter General Data*	Corresponds to the shipment unit in ABAP. Cannot be changed.
ESR Namespaces for Modeling and Production	M	*Enter ESR Data*	Important architectural decision: See section 3.3.7 above.
PON Details – Node Type Identification	M	*Define Process Object Nodes*	For SAP-internal development: • *ESR Name* and *Description* must be identical to those approved in the PIC process (PIC 2). • The supplementary components (*Code List ID* etc.) must be identical to those maintained centrally by integration architects. For partner/customer development: • Separate `listID/listAgencyID`s must be maintained for the complete system landscape. Each BPON of a POT must be assigned a code value that makes the BPON uniquely identifiable within the context of the `listID` and `listAgencyID`. We recommend to follow `UpperCamelCase` style when specifying the ESR names.
PON Details – BO Type Identification	M	*Define Process Object Nodes*	It is used by some of the connected tools, such as the Process Observer and ECH. As the Business Object Type code must be unique for each POT (and its version), we recommend to also include the version of the POT.
Process Object Node Details – Data Type Settings & Back-End Operations	M	*Define Process Object Nodes*	For SAP-internal development: • Values must correspond to the content of the approved PIC document (PIC 2) for the POT with the corresponding models
Package names (Proxy Objects, Common Internal Types, and POT specific)	S	*Define ABAP Environment*	Proxy objects and common internal types that are used in multiple POTs must be located in a package that is different from the root package of the POT. Packages must exist and sufficient use accesses must be specified. Refer to 5.2 above for details on how to specify the packages.

Input / Activity	Wizard	Step	Comment / Motivation / Suggestion
Namespace, ABAP abbreviation and SPROXY Prefix	S	*Define ABAP Environment*	The *ABAP Namespace* (if declared) must also be reflected in the package names. The abbreviations are used to derive the names of ABAP artifacts according to specified derivation rules: Derivation rule for classes/interfaces: **[<Customer Namespace>|<ABAP Namespace>][IF|CL]_<ABAP Abbreviation>_<POTAbbreviation><POTVersion>_...** For proxy types, the *SPROXY Prefix* is used to derive the proxy type name: **[SPROXY Prefix]**... We therefore recommend to limit the abbreviations and the SPROXY prefix to as few characters as possible to avoid a violation of length restrictions in the ABAP artifact names. The SPROXY Prefix will be identical for all objects of one ESR namespace. The SPROXY prefix must start with the *Customer Namespace* or the *ABAP Namespace*, if any.
Internal Types Prefix	S	*Define ABAP Environment*	There are two prefixes required for internal types: *Common Internal Types Prefix* and *POT-Specific Internal Types Prefix*. Both have to be exactly one character shorter than the SPROXY prefix. The prefix for common internal types is used for all internal DDIC types that are shared within an ESR namespace, such as DDIC types for back-end-related message types. This prefix will be identical for all objects of one ESR namespace. The prefix for POT specific internal types is used for DDIC types that are specific for a POT, such as DDIC types for POT-related message types.
Abbreviations for back-end service counterparts	S	*Specify Counterparts for Back-End Services*	The abbreviations are used to derive the names of ABAP artifacts (see above). We therefore recommend to limit the length of the abbreviations as far as possible. Suggested naming convention for interface types (maximum abbreviation length = 4 characters): • Manage - **<BO>M(N)** • Action - **<BO>A(C)** • Query - **<BO>Q(Y)** <BO> = back-end business object abbreviation (maximum 2-3 characters) Suggested naming convention for specifying abbreviations for some operations (max. abbreviation length 4 characters): • Find – **FIND** • CheckCreate – **CHCR** • Cancel – **CNCL** • ...

Input / Activity	Wizard	Step	Comment / Motivation / Suggestion
CNS Application ID and Information Message Receiver	S	*Enter Information for CNS*	The *Application ID* identifies the POT within the CNS customizing, it must be unique for each POT (across all POTs) *Receiver (Information Message)* identifies the CNS receiver implementation that triggers the information messages that are sent by the POT after status changes. Suggested naming conventions are as follows: • `Application ID` - **<POTAbbreviation><POT Version>** • `Receiver` - **<POTAbbreviation><POT Version>R** (for Status Change event) • `Receiver` - **<POTAbbreviation><POT Version>S** (For Execution Completion event)
Development package creation	G	*Create Packages*	See section 5.2
Maintenance of package interfaces and use accesses	G	*Create Package Interfaces* & *Create Use Accesses*	See section 5.2.3

In addition to the input values listed above, which are required by POB, development architects must also decide on the naming conventions and the documentation of artifacts for business rules (BRFplus) and enhancements (BAdIs) for the *Check* phase of the POT.

Architecture and Design Decisions for POT

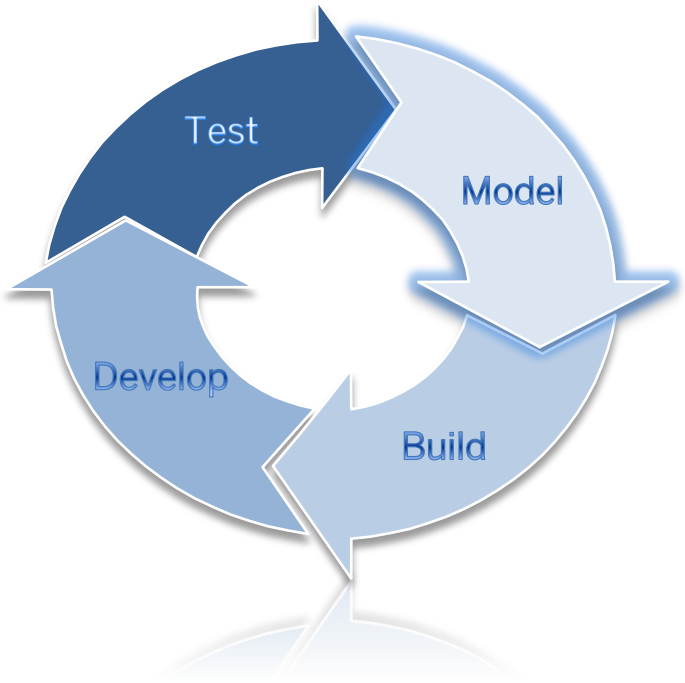
The creation of a POT using POB requires several decisions, for example on input values, which must be made by an architect. It is strongly recommended that these decisions are sufficiently documented.

Part II – Detailed Design & Implementation

POB heavily supports the implementation of a POT by generating most of the required artifacts. However, in order to complete the implementation, implementation code must be inserted in specific code slots that are generated into the POT artifacts. In order to successfully leverage the programming model of a POT and ensure conformance to this model, you need to know about the generated artifacts, about which parts of the se artifacts can be used, and where custom objects (ABAP) can be introduced.

This part of the Developer's Guide provides in-depth information about a POT, including a description of the generated artifacts and a sample implementation of the predefined code slots. It also provides information on the APIs of the generated artifacts and how to use them.

Make sure that you are familiar with the concepts explained in Part I of this guide before proceeding.



6 POT Detailed Design

"Unix was not designed to stop people from doing stupid things, because that would stop them from doing clever things "

- Doug Gwyn

This chapter marks the start of Part II of the POT Developer's Guide. Before you start on this part, you should be familiar with the overall background and motivation for a POT, the high-level architecture, the programming model and the high-level design of a POT, all of which are covered in Part I of this guide
This part of the Developer's Guide provides a deeper insight into the detailed design of a POT, with a focus on the implementation of code slots to complete the POT development. Nevertheless, it is helpful to know exactly, what artifacts are generated for a POT by POB, before starting to implement the code slots.

In general, a POT implementation is responsible for:

- Initializing a POT with the process-relevant data, either as a whole or in a step-by-step manner
- Checking the process-relevant data against business rules
- Executing back-end services, receiving the confirmations from the back end and reacting to those confirmations by changing the status of the POT as required
- Handling errors and business conflicts, editing the POT for corrections, reprocessing or canceling a POT

In order to fulfill these responsibilities, the POT implementation must have a suitable detailed design with respect to the following key aspects (among others):

- Lifecycle and associated status management
- Persistence
- Service handling with validation & mapping, if required

As a POT is built following a MDSD approach, POB generates the artifacts required to cover the aspects mentioned above. Some of these artifacts contain code slots that have to be implemented with custom code by a POT developer.
This approach of, on the one hand, generating the major parts of a POT, and on the other hand, providing code slots for the custom code makes the POT implementations uniform across several POTs. In order to best leverage the design of a POT during code slot implementation, it is very important for the POT developer to understand how the generated artifacts conform to the detailed design, understand the boundary conditions of a POT design and also the interfaces offered by a generated POT implementation towards the custom coding in code slots.

Therefore, this chapter focuses on the detailed design in the following aspects:

- POT layer concept and major building blocks
- Correspondence of layers to the actual development packages of a POT
- Drill-down to artifacts in each layer
- Static structures (class diagrams) and dynamic views (sequence diagrams) to explain key concepts and cross-cutting concerns

6.1 Layer Concept

POT design follows a standardized development package concept. This concept foresees an abstract layering of ABAP artifacts.

Figure 23 shows all the involved layers in a block diagram:

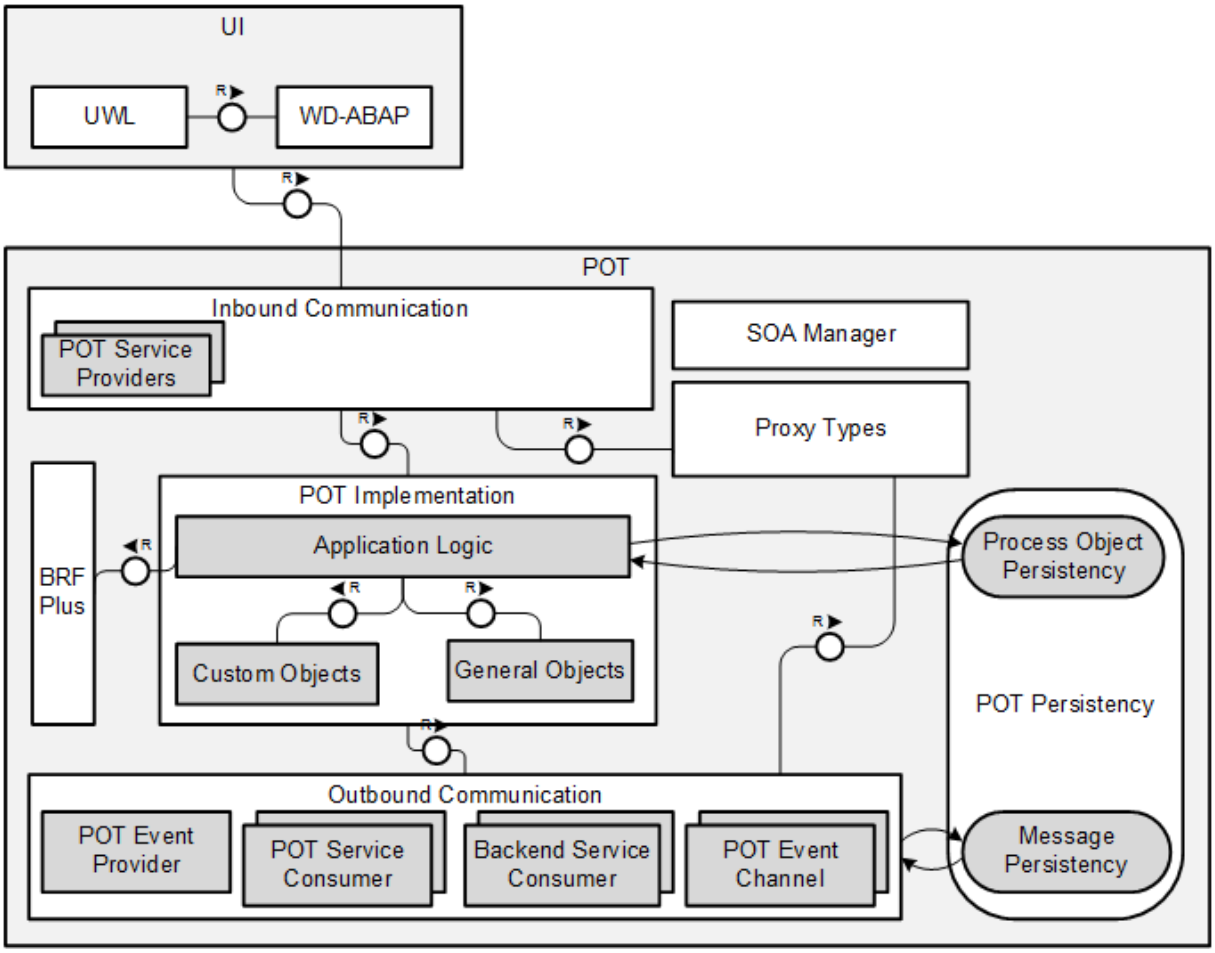


Figure 23: Block Diagram of POT Layer Concept

A POT basically consists of the following layers:

- Inbound Communication layer – responsible for connecting the services offered by a POT to the POT's application logic
- Application logic layer – responsible for handling the core processing logic of a POT with its APIs and the persistence
- Outbound Communication layer – responsible for connecting the outbound service calls to the back ends and to publish events
- UI layer – responsible for handling the processing logic of an editing UI, a generated user interface based on Floorplan Manager for Web Dynpro ABAP

The following subsections provide more detailed information on each layer.

6.2 Application Logic Layer

The application logic layer handles the core POT implementation. It is primarily responsible for handling POT instances, phase execution and status management. The single point of entry for the Application Logic Layer is the application façade.

Figure 24 shows a static view of the application façade:

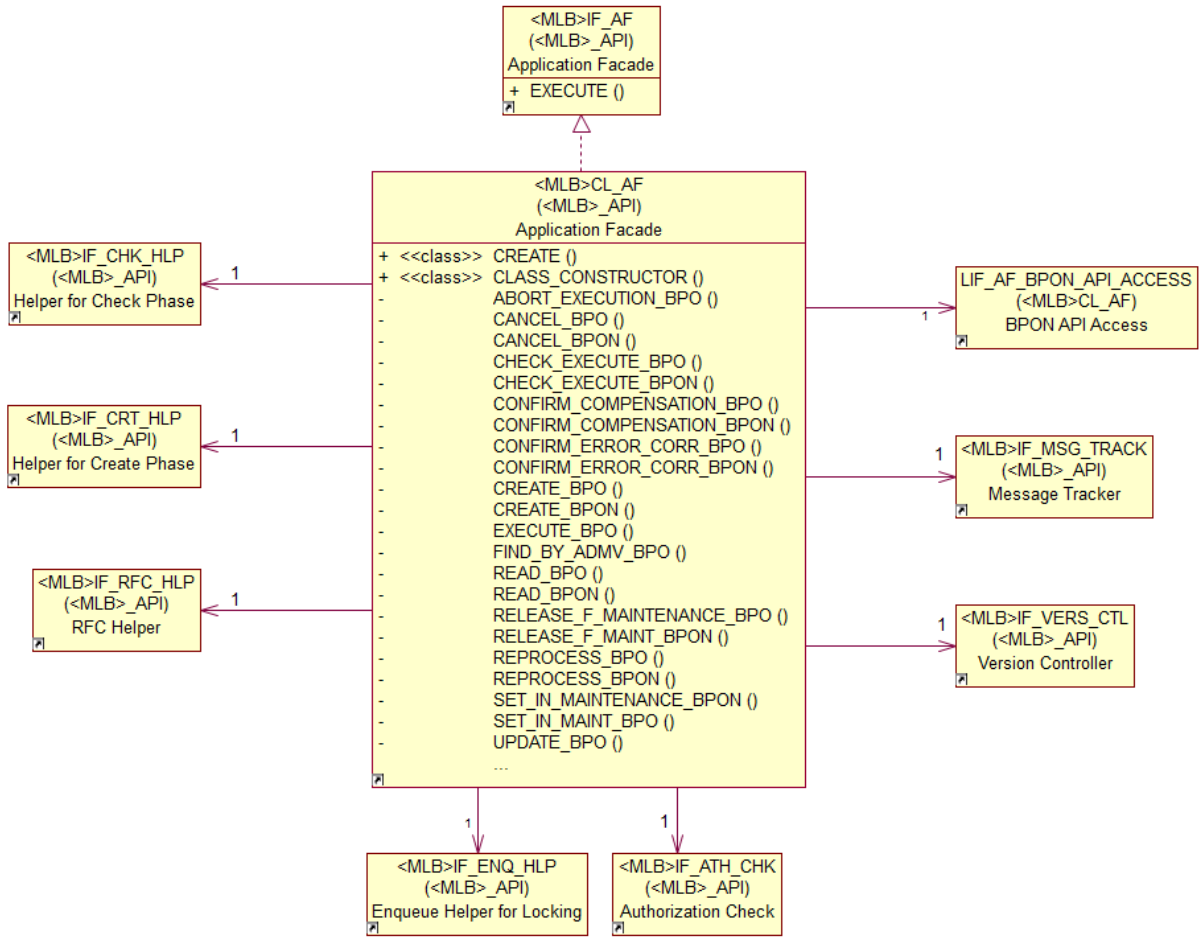


Figure 24: Application Façade

The façade class `<MLB>CL_AF` implements the façade interface `<MLB>IF_AF`, which provides an `EXECUTE` method that is used by all POT inbound service implementations.
The façade class provides an operation-specific private method for each service operation type. These operation-specific methods perform the complete call sequence that is necessary to execute the service call.

In order to support authorization checks and locking, the façade class uses corresponding helpers (shown at the bottom of the diagram). It also uses the APIs for BPO/BPON instance management and versioning (shown at the right hand side of the diagram). For phase execution, it uses separate helpers for the *Create*, *Check* and *Execute* phase (shown at the left hand side of the diagram).

Figure 25 shows an overall call sequence to visualize the responsibilities of the application façade:

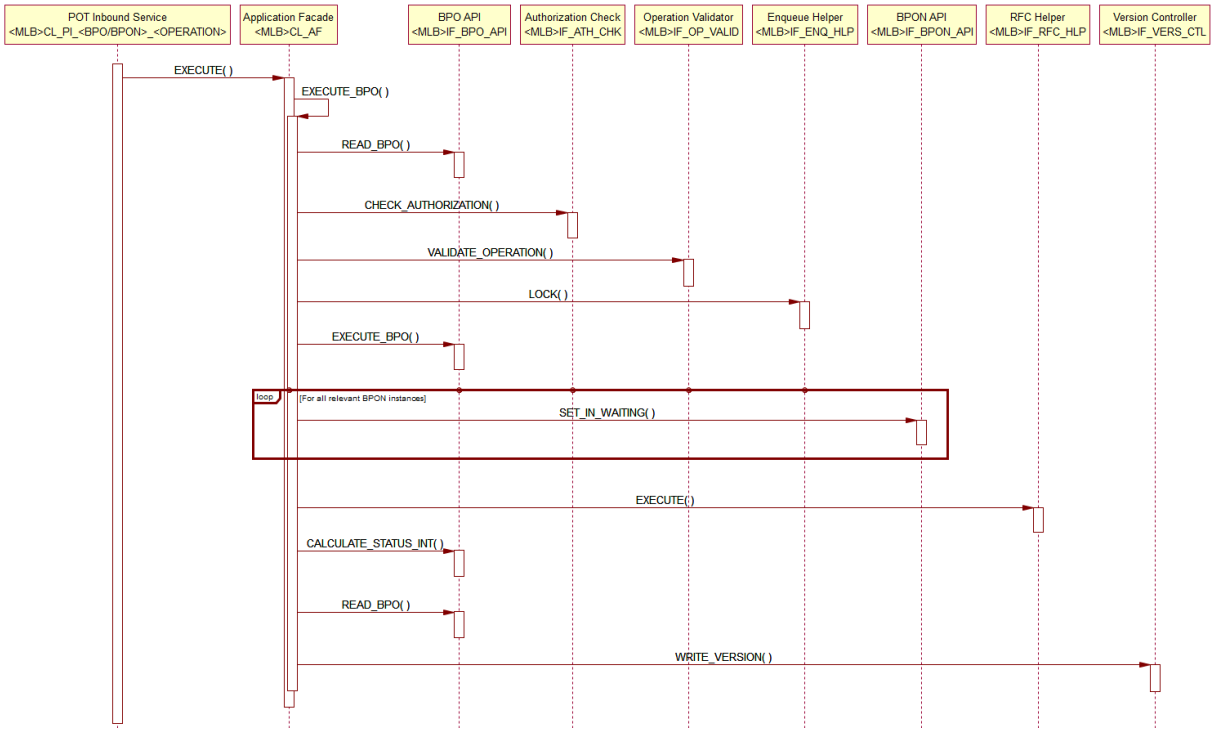


Figure 25: Application Facade for Operation CheckExecute - Call Sequence

The application façade is called by the POT inbound service via the common interface method `EXECUTE`. The application façade delegates this call to an operation-specific method, which in this case is `EXECUTE_BPO`. First, the BPO instance is read and an authorization check is performed. This check validates whether the POT status model allows for the execution of the operation. After that, the POT instance is locked to prevent other users or processes from changing the instance in parallel. The execution of the POT is started by calling the `EXECUTE_BPO` method from the BPO API and setting all relevant BPON instances to status *Waiting*.
The RFC helper then processes the phase execution in background by starting a new task. At the end of the sequence, the status of the BPO is calculated and a new version of the POT instance is written.

The remainder of this chapter will focus on the design of the APIs for BPO and BPON instance management and Versioning. In addition, the API for message persistence as well as the phase helper classes and their relation to the phase implementation (where the code-slots are located) will also be explained for the *Create*, the *Check* and the *Execute* phase.

6.2.1 API Design

Before the APIs of the application logic layer are discussed in detail, you should know that the API for BPO and BPON instance management as well as the Version API are built following the same layer design.

Figure 26 shows the design of the BPO API as an example:

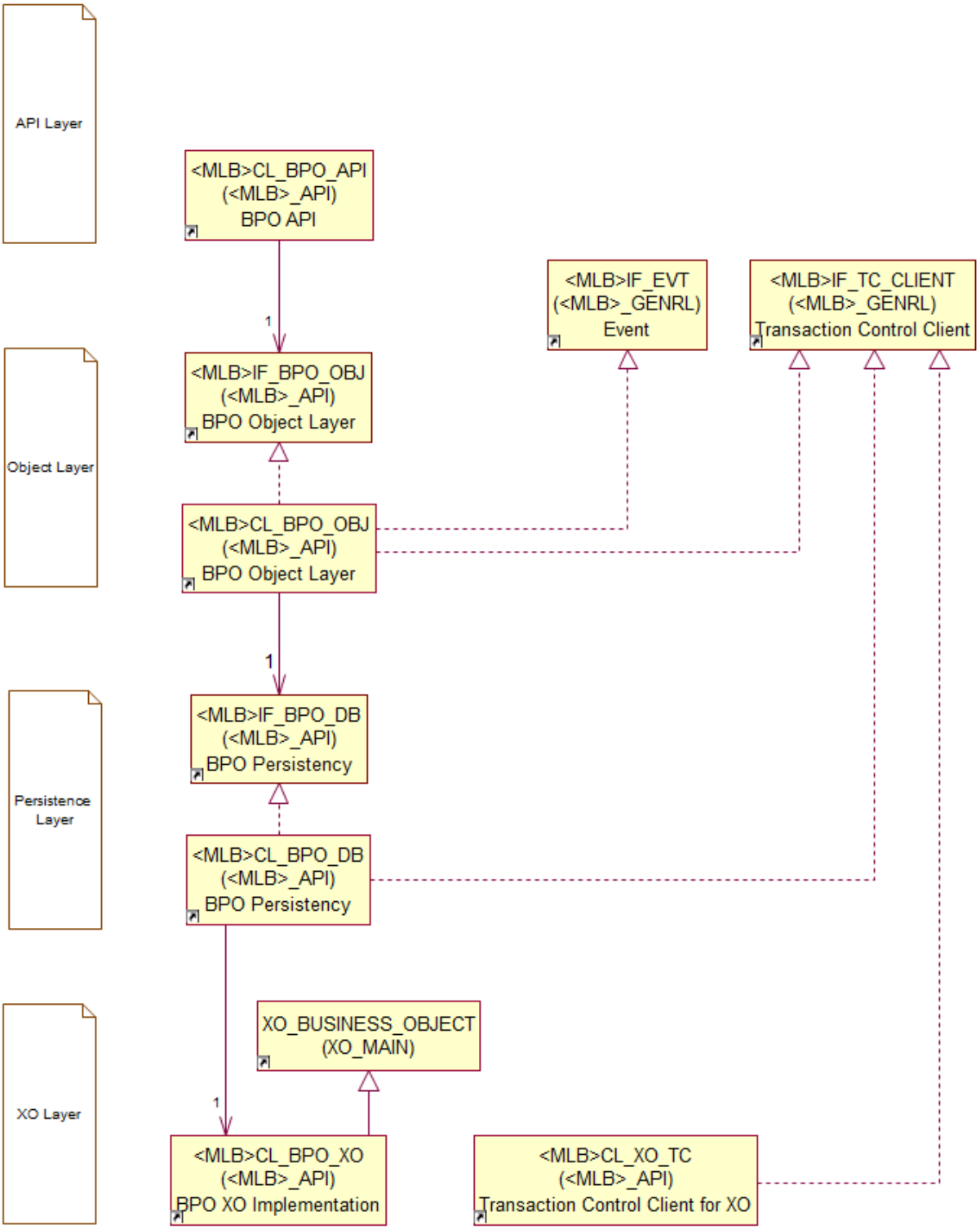


Figure 26: BPO API - Static View

The different layers have the following responsibilities:

The API Layer provides and implements the “public” interface, which is mainly used by the application façade and other helper classes. It is not intended to be used by other consumers such as code-slot implementations or any other custom code. It usually offers CRUD methods and other convenience methods via the API interface (`<MLB>IF_BPO_API` in our example). The implementing class `<MLB>CL_BPO_API` holds a reference to its corresponding interface from the object layer `<MLB>IF_BPO_OBJ`.

The Object Layer is responsible for buffering the data from the persistence layer and raises events in case data is changed. It basically offers CRUD methods via the interface <MLB>IF_BPO_OBJ. The implementing class contains the buffer and holds a reference to its corresponding persistence interface <MLB>IF_BPO_DB. In order to be able to raise events, the class <MLB>CL_BPO_OBJ implements the event interface <MLB>IF_EVT. The transaction control client interface <MLB>IF_TC_CLIENT is implemented to reset the buffer at the end of a transaction.

The Persistence Layer builds the bridge between the POT object layer and the XO framework. It draws new DB keys, and manages the buffering of XO object instances. It is based on the complete types, the total type and more specific interface types. The persistence interface usually offers methods like READ, WRITE and DELETE. The class implements the buffer and the connection to the XO framework. As this class also needs to reset its buffers at the end of a transaction, it also implements the transaction control interfaces <MLB>IF_TC_CLIENT.

The XO Layer transforms the data into DB representations. It takes care of creating, deleting, updating and reading entries in the POT database tables. The POT-specific XO implementation classes inherit from XO_BUSINESS_OBJECT and are registered in the POT-specific XO customizing. The transaction handling is done by a common XO transaction controller (<MLK>CL_XO_TC), which implements the transaction control client interface. At end of a transaction, the controller calls the corresponding XO framework methods to save or reset data.

6.2.2 Database Tables

Table 6-1 contains the POT database tables together with the corresponding APIs:

Table	Description
<MLB>_XO	Contains the instance data of a process object. The table is used to store all the instance data of the BPO and its BPONs. It is managed by the BPO and BPON API (described in sections 6.2.3 and 6.2.4).
<MLB>_XS	Used to store all the elements that are used as selection fields for search queries in a search-friendly manner. Whenever search-relevant fields of the table <MLB>_XO are updated, the corresponding BPO or BPON API needs to ensure that the table <MLB>_XS is also updated accordingly. The read access to this table is managed by the BPO API described in section 6.2.3.
<MLB>_XM	Used for tracking asynchronous messages. The table is managed by the message tracker described in section 6.2.6.
<MLB>_XV	Keeps the versions of all process object instances. The table is used to store the complete history of the process object instance throughout its lifetime. With each change a new entry is written (Type BPO_TOTAL + additional data). The corresponding API is described in section 6.2.5.

Table 6-1: Database tables of a POT

6.2.3 API for BPO

Apart from the generic API design described in section 6.2.1, this section describes the most important specifics of the BPO API. A detailed view is shown in Figure 27:

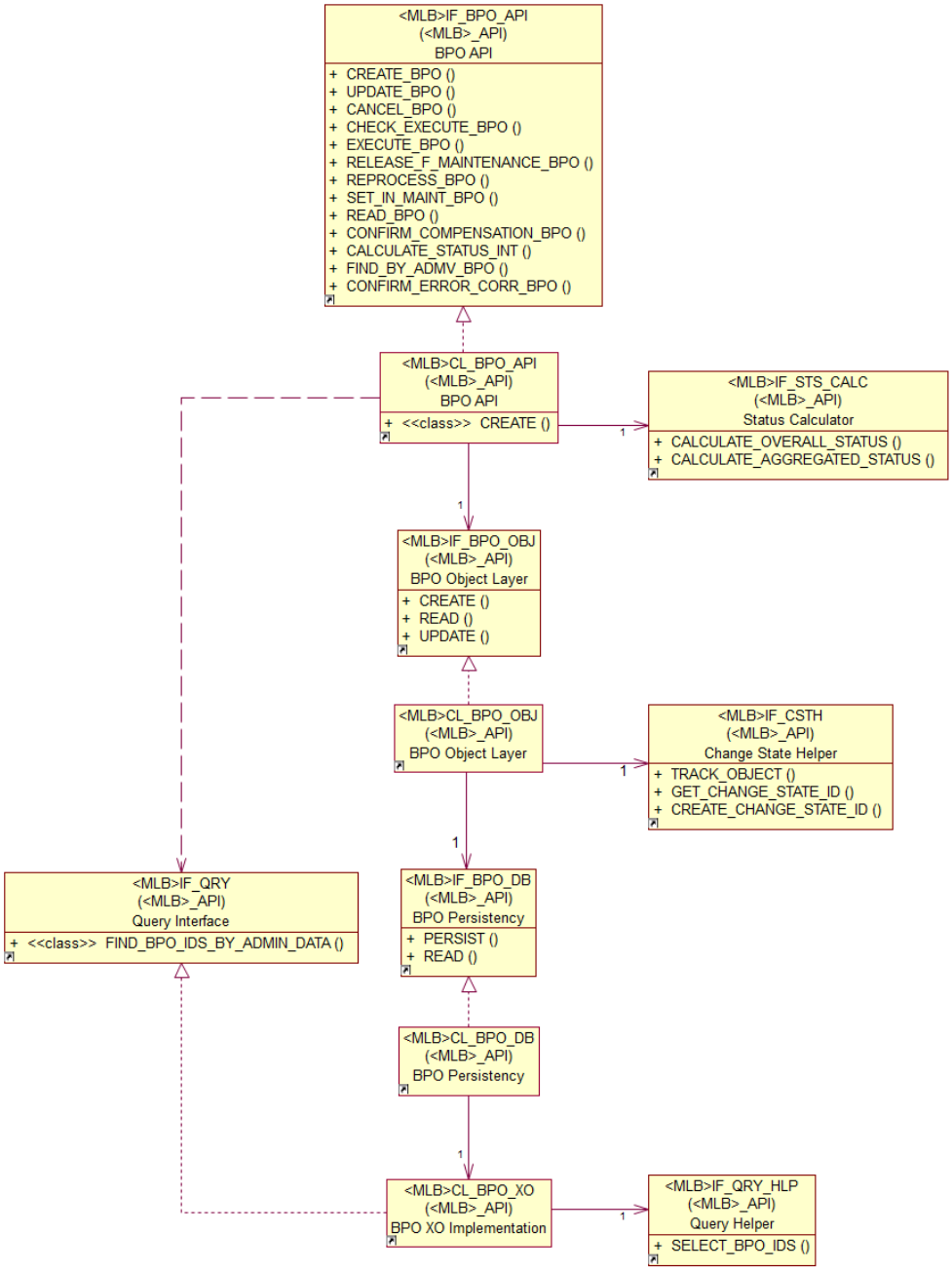


Figure 27: Specific Details of the BPO API - Static View

The API interface `<MLB>IF_BPO_API` contains all kinds of methods to manage data and the status of BPO instances.

Various helpers are used on all levels of the API stack:

For the calculation of the BPO overall status (explained in details in section 6.5.4), the API class uses the *Status Calculator* that calculates the overall status of the BPO based on the statuses of its BPON instances.

The object layer is responsible for the handling of the change state ID. For this purpose it uses the *Change*

State Helper interface <MLB>IF_CSTH.
The *Change State Helper* provides methods to track an object for subsequent changes (TRACK_OBJECT) and to retrieve a new change state ID whenever necessary. The method GET_CHANGE_STATE_ID will return the old change state ID if there has not been a change. The method CREATE_CHANGE_STATE_ID unconditionally returns a new ID for example when a new object is created. However, per LUW only one new change state ID is drawn for each BPO instance.

In order to implement the search feature of the BPO API (method FIND_BY_ADMV_BPO), the XO class implements the *Query Interface* <MLB>IF_QRY, which contains a method for finding BPO instances by administrative data. Even though this method is implemented by the XO class, the actual processing logic is not done by the XO framework but forwarded to the *Query Helper* <MLB>IF_QRY_HLP, which selects the BPO instances directly from the database <MLB>_XS.

6.2.4 API for BPON

Apart from the generic API design described in section 6.2.1, there is one important aspect of the BPON API that is worth knowing:

As shown in Figure 28, there is one class implementation for each BPON type. This is indicated by the placeholder <BPON>. All BPON implementations share the same interface <MLB>IF_BPON_API. The BPON API operates on database table <MLB>_XO.

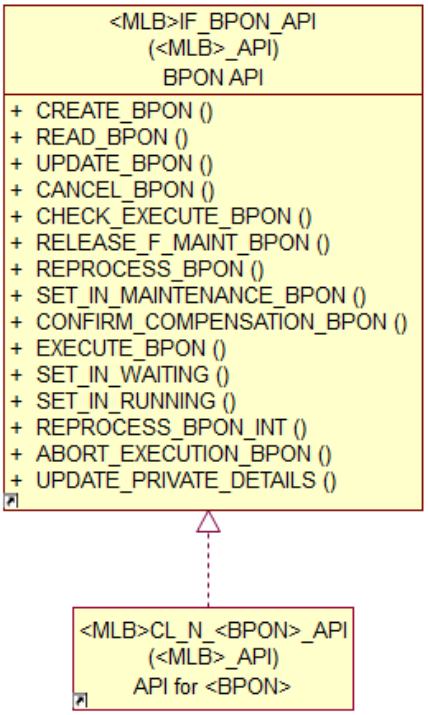


Figure 28: BPON API - Static View

6.2.5 Version API

In order to be able to reconstruct the complete history of a BPO instance, a complete image of the instance is written to the version database. The data from the version database is also used for change pointers and, consequently, for sending the information messages.

Figure 29 shows a static view of the versioning functionality:

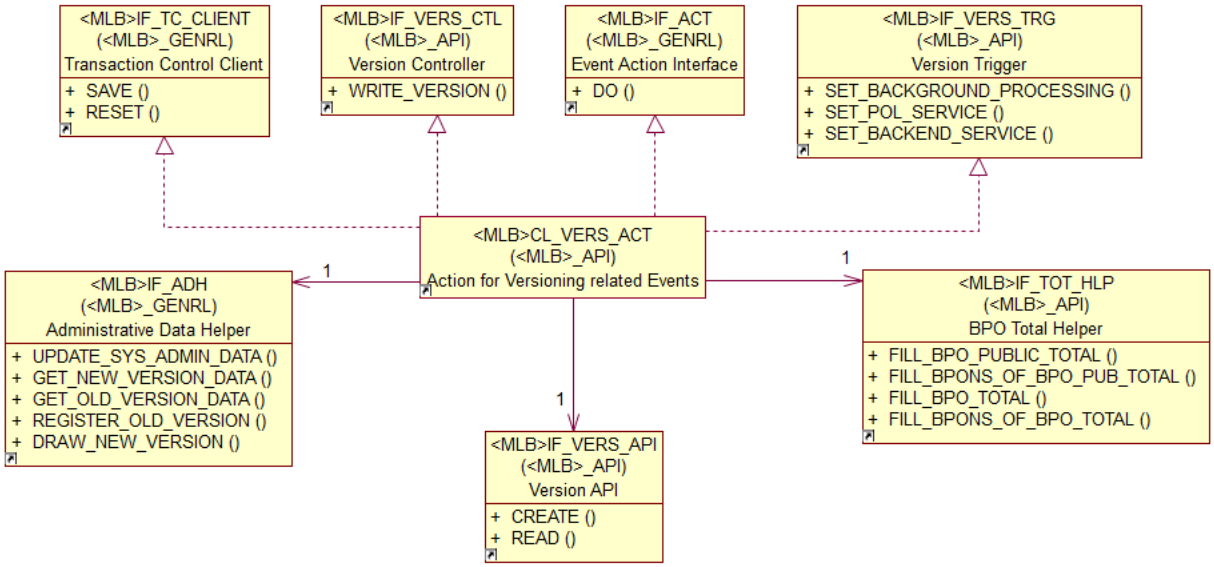


Figure 29: Versioning – Static View

The *Action Class* `<MLB>CL_VERS_ACT` plays a major role in versioning. It implements several interfaces exposed to different users. As an action, it implements the action interface `<MLB>IF_ACT`. The action is registered on changes and status changes of BPO and BPONs. If such an event occurs, the action class buffers this information.

A version is written each time the method `WRITE_VERSION` of the *Version Controller* `<MLB>IF_VERS_CTRL` is called. This interface is exposed to the application facade, which calls the method `WRITE_VERSION` at the end of the call sequence of a changing service operation. The version information is built by the *Total Helper*. This helper builds the "BPO Total" structure, which consists of the BPO instance data with all its BPON instances, including private administrative data. The action class writes the data to the database `<MLB>_XV` using the Version API `<MLB>IF_VERS_API`.

There must be a version trigger, as the information about the POT Service, the back-end service or background processing is a vital part of the version information. The *Version Trigger* interface is exposed to all possible channels (POT/back-end service implementation and RFCs). The trigger can only be set once per LUW.

6.2.6 Message Persistence

Figure 30 shows a static view of the *Message Tracker*:

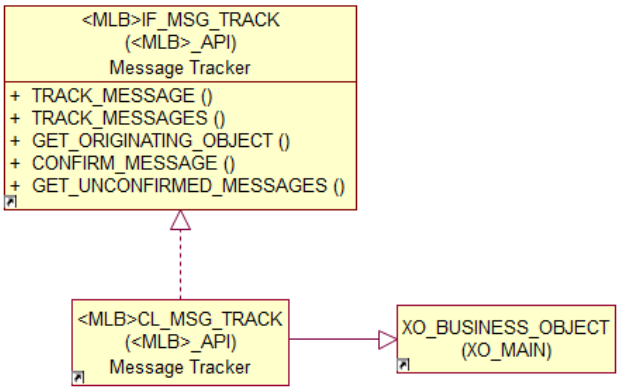


Figure 30: Message Persistence - Static View

The message tracker keeps track of outgoing asynchronous back-end calls for a BPON and it is used by the BPON facade to assign an incoming confirmation messages to the corresponding BPON. Each time an asynchronous request is sent out, the message tracker stores its message ID and the corresponding BPON UUID. This is done by the methods `TRACK_MESSAGE` and `TRACK_MESSAGES`.

If an inbound confirmation arrives, the status of the corresponding BPON must be set accordingly. This is done by the method `GET_ORIGINATING_OBJECT`. In case the confirmation message could be processed successfully, the method `CONFIRM_MESSAGE` is used in order to tick off this message.

Details about the handling of asynchronous messages is discussed in section 6.3.

The *Message Tracker* class `<MLB>CL_MSG_TRACK` acts as an XO class by implementing `XO_BUSINESS_OBJECT`. This class operates on the database table `<MLB>_XM`. The Message Tracker API can be considered as an exception from the standard API design, as it omits the object and the persistence layer.

6.2.7 Phase Handling

The processing of a POT can be divided into the three phases CREATE, CHECK and EXECUTE.

The execution of each phase is triggered by different service operations, this will be discussed in the following sections.

Chapter 7 contains further details on the phase implementation from a code slot implementer's point of view.

6.2.7.1 Create

The *Create* phase is triggered by the following service operations:

- `Create<BPO>`
- `Create<BPON>`
- `Update<BPO>` (in case a BPON is created using this BPO-level update service)

Subsequently, the corresponding Create Phase Helper is called by the Application Façade and manages the processing of the *Create* phase. A static view is shown in Figure 31:Figure 31

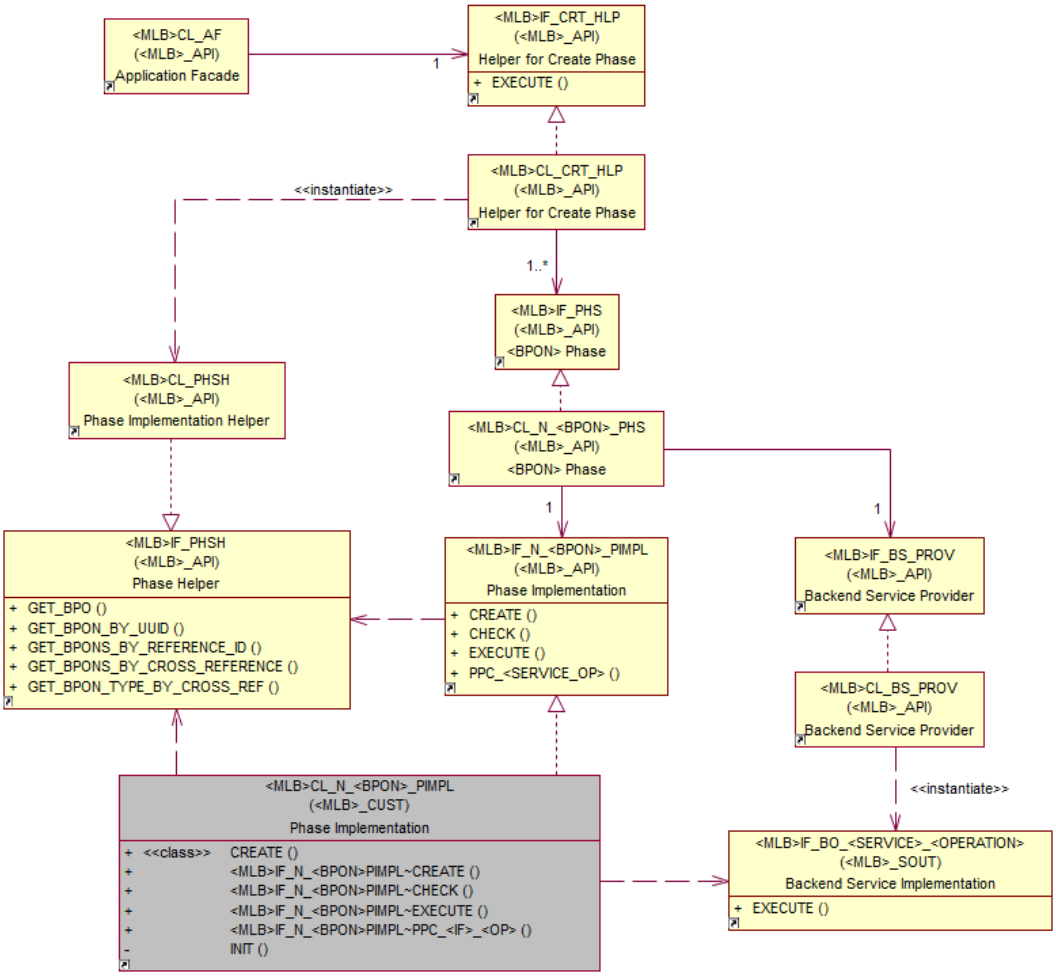


Figure 31: Create Phase Helper – Static View

The Create Phase Helper holds references to all BPON-specific phase classes via the generic phase interface `<MLB>IF_PHS`. The phase classes delegate to the BPON Phase Implementation, which contains the code slots for back-end service execution (marked in grey). If back-end services are defined for the *Create* phase, these services are provided by the Backend Service Provider and can be used by the Phase Implementation. In all code slots, the Phase Helper can be used to access other BPONs.

The runtime behavior of the create phase helper depends on the calling operation. The call sequence for `Create<BPO>` is shown in Figure 32.

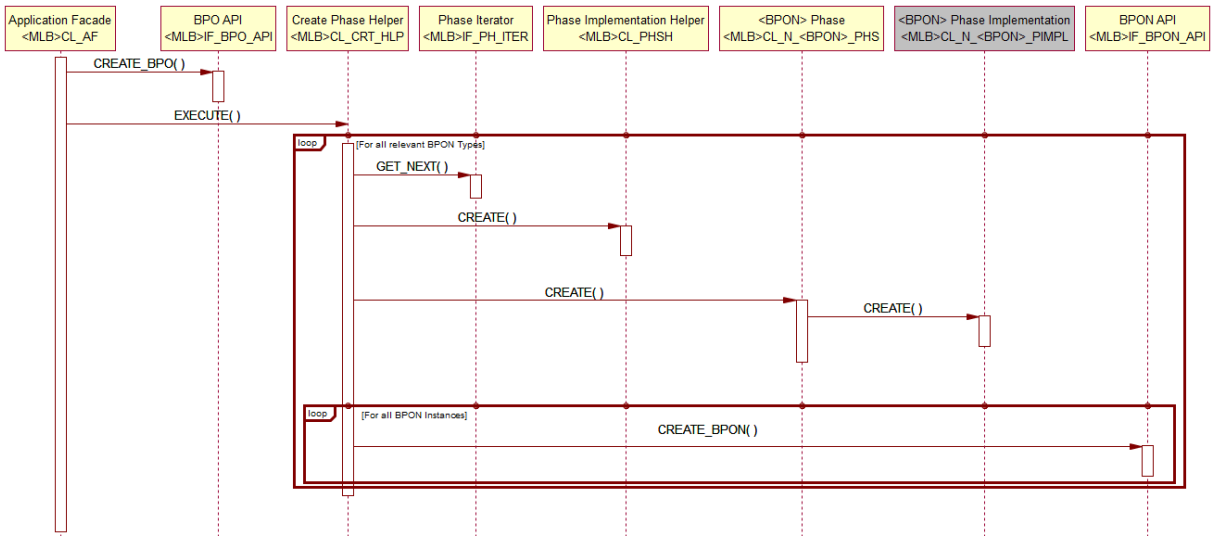


Figure 32: *Create* Phase for `Create<BPO>` - Call Sequence

In this case, the *Application Façade* first creates a new BPO instance by calling the `CREATE` method of the *BPO API.* The creation of the BPON instances is handled by the *Create phase Helper*. This helper processes all BPON types in the defined processing order and calls the *Phase Implementation* class (marked in grey) for each BPON type. The BPON data that is returned by this class is then created as a BPON instance by calling the corresponding BPON API.

For the operation `Update<BPO>`, the call sequence is basically the same, with the difference that in this case the *Application Façade* does not call the `CREATE_BPO`, as the BPO already exists.

Figure 33 shows the call sequence for when the *Create phase Helper* is triggered by the `Create<BPON>` operation.

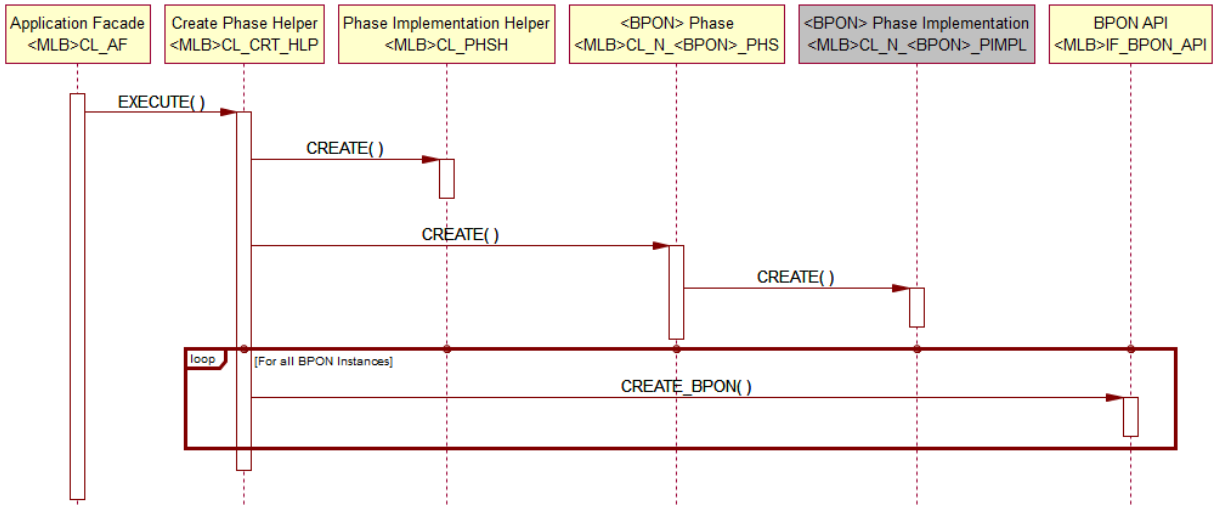


Figure 33: *Create* phase for `Create<BPON>` - Call Sequence

In the given case, the Create Phase Helper executes the *Create* phase for a specific BPON type. Therefore, compared to the call sequence for `Create<BPO>`, there is no loop over all the BPON types and no phase iterator is required.

6.2.7.2 Check

The *Check* phase is triggered by the following service operations:

- `Check<BPO>`
- `Check<BPON>`

The Check Phase Helper works similar to the Create Phase Helper. A static view is shown in Figure 34:

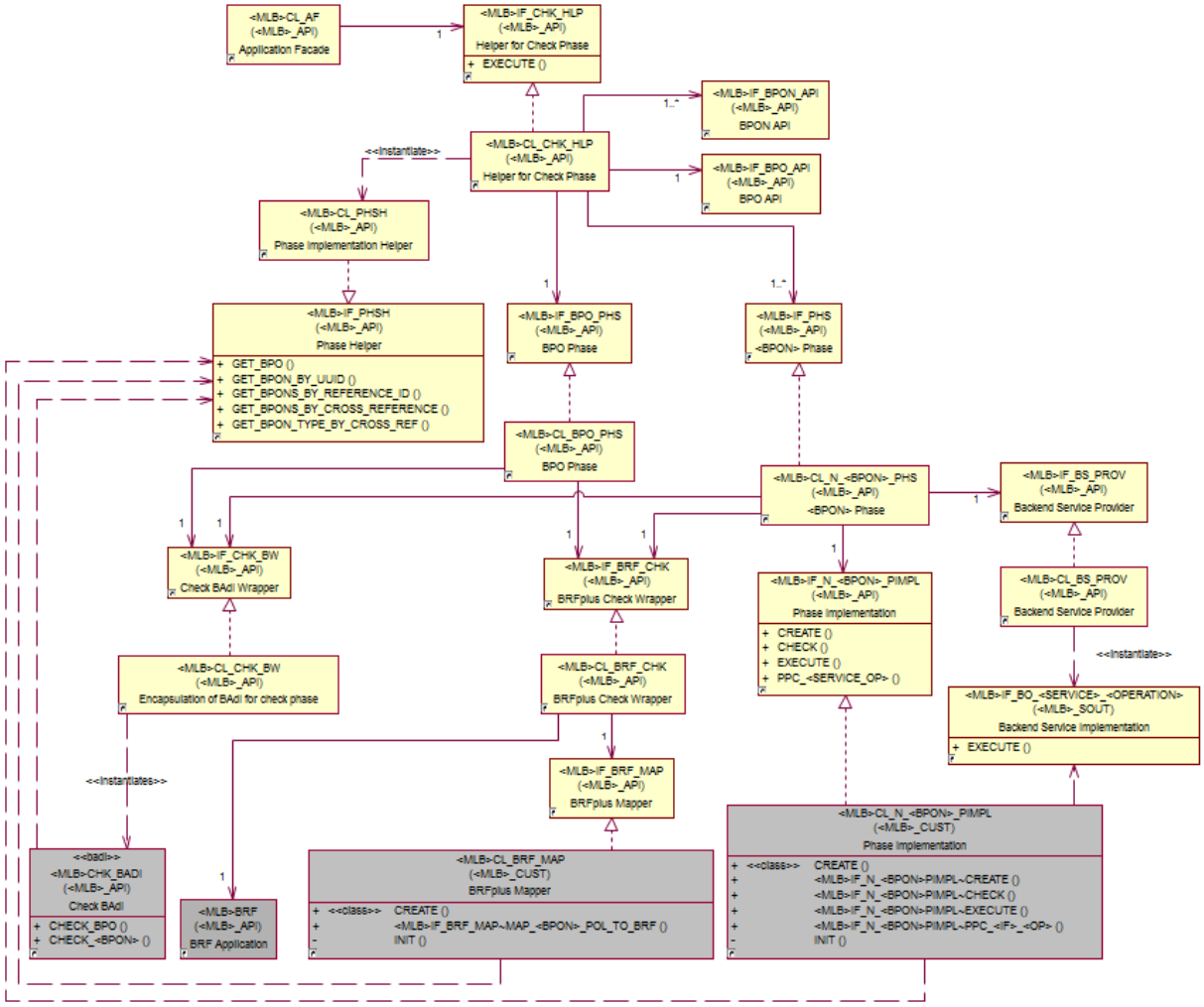


Figure 34: Check Phase Helper – Static View

The major difference in the implementation is that there is no phase iterator required, as for the *Check* phase there is no processing order defined for the nodes. In addition to the BPON Phases, the helper also uses the *BPO Phase*, which is responsible for performing checks on BPO level. In order to reflect the different check options on BPO and BPON level, the phase classes use a Check BAdI Wrapper and a BRF Check Wrapper. The wrappers contain check methods for each BPON type for which one of the corresponding check option has been selected (BRFplus check, BAdI). Both wrappers delegate to the corresponding custom implementations (marked in grey).

If back-end services are defined for the *Check* phase, these services are provided by the Backend Service Provider and can be used by the Phase Implementation. In all code slots, the Phase Helper can be used to access other BPONs.

During the *Check* phase, the following checks are performed in the order specified below:

1. **Standard Validation**
2. **Check BAdI**
3. **BRFplus rule-based checks**
4. **Check services**

Figure 35 shows the overall call sequence for the operation `Check<BPO>`.

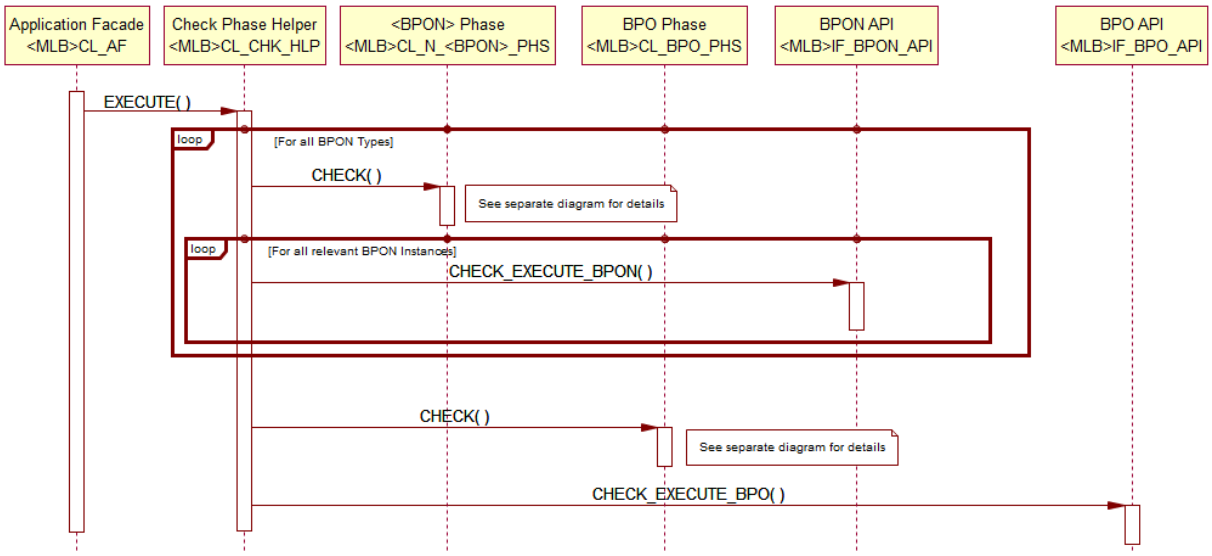


Figure 35: *Check* Phase for Operation Check<BPO> - Overall Call Sequence

First, the Check Phase Helper loops over all the BPON types and calls the `CHECK` method of the corresponding phase class. (A detailed call sequence for this call is shown in a separate diagram and will be explained later). The phase class returns a list of check results. These check results are persisted by calling the method `CHECK_EXECUTE_BPON` of the corresponding BPON API for each BPON instance.

After the BPONs checks are executed, the Check Phase Helper executes the checks on BPO level. (The detailed call sequence for this call is also shown in a separate diagram.)The check result is then persisted by calling the method `CHECK_EXECUTE_BPO` of the BPO API.

The BPON part of the *Check* phase is shown in more detail in Figure 36:

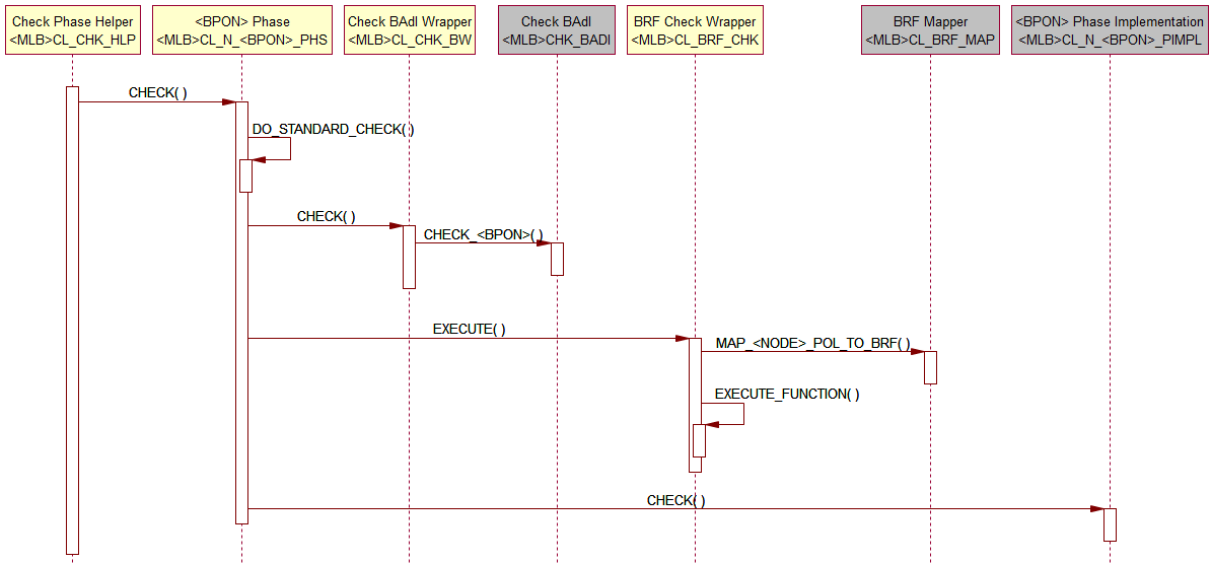


Figure 36: BPON-related Checks - Detailed Call Sequence

During the check of a BPON type, the following checks are executed in the order specified below: (Objects that contain custom code are marked in grey.)

1. The <BPON>Phase class executes a standard validation to check the cross-references of a BPON instance.
2. The Check BAdI Wrapper is called to trigger the corresponding check method for the BPON type (indicated by `CHECK<BPON>`).
3. The BRF Check Wrapper triggers the execution of the BRFplus functions. These functions execute all the assigned custom BRFplus rule sets (if any).

   In order to fill the BRFplus context, the BRF Mapper maps the BPON complete type to the DDIC structure that has been defined for BRFplus.
4. The `CHECK` method of the Phase Implementation is called to execute the assigned check services (if any).

The resulting messages of all checks mentioned above are collected and returned. However, if the standard check (1) fails, the subsequent checks are not executed.

Once the BPON part has been checked, the BPO-related checks are executed. The BPO part of the *Check* phase is shown in more detail in Figure 37:

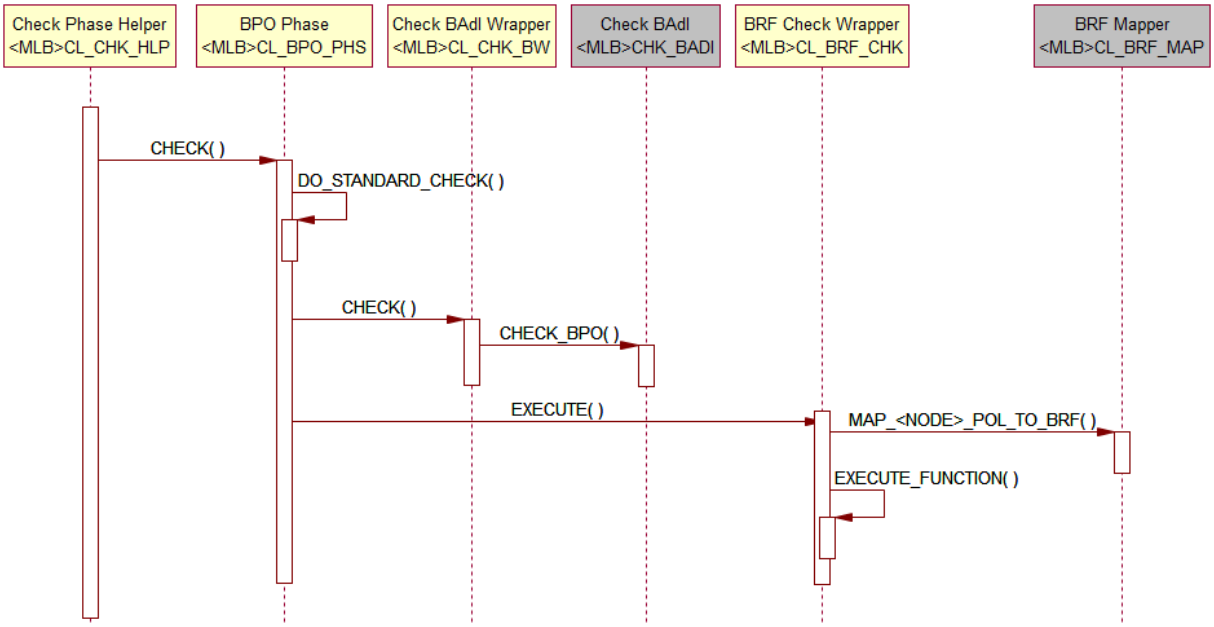


Figure 37: BPO-related Checks - Detailed Call Sequence

The execution of checks on BPO level basically contains the same steps as for the BPON checks:

1. The BPO Phase class executes a standard validation to check the cardinalities for all BPON types.
2. The Check BAdI Wrapper is called to perform the BAdI checks on BPO level (method `CHECK_BPO`).
3. The BRF Check Wrapper triggers the execution of the BRFplus functions.

6.2.7.3 Execute

Due to the asynchronous execution of the back-end changes, there are two key aspects that are important to note for the *Execute* phase of a POT:

- The operation `Execute<BPO>` is a synchronous call, whereas the execution of the BPONs towards the back end is performed asynchronously.

  This asynchronous processing is enabled by an RFC Helper.
- The requests for back-end object changes that are sent by the POT implementation must be tracked in order to make sure that the incoming confirmations of these requests can be mapped accordingly.

The *Execute* phase is handled by the Execute Phase Helper. A static view is shown in Figure 38:

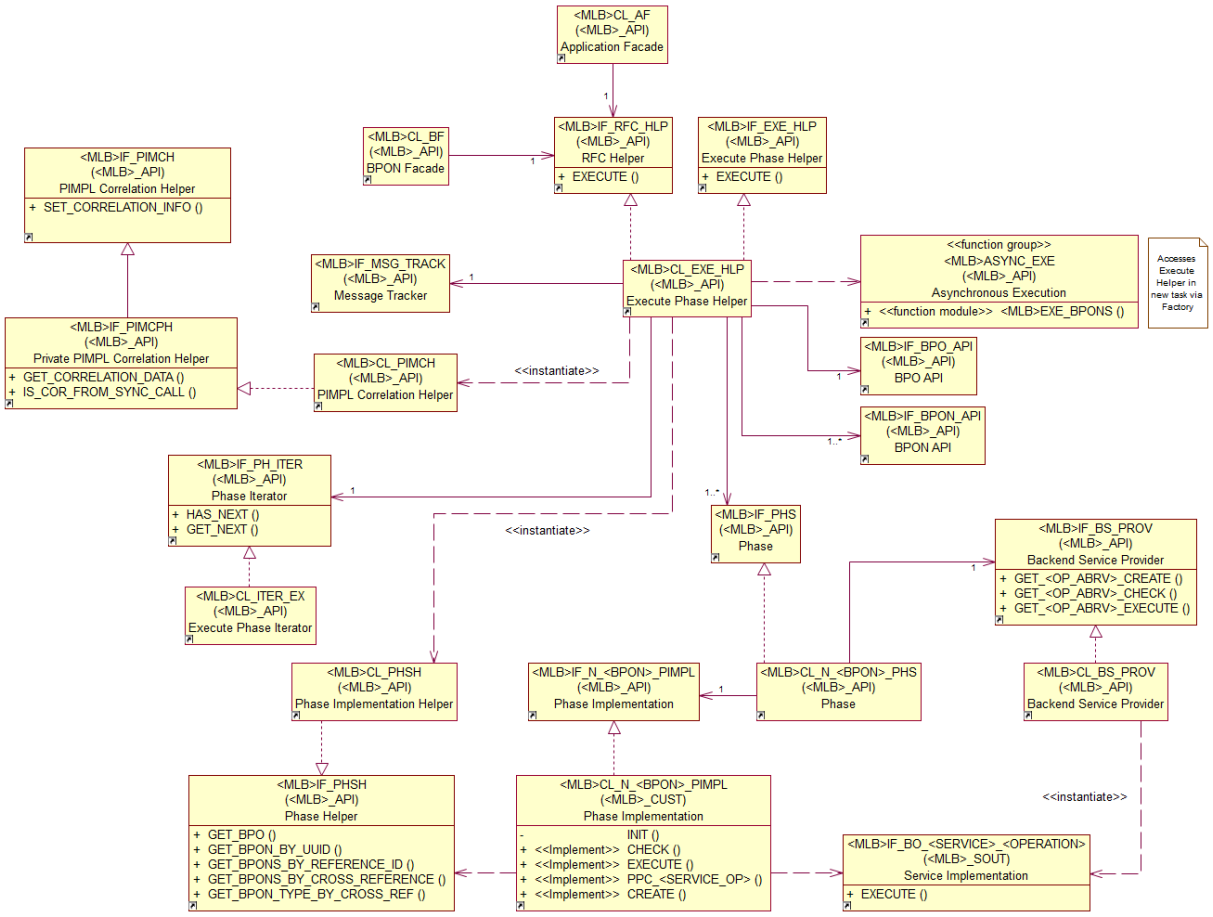


Figure 38: Execute Phase Helper – Static View

The *Execute* phase is triggered by the service operation `Execute<BPO>` and by all inbound services that are responsible for processing back-end confirmations.

Consequently, there are two users of the *Execute* phase helper: The Application Façade that handles the service operation `Execute<BPO>` and the BPON Façade that is responsible for the handling of asynchronous confirmation messages from the back end.

The Execute Phase Helper holds a reference to the Message Tracker to keep track of asynchronously sent messages (see section 6.2.6). As for the other phases it accesses BPO and BPON via the usual APIs and delegate to the Phase Implementation (marked in grey) in which the code-slots are located and the back-end service execution is done.

The Execute Phase Helper implements two interfaces:

- `<MLB>_IF_EXE_HLP` - encapsulates the processing logic of the *Execute* phase
- `<MLB>_IF_RFC_HLP` - encapsulates the processing logic in an asynchronous mode

Figure 39 shows the call sequence for the `Execute<BPO>` method:

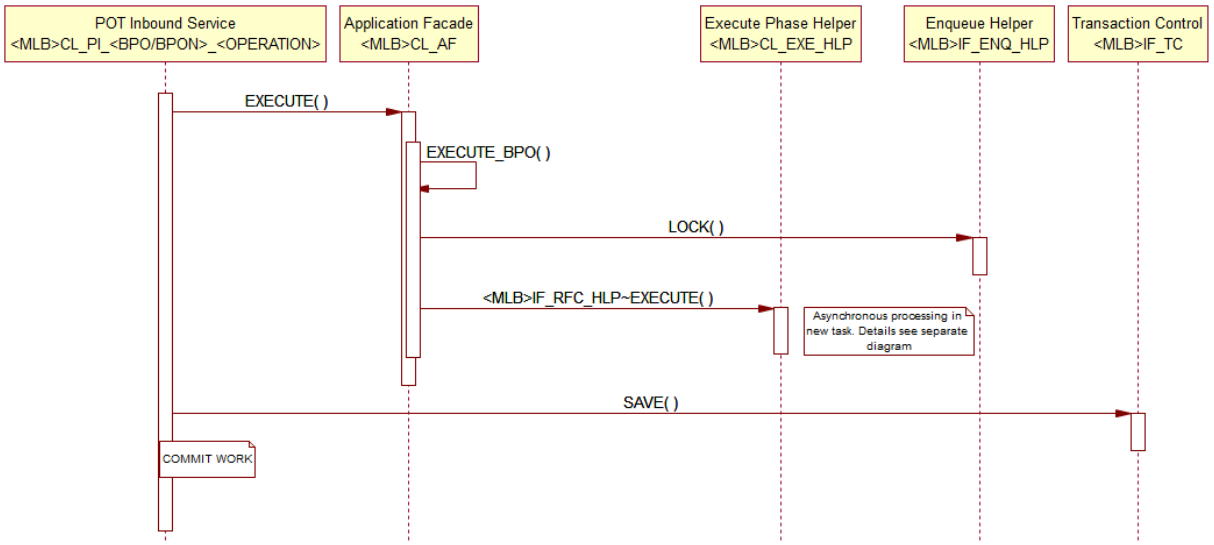


Figure 39: Execute Phase Triggered by Operation `Execute<BPO>` - Call Sequence

The POT Inbound Service for `Execute<BPO>` calls the Application Façade, which first locks the BPON instance and triggers the phase execution via the RFC Helper. The service implementation calls the `SAVE` method and triggers a `COMMIT WORK`.

The call to the Execute Phase Helper is shown in more detail in Figure 40:

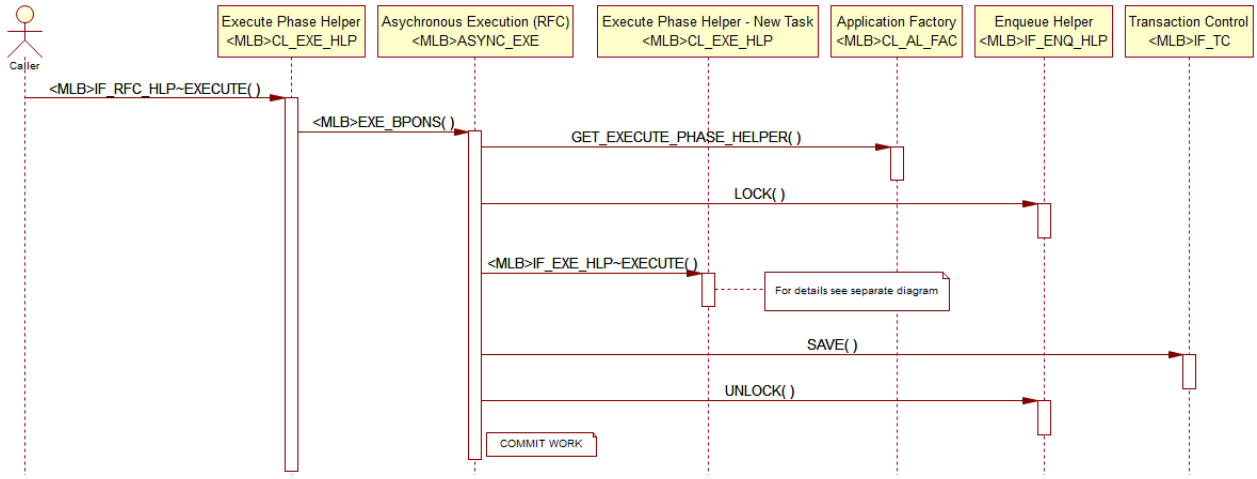


Figure 40: Asynchronous Execution via RFC – Overall Call Sequence

The execution is handed over to a function module by calling `<MLB>EXE_BPONS`. This call is performed as a background task with destination '`NONE`'.

Within the newly created session, the function module gets the Execute Phase Helper from the Application Factory and triggers the execution via interface `<MLB>_IF_EXE_HLP`.

After successful execution, the method `SAVE` of the Transaction Control interface is called and a `COMMIT WORK` is triggered to save all changes.

To lock and unlock the BPO instance during execution, the corresponding methods from the Enqueue Helper are used.

Note that, for the sake of simplicity, versioning is not part of this diagram.

The call of the Execute Phase Helper (in a new task) is shown in more detail in Figure 41:

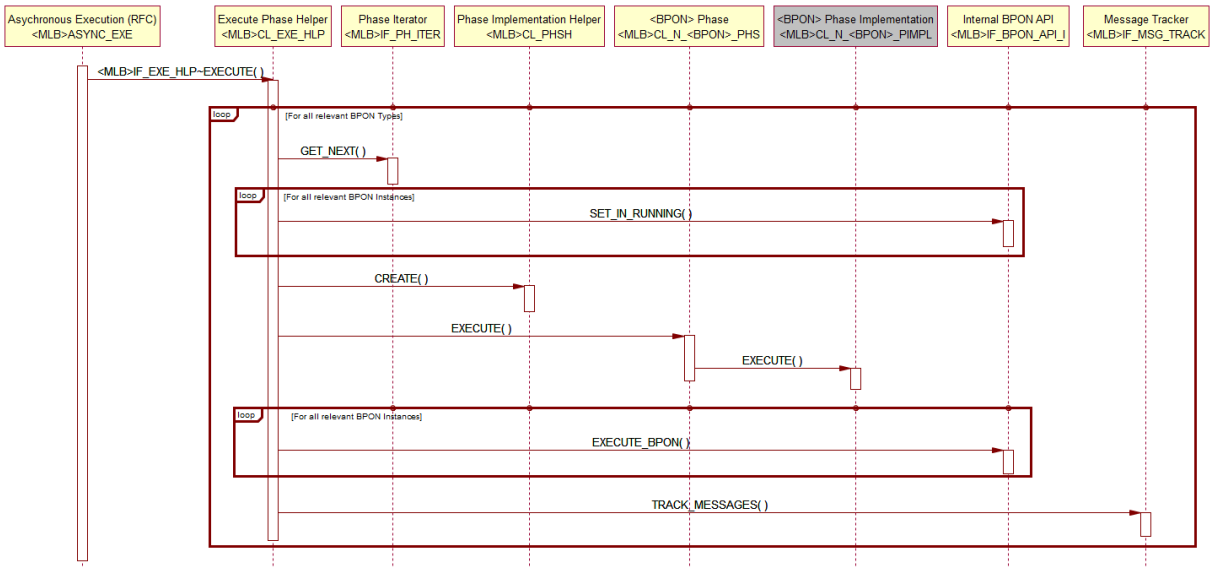


Figure 41: Asynchronous Execution via RFC – Detailed Call Sequence

The Execute Phase Helper (called via the *Execute* phase helper interface) performs the following steps for each node type in the right order by using the `GET_NEXT` methods of the Phase Iterator:

First, all relevant BPON instances are prepared for execution by setting their internal status to *Running*. This status means that the instances are released for execution but have not yet been processed by the Phase Implementation.

After the Phase Implementation has been called, the method `EXECUTE_BPON` of the corresponding BPON API is called. If the execution of a BPON instance involves any asynchronously sent messages, these messages are tracked by calling the `TRACK_MESSAGES` method. In case that exceptions are raised by the Phase Implementation, all instances are set to status *Failed*.

Another way to trigger the *Execute* phase is to process a back-end confirmation message. This is handled by the BPON Façade.

Figure 42 shows a static view of the BPON Façade:

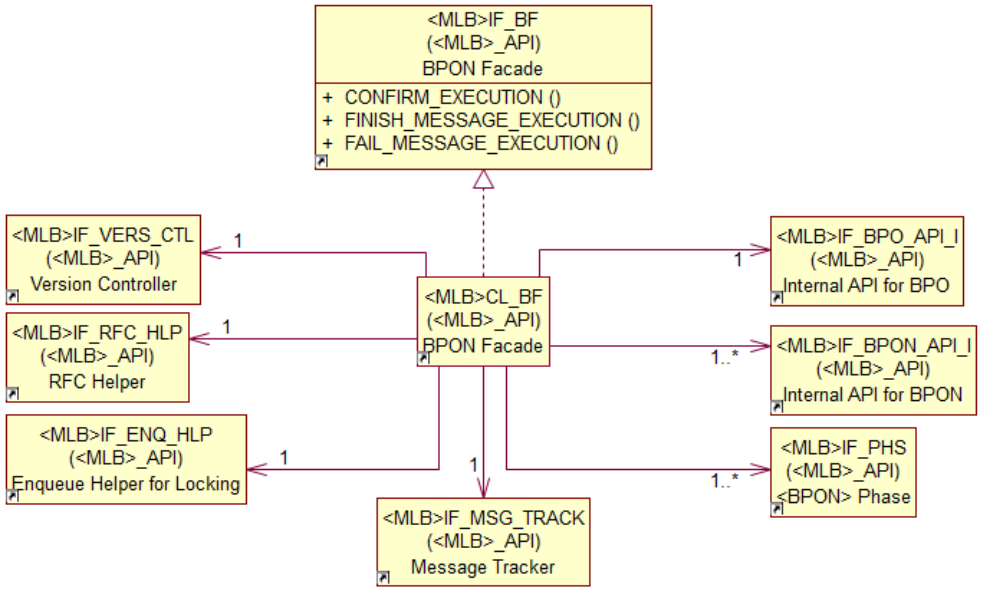


Figure 42: BPON Facade - Static View

The BPON Facade holds references to all BPON APIs and phase interfaces and also to some additional helpers that are required for processing a confirmation message.

A call sequence is shown in Figure 43:

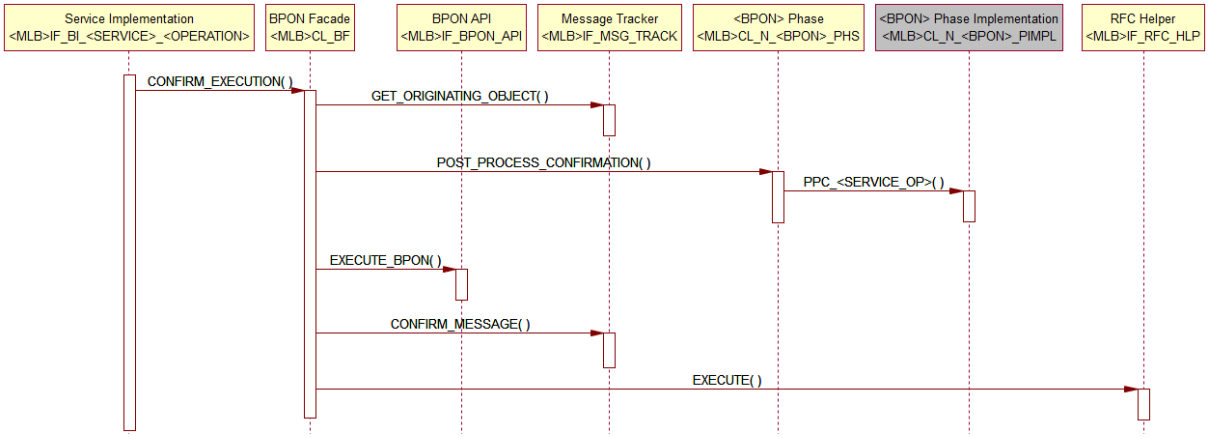


Figure 43: Processing of Incoming Asynchronous Confirmations – Call Sequence

After the BPON Façade has been triggered by the Service Implementation via the `CONFIRM_EXECUTION` method, it first determines the corresponding BPON instance that the confirmation message is associated with. This is done by the `GET_ORIGINATING_OBJECT` method of the Message Tracker.

In a next step, the corresponding Phase Implementation for post-processing the confirmation message is called. The result of the phase implementation is persisted by calling the method `EXECUTE_BPON` of the corresponding BPON API. The confirmation message is ticked off by calling the `CONFIRM_MESSAGE` method of the Message Tracker.

Finally, the Execute Phase Helper is called in order to trigger the processing of any pending BPONs to continue the execution of the BPO (as shown in Figure 41).

6.3 Communication Layers

The communication layers (Figure 44) are primarily responsible to connect the POT application logic to external components via enterprise service interfaces and ABAP Channels.

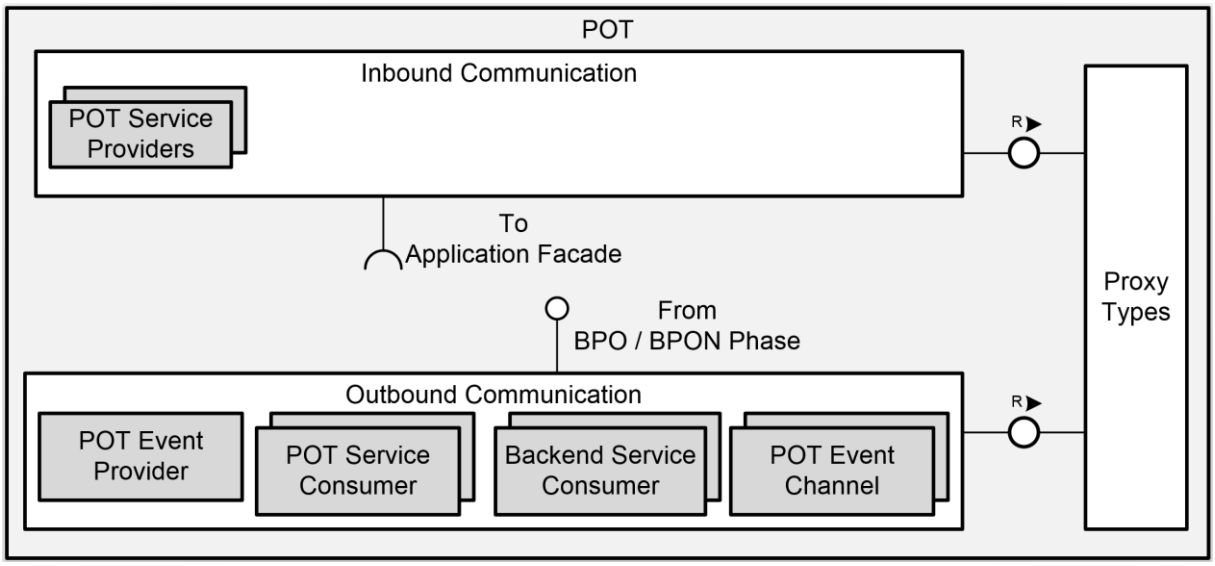


Figure 44: Service Implementation Layers

A POT provides services that are either called by external consumers (like BPM or other UIs) or called by POT editing UIs. The editing UIs also use services to interact with a POT and do not make any direct calls to the POT implementation. All the services provided by a POT are generated into ESR by POB. POB also generates the implementation for these services.

In addition to providing services, a POT also consumes services of the back ends during the POT phase implementation. POB also generates the ESR artifacts and the corresponding implementation for outbound service consumers for back-end services, the service consumers for calling the POT provider side itself (required for editing UI calls) and it generates the Event Provider service interface (to inform of the status changes of a POT to any interested listener applications like BPM) and ABAP Channels (to inform of POT changes and status changes of a POT to any interested listener based on WebSocket protocol).

The following sections provide the details of the POT inbound and outbound communication.

6.3.1 Inbound Communication

Figure 45 shows the standardized service interfaces and operations that are provided by a POT:

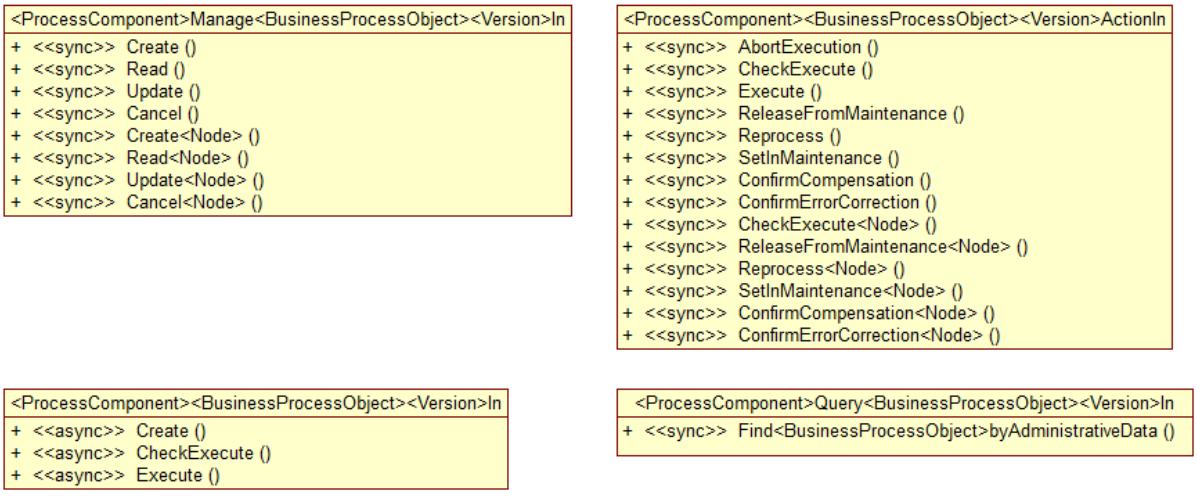


Figure 45: Overview of the Services Provided by a POT

The design of synchronous POT service implementations is described in detail in section 6.3.1.1. The asynchronous POT service implementations are described in section 6.3.1.2.

Besides the synchronous POT standard services, a POT might contain several back-end related asynchronous inbound services to handle confirmation messages from the back-end. The service implementation design is described in detail in section 6.3.1.3.

6.3.1.1 POT Inbound Services (Synchronous)

Figure 46 shows a static view on a POT standard synchronous inbound service implementation:

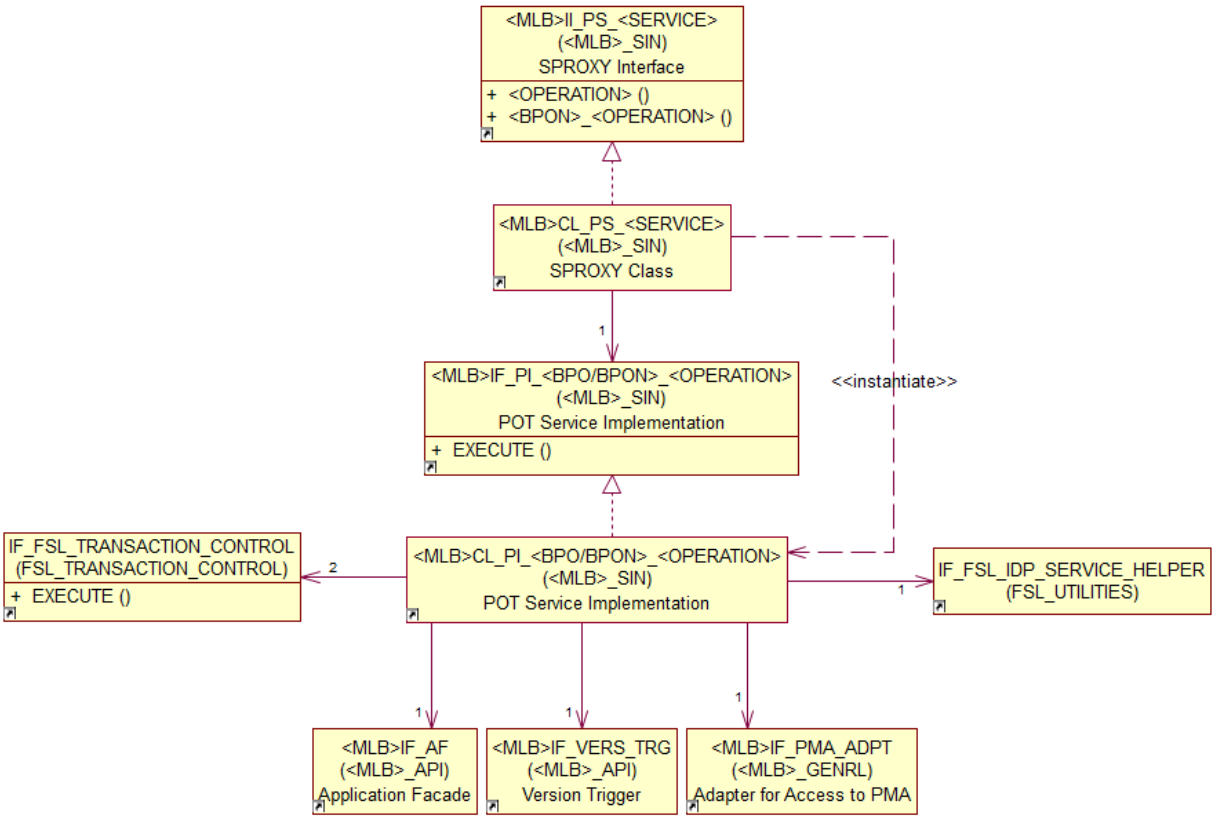


Figure 46: POT Standard Synchronous Inbound Service Implementation - Static View

The service implementation shown here is a changing service that follows the request/confirmation pattern. POB generates one service implementation interface (and the implementing class) for each service operation. The POT Service Implementation is directly instantiated by the SPROXY Class.

The POT Service Implementation covers the following aspects:

- Transaction control, which means that the final transaction end event is triggered by the POT Service Implementation. This is supported by using the transaction control interface from FSL, which that is implemented twice: One implementation for `RESET/ROLLBACK` and one implementation for `SAVE/COMMIT`.
- Idempotency, to support exactly one execution of messages in case the same message is sent twice. This is supported by the IDP Services Helper from FSL.
- Execution of the POT application logic using the Application Façade.
- Provision of information to the Process Observer during service execution using the PMA Adapter.

- Setting the version information using the Version Trigger.

The call sequence is shown in Figure 47:

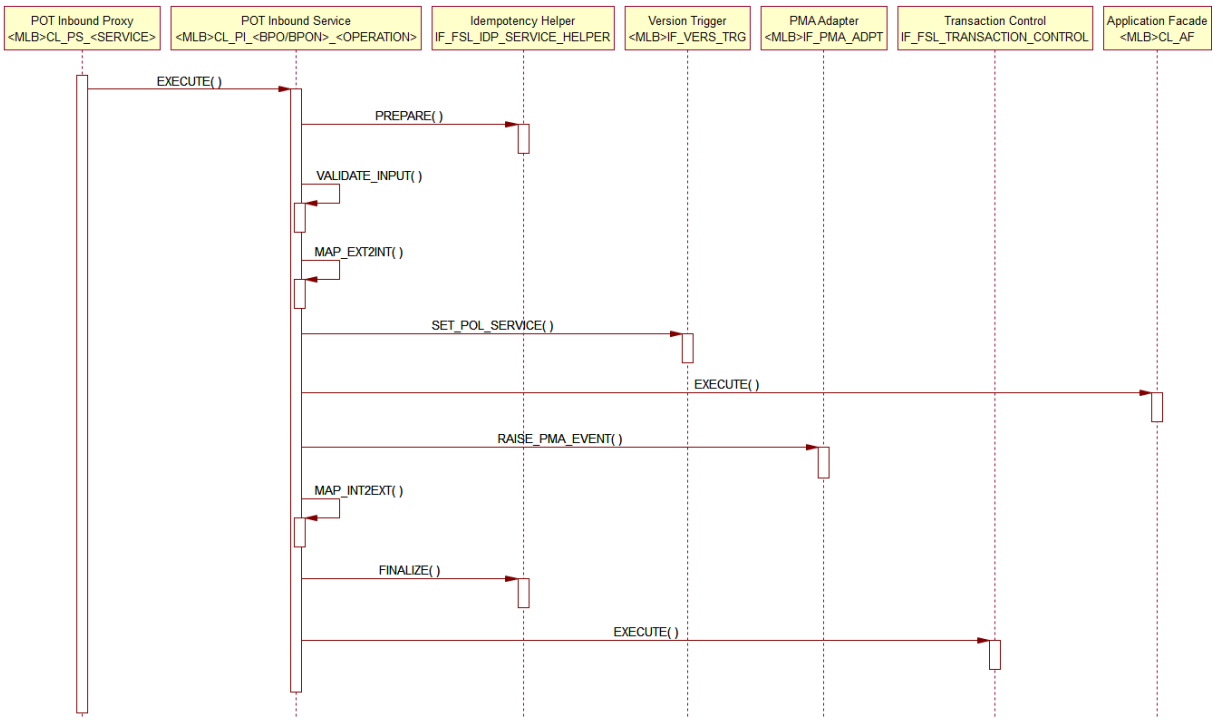


Figure 47: Synchronous POT Standard Inbound Service (Request/Confirmation) – Call Sequence

The execution of the service implementation is triggered by the POT Inbound Proxy by calling the interface method `EXECUTE` of the POT Service Implementation.

The method `PREPARE` checks whether or not the request has already been processed. If it has, the response for the already processed request is returned and the processing is finished. If the request has not yet been processed, the execution of the service implementation continues as follows:
First, the inbound request is validated to make sure that the data is plausible and fulfils the service contract. The inbound request is then mapped from the external representation (generated by SPROXY) to the internal DDIC representation based on which the Application Façade is called.

After the execution of the Application Facade, the information about the execution of the service is provided to the Process Observer by calling the `RAISE_PMA_EVENT` method. The response data is then mapped from the internal DDIC structure back to the external representation (generated by SPROXY).
Finally, the `FINALIZE` method stores the UUID of the request (taken from the message header) together with the corresponding response message. All changes are committed via the FSL transaction control interface.

Note that transaction control, idempotency and versioning are not required for read-only service operations (query/response pattern).

6.3.1.2 POT Inbound Services (Asynchronous)

Figure 46 shows a static view on a POT standard asynchronous inbound service implementation:

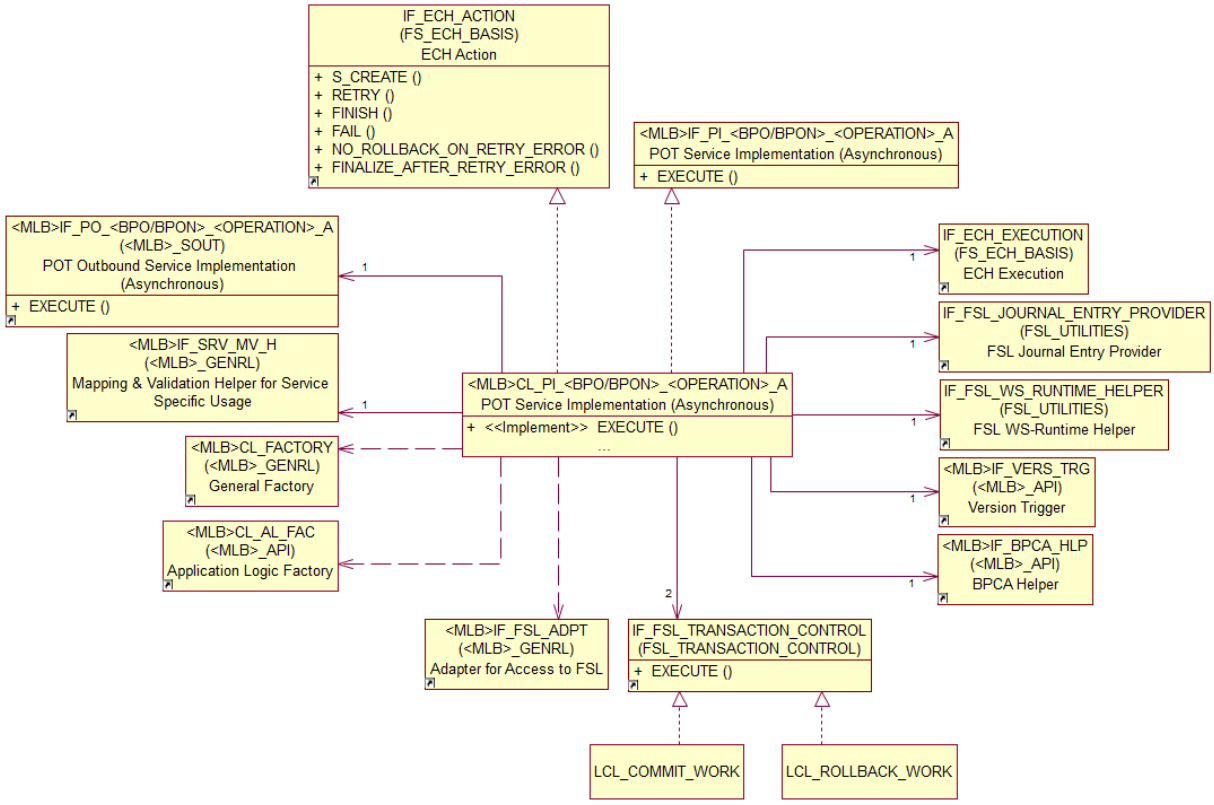


Figure 48: POT Inbound Services (Asynchronous) static view

The service implementation shown here is a changing service that follows the request/confirmation pattern. POB generates one service implementation interface (and the implementing class) for each service operation. The POT Service Implementation is directly instantiated by the SPROXY Class.

The POT Service Implementation covers the following aspects:

- Transaction control, which means that the final transaction end event is triggered by the POT Service Implementation. In contrast to synchronous services, for asynchronous execution the final commit or rollback is triggered by the WS runtime and the service implementation must not call `COMMIT WORK` or `ROLLBACK WORK` at all. Therefore, the implementations for transaction control only call `RESET` or `SAVE` from the Transaction Control Interface. This is supported by using the transaction control interface from FSL, which that is implemented twice: One implementation for `RESET/ROLLBACK` and one implementation for `SAVE/COMMIT`.
- Execution of the POT application logic using the Application Façade and call of the corresponding outbound confirmation message.
- Provision of information to the Process Observer during service execution using the PMA Adapter.
- Setting the version information using the Version Trigger.
- PSJ Integration

  The WS-Runtime Helper and the FSL Journal Entry Provider are used to write PSJ entries related to the processing status of the service execution.
- Error and Conflict Handling for errors which occur during the service processing

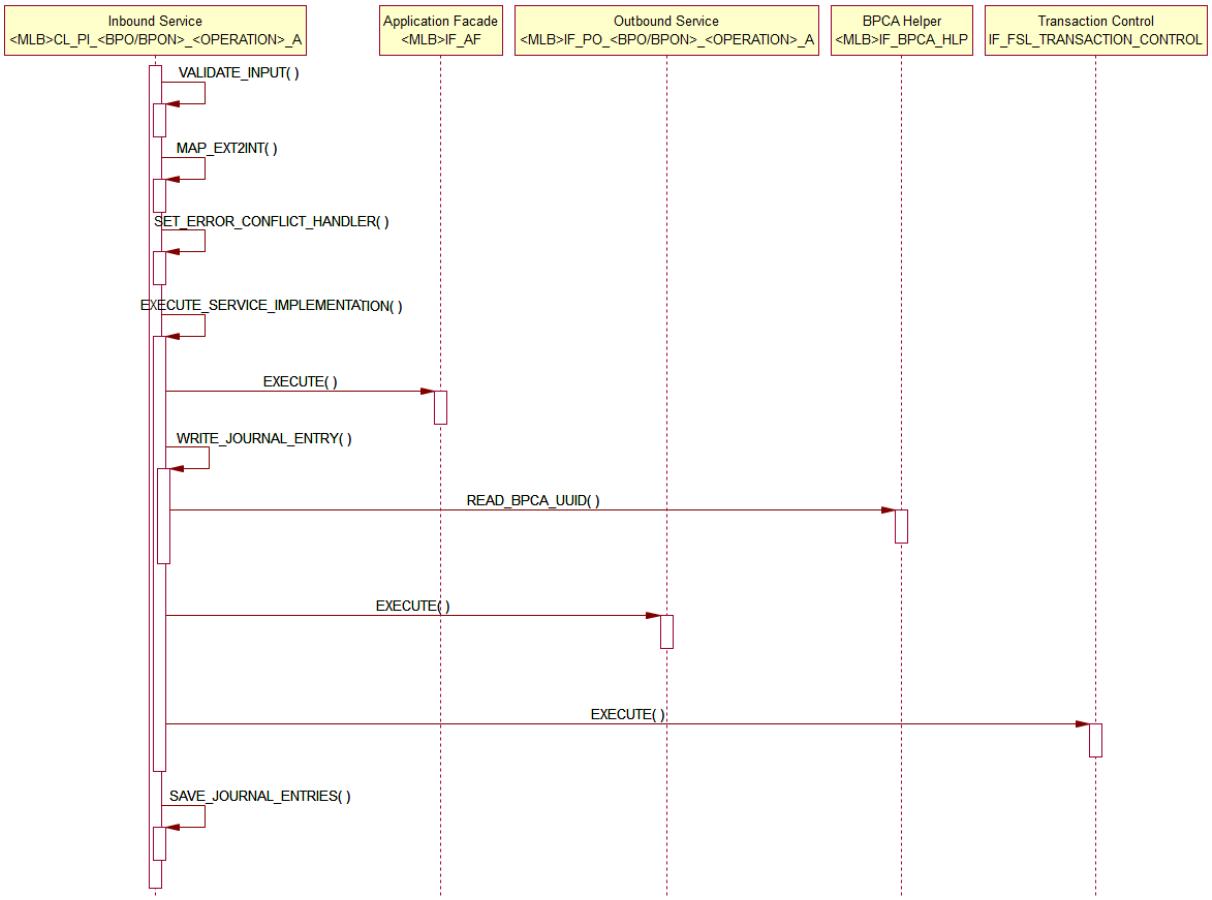


Figure 49: Synchronous POT Standard Inbound Service (Request/Confirmation) – Call Sequence

At first the message validated, mapped to its internal representation and the ECH is initialized. After the application logic has been executed successfully, a PSJ entry is written. In order to write a PSJ entry the BPCA UUID of the BPO is required.

As the BPCA UUID is not part of the payload of the asynchronous check and execute service, the BPCA Helper is used to read the BPCA UUID from the BPO instance. If the BPCA cannot be read, the writing of the PSJ entry is skipped.

The response from the Application Façade is then used to trigger the corresponding outbound confirmation. Finally, the transaction control saves the changes to the BPO.

6.3.1.3 Back-End Inbound Services (Asynchronous)

In addition to the POT standard synchronous inbound service, a POT must also implement the inbound side for the asynchronous messages that are sent the by back ends in order to inform a POT about the processing result of a request message that has been sent asynchronously during the EXECUTE phase. In general, back-end inbound services are assumed to be asynchronous.

Figure 50 shows a static view of the back-end related asynchronous inbound service implementation:

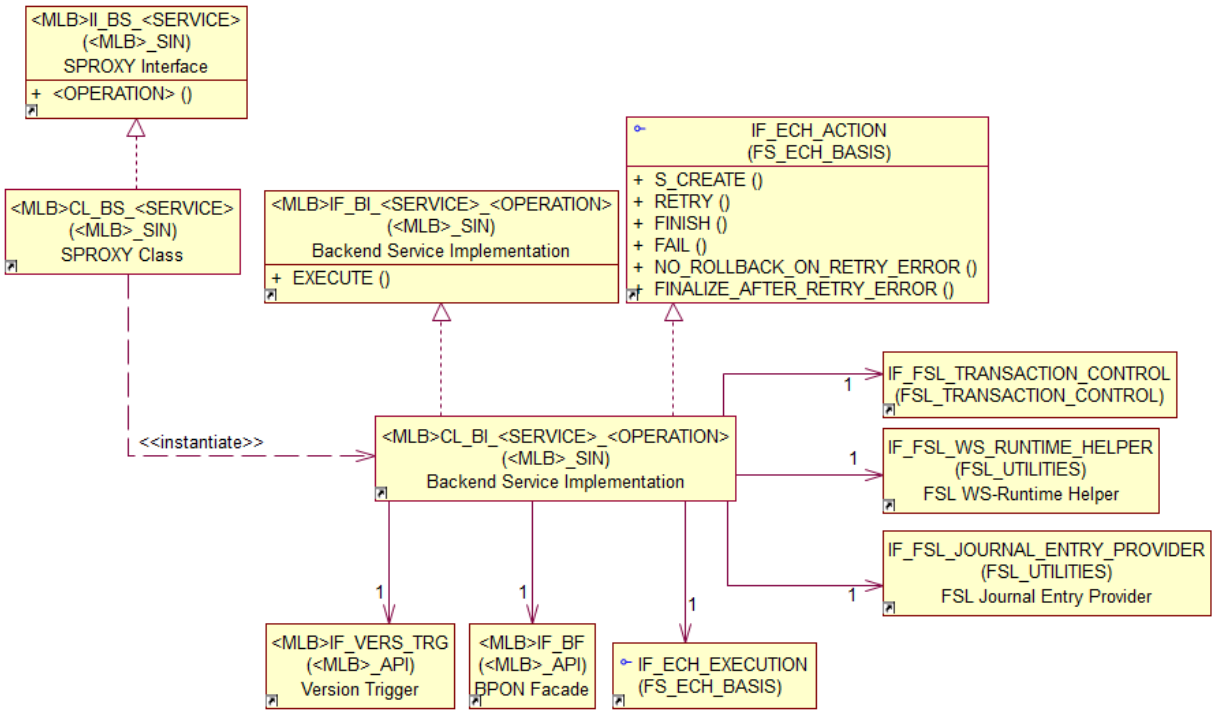


Figure 50: Back-End Related Asynchronous Inbound Service Implementation – Static View

The POT Service Implementation covers the following aspects:

- Transaction Control

  The behavior for transaction control is the same as for asynchronous POT inbound services as described in section 6.3.1.2.

- Error Handling

  For error handling, the service implementation class implements the interface `IF_ECH_ACTION`.

  This is required to enable the service implementation to act as an ECH action class. The reference to the interface `IF_ECH_EXECUTION` is required for handing over erroneous messages to ECH.

- Execution of the application logic by the BPON Facade, which is used across all asynchronous inbound service operations.

- PSJ Integration

  The WS-Runtime Helper and the FSL Journal Entry Provider are used to write PSJ entries related to the processing status of the service execution.

The call sequence for the processing of a back-end related asynchronous inbound service is shown in Figure 51.

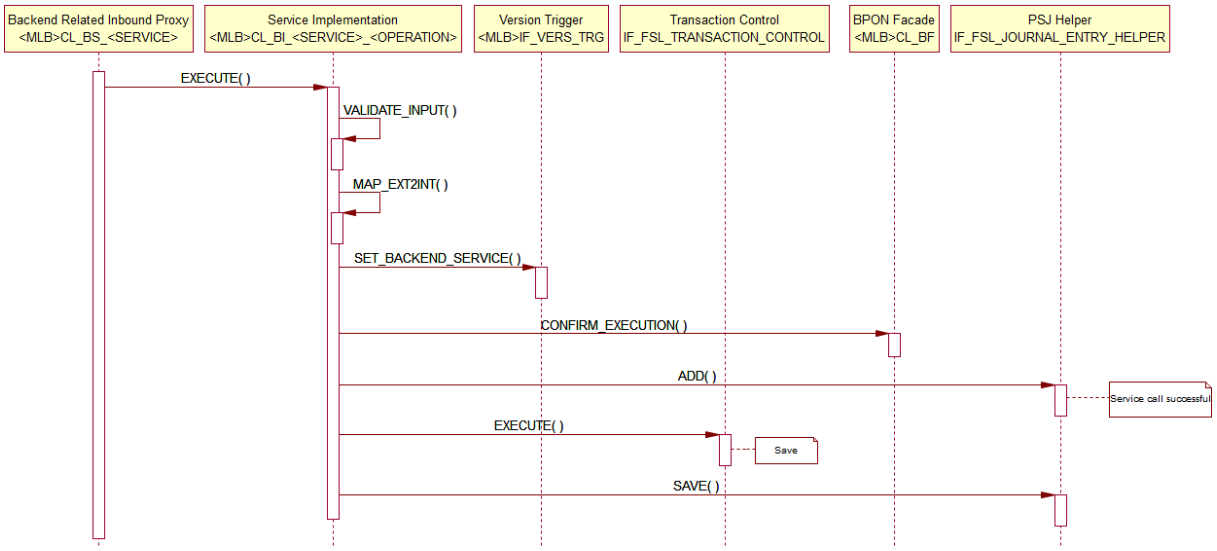


Figure 51: Back-End Related Asynchronous Inbound Service – Call Sequence (Success)

First, the inbound request is validated to make sure that the data is plausible and fulfils the service contract. The inbound request is then mapped from the external to the internal representation, based on which the BPON Façade is called.
Once the BPON Facade has been called to confirm the message, a PSJ entry is written in order to log the success of the service call. The `EXECUTE` method of the Transaction Control Interface is called to save the changes, however, the final `COMMIT WORK` is issued by the WS runtime.

As ECH is used in asynchronous services, error handling is a bit more sophisticated than in the synchronous case.

Figure 52 shows the call sequence for the case that an error is raised by the BPON Façade.

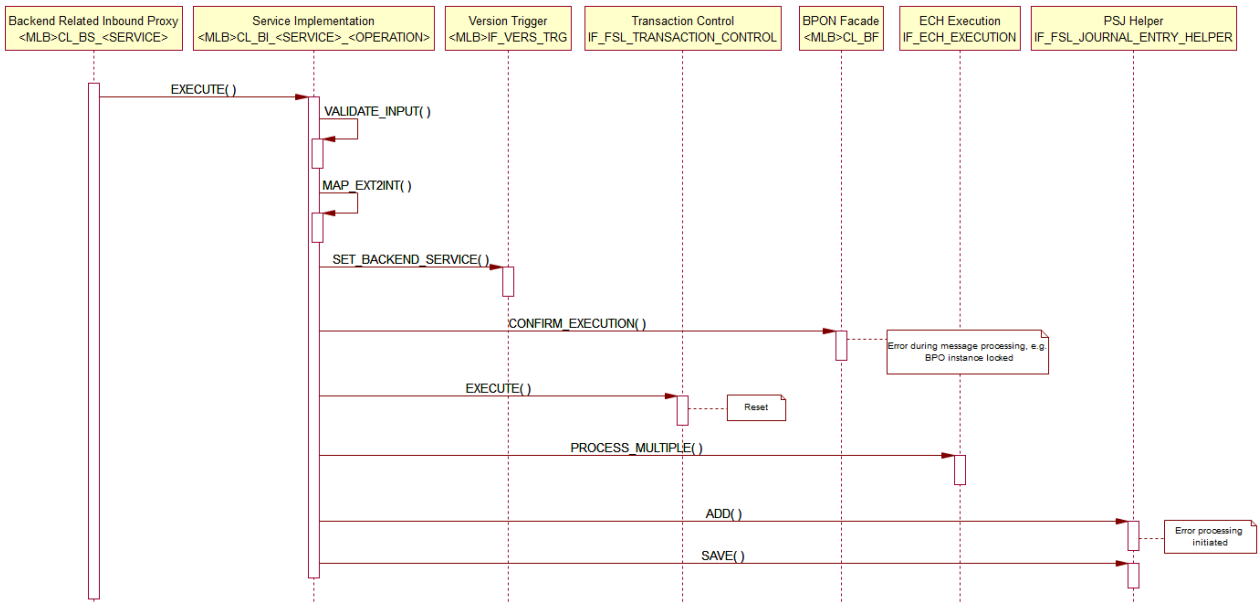


Figure 52: Back-End Related Asynchronous Inbound Service – Call Sequence (Failure)

In case an error occurs while the message is processed within the BPON Facade, the Transaction Control Interface first makes sure that all buffers related to the POT instance are reset.
The erroneous message is then passed to ECH using the `PROCESS_MULTIPLE` method of the ECH Execution

Interface.
After that, an entry is written to PSJ stating that the message has been passed to ECH.

If an erroneous message has been passed to ECH, there are three actions that can be performed on the message (automatically or manually from the post processing desktop). All actions are handled by the service implementation class that implements the interface `IF_ECH_ACTION`. This interface provides the following methods:

- `RETRY`

  The message is processed again. A PSJ entry is written, stating that error handling has been completed.

- `FAIL`

  The message cannot be processed in a meaningful way. If possible, the message is marked as completed and the status of the corresponding BPON is set to *Failed*.

  PSJ entries are written, stating that error handling and service execution have failed.

- `FINISH`

  This confirms the message. PSJ entries are written, stating that the steps for error handling and service execution have been completed.

  The message is marked as completed and the status of the BPON is set to *Confirmed*.

Figure 53 shows an example on how for ECH and the Service Implementation interact for the `RETRY` method:

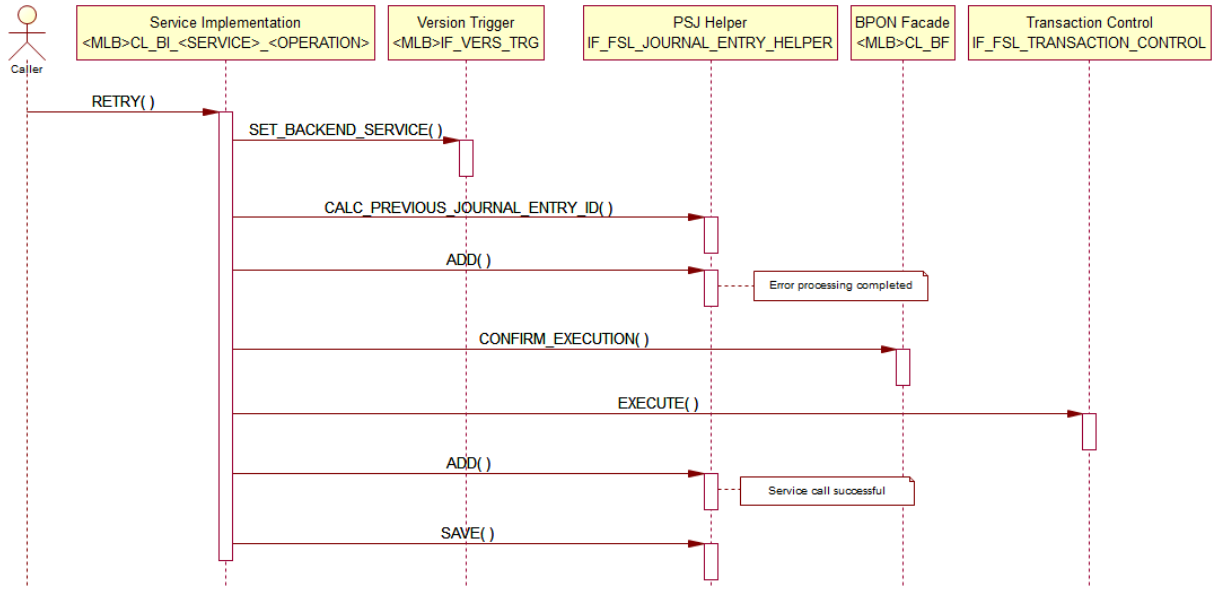


Figure 53: Successful Retry - Call Sequence

If a POT uses asynchronous back-end services that do not comply with the SAP standards (that is, if the back-end service operation signatures do not contain a `BusinessDocumentMessageHeader` or `BasicBusinessDocumentMessageHeader`), the correlation key that maps an asynchronous response to a BPON has to be explicitly specified using the custom correlation class. In case of asynchronous non-standard bulk services, the bulk message has to be split into a table of single messages.

A static view is shown in Figure 55:

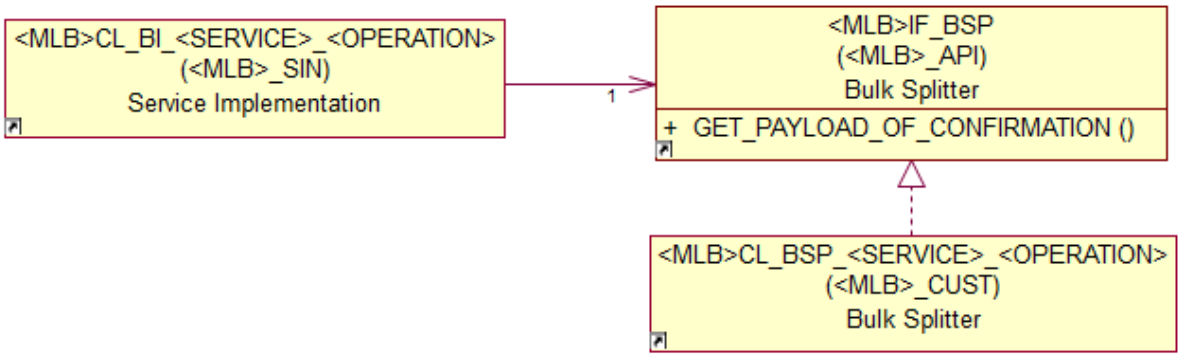


Figure 54: Bulk Splitter – Static View

The Bulk Splitter is called by the service implementation that processes the asynchronous confirmation from the back end. The implementation class includes a code-slot (marked in grey) and is specific for each service operation.

The single messages provided by the bulk splitter are handed over to the custom correlation class to determine the correlation key.

6.3.2 Outbound Communication

As the implementation of POT standard services and back-end-related services are basically the same, the following sections focus on the design of the back-end-related outbound services.

The last section elaborates on ABAP Channels.

6.3.2.1 Back-End Outbound Services (Synchronous)

The static view on a synchronous back-end outbound service is shown in Figure 55. This figure shows the implementation for a back-end-related service for which PMA support (Process Observer Integration) has been activated.

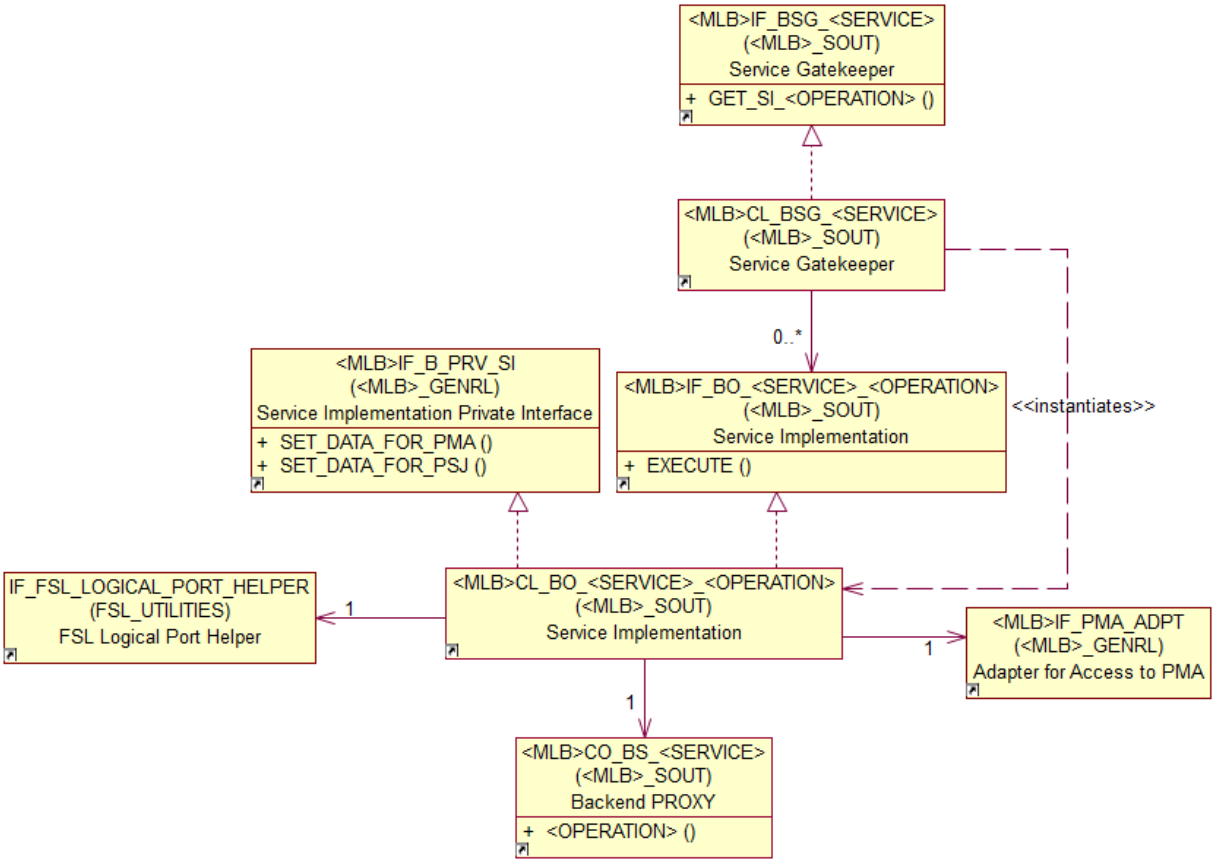


Figure 55: Back-End-Related Synchronous Outbound Service – Static View

In the outbound case, the service gatekeepers are the central point of entry for all service implementations. There is one gatekeeper for each ESR outbound service interface. The gatekeeper holds references to all service implementation classes. As with the inbound case, there is one service implementation interface (and a corresponding implementing class) for each service operation. The class also implements the Service Implementation Private Interface which is required for the integration with the Process Observer. It provides a method `SET_DATA_FOR_PMA` to supply the service implementation class with BPO-related data that is required for the provision of information to the Process Observer during service execution (by using the PMA Adapter). For the instantiation of the Backend Proxy a logical port is required that is obtained from the Logical Port Helper. The logical port is calculated based on the information transmitted in the message header of the request message.

The execution of a synchronous back-end-related outbound service is shown in Figure 56:

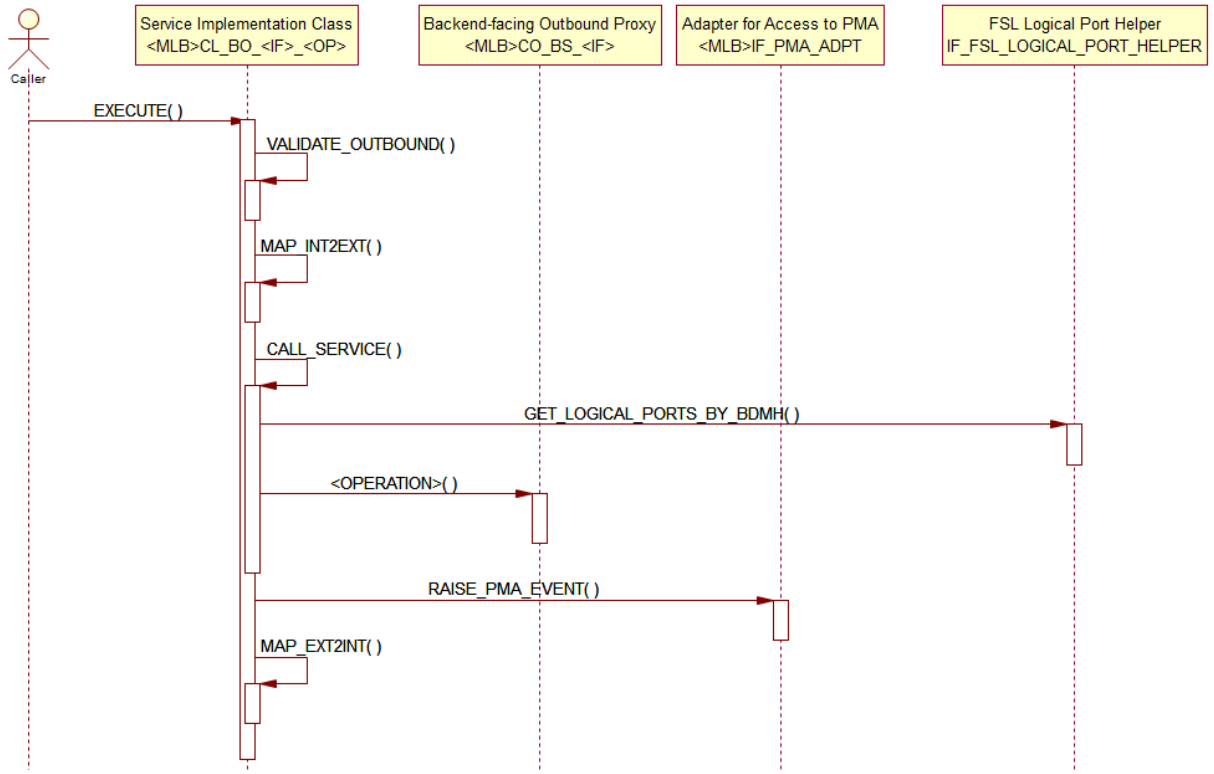


Figure 56: Execution of Back-End-Related Synchronous Outbound Service - Call Sequence

The consumer calls the `EXECUTE` method of the service outbound implementation interface. The implementing class performs a validation and maps the data from internal to external SPROXY representations.
Then, method `GET_LOGICAL_PORT_BY_BDMH` is used to determine the logical port based on the data that is located in the message header by using. This logical port is used to instantiate the Outbound PROXY. After the corresponding operation of the Outbound Proxy has been called, the method `RAISE_PMA_EVENT` provides the execution status of the service to the Process Observer. Finally, the response message is mapped from the SPROXY representation to the internal representation and is then returned to the consumer.

6.3.2.2 Back-end Outbound Services (Asynchronous)

Asynchronous back-end-related outbound services basically follow the same principle as their synchronous siblings.

A static view is shown in Figure 57:

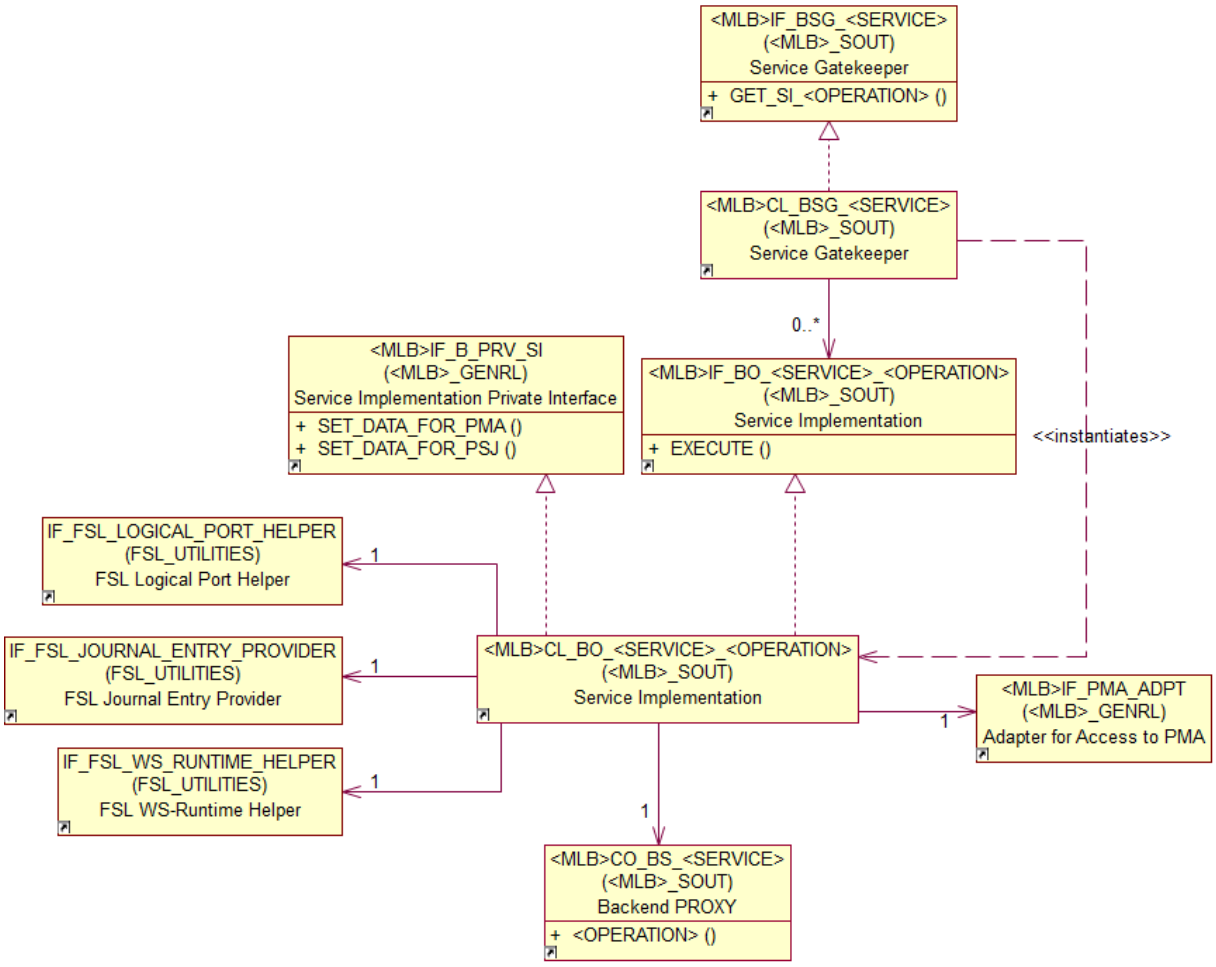


Figure 57: Asynchronous Outbound Service – Static View

Compared to the synchronous service implementation, there are the following differences:

- There is no inbound mapping (`MAP_EXT2INT`) in asynchronous operations.
- More than one logical port is allowed for asynchronous operations.
- For PSJ support, the FSL Journal Entry Provider and the FSL WS-Runtime Helper are required. Additional data which is required for the creation of PSJ entries is provided via method SET_DATA_FOR_PSJ (implemented by the service implementation).

A call sequence is shown in Figure 58:

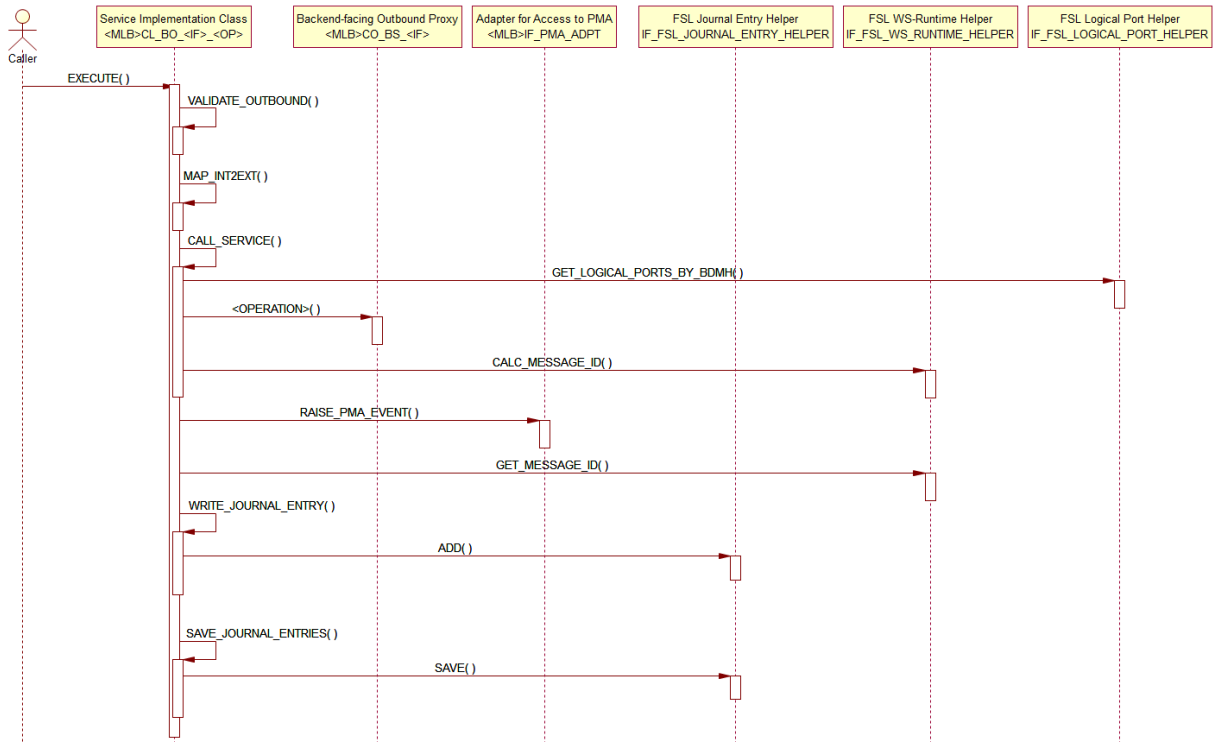


Figure 58: Asynchronous Outbound Service – Call Sequence

The call sequence is basically the same as for synchronous outbound services. Due to PSJ integration, there are some additional calls:
After the operation of the Outbound Proxy has been called, the WS-Runtime Helper is used to calculate the technical message ID. Then, the PMA event is raised (as in the synchronous case).
In the next step, the message ID is obtained from the WS-Runtime Helper as it is part of the PSJ entry that is written by methods `ADD` and `SAVE`.

6.3.2.3 POT Event Channels

Based on WebSocket protocol the following channels inform about changes of a POT:

- /nodeinstancechanged – Informs about changes of the POT instance and its node instances
- /nodeinstancestatuschanged – Informs about status changes of the POT instance and its node instances

In order to register to a push channel the following URL parameters are defined:

- bpo_uuid – UUID of the POT instance (mandatory)
- bindings – Channels to be informed about (optional). If the bindings parameter is not submitted, information about all channels are transmitted.

The events are transmitted in JSON format. An event consists of the event data as well as on metadata.

Events of the channel “/nodeinstancechanged” have the following format:

```
{
        "EVENT":{
                "BPO_UUID":"<uuid in char 36 format>",
                "UUID":"<uuid in char 36 format>",
                “REFERENCE_ID”: “<reference id in GDT format>”,
                “PROVIDER_ID”: “<provider id in GDT format>”,
                "NODE_TYPE":"<scalar value of node type>",
                "STATUS":"<scalar value of status code>",
                "TIMESTAMP":"<timestamp in GDT format>",
        },
        "METADATA":{
                "APPLICATION_ID":"<name of ABAP messaging channel application of POT>",
                "CHANNEL_ID":"/nodeinstancechanged",
                "NODE_TYPE":"<scalar value of node type>",
        }
}
```

Events of the channel “/nodeinstancestatuschanged” have the following format:

```
{
        "EVENT":{
                "BPO_UUID":"<uuid in char 36 format>",
                "UUID":"<uuid in char 36 format>",
                “REFERENCE_ID”: “<reference id in GDT format>”,
                “PROVIDER_ID”: “<provider id in GDT format>”
                "NODE_TYPE":"<scalar value of node type>",
                "FROM_STATUS":"<scalar value of status code>",
                "TO_STATUS":"<scalar value of status code>",
                "TIMESTAMP":"<timestamp in GDT format>"
        },
        "METADATA":{
                "APPLICATION_ID":"<name of ABAP messaging channel application of POT>",
                "CHANNEL_ID":"/nodeinstancestatuschanged ",
                "NODE_TYPE":"<scalar value of node type>",
        }
}
```

For both channels the field “PROVIDER_ID” is only transmitted in case of a node instance, for the POT instance the field is not transmitted.

The transmission of the events is based on the Event Handling of the POT (see chapter 6.5.5) and the transaction control of the POT (see chapter 6.5.1). The events are transmitted only in case of COMMIT WORK.

Figure 59 shows a static view on ABAP channel of a POT:

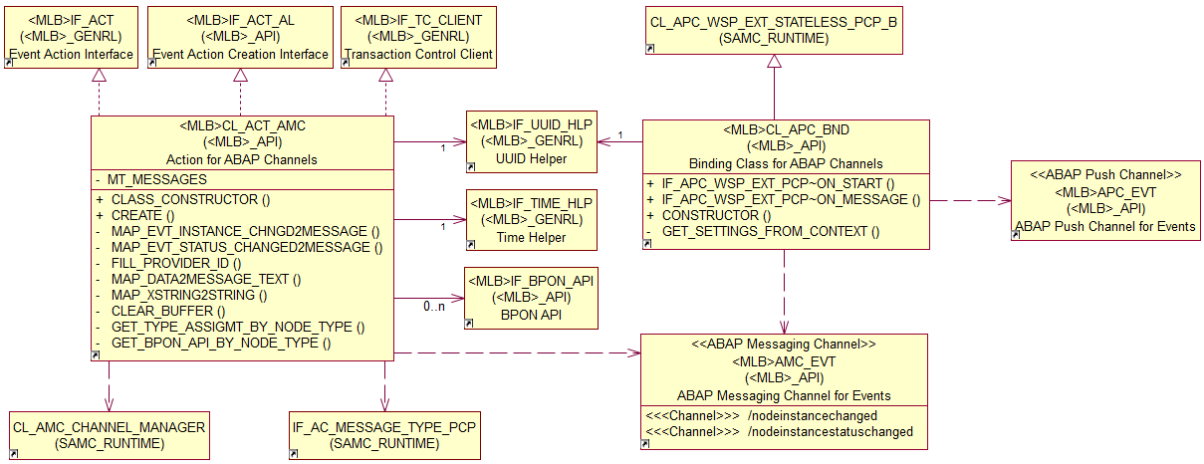


Figure 59: ABAP Channel - static view

The action class <MLB>CL_ACT_AMC collects the changes to be sent out as messages and hands the collected messages over to the ABAP Messaging Channel <MLB>AMC_EVT.

The binding class <MLB>CL_APC_BND creates the connection between the messaging channel and interested parties that have connected to the Push Channel <MLB>APC_EVT.

6.4 UI Layer

The POB Editing UI Wizard generates a user interface based on Floorplan Manager for Web Dynpro ABAP (FPM) out of the box. This generated user interface (usually referred to as the editing UI) is almost ready to use and requires only little extra work by the POT developer for finalization and extension.

Figure 60 shows an example for a main page of a generated Editing UI. The main page consists of building blocks that list all the BPON instances by each node type for a specific PO instance. For each BPON instance, it is possible to navigate to a details page.

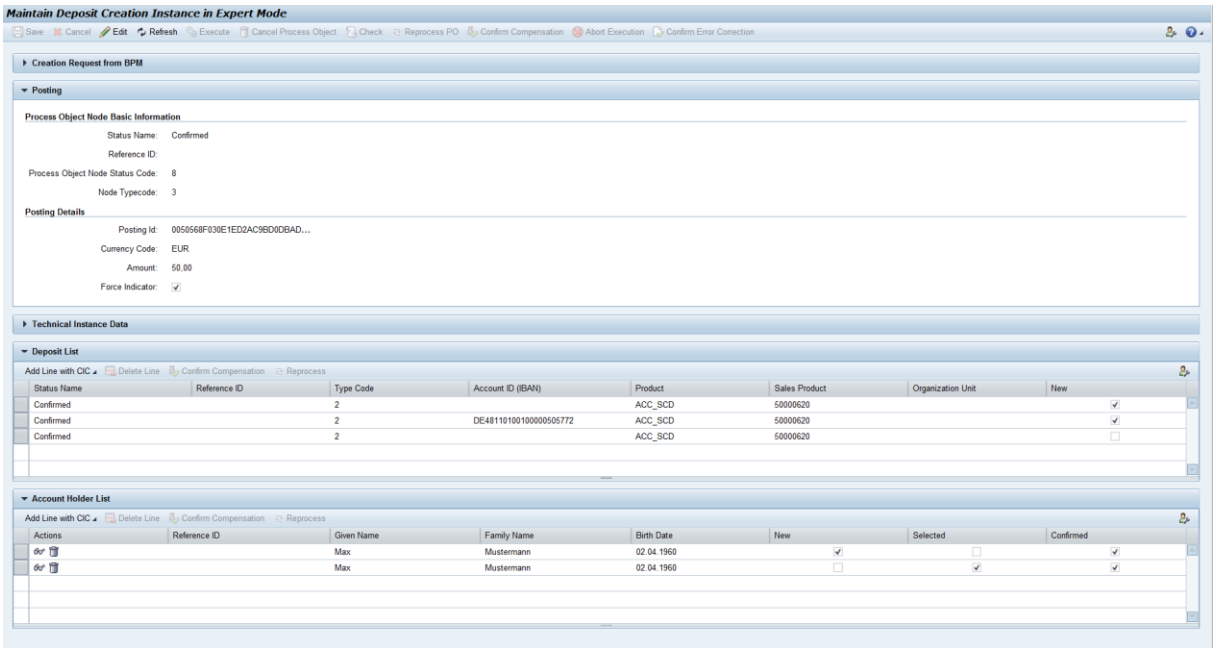


Figure 60: Main Page of a Generated Editing UI

Figure 61 shows the high-level architecture of a generated editing UI:

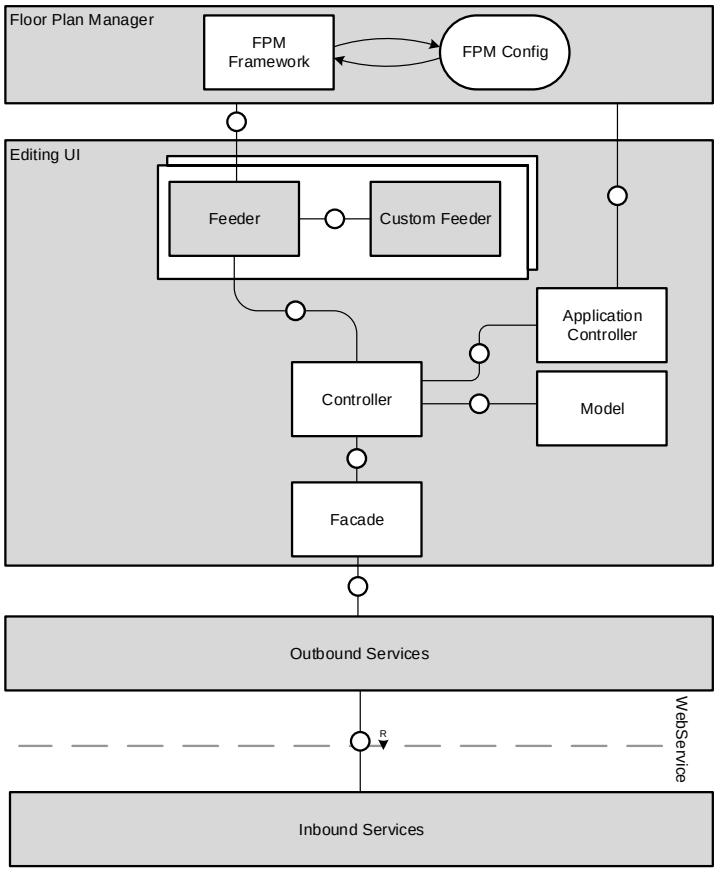


Figure 61: Architecture of the Generated Editing UI

The architecture of the editing UI follows the Model View Controller (MVC) pattern. The view is represented by an FPM Overview Page (OVP), which contains several FPM configurations (for form and list UIBBs) as building

blocks (along with the feeder classes that are associated with these configurations). Feeder classes connect the (generated) editing UI implementation to the FPM framework (i.e. the form and list UIBBs).

The feeder classes provide design-time data like:

- What does the default configuration look like?
- Which actions are available for a screen?
- Which fields are available?
- What are the properties of these fields?

Additionally, feeder classes are responsible to transfer data at runtime and for event handling.

To allow POT developers to extend the generated Editing UI, custom feeder classes are generated that contain dedicated code slots for custom implementations.

Such a custom feeder class is generated for each feeder class and is located in the custom package. The main purpose of the custom feeder class is to do the mapping between the data structure of the model and the feeder structure (field catalog).

The controller is the central entity of the Editing UI that keeps all parts together. It connects the feeder classes to the model and the façade. Therefore, the controller contains methods for each (supported) service operation. These methods call the corresponding methods of the façade class. Additionally, the controller also contains methods to communicate with the model (e.g. `GET_DATA`, `SET_DATA` and `SET_BPO_UUID`). The integration between the FPM Application (`FPM_OVP_COMPONENT`) and the controller is performed by the application controller (which implements the interface `IF_FPM_OVP_CONF_EXIT`). The application controller delegates application level events raised by the FPM framework to the controller.

The main function of the model is to store runtime data that is required for the editing UI. With the `GET_DATA` method the complete model (PO data) is returned to the caller. The `SET_DATA` method is used to overwrite the complete model with the data provided. A partial or delta update is not possible.

The model is also used to store the search result and the current BPON lead selections (selected BPON instances).

The façade class does the mapping from the UI POT structure to the structures of the service calls and executes the service calls. For each POT inbound service, a corresponding outbound interface is available to connect the Editing UI to the POT application logic in a decoupled way.

6.4.1 Screen Structure of the Editing UI

6.4.1.1 Initial Screen

The first screen that appears when the Editing UI is opened contains one block to open a PO instance directly based on a GUID and one block to search for PO instances via administrative data. This screen is generated and ready to use without the need to implement any code slots. The only remaining task for the POT developer is to adopt the screen texts, especially for the result list, where the headings of the generated configuration contain technical names (see Figure 62). UUIDs can be entered in CHAR32 or CHAR36.

A POT developer can extend the search section of the initial screen by adding additional search criteria and by adding additional fields to the result list.

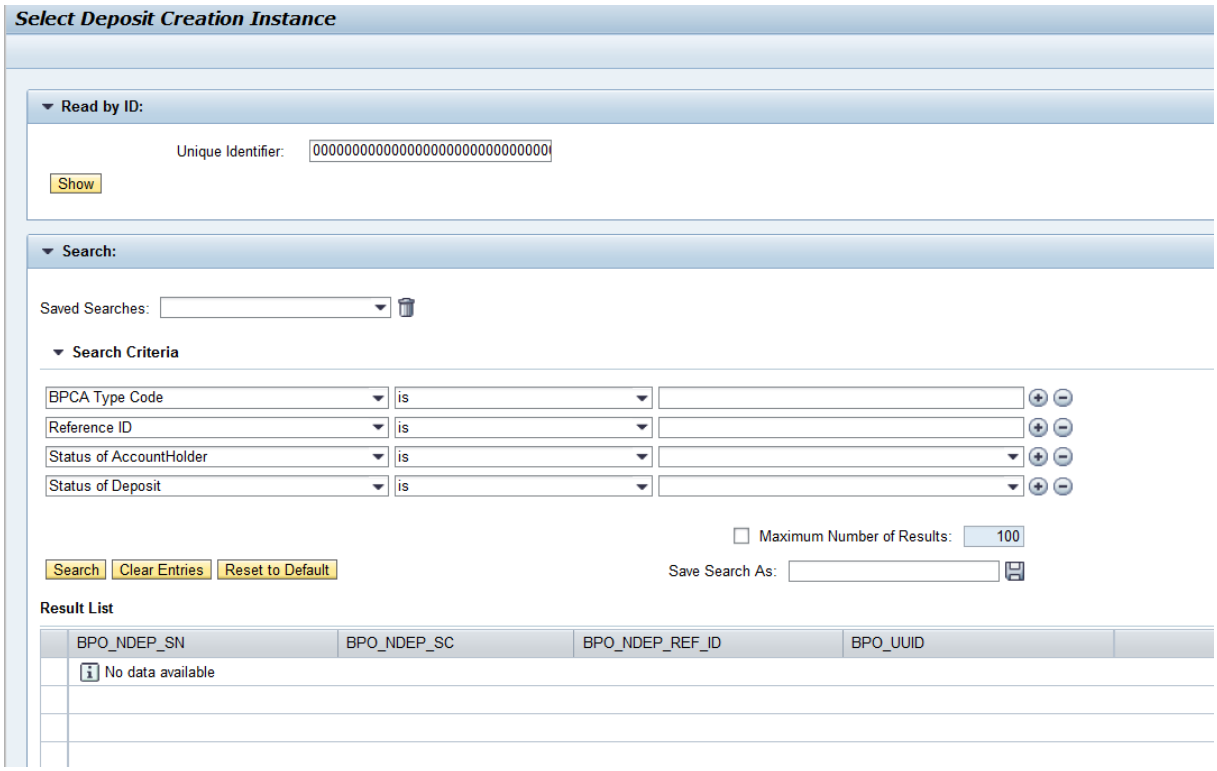


Figure 62: Initial Screen of the Editing UI

The search result is displayed in the same window. The user of the Editing UI can navigate to the respective instance data using a link.

6.4.1.2 Main Pages and Subpages

By default, the editing UI consists of one Web Dynpro application, one application configuration and several component configurations. For each feeder assignment to a model node (which is performed on the first step of the Editing UI Wizard), one component configuration is generated.

The default structure of the editing UI looks as follows:

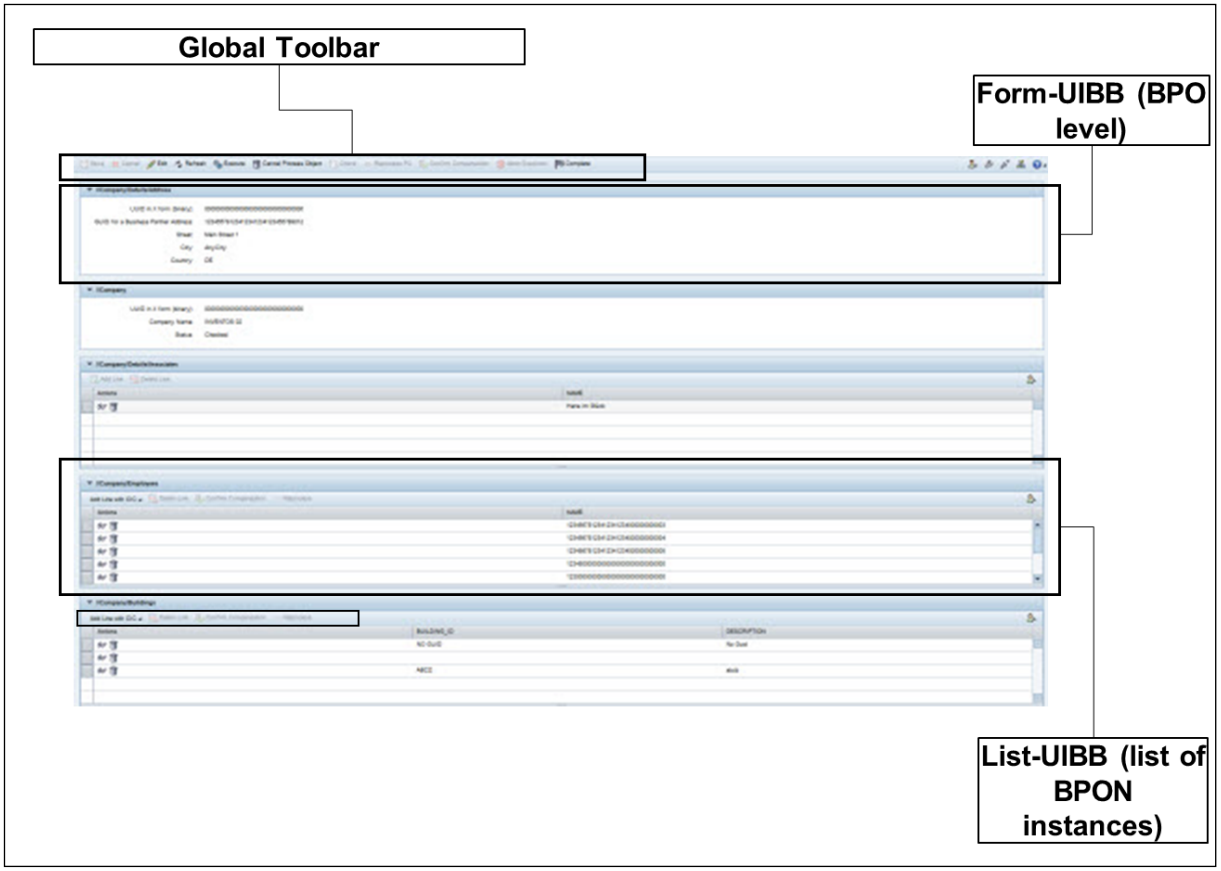


Figure 63: Main Page of an Editing UI

If a POT developer has assigned feeders to details of model nodes on BPON level, subpages are generated. Along with the main and subpages, the required screen navigation from the main page to the subpage and vice versa is also generated. This navigation is performed by wires (the standard FPM concept for controlling navigation and transferring information between UIBBs).

In fact, if required, a deeply structured screen hierarchy can be created out-of-the-box that consists of a main page, subpages, subpages of subpages and so on).

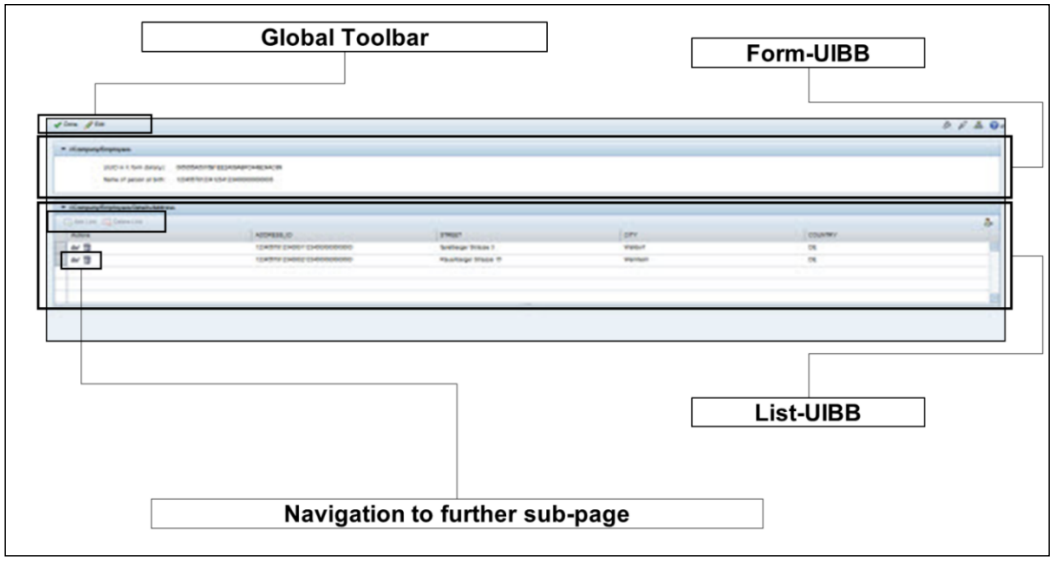


Figure 64: Subpage of an Editing UI

6.4.2 Actions on the Editing UI

Actions are triggered by buttons on the UI. Application level actions (which usually lead to service calls to the POT) are triggered from the global toolbar. Actions that only affect specific nodes or even smaller parts of the model are triggered on GUIBB level.

Service operations for the process object instance can only be executed on the main page. Depending on the current status of the instance, buttons for different actions are available on the main page. The status of an instance also determines whether an instance can in fact be edited (switch to edit mode).

Table 6-2 lists the default actions along with the corresponding buttons from the global application toolbar. It is also possible to define custom buttons (see sections 6.4.4.4 and 8.2.).

Button	Availability of Function	Features
Display / Edit	Always	Switch between Display and Edit mode (Change to edit mode depending on status of instance)
Save	Edit mode	Save data and switch to display mode. Execute `Update` service operation
Cancel	Edit mode	Terminate processing and discard changes that have not been saved Switch to display mode
Refresh	Always	Read and display current instance data Execute `Read` service operation
Execute	Display mode Depending on status of PO instance	Execute `Execute` service operation
Cancel Process Object	Display mode Depending on status of PO instance	Execute `Cancel` service operation
Check	Display mode Depending on status of PO instance	Execute `Check` service operation
Reprocess PO	Display mode Depending on status of PO instance	Execute `Reprocess` service operation
Confirm Compensation	Display mode Depending on status of PO instance	Execute `Confirm` Compensation service operation
Abort Execution	Display mode Depending on status of PO instance	Execute `AbortExecution` service operation
Confirm Error Correction	Display mode Depending on status of PO instance Only when called using CHIPs	Execute `ConfirmErrorCorrection` service operation in order to clear the provider log
Complete	Display mode Depending on status of PO instance	Trigger CHIP event `PL1_COMPLETE`

Table 6-2: Pushbuttons on the Global Application Toolbar

Table 6-3 lists the actions that are available on GUIBB level along with the corresponding buttons. It is also possible to define custom actions and buttons GUIBB level (see sections 6.5.5.4 and 8.2) or to remove standard buttons that are not needed on this UI.

Button	Availability of Function	Features
Add Line	Edit mode Lists (except for process object nodes)	Add line to lost
Add Line with CIC	Edit mode Lists (only for process object nodes)	Add line with a specific Creation Instruction Code
Delete Line	Edit mode Lists	Delete line from list
Display	Always Lists	Call subpage for selected line
Delete	Always Lists	Delete line from list (Implicit switch to edit mode)
Confirm Compensation	Display mode Depending on status of process object node	Execute `Confirm Compensation<Node>` service operation for selected PO node
Reprocess	Display mode Depending on status of process object node	Execute `Reprocess <Node>` service operation for selected PO node

Table 6-3: Pushbuttons on the GUIBBs

6.4.3 Technical Structure of the Editing UI

The application configuration(s) are generated by POB depending on the wizard settings. Each application configuration contains one or several component configurations. Application configurations are linked to exactly one application controller.

The technical structure of the Editing UI is divided into two parts: The initial screen provides functions to search for POT instances and consists of special feeder classes that are the same for each configuration. The main and subpages consist of several component configurations (depending on the settings in the Editing UI Wizard). Each GUIBBs is linked to exactly one feeder (however, on feeder can be reused in several GUIBBs).

6.4.3.1 Feeders for Search Function

The search function is provided by two feeders (as shown in Figure 65). The feeder for search handles the search criteria and the event for starting the search. The feeder for search result (`<MLB>CL_FS_SBA_R`) displays the result of the search on the screen. Both feeders interact with the controller to start the search and

to get/set the search result. Custom implementations that allow adding additional search criteria and result fields are placed in the feeder for custom search (marked in grey).

The integration into the FPM framework is defined by the interfaces `IF_FPM_GUIBB_LIST`, `IF_FPM_GUIBB_SEARCH` and `IF_FPM_GUIBB`.

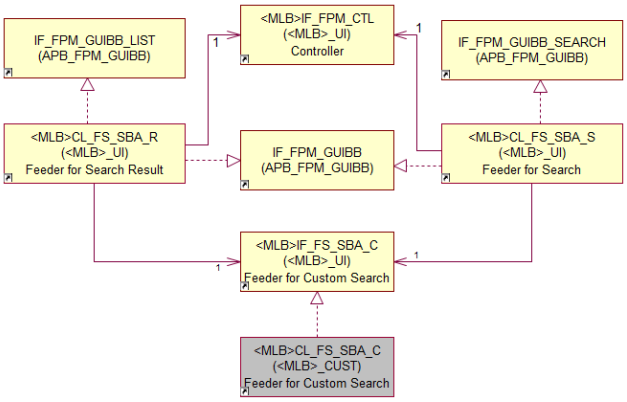


Figure 65: Feeders for Search – Static View

6.4.3.2 Application Controller

The application controller manages the application as a whole. The editing UI uses the application controller as an integration point to the FPM application in order to handle application-level events.

Figure 66 shows how this integration is implemented and where custom implementations can be performed:

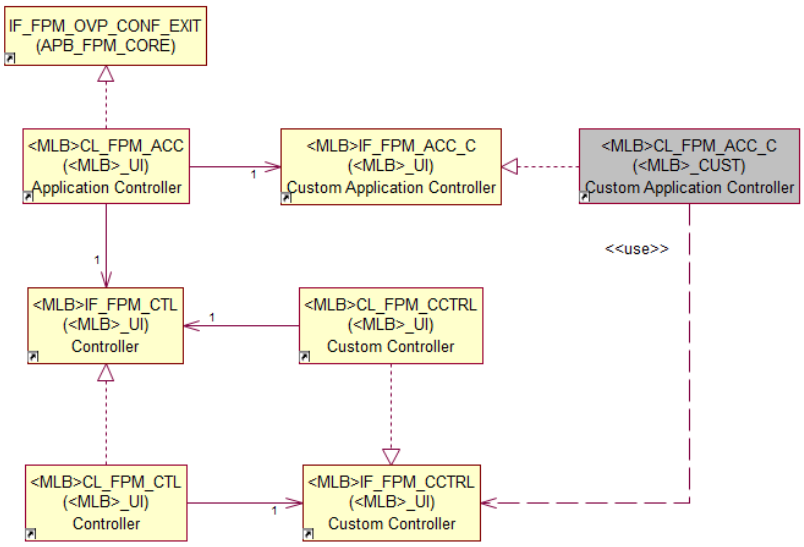


Figure 66: Application Controller – Static View

The editing UI is always generated as an FPM Overview Page (OVP). The integration of the editing UI with the OVP is ensured by the application controller that implements the interface `IF_FPM_OVP_CONF_EXIT`.

The application controller interprets application level events and delegates the corresponding actions to the controller via interface `<MLB>IF_FPM_CTL`.

Customer implementations are located in the custom application controller (shown in grey). The custom application controller interacts with the controller via the custom controller (which provides a subset of the controller functionality). In the custom application controller, it is possible to trigger service operations by using the custom controller (e.g. as a reaction to a customer-defined button that has been pressed by the user of the editing UI).

6.4.3.3 Form and List Feeders

In FPM, feeders are responsible for linking the application to a generic user interface building block (GUIBB) such as a form or a list GUIBB. The Editing UI only supports form and list feeders. The feeder classes `<MLB>CL_FL<FEEDER_ABBR>` (for list feeders) and `<MLB>CL_FF_<FEEDER_ABBR>` (for form feeders) are generated by POB.

Figure 67 shows how these generated form and list feeders are integrated into the FPM framework and how they interact with the rest of the generated Editing UI. It also shows the classes where the code slots for custom implementations are located (marked in grey).

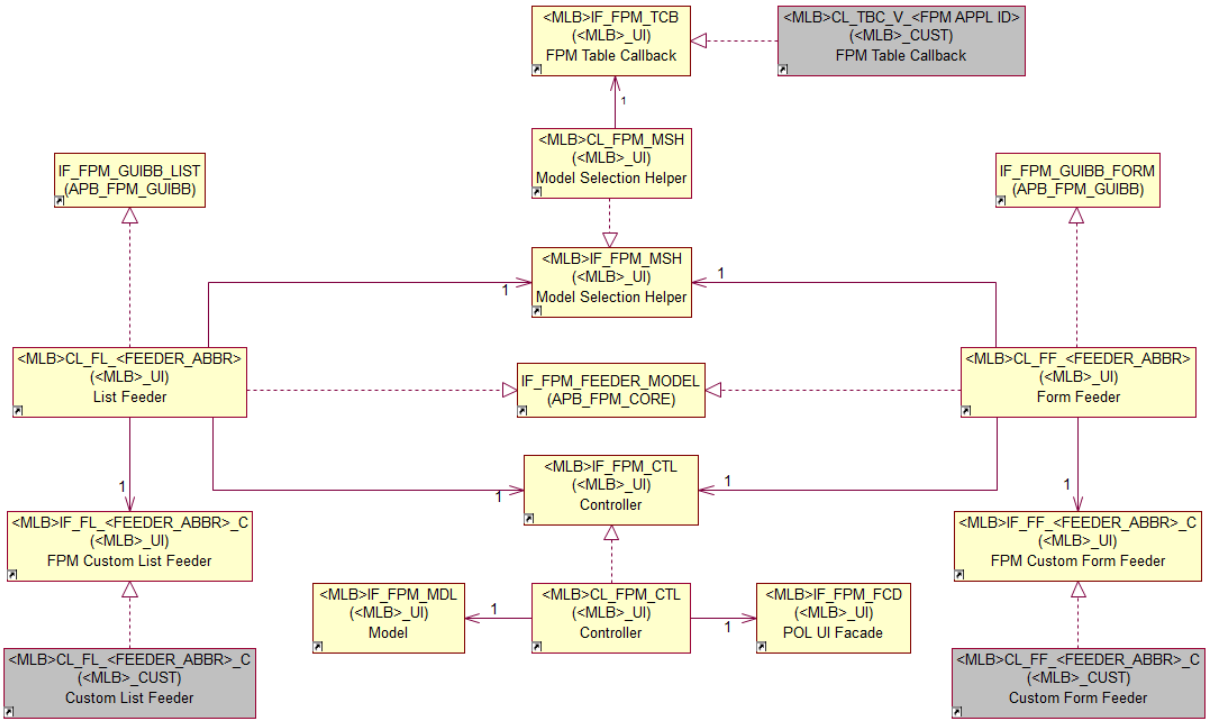


Figure 67: Form and List Feeders – Static View

The integration into the FPM framework is performed via the interfaces `IF_FPM_FEEDER_MODEL`, `IF_FPM_GUIBB_LIST` and `IF_FPM_GUIBB_FORM`. Feeder classes are instantiated and initialized by the FPM framework according to the settings in the component configuration of the GUIBB(s) that the feeders are assigned to. It is possible to assign a feeder class to more than one GUIBB.

Feeder classes interact with the rest of the application using the controller. The controller enables access to the application state (i.e. application mode), the model (containing the data) and the façade (for triggering service operations that communicate with the POT).

Custom implementations for feeders are performed in the custom classes for form and list feeders. From the custom feeder classes, it is possible to access data from the model. It is not possible, however, to trigger service operations.

In order to connect the feeder to the correct data in the model, the model selection helper is used. In the special case that no list feeder is assigned to a tabular node in the model (and no lead selection can be set by the user of the editing UI), the FPM table callback classes can be used to set the lead selection automatically and to modify the data of this node if needed. They are implemented in class `<MLB>CL_TCB_V_DFT`.

6.4.4 Technical Behavior of the Editing UI

The technical behavior of the editing UI has design-time and runtime aspects. The design-time aspects comprise the dynamic definition of feeder properties such as field catalog, field characteristics, and possible actions. Runtime aspects include the search function on the initial screen, displaying and saving data, the handling of events as well as the communication with the POT.

6.4.4.1 Search

Figure 68 shows the call sequence of the search operation:

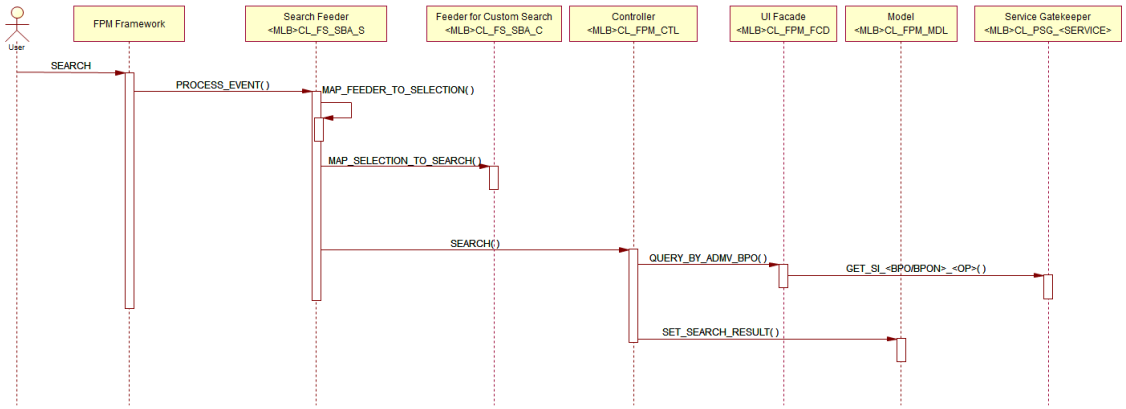


Figure 68: Search – Call Sequence

The search feeder first maps the standard selection parameters to the search query. In order to perform the mapping for additional customer-defined search fields, the `MAP_SELECTION_TO_SEARCH` method of the custom controller is called (see section 8.1.1). The controller triggers the consumer proxy (which is retrieved from the Service Gatekeeper for the `<ProcessComponent>Query<BusinessProcessObject><Version>In` service interface) for the `Find<BusinessProcessObject>byAdministrativeData` service operation by calling the Facade. The search result that is returned by the consumer proxy is buffered in the model for later retrieval by the controller. This information is only available during the application lifetime.

Figure 69 shows how the buffered search result is retrieved from the model and prepared for display in the corresponding list GUIBB:

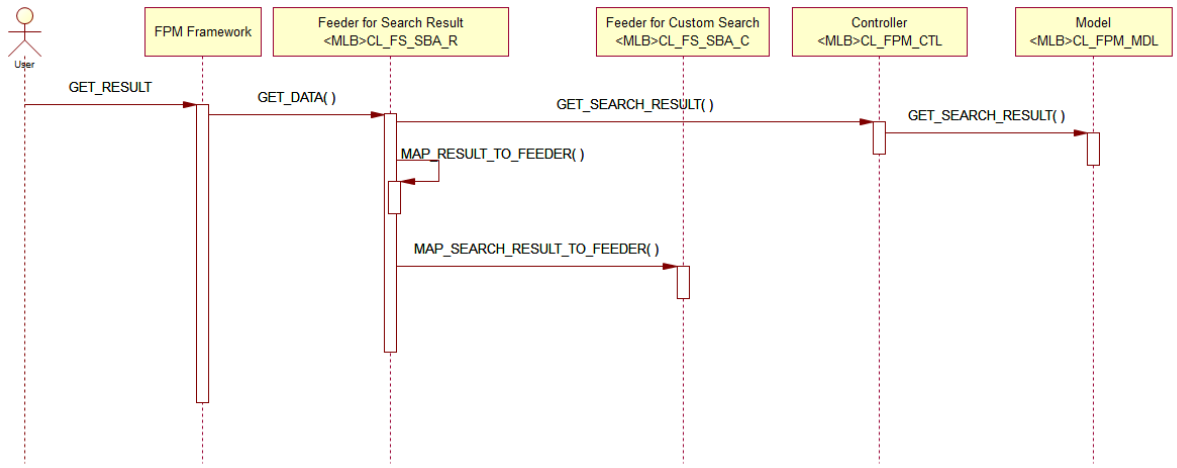


Figure 69: Search Result Retrieval – Call Sequence

The feeder requests the search result from the controller. It then performs the field mapping for the default fields. The mapping of the custom fields is then performed in the corresponding code slot of the `MAP_SEARCH_RESULT_TO_FEEDER` method of the feeder for custom search (see section 8.1.2).

6.4.4.2 Feeder Definitions

Figure 70 shows how the FPM framework retrieves definitions for fields and actions from a feeder (in this case a list feeder is used as an example; the call sequence for a form feeder is the same):

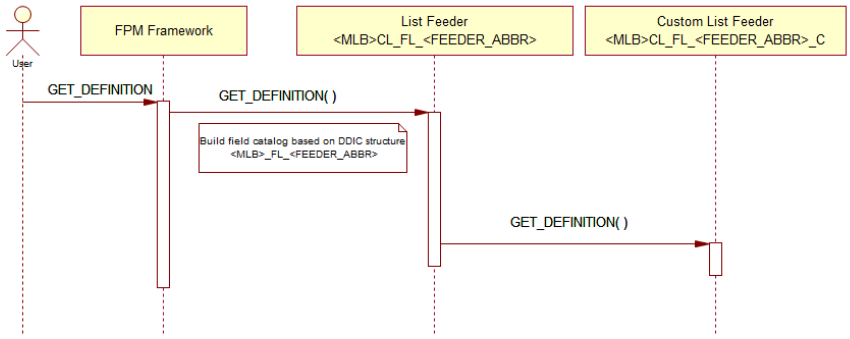


Figure 70: Custom Feeder Definitions – Call Sequence

The FPM Framework calls the `GET_DEFINITION` method of the feeder at design time (creating/changing the component configuration of the GUIBB) and the first time the feeder is called during the bootstrapping of the FPM application. The feeder builds the field catalog based on the DDIC structure created for the feeder (`<MLB>_FL_<FEEDER_ABBR>`).

Field characteristics and possible actions can then be enhanced or changed in the code slot of the GET_DEFINITION method of the custom feeder (see sections 8.2.3 and 8.2.6).

6.4.4.3 Displaying and Saving Data

To display data on the UI, the current data is retrieved from the model. From an FPM perspective, this happens in the feeder method GET_DATA:

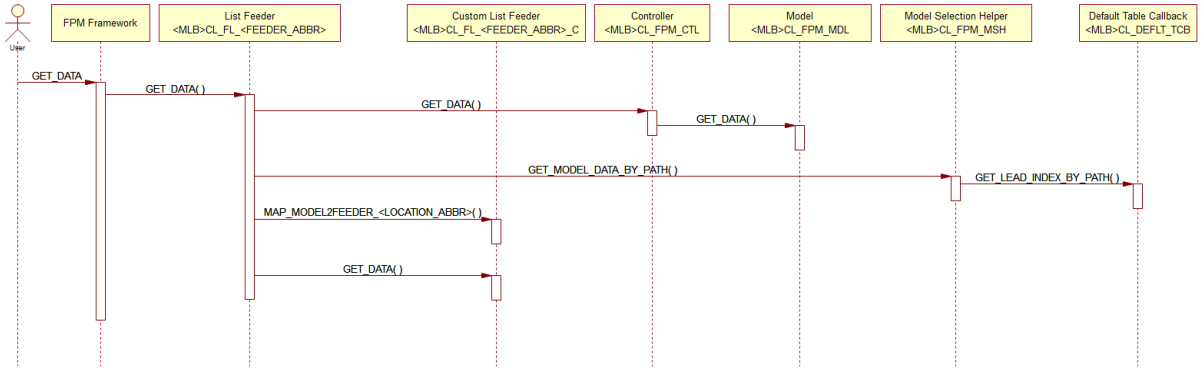


Figure 71: Display Data on the Editing UI – Call Sequence

Because the model always returns the complete instance data, it must be determined, which of this data is relevant for the current feeder. This is done using the GET_MODEL_DATA_BY_PATH method. For tabular model nodes to which a list feeder is assigned, the marked table line of this feeder (lead selection) is used by default.

In the special case that no list feeder is assigned (and therefore no lead selection from the screen is available), the GET_LEAD_INDEX_BY_PATH method is called. This method contains a code slot that needs to be implemented by the POT developer (see section 8.2.8).

From the GET_DATA method of the feeder, the methods following methods are called. These methods contain code slots where the POT developer can place custom implementations (see sections 8.2.5 and 8.2.6):

- MAP_MODEL2FEEDER<LOCATION_ABBR>

  For this method, the code slot implementation is mandatory.

  Note that <LOCATION_ABBR> is unique for each assignment of a feeder to a model node; a feeder can be assigned more than once.

  GET_DATA of the custom feeder class

  The implementation of this code slot is optional.

When the user performs an action on the screen or leaves the screen, the FPM framework executes the FLUSH method in order to transfer data that has been changed on the screen to the model. Figure 73 shows this procedure as an example for a list feeder:

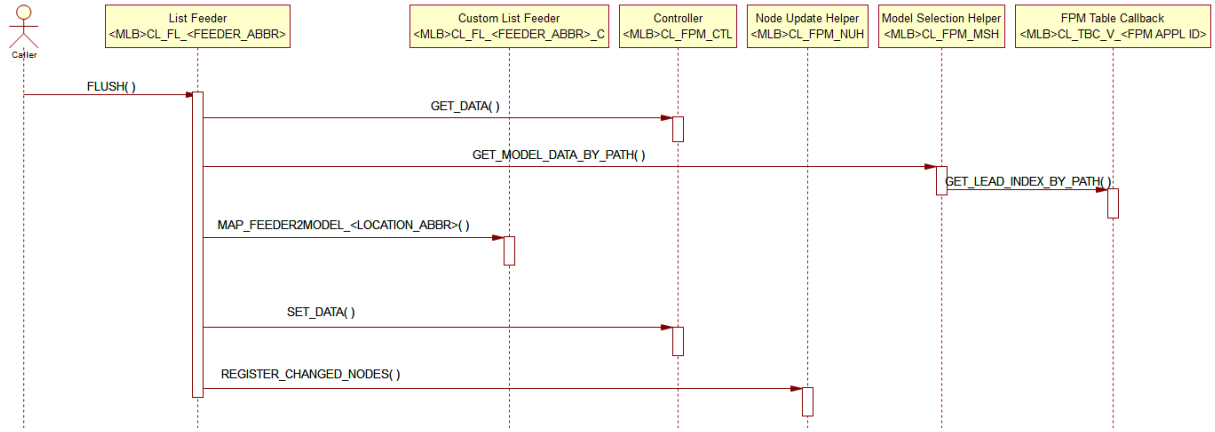


Figure 72: Flush After End of Roundtrip – Call Sequence

As you can see from this figure, the first steps in the feeder implementation (methods `GET_DATA` and `GET_MODEL_DATA_BY_PATH`) are identical to the procedure for displaying the data. The method `MAP_FEEDER2MODEL_<LOCATION_ABBR>` updates the corresponding model node with the data entered/changed on the GUIBB. For this method, the code slot implementation is mandatory for data that can be edited on the GUIBB, see section 8.2.5). The `SET_DATA` method triggers an update of the model via the controller.
The method `REGISTER_CHANGED_NODES` informs the model update helper about changes in the model (new nodes, changed nodes, deleted nodes) so that the node update helper can derive the correct ActionCode (important for saving, see below).

To save the data (see Figure 73), the user of the editing UI user presses the *SAVE* button on the editing UI. When this button is used, the application controller delegates the corresponding event to the controller. The controller calls the facade class in order to call the `Update` service of the POT. The ActionCode for the instance is set based on the changes that have been performed for the instance (created, change, deleted or nothing).
The result of the `Update` service call is mapped back to the model.
Finally, several initializations are done. The "dirty" flag of the model is reset, all lead selections are cleared and all the helpers are reset. As a result, the editing UI reflects the correct state of the BPO instance.

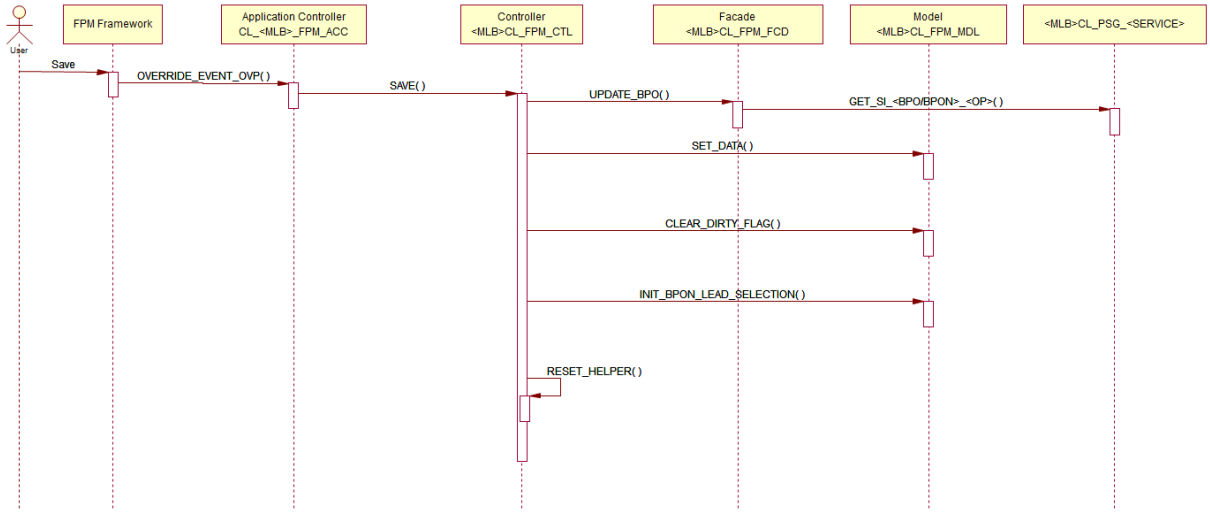


Figure 73: Saving Data from the Editing UI – Call Sequence

6.4.4.4 Handling Events and Button Status

Event handling takes place on application level (e.g. for events from the global application toolbar) or on GUIBB level. Application-level events are handled by the application controller. Events on GUIBB level are handled by the corresponding feeder.

Application-level events are raised, for example, if a user presses the *SAVE* button or a custom button from the global application toolbar.

Figure 74 shows an example of how application events are handled:

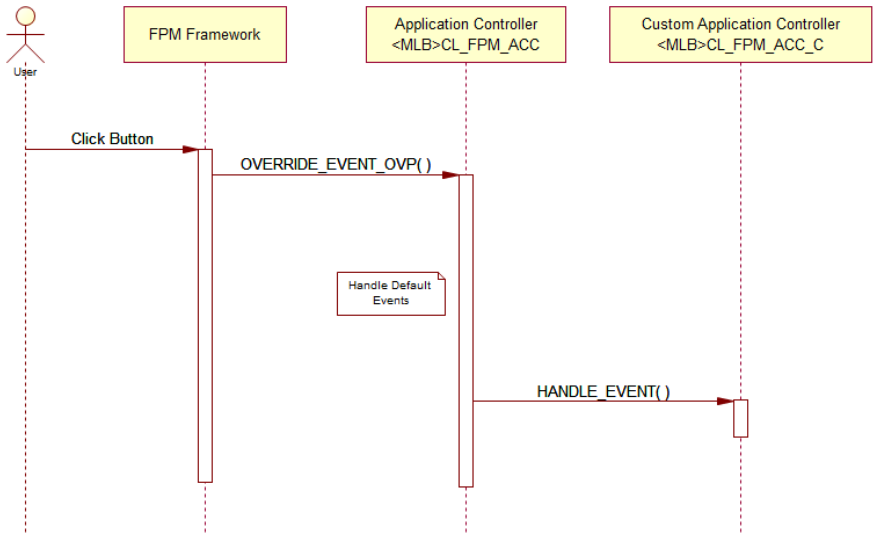


Figure 74: Handling Application Level Events – Call Sequence

If the user raises an event (e.g. by pressing a button), the FPM Framework calls the `OVERRIDE_EVENT_OVP` method of the application controller.

In this method, the application controller first performs the default event handling (not completely shown in Figure 74) for standard events like `FPM_SAVE` (FPM standard event for saving) and `PL1_EXECUTE_BPO` (POT-specific standard event for performing the `Execute` service operation from the editing UI). A specific example for this is shown in Figure 73, where the application controller handles the save event.

If the event is a custom event (starting with `CPL1_`, e.g. for a customer-defined button), the application controller calls the `HANDLE_EVENT` method of the custom application controller. This method contains a code slot where custom event handling on application level can be performed.

The activation status of buttons on application level is also performed in method `OVERRIDE_EVENT_OVP` in the application controller. Figure 75 shows the corresponding call sequence:

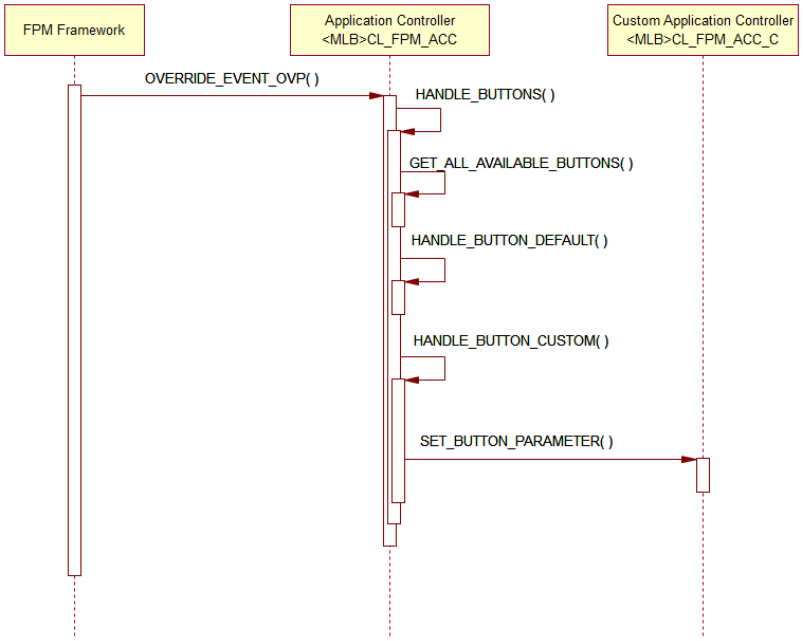


Figure 75: Handling Button Status on Application Level – Call Sequence

The `HANDLE_BUTTONS` method (which is called after event handling is completely finished) first retrieves all available buttons (`GET_AVAILABLE_BUTTONS`) and processes each button in a loop. Depending on the type of event that is assigned to the button (`FPM_<EVENT_ABBR>` or `PL1_<EVENT_ABBR>` for default events or `CPL1_<EVENT_ABBR>` for standard events), either the method `HANDLE_BUTTON_DEFAULT` or the method `HANDLE_BUTTON_CUSTOM` is called. The method `HANDLE_BUTTON_CUSTOM` prepares the button handling and delegates the call to the `SET_BUTTON_PARAMETER` method of the custom application controller. This method contains a code slot for custom button handling.

For feeders, the FPM framework calls the method `PROCESS_EVENT` of the corresponding feeder. Figure 76 shows an example for a list feeder:

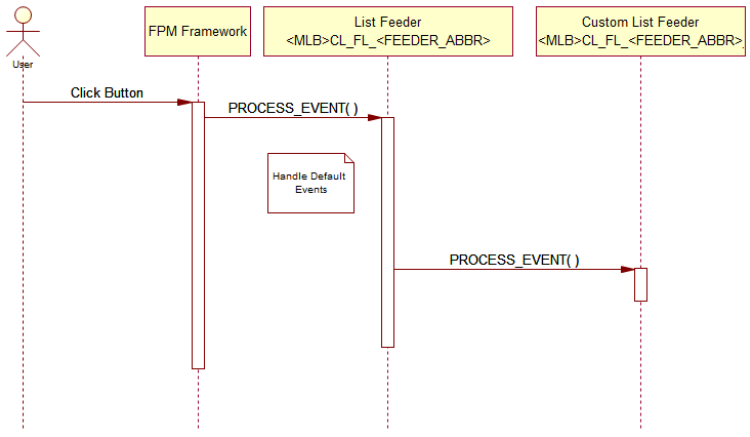


Figure 76: Handling Feeder Events – Call Sequence

First, a standard event handling procedure is performed for default events. For the default event handling on feeder level, only events that have been raised on the GUIBB to which the feeder is assigned will be taken into account. Examples for events on GUIBB level are:

- Navigation events (e.g. from a list UIBB to a form UIBB to display additional details and vice versa)

- Events for adding and deleting lines (for list feeders)

The handling of custom events is delegated to `PROCESS_EVENT` method of the custom feeder, which contains a corresponding code slot (see sections 8.2.3 and 8.2.6).

The handling of the button activation status can be performed in the code slot of the `GET_DATA` method of the custom feeder. (This method has already been mentioned in the context of data display, see Figure 71.) The definitions for the actions that are required for custom buttons need to be implemented in the `GET_DEFINITION` method of the custom feeder (see Figure 70).

6.4.5 Data Buffering in the Model

The editing UI buffers the POT instance data in the model. The controller coordinates the interaction between the feeders, the buffer and the POT (see section 6.4.4.3 for an explanation of this interaction for the `Save` operation). For the model, an internal data type is required that contains the complete exposed public data of the process object.

The PO type `<MLB>IF_IT_COMP=>TS_BPO_PUBLIC_TOTAL`, which contains the public instance data (that is generated together with the POT) is not suitable for use in the editing UI:
Obviously, it should be possible to define list components on table-like data types that are located in a Details node of a BPON. Usually, these list components are wired to corresponding form components, which are used to display details of a selected line. To be able to identify a selected line and propagate the lead selection to dependent components, a unique identifier is needed on line-type level. However, the `<MLB>IF_IT_COMP=>TS_BPO_PUBLIC_TOTAL` type does not contain such a field. Consequently, a new type (`<MLB>IF_FPM_TAC=>TS_BPO_PUBLIC_TOTAL_UI`) for UI-specific purposes needs to be generated. This type contains a GUID within each line type where a list component is located. This unique identification is also important when changes on a list GUIBB need to be mapped back to the model by a list feeder.

The buffer is refreshed in the following situations:

- An instance is selected on the initial screen during the navigation to the instance details.
- The *Refresh* button is pressed.
- The *Cancel* button is pressed.

Moreover, after a service operation is performed, the current status of the POT instance is retrieved by calling the read service operation (to make sure that the buffer in the model class reflects the current state of the instance). The only exception from this rule is the save operation where this is not necessary.

6.5 General Concepts

With the background information on the artifacts that are generated by POB for the POT implementation, it would also be useful to understand how the POT implementation handles key concerns like phase and status handling and so on. An understanding of these concepts is required for the implementation of custom code slots in POT, which will be described in detail in chapter 7.

6.5.1 Transaction Control

In order to ensure proper transaction handling for a POT instance and a uniform behavior of all runtime artifacts, a central transaction control functionality is generated for each POT.

Proper transaction handling means:

- Nothing is persisted unless the `SAVE` method is called.
- At the end of a transaction, all POT-instance-related buffers are empty.

Figure 77 shows a static view of the POT transaction control:

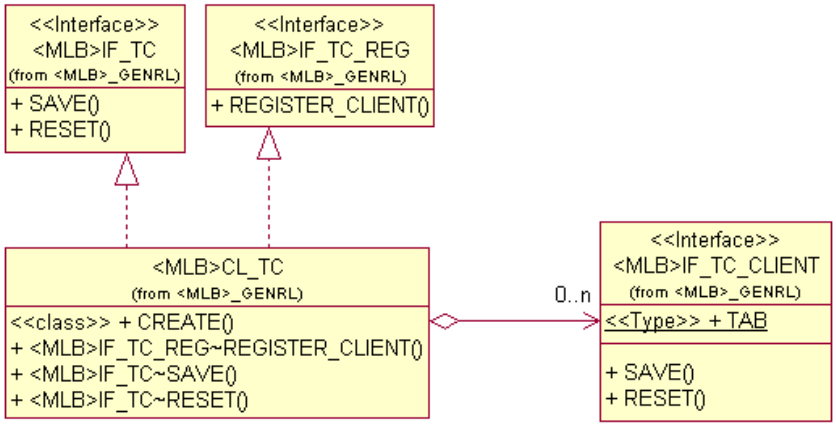


Figure 77: POT Transaction Control – Static View

All classes of POT that need to connect to transaction handling, implement the client interface `<MLB>IF_TC_CLIENT`. These clients are registered with the transaction controller `<MLB>CL_TC`. This registration is usually done by the factory that is in charge of instantiating the client class. When a transaction is finalized, the channel that is in charge of the transaction handling must decide whether a `SAVE` or a `RESET` is needed (depending on the status of the transaction) and communicate this decision to the POT runtime transaction control interface `<MLB>IF_TC`. The transaction controller then forwards this call to all its registered clients.

Note that the transaction control class neither calls `COMMIT WORK` nor `ROLLBACK WORK`. This has to be done by the channel, e.g. the POT service implementation that executes the request of a service consumer.

6.5.2 Exceptions and Message Classes

Each POT instance might find itself in an error situation. Depending on the situation, the POT instance might need to raise an exception to enable the surrounding code to do the error handling. There are three main groups of error situations and each must be handled differently:

- Coding errors that violate the assumptions of a called method will be uncovered via ASSERT statements and most often have to be fixed by developers.
- Error situations that occur in the generated part of the source code (“POT framework”) will automatically be handled by the framework.
- Errors that arise from custom implementations might be handled by the POT framework source code. However, in most cases these errors have to be handled in a different way from framework-inherent errors.

This classification is the basis for the exception handling concept in the POT design. The exception hierarchy is shown in Figure 78:

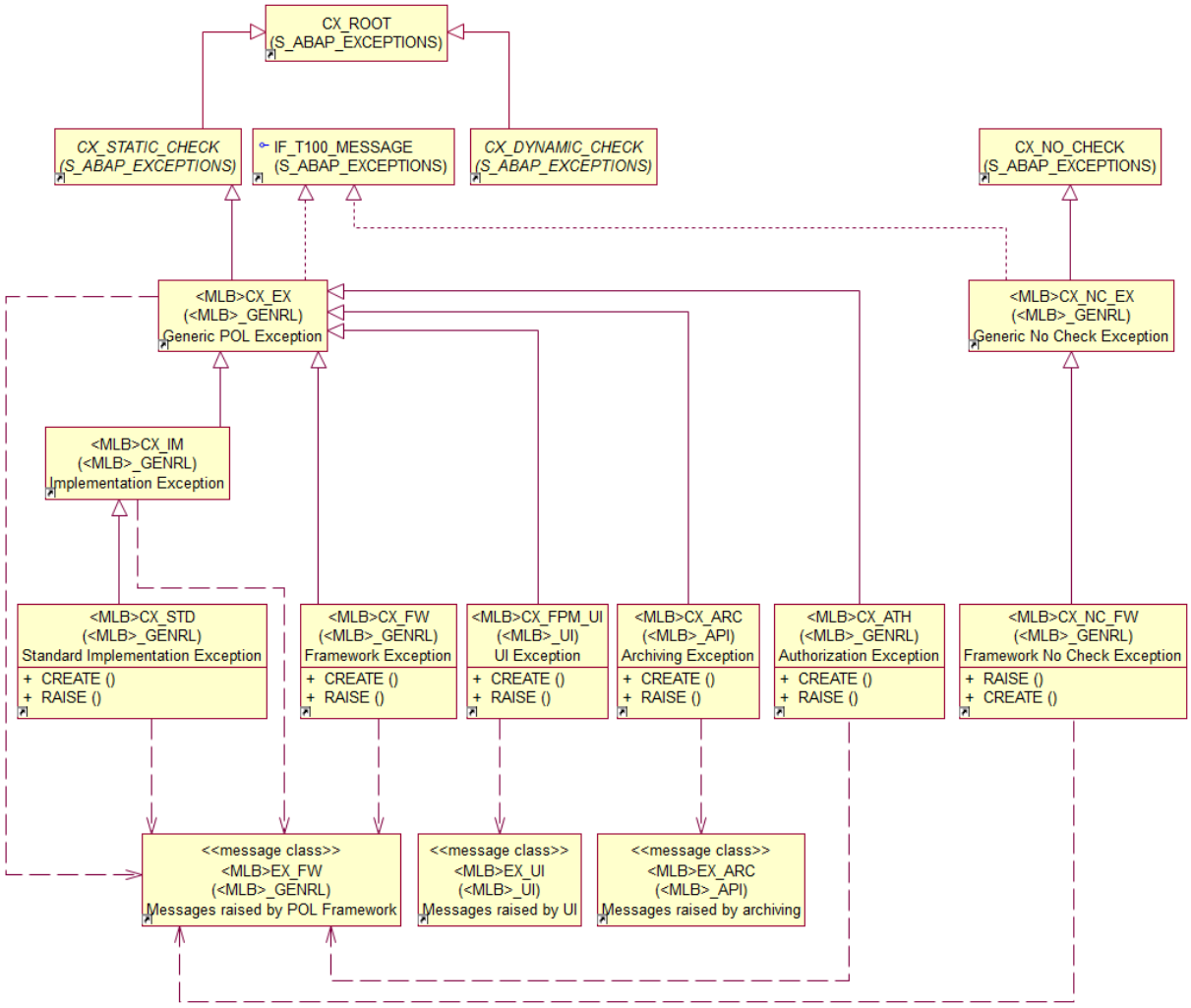


Figure 78: POT Exception Hierarchy – Static View

A central POT exception class, `<MLB>_CX_EX`, is generated for all static checks. This class inherits from `CX_STATIC_CHECK`. All other static check exception classes of the POT inherit from this class. All classes have a number of text IDs that are used for different error situations. The messages for the text IDs are taken from the message class that is associated with the exception class. In all cases except UI and archiving, the T100-texts are taken from the class `<MLB>EX_FW`.

The following exception classes are used:

- <MLB>_CX_IM for errors in custom implementations

  Implementation-related exceptions (e.g. during the implementation of code slots) should be derived from class <MLB>CX_IM. Customer-specific exceptions should also be defined as subclasses of this class.
- <MLB>_CX_STD for standard exceptions that occur in custom implementations.
- <MLB>CX_FPM_UI for UI related errors

  The T100-texts are taken from the class <MLB>EX_UI.
- <MLB>CX_ARC for errors that occur during BPO archiving

  The T100-texts are taken from the class <MLB>EX_ARC.
- <MLB>CX_ATH for authorization errors

For error situation where a "no check" exception is required, exception class <MLB>CX_NC_FW is used. This class inherits from <MLB>CX_NC_EX, which has CX_NO_CHECK as the super class.

The exceptions that are raised in service implementations are based on FSL exceptions using messages of the message class <MLB>EX_SRV.

6.5.3 Instance Management

Object instance management is based on the following basic principles:

- Each package with users has a package interface (see chapter 5.2.1).
- There is one (and only one) dedicated class (called Gatekeeper or Factory), which is exposed at the package interface and is responsible to provide instances of all other objects of this package
- The instance-providing-methods of this dedicated class are encapsulated by an interface.
- Beside this dedicated class, only interfaces are exposed at the package interface (with the exception of enumeration classes)
- Gatekeepers & factories are implemented as singletons (the static method is called CREATE); all other classes offer CREATE-methods that provide a new instance for each call (with the exceptions of enumeration classes, exception classes and SPROXY generated classes)

  This means that instance management, buffering of instances, lazy objects instantiation, virtual proxies, etc. is done by the gatekeepers & factories only.
- The returning parameters of all create/instance-providing methods are references to an interface (REF TO <MLB>IF).

6.5.4 Status Handling

The status concept of a POT has already been introduced in section 4.3:

Each POT instance has a defined overall status, which depends on the statuses of the individual BPONs as well as on the status of the BPO root node. Each of these statuses can be changed by executing a service operation on a BPON or the BPO itself. Therefore, the overall status of the BPO must be calculated each time a (changing) service operation on a BPON or the BPO is executed. To facilitate the calculation of the overall status of a POT, the design of POT incorporates the concept of a private status (Figure 79). The private status attribute is part of the private administrative data of the BPO and is not exposed to the outside.

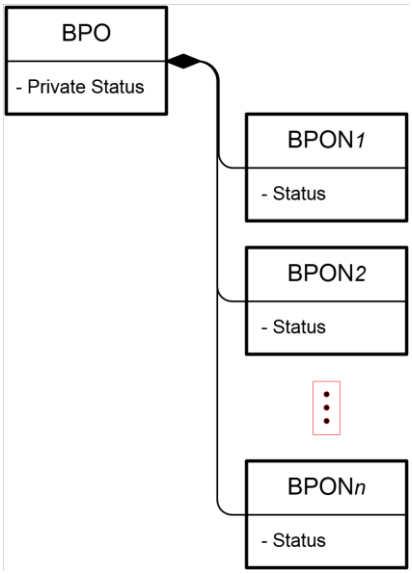


Figure 79: Private Status of a BPO

The private status of the BPO is set by the BPO API according to the relations listed in

Table 6-4:

BPO Operation	Private State
`Create`	`Unchecked`
`Update`	`Unchecked`
`Check`	(according to check result)
`ConfirmErrorCorrection`	`Unchecked`
`Execute`	`Confirmed`
`SetInMaintenance`	`In Maintenance`
`ReleaseFromMaintenance`	`Unchecked`
`Cancel`	`Canceled`
`ConfirmCompensation`	`Canceled`
`Reprocess`	`Unchecked`
`AbortExecution`	(left as is)

Table 6-4: Private Statuses of BPO set by BPON API

The overall status of the BPO is calculated by the status calculator class (Figure 80). The calculation is based on a decision matrix. The status calculator interface contains two methods that are used for different purposes:

- `CALCULATE_OVERALL_STATUS` calculates the overall status based on the BPO data.
  It uses the private method `EXTRACT_STATUSES_FROM_RELATNS` to get the status of all BPONs.
- `CALCULATE_AGGREGATED_STATUS` calculates an overall status based on a table of statuses.

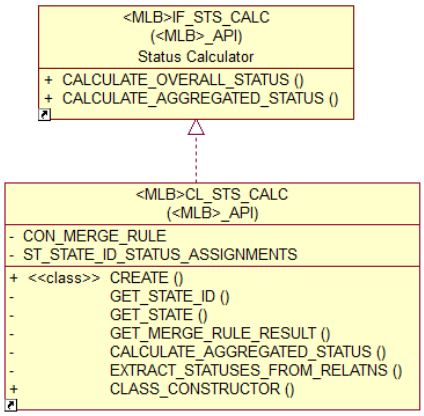


Figure 80: POT Status Calculator – Static View

In order to determine the overall status of the BPO, the statuses of the BPON instances and the private status of the BPO itself are merged together. In order to merge two statuses, the calculation compares the two statuses and retains the status that has a higher impact on the overall status as the winner. For example, the status *Erroneous* has a higher impact than the status *Checked*, because if at least one BPON is erroneous, the aggregated status of the BPO is also *Erroneous*. The calculation then repeats this comparison using the winner of each comparison for the next one until all BPON statuses and the BPO private status are compared. Once all statuses have been merged, there is one "*last man standing*", which is the final overall status.

The rules for determining the winner of a comparison between two statuses are based on the following status order:

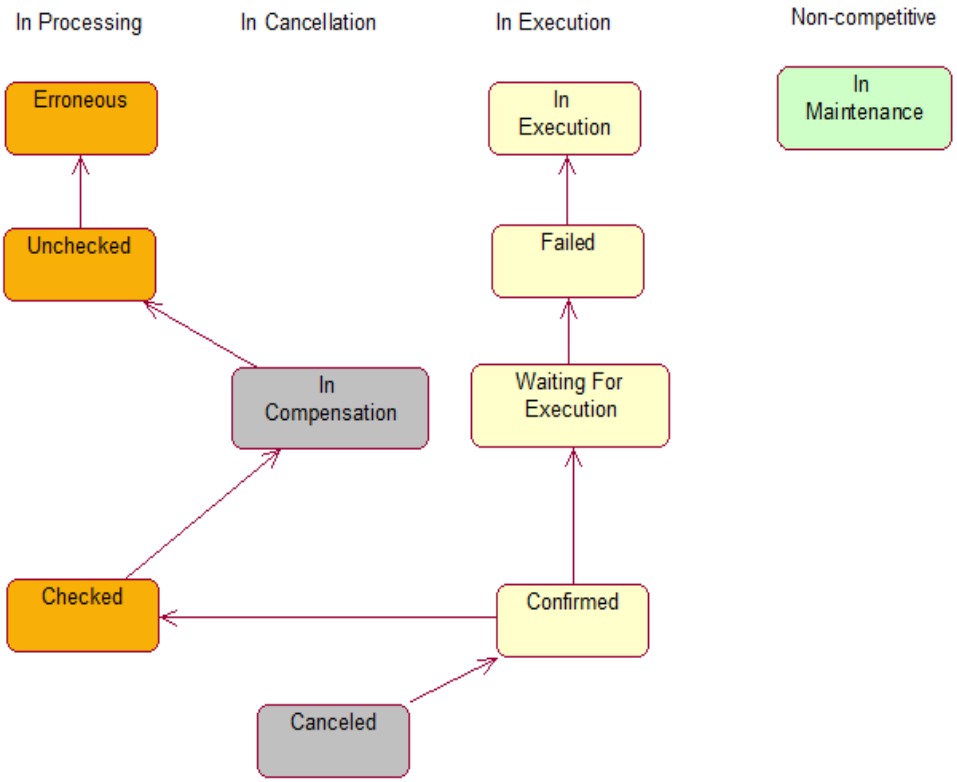


Figure 81: Order of Statuses for Overall Status Calculation

Figure 81 should be interpreted as follows:

Given two states A and B, the relation → is defined as follows:

- A → B means that A is overruled by B.
- A → B and B → C implies that A is also overruled by C (transitive).
- If there is no path from A to B and vice versa, the combination of A and B is not defined, which means that this combination must not occur during the lifetime of a process object.

The status *In Maintenance* is non-competitive in this model, because it is handled in a special way: If the private status of the BPO is *In Maintenance* the overall status is also defined as *In Maintenance* (without performing further status calculations). If a BPON has the status *In Maintenance*, it is regarded as *Unchecked* and handled as such during the overall status calculation.

6.5.5 Event Handling

The POT design follows some basic principles that necessitate the decoupling of POT artifacts at runtime:

- There should be no direct communication between the BPO and the BPONs in order to avoid direct dependencies and enable extensibility.
- There are multiple artifacts that react to a single change of a BPO/BPON instance, for example the Process Observer interface.

In general, this leads to a requirement for decoupling as the artifacts that undergo changes (or, in other words, raise events) must not be forced to know all the other artifacts that are interested in and should react on the event. This decoupling is achieved by event-based communication between the talkative (event-rising) artifacts and the interested artifacts (listeners) using the mediator design pattern. The mediator is the "man in the middle" that registers on all events of all "talkative" objects and forwards these events to interested parties.

Based on this pattern, the event handling in a POT is implemented as follows:

Figure 82 shows a static view of the event raising.

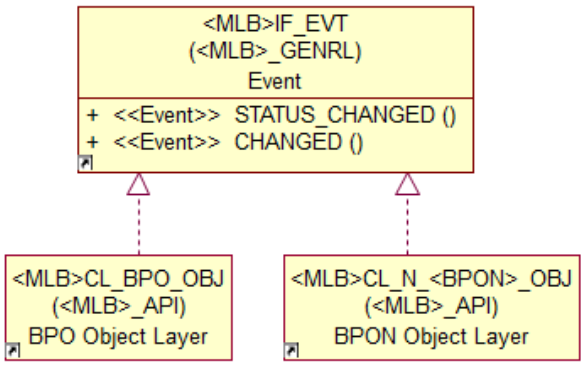


Figure 82: Raising of Events - Static View

Each BPO/BPON can raise the events `CHANGED` (if the data contained in the Details or the Process Control Constraints has been changed) and `STATUS_CHANGED` (if the status of the sending instance has changed). Event handling will be initialized by a central event listener, called Event Mediator.

A static view is shown in Figure 83 :

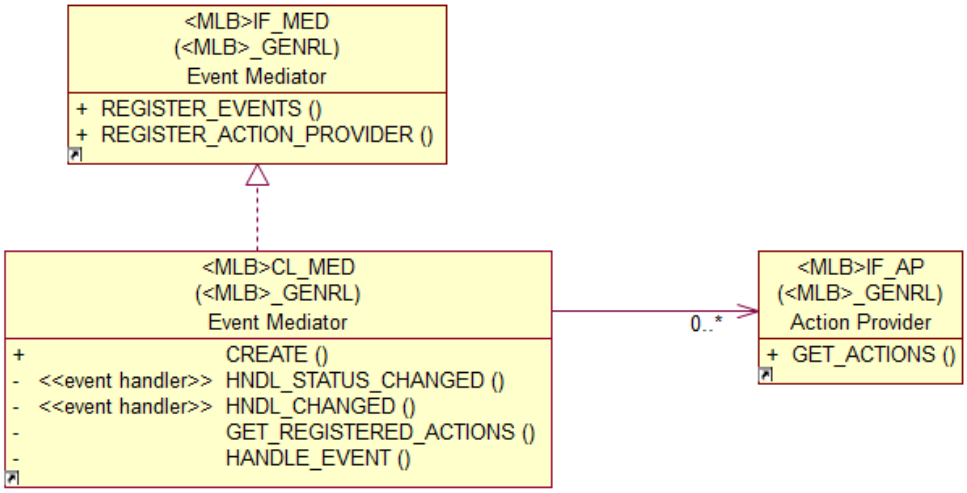


Figure 83: Event Mediator - Static View

The Event Mediator listens for events on all registered objects.

Action providers can register at the event mediator, which will in turn ask the providers to provide relevant actions for a specific event. The concrete actions for a given event type are provided by Action Providers, which implement the interface `<MLB>IF_AP`. Factories are responsible for registering the events and action providers with the mediator.

6.5.6 Cross-Reference Handling

Cross-references (introduced in section 4.5.5) between BPON nodes need to be handled carefully in the life cycle of a POT instance.

Cross-references are handled at the following points in time:

- The cross-references are validated during service implementations.

  If a cross-reference is filled at all: Only `UUID` XOR (!) `TypeCode` and `Reference` is allowed. The `TypeCode` must be valid and must not refer to the BPO.
- During the *Check* phase, the cross-references are checked for
  - Occurrence matching and reference to allowed node types (according to modeling input)
  - Validity (`UUID` XOR (!) `TypeCode` and `Reference`)
  - Existence of referenced BPON
  - Uniqueness
  - Status of referenced BPON

The functionality for handling cross-references is split into a general helper (`<MLB>CL_CRH`), which provides the concrete cross-references of an instance, and a check helper (`<MLB>CL_CRC_HLP`) that executes the cross-reference checks during the *Check* phase.

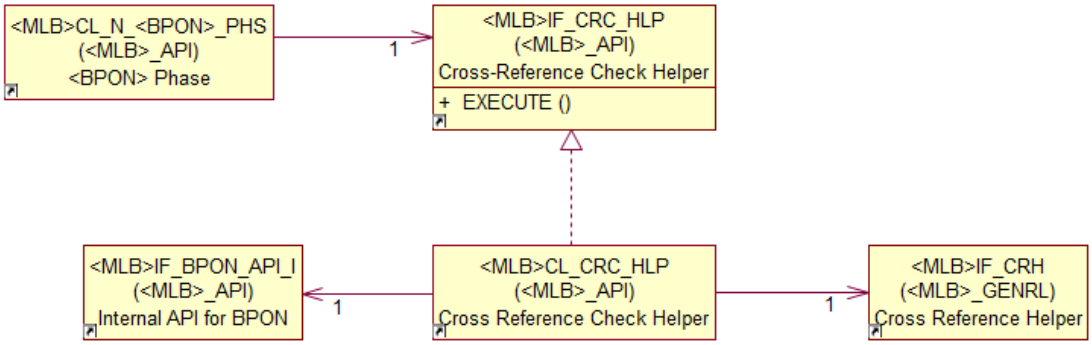


Figure 84: Cross-Reference Handling – Static View

The Cross-reference Helper provides the concrete cross-references of a concrete node instance.

The Cross-reference Check Helper executes all cross-reference-related checks during the *Check* phase, based on the concrete BPON data and the information from the Cross-reference Helper. In contrast to the validation of a service implementation, the check does not stop at the first error but tries to fetch all the errors and merges the results.

Figure 85 shows the dynamic view of the Cross-reference Check Helper:

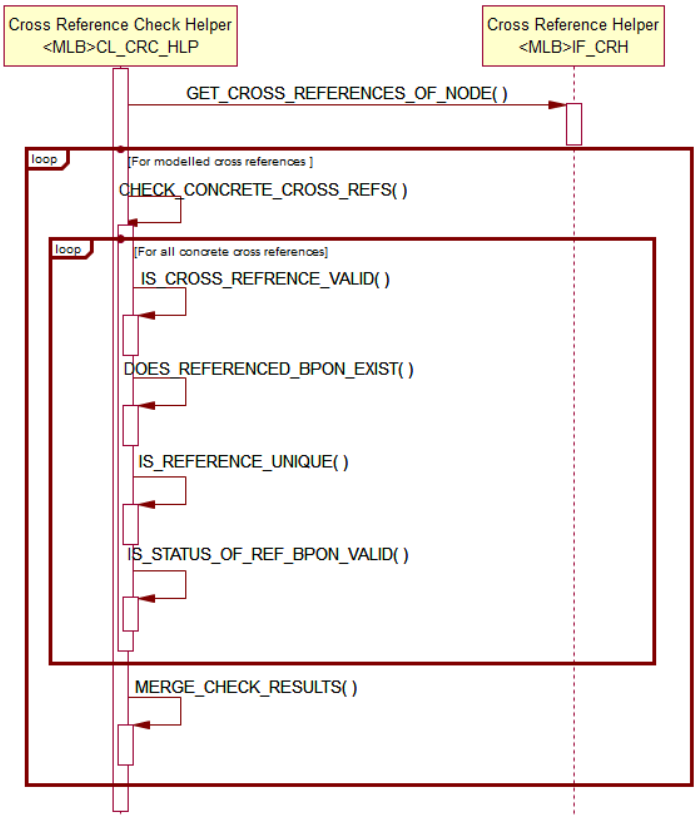


Figure 85: Cross-Reference Check Helper – Call Sequence

6.5.7 Process Monitoring & Analytics

The Process Observer is used to monitor a POT instance during its complete life cycle. The obtained data can be used to analyze and optimize processes with respect to predefined KPIs. The Process Observer provides an appropriate infrastructure to log the activities (also known as the process steps) of an executed POT instance.

For a POT, the following steps are logged as process observer activities:

- Execution of a POT-related inbound service implementation (except read-only operations)
- Execution of a back-end-facing outbound service implementation (only for those services for which this feature has been activated in the POB Specification Wizard)
- Status Changes of the BPO

Figure 86 shows the static view of the Process Observer integration:

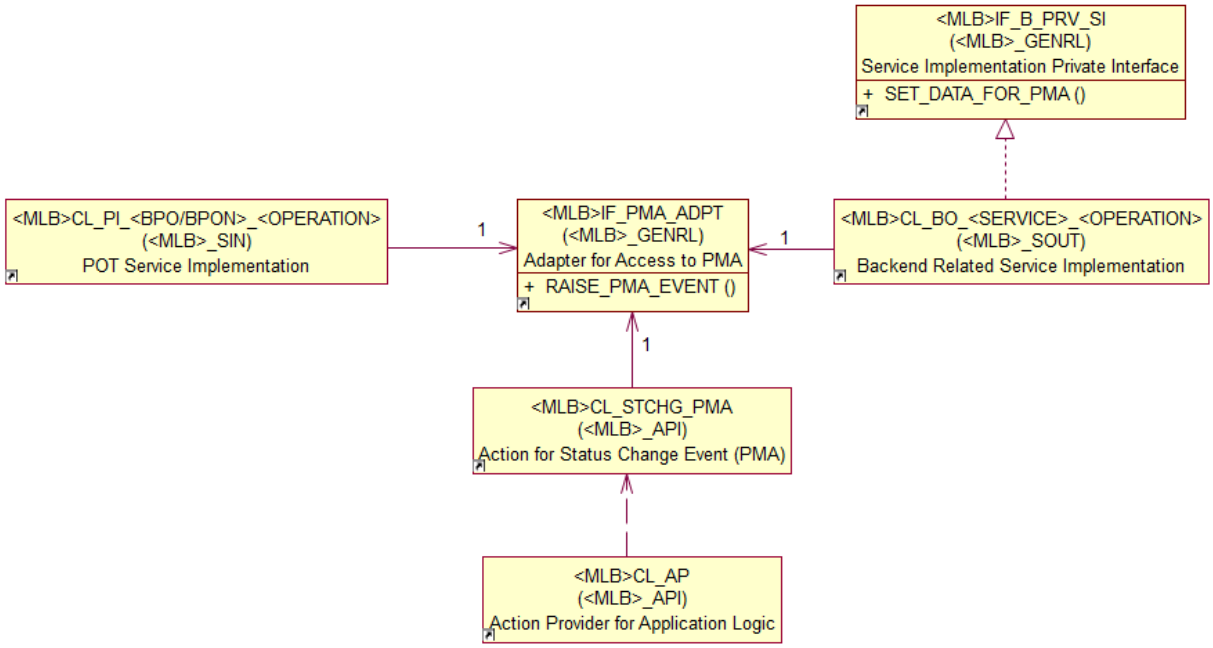


Figure 86: Process Observer Integration – Static View

The service implementation classes and the action class for the "status changed" event use the PMA Adapter to log their activities. The outbound service implementation implements the additional interface `<MLB>IF_B_PRV_SI`. This interface is used to set that data beforehand that is no longer accessible during execution.

Figure 87 shows the call sequence for that part for an outbound service execution where the Process Observer integration is relevant.

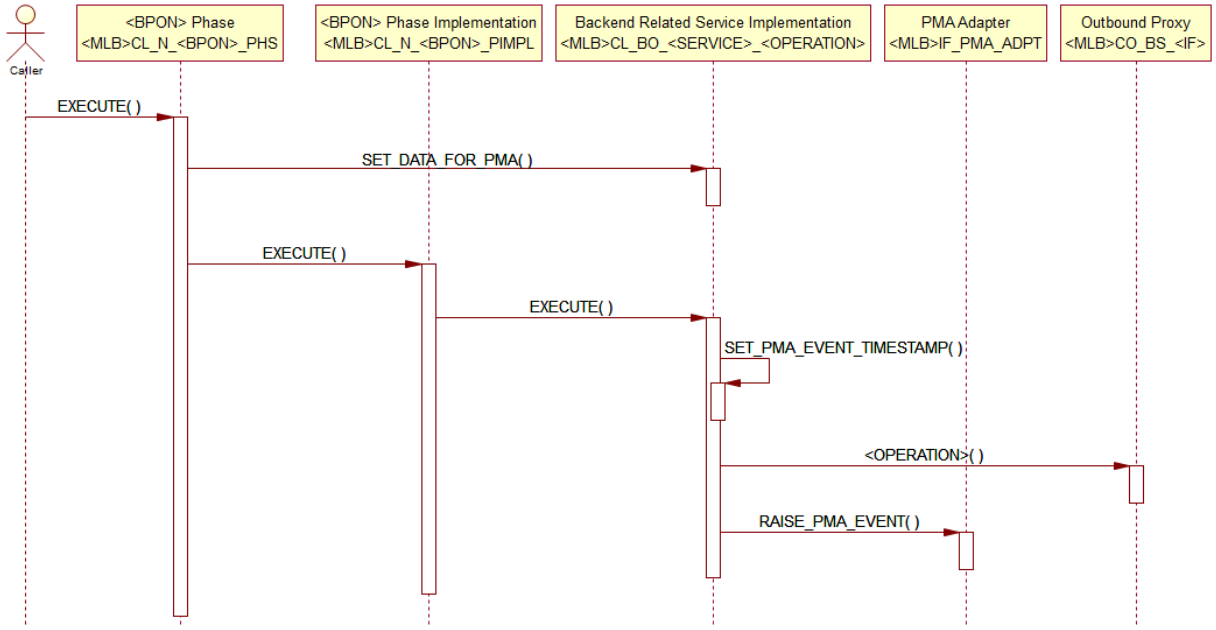


Figure 87: Process Observer Integration into Outbound Service Implementation – Call Sequence

Once the phase class for a specific BPON has been executed, the data that is relevant for the Process Observer integration is set to the service implementation by calling the method `SET_DATA_FOR_PMA`. This data is required to log the Process Observer activity when it comes to execution of the service implementation. When the service is executed by the Phase Implementation, first a current timestamp is drawn, then the back-end operation is executed and finally the Process Observer activity is logged by calling the method `RAISE_PMA_EVENT`.

For information on the integration of the Process Observer with regard to POT-related inbound services, see section 6.3.1.1.

Figure 88 shows the call sequence for that part of the processing of a "status changed" event where the Process Observer integration is relevant:

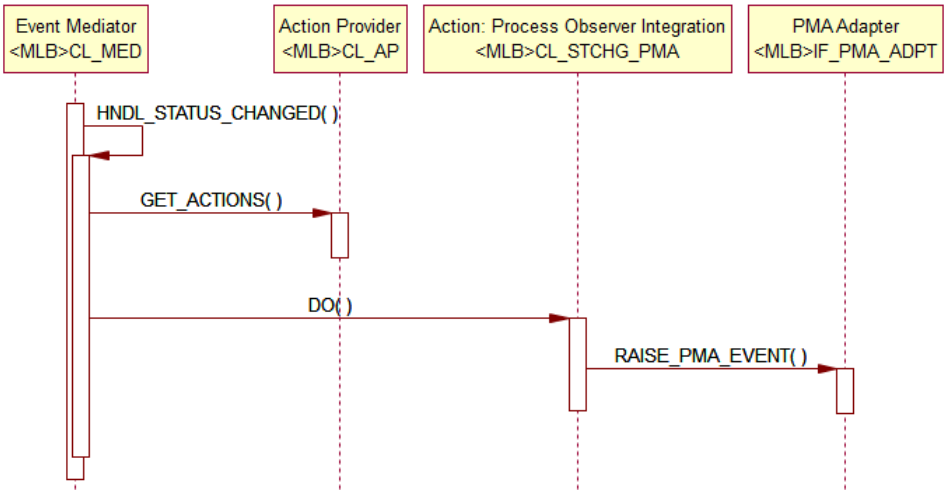


Figure 88: PMA Events on Status Change of BPO – Call Sequence

The event is handled by the Event Mediator, which gets all the relevant actions from the Action Provider(s) and executes all provided actions. The diagram only focuses on the action for process observer integration. This action uses the PMA adapter to log the status change as a Process Observer activity.

The processes can be displayed using the ABAP transaction `POC_MONITOR`. From this transaction it is also possible to open the editing UI for a POT instance. To provide this function, we use an implementation of the filter-dependent BAdI `POC_MONITOR`.

Figure 89 shows a static view of the BAdI:

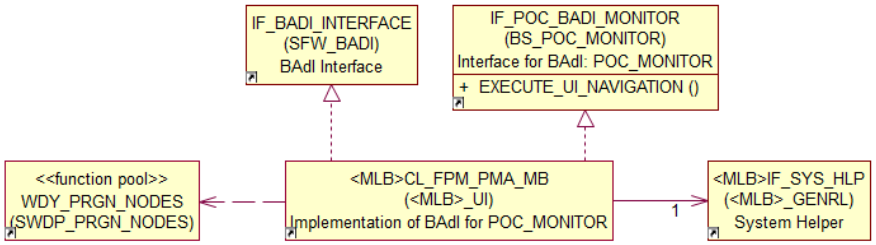


Figure 89: PMA BAdI for UI Navigation

6.5.8 Archiving

The archiving solution for POT instances follows the general approach for archiving and the product standard SAP Information Lifecycle Management (also known as ILM). As with most of the other features, POB also generates most of the required artifacts for archiving.

The archiving of process object instances is done in two steps:

- POT instances are selected from the database and written into the archive using the archiving write program.
- Archived objects are deleted from the productive databases using the archiving delete program.

In addition to the standard archiving functionality, the write program also provides the following two ILM features:

- Data Destruction

  Data is deleted from the database without writing it into the archive.
- Snapshot

  All data is written into the archive without deleting it from the database.

Only process object instances in a "final" state i.e. instances with status *Confirmed* or *Cancelled*) can be archived. In addition, the residence time of an instance has to be passed. Residence and retention times must be configured in ILM customizing by the customer.

During an archiving write run, all relevant entries from the tables `<MLB>_XO` (POT instance data), `<MLB>_XS` (POT search) and `<MLB>_XV` (POT version) tables are selected and written into the archive. Entries from the `<MLB>_XM` table (message tracker) are not archived but only deleted from the database during the delete run, because for completed process object instances this information is not needed any longer.

The Archive Search UI is a separate application based on Floorplan Manager for Web Dynpro ABAP (FPM). It can be used to access archived data as well as instances that are still on the database. Therefore, the archive search directly accesses the POT archive files and database tables. Archive search is not included into the `Read` or `Find` services of a POT in order not to mix up access to productive data with access to data that has been archived. (The retrieval of archive data is probably only needed in very rare cases).

As shown in the static view in Figure 90, the central class related to the archiving functions is the Archivist (<MLB>_A_AVST):

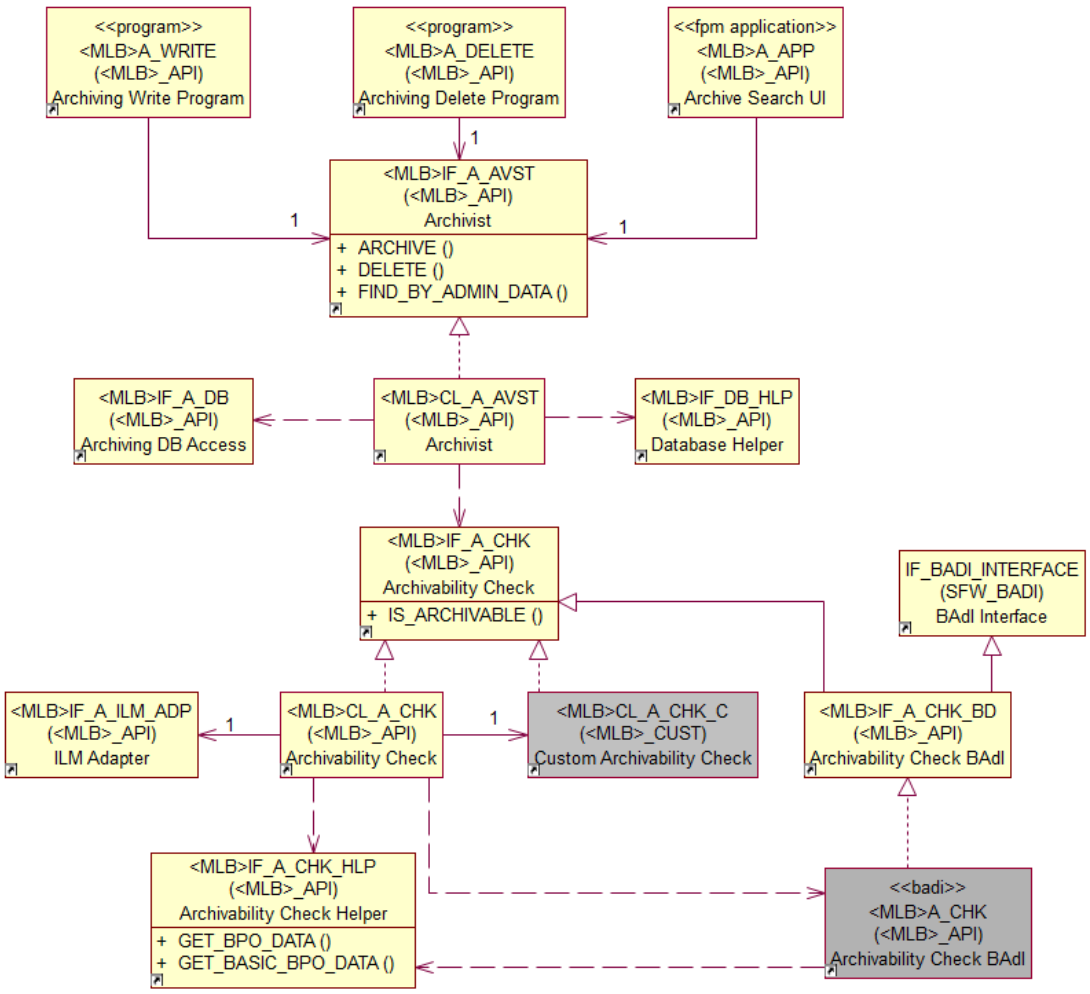


Figure 90: POT Archiving – Static View

The Archivist reads and deletes data from the database via the Archiving Database Access (<MLB>_A_DB) and writes data into the archive via the Archiving Adapter. Archivability checks are executed using the Archivability Check (<MLB>_A_CHK), which checks the residence time via the ILM Adapter (<MLB>_A_ILM_ADP) and calls the Archivability Check BAdI (<MLB>_A_CHK_BD) for additional custom checks.

The Archiving Write Program (<MLB>_A_WRITE) and the Archiving Delete Program (<MLB>_A_DELETE) are simple reports that delegate the work to the Archivist.

The archive method implementation uses the DB Access class to select all the relevant DB entries for the BPO instances that are to be archived. All calls to the Archive Development Kit (ADK) are encapsulated by the Archiving Adapter. Before an instance can be archived, archivability checks are performed to ensure that all prerequisites are fulfilled.

Figure 91 shows the call sequence for the archiving run:

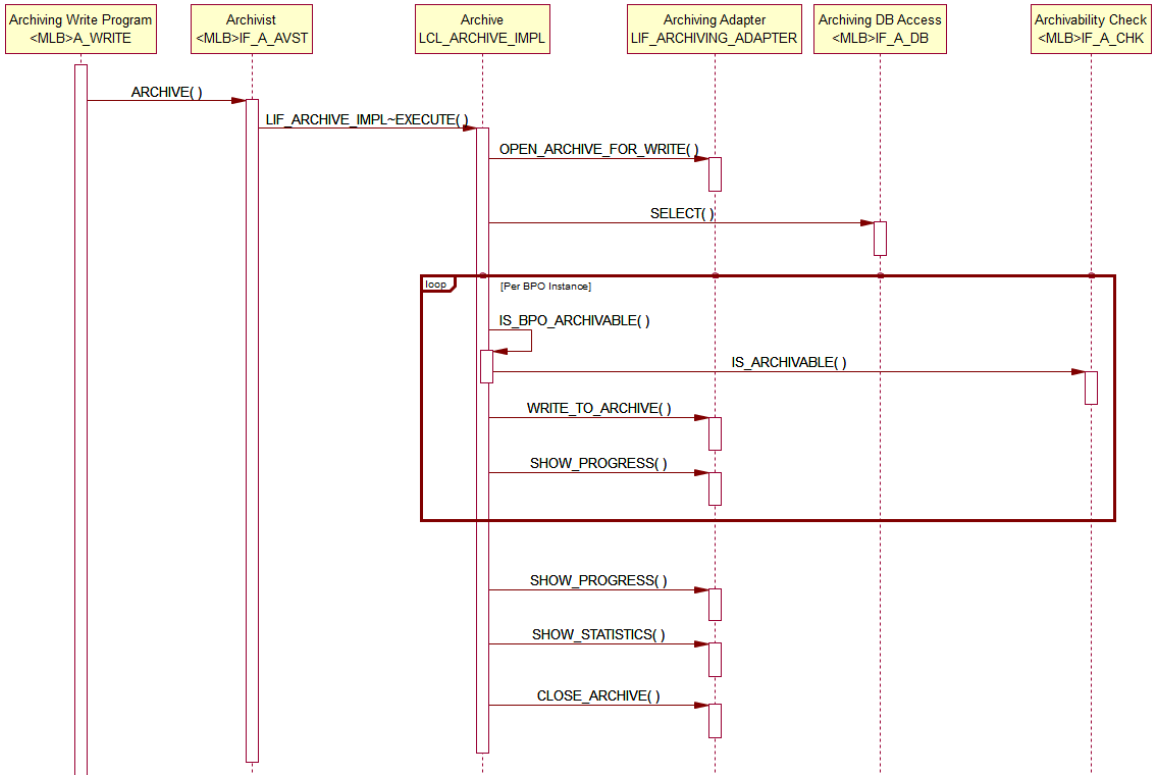


Figure 91: Archiving Run – Call Sequence

The following three steps are carried out during an archiving run in order to check whether an object can be archived:

- Standard check

  Checks residence time settings and status of an instance
- Custom check

  Code slot where the check result can be further restricted (but not revised)
- BAdI

  Same as for the code slot, but in a BAdI

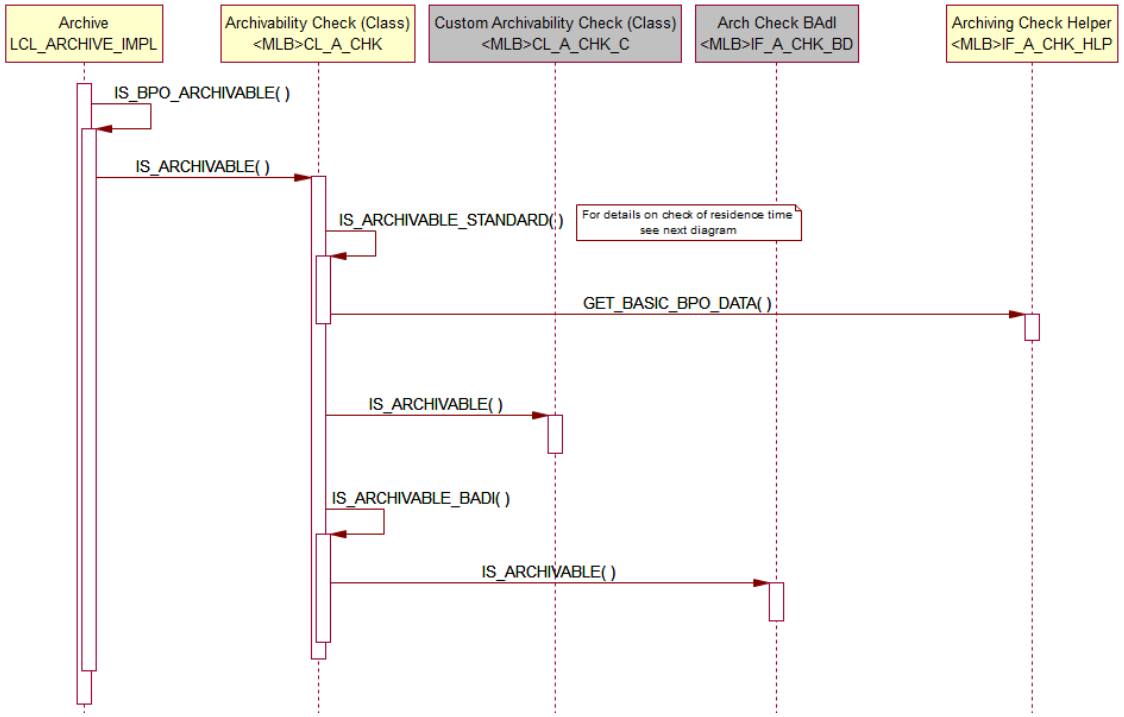


Figure 92: Archivability Check (Standard, Custom and BAdI) – Call Sequence

The BAdI is mainly intended for POTs delivered by SAP for which a customer wants to further restrict the archivability check. If the POT is built by the customer, the code slot can be used instead.

The Archivability Check Helper is passed to all check methods. It provides access to BPO data (`TS_BPO_TOTAL`) and basic BPO data to be used in the checks. Basic BPO data provides better performance, as in this case, it is not the complete BPO that has to be assembled.

Figure 93 shows the call sequence for the determination of the residence time:

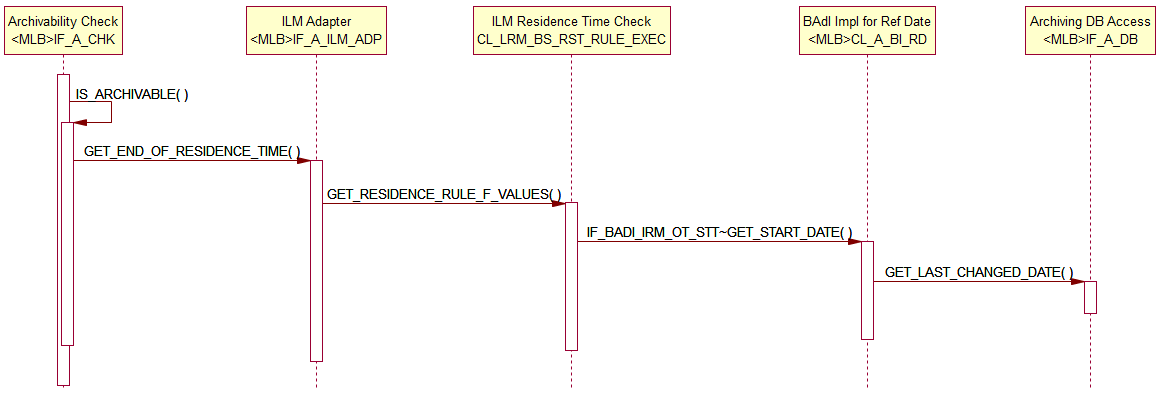


Figure 93: Check of Residence Time – Call Sequence

To check the residence time of an instance, the reference date for the residence time has to be determined. For a process object instance, this is the last change date, i.e. the creation date of the newest version of this instance in the version database (XV).

In case the reference date cannot be specified directly (one dedicated field on the database), ILM provides a BAdI to determine the date. As the last change date has to be calculated from all versions belonging to an instance, an implementation of this BAdI belongs to each process object type.

6.5.8.1 Archive Search UI

The archive search UI is based on Floorplan Manager for Web Dynpro ABAP (FPM). It is completely generated and executable without any manual steps. The UI provides a search screen with a result list (see Figure 94). If an entry from the result list is selected, the process object instance data is displayed as an XML document in a new browser window. The reason why the archive search UI is not designed similar to the editing UI but displays the instance data as plain XML, is that the archive search UI must be able to display old data. This old data does not always comply with the current structure of the data and could therefore not be displayed otherwise.

On the selection screen, an archiving data source can be selected to specify whether the search should be performed in the archive or on the database, or using both sources.

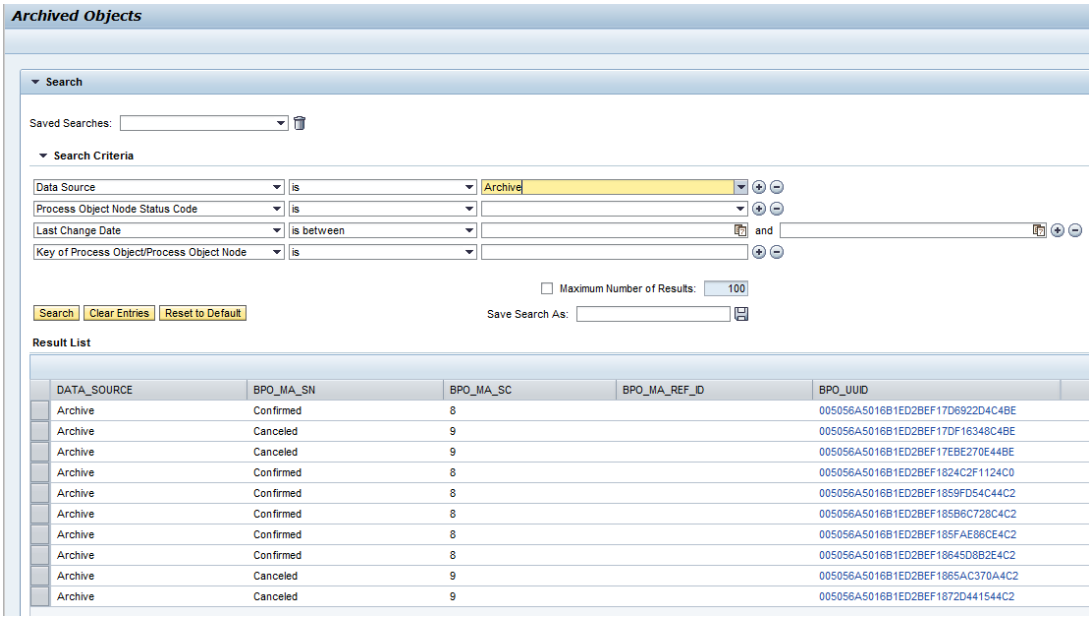


Figure 94: Archive Search UI

In the search result list, the process object instance ID is displayed as a link to the instance details. The details are displayed in XML format (see Figure 95).

Figure 95: Instance Data as XML

7 POT Implementation

The Process Object Builder generates most of the parts that a process object type consists of. Once a POT has been generated completely, a POT developer can implement custom logic in dedicated places.

This chapter explains what has to be done to complete a POT implementation. We assume that you are already familiar with the sample business scenario (see section 2.5) and the high-level and the detailed design of a POT, especially with phase and status handling (see sections 6.5.1 and 6.5.4).

7.1 Custom Phase Implementation

This section explains how a developer implements the code slots for the phases. It first discusses basic principles and afterwards provides details for each phase implementation.

7.1.1 Conventions and Disclaimers

The following sections contain source code examples (listings) that are used to illustrate how to implement the code slots of the phase implementation. The examples are taken from the training course for SAP Process Object Builder. Therefore the prefix for all artifacts is `/PL9/<...>_GXX_FB01_`.

Usually, POT developers add their own methods to the code slot classes in order to structure the code slots. However, to improve conciseness, the code samples in this guide do not contain any additional methods.

Moreover, exception handling should normally take place at the end of a code slot, i.e. all logic is embraced by the try-catch-block. However, the code samples in this guide keep the try-catch-block as short as possible to avoid unnecessary line breaks.

The following listing shows a small sample excerpt:

```
* common prefix for all artifacts: /pl9/<...>_gxx_fb01_
data: lr_bpon type /pl9/if_gxx_fb01_n_flgt_pimpl=>ty_ref_to_create_request.

try.
    ... "something has been omitted here

    loop at c_bpons into lr_bpon.
      "where exception handling is in the examples
    endloop.

  catch cx_fsl_error_conflict_mapping. "where exception handling should be

endtry.
```

Listing 7-1: Source Code Example

7.1.2 Goals of Phase Implementation

The main goal of the phase implementation of a POT is to call the back-end services that were selected during POT modeling. In order to do this, the developer has to do the following in the corresponding code slots:

- Check the preconditions of the back-end services
- Build the service requests (map from BPON representation to service signatures)
- Evaluate the service responses (map from service signature back to BPON representation, set BPON status, handle errors and exceptions)

Depending on the phase this is done with different focuses (see section 4.5).

7.1.3 Basic Principles of Phase Implementation

All classes that contain code slots are located in the custom package (`<MLB>CUST`). The code slots for the phases are contained in the phase implementation classes; there is one class for each BPON (`<MLB>CL_N_<BPON>_PIMPL`).

These classes contain the following methods:

- An `INIT` method to initialize additional attributes that are required for the phase implementation
- Methods that correspond to the phases of a POT:
  - `<MLB>IF_N_<BPON>_PIMPL~CREATE`
  - `<MLB>IF_N_<BPON>_PIMPL~CHECK`
  - `<MLB>IF_N_<BPON>_PIMPL~EXECUTE`
- A method `<MLB>IF_N_<BPON>_PIMPL~PPC_<IF>_<OP>` for processing asynchronous inbound confirmations (if available; `<IF>` = service interface abbreviation, `<OP>` = service operation abbreviation)

Figure 96 gives an overview of the phase implementation and the related classes:

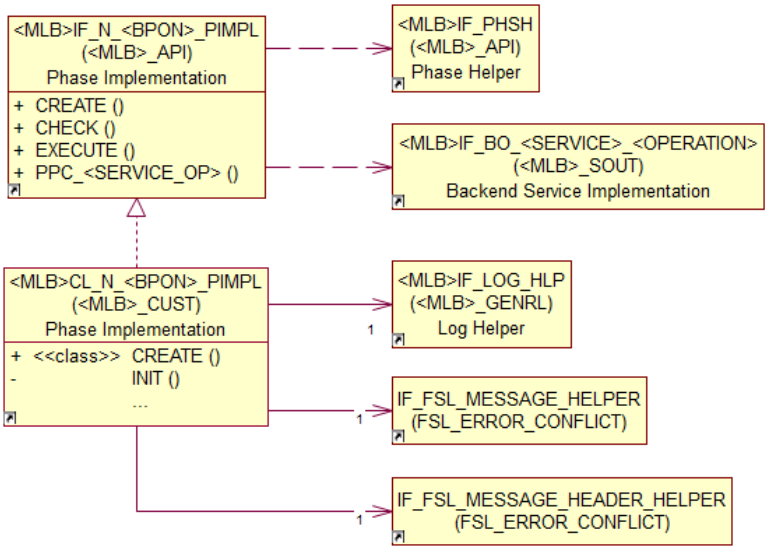


Figure 96: Phase Implementation

7.1.3.1 Custom Implementations

During the implementation of the code slots, a POT developer can add attributes and methods to the phase implementation classes to structure the implementation. Just like the code slots themselves, additional attributes and methods do not get lost when a POT is regenerated.

In addition, POT developers can create their own classes and interfaces to be used by the code slot implementations. In order to ensure that custom artifacts are not overwritten by the builder, these artifacts must follow the naming convention `<MLB>*_C_*`, with `C` denoting a custom artifact.

Apart from classes and interfaces, typical examples for custom implementations are exceptions and message classes (see section 7.1.3.3).

7.1.3.2 Access to Back-End Services

The importing parameter `I_BACKEND_SERVICES` provides the access to the back-end services of the node and phase that were selected for a specific BPON. This parameter is available in the code slots. It is a structured type and contains one entry for each service operation. The entries are references to the implementations of the back-end service counterparts and follow the naming convention <service interface abbreviation>_<service operation abbreviation>. A POT developer only needs to call the `EXECUTE` method of the referenced implementation to execute a back-end service.

7.1.3.3 Error Handling

A POT developer has two options for handling errors in the code slots: Raise an exception or add an error message to one of the logs.

Business errors should be returned as log messages, whereas situations that require an immediate abortion of the code slot should be handled via an exception. The procedure for returning error messages through the log is identical for all phases:

- Set the node status to *Erroneous* (*Failed* in the *Execute* phase), based on the value of the result code (or its equivalent in case of non-standard services) returned by the back-end services
- Add errors from the back-end services to the provider log (*Create* and *Execute* phase) or the process log (*Check* phase)
- Add errors from the business checks in the *Check* phase to the process log

Exceptions can be raised using the generated standard exception class (`<MLB>CX_STD`), or a custom subclass of the generated implementation exception class (`<MLB>CX_IM`). The custom subclass must be created manually and should be the preferred way to raise custom exceptions. As the creation of the implementation exception class is private, the statement `RAISE EXCEPTION TYPE` is not possible, as it implicitly calls the constructor of the class. Instead, public static `CREATE` and `RAISE` methods have to be created for the subclass. Figure 97 illustrates the general pattern for raising exceptions:

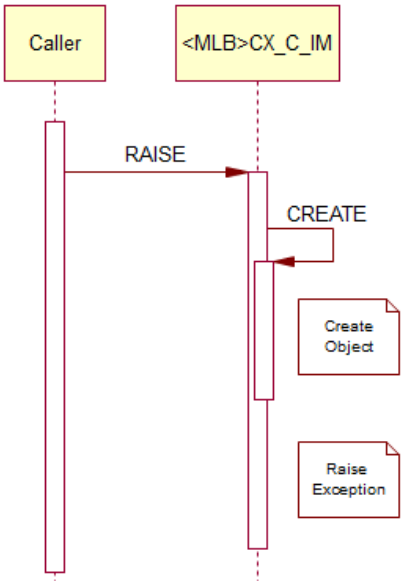


Figure 97: POL-specific Pattern for Raising Exceptions

When an exception is raised during code slot execution, the execution of the code slot is aborted. The exception will be caught in the service implementation from which the code slot had originally been called (e.g. the `Create<BPO>` service for the *Create* phase). In this implementation, the error messages from the exception are transferred to the service log and the result code is set to Failed.

Example 1: Fill Provider Log

Listing 7-2 gives an example for how to use the log helper to add error messages that have been received from a back-end service to the provider log. This example is taken from the *Create* phase of the Flight BPON of the sample POT:

```
data: lr_bpon type /pl9/if_gxx_fb01_n_flgt_pimpl=>ty_ref_to_create_request.

loop at c_bpons into lr_bpon.
  ...
  if ( ce_fsl_proc_result_code=>s_get_instance(
    ls_flqo_qry_response-log-business_document_processing_r ) =
      ce_fsl_proc_result_code=>successful ).
    ...
  else.
    lr_bpon->public_admin_data-log-provider_log = ls_flqo_qry_response-log

    lr_bpon->public_admin_data-status_code =
      /pl9/ce_gxx_fb01_node_sc=>erroneous->get_int_value( ).
  endif.
endloop.
```

Listing 7-2: Transferring Back-End Errors to the Provider Log

In case the back-end service could not be executed successfully (business document processing result code <> `Successful`), the provider log (contained in the public administrative data) is filled with the log that is returned by the back end. The log helper is used to convert the log from the external GDT representation into the internal FSL representation. The data type of the external log (`Log_V1`) is indicated by FSL enumeration class `CE_FSL_DATATYPE`.

Example 2: Fill Process Log

Listing 7-3 shows an example for adding an error message to the process log. The example is taken from the *Check* phase of the Flight Booking BPON of the sample POT. The message is first written into the system variable. From there, it is extracted by the message helper. The log helper then fills the process log with the given table of messages and the processing result code.

```
data: lr_bpon type /pl9/if_gxx_fb01_n_fbok_pimpl=>ty_ref_to_public_complete,
      lr_checked_bpon type ref to /pl9/if_gxx_fb01_it_comp=>ts_bpon_al_check_res,
      ls_availability type /pl9/wfccflight_booking_avail4,
      lv_flight_id    type string,
      lv_dummy        type string ##needed,
      lt_messages     type bapirettab.

loop at i_bpons into lr_bpon.
  ...
  append initial line to e_checked_bpons reference into lr_checked_bpon.
  ...
  if ( ce_fsl_proc_result_code=>s_get_instance(
    ls_fbao_chk_response-log-business_document_processing_r ) =
      ce_fsl_proc_result_code=>successful ).

    ls_availability = ls_fbao_chk_response-flight_booking_availability.

    if ( ls_availability-available_seat_number_value > 0 ).
      lr_checked_bpon->public_admin_data-status_code =
        /pl9/ce_gxx_fb01_node_sc=>checked->get_int_value( ).
    else.
*     1. set BPON status to Erroneous
      lr_checked_bpon->public_admin_data-status_code =
        /pl9/ce_gxx_fb01_node_sc=>erroneous->get_int_value( ).

      concatenate ls_availability-flight_id-airline_id
                  ls_availability-flight_id-connection_id
                  ls_availability-flight_id-planned_flight_date
            into lv_flight_id separated by space.
      message e010(/pl9/gxx_fb01_ex_cus) with lv_flight_id into lv_dummy.
      append mr_message_helper->fill_bapiret2_by_sy( sy ) to lt_messages.

*     2. write error message into process log
      lr_checked_bpon->public_admin_data-log-process_log =
        mr_log_helper->get_log_by_messages(
          i_messages = lt_messages
          i_bus_doc_proc_result_code = ce_fsl_proc_result_code=>failed ).
    endif.
  else.
    lr_checked_bpon->public_admin_data-status_code =
      /pl9/ce_gxx_fb01_node_sc=>erroneous->get_int_value( ).
  endif.
endloop.
```

Listing 7-3: Writing an Error Message into the Process Log

1. The back-end service was executed successfully, but there are no seats available. The BPON status is therefore set to *Erroneous*.

2. An error message is written into the system variable using the "`MESSAGE…INTO`" statement. From there, it is extracted by the message helper and written into the process log by the log helper. The processing result code is set to `Failed` using FSL enumeration class `CE_FSL_PROC_RESULT_CODE`.

> **Recommendation**
>
> A convenient way for adding errors to a log is to use the message helper to extract an error message from the system variable, and then the log helper to convert the extracted message into an FSL log.

Example 3: Raise Custom Exception

Listing 7-4 shows an example for raising a custom exception. In this example the exception is due to an unsupported creation instruction code (see section 7.1.3.6 for details). The sample code has been taken from the *Create* phase of the Flight BPON (sample POT).

```
data: lr_bpon type /pl9/if_gxx_fb01_n_flgt_pimpl=>ty_ref_to_create_request.

* validate creation instruction code
loop at c_bpons into lr_bpon.
  if ( /pl9/ce_gxx_fb01_crte_ic=>get_instance_by_value(
    lr_bpon->public_admin_data-creation_instruction_code ) <>
      /pl9/ce_gxx_fb01_crte_ic=>all_matching ).

    /pl9/cx_gxx_fb01_c_custom=>raise(
      textid = /pl9/cx_gxx_fb01_c_custom=>invalid_instruction_code
      code   = |{ lr_bpon->public_admin_data-creation_instruction_code-content }|
      type   = /pl9/ce_gxx_fb01_node_ty_c=>flgt->get_name( ) ).
  endif.
endloop.
```

Listing 7-4: Raising a Custom Exception

The custom exception is a subclass of the generated implementation exception. As its instantiation is private, exceptions are raised via its public static `RAISE` method. If an instance of the exception is required without raising the exception, the public static `CREATE` method can be used (not shown in this example).

7.1.3.4 Data Access in the Phase Implementation

The following Table 7-1 gives you an overview about what can be accessed or executed in the code slots of which phase:

What	Create Phase	Check Phase	Execute Phase
Synchronous Service	Read (Change)	Read (Change)	Read, Change
Asynchronous Service	Not allowed	Not allowed	Change
Details	Available, Changeable	Available	Available
Private Details	Available, Changeable	Available, Changeable	Available, Changeable
Provider ID (in Administrative Data)	Available, Changeable	Available	Available, Changeable

What	Create Phase	Check Phase	Execute Phase
Process Control Constraints	Available, Changeable	Available	Available
Process Log	Not available	Changeable	Not available
Provider Log	Changeable	Not available	Changeable
Other Nodes	Available (only predecessor nodes)	Available	Available (only predecessor nodes)

Table 7-1: Read and Write Accesses in the Phase Implementation

Caution

Although it is technically possible, we strongly recommend not to call any changing service in the *Create* and *Check* phases, as changes to the back end cannot be tracked in these phases (see back-end modification state in section 7.1.3.5). In case of errors, such changes are difficult to compensate, because the changes are not reflected by the POT instance.

Caution

Although technically possible, we strongly recommend not to call more than one changing service per code slot, because in case of errors it might not be clear which changes to the back end were already executed, and must therefore not be executed again during reprocessing.

7.1.3.5 Back-End Modification State

The back-end modification state of a BPON indicates whether the corresponding back-end business object was modified during execution. This information is particularly relevant in cases where you need to decide whether existing modifications need to be compensated if a POT instance is canceled at a later point in time (service operation `Cancel`). A POT developer must explicitly set the back-end modification state in the code slot of the *Execute* phase.

7.1.3.6 Important GDTs

The following table contains GDTs that are important during code slot implementation, together with a description and a sort name that will be used throughout this document:

GDT	Short Name	Description
`BusinessDocumentMessageHeader`	Message header	Contains administrative information about a service message. It is part of all standard services from SAP.
`BasicBusinessDocumentMessageHeader`	Basic message header	Only contains identifying information about a service message. It is part of some of the services in the SAP Business Suite.
`BusinessDocumentProcessingResultCode`	Result code	Indicates whether or not a service call was successful. It is part of all standard services

GDT	Short Name	Description
		from SAP.
`Log`	-	Contains messages that occur during application processing (the log items) together with a `BusinessDocumentProcessingResultCode` and a `MaximumLogItemSeverityCode`.
`Log_V1`	-	New version of GDT `Log`
`LogItemSeverityCode`	Severity Code	Indicates the severity of a log message (GDT `Log`).
`MaximumLogItemSeverityCode`	Maximum severity code	Denotes the highest severity of all messages contained in a log.
`RelatedObjectExistenceAssumptionNodeCreationInstructionCode`	Creation instruction code (CIC)	See explanation below

Table 7-2: Important GDTs

Creation Instruction Code

The GDT `RelatedObjectExistenceAssumptionNodeCreationInstructionCode` is an instruction to create one or more nodes based on the assumption that the related back-end business objects exist. The instruction defines whether the data of the request to the process object or the related back-end business object is to be taken over into the node data (also see the GDT catalog). In other words, a consumer of a POT must specify the following aspects in the create request:

- Whether it is expected that one or more back-end business objects (the related object) exist for the given criteria, or not (the assumption)
- Whether the data from the back end or from the create request should be used as node data in case objects exist (the instruction)

Assumption and instruction are then combined into one code value. The following table lists all possible values:

Code	Name	Description
1	New	The node is to be created based on the assumption that no related object exists in the back end. Values are to be taken from the request message.
2	Keep	The node is to be created based on the assumption that a related object exists. Values are to be taken from the request message. Values that were not transmitted are taken from the existing object.
3	Overwrite	The node is to be created based on the assumption that a related object exists. Values are to be taken from the related object.
4	All Matching or New	The values of the request message are used to find matching related objects

Code	Name	Description
		based on the assumption that there are 0..n related objects. A node is to be created for each object, using the values of the related objects. In case no related object is found, a node is created and the values are taken from the request message.
5	All Matching	The values of the request message are used to find matching related objects based on the assumption that 1..n related objects exist. A node is to be created for each object, using the values of the related objects.

Table 7-3: Values of the Creation Instruction Code

As you can see, the creation instruction code is defined from the perspective of the request message: "Overwrite" means that the values of the request message are replaced by the values from the back-end BO. "Keep" means that the values of the request message are kept (unless they were not transmitted, in which case the values from the back-end BO are taken).

A POT developer must take the creation instruction code into account when implementing the code slots of the *Create* phase:

First, the code must be validated and an exception has to be raised in case of unsupported values (see Listing 7-4 in section 7.1.3.3 as an example). Then, the implementation must follow the provided instruction, i.e. either fill the BPON with data from the back end or with data from the request message (see the implementation of the *Create* phase in section 7.1.3.5 for details and examples).

7.1.4 Helpers for Phase Implementation

The POT developer is provided with a number of helpers that can be used in the code slot implementations:

7.1.4.1 Phase Helper

The phase helper (`<MLB>IF_PHSH`) provides access to BPO and BPON data. The helper is passed to the code slots via importing parameter `I_PHASE_HELPER` and offers the methods described in the following table. Methods involving BPONs usually contain generic data types in their signatures, because every BPON has its own specific type. The descriptions below state the actual type for your convenience. For details on the used data types see section 7.4.

Method Name	Description
`GET_BPO`	Provides the complete data of the BPO including all BPONs (`<MLB>IF_IT_COMP=>TS_BPO_PUBLIC_TOTAL`).
`GET_BPON_BY_UUID`	Provides the complete data of a BPON (`<MLB>IF_IT_COMP=>TS_BPON_<BPON>_COMPLETE`) identified by its UUID.
`GET_BPONS_BY_REFERENCE_ID`	Provides a list of complete BPON data, identified by their node type and a reference ID. The returning parameter of this method is a table of references to

Method Name	Description
	generic type DATA. The actual type behind the references during runtime is the public complete type of a BPON (<MLB>IF_IT_COMP=>TS_BPON_<BPON>_PUBLIC_COMPLETE).
GET_BPONS_BY_CROSS_REFERENCE	Provides a list of complete BPON data identified by a cross-reference (<MLB>IF_IT_COMP=>TS_CROSS_REFERENCE). The returning parameter of this method is a table of references to the generic type DATA. The actual type behind the references during runtime is the public complete type of a BPON (<MLB>IF_IT_COMP=>TS_BPON_<BPON>_PUBLIC_COMPLETE). See section 6.5.6 for details on cross-references.
GET_BPON_TYPE_BY_CROSS_REF	Provides the node type of a BPON, identified by a cross-reference (<MLB>IF_IT_COMP=>TS_CROSS_REFERENCE).

Table 7-4: Methods of the Phase Helper

> **Note**
>
> Methods retrieving BPON data (GET_BPON...) only provide data if the access to the requested node(s) is allowed (according to the node dependencies defined in the Specification Wizard).

7.1.4.2 Phase Correlation Helper

The phase correlation helper (<MLB>IF_PHSCH) is passed to the code slot method for post-processing asynchronous confirmations. It provides methods to change the correlation key for a service call. The methods are explained in the following table:

Method Name	Description
GET_CORRELATION_KEY	Provides the current correlation key for the service call.
SET_NEW_CORRELATION_KEY	Determines and stores a new correlation key for the service call based on the data that is put into the method (I_CORRELATION_DATA). Usually, the data that should be passed to the method is the same as the data of the confirmation that is passed to the post-processing code slot (I_CONFIRMATION).

7-5: Methods of the Phase Correlation Helper

7.1.4.3 PIMPL Correlation Helper

The PIMPL correlation helper (<MLB>IF_PIMCH) is passed to the code slot method of the phase implementation for the *Execute* phase. It provides a method to set the correlation key for a service call and BPON instance. This method has to be called whenever asynchronous services are called from the *Execute* phase that expect a confirmation or further asynchronous information messages to complete the phase. In case of synchronous messages, the method can be called if further asynchronous information messages are expected to confirm the phase.

Method Name	Description
SET_CORRELATION_INFO	Determines and stores the correlation information for a certain BPON instance (I_BPON_UUID) and service call (I_BACKEND_SERVICE). The correlation key is determined based on the data that is passed to the method (I_DATA). I_DATA should be provided with the payload of the service call: In case of bulk messages, it should be provided with the payload of the single message (for each single message). In case of synchronous messages, the request and the response should be provided.

7-6: Methods of the PIMPL Correlation Helper

7.1.4.4 Log Helper

The log helper (<MLB>IF_LOG_HLP) provides convenience methods for handling internal and external representations of the GDT Log. It is provided via the attribute MR_LOG_HELPER in the phase implementation classes. The log helper is implemented locally by the FSL adapter class <MLB>CL_FSL_ADPT and offers methods similar to the log helper of the Financial Services Library (FSL). In addition, it can merge two logs into one.

Externally, the GDTs Log and Log_V1 are supported. Internally, the helper always uses the FSL representation of Log_V1, which is FSL_STR_GDT_LOG_V1.

The log helpers contain the following methods:

Method Name	Description
GET_LOG_BY_MESSAGES	Fills an internal log based on a table of BAPIRET2 messages, taking the provided error category (if available) and the result code into account. Importing parameter I_MESSAGES must be compatible to IF_FSL_LOG_FAULT_TYPES_CONS=>T_TAB_MSG_BAPI or IF_FSL_LOG_FAULT_TYPES_CONS=>T_TAB_MSG_BAPI_WD.
GET_LOG_BY_SY	Fills an internal log based on the provided system variable (SY), taking the provided error category (if available) and result code into account. The method extracts the message ID, message type, and message variables from the provided system variable. Hint: The system variable can easily be filled with the desired error message using the "MESSAGE…INTO" statement.
GET_MESSAGES_BY_LOG	Returns a list of messages, the severity code, and the result code based on the provided internal log. Exporting parameter E_MESSAGES must be compatible to IF_FSL_LOG_FAULT_TYPES_CONS=>T_TAB_MSG_BAPI or IF_FSL_LOG_FAULT_TYPES_CONS=>T_TAB_MSG_BAPI_WD.
CONVERT_GDT_LOG_TO_FSL_LOG CONVERT_FSL_LOG_TO_GDT_LOG	These methods convert a given log from external to internal representation and vice versa. The expected external representations are the SPROXY types generated for the GDTs Log and Log_V1. Importing parameter I_GDT_TYPE of method CONVERT_GDT_LOG_TO_FSL_LOG must be used to indicate whether

Method Name	Description
	the provided log is of type `Log` or `Log_V1`.
`FILL_MIN_FSL_LOG`	Fills all the fields of an internal log that are required as a minimum according to the provided result code. The minimally required field is the business document processing result code itself.
`MERGE_LOGS`	Merges two internal logs, taking the severest result code and severity code as new values.
`GET_LOG_BY_ECH_DATA`	Fills an internal log based on error information from ECH, taking the provided result code into account. It can be used in methods for ECH actions such as `FINISH` and `FAIL` for filling the log of the confirmation message based on data provided by ECH. Importing parameter `I_ECH_DATA` must be compatible to type `IF_FSL_LOG_FAULT_TYPES_CONS=>T_STR_ECH_DATA_STD`.

Table 7-7: Methods of the Log Helper

For details on the result code and the severity code see section 7.1.3.6.

7.1.4.5 Message Helper

The FSL message helper (`IF_FSL_MESSAGE_HELPER`) offers convenience methods for filling `BAPIRET2` messages and tables thereof from various input representations. It is provided via attribute `MR_MESSAGE_HELPER` in the phase implementation classes.

The message helper contains the following methods:

Method Name	Description
`FILL_BAPIRET2_BY_APP_LOG_MSG`	Fills a `BAPIRET2` message from the provided application log message.
`FILL_BAPIRETTAB_BY_APP_LOG_MSG`	Fills a table of `BAPIRET2` messages from the provided table of application log messages.
`FILL_BAPIRET2_BY_MESSAGE`	Fills a `BAPIRET2` message from the provided message details.
`FILL_BAPIRET2_BY_SY`	Fills a `BAPIRET2` message from the provided system variable.
`FILL_BAPIRET2_BY_T100`	Fills a `BAPIRET2` message from the provided exception that implements the `IF_T100_MESSAGE` interface, thereby taking the provided message type into account.
`FILL_BAPIRET2_BY_IF_MESSAGE`	This method fills a `BAPIRET2` message from the provided `IF_MESSAGE` message interface, thereby taking the provided message type into account. Note: If possible, use method `FILL_BAPIRET2_BY_T100` instead, as it preserves the content of the original message variables. The base exception class `CX_ROOT` also implements the `IF_MESSAGE` interface, but for exceptions method `FILL_BAPIRET2_BY_EXC` should be used.
`FILL_BAPIRET2_BY_EXC`	Fills a `BAPIRET2` message from the provided exception, thereby

Method Name	Description
	taking the provided message type into account.
FILL_BAPIRETTAB_BY_EXC	Fills a table of BAPIRET2 messages from the provided exception and all its predecessors stored in the PREVIOUS attribute, thereby taking the provided message type into account.
FILL_BAPIRET2_WD_BY_EXC	Fills a BAPIRET2 message from the provided exception, thereby taking the provided message type into account. BAPIRET2 with details is represented by data type IF_FSL_MESSAGE_HELPER=>T_STR_BAPIRET2_W_DETAILS.
FILL_BAPIRETTAB_WD_BY_EXC	Fills a table of BAPIRET2 with details messages from the provided exception and all its predecessors stored in the PREVIOUS attribute, thereby taking the provided message type into account. The table of BAPIRET2 with details is represented by data type IF_FSL_MESSAGE_HELPER=>T_TAB_BAPIRET2_W_DETAILS.

Table 7-8: Methods of the FSL Message Helper

BAPIRET2 and its table type BAPIRETTAB are widely used data types for error messages. They are important in cases where standard services are used during the phase implementation, but other interfaces such as BAPIs or RFCs instead. In addition, they are often employed as intermediate types for other error representations. For example, the log helper can fill a GDT Log from a table of BAPIRET2 messages.

The FILL_*_BY_EXC methods preserve the content of the original message variables in case that the source exception is an FSL exception (CX_FSL_) or any other exception that implements the IF_T100_MESSAGE interface.

7.1.4.6 Message Header Helper

The FSL message header helper (IF_FSL_MESSAGE_HEADER_HELPER) offers convenience methods for handling the GDT BusinessDocumentMessageHeader. It is provided via the attribute MR_MESSAGE_HEADER_HELPER in the phase implementation classes.

The message header helper contains the following methods:

Method Name	Description
VALIDATE_EXT	Validates the external (GDT) representation of a message header.
VALIDATE_INT	Validates the internal (FSL) representation of a message header.
CONVERT_EXT_TO_INT CONVERT_INT_TO_EXT	These methods convert a message header from the external to the internal representation and vice versa.
CONVERT_IN_OUT_INT	Converts a message header from the internal inbound to the internal outbound representation. That means that a new ID and UUID are drawn, and the ID and UUID from the origin message header are used as the REFERENCE_ID and the REFERENCE_UUID, respectively. The sender system of the origin message header becomes the receiver system of the new message header and vice versa.
CONVERT_IN_OUT_EXT	Provides the same functionality as CONVERT_IN_OUT_INT, but supports the external representation of a message header.
FILL_MIN_HEADER_INT	Fills all the fields of a message header that are required as a minimum (ID,

Method Name	Description
	UUID, CREATION_DATE_TIME, SENDER_BUSINESS_SYSTEM_ID). It is particularly useful when filling the request message of a back-end service.

Table 7-9: Methods of the FSL Message Header Helper

7.1.5 Create phase – Custom Implementation

The signature of the CREATE method of the phase implementation class (<MLB>CL_N_<BPON>_PIMPL) looks as follows:

Method Name	CREATE
Short Description	Creates BPONs
What to do	Enhance the BPON data provided as changing parameter C_BPONS with data retrieved from the back-end services according to the creation instruction code. Fill the provider ID for existing back-end business objects. Set the BPON status to Unchecked in case no errors occurred. In case of errors either: • Set the BPON status to Erroneous and fill the provider log with details about the error. This leads to overall status Erroneous of the POT instance. • Raise an exception of type <MLB>CX_IM in case of non-business related errors. The exception is propagated to the service implementation and thus leads to an unsuccessful service call (processing result code Failed) with an error message in the log.
Preconditions	None
Result	BPONs are created successfully or error details are given in the provider log

Parameter name	Parameter Type	Ref.	Data Type
I_PHASE_HELPER	Importing	X	<MLB>IF_PHSH
I_BACKEND_SERVICES	Importing		<MLB>IF_BS_PROV=>TY_BS_<BPON>_CREATE
C_BPONS	Changing		<MLB>IF_N_<BPON>_PIMPL=>TY_CREATE_REQUEST_REFS
	Exception	X	<MLB>CX_IM

An implementation of the CREATE method typically has an activity flow similar to the one shown in Figure 98:

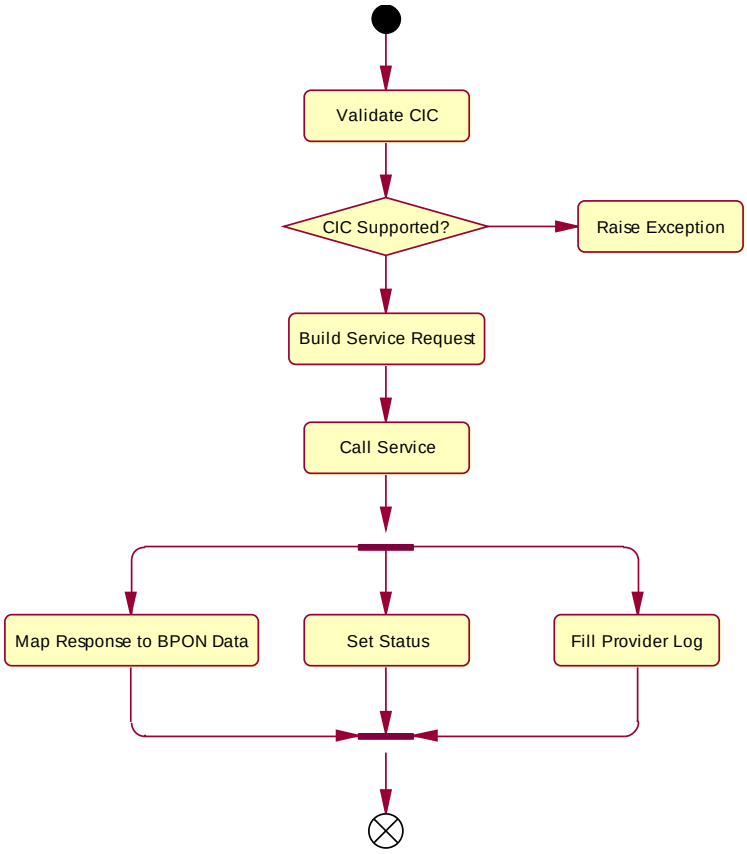


Figure 98: Activity Flow of *Create* Phase Code Slot

Access to data of other BPONs (e.g. via the cross-reference helper) is possible if the corresponding dependencies were modeled between the BPONs for the *Create* phase in the Specification Wizard.

Example 1: Create Phase

The following two listings provide an example for the implementation of the *Create* phase for the Flight BPON. The generated data declarations in Listing 7-5 are located outside of the code slot. The content of the code slot is shown in Listing 7-6 and explained below.

```
* data declaration for handling the BPON data
data: lr_bpon type /pl9/if_gxx_fb01_n_flgt_pimpl=>ty_ref_to_create_request.

* data declaration for backend service request and response types
data: ls_flqo_qry_request  type /pl9/wfccflight_by_elements_q2,
      ls_flqo_qry_response type /pl9/wfccflight_by_elements_rs.
```

Listing 7-5: Data Declarations of the *Create* Phase

```
data: lr_flight    type ref to /pl9/wfccflight_v1,
      lr_new_bpon  type /pl9/if_gxx_fb01_n_flgt_pimpl=>ty_ref_to_create_request,
      lt_new_bpons type /pl9/if_gxx_fb01_n_flgt_pimpl=>ty_create_request_refs,
      lt_messages  type bapirettab,
      lv_dummy     type string ##needed,
      lx_execution_error type ref to cx_fsl_execution_error.

* 1. validate creation instruction code
loop at c_bpons into lr_bpon.
  if ( /pl9/ce_gxx_fb01_crte_ic=>get_instance_by_value(
    lr_bpon->public_admin_data-creation_instruction_code ) <>
      /pl9/ce_gxx_fb01_crte_ic=>all_matching ).

    /pl9/cx_gxx_fb01_c_custom=>raise(
      textid = /pl9/cx_gxx_fb01_c_custom=>invalid_instruction_code
      code   = |{ lr_bpon->public_admin_data-creation_instruction_code-content }|
      type   = /pl9/ce_gxx_fb01_node_ty_c=>flgt->get_name( ) ).
  endif.
endloop.

* 2. access all BPONs
loop at c_bpons into lr_bpon.
* 3. fill message header
  ls_flqo_qry_request-message_header =
    mr_message_header_helper->fill_min_header_int( ).

* 4. map BPON data to request
  ls_flqo_qry_request-selection1-planned_flight_date =
    lr_bpon->process_control_constraints-planned_flight_date.
  move-corresponding lr_bpon->details-from-airport_id
    to ls_flqo_qry_request-selection1-from-airport_id. "#EC ENHOK
  move-corresponding lr_bpon->details-to-airport_id
    to ls_flqo_qry_request-selection1-to-airport_id. "#EC ENHOK
* 5. execute backend service
  try.
      ls_flqo_qry_response =
        i_backend_services-flqo_qry->execute( ls_flqo_qry_request ).
    catch cx_fsl_internal_error
          cx_fsl_error_conflict_mapping
          cx_fsl_error_conflict_format into lx_execution_error.

      /pl9/cx_gxx_fb01_c_custom=>raise(
        textid   = /pl9/cx_gxx_fb01_c_custom=>err_execution
        previous = lx_execution_error ).
  endtry.

* 6. process response
  if ( ce_fsl_proc_result_code=>s_get_instance(
    ls_flqo_qry_response-log-business_document_processing_r ) =
      ce_fsl_proc_result_code=>successful ).

    loop at ls_flqo_qry_response-flight reference into lr_flight.
      create data lr_new_bpon.
```

```
      lr_new_bpon->* = lr_bpon->*.

*     7. map response to BPON data
      move-corresponding lr_flight->from-location-airport_id
        to lr_new_bpon->details-from-airport_id. "#EC ENHOK
      move-corresponding lr_flight->to-location-airport_id
        to lr_new_bpon->details-to-airport_id. "#EC ENHOK
      move-corresponding lr_flight->price_amount
        to lr_new_bpon->details-price_amount. "#EC ENHOK
      lr_new_bpon->details-from-date_time = lr_flight->from-date_time.
      lr_new_bpon->details-to-date_time   = lr_flight->to-date_time.

      move-corresponding lr_flight->id
        to lr_new_bpon->public_admin_data-provider_id. "#EC ENHOK

*     8. set BPON status to Unchecked
      lr_new_bpon->public_admin_data-status_code =
        /pl9/ce_gxx_fb01_node_sc=>unchecked->get_int_value( ).

*     9. fill provider log
      lr_new_bpon->public_admin_data-log-provider_log = ls_flqo_qry_response-log.

      append lr_new_bpon to lt_new_bpons.
    endloop.
  else.
    create data lr_new_bpon.
    lr_new_bpon->* = lr_bpon->*.

*   10. fill provider log
    lr_new_bpon->public_admin_data-log-provider_log = ls_flqo_qry_response-log.

*   11. set BPON status to Erroneous
    lr_new_bpon->public_admin_data-status_code =
      /pl9/ce_gxx_fb01_node_sc=>erroneous->get_int_value( ).

    append lr_new_bpon to lt_new_bpons.
  endif.
endloop.

if ( lt_new_bpons is not initial ).
* 12. flights found or error occurred => transfer data
  c_bpons = lt_new_bpons.
else.
* 13. no flights found => fill log with error message
  loop at c_bpons into lr_bpon.
    lr_bpon->public_admin_data-status_code =
      /pl9/ce_gxx_fb01_node_sc=>erroneous->get_int_value( ).
    message e030(/pl9/gxx_fb01_ex_cus) into lv_dummy.
    append mr_message_helper->fill_bapiret2_by_sy( sy ) to lt_messages.
    lr_bpon->public_admin_data-log-provider_log =
      mr_log_helper->get_log_by_messages(
        i_messages                 = lt_messages
        i_bus_doc_proc_result_code = ce_fsl_proc_result_code=>failed ).
```

```
  endloop.
endif.
```

Listing 7-6: Code Slot of the *Create* Phase

1. The creation instruction code is validated using the corresponding enumeration class `<MLB>CE_CRTE_IC` and its static `GET_INSTANCE_BY_VALUE` method.

   In this example, only the value `ALL_MATCHING` is allowed. For all other values a custom exception is raised.
2. All provided BPONs are accessed in a loop. The following steps are executed for each BPON:
3. The request message of the back-end service is filled successively.

   At first, a minimal message header is created in the internal FSL format using the message header helper.
4. The request is filled with the provided search criteria.

   The planned flight date is taken from the process control constraints, the airports are taken from the details of the BPON itself.
5. The back-end service is executed.

   Exceptions are caught and attached to a custom exception, which is raised using its public static `RAISE` method. In this case, the execution of the code slot is aborted.
6. The response message of the back-end service is processed.

   The business document processing result code contained in the log indicates whether or not the call was successful. This check is done using enumeration class `CE_FSL_PROC_RESULT_CODE` from the FSL. In case the call was successful, all found flights are accessed in a loop.
7. A new BPON is created and first filled with the data provided in the request message. Afterwards, the data returned by the back end is moved to the BPON.

   This ensures that the BPON is filled according to the creation instruction code `ALL_MATCHING`. (Data is taken from the back-end business object, if available, and enhanced with data from the request.) In addition to filling the details of the BPON, the provider ID (contained in the public administrative data) is filled with the ID of the business object.
8. The status of the BPON is set to *Unchecked* using enumeration class `<MLB>CE_NODE_SC` for the node status code.

   Unchecked is the required value for all BPONs that are created for successfully found flights.
9. The provider log (contained in the public administrative data) is filled with the log returned by the back end.

   The log helper is used to convert the log from the FSL representation (Log or Log_V1) to the internal POT representation, which is always based on the FSL representation of Log_V1.

   Although the service call was successful, the log might still contain (information, success, etc.) messages that might be useful for a consumer. This is why the provider log is filled in this case, too.
10. The provider log is filled (see 9).In this example, it probably contains error messages.
11. The status of the BPON is set to *Erroneous*.

    This is the required status for BPONs that could not be executed successfully. In this example, it means that the service call returned an error.
12. If one or more flights could be found (or an error occurred), the table with the newly created BPONs is transferred to the changing parameter `C_BPONS`.
13. In case that no flights could be found at all, the status of all BPONs is set to *Erroneous* and an error message is written into the provider log informing the consumer about this situation.

    The log is filled using the common pattern involving the message helper and the log helper.

Example 2: Creation Instruction Code “Overwrite”

Listing 7-7 provides an example for the creation instruction code "Overwrite" in the code slot of the *Create* phase for the Traveler BPON. "Overwrite" means that the BPON is filled with data from the back end, thereby “overwriting” the original data of the BPON.


```
data: lx_execution_error type ref to cx_fsl_execution_error.

* 1. validate creation instruction code
loop at c_bpons into lr_bpon.
  if ( /pl9/ce_gxx_fb01_crte_ic=>get_instance_by_value(
      lr_bpon->public_admin_data-creation_instruction_code ) <>
        /pl9/ce_gxx_fb01_crte_ic=>overwrite ).

    /pl9/cx_gxx_fb01_c_custom=>raise(
      textid = /pl9/cx_gxx_fb01_c_custom=>invalid_instruction_code
      code   = |{ lr_bpon->public_admin_data-creation_instruction_code-content }|
      type   = /pl9/ce_gxx_fb01_node_ty_c=>trav->get_name( ) ).
  endif.
endloop.

* 2. access all BPONs
loop at c_bpons into lr_bpon.
* 3. fill message header
  ls_bpmo_read_request-message_header =
    mr_message_header_helper->fill_min_header_int( ).

* 4. map BPON data to request
  ls_bpmo_read_request-selection1-business_partner_id =
    lr_bpon->public_admin_data-provider_id.

* 5. execute backend service
  try.
      ls_bpmo_read_response =
        i_backend_services-bpmo_read->execute( ls_bpmo_read_request ).
    catch cx_fsl_internal_error
          cx_fsl_error_conflict_mapping
          cx_fsl_error_conflict_format into lx_execution_error.

      /pl9/cx_gxx_fb01_c_custom=>raise(
        textid   = /pl9/cx_gxx_fb01_c_custom=>err_execution
        previous = lx_execution_error ).
  endtry.

* 6. fill provider log
  lr_bpon->public_admin_data-log-provider_log = ls_bpmo_read_response-log.

* 7. process response
  if ( ce_fsl_proc_result_code=>s_get_instance(
      ls_bpmo_read_response-log-business_document_processing_r ) =
        ce_fsl_proc_result_code=>successful ).

*   8. fill BPON with back-end data
    move-corresponding ls_bpmo_read_response-business_partner-address
      to lr_bpon->details-address. "#EC ENHOK
    move-corresponding
      ls_bpmo_read_response-business_partner-communication_details-email_uri
      to lr_bpon->details-email_uri. "#EC ENHOK
    lr_bpon->details-name =
```

```
      ls_bpmo_read_response-business_partner-name.

*   9. set BPON status to Unchecked
    lr_bpon->public_admin_data-status_code =
      /pl9/ce_gxx_fb01_node_sc=>unchecked->get_int_value( ).
  else.
*   10. set BPON status to Erroneous
    lr_bpon->public_admin_data-status_code =
      /pl9/ce_gxx_fb01_node_sc=>erroneous->get_int_value( ).
  endif.
endloop.
```

Listing 7-7: Code Slot of the *Create* Phase with CIC Overwrite

The following aspects in Listing 7-7 above are noteworthy compared to Listing 7-6 described above:

4. As the ID of the requested business partner is already known, the provider ID of the BPON is filled already.
6. The provider log is filled independently of the processing result code. However, in this example it can be filled outside of the response evaluation.
8. According to the creation instruction code "Overwrite", the BPON data is completely taken from the data that has been retrieved from the back end.

7.1.6 Check Phase – Custom Implementation

The *Check* phase requires the following types of custom implementations, depending on the settings made in the Modeling Wizard:

- Code slot for service calls towards the back end
- Implementation of the Check BAdI
- BRFplus rules and implementation of BRFplus mapping

7.1.6.1 Calling Back-end Services in Check Phase

The signature of the `CHECK` method of the phase implementation class (`<MLB>CL_N_<BPON>_PIMPL`) looks as follows:

Method Name	CHECK
Short Description	Checks BPONs
What to do	Check the BPON data provided as importing parameter `I_BPONS` with the help of the back-end services handed in via `I_BACKEND_SERVICES`. If private details are switched on for this BPON type they might be changed using changing parameter `C_PRIVATE_DETAILS`. In case of errors either: • Set the BPON status to Erroneous and fill the process log with details about the error. This leads to overall status Erroneous of the POT instance. • Raise an exception of type `<MLB>CX_IM` in case of non-business related errors. The exception is propagated to the service implementation and thus leads to an unsuccessful service call (processing result code Failed) with an error message in the log.
Preconditions	Standard checks were executed successfully
Result	BPONs are checked successfully or errors are returned in the process log

Parameter name	Parameter Type	Ref.	Data Type
`I_BPONS`	Importing		`<MLB>IF_N_<BPON>_PIMPL=>TY_PUBLIC_COMPLETE_REFS`
`I_PHASE_HELPER`	Importing	X	`<MLB>IF_PHSH`
`I_BACKEND_SERVICES`	Importing		`<MLB>IF_BS_PROV=>TY_BS_<BPON>_CHECK`
`E_CHECKED_BPONS`	Exporting		`<MLB>IF_IT_COMP=>TT_BPON_AL_CHECK_RES`
`C_PRIVATE_DETAILS`	Changing		`<MLB>IF_IT_COMP=>TT_BPON_<BPON>_AL_PD_CHK_EX`
	Exception	X	`<MLB>CX_IM`

> **Note**
>
> This code slot is only available if back-end services were modeled for the *Check* phase of this BPON.

A code slot implementation in the *Check* phase typically has an activity flow similar to the one shown in Figure 99 below:

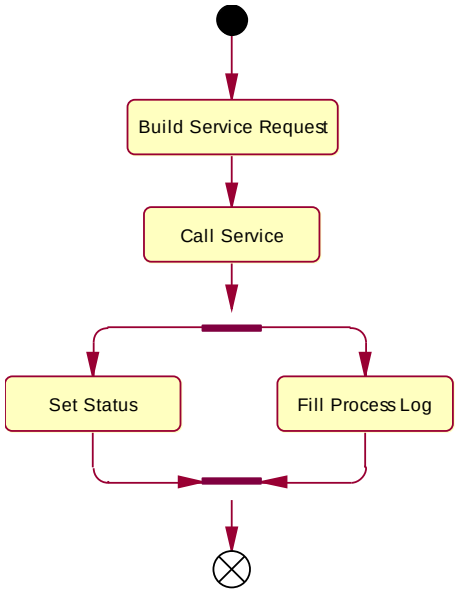


Figure 99: Activity Flow of *Check* Phase Code Slot

Access to the data of other BPONs is possible (e.g. via the cross-reference helper). In the *Check* phase, node dependencies are not taken into consideration.

Example 1: Calling Back-End Services

The following two listings show examples of how to call back-end services in a custom implementation for the *Check* phase of the Flight Booking BPON. The generated data declarations in Listing 7-8 are located outside of the code slot. The content of the code slot is shown in Listing 7-9 and explained below.

```
data: lr_bpon type /pl9/if_gxx_fb01_n_fbok_pimpl=>ty_ref_to_public_complete,
      lr_checked_bpon type ref to /pl9/if_gxx_fb01_it_comp=>ts_bpon_al_check_res.

* data declaration for backend service request and response types
data: ls_fbao_chk_request  type /pl9/wfccflight_booking_availa,
      ls_fbao_chk_response type /pl9/wfccflight_booking_avail5.
```

Listing 7-8: Data Declarations of the *Check* Phase

```
data: lr_flight type ref to /pl9/wfisbusiness_trip_fs_cros,
      lt_flight_data     type /pl9/if_gxx_fb01_phsh=>ty_data_tab,
      lr_flight_data     type line of /pl9/if_gxx_fb01_phsh=>ty_data_tab,
      lv_flight_id       type string,
      lv_dummy           type string ##needed,
      lt_messages        type bapirettab,
      lx_execution_error type ref to cx_fsl_execution_error.

field-symbols:
  <ls_flight_data> type /pl9/if_gxx_fb01_it_comp=>ts_bpon_flgt_public_complete.

* 1. access all BPONs
loop at i_bpons into lr_bpon.
* 2. get flight via cross-reference
  assert lines( lr_bpon->details-flight ) = 1. " exactly one flight per booking
  read table lr_bpon->details-flight index 1 reference into lr_flight.
  lt_flight_data = i_phase_helper->get_bpons_by_cross_reference( lr_flight->* ).
  assert lines( lt_flight_data ) = 1. " cross-reference is unique
  read table lt_flight_data into lr_flight_data index 1.
  assign lr_flight_data->* to <ls_flight_data>.
  assert sy-subrc = 0.

* 3. fill message header
  ls_fbao_chk_request-message_header =
    mr_message_header_helper->fill_min_header_int( ).

* 4. map BPON data to request
  move-corresponding <ls_flight_data>-public_admin_data-provider_id
    to ls_fbao_chk_request-flight_booking_availability-flight_id. "#EC ENHOK

* 5. execute backend service
  try.
      ls_fbao_chk_response =
        i_backend_services-fbao_chk->execute( ls_fbao_chk_request ).
    catch cx_fsl_internal_error
          cx_fsl_error_conflict_mapping
          cx_fsl_error_conflict_format into lx_execution_error.

      /pl9/cx_gxx_fb01_c_custom=>raise(
        textid   = /pl9/cx_gxx_fb01_c_custom=>err_execution
        previous = lx_execution_error ).
  endtry.

* 6. for each checked BPON instance add an entry to table e_checked_bpons
  append initial line to e_checked_bpons reference into lr_checked_bpon.
  lr_checked_bpon->public_admin_data-uuid =
    lr_bpon->public_admin_data-uuid.

* 7. fill process log
  lr_checked_bpon->public_admin_data-log-process_log = ls_fbao_chk_response-log.

* 8. process response
  if ( ce_fsl_proc_result_code=>s_get_instance(
```

```
    ls_fbao_chk_response-log-business_document_processing_r ) =
      ce_fsl_proc_result_code=>successful ).

*   9. check if seats are available and set BPON status accordingly
    if ( ls_fbao_chk_response-flight_booking_availability-
      available_seat_number_value > 0 ).
      lr_checked_bpon->public_admin_data-status_code =
        /pl9/ce_gxx_fb01_node_sc=>checked->get_int_value( ).
    else.
      lr_checked_bpon->public_admin_data-status_code =
        /pl9/ce_gxx_fb01_node_sc=>erroneous->get_int_value( ).

*     10. no seats available => fill process log with error message
      concatenate
        ls_fbao_chk_response-flight_booking_availability-flight_id-airline_id
        ls_fbao_chk_response-flight_booking_availability-flight_id-connection_id
        ls_fbao_chk_response-flight_booking_availability-flight_id-
          planned_flight_date
        into lv_flight_id separated by space.
      message e010(/pl9/gxx_fb01_ex_cus) with lv_flight_id into lv_dummy.
      append mr_message_helper->fill_bapiret2_by_sy( sy ) to lt_messages.
      lr_checked_bpon->public_admin_data-log-process_log =
        mr_log_helper->get_log_by_messages(
          i_messages = lt_messages
          i_bus_doc_proc_result_code = ce_fsl_proc_result_code=>failed ).
    endif.
  else.
*   11. set BPON status to Erroneous
    lr_checked_bpon->public_admin_data-status_code =
      /pl9/ce_gxx_fb01_node_sc=>erroneous->get_int_value( ).
  endif.
endloop.
```

Listing 7-9: Code Slot of the *Check* Phase

1. All provided BPONs are accessed in a loop. The following steps are executed for each BPON:
2. The Flight BPON for which this flight booking will be created is retrieved via cross-reference (`lr_bpon->details-flight`) using the phase helper.
3. The request message of the back-end service is filled successively.

   At first, a minimal message header is created in the internal FSL format using the message header helper.
4. The request is filled with the provider ID of the flight.
5. The back-end service is executed.

   Exceptions are caught and attached to a custom exception, which is raised using its public static `RAISE` method. In this case, the execution of the code slot is aborted.
6. An entry is added to the table `E_CHECKED_BPONS` for the current BPON instance containing the UUID of the BPON.
7. The process log (contained in the public administrative data) is filled with the log returned by the back end.

   This is done independently of the processing result code, because also in the success case the log could contain (information, success, ...) messages that might be useful for the consumer. The log helper is used to convert the log from the FSL representation (Log or Log_V1) to the internal POT representation, which is always based on the FSL representation of Log_V1.

8. The response message of the back-end service is processed. The business document processing result code contained in the log indicates whether the call was successful or not. This check is done using enumeration class `CE_FSL_PROC_RESULT_CODE` from the FSL.
9. If seats are available on the given flight, the BPON status is set to *Checked* using enumeration class `<MLB>CE_NODE_SC` for the node status code. Otherwise it is set to *Erroneous*.
10. In the latter case, an error message is written to the process log informing the consumer about this situation. The log is filled using the common pattern involving the message helper and the log helper.

> **Note**
>
> In the *Check* phase only the process log is available, because this phase validates whether the *process* can be executed with the given BPON data. Therefore, errors are process-relevant and are therefore written into the process log. As a consequence, messages from the back end are also written into the process log here.

Example 2: Accessing Private Details

Listing 7-10 shows how private details can be accessed during the *Check* phase:

```
loop at i_bpons into lr_bpon.
  ...
  read table c_private_details
    with key public_admin_data-uuid = lr_bpon->public_admin_data-uuid
    into lr_private_details.
  assert sy-subrc = 0.
  ...
endloop.
```

Listing 7-10: Private Details in the *Check* Phase

7.1.6.2 Implementing the Check BAdI

When BAdI checks were selected in the Modeling Wizard, a POT developer can create BAdI implementations in transaction `SE19` and implement the corresponding BAdI methods. As the BAdI is typed for multiple use, there can be several active implementations. To access other nodes and further data, the phase helper is provided to the BAdI methods.

Depending on the check settings, the following methods are available:

- `CHECK_BPO` (to check BPO data)
- `CHECK_BPON_<BPON>` (to check BPON data)

Figure 100 provides an overview of the check BAdI:

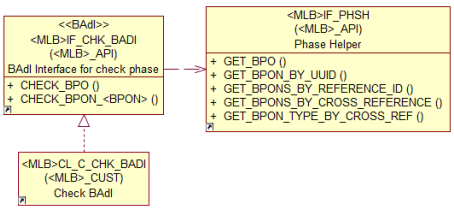


Figure 100: Check BAdI

The signatures of the BAdI methods are described below:

Method Name	CHECK_BPO
Short Description	Performs checks on BPO
What to do	Check the BPO data. Use additional data provided by the phase helper, if required. Indicate whether or not the check was successful, and provide error details in C_CHECK_RESULT. (As there can be multiple active BAdI implementations, the result is provided as changing parameter.) Raise an implementation exception in case of technical errors during the check. In this case, the execution is aborted and the Check service returns the processing result code "Failed".
Preconditions	Standard checks were executed successfully.
Result	BPO data is checked and the result is stored in C_CHECK_RESULT.

Parameter name	Parameter Type	Ref.	Data Type
I_DATA	Importing		<MLB>IF_IT_COMP=>TS_BPO_PUBLIC_TOTAL
I_PHASE_HELPER	Importing	X	<MLB>IF_PHSH
C_CHECK_RESULT	Changing		<MLB>IF_CHK_BW=>TY_BADI_CHECK_RESULT
	Exception	X	<MLB>CX_IM

Method Name	CHECK_BPON_<BPON>
Short Description	Performs checks on BPON <BPON>
What to do	Check the BPON data. Use additional data provided by the phase helper, if required. Indicate whether or not the check was successful, and provide error details in C_CHECK_RESULT. (As there can be multiple active BAdI implementations, the result is provided as changing parameter.) If available, the private details of the BPON provided in C_PRIVATE_DETAILS can be changed. Raise an implementation exception in case of technical errors during the check. In this case, the execution is aborted and the Check service returns the processing result code "Failed".
Preconditions	Standard checks were executed successfully.
Result	BPON data is checked and the result is stored in C_CHECK_RESULT.

Parameter name	Parameter Type	Ref.	Data Type
I_DATA	Importing		<MLB>IF_IT_COMP=> TS_BPON_<BPON>_PUBLIC_COMPLETE
I_PHASE_HELPER	Importing	X	<MLB>IF_PHSH
C_CHECK_RESULT	Changing		<MLB>IF_CHK_BW=>TY_BADI_CHECK_RESULT
C_PRIVATE_DETAILS	Changing		<MLB>IF_IT_COMP=> TS_BPON_<BPON>_PRIVATE_DETAILS
	Exception	X	<MLB>CX_IM

7.1.6.3 Implementing BRFplus Checks

When BRFplus checks were selected in the Modeling Wizard, a POT developer must create the corresponding check functions in BRFplus and implement a code slot to map the node data to the structure that is handed over to BRFplus. If the BRFplus check is performed on BPON level, the phase helper provides access to the BPO and other BPONs.

Figure 101 provides an overview of the wrapper class for the BRFplus check and the related mapping class:

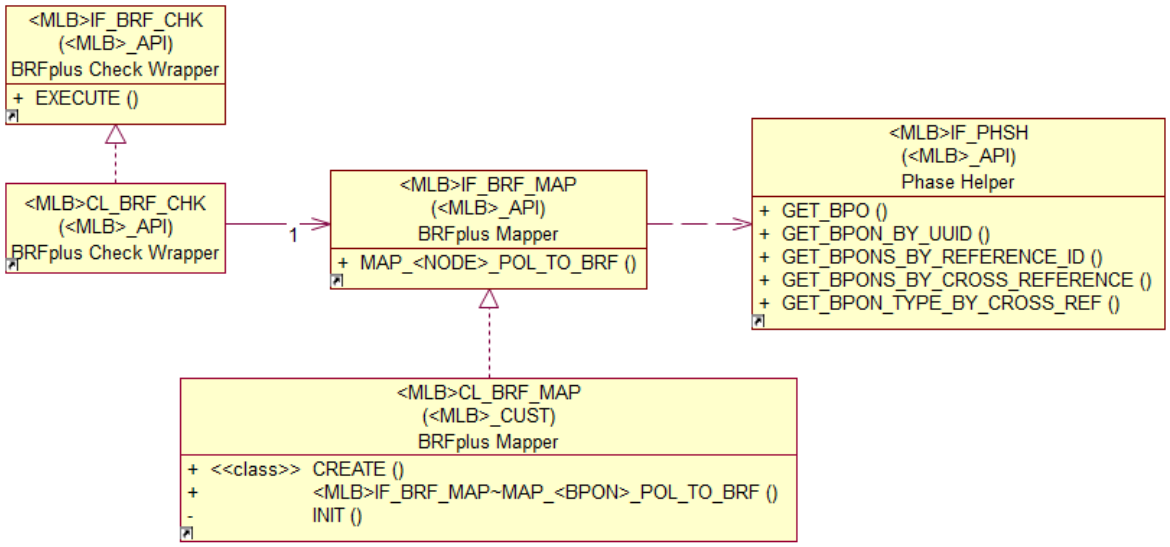


Figure 101: BRFplus Check Wrapper and Mapper

The mapping methods containing the code slots to be implemented are described below:

Method Name	INIT		
Short Description	Initializes instance		
What to do	Instantiate additional attributes.		
Preconditions	Standard checks were executed successfully.		
Result	All attributes are instantiated.		
Parameter name	**Parameter Type**	**Ref.**	**Data Type**

Method Name	MAP_<BPO>_POL_TO_BRF		
Short Description	Mapping for <BPO>		
What to do	Map BPO data to BRFplus data.		
Preconditions	Standard checks were executed successfully.		
Result	BPO data for BRFplus		
Parameter name	**Parameter Type**	**Ref.**	**Data Type**
I_DATA	Importing		<MLB>IF_IT_COMP=>TS_BPO_PUBLIC_TOTAL
R_DATA	Returning		<MLB>N_<BPO>_BRF_R

Method Name	MAP_<BPON>_POL_TO_BRF		
Short Description	Mapping for <BPON>		
What to do	Map BPON data to BRFplus data. (Other nodes can be accessed via the phase helper.)		
Preconditions	Standard checks were executed successfully.		
Result	BPON data for BRFplus		
Parameter name	**Parameter Type**	**Ref.**	**Data Type**
I_DATA	Importing		<MLB>IF_IT_COMP=> TS_BPON_<BPON>_PUBLIC_COMPLETE
I_PHASE_HELPER	Importing	X	<MLB>IF_PHSH
R_DATA	Returning		<MLB>N_<BPON>_BRF_R

In this context, BRFplus can only be used to raise error messages in case a check fails. It is not possible to derive new values etc.

Figure 102 shows the BRFplus function CHECK_BPO as the entry point to BRFplus:

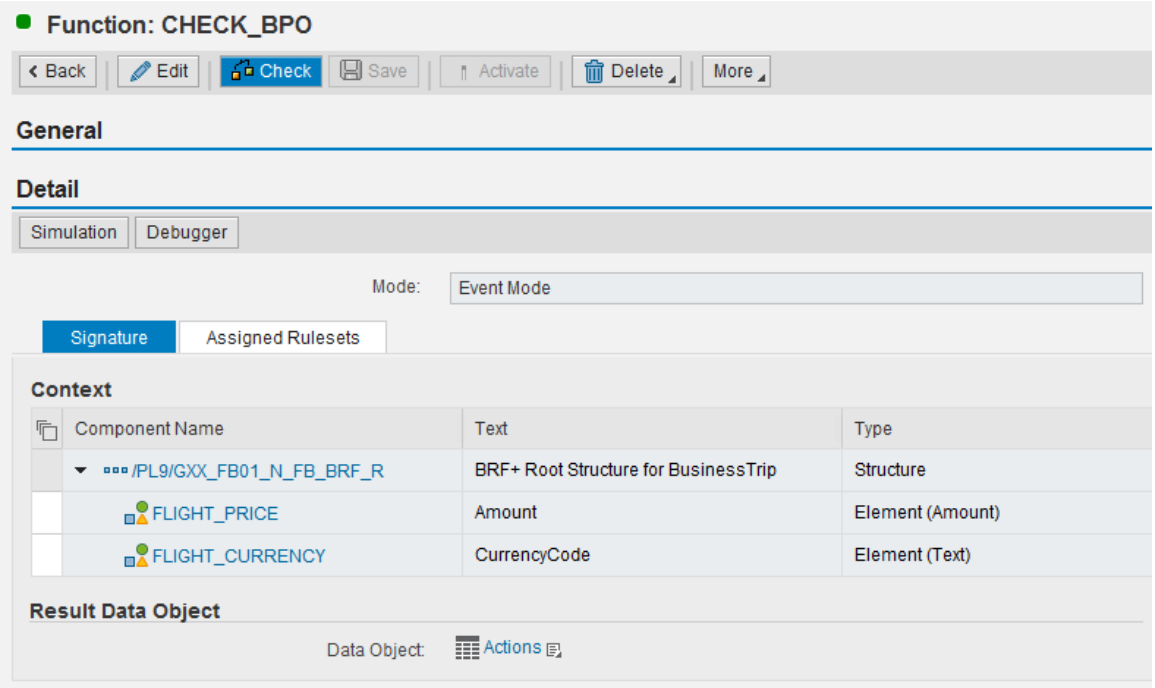


Figure 102: BRFplus Function CHECK_BPO

The data that has been prepared in the mapping code slot is available as value range from the context (see Figure 103):

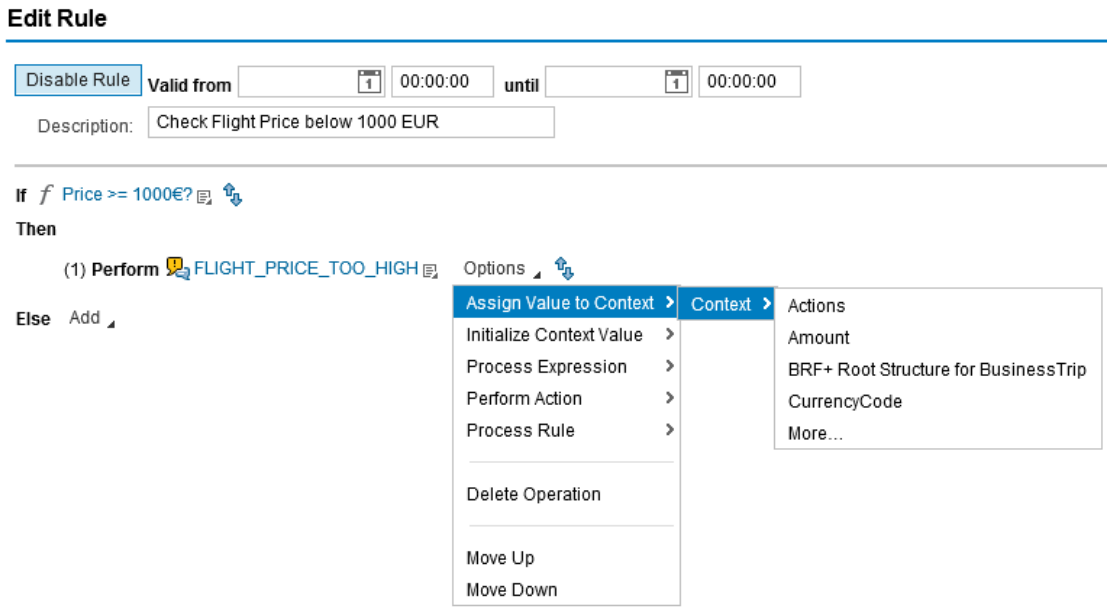


Figure 103: Accessing Input Data in BRFplus

Messages can be raised using the BRFplus action “Log Message” (see Figure 104). They are written into the process log.

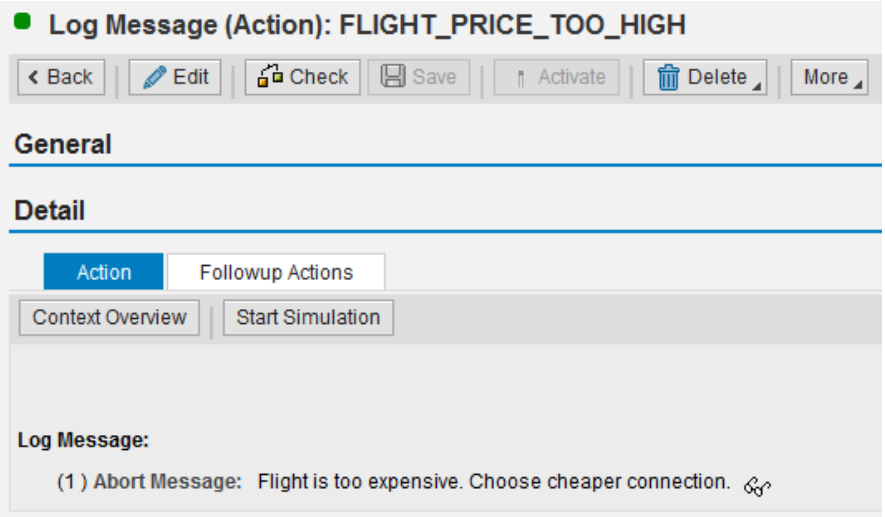


Figure 104: Raising an Error Message in BRFplus

7.1.7 Execute Phase – Custom Implementation

The *Execute* phase requires the following types of custom implementations, depending on the type of services selected in the Modeling Wizard:

- Code slot for service calls towards the back end
- Code slot for asynchronous confirmations from the back end
- Code slot for custom correlation
- Code slot for splitting non-standard bulk services (bulk splitter)

7.1.7.1 Calling Back-End Services in the Execute Phase

The `EXECUTE` method of the phase implementation class (`<MLB>CL_N_<BPON>_PIMPL`) has the following signature:

Method Name	EXECUTE
Short Description	Executes BPONs
What to do	Execute the back-end services provided via `I_BACKEND_SERVICES` for the BPONS provided in `I_BPONS`. (Additional data can be accessed via the phase helper.) For synchronous services: • Add an entry to `R_EXECUTED_BPONS`. • Set provider ID, status (*Confirmed* or *Failed*), and back-end modification state For asynchronous services: • Call the PIMPL Correlation Helper for each executed service call and BPON instance with the data of the service request If private details are activated for this BPON type, they can be changed using changing parameter `C_PRIVATE_DETAILS`. In case of errors either: • Set the BPON status to *Failed* and fill the provider log with details about the error. This leads to overall status *Failed* of the POT instance. • In case of non-business-related errors, raise an exception of type `<MLB>CX_IM`. Unlike in the *Create* and *Check* phase, this does not lead to an unsuccessful service call (processing result code "Failed"), because the *Execute* phase is processed asynchronously. Instead, all instances of this BPON type are set to *Failed*. Private Details are not changed.
Preconditions	BPONs have status *Checked*.
Result	BPONs are executed successfully or errors are returned in the log (synchronous services). Execution of asynchronous services has been started. In case you are waiting for further incoming messages, you can set the status to *In Execution*.

Parameter name	Parameter Type	Ref.	Data Type
`I_BPONS`	Importing		`<MLB>IF_N_<BPON>_PIMPL=>TY_PUBLIC_COMPLETE_REFS`
`I_PHASE_HELPER`	Importing	X	`<MLB>IF_PHSH`
`I_BACKEND_SERVICES`	Importing		`<MLB>IF_BS_PROV=>TY_BS_<BPON>_EXECUTE`
`I_PIMPL_CORRELATION_HELPER`	Importing	X	`<MLB>IF_PIMCH`
`R_EXECUTED_BPONS`	Returning		`<MLB>IF_IT_COMP=>TT_BPON_<BPON>_AL_EXE_RQ`
`C_PRIVATE_DETAILS`	Changing		`<MLB>IF_IT_COMP=>TT_BPON_<BPON>_AL_PD_CHK_EX`
	Exception	X	`<MLB>CX_IM`

In the *Execute* phase, the activity flows are different for synchronous and asynchronous back-end services (see next sections).

7.1.7.1.1 Calling Synchronous Services in the Execute Phase

For synchronous back-end services, a code slot implementation in the *Execute* phase typically has an activity flow similar to the one shown in Figure 105 below:

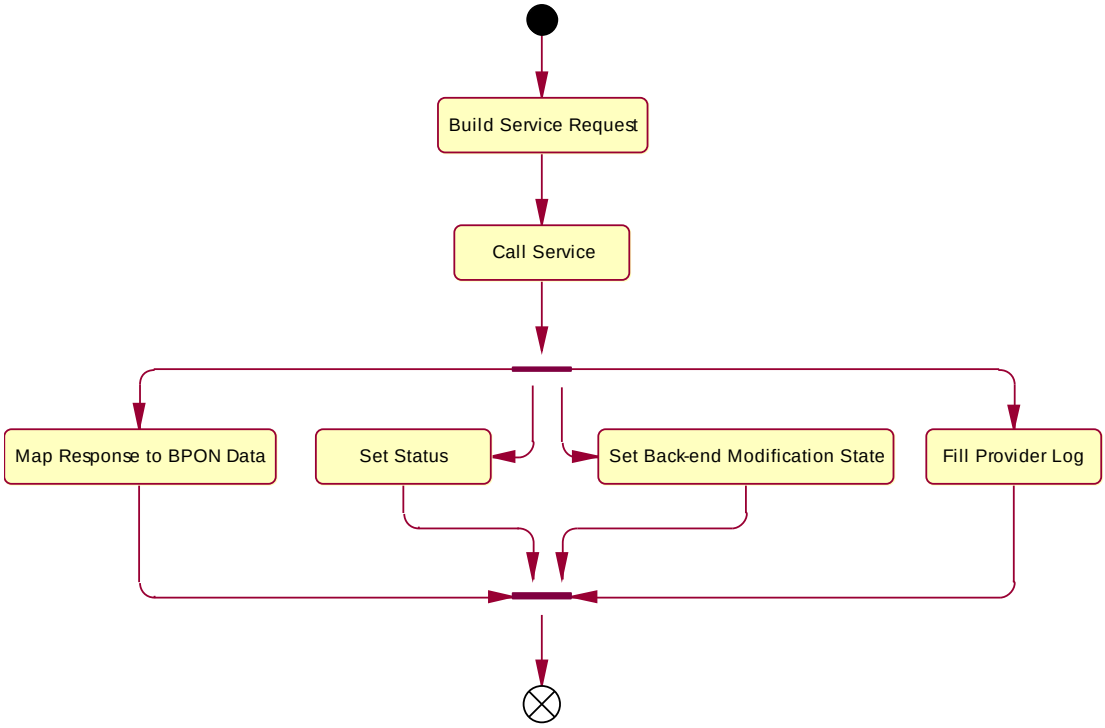


Figure 105: Activity Flow of the *Execute* Phase Code Slot (Synchronous Service)

Access to data of other BPONs is possible (e.g. via the cross-reference helper) if dependencies were modeled between the BPONs for this phase in the Specification Wizard.

Example: Calling Synchronous Service

Listing 7-11 provides an example of how to implement the *Execute* phase of the Flight Booking BPON:

```
data:
  lr_traveller_ref   type ref to /pl9/wfisbusiness_trip_fs_cros,
  lt_traveller_data  type /pl9/if_gxx_fb01_phsh=>ty_data_tab,
  lr_traveller_data  type line of /pl9/if_gxx_fb01_phsh=>ty_data_tab,
  lr_flight_ref      type ref to /pl9/wfisbusiness_trip_fs_cros,
  lt_flight_data     type /pl9/if_gxx_fb01_phsh=>ty_data_tab,
  lr_flight_data     type line of /pl9/if_gxx_fb01_phsh=>ty_data_tab,
  ls_bpo             type /pl9/if_gxx_fb01_it_comp=>ts_bpo_public_total,
  lx_execution_error type ref to cx_fsl_execution_error.

field-symbols:
   <ls_traveller> type /pl9/if_gxx_fb01_it_comp=>ts_bpon_trav_public_complete,
   <ls_flight>    type /pl9/if_gxx_fb01_it_comp=>ts_bpon_flgt_public_complete.

* 1. access all BPONs
loop at i_bpons into lr_bpon.
* 2. get traveller via cross-reference
  assert lines( lr_bpon->details-business_traveller ) = 1. "exactly one traveller
  read table lr_bpon->details-business_traveller
    index 1 reference into lr_traveller_ref.
  lt_traveller_data =
    i_phase_helper->get_bpons_by_cross_reference( lr_traveller_ref->* ).
  assert lines( lt_traveller_data ) = 1. "cross-reference is unique
  read table lt_traveller_data index 1 into lr_traveller_data.
  assign lr_traveller_data->* to <ls_traveller>.
  assert sy-subrc = 0.

* 3. get flight via cross-reference
  assert lines( lr_bpon->details-flight ) = 1. "exactly one flight per booking
  read table lr_bpon->details-flight
    index 1 reference into lr_flight_ref.
  lt_flight_data =
    i_phase_helper->get_bpons_by_cross_reference( lr_flight_ref->* ).
  assert lines( lt_flight_data ) = 1. "cross-reference is unique
  read table lt_flight_data into lr_flight_data index 1.
  assign lr_flight_data->* to <ls_flight>.
  assert sy-subrc = 0.

* 4. fill message header
  ls_fbmo_crt_request-message_header =
    mr_message_header_helper->fill_min_header_int( ).

* 5. map BPON data to request
  ls_bpo = i_phase_helper->get_bpo( ).

  ls_fbmo_crt_request-flight_booking-business_partner_id =
    <ls_traveller>-public_admin_data-provider_id.
  move-corresponding <ls_flight>-public_admin_data-provider_id
    to ls_fbmo_crt_request-flight_booking-flight_id. "#EC ENHOK
  move-corresponding ls_bpo-public_admin_data-bpca-bpca_type_code
    to ls_fbmo_crt_request-flight_booking-business_process_chain_assignm-
    bpca_type_code.
  ls_fbmo_crt_request-flight_booking-business_process_chain_assignm-
```

```
    bpca_uuid-content = ls_bpo-public_admin_data-bpca-bpca_uuid.

* 6. execute backend service
  try.
      ls_fbmo_crt_response =
        i_backend_services-fbmo_crt->execute( ls_fbmo_crt_request ).
    catch cx_fsl_internal_error
          cx_fsl_error_conflict_mapping
          cx_fsl_error_conflict_format into lx_execution_error.

      /pl9/cx_gxx_fb01_c_custom=>raise(
        textid   = /pl9/cx_gxx_fb01_c_custom=>err_execution
        previous = lx_execution_error ).
  endtry.

* 7. for each synchronously executed BPON add an entry to table e_executed_bpons
* if required ( not required if you just want to leave the status in
* 'In Execution'
  append initial line to r_executed_bpons reference into lr_executed_bpon.

* 8. map response to BPON data
  lr_executed_bpon->public_admin_data-uuid =
    lr_bpon->public_admin_data-uuid.
  lr_executed_bpon->public_admin_data-provider_id =
    ls_fbmo_crt_response-flight_booking-id.

* 9. fill provider log
  lr_executed_bpon->public_admin_data-log-provider_log = ls_fbmo_crt_response-log.

* 10. evaluate response
  if ( ce_fsl_proc_result_code=>s_get_instance(
    ls_fbmo_crt_response-log-business_document_processing_r ) =
      ce_fsl_proc_result_code=>successful ).

*   11. set BPON status to Confirmed and backend to Modified
    lr_executed_bpon->public_admin_data-status_code =
      /pl9/ce_gxx_fb01_node_sc=>confirmed->get_int_value( ).
    lr_executed_bpon->backend_modification_state =
      /pl9/ce_gxx_fb01_mod_state=>modified->get_value( ).
  else.
*   12. set BPON status to Failed and backend to Unmodified
    lr_executed_bpon->public_admin_data-status_code =
      /pl9/ce_gxx_fb01_node_sc=>failed->get_int_value( ).
    lr_executed_bpon->backend_modification_state =
      /pl9/ce_gxx_fb01_mod_state=>unmodified->get_value( ).
  endif.
endloop.
```

Listing 7-11: Code Slot of the *Execute* Phase with Synchronous Service

1. All provided BPONs are accessed in a loop. The following steps are executed for each BPON:
2. The traveler for this flight booking is retrieved via cross-reference using the phase helper.
3. The flight for this flight booking is also retrieved via cross-reference.

4. The request message of the back-end service is filled successively.

   At first, a minimal message header is created in the internal FSL format using the message header helper.
5. The BPON data is mapped to the request.
6. The back-end service is executed.

   Exceptions are caught and attached to a custom exception, which is raised using its public static `RAISE` method. In this case, the execution of the code slot is aborted.
7. An entry for this instance is added to table `E_EXECUTED_BPONS`...
8. ...and filled with the response data.
9. The provider log (contained in the public administrative data) is filled with the log returned by the back end.

   The log helper is used to convert the log from the FSL representation (Log or Log_V1) to the internal POT representation, which is always based on the FSL representation of Log_V1.
10. The response message of the back-end service is processed.

    The business document processing result code contained in the log indicates whether or not the call was successful. This check is done using enumeration class `CE_FSL_PROC_RESULT_CODE` from the FSL.
11. In case the service call was successful, the status of the BPON is set to *Confirmed*; the back-end modification state is set to *Modified*, because the back end has been altered.
12. Otherwise, the BPON status is set to *Failed* and the back-end modification state is set to *Unmodified*, because nothing happened on the back-end side.

If the response of the synchronous call is not sufficient to set the status of the BPON to *Completed* and you expect a further information message to complete the BPON you have to set the status of the BPON instance to *In Execution* and call the PIMPL Correlation helper to set the correlation data. Please note that you can only set the BPON status to "In Execution" of you call the PIMPL Correlation Helper for a synchronous service.

```
<…>
* 6. execute backend service
  try.
      ls_fbmo_crt_response =
        i_backend_services-fbmo_crt->execute( ls_fbmo_crt_request ).


*   Call the PIMPL Correlation Helper
      i_pimpl_correlation_helper->set_correlation_info
        ( i_data = value #( request  = ref #( ls_fbmo_crt_request )
                            response = ref #( ls_fbmo_crt_response ) )
  i_backend_service =   /pl9/cx_gxx_fb01_be_srv=>fbmo_crt
  i_bpon_uuid       =  ls_bpon-public_admin_data-uuid ).

    catch cx_fsl_internal_error
          cx_fsl_error_conflict_mapping
          cx_fsl_error_conflict_format into lx_execution_error.

      /pl9/cx_gxx_fb01_c_custom=>raise(
        textid   = /pl9/cx_gxx_fb01_c_custom=>err_execution
        previous = lx_execution_error ).
  endtry.
<…>
```

Listing 7-12: Call of PIMPL Correlation Helper

7.1.7.1.2 Calling Asynchronous Services in the Execute Phase

For asynchronous back-end services, a code slot implementation in the *Execute* phase typically has an activity flow similar to the one shown in Figure 105 Figure 106 below:

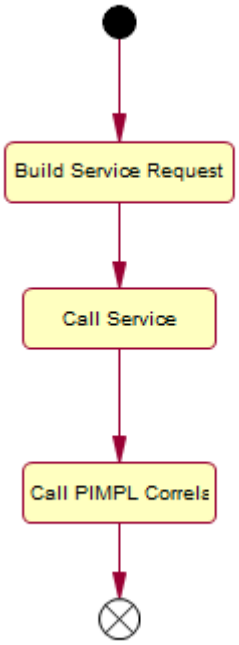


Figure 106: Activity Flow of the *Execute* Phase Code Slot (Asynchronous Service)

In the asynchronous case, only the BPON UUID is stored together with the correlation key which is determined by the call of SET_CORRELATION_INFO of the PIMPL Correlation Helper. The BPON status and the back-end modification state are set in the code slot for the asynchronous confirmation (see below).

Example: Calling Asynchronous Service

Listing 7-13 shows an example of how to implement the *Execute* phase of the Shuttle Booking BPON:

```
data:
  lr_traveller_ref   type ref to /pl9/wfisbusiness_trip_fs_cros,
  lt_traveller_data  type /pl9/if_gxx_fb01_phsh=>ty_data_tab,
  lr_traveller_data  type line of /pl9/if_gxx_fb01_phsh=>ty_data_tab,
  lr_flight_ref      type ref to /pl9/wfisbusiness_trip_fs_cros,
  lt_flight_data     type /pl9/if_gxx_fb01_phsh=>ty_data_tab,
  lr_flight_data     type line of /pl9/if_gxx_fb01_phsh=>ty_data_tab,
  ls_bpo             type /pl9/if_gxx_fb01_it_comp=>ts_bpo_public_total,
  lx_execution_error type ref to cx_fsl_execution_error.

field-symbols:
  <ls_traveller> type /pl9/if_gxx_fb01_it_comp=>ts_bpon_trav_public_complete,
  <ls_flight>    type /pl9/if_gxx_fb01_it_comp=>ts_bpon_flgt_public_complete.

* 1. access all BPONs
loop at i_bpons into lr_bpon.
* 2. get traveller via cross-reference
  assert lines( lr_bpon->details-business_traveller ) = 1. "exactly one traveller
  read table lr_bpon->details-business_traveller
    index 1 reference into lr_traveller_ref.
  lt_traveller_data =
    i_phase_helper->get_bpons_by_cross_reference( lr_traveller_ref->* ).
  assert lines( lt_traveller_data ) = 1. "cross-reference is unique
  read table lt_traveller_data into lr_traveller_data index 1.
  assign lr_traveller_data->* to <ls_traveller>.
  assert sy-subrc = 0.

* 3. get flight via cross-reference
  assert lines( lr_bpon->details-flight ) = 1. "exactly one selected flight
  read table lr_bpon->details-flight index 1 reference into lr_flight_ref.
  lt_flight_data =
    i_phase_helper->get_bpons_by_cross_reference( lr_flight_ref->* ).
  assert lines( lt_flight_data ) = 1. "cross-reference is unique
  read table lt_flight_data into lr_flight_data index 1.
  assign lr_flight_data->* to <ls_flight>.
  assert sy-subrc = 0.

* 4. fill message header
  ls_sbo_crt_request-message_header =
    mr_message_header_helper->fill_min_header_int( ).

* 5. map BPON data to request
  ls_sbo_crt_request-shuttle_booking-business_partner_id =
    <ls_traveller>-public_admin_data-provider_id.

  if ( lr_bpon->details-from_indicator = abap_true ).
    move-corresponding <ls_flight>-details-from-airport_id
      to ls_sbo_crt_request-shuttle_booking-airport_id. "#EC ENHOK
    ls_sbo_crt_request-shuttle_booking-date =
      <ls_flight>-details-from-date_time(8).
  else.
    move-corresponding <ls_flight>-details-to-airport_id
      to ls_sbo_crt_request-shuttle_booking-airport_id. "#EC ENHOK
```

```
    ls_sbo_crt_request-shuttle_booking-date =
      <ls_flight>-details-to-date_time(8).
  endif.

  ls_bpo = i_phase_helper->get_bpo( ).
  move-corresponding ls_bpo-public_admin_data-bpca-bpca_type_code
    to ls_sbo_crt_request-shuttle_booking-business_process_chain_assignm-
    bpca_type_code.
  ls_sbo_crt_request-shuttle_booking-business_process_chain_assignm-
    bpca_uuid-content = ls_bpo-public_admin_data-bpca-bpca_uuid.

* 6. execute back-end service
  try.
      i_backend_services-sbo_crt->execute( ls_sbo_crt_request ).

* 7. for each asynchronously executed BPON call the PIMPL Correlation Helper with
* the BPON UUID, the backend service and the payload of the request du determine
* and store the correlation ke
    i_pimpl_correlation_helper->set_correlation_info
      ( i_data            = #value( request   = ref #( ls_sbo_crt_request )
        i_bpon_uuid       = lr_bpon->public_admin_data-uuid
        i_backend_service = /pl9/ce_gxx_fb01_be_srv=>sbo_crt ) ).

    catch cx_fsl_internal_error
          cx_fsl_error_conflict_mapping
          cx_fsl_error_conflict_format into lx_execution_error.

      /pl9/cx_gxx_fb01_c_custom=>raise(
        textid   = /pl9/cx_gxx_fb01_c_custom=>err_execution
        previous = lx_execution_error ).
  endtry.
endloop.
```

Listing 7-13: Code Slot of the *Execute* Phase with Asynchronous Service

The following aspects are noteworthy in the asynchronous case, compared to the synchronous case explained before:

6. As the back-end service is asynchronous, no response is received here.
7. Call the PIMPL Correlation Helper for each executed service call and BPON instance with the data of the service request.

   When the asynchronous confirmation is processed, this information is used to map the received message to the corresponding BPON instance (see section 7.1.7.2 below).

7.1.7.2 Post-Processing of Asynchronous Confirmations

The method for post-processing asynchronous confirmations is only available in the phase implementation class if asynchronous services were modeled for this BPON type in the Modeling Wizard. Its signature looks as follows:

Method Name	PPC_<IF>_<OP>
Short Description	Post-processing: Operation <OP> of interface <IF>
What to do	Set provider ID, status (*Confirmed* or *Failed*), and back-end modification state of the BPON in `R_EXECUTED_BPON`, according to the confirmation `I_CONFIRMATION` that has been retrieved from the back end. If private details are activated for this BPON type, they can be changed using the changing parameter `C_PRIVATE_DETAILS`. In case of errors either: • Set the BPON status to *Failed* and fill the provider log with details about the error. This leads to the overall status *Failed* of the POT instance. • In case of non-business-related errors, raise an exception of type `<MLB>CX_IM`. Unlike in the *Create* and *Check* phase, this does not lead to an unsuccessful service call (processing result code "Failed"), because the *Execute* phase is processed asynchronously. Instead, this BPON instance remains in status *Running* and the service call is propagated to the Error and Conflict Handler (ECH). Private details are not changed in this case • In case you expect further incoming messages which complete the BPONs set the status of the BPONs to "In Execution"
Preconditions	None
Result	BPONs are executed successfully or errors are returned in the log or BPONs are left in In Execution.

Parameter name	Parameter Type	Ref.	Data Type
`I_CONFIRMATION`	Importing	X	`DATA`
`I_PHASE_HELPER`	Importing	X	`<MLB>IF_PHSH`
`I_CORRELATION_HELPER`	Importing	X	`<MLB>IF_PHSCH`
`I_BPON`	Importing		`<MLB>IF_N_<BPON>_PIMPL=>` `TY_REF_TO_PUBLIC_COMPLETE`
`R_EXECUTED_BPON`	Returning		`<MLB>IF_IT_COMP=>TS_BPON_<BPON>_AL_E_Q_NU`
`C_PRIVATE_DETAILS`	Changing		`<MLB>IF_IT_COMP=>` `TT_BPON_<BPON>_AL_PD_CHK_EX`
	Exception	X	`<MLB>CX_IM`

The activity flow for post-processing asynchronous confirmations typically looks as in Figure 107 below:

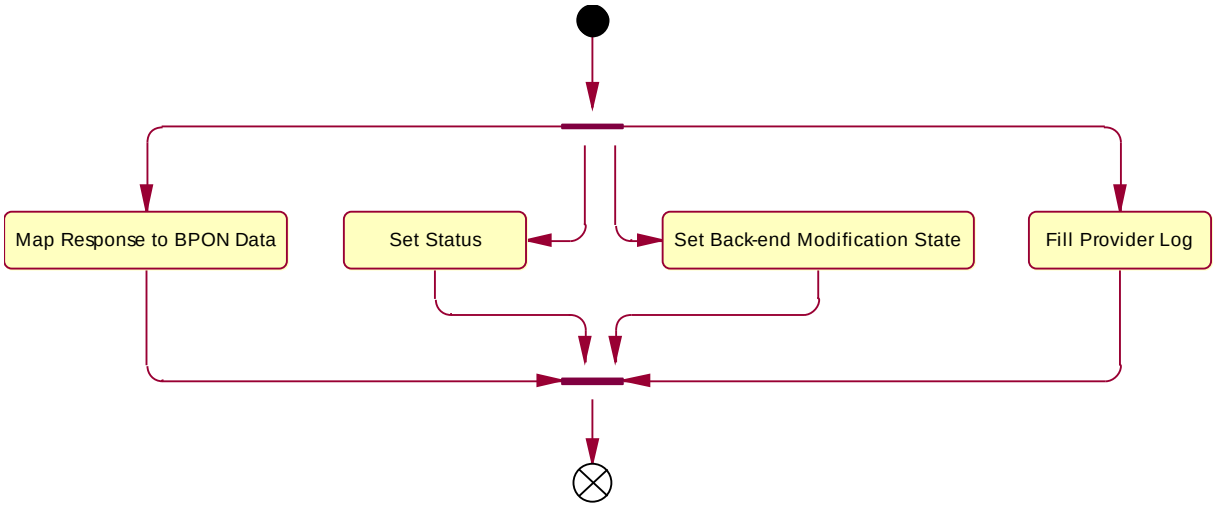


Figure 107: Activity Flow for the Post-Processing of Asynchronous Messages

Example: Post-Processing of Asynchronous Confirmations

Listing 7-14 shows an example of how to implement the post-processing of an asynchronous confirmation in the Shuttle Booking BPON:

```
* 1. fill provider log
e_executed_bpon-public_admin_data-log-provider_log = <ls_confirmation>-log.

* 2. evaluate response
if ( ce_fsl_proc_result_code=>s_get_instance(
  <ls_confirmation>-log-business_document_processing_r ) =
    ce_fsl_proc_result_code=>successful ).

* 3. set BPON status to Confirmed
*    and back-end modification state to Modified
  r_executed_bpon-public_admin_data-provider_id =
    <ls_confirmation>-shuttle_booking-id.
  r_executed_bpon-public_admin_data-status_code =
    /pl9/ce_gxx_fb01_node_sc=>confirmed->get_int_value( ).
  r_executed_bpon-backend_modification_state    =
    /pl9/ce_gxx_fb01_mod_state=>modified->get_value( ).
else.
* 4. set BPON status to Failed
*    and back-end modification state to Unmodified
  r_executed_bpon-public_admin_data-status_code =
    /pl9/ce_gxx_fb01_node_sc=>failed->get_int_value( ).
  r_executed_bpon-backend_modification_state =
    /pl9/ce_gxx_fb01_mod_state=>unmodified->get_value( ).
endif.
```

Listing 7-14: Code Slot of an Asynchronous Confirmation

1. The response log is mapped to the provider log using the log helper.
2. The processing result code of the response is evaluated.

3. In case the service was executed successfully, the BPON status is set to *Confirmed* and the back-end modification state to *Modified*.
4. Otherwise, the status is set to *Failed* and the modification state to *Unmodified*.

With the help of the correlation provider you have the option to change the correlation key in case you are expecting further asynchronous messages to come which correlate with this message but with a different key.

```
<...>
i_correlation_helper->set_new_correlation_key( <ls_confirmation> ).
<..>
```

Listing 7-15: Call of Correlation Helper

More information can be found in chapter Error! Reference source not found..

7.1.7.3 Custom Correlation of Asynchronous Services

If you call an asynchronous service in the *Execute* phase, the incoming confirmation and probably further information messages have to be matched (correlated) with the original request. By default, this is done with the MessageHeaderUUID of the request and the MessageHeaderReferenceUUID of the confirmation. If you want to use a different key for correlation, you need to activate custom correlation for the respective back-end services that are assigned to a BPON using the *Custom Correlation* checkbox in step *Define Process Object Nodes* of the Modeling Wizard. Using custom correlation also allows you to wait for other incoming messages that correlate with the original request.

For each BPON that has services with custom correlation assigned, the Builder generates one interface, class and DDIC structure. The generated DDIC structure contains a dummy field. You need to adapt this structure by inserting the fields that are needed for creating the correlation. The interface `<MLB>IF_<BPON>_CCOR` contains one method for each back-end service operation that is flagged with “Custom Correlation”:

Method Name	GET_CC_<IF>_<OP>
Short Description	Provides Custom Correlation for Service Operation <IF>_<OP>
What to do	The method receives the message data of the corresponding service call (`I_PAYLOAD`). In case of synchronous services, it receives the data of the request and the response. The implementer has to map this data to the correlation structure that the method returns (`R_CORRELATION`).
Preconditions	None
Result	The correlation structure is filled.

Parameter name	Parameter Type	Ref.	Data Type
`I_PAYLOAD`	Importing	X	`<PAYLOAD of Request>`
`R_CORRELATION`	Returning	X	`<MLB>_<BPON>_COR`

With the correlation data that you provide in the method `GET_CC_<IF>_<OP>`, it is assured that the incoming confirmation message will match the outgoing call. Sometimes, however, the confirmation message of an asynchronous call (or the response of a synchronous call) is not sufficient to complete a BPON instance. It might be the case that only an information message received later (e.g. an information about the status change of a back-end object) can confirm the BPON instance. I could furthermore be the case that this information message does not match the correlation data of the original request. You therefore have the possibility to change the correlation data in the PPC method of an incoming asynchronous call. You can find an example for this in chapter 7.1.7.2.

7.1.7.4 Correlation of Non-Standard Bulk Services

If a POT uses asynchronous back-end services that do not comply with the SAP standards for enterprise services (that is, if the back-end service operation signatures do not contain a `BusinessDocumentMessageHeader` or `BasicBusinessDocumentMessageHeader`), the correlation information that maps an asynchronous response to a BPON must be explicitly specified using custom correlation (see 7.1.7.3). Before the custom correlation can be called, the bulk message has to be split into single messages using the so-called bulk splitter.

When an asynchronous confirmation is processed, the correlation key of the message is used together with the UUID of the BPON to map the message to the corresponding BPON instance.

The bulk splitter is provided for each non-standard bulk service that is used.

Figure 108 provides an example:

<MLB>CL_BSP_<IF>_<OP>

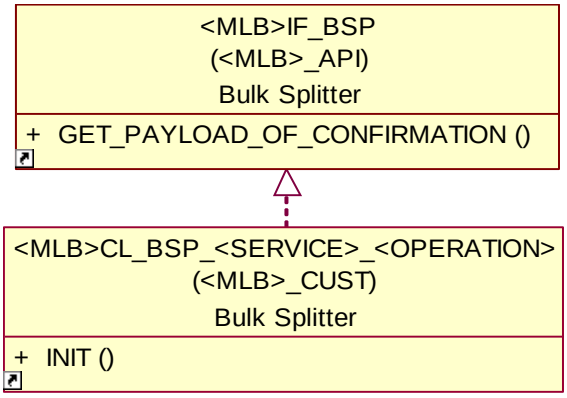


Figure 108: Bulk Splitter

Method Name	INIT		
Short Description	Initializes Instance		
What to do	Initialize additional attributes		
Preconditions	None		
Result	Instance is initialized		
Parameter name	**Parameter Type**	**Ref.**	**Data Type**

Method Name	GET_PAYLOAD_OF_CONFIRMATION
Short Description	Splits a Non-Standard Bulk message into single messages
What to do	The method receives the complete bulk message and has to return a table of single messages.
Preconditions	None
Result	Table of single messages is returned.

Parameter name	Parameter Type	Ref.	Data Type
I_CONFIRMATION	Importing	X	<PAYLOAD of Request>
R_PAYLOAD	Returning	X	TY_PAYLOAD

The resulting payload is then forwarded to the code slot of the custom correlation method and to the code slot of the phase implementation class that is responsible for post-processing asynchronous confirmations.

Example: Extracting the Payload from Asynchronous Confirmations

Listing 7-16 shows a sample implementation of the GET_PAYLOAD_OF_CONFIRMATION method for a mass service:

```
data: lr_shuttle_booking
        type ref to /pl9/wfccshuttle_booking_crea1,
      lr_payload
        type ref to /pl9/if_gxx_fb01_bsp=>ty_payload_line.

loop at <ls_confirmation>-shuttle_booking reference into lr_shuttle_booking.
  append lr_shuttle_booking to r_payload
endloop.
```

Listing 7-16: Code Slot of a Bulk Splitter

All entries in the confirmation message are accessed in a loop. For every shuttle booking contained in the confirmation an entry is added to the result list containing the the payload. The payload is a reference to generic type DATA, because the actual type is of course different for every service operation.

7.2 Implementing Archiving Extensions

The generated archiving functionality can be extended in dedicated places, as described in the following subsections.

7.2.1 Additional Selection Criteria for Archiving Write Program

By default, data can be selected by BPO status and timeframe in the archiving write program. This standard selection can be extended to allow for a selection by provider ID and/or reference ID. To implement this additional selection criteria, a POT developer must implement two code slots in the write program: one for defining the additional selection fields and one for mapping the input from these fields to the selection data.

Example: Define Additional Parameter in Archiving Write Program

The following example defines an additional parameter for the provider ID of the Traveler in the first code slot and maps it to the selection structure in the second one.

```
* first code slot
parameters: p_trv_id type /pl9/if_gxx_fb01_it_comp=>ts_bpon_trav_provider_id.

* second code slot
append p_trv_id to ls_selection-bpon_trav-provider_ids.
```

Listing 7-17: Additional Parameters for Archiving

7.2.2 Additional Archivability Checks

According to the standard implementation of a POT, the archivability of a BPO instance is determined by its status and the rules for residence time as defined in ILM Customizing. A POT developer can further restrict this check in a code slot that is provided in the custom archivability check class (`<MLB>CL_A_CHK_C`). However, the result of the standard check cannot be revised but can only be further restricted.

The signature of the corresponding method `IS_ARCHIVABLE` looks as follows:

Method Name	IS_ARCHIVABLE
Short Description	Checks whether a PO instance is archivable
What to do	Implement additional archivability checks and store the result in parameter `R_IS_ARCHIVABLE`. You can add a message for explanation. BPO data is accessible via the archivability check helper, which is provided as importing parameter `I_CHECK_HELPER`. Note: If possible, use method `GET_BASIC_BPO_DATA` of the check helper, because it offers better performance than `GET_BPO_DATA`.
Preconditions	None
Result	The result of the archivability check is returned.

Parameter name	Parameter Type	Ref.	Data Type
I_CHECK_HELPER	Importing	X	<MLB>IF_A_CHK_HLP
R_IS_ARCHIVABLE	Returning		<MLB>IF_A_CHK_HLP=>TY_CHECK_RESULT
	Exception	X	<MLB>CX_ARC

7.2.3 Data Mapping for Archive Search

By default, the archive search UI allows users to search for BPO instances by administrative data - such as UUID, status, and the last change date. A POT developer can add fields for provider IDs and reference IDs to the standard search criteria by implementing the code slots in the custom search class (<MLB>CL_A_SEA_C).

The corresponding methods are described below:

Method Name	MAP_SELECTION_TO_SEARCH
Short Description	Maps Feeder Data to Find Operation
What to do	Map additional UI fields for provider IDs and/or reference IDs from FPM representation to the custom search structure.
Preconditions	Fields are defined
Result	The provided IDs are returned in POT-specific format

Parameter name	Parameter Type	Ref.	Data Type
I_FEEDER_SEARCH_DATA	Importing		FPMGB_T_SEARCH_CRITERIA
R_CUSTOM_SEARCH	Returning		<MLB>IF_A_SEA_C=>TY_CUSTOM_SEARCH

Method Name	MAP_SEARCH_RESULT_TO_FEEDER
Short Description	Maps Search Result to Feeder Data
What to do	Map the result of the archive search provided in I_SEARCH_RESULT to the feeder data in order to extend the list of results with the provided entries.
Preconditions	None
Result	The provided search result is returned in FPM-specific format

Parameter name	Parameter Type	Ref.	Data Type
I_SEARCH_RESULT	Importing		<MLB>IF_A_MDL=>TY_MODEL_DATA
C_FEEDER_DATA	Changing		<MLB>A_RSLS

7.2.4 Condition Fields for ILM Rules

In the implementation of the ILM BAdI `BADI_ILM_OT_FLD`, additional condition fields and their values for an ILM rule can be defined using code slots. The defined fields and values are used in the ILM rules of the ILM object to determine the residence time and the retention period in the archive for the data of an ILM object. The code slots are used to determine these times separately for specific data (for example, different retention periods for private and corporate customers).

To do so, the BAdI implementation calls the class for custom field value determination (`<MLB>CL_A_FVD_C`), which contains the following methods:

Method Name	REGISTER_FIELDS
Short Description	Registers Condition Fields for Archiving
What to do	Define additional condition fields in parameter `CT_FIELDS`
Preconditions	None
Result	Additional parameters are defined

Parameter name	Parameter Type	Ref.	Data Type
`CT_FIELDS`	Changing		`<MLB>IF_A_TYPES=>TY_CONDITION_FIELDS`

Method Name	GET_FIELD_VALUES
Short Description	Provides Field Values
What to do	Add the value for the requested field provided via `I_FIELD_NAME` to table `CT_FIELD_VALUES`. BPO data can be accessed via the archiving data helper, which is provided as attribute `MR_DATA_HELPER` of the class.
Preconditions	The importing parameters are filled. The requested field has been registered in method `REGISTER_FIELDS`.
Result	The value for the requested field is contained in the value table.

Parameter name	Parameter Type	Ref.	Data Type
`I_BPO_UUID`	Importing		`<MLB>IF_IT_BASE=>TS_BPO_BPON_KEY`
`I_FIELD_NAME`	Importing		`<MLB>IF_A_TYPES=>TY_CONDITION_FIELD_NAME`
`CT_FIELD_VALUES`	Changing		`<MLB>IF_A_TYPES=>` `TY_CONDITION_FIELD_VALUES`
	Exception	X	`CX_IRM_CALLBACK_INT_ERROR`

7.3 Implementing Additional Authorization Checks

If additional authorization checks were selected during specification, these checks have to be implemented in the class for custom authorization checks (`<MLB>CL_ATH_CHK_C`). You can restrict the authorizations (for example, based on the payload of a service request) or specifically prohibit the execution of certain service operations.

The signature of the corresponding method looks as follows:

Method Name	IS_AUTHORIZED
Short Description	Performs Additional Authorization Checks
What to do	Implement additional authorization checks according to your needs
Preconditions	None
Result	The result of the authorization check is returned

Parameter name	Parameter Type	Ref.	Data Type
I_ACTIVITY	Importing	X	<MLB>CE_ATH_ACT
I_NODE_TYPE	Importing	X	<MLB>CE_NODE_TY_C
I_OPERATION	Importing	X	<MLB>CE_OP_PURP
I_REQUEST_DATA	Importing		DATA
I_BPO_TOTAL	Importing		<MLB>IF_IT_COMP=>TS_BPO_PUBLIC_TOTAL
R_IS_AUTHORIZED	Returning		ABAP_BOOL

7.4 Internal Types

Internal types are the internal representation of generated SPROXY types. They can be divided into common internal types (shared across different POTs with the same namespace) and POT-specific internal types:

The common internal types are clones of the generated SPROXY types for the message types of the back-end services. These types are ABAP Dictionary (DDIC) types and include controller tables to support extended XML handling.

The POT-specific internal types are cloned from the generated SPROXY types of the message types of the POT services. These types are also DDIC-based, but they do not contain controller tables. In addition to the POT-specific DDIC types, there also are some POT-specific interface types available. These are grouped into several interfaces (see Table 7-10 below), which are located in the general package. However, the internal types are not 1:1 copies of the SPROXY types, but have been adapted and enriched where useful.

Interface	Description
<MLB>IF_IT_BASE	Base data types used as part of the other types
<MLB>IF_IT_COMP	Complete types (see Table 7-11 and Table 7-12 below)

Table 7-10: Interfaces for Internal Types

For each type used in POL implementations, two type definitions are generated: A flat type (scalar or structured with prefix 'TS_') and a corresponding table type of this flat structure (prefix 'TT_'). For a POT developer, <MLB>IF_IT_COMP is the most important interface, because it contains the majority of the types that are used in the code slots.

The following two tables explain commonly used types of the BPO and the BPONs from the complete types interface. The structured types (TS_) listed below all have corresponding tabular types (TT_), which are not mentioned in the tables.

Internal Type	Description
TS_BPO_DETAILS	Details of the BPO
TS_BPO_PRIVATE_DETAILS	Private details of the BPO
TS_BPO_PUB_ADM	Public administrative data of the BPO containing its UUID, provider ID, reference ID, type code, status code and log
TS_BPO_PRIV_ADM	Private administrative data of the BPO containing system administrative data, internal status code and version data
TS_BPO_COMPLETE	Complete data of the BPO containing public and private administrative data, details, and a table with the UUIDs of its BPONs
TS_BPO_PUBLIC_COMPLETE	Public complete data of the BPO containing public administrative data, details, and a table with the UUIDs of its BPONs (no private administrative data)
TS_BPO_TOTAL	Total type of the BPO containing public and private administrative data, details, and the complete types of its BPONs
TS_BPO_PUBLIC_TOTAL	Public total type of the BPO containing public administrative data, details, and the complete types of its BPONs (no private administrative data)

Table 7-11: Important BPO Types

Internal Type	Description
TS_BPON_<BPON>_DETAILS	Details of <BPON>
TS_BPON_<BPON>_PRIVATE_DETAILS	Private details of <BPON>
TS_BPON_<BPON>_PUB_ADM	Public administrative data of <BPON> containing its UUID, provider ID, reference ID, type code, status code and log
TS_BPON_<BPON>_PRIV_ADM	Private administrative data of <BPON> containing the BPO UUID, back-end modification state, system administrative data and version data
TS_BPON_<BPON>_COMPLETE	Complete type of <BPON> containing public and private administrative data and details
TS_BPON_<BPON>_PUBLIC_COMPLETE	Public complete type of <BPON> containing public administrative data and details (no private administrative data)
TS_CROSS_REFERENCE	Representation of a cross-reference containing UUID, reference ID and the type code. Either the UUID is filled or the reference ID together with the type code.

Table 7-12: Important BPON Types

7.5 Usage of WSDL-based backend services

From POL 2.0 on the usage of WSDL-based backend services in a POT is supported. However the procedure differs a little bit from the usage of ESR defined services. In the following it is described step by step how a WSDL-based service can be used within a POT.

1. Upload WSDL into one of your namespaces as External Definition
    a. Create a new External Definition in your namespace in ESR

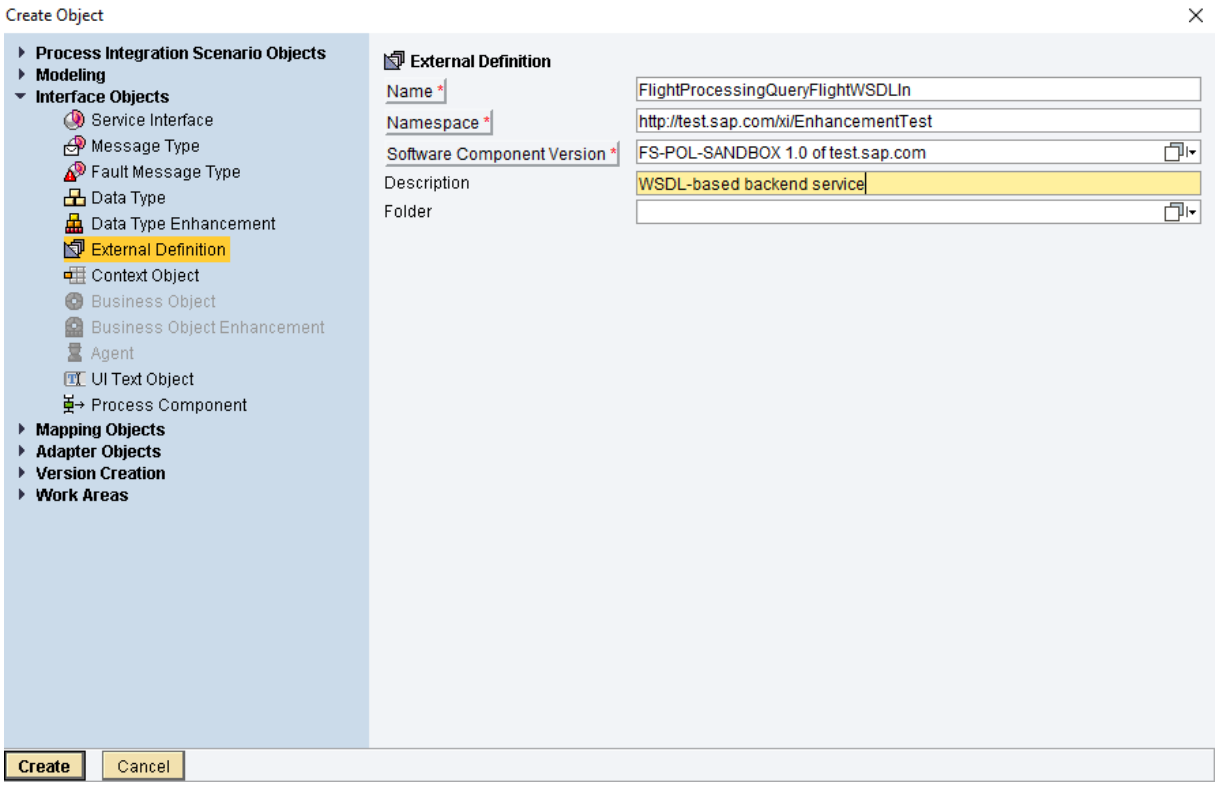


Figure 109: Create External Definition in ESR

    b. Create a new External Definition in your namespace in ESR and save

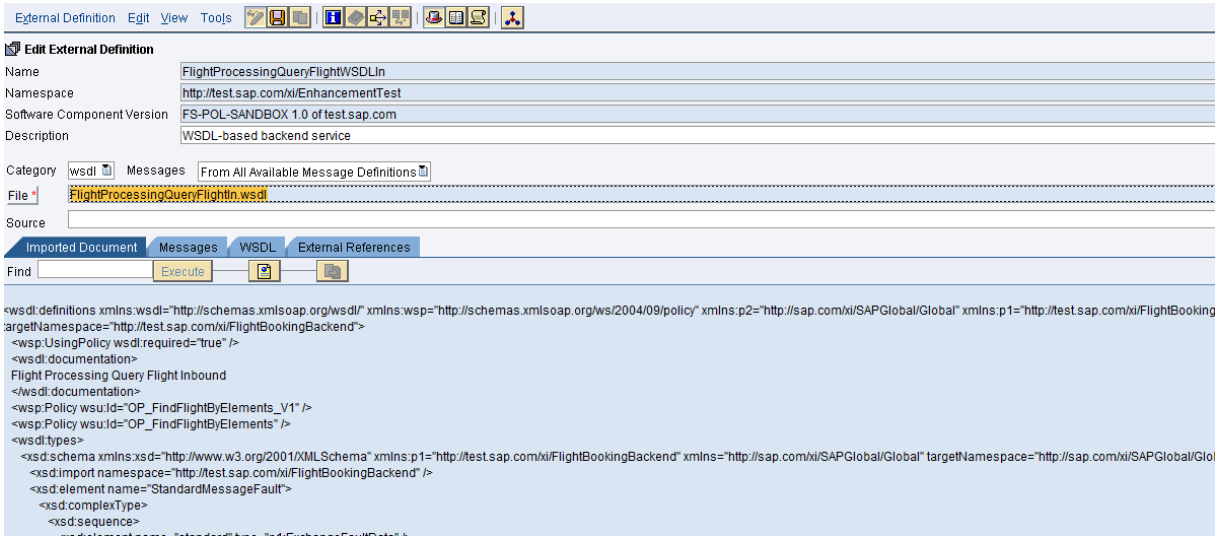


Figure 110: Select WSDL-file

c. Create a new Service Interface in your namespace and add the operations from the WSDL-based service which you would like to use. Select the corresponding External Message elements. Save and activate your change list.

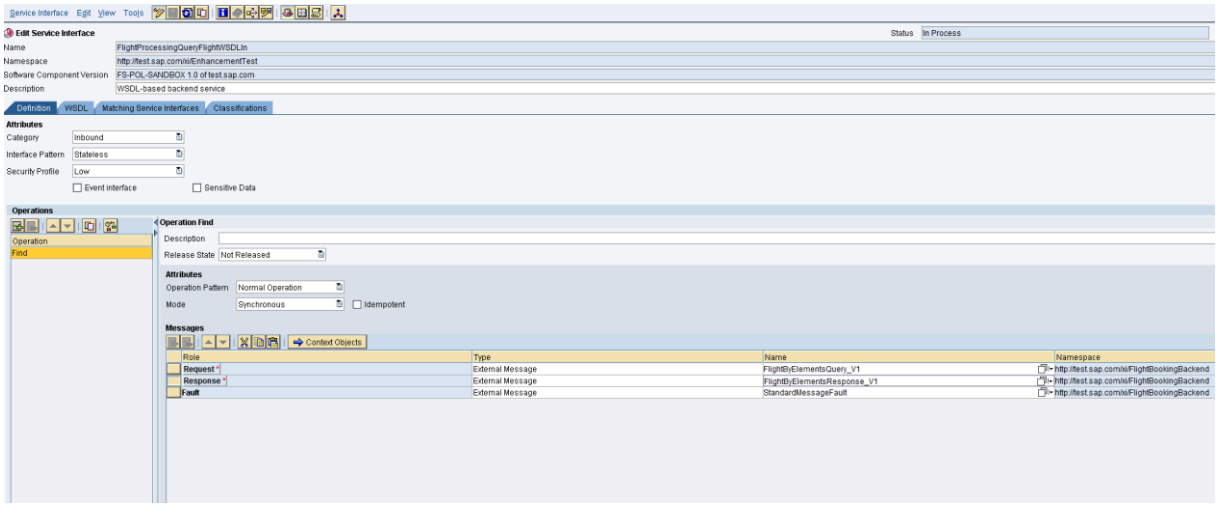


Figure 111: Creation of service interface for WSDL-based service

2. Use your newly created Service Interface/Operation as usual in your POT. It is strongly recommended to copy the External Definition (see above) to the namespace of your POT. This is also mentioned on step "Prepare ESR Environment" of the modeling wizard.
3. Complete the Modeling Wizard and the Specification Wizard as usual
4. In step "Generate BE Counterpart Services" of the Generation Wizard the ESR Counterpart for the WSDL-based service is generated. The generated ESR service interface/operations have to be completed. To do so open the service interface and select "External Message" as type for Request, Response and Fault. Select the corresponding message type using the F4 help.

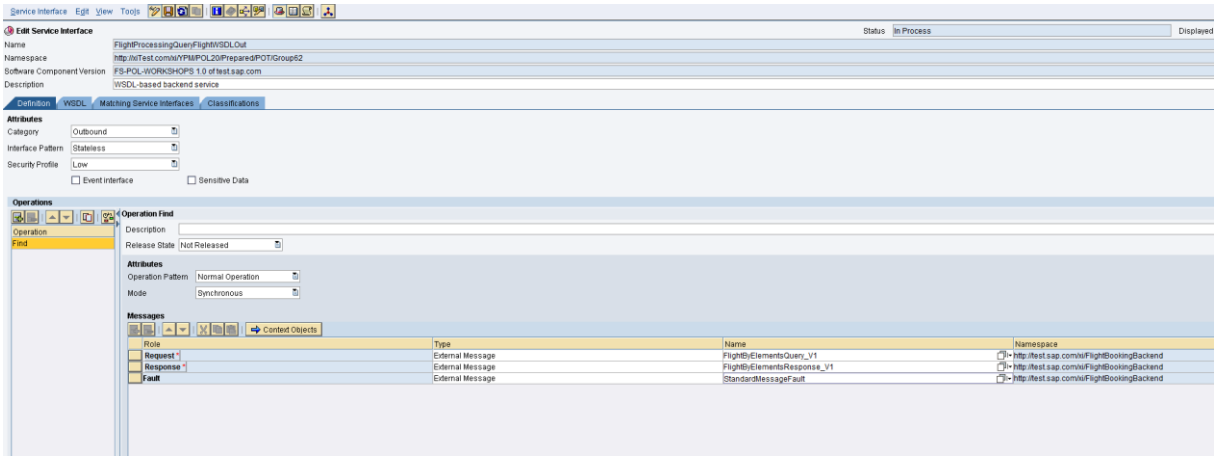


Figure 112: Complete ESR counterpart for WSDL-based service

5. Save and activate your change list. Complete the generation wizard as usual.

8 Implementation and Extension of the Editing UI

The Process Object Builder generates one or- if you choose to do so - multiple user interfaces that are adapted depending on the application scenario for executing process object instances (editing UIs). Multiple application configurations can be used in order to provide UIs that are tailored to specific roles or use cases. A typical use case is the handling of error situations. Such error situations usually require human interaction to change the PO instance or make decisions on subsequent actions. The editing UI provides full access to all public data (such as details and process control constrains of the PO, see section 3.3) as allowed by the service operations (see section 6.3.1).

The individual screens of the editing UI are generated as FPM overview pages and are composed of form and list GUIBBs. Out-of-the box only list and form GUIBBs are used. Other GUIBBs will also work with some manual effort. The POB also supports a POT developer in manually creating CHIPs (Collaborative Human Interface Parts) that can be used in BPM processes to trigger human interaction steps.

Some parts of the generated editing UI work out-of-the-box, for example the search functionality. Other parts, such as the main and subpages, are generated but need to be finalized by the POT developer who has to add the missing custom logic (e.g. mapping of instance details in the model to the screen fields).

This section explains where and how a developer must perform the implementation and (if required) the extension of the generated editing UI.

8.1 Extending the Search Functionality

The search functionality on the initial screen of the UI is completely generated. Users can search for process object instances using different administrative POT attributes (for example, status or reference ID).
The search results list displays the UUID, the reference ID, and the status of the process object instance.

This search functionality can be extended or reduced, if required:

- Add or remove search criteria
- Add or remove result fields in the result list

Prerequisites:

- The required additional fields have been added to the generated feeder structures (`<MLB>FS_SBA_S` for search criteria and `<MLB>FS_SBA_R` for search results)
- The additional search fields have been added to the component configuration of the search UIBB (`<MLB>SBA_S`).

8.1.1 Adding Additional Search Criteria

The signature of the MAP_SELECTION_TO_SEARCH method of the feeder for custom search (<MLB>CL_FS_SBA_C) looks as follows:

Method Name	MAP_SELECTION_TO_SEARCH
Short Description	Maps selection to search
What to do	Map additional fields in the search feeder structure to the corresponding fields for the generic search functionality that is based on the service operation Find<POT>ByAdminstrativeData. Provide these additional fields in parameter R_CUSTOM_SEARCH. The fields can contain the reference IDs of the PO as well as the provider IDs and the reference IDs of the PONs. I_FEEDER_SEARCH_DATA is populated with the search values from the search UI.
Preconditions	None
Result	Mapping is done

Parameter name	Parameter Type	Ref.	Data Type
I_FEEDER_SEARCH_DATA	Importing		FPMGB_T_SEARCH_CRITERIA
R_CUSTOM_SEARCH	Returning		TY_CUSTOM_SEARCH

Example: Search by Provider ID

Listing 8-1 shows a MAP_SELECTION_TO_SEARCH implementation. The field TRAVELLER_ID is added to the DDIC structure <MLB>_FS_SBA_S:

```
data: lr_feeder_search_data type ref to fpmgb_s_search_criteria,
      lr_provider_id
        type ref to /pl9/if_gxx_fb01_it_comp=>ts_bpon_trav_provider_id.

* 1. Pick search entries
loop at i_feeder_search_data reference into lr_feeder_search_data
                                    where search_attribute = 'TRAVELLER_ID'.
  append initial line to r_custom_search-bpon_trav-provider_ids
    reference into lr_provider_id.

* 2. Set search value
  lr_provider_id->* = lr_feeder_search_data->low.
endloop.
```

Listing 8-1: Mapping Additional Search Criteria

1. All search entries referring to the field TRAVELLER_ID are picked from the DDIC structure <MLB>_FS_SBA_S.

   It is possible to search for more than one value for one criterion (e.g. search for all instances with TRAVELLER_ID 4711 or 4813 or 4914).

2. The search value for the provider ID of the traveler is set.

   On the search UI, the TRAVELLER_ID is always maintained as “is equal to”. The entry on the UI is taken over into field low of the value range of I_FEEDER_SEARCH_DATA.

8.1.2 Adding Fields to the Search Result

The signature of the MAP_SEARCH_RESULT_TO_FEEDER method of the feeder for custom search (<MLB>CL_FS_SBA_C) looks as follows:

Method Name	MAP_SEARCH_RESULT_TO_FEEDER
Short Description	Maps search result to feeder
What to do	If additional result fields are defined in the corresponding DDIC structure <MLB>_FS_SBA_R, fill those fields based on the information from the result of the service operation Find<POT>ByAdministrativeData, which provides the full details of all selected instances.
Preconditions	None
Result	The search results are enriched with additional data.

Parameter name	Parameter Type	Ref.	Data Type
I_SEARCH_RESULT	Importing	X	<MLB>IF_FPM_FCD=>TY_SEARCH_RESULT
C_FEEDER_DATA	Changing		<MLB>_FS_SBA_RT

Example: Adding an Additional Counter Field to the Search Results

Listing 8-2 shows a sample implementation of the MAP_SEARCH_RESULT_TO_FEEDER code slot:
The field FLIGHT_COUNT is added to the DDIC structure <MLB>_FS_SBA_R in order to enhance each line of the search result by the number of found flights.

```
data: lr_feeder_data type ref to /pl9/gxx_fb01_fs_sba_r,
      lr_result      type ref to /pl9/wfisbusiness_trip_fs_b125.

* 1. process loop
loop at c_feeder_data reference into lr_feeder_data.
* 2. select corresponding instance data
  read table i_search_result-data with key uuid = lr_feeder_data->pl1_bpo_uuid
     reference into lr_result.
  assert sy-subrc = 0.

* 3. add number of found flights
  loop at lr_result->flgt transporting no fields where
       public_admin_data-status_code <>
       /pl9/ce_gxx_fb01_node_sc=>canceled->get_int_value( ).
    lr_feeder_data->flights_count = lr_feeder_data->flights_count + 1.
  endloop.
endloop.
```

Listing 8-2: Mapping Additional Search Result Fields to the Search Feeder

1. All provided feeder data (C_FEEDER_DATA) is processed in a loop. The following steps are executed for each line separately:
2. The instance data is selected.

`I_SEARCH_RESULT-DATA` contains all the instance data of the found instances. When the code slot is entered, `C_FEEDER_DATA` contains a subset of this information. This means that for each line in `C_FEEDER_DATA` there exists a corresponding line in `I_SEARCH_RESULT`.

3. The field `FLIGHT_COUNT` is populated.

   When the number of flights are counted, all instances that have the status *Canceled* are excluded, because BPONs with this status are not displayed on the editing UI.

8.2 Implementing Main and Subpages

The application configurations for main and subpages that are required to display the data of an instance are generated based on the settings in the Editing UI Wizard. A large part of the implementation of the application controller and feeder classes is also already generated. However, this part of the editing UI is not ready to be used but first the missing implementation of the custom logic needs to be finalized.

8.2.1 Goals of the Implementation of Main and Subpages

In order to finalize the implementation of the main and subpages, a developer needs to perform the following steps:

- Finalize layout and styling

  This includes hiding unnecessary fields, removing buttons that are not needed and adding/ refining descriptions for all fields that are displayed on the screen (e.g. replacing the technical names of the column headings in list GUIBBS with meaningful descriptions). See chapter 8.2.9.
- Implement the mapping between model and feeder for form and list feeders

Optionally, the standard behavior of the generated UI can also be extended:

- Programmatically control the properties of input fields (like visibility and value help)
- Extend the UI with custom buttons
- Improve user experience by automating the navigation on the Editing UI in order to reduce the number of clicks required to get to a certain screen

Table 8-1 describes typical scenarios for extending the Editing UI:

Extension Scenario	What To Do
Add buttons to the global application toolbar	Add the button to the global toolbar in the component configuration of the main page. Implement the methods `SET_BUTTON_PARAMETER` and `HANDLE_EVENT` in the Custom Application Controller. (See sections 8.2.3.1, 8.2.3.4, 8.2.7.1 and 8.2.7.2)
Add button to a feeder	Implement the required functionality in the custom feeder. Define the custom event in method `GET_DEFINITION`. Use method `GET_DATA` to activate the button that is attached to the custom event Implement he event handling in method `PROCESS_EVENT`. (See sections 8.2.3.1, 8.2.3.4, and 8.2.6.) Add the button to the component configuration that the feeder is assigned to using the event defined in `GET_DEINITION`.

Extension Scenario	What To Do
Change field attributes	Extend the field usage of the field for which the value help is desired in method `GET_DATA` of the corresponding custom feeder. (See sections 8.2.3.1, 8.2.2.2, 8.2.5.5 and 8.2.5.6.)
Add value help for field	Extend the field usage of the field for which the value help is desired in method `GET_DATA` of the corresponding custom feeder. (See sections 8.2.3.2, 8.2.6.5 and 8.2.6.6.)
Automating navigation steps	Implement the methods `GET_LEAD_INDEX_BY_PATH` and, if required, `SET_DEFAULT_LINES_FOR_PATH` of the class for default table callback. (See sections 8.2.8.1 and 8.2.8.)

Table 8-1: Scenarios for Extending the Editing UI

8.2.2 Phase Model for Processing Main and Subpages

The following section explains the interplay of the FPM framework, the generated part of the Editing UI and the code slots that can be implemented for main and subpages. It also describes when and in which sequence the corresponding methods are called. Because the phase model of FPM is rather complex and has many variations, only the parts that are relevant from an implementation point of view are explained. Some methods are only relevant during feeder initialization, others are processed with each roundtrip of the FPM application.

To illustrate the phase model, overview tables are used, which provide the following information:

- The **Phase** column shows the phase of the “FPM Application/GUIBB Lifecycle” (Initialization/ After Input/ Before Output).
- The **FPM Method** column contains the corresponding method in the generated part of the Editing UI.
- The **Code Slots** column displays the corresponding methods that contain the code slot for the custom implementation.

The row sequence, as well as the sequence in which methods in the "Code Slots" column are listed, corresponds to the sequence in which the methods are actually called.

Table 8-2 lists the sequence of methods that are called during feeder initialization:

Phase	FPM Method	Code Slots
Initialization	Feeder: INITIALIZE	Custom Feeder: INITIALIZE
Initialization	Feeder: GET_DEFINITION	Custom Feeder: GET_DEFINITION

Table 8-2: Methods Called During Feeder Initialization

Table 8-3 shows the sequence in which methods are called during one single round trip. A round trip starts when the server processes the input from the screen of the user and ends when the next screen is displayed. Each of the feeder methods listed in the "FPM Method" column is first called for all feeders that are displayed on the screen before the method in the next row is called.

Phase	FPM Method	Code Slots
After Input	Feeder: FLUSH	Custom Feeder: MAP_FEEDER_TO_MODEL
After Input	Application Controller: OVERRIDE_EVENT_OVP	Custom Application Controller: HANDLE_EVENT HANDLE_BUTTONS
After Input	Feeder: PROCESS_EVENT	Default Table Callback: SET_DEFAULT_LINES_FOR_PATH Custom Feeder: PROCESS_EVENT
Before Output	Feeder: GET_DATA	Default Table Callback: GET_LEAD_INDEX_BY_PATH Custom Feeder: MAP_MODEL_TO_FEEDER GET_DATA

Table 8-3: A Round Trip of the Editing UI

8.2.3 Basic Principles

8.2.3.1 Application Mode

The editing UI distinguishes between edit and display mode. The application mode is controlled by the FPM framework.

The buttons that are used to switch from display to edit mode (and vice versa) are already part of the generated configuration. The link between the activation status of the standard buttons and the application mode is also automatically provided.

However, it might be required to react on such a mode switch, for example by changing the "ready-for-input" status of fields on the GUIBB. In case the generated Editing UI is extended with custom buttons, it may also be required to change the activation status of custom buttons. The application mode is available as an input parameter in the relevant code slots of the custom application controller (`HANDLE_EVENT`) and in the feeders (`GET_DATA`). More details will follow in later sections.

It may also be required to explicitly trigger a mode switch, for example because a custom button has been pressed. To achieve this, the custom application controller contains the methods `SET_EDIT_MODE` and `SET_DISPLAY_MODE`.

8.2.3.2 Connecting GUIBBs to the Model

Figure 113 shows the basic aspects that need to be taken into consideration for GUIBB definition, GUIBB - model-assignment and custom feeder implementation.

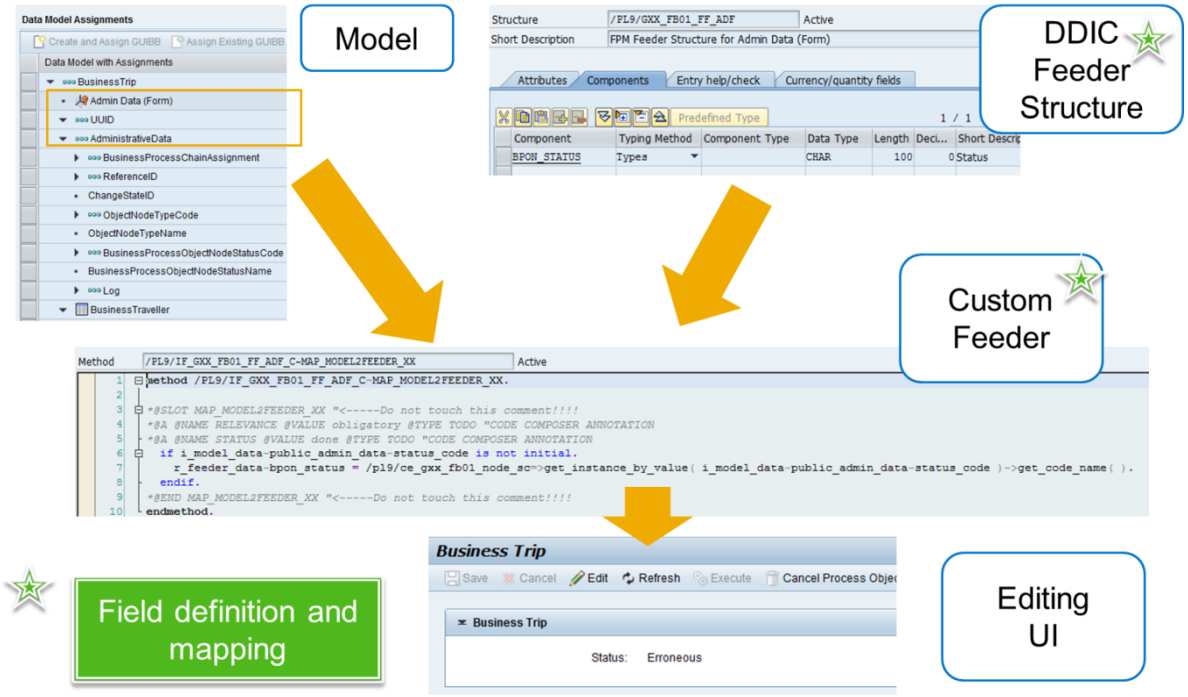


Figure 113: GUIBBs and Feeders - Basic Principles

The model contains the full public data of a POT (see section 6.5.6) and can be imagined as a deeply structured hierarchical tree. The nodes in this tree can be structures or tables. In the Editing UI Wizard of the POB, the POT implementer decides, which parts of the model are to be displayed on the editing UI.

To display or edit data, (form or list) GUIBBs are assigned to individual nodes of the model tree (showing different parts of the model). Each GUIBB is linked to exactly one feeder. However, one GUIBB can be assigned to several nodes of the model. To distinguish these assignments from each other, the combination of the ABAP short name for the GUIBB and the ABAP short name for the assignment must always be unique.

Structure nodes are typically assigned to form feeders. Tabular nodes are typically assigned to list feeders (sometimes extended by form feeders in order to display details for the selected line in the list feeder). In fact, in order to ensure that a user can navigate through the hierarchical structure of the editing UI, each tabular node that is contained in the path of data that is relevant for display must be assigned to a list feeder. In the special case that the navigation through the model via list feeder lead selection is not possible or desired, a POT developer needs to deal with these “gaps” as explained in section 8.2.8.1.

The fields/columns that can be displayed in a feeder are defined in a DDIC feeder structure (`<MLB>_<FEEDER_TYPE_ABBR>_<FEEDER_ABBR>`, exactly one per feeder). To map the model data to this feeder structure, a custom feeder (`<MLB>CL_<FEEDER_TYPE_ABBR>_<FEEDER_ABBR>_C`) must be implemented. For each feeder assignment to a model node, one pair of methods (`MAP_MODEL2FEEDER_<ASSIGNMENT_ABBR>` and `MAP_FEEDER2MODEL_<ASSIGNMENT_ABBR>`) is available. The method for mapping the model to the feeder must always be implemented, otherwise the corresponding GUIBB does not display any data. The method for mapping the feeder to the model only needs to be

implemented if data is supposed to be changed on the editing UI (a decision that is taken by the POT implementer).

The field arrangement and UI styling of the GUIBB can be adjusted by the POT implementer in the component configuration.

8.2.3.3 Field Descriptions

Field descriptions are used to control properties that define the use of screen fields on the UI (for example, visibility, required entries, display options and value help). The generated editing UI is fully compliant with the FPM standard.

All screen fields are listed in the field catalog. The field catalog is based on the feeder structures `<MLB>_<FEEDER_TYPE_ABBR>_<FEEDER_ABBR>`. These structures have been generated automatically and manually extended by the developer. Based on these structures, the field catalog is created automatically in the generated parts of the Editing UI.

The field descriptions (ABAP type `FPMGB_T_FIELDUSAGE`) can then be programmatically modified in the `GET_DATA` code slot of the corresponding custom feeder class, according to the FPM standard.

8.2.3.4 Actions and Events

In FPM, events are used to react to user actions (for example pressing a pushbutton). There are standard FPM events and standard editing UI events that are available in the editing UI by default. The event IDs of these events are not relevant for developers as these are not available in the relevant code slots. However, it is possible for the POT developer to define custom events. New events can be defined in the `GET_DEFINITION` method of custom feeders. Custom event IDs must start with `CPL1_` in order to distinguish them from the FPM event IDs defined by the Builder and from the standard FPM IDs. The handler code for custom events is placed in the `PROCESS_EVENT` method of the custom feeders.

8.2.3.5 Handling of Error Messages

Business rule checks or consistency checks that are related to the PO should not be performed in the editing UI as these types of checks need to be UI independent. (There will be probably other UIs from which the same PO is accessed.) Therefore, these checks are implemented in the *Check* phase of the POT (see section 7.1.6). Error messages from the PO are automatically displayed by the generated part of the Editing UI.

However there may be checks that are specific to the UI (for example for custom buttons that assume that a line in a list feeder has been selected). In such cases, it makes sense that the developer explicitly raises messages in the corresponding code slot implementations.

Messages are raised according to the FPM standard. Messages can be raised from custom feeder code slots (`GET_DATA` and `PROCESS_EVENT`) based on the ABAP type `FPMGB_T_MESSAGES`, or they can be raised from the custom application controller code slot `HANDLE_EVENT` via the FPM message helper `<MLB>FPM_MSG_H`.

8.2.3.6 Providing Meaningful Text for Internal Codes

Value lists in the model are usually implemented as enumerations. From the perspective of a UI developer, enumerations help to convert internal codes to meaningful text.
Take, for example, the status code of a BPON: With the enumeration `<MLB>NODE_SC` you get the correct enumeration instance from a code value via method `GET_INSTANCE_BY_VALUE`. The method `GET_CODE_NAME` then provides a meaningful text for the code. This text, however, is not language-dependent. It is provided in the language in which the enumeration has been generated - which is not necessarily the current logon language

8.2.4 Tools and Helpers

There are a number of helpers for use in the code slot implementations that help the developer to implement the main and subpages of the Editing UI.

8.2.4.1 Custom Controller

The custom controller (`<MLB>IF_FPM_CCTRL`) is used to implement the event handlers of those buttons that are placed on the global application toolbar. It can be used to trigger actions on BPO and BPON level. Compared to the generated main FPM Controller `<MLB>IF_FPM_CTL` (which is not accessible from code slots), the custom controller provides a subset of the methods of this controller and also the additional method `GET_APPLICATION_CONFIGURATION`.

Some of the methods can only be called depending on the status of the POT and the display mode of the UI, which is something that a code slot implementer should keep in mind. The methods will otherwise raise the exception `<MLB>CX_IM`.

The custom controller provides the following methods:

Method Name	Description
GET_APPLICATION_CONFIGURATION	Provides the application configuration that the controller is currently attached to.
CHECK_BPO	Performs the check service operation on BPO level. Returns the result (successful/ not successful, log) of the check (type `<MLB>IF_FPM_TAC=>TY_CHECK_RESULT`).
EXECUTE_BPO	Triggers the execute service operation on BPO level. Returns the resulting message log (type `<MLB>IF_FPM_TAC=>TY_LOG`).
CANCEL_BPO	Triggers the cancel service operation on BPO level. Returns the resulting message log (type `<MLB>IF_FPM_TAC=>TY_LOG`).
REPROCESS_BPO	Triggers the reprocess service operation on BPO level. Returns the resulting message log (type `<MLB>IF_FPM_TAC=>TY_LOG`).

Method Name	Description
ABORT_EXECUTION_BPO	Triggers the abort execution service operation on BPO level. Returns the resulting message log (type <MLB>IF_FPM_TAC=>TY_LOG).
CONFIRM_COMPENSATION_BPO	Triggers the confirm compensation service operation on BPO level. Returns the resulting message log (type <MLB>IF_FPM_TAC=>TY_LOG).
CONFIRM_ERROR_CORRECTION_BPO	Triggers the confirm error correction service operation. This operation confirms that errors from provider log are corrected.
CONFIRM_COMPENSATION_BPON	Triggers the confirm compensation service operation on BPON level. The BPON key (type <MLB>IF_FPM_TAC=>TY_BPON_KEY_CUSTOM), consisting of node type and BPON UUID, needs to be provided as input parameter. Returns the resulting message log (type <MLB>IF_FPM_TAC=>TY_LOG).
SAVE	Triggers the update service operation on BPO level. Returns the resulting message log (type <MLB>IF_FPM_TAC=>TY_LOG).
REPROCESS_BPON	Triggers the reprocess service operation on BPON level. The BPON key (type <MLB>IF_FPM_TAC=>TY_BPON_KEY_CUSTOM), consisting of node type and BPON UUID, needs to be provided as input parameter. Returns the resulting message log (type <MLB>IF_FPM_TAC=>TY_LOG).
SET_EDIT_MODE	Switches to edit mode.
SET_DISPLAY_MODE	Switches to display mode.
INITIALIZE	Initializes the complete model (re-reads the BPO via read service)
SEARCH	Triggers the find service operation. The selection (type <MLB>IF_FPM_TAC=>TY_SEARCH_CRITERIA) needs to be provided as input parameter. Returns the result (type <MLB>IF_FPM_TAC=>TY_SEARCH_RESULT) of the find operation.
RAISE_COMPLETE_EVENT	Raises PL1_COMPLETE event. The CHIP output structure <MLB>CH_O_V_DFT needs to be provided as importing parameter I_DATA.
IS_DISPLAY_MODE	Checks whether display mode is active (returns the result as boolean parameter).
RAISE_FPM_EVENT_BY_ID	Raises an FPM event. The event ID (type <MLB>IF_FPM_TAC=>TY_EVENT_ID) needs to be provided as an importing parameter.

Method Name	Description
RAISE_CUSTOM_CHIP_EVENT	Raises a custom CHIP event. The outport name (type <MLB>IF_FPM_TAC=>TY_OUTPORT_NAME) and the outport structure (<MLB>CH_O_V_DFT) need to be provided.

Table 8-4: Methods of the Custom Controller

8.2.4.2 Custom Data Helper

The custom data helper is used to read from and/or write to the model from the custom application controller <MLB>CL_FPM_ACC_C (which should not be confused with the custom controller (<MLB>IF_FPM_CCTRL).

The custom data helper is available in two variants (different interfaces which are implemented by the same class):

- <MLB>IF_FPM_CDTHR contains methods for read operations.
  This interface is available as importing parameter in the method SET_BUTTON_PARAMETER.
- <MLB>IF_FPM_CDTHS contains all the methods from the interface <MLB>IF_FPM_CDTHR as well as methods that allow for changing the model.
  This interface is available as importing parameter in the method HANDLE_EVENT.

The <MLB>IF_FPM_CDTHR interface contains the following methods:

Method Name	Description
GET_BPON_<BPON_ABBR>	Gets the data of a specific BPON. This method is available once per BPON. It receives the BPON UUID (<MLB>IF_IT_BASE=>TS_UUID_INTERN) as importing parameter and returns the BPON details (<MLB>IF_IT_COMP=>TS_BPON_<BPON_ABBR>_COMPLETE).
GET_BPO_DATA	Returns the complete model data (type <MLB>IF_FPM_TAC=>TS_BPO_PUBLIC_TOTAL_UI).
GET_BPON_TYPE_BY_CROSS_REF	Provides the type of the BPON that a cross-reference points to (target BPON). Receives the public admin data as importing parameter of the generic type DATA (the concrete type depends on the node type: <MLB>IF_IT_COMP=>TS_BPON_<BPON_ABBR>_PRIV_ADM) and returns the node type (type ref to <MLB>CE_NODE_TY_C).
GET_BPONS_BY_CROSS_REFERENCE	Provides data for the target BPONs of a cross-reference. Receives the public admin data as importing parameter of the generic type DATA (the concrete type depends on the node type: <MLB>IF_IT_COMP=>TS_BPON_<BPON_ABBR>_PRIV_ADM) and returns the data for the current node using a table of the generic type DATA (the concrete type of each line depends on the node type: <MLB>IF_IT_COMP=>TS_BPON_<BPON_ABBR>_COMPLETE)

Table 8-5: Custom Data Helper – Read Methods

Compared to <MLB>IF_FPM_CDTHR, the interface <MLB>IF_FPM_CDTHS contains a set of additional methods.

For each BPON, the interface provides the following methods:

Method Name	Description
CREATE_BPON_<BPON_ABBR>	Creates a new BPON. The public details of the BPON (<MLB>IF_FPM_IT_C=>TS_BPON_<BPON_ABBR>_UT_CRT_MD) and the creation instruction code (<MLB>CE_CRTE_IC) need to be provided.
DELETE_BPON_<BPON_ABBR>	Deletes a BPON. The UUID (<MLB>IF_IT_BASE=>TS_UUID_INTERN) needs to be provided as an importing parameter.
UPDATE_BPON_<BPON_ABBR>	Updates a BPON. The public details of the BPON (<MLB>IF_FPM_IT_C=>TS_BPON_<BPON_ABBR>_UT_UPD_MD) need to be provided.

Table 8-6: Custom Data Helper – Set Methods

8.2.4.3 Feeder Helper

The feeder helper allows a customer to read from and/or write to the model in several code slots from custom feeder implementations. Internally, the feeder helper uses the methods provided by the data helper (see section 8.2.4.2).

The feeder helper is available in two variants:

- <MLB>IF_FPM_FH_R provides access to data from other BPONs.

  It is provided as importing parameter for the methods MAP_MODEL2FEEDER_<ASSIGNMENT_ABBR> and MAP_FEEDER2MODEL_<ASSIGNMENT_ABBR>.
- <MLB>IF_FPM_FH_S includes all methods from <MLB>IF_FPM_FH_R. Additionally, it allows the data of other BPONS to be changed.

  This variant of the feeder helper is provided as importing parameter for the method PROCESS_EVENT.

The <MLB>IF_FPM_FH_R interface contains the following methods:

Method Name	Description
GET_BPO_DATA	Returns the complete model data (type <MLB>IF_FPM_TAC=>TS_BPO_PUBLIC_TOTAL_UI).
GET_NODE_DATA	Returns the data for the current node (<MLB>IF_IT_COMP=>TS_BPON_<BPON_ABBR>_COMPLETE) if it is called on BPON level and a lead selection is set. Returns the complete model data (type <MLB>IF_FPM_TAC=>TS_BPO_PUBLIC_TOTAL_UI) if it is called on BPO level or no lead selection is set. The technical type for the return value is the generic type DATA in order to reflect these different scenarios.
GET_NODE_TYPE	Returns the type of the current node (type <MLB>CE_NODE_TY_C).
GET_BPONS_BY_CROSS_REFERENCE	Provides data for target BPONs of a Cross-Reference. Receives the

Method Name	Description
	public admin data as importing parameter of the generic type DATA (the concrete type depends on the node type: <MLB>IF_IT_COMP=>TS_BPON_<BPON_ABBR>_PRIV_ADM) and returns the data for the current node using a table of the generic type DATA (the concrete type of each line depends on the node type: <MLB>IF_IT_COMP=>TS_BPON_<BPON_ABBR>_COMPLETE)
GET_BPON_TYPE_BY_CROSS_REF	Provides the type for the BPON a cross-reference is pointing to. Receives the public admin data as importing parameter of the generic type DATA (the concrete type depends on the node type: <MLB>IF_IT_COMP=>TS_BPON_<BPON_ABBR>_PRIV_ADM) and returns the node type (type ref to <MLB>CE_NODE_TY_C).

Table 8-7: Feeder Helper – Read Methods

Compared to <MLB>IF_FPM_FH_R, the interface <MLB>IF_FPM_FH_S contains a set of additional methods.

For each BPON, the interface provides the following methods:

Method Name	Description
CREATE_BPON_<BPON_ABBR>	Creates a new BPON. The public details of the BPON (<MLB>IF_FPM_IT_C=>TS_BPON_<BPON_ABBR>_UT_CRT_MD) and the creation instruction code (<MLB>CE_CRTE_IC) need to be provided.
DELETE_BPON_<BPON_ABBR>	Deletes a BPON. The UUID (<MLB>IF_IT_BASE=>TS_UUID_INTERN) needs to be provided as an importing parameter.
UPDATE_BPON_<BPON_ABBR>	Updates a BPON. The public details of the BPON (<MLB>IF_FPM_IT_C=>TS_BPON_<BPON_ABBR>_UT_UPD_MD) need to be provided.

Table 8-8: Feeder Helper – Set Methods

8.2.4.4 Helper for UI Messages

The helper for UI messages (<MLB>IF_FPM_MSG_H) is used to raise messages in the format that is required by FPM. The helper is available in the HANDLE_EVENT code slot of the custom application controller.

The helper contains the following methods:

Method Name	Description
REPORT_MESSAGES_BY_EXCEPTION	Reports UI messages by exception. The exception needs to be supplied as importing parameter (type <MLB>CX_EX or a subclass of this type).
REPORT_MESSAGES_BY_LOG	Reports UI Messages by log.

Method Name	Description
	The log is supplied as importing parameter and should be an internal representation of GDT `Log` (like for example `<MLB>IF_FPM_TAC=>TY_LOG`).
REPORT_MESSAGE_BY_SY	Reports UI Messages by system fields. The system fields of type `SYST` have to be supplied as importing parameter.

Table 8-9: Methods of the Helper for UI Messages

8.2.5 Custom Feeder Class Implementation – Basics

The generated Editing UI only supports list and form feeders. The purpose of a feeder is to manage the data transfer between the screen and the model and to react on events.

The implementation of the code slots for list and form feeders is slightly different, due to the nature of the displayed data (flat structures for form feeders and tables for list feeders). The only code slot that is exactly the same for list and form feeders is the `INITIALIZE` code slot.

The code slot for the data transfer from model to feeder (located in method `MAP_MODEL2FEEDER_<ASSIGNMENT_ABBR>`) is required to display model data on the screen. This method is called from the generated implementation of the standard FPM method `GET_DATA`. In order to update the model with the data that has been entered on the screen, the corresponding code slot in method `MAP_FEEDER2MODEL_<ASSIGNMENT_ABBR>` is called. This method is called from the generated implementation of the standard FPM method `FLUSH`.

8.2.5.1 Initializing a Feeder

The signature of the `INITIALIZE` method of any feeder class (`<MLB>CL_<FL or FF>_<FEEDER_ABBR>_C`) looks as follows:

Method Name	INITIALIZE
Short Description	Initializes the feeder for a generic UIBB
What to do	Initialize the feeder and instantiate attributes of the feeder class.
Preconditions	None
Result	The instance is initialized along with all dependencies.

Parameter name	Parameter Type	Ref.	Data Type
`I_PARAMETER`	Importing	X	`FPMGB_T_PARAM_VALUE`
`I_APP_PARAMETER`	Importing		`IF_FPM_PARAMETER`
`I_COMPONENT_NAME`	Importing		`FPM_COMPONENT_NAME`
`I_CONFIG_KEY`	Importing	X	`WDY_CONFIG_KEY`
`I_INSTANCE_ID`	Importing		`FPM_INSTANCE_ID`

Example: Feeder Initialization

The following listing shows the `INITIALIZE` code slot of a feeder class:

```
* 1. initialize the common helper
  mr_common_helper = /pl9/cl_workshop_common_helper=>create( ).
```

Listing 8-3: Feeder Initialization

1. Initialize the private member attribute `mr_common_helper`, which will be needed later in the feeder implementation.

8.2.5.2 Map Model to Feeder (Form Feeder)

For form feeders, there is one model-to-feeder method for each combination of feeder and feeder assignment.

The signature of the `MAP_MODEL2FEEDER_<ASSIGNMENT_ABBR>` method of the form feeder class (`<MLB>CL_FF_<FEEDER_ABBR>_C`) looks as follows:

Method Name	MAP_MODEL2FEEDER_<ASSIGNMENT_ABBR>
Short Description	Maps deep model data structure to flat feeder data structure
What to do	Map the model data that corresponds to the feeder location to the feeder data. If access to other nodes is required, the feeder helper provides access to these nodes.
Preconditions	The correct model data is supplied.
Result	Model data is mapped to feeder data.

Parameter name	Parameter Type	Ref.	Data Type
`I_FEEDER_HELPER`	Importing	X	`<MLB>IF_FPM_FH_R`
`I_MODEL_DATA`	Importing		`<MLB>IF_FPM_IT_C=>TS_<Model Data>`
`R_FEEDER_DATA`	Returning		`<MLB><FEEDER>`

Example: Displaying a BPON Status Name

Listing 8-4 provides an example of a `MAP_MODEL_TO_FEEDER` implementation. The form UIBB is used to display the status of one BPON. The field catalog of the UIBB contains only one field – which is `BPON_STATUS`.

```
* 1. Mapping of the status code
if i_model_data-public_admin_data-status_code is not initial.
  r_feeder_data-bpon_status
     = /pl9/ce_gxx_fb01_node_sc=>get_instance_by_value(
            i_model_data-public_admin_data-status_code )->get_code_name( ).
endif.
```

Listing 8-4: Mapping Model Data to Feeder Data (Form Feeder)

1. The status code provided in `I_MODEL_DATA` is mapped to the field `BPON_STATUS` of the feeder structure.

The transformation between STATUS_CODE and BPON_STATUS is done by the status code enumeration (see section 8.2.3.6).

8.2.5.3 Map Model to Feeder (List Feeder)

For list feeders, there is one model-to-feeder method for each combination of feeder and feeder assignment.

The signature of the MAP_MODEL2FEEDER_<ASSIGNMENT_ABBR> method of the list feeder class (<MLB>CL_FL_<FEEDER_ABBR>_C) looks as follows:

Method Name	MAP_MODEL2FEEDER_<ASSIGNMENT_ABBR>
Short Description	Maps the tabular model data structure to the tabular feeder data structure
What to do	Map the model data that corresponds to the feeder location to the tabular feeder data. Transfer _LS_GUID from model data to feeder data, too, in order to uniquely identify the lines in the feeder table. If access to other nodes is required, the feeder helper provides access to these nodes.
Preconditions	The correct model data is supplied.
Result	Model data is mapped to the feeder table.

Parameter name	Parameter Type	Ref.	Data Type
I_FEEDER_HELPER	Importing	X	<MLB>IF_FPM_FH_R
I_MODEL_DATA	Importing		<MLB>IF_FPM_IT_C=>TT_<Model Data>
R_FEEDER_DATA	Returning		<MLB><FEEDER>_T

Example: Showing a List of Flights

Listing 8-5 provides an example of a MAP_MODEL_TO_FEEDER implementation. The list UIBB is used to display an overview of flights. The structure of a tabular model node needs to be mapped to the table that is displayed with the list feeder.

```
data: lr_model_data
        type ref to /pl9/if_gxx_fb01_fpm_it_c=>ts_bpon_flgt_public_complete,
      lr_feeder_data type ref to /pl9/gxx_fb01_fl_flgtl.
* 1. process each model entry separately
loop at i_model_data reference into lr_model_data.
* 2. add new line to feeder table
  append initial line to r_feeder_data reference into lr_feeder_data.
* 3. map obligatory parameter pl1_ls_guid
  lr_feeder_data->pl1_ls_guid  = lr_model_data->pl1_ls_guid.
* 4. map any other data
  lr_feeder_data->airport_from = lr_model_data->details-from-airport_id-content.
  lr_feeder_data->airport_to   = lr_model_data->details-to-airport_id-content.
  lr_feeder_data->amount       = mr_common_helper->convert_amount_to_external(
              i_amount    = lr_model_data->details-price_amount-content
```

```
                i_currency = lr_model_data->details-price_amount-currency_code ).
  if lr_model_data->public_admin_data-status_code is not initial.
    lr_feeder_data->bpon_status
       = /pl9/ce_gxx_fb01_node_sc=>get_instance_by_value(
            lr_model_data->public_admin_data-status_code )->get_code_name( ).
  endif.
  lr_feeder_data->flight_date
    = lr_model_data->process_control_constraints-planned_flight_date.
  if lr_model_data->public_admin_data-provider_id is not initial.
    lr_feeder_data->provider_id
      = |{ lr_model_data->public_admin_data-provider_id-airline_id
        }-{ lr_model_data->public_admin_data-provider_id-connection_id
        }-{ lr_model_data->public_admin_data-provider_id-planned_flight_date}|.
  else.
    lr_feeder_data>provider_id = 'No Flight Found!' "#EC NOTEXT.
*   Text directly in source code to make the example simpler to read
  endif.
  lr_feeder_data->reference_id
    = lr_model_data->public_admin_data-reference_id-content.
  lr_feeder_data->uuid = lr_model_data->public_admin_data-uuid.
endloop.
```

Listing 8-5: Mapping Model Data to Feeder Table (List Feeder)

1. All provided model data `I_MODEL_DATA` is processed in a loop. The following steps are executed separately for each model entry.
2. A new line is added to `R_FEEDER_DATA`.
3. The mandatory parameter `PL1_LS_GUID` is filled from the model data.

   This parameter is not intended for display. It is needed to identify the lines in case of write-back or navigation.
4. All parameters of `R_FEEDER_DATA` are filled.
   - The amount is converted to the format required for external display.

     Because this amount conversion functionality is needed in several places, it is implemented once in a common helper class for reuse.
   - The `STATUS_CODE` is transformed from an enumeration value into a readable code name.
   - The `PROVIDER_ID` is filled from a structure of three fields by concatenating these fields and separating the content by a dash (using an ABAP string template).

8.2.5.4 Map Feeder to Model (Form Feeder)

The `MAP_FEEDER2MODEL` method is used to map the data displayed on the screen (GUIBB) to the corresponding node of the model.

The signature of the `MAP_FEEDER2MODEL_<ASSIGNMENT_ABBR>` method of the form feeder class (`<MLB>CL_FF_<FEEDER_ABBR>_C`) looks as follows:

Method Name	MAP_FEEDER2MODEL_<ASSIGNMENT_ABBR>
Short Description	Maps the flat feeder structure to the structure of the model node
What to do	Map the fields of the feeder structure back to the model. It might be required to convert the format of the field. If access to other nodes is required, the feeder helper provides access to these nodes.
Preconditions	The model data has been supplied from the correct node.
Result	The model has been updated with the current feeder data.

Parameter name	Parameter Type	Ref.	Data Type
`I_FEEDER_HELPER`	Importing	X	`<MLB>IF_FPM_FH_R`
`I_FEEDER_DATA`	Importing		`<MLB><FEEDER>`
`C_MODEL_DATA`	Returning		`<MLB>IF_FPM_IT_C=>TS_<Model Data>`

Example: Code Slot Implementation if No Data is Changed

The feeder-to-model code slot is only required if data on the screen can been changed. Otherwise (if the screen does not allow for changing the feeder data) mapping the feeder data back into the model is not useful.

Listing 8-6 shows an example of an empty `MAP_FEEDER_TO_MODEL` implementation:

```
return.
```

Listing 8-6: An Empty Code Slot Implementation

8.2.5.5 Map Feeder to Model (List Feeder)

The MAP_FEEDER2MODEL method is used to map the data displayed on the list GUIBB to the corresponding node of the model. For list feeders, the lines of the feeder table must be mapped to the corresponding lines of the model.

The signature of the `MAP_FEEDER2MODEL_<ASSIGNMENT_ABBR>` method of the form feeder class (`<MLB>CL_FL_<FEEDER_ABBR>_C`) looks as follows:

Method Name	MAP_FEEDER2MODEL_<ASSIGNMENT_ABBR>
Short Description	Maps the flat feeder table to the corresponding model node
What to do	Map rows and fields of the feeder table to the model: First, for each line of the feeder table, read the corresponding line of the model (based on the field `PL1_LS_GUID`). Then, perform the field mapping.
Preconditions	The model data has been supplied from the correct node according to the feeder path.
Result	The model is updated with the current feeder data.

Parameter name	Parameter Type	Ref.	Data Type
`I_FEEDER_HELPER`	Importing	X	`<MLB>IF_FPM_FH_R`
`I_FEEDER_DATA`	Importing		`<MLB><FEEDER>_T`
`C_MODEL_DATA`	Returning		`<MLB>IF_FPM_IT_C=>TT_<Model_Data>`

Example: Changing Data in a List Feeder to Search for New Flights

Listing 8-7 shows an example of a `MAP_FEEDER_TO_MODEL` implementation for a list feeder. In the list GUIBB that is used to display an overview of the selected flights, new table lines can be added. For these new lines, the user can change the fields `AIRPORT_FROM`, `AIRPORT_TO` and `PLANNED_FLIGHT_DATE` in order to search for additional flights.

```
* 1. process each feeder entry
loop at i_feeder_data reference into lr_feeder_data.
* 2. Read the corresponding model entry
  read table c_model_data with key pl1_ls_guid = lr_feeder_data->pl1_ls_guid
                          reference into lr_model_data.
  if sy-subrc = 0.
* 3. map data
    lr_model_data->details-from-airport_id-content =
    lr_feeder_data->airport_from.
    lr_model_data->details-from-airport_id-scheme_agency_id =
      /pl9/if_workshop_common_helper=>con_airport_id-supplementary_components-
                                                          scheme_agency_id.
    lr_model_data->details-to-airport_id-content =
    lr_feeder_data->airport_to.
    lr_model_data->details-to-airport_id-scheme_agency_id =
      /pl9/if_workshop_common_helper=>con_airport_id-supplementary_components-
                                                          scheme_agency_id.
    lr_model_data->process_control_constraints-planned_flight_date =
    lr_feeder_data->flight_date.
  endif.
endloop.
```

Listing 8-7: Mapping Data from a List Feeder to the Model

1. All entries in the feeder table `I_FEEDER_DATA` are processed in a loop. The following steps are executed separately for each entry.
2. The corresponding line in `C_MODEL_DATA` is selected using `PL1_LS_GUID`.
3. The feeder data for `airport from`, `airport to` and `planned flight date` is taken over into the corresponding fields of `C_MODEL_DATA`.

   The `SCHEME_AGENCY_ID` for the airports is not displayed. Its default value is automatically added in this code slot.

8.2.6 Custom Feeder Class Implementation – Advanced Features

To get a full-fledged editing UI with basic features, no further implementation is necessary.

Advanced features of custom feeder class implementations include event handling, additional feeder-specific definitions as well as the implementation of custom feeder behavior (such as changing field visibility and activation status). The implementation of these features in the methods `PROCESS_EVENT`, `GET_DEFINITION` and `GET_DATA` is very similar to that of the FPM standard. This is also reflected in the signatures of these methods, which are only slightly different. Sometimes a standard FPM parameter (like `EO_FIELD_CATALOG` in `GET_DEFINITION`) is missing because the generated part of the editing UI already takes care of the feature (in this case, by building the field catalog from the feeder structure). Sometimes, parameters such as the generated helpers are additionally included, e.g. the feeder helper in the `GET_DATA` code slot. The implementation of the code slots in these methods is optional.

8.2.6.1 Feeder Definition (Form Feeder)

The signature of the `GET_DEFINITION` method of the form feeder class (`<MLB>CL_FF_<FEEDER_ABBR>_C`) looks as follows:

Method Name	GET_DEFINITION
Short Description	Defines field catalog/actions for feeder
What to do	Set up additional feeder-specific definitions, if required, according to the FPM standard procedures. Examples: • Adding a custom event • Changing field properties
Preconditions	None
Result	Additional definitions are provided.

Parameter name	Parameter Type	Ref.	Data Type
`C_MESSAGE`	Changing		`FPMGB_S_T100_MESSAGE`
`C_FIELD_DESCRIPTION`	Changing		`FPMGB_T_FORMFIELD_DESCR`
`C_ACTION_DEFINITION`	Changing		`FPMGB_T_ACTIONDEF`
`C_SPECIAL_GROUPS`	Changing		`FPMGB_T_SPECIAL_GROUPS`
`C_ADDITIONAL_ERROR_INFO`	Changing		`DOKU_OBJ`
`C_DND_DEFINITION`	Changing		`FPMGB_T_DND_DEFINITION`

The implementation of the `GET_DEFINITION` method of the form feeder class is quite similar to that of the `GET_DEFEINITION` method of the list feeder class. The main difference is that additional parameters that deal with list properties, such as `ROW_ACTIONS`, need to be considered in the list feeder.

8.2.6.2 Feeder Definition (List Feeder)

The signature of the `GET_DEFINITION` method of the form feeder class (`<MLB>CL_FL_<FEEDER_ABBR>_C`) looks as follows:

Method Name	GET_DEFINITION
Short Description	Defines Field Catalog/Actions for Feeder
What to do	Set up additional feeder-specific definitions, if required, according to the FPM standard procedures. Examples: • Adding a custom event • Changing field properties
Preconditions	None
Result	Additional definitions are provided.

Parameter name	Parameter Type	Ref.	Data Type
C_ADDITIONAL_ERROR_INFO	Changing		DOKU_OBJ
C_DND_DEFINITION	Changing		FPMGB_T_DND_DEFINITION
C_ROW_ACTIONS	Changing		FPMGB_T_ROW_ACTION
C_OPTIONS	Changing		FPMGB_S_LIST_OPTIONS
C_FIELD_DESCRIPTION	Changing		FPMGB_T_LISTFIELD_DESCR
C_ACTION_DEFINITION	Changing		FPMGB_T_ACTIONDEF
C_SPECIAL_GROUPS	Changing		FPMGB_T_SPECIAL_GROUPS
C_MESSAGES	Changing		FPMGB_S_T100_MESSAGE

Example: Adding a custom event

Listing 8-8 shows an example of a `GET_DEFINITION` implementation for a list feeder. An event for copying a line is added to the feeder.

```
data: lr_action_def type ref to fpmgb_s_actiondef.

* add event to the action definition table
append initial line to c_action_definition reference into lr_action_def.
lr_action_def->id = 'CPL1_COPY_LINE'.
```

Listing 8-8: Adding a Custom Event

8.2.6.3 Handling Feeder Events (Form Feeder)

The signature of the PROCESS_EVENT method of the form feeder class (<MLB>CL_FF_<FEEDER_ABBR>_C) looks as follows:

Method Name	PROCESS_EVENT
Short Description	Process Event
What to do	Execute action depending on event. If an error occurs during event processing, set C_RESULT to FAILED. This will abort the processing of the event loop along with all subsequent events.
Preconditions	The required actions have been defined in the GET_DEFINITION method.
Result	The event is processed.

Parameter name	Parameter Type	Ref.	Data Type
I_FEEDER_HELPER	Importing	X	<MLB>IF_FPM_FH_S
I_EVENT	Importing	X	CL_FPM_EVENT
I_RAISED_BY_OWN_UI	Importing		BOOLE_D
C_RESULT	Changing		FPM_EVENT_RESULT
C_MESSAGES	Changing		FPMGB_T_MESSAGES

The implementation of the PROCESS_EVENT method of the form feeder class is quite similar to the PROCESS_EVENT method of the list feeder class. The main difference is that parameters of the list feeder that deal with list indices, such as LEAD_INDEX and EVENT_INDEX, do not matter for form feeders.

8.2.6.4 Handling Feeder Events (List Feeder)

The signature of the PROCESS_EVENT method of the form feeder class (<MLB>CL_FL_<FEEDER_ABBR>_C) looks as follows:

Method Name	PROCESS_EVENT
Short Description	Process Event
What to do	Execute an action depending on the corresponding event.
Preconditions	The required actions have been defined in the GET_DEFINITION method. If an error occurs during event processing, set C_RESULT to FAILED. This will abort the processing of the event loop along with all subsequent events.
Result	The event is processed.

Parameter name	Parameter Type	Ref.	Data Type
I_FEEDER_HELPER	Importing	X	<MLB>IF_FPM_FH_S
I_EVENT	Importing	X	CL_FPM_EVENT
I_RAISED_BY_OWN_UI	Importing		BOOLE_D

I_LEAD_INDEX	Importing		SYTABIX
I_EVENT_INDEX	Importing		SYTABIX
I_SELECTED_LINES	Importing		RSTABIXTAB
I_UI_INFO	Importing	X	IF_FPM_LIST_ATS_UI_INFO
C_RESULT	Changing		FPM_EVENT_RESULT
C_MESSAGES	Changing		FPMGB_T_MESSAGES

Example: Create New BPON Based on an Existing One

Listing 8-9 shows an example of a PROCESS_EVENT implementation. Properties from an existing node are copied to a new node.

```
data:
  lr_node_data   type ref to data,
  ls_create_data type /pl9/if_gxx_fb01_fpm_it_c=>ts_bpon_flgt_ut_crt_md.
field-symbols:
  <ls_node_data> type /pl9/if_gxx_fb01_fpm_it_c=>ts_bpon_flgt_public_complete.



* 1. perform action only for copy line event
if i raised by own ui = abap true and i_event->mv_event_id = 'CPL1_COPY_LINE'.
* 2. get data from selected node
  lr_node_data  = i_feeder_helper->/pl9/if_gxx_fb01_fpm_fh_r~get_node_data( ).
  assign lr_node_data->* to <ls_node_data>.

* 3. fill create structrue of new node (copy properties)
  ls_create_data-details-from-airport_id = <ls_node_data>-details-from-airport_id.
  ls_create_data-details-to-airport_id   = <ls_node_data>-details-to-airport_id.


* 4. create new BPON for Flight
  i_feeder_helper->create_bpon_flgt( i_data     = ls_create_data
                                     i_crte_ic = /pl9/ce_gxx_fb01_crte_ic=>
                                                                 all_matching ).
endif.
```

Listing 8-9: Create New BPON Based on an Existing One

1. The event ID is checked to make sure it is CPL1_COPY_LINE.
2. The BPON for the selected line is read.

   The data of this BPON serves as the source of the copy operation.
3. From- and to AIRPORT_ID of structure LS_CREATE_DATA is filled based on the selected BPON.
4. A new BPON is created based on LS_CREATE_DATA along with the creation instruction code <MLB>CE_CRTE_IC=>ALL_MATCHING. This CIC tells the POT to look for all flights in the back end that match the properties defined in the BPON.

8.2.6.5 Dynamic Feeder Behavior (Form Feeder)

The signature of the GET_DATA method of the form feeder class (<MLB>CL_FF_<FEEDER_ABBR>_C) looks as follows:

Method Name	GET_DATA
Short Description	Changes field properties; provides additional data
What to do	Dynamically change field properties such as visibility, required entries, display options and value help. Provide additional data in case this is not possible in MAP_MODEL2FEEDER_<ASSIGNMENT_ABBR>. In case of UI-related errors, add the corresponding messages to C_MESSAGES. (All the steps mentioned above can be implemented according to the FPM standard procedures.)
Preconditions	None
Result	Field properties and data are provided.

Parameter name	Parameter Type	Ref.	Data Type
I_FEEDER_HELPER	Importing	X	<MLB>IF_FPM_FH_R
I_EVENT	Importing	X	CL_FPM_EVENT
I_RAISED_BY_OWN_UI	Importing		BOOLE_D
I_SELECTED_FIELDS	Importing		FPMGB_T_SELECTED_FIELDS
I_EDIT_MODE	Importing		FPM_EDIT_MODE
C_MESSAGES	Changing		FPMGB_T_MESSAGES
C_DATA_CHANGED	Changing		BOOLE_D
C_FIELD_USAGE_CHANGED	Changing		BOOLE_D
C_ACTION_USAGE_CHANGED	Changing		BOOLE_D
C_DATA	Changing		DATA
C_FIELD_USAGE	Changing		FPMGB_T_FIELDUSAGE
C_ACTION_USAGE	Changing		FPMGB_T_ACTIONUSAGE

The implementation of the GET_DATA method of the form feeder class is quite similar to that if of the list feeder class. The main difference is that parameters that deal with list properties like visible rows do not matter for form feeders.

8.2.6.6 Dynamic Feeder Behavior (List Feeder)

The signature of the GET_DATA method of the form feeder class (<MLB>CL_FL_<FEEDER_ABBR>_C) looks as follows:

Method Name	GET_DATA
Short Description	Changes field properties; provides additional data
What to do	Dynamically change field properties such as visibility, required entries, display options and value help. Provide additional data in case this is not possible in `MAP_MODEL2FEEDER_<ASSIGNMENT_ABBR>`. In case of UI related errors, add the corresponding messages to `C_MESSAGES`. (All steps mentioned above can be implemented according to the FPM standard procedures.)
Preconditions	None
Result	Field properties and data are provided.

Parameter name	Parameter Type	Ref.	Data Type
I_FEEDER_HELPER	Importing	X	<MLB>IF_FPM_FH_R
I_EVENTID	Importing	X	CL_FPM_EVENT
I_SELECTED_FIELDS	Importing		FPMGB_T_SELECTED_FIELDS
I_RAISED_BY_OWN_UI	Importing		BOOLE_D
I_VISIBLE_ROWS	Importing		I
I_EDIT_MODE	Importing		FPM_EDIT_MODE
I_EXTENDED_CTRL	Importing	X	IF_FPM_LIST_ATS_EXT_CTRL
C_MESSAGES	Changing		FPMGB_T_MESSAGES
C_DATA_CHANGED	Changing		BOOLE_D
C_FIELD_USAGE_CHANGED	Changing		BOOLE_D
C_ACTION_USAGE_CHANGED	Changing		BOOLE_D
C_SELECTED_LINES_CHANGED	Changing		BOOLE_D
C_DND_ATTR_CHANGED	Changing		BOOLE_D
C_ITAB_CHANGE_LOG	Changing	X	IF_SALV_ITAB_CHANGE_LOG
C_DATA	Changing		DATA
C_FIELD_USAGE	Changing		FPMGB_T_FIELDUSAGE
C_ACTION_USAGE	Changing		FPMGB_T_ACTIONUSAGE
C_SELECTED_LINES	Changing		RSTABIXTAB
C_LEAD_INDEX	Changing		SYTABIX
C_FIRST_VISIBLE_ROW	Changing		I
C_ADDITIONAL_INFO	Changing		FPMGB_S_ADDITIONAL_INFO
C_DND_ATTRIBUTES	Changing		FPMGB_T_DND_DEFINITION

Example: Changing the Activation Status of an Event

Listing 8-10 shows an example of a GET_DATA implementation. The activation status of a custom event (which also affects the corresponding button) is set depending on the application mode.

```
data: lr_action_usage type ref to fpmgb_s_actionusage.

* 1. access the action for copying a line using a loop with where condition
loop at c_action_usage reference into lr_action_usage where id = 'CPL1_COPY_LINE'.

* 2. set the activation status of the action
  lr_action_usage->enabled = boolc(
    i_edit_mode <> 'R' and i_edit_mode <> ' ' and lines( c_selected_lines ) = 1 ).
endloop.

* 3. tell the FPM framework that the action usage has changed
c_action_usage_changed = abap_true.
```

Listing 8-10: Changing the Activation Status of an Event

1. The action for copying a line of the list feeder with the ID CPL1_COPY_LINE is accessed using a loop and a where condition.

   It is expected that there is only one loop pass, because the ID CPL1_COPY_LINE is unique.
2. The activation status (LR_ACTION_USAGE->ENABLED) of the event CPL1_COPY_LINE is set.

   The event is enabled only if the UI is in edit mode and exactly one line has been selected.
3. Set the C_ACTION_USAGE_CHANGED parameter to ABAP_TRUE in order to tell the FPM framework to look for changes in the action usage.

8.2.7 Custom Application Controller Implementation

The custom application controller (<MLB>CL_FPM_ACC_C) is used to add custom pushbuttons to the global application toolbar and to handle global events depending on the application configuration.

Moreover, with the custom application controller, a behavior can be implemented that is based on the application configuration (the editing UI supports multiple application configurations):
In the code slots HANDLE_EVENT and SET_BUTTON_PARAMETER, the custom controller (<MLB>IF_FPM_CCTRL) is available as importing parameter that can be used to retrieve the application configuration to which the controller is currently attached.

Finally, the custom application controller is able to enhance the CHIP outport by adding additional outport fields:

- The developer extends the corresponding generated DDIC structure <MLB>CH_O_V_DFT.
- The pushbutton that has been generated by the POB for the event PL1_COMPLETE must be removed from the global application toolbar.
- Instead of this button, a new custom pushbutton with a custom event ID must be added to the application toolbar.
- To fill the additional fields in the outport data DDIC structure, the HANDLE_EVENT method must be implemented (which reacts on the new custom event).

At the end of the `HANDLE_EVENT` method, the original final CHIP event `PL1_COMPLETE` (which has been removed and is no longer attached to a button) is then triggered programmatically by calling the `RAISE_FINISH_EVENT` method of the custom controller (see section 8.2.4.1).

8.2.7.1 Event Handling

The signature of the HANDLE_EVENT method of the custom application controller class (<MLB>CL_FPM_ACC_C) looks as follows:

Method Name	HANDLE_EVENT
Short Description	Handles custom event on application level
What to do	Handle custom event (e.g. an event raised from the global toolbar). Add additional data to the default CHIP outport, if required. In case of critical UI-related exceptions, raise an exception of type <MLB>CX_IM to abort the processing of the current event loop and of all subsequent events. In case of errors, raise an error message using I_MESSAGE_HELPER.
Preconditions	None
Result	The event is processed.

Parameter name	Parameter Type	Ref.	Data Type
I_EVENT	Importing	X	CL_FPM_EVENT
I_CUSTOM_CONTROLLER	Importing	X	<MLB>IF_FPM_CCTRL
I_DATA_HELPER	Importing	X	<MLB>IF_FPM_CDTHS
I_MESSAGE_HELPER	Importing	X	<MLB>IF_FPM_MSG_H
	Exception	X	<MLB>CX_IM

Example: Performing Check and Execute With One Click

Usually, two clicks are required to first check and then execute a PO. This example explains how to perform these two actions with only one click using a custom event.

Listing 8-11 shows an example of a HANDLE_EVENT implementation for this scenario: If the CPL1_CHECK_EXECUTE action is triggered (user clicks on the custom button), the check as well as the execute service operation are performed , one after the other.

```
data: ls_check_result type /pl9/if_gxx_fb01_fpm_tac=>ty_check_result,
      ls_execute_log  type /pl9/if_gxx_fb01_fpm_tac=>ty_log.

* 1. evaluate the event id
if i event->mv event id = 'CPL1 CHECK_EXECUTE'.
* 2. call the check service operation
  ls check result = i custom controller->check bpo( ).
* 3. display result of the check on the screen
  i_message_helper->report_messages_by_log( ls_check_result-log ).

* 4. if the check was successful, call the execute service operation
  if ls_check_result-successfull = abap_true.
    ls execute log = i custom controller->execute_bpo( ).
* 5. display the log from the service operation
    i_message_helper->report_messages_by_log( ls_execute_log ).
  endif.
endif.
```

Listing 8-11: Handling the CHECK_EXECUTE Event

1. It is checked that the event ID equals `CPL1_CHECK_EXECUTE`.
2. The check service operation is performed for the BPO using the custom controller (available via parameter `I_CUSTOM_CONTROLLER`).
3. In order to display the log of the check service operation on the screen, the helper for UI messages (`I_MESSAGE_HELPER`) is used.
4. If the check was successful, the execute service operation is called using the custom controller (`I_CUSTOM_CONTROLLER`)
5. The helper for UI messages (`I_MESSAGE_HELPER`) is used to display the log of the execute service operation on the screen.

   Note that the execution itself is performed asynchronously in the background by the POT.

8.2.7.2 Setting Parameters for Global Buttons

The signature of the `SET_BUTTON_PARAMETER` method of the custom application controller class (`<MLB>CL_FPM_ACC_C`) looks as follows:

Method Name	SET_BUTTON_PARAMETER
Short Description	Sets button parameters
What to do	Set visibility and activation status of buttons. In case of critical UI-related exceptions, raise an exception of type `<MLB>CX_IM` to abort the processing of the current event loop and of all subsequent events.
Preconditions	None
Result	The button parameters are correctly set.

Parameter name	Parameter Type	Ref.	Data Type
`I_PARAMETER`	Importing	X	`<MLB>IF_FPM_TAC=>` `TY_SET_BUTTON_PARAMETER_IN`
`I_FPM_EVENT_ID`	Importing	X	`<MLB>IF_FPM_TAC=>TY_FPM_EVENT_ID`
`I_DATA_HELPER`	Importing	X	`<MLB>IF_FPM_CDTHR`
`R_BUTTON_PARAMETER`	Returning	X	`<MLB>IF_FPM_TAC=>` `TY_SET_BUTTON_PARAMETER_RESULT`
	Exception	X	`<MLB>CX_IM`

Example: Setting Parameters for the Custom “Check-Execute” Button

Listing 8-12 shows an example of a `SET_BUTTON_PARAMETER` implementation. The activation status of the “Check-Execute” Button is set depending on the application mode and the process object instance status.

```
r_button_parameter-enabled    = abap_true.
r_button_parameter-visibility = abap_true.

* 1. evaluate the event id
if i_fpm_event_id = 'CPL1_CHECK_EXECUTE'.

* 2. set the activation status of the button
  r_button_parameter-enabled = boolc(
    i_parameter-is_display_mode = abap_true and
    i_parameter-bpo_status = /pl9/ce_gxx_fb01_node_sc=>unchecked ).
endif.
```

Listing 8-12: Setting Parameters for the “Check-Execute” Button

1. It is checked that he event id equals `CPL1_CHECK_EXECUTE`.
2. The activation status of the button is set depending on the node status code and the application mode:

   The button is active if the application is in display mode (`I_PARAMETER-IS_DISPLAY_MODE`) and the node type status is *Unchecked* (`<MLB>CE_NODE_SC=>UNCHECKED`).

8.2.8 Custom Implementation of Navigation and Data Creation

The code slots `GET_LEAD_INDEX_BY_PATH` and `SET_DEFAULT_LINES_FOR_PATH` can be used to automate navigation and data creation in the background in cases where there is no list feeder assigned to a table node.

8.2.8.1 Feeder Path

The data of a PO instance has a hierarchical structure (see below) and the UI reflects this structure (a form feeder is assigned to a flat structure and a list feeder is assigned to a table). In this (default) case, a developer does not need to implement any navigation because the generated part of editing UI takes care of the navigation through the hierarchical structures (navigation by lead selection of the user and wiring between the GUIBBs).

However, when the data model assignments are defined in the Editing UI Wizard, it is possible to omit node assignments for locations where no data is to be displayed on the UI (to simplify the navigation for the user). In case list feeders are left out for a table node, the model is not accessible by the UI at this node.

This means that tasks that are usually done by the user need to be done automatically in the background.

This includes:

- Selecting a table node line for display in a form feeder
- Adding a line to a table node

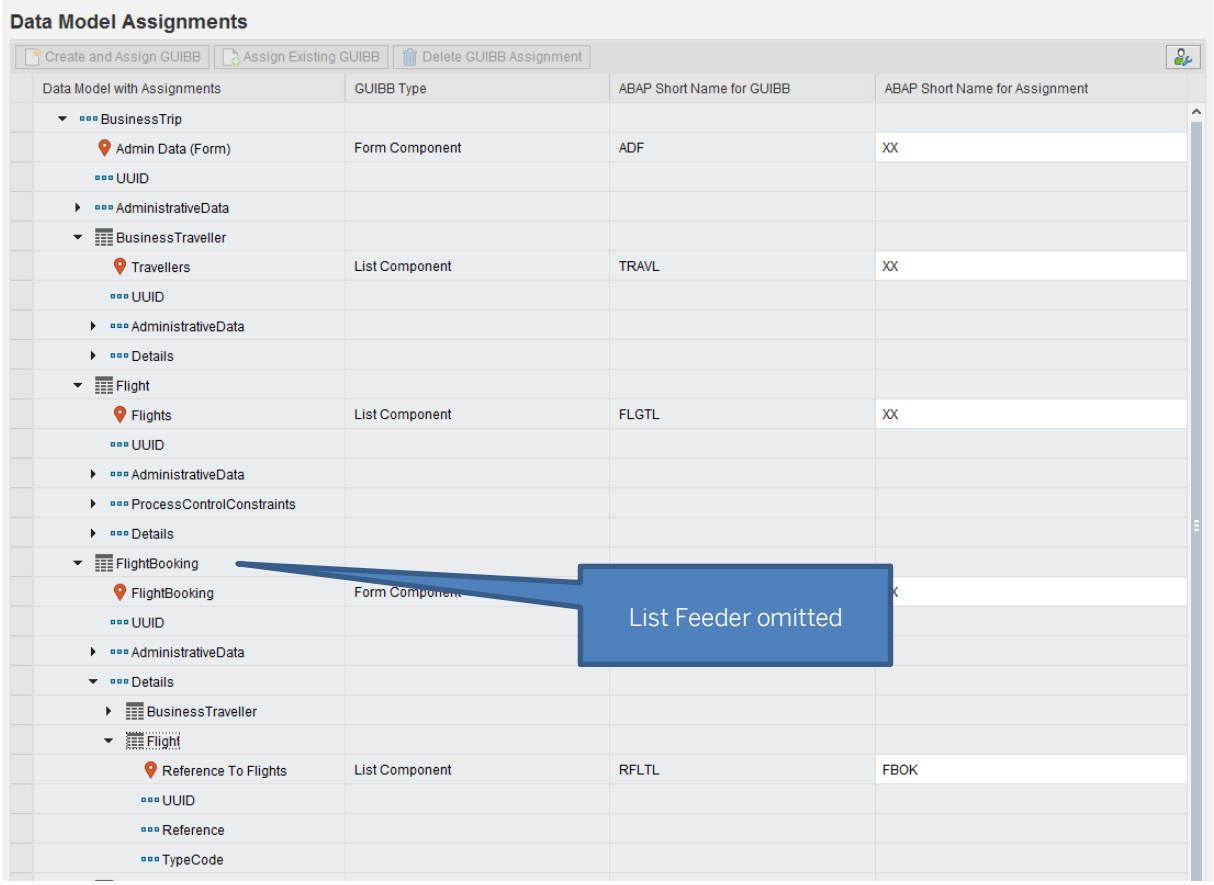


Figure 114 Data Model With Assigned List and Form Feeder

The code slots `GET_LEAD_INDEX_BY_PATH` and `SET_DEFAULT_LINES_FOR_PATH` of the table controller class `<MLB>CL_TCB_V_<ABBR_APCF>` can be used to set the lead index/add additional lines depending on the path (`<ABBR_APCF>` – abbreviation for application configuration).

The path is built as a string according to the following pattern:

`<ROOT_ABBR>-BPON_<BPON_ABBR>-<ITPATH>`

`<ROOT_ABBR>:`
ABAP Short Name for Root Node

`<BPON_ABBR>:`
ABAP Short Name for BPON

`<ITPATH>:`

Path according to internal types for BPON `<MLB>FPM_IT_C=>TS_BPON_<ASBPON>_PUBLIC_COMPLETE`.

8.2.8.2 Automatic Navigation

If no list feeder is assigned to a table node of the model, there is no possibility for the user to set the lead selection for the active table entry on the screen. With the `GET_LEAD_INDEX_BY_PATH` method, lead selection in the background can be set.

The signature of the `GET_LEAD_INDEX_BY_PATH` method of the default table callback class (`<MLB>CL_TCB_V_<ABBR_APCF>`) looks as follows:

Method Name	GET_LEAD_INDEX_BY_PATH
Short Description	Provides the lead index depending on the model path of a table node
What to do	Evaluate the path, ID and parameters of the calling feeder (`I_CALLING_FEEDER`). Set the lead index depending on the results of the feeder evaluation. Hint: In a newly generated `GET_LEAD_INDEX_BY_PATH` code slot, the relevant paths for table nodes in the model (those with no list feeder assigned) are generated as code comments.
Preconditions	No list feeder is assigned to the list node or the method `IS_CUSTOM_IMPL_NEEDED` has returned `true`.
Result	The lead index is provided.

Parameter name	Parameter Type	Ref.	Data Type
`I_CALLING_FEEDER`	Importing		`<MLB>IF_FPM_TAC=>TY_CALLING_FEEDER`
`I_DATA`	Importing		`DATA`
`R_LEAD_INDEX`	Returning		`I`

Example: Automatically Set the Lead Index for Shuttle Booking - Flight

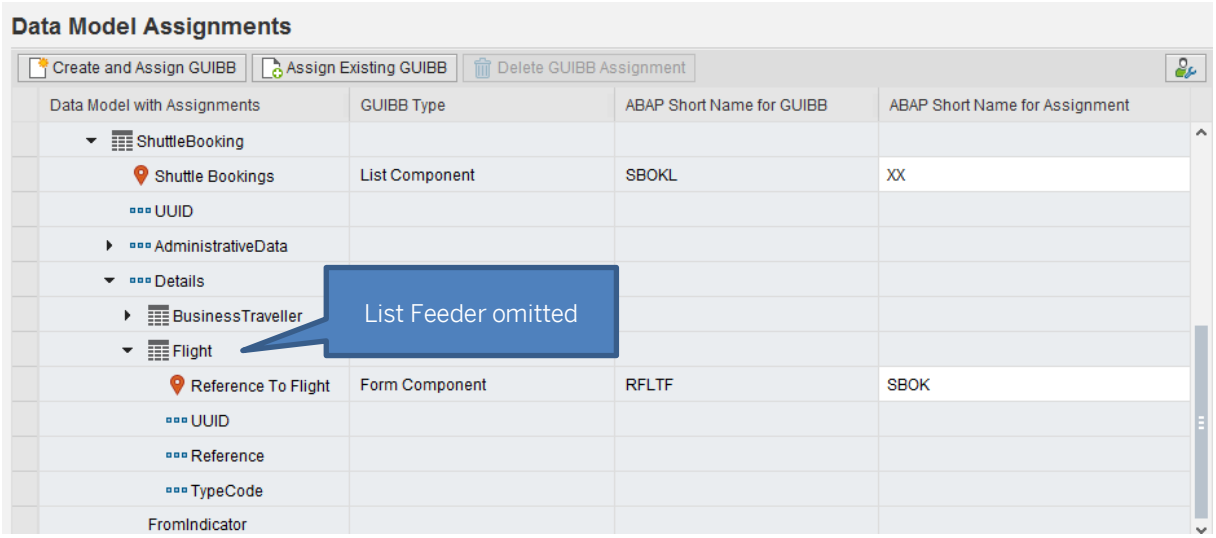


Figure 115 Data Model: Form Feeder With Omitted Corresponding List Feeder

Listing 8-13 shows an example for a `GET_LEAD_INDEX_BY_PATH` implementation.

```
  field-symbols:
    <lt_bpon>
      type /pl9/if_gxx_fb01_fpm_it_d=>/pl9/wfitbusiness_trip_fs_cros.

* 1. Evaluate the path of the calling feeder
  assert i_calling_feeder-path = 'FB-BPON_SBOK-DETAILS-FLIGHT'.

  r_lead_index = 0.
```

```
* 2. Assign i_data to the concreta data type for the shuttle booking -> flight
  assign i data to <lt bpon>.
* 3. Set the lead index to 1 in case the list feeder table is not empty
  if lines( <lt_bpon> ) > 0.
    r_lead_index = 1.
  endif.
```

Listing 8-13: Automatically set the Lead Index for Shuttle Bookings - Flight

1. The path of the calling feeder is evaluated.

   There are specific patterns for the imported values (see section 8.2.8.1 on feeder paths).

   In this case, the path `FB-BPON_SBOK-DETAILS-FLIGHT` is relevant.
2. The feeder data (provided as general `DATA` type via importing parameter `I_DATA)` is converted to the specific data type for the node `/PL9/IF_GXX_FB01_FPM_IT_D=>/PL9/WFITBUSINESS_TRIP_FS_CROS`.
3. If the table contains data, the lead index is set to the first line.

8.2.8.3 Automatic Data Creation

If no list feeder is assigned to a table node of the model, there is no possibility for the user to create entries for this node on the screen. With the `SET_DEFAULT_LINES_FOR_PATH` method, such table node entries can be created automatically in the background.

The signature of the `SET_DEFAULT_LINES_FOR_PATH` method of the form feeder class (`<MLB>CL_TCB_V_<ABBR_APCF>`) looks as follows:

Method Name	SET_DEFAULT_LINES_FOR_PATH
Short Description	Inserts default lines into a table node depending on the path
What to do	Evaluate the path of the calling feeder (`I_PATH`). Update the data of the table node (`C_TABLE`), if required.
Preconditions	No list feeder is assigned to the list node or method `IS_CUSTOM_IMPL_NEEDED` has returned `true`.
Result	Default lines are filled.

Parameter name	Parameter Type	Ref.	Data Type
`I_PATH`	Importing		`<MLB>IF_FPM_TAC=>TY_FEEDER_LOCATION`
`C_TABLE`	Changing		`DATA`

Example: Automatically Create Data for Flight Booking

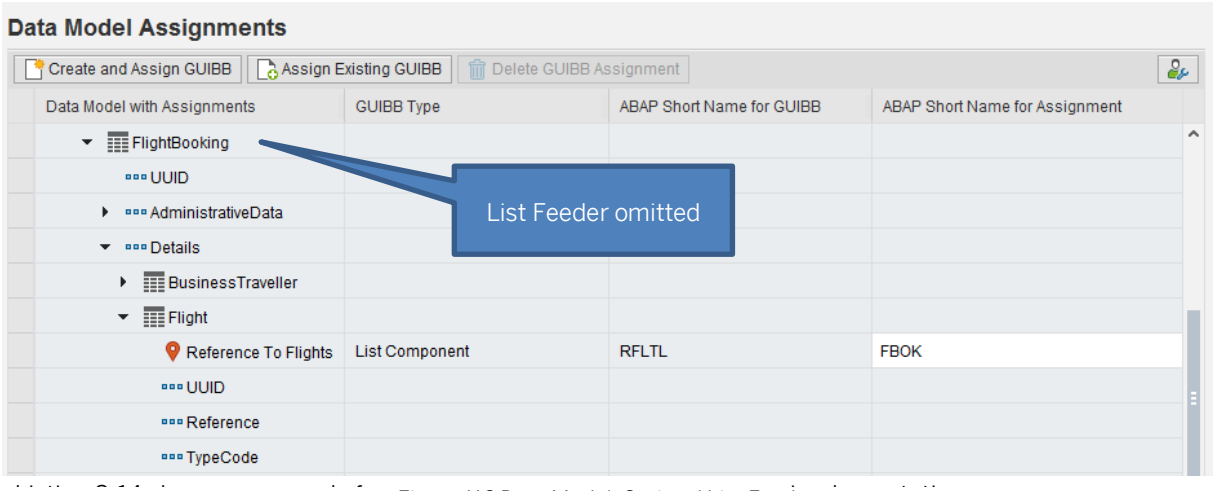


Figure 116 Data Model: Omitted List Feeder

```
  data:
    lr_bpon
      type ref to /pl9/if_gxx_fb01_fpm_it_c=>ts_bpon_fbok_public_complete.
  field-symbols:
    <lt_bpon>
      type /pl9/if_gxx_fb01_fpm_it_c=>tt_bpon_fbok_public_complete.

* 1. Evaluate the path of the feeder location
  assert i_path = 'FB-BPON_FBOK'.

* 2. Assign c_table to the concrete data type for the flight booking
  assign c_table to <lt_bpon>.

* 3. Add initial line to table flight booking
  append initial line to <lt_bpon> reference into lr_bpon.

* 4. Fill line with data
* lr_bpon->…  = …
```

Listing 8-14: Automatic Data Creation for Flight Bookings

1. The path of the feeder location is evaluated.

   There are specific patterns for the imported values (see section 8.2.8.1 on feeder paths).

   In this case, the path `FB-BPON_FBOK` is relevant.
2. The feeder data (provided as general `DATA` type via importing parameter `C_TABLE`) is converted to the specific data type for the node `<MLB>IF_FB01_FPM_IT_C=>TT_BPON_FBOK_PUBLIC_COMPLETE`.
3. A line is added to the table.
4. The table line is filled with data.

8.2.8.4 Enforcing Automatic Navigation and Data Creation

In the configurations generated by the POB, the methods `GET_LEAD_INDEX_BY_PATH` and `SET_DEFAULT_LINES_FOR_PATH` described above are called if no list GUIBBs are assigned for a node. These code slots are not called for any manually created configurations.

The POT developer can overwrite this path-dependent default behavior using the code slot of method `IS_CUSTOM_IMPL_NEEDED`, and determine if the methods listed above should be called for that specific path.

The signature of the `IS_CUSTOM_IMPL_NEEDED` method of the form feeder class (`<MLB>CL_TCB_V_<ABBR_APCF>`) looks as follows:

Method Name	IS_CUSTOM_IMPL_NEEDED
Short Description	Overwrites default behavior for methods `GET_LEAD_INDEX_BY_PATH` and `SET_DEFAULT_LINES_FOR_PATH`
What to do	Enforce the calling of the methods `GET_LEAD_INDEX_BY_PATH` and `SET_DEFAULT_LINES_FOR_PATH` for a path by updating `C_RESULT`.
Preconditions	None
Result	The overwritten behavior is returned.

Parameter name	Parameter Type	Ref.	Data Type
`I_PATH`	Importing		`<MLB>IF_FPM_TAC=>TY_FEEDER_LOCATION`
`C_RESULT`	Changing		`ABAP_BOOL`

Example: Enforce Automatic Navigation and Data Creation for Shuttle Booking

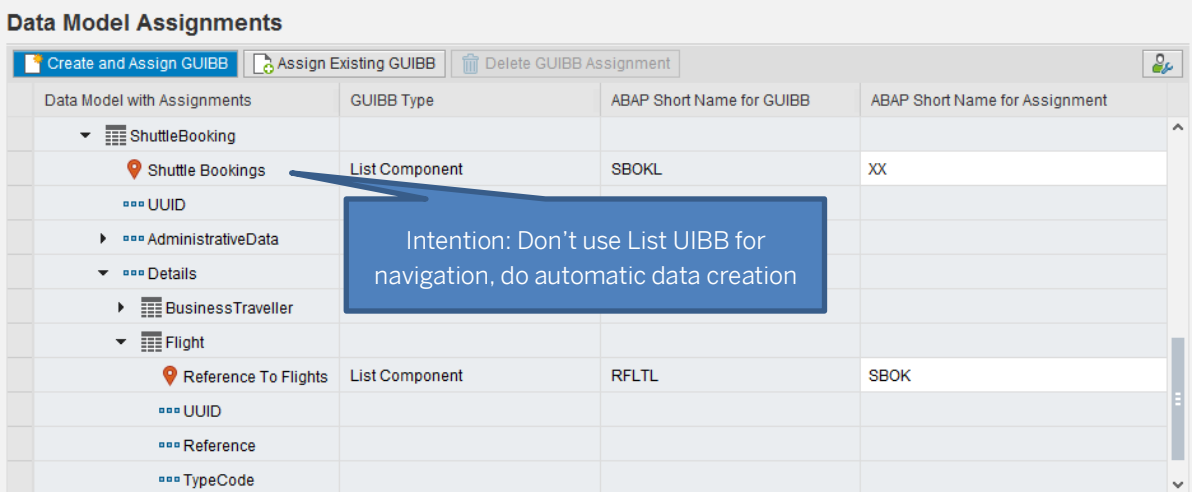


Figure 117 Data Model: List Feeder not Intended for Manual Navigation

Listing 8-15 shows an example for an `IS_CUSTOM_IMPL_NEEDED` implementation.

```
c_result = boolc( i_path = 'FB-BPON_SBOK' ).
```

Listing 8-15: Enforce Automatic Navigation and Data Creation for Shuttle Booking

8.2.9 Adaption of the generated FPM configurations

The generated FPM configurations cannot be changed directly. All adaptions have to be done using the FPM enhancement concept. This assures that your adaptions are not overwritten if the FPM configurations are re-generated using the Editing UI Wizard. In the SAP Help Portal for Floor Plan Manager for Web Dynpro ABAP it is described how to proceed.

Terms and Abbreviations

Term / Abbreviation	Full Name
ADK	Archive Development Kit
BAdI	Business Add-In
BRFplus	Business Rules Framework Plus
CIC	Creation Instruction Code (short for GDT `RelatedObjectExistenceAssumptionNodeCreationInstructionCode`)
CNS	Change and Notification Service
ECH	Error and Conflict Handler
ESR	Enterprise Services Repository
FSL	Financial Services Library
GDT	Global Data Type
ILM	Information Lifecycle Management
OAF	Outbound Agent Framework
POB	SAP Process Object Builder
POT	Process Object Type
PPO	Post Processing Office
PSJ	Process Step Journal
Result Code	Short for GDT `BusinessDocumentProcessingResultCode`
SAP BPM	SAP Business Process Management
SAP PI	SAP Process Integration
Severity Code	Short for GDT `LogItemSeverityCode`
SIW	Service Implementation Workbench

Table of Figures

Figure 1: Typical IT Landscapes and Integration Challenges 13
Figure 2: Motivation for Process Object Types 14
Figure 3: Travel Booking 18
Figure 4: Model-Driven Software Development in the POB/POT Context 20
Figure 5: BPMN Model of the Travel Booking Process 23
Figure 6: SAP SOA Modeling Methodology Metamodel 25
Figure 7: Business Object Map for Sample Process (SAP SOA Modeling Methodology) 26
Figure 8: Metamodel of a POT 27
Figure 9: Preliminary Object Model for `FlightAndShuttleBooking` POT 28
Figure 10: Abstract POT Model Template 30
Figure 11: Detailed Object Model for sample POT `FlightAndShuttleBooking` 31
Figure 12: High Level Phase Model of a POT 40
Figure 13: Status Transitions and Calculation of Aggregated POT Status 41
Figure 14: POT Status Transitions Overview 42
Figure 15: Legend for Sequence Diagrams 43
Figure 16: *Create* phase – Dynamic View 44
Figure 17: *Check* Phase – Dynamic View 46
Figure 18: *Execute* Phase – Dynamic View 48
Figure 19: POT Static Structure Overview 52
Figure 20: ABAP Environment of a POT 54
Figure 21: POT Core Package Structure 55
Figure 22: POT Package Structure – Editing UI View 56
Figure 23: Block Diagram of POT Layer Concept 63
Figure 24: Application Façade 64
Figure 25: Application Facade for Operation CheckExecute - Call Sequence 65
Figure 26: BPO API - Static View 66
Figure 27: Specific Details of the BPO API - Static View 68
Figure 28: BPON API - Static View 69
Figure 29: Versioning – Static View 70
Figure 30: Message Persistence - Static View 71
Figure 31: Create Phase Helper – Static View 72
Figure 32: *Create* Phase for `Create<BPO>` - Call Sequence 73
Figure 33: *Create* phase for `Create<BPON>` - Call Sequence 73
Figure 34: Check Phase Helper – Static View 74
Figure 35: *Check* Phase for Operation Check<BPO> - Overall Call Sequence 75
Figure 36: BPON-related Checks - Detailed Call Sequence 76
Figure 37: BPO-related Checks - Detailed Call Sequence 77
Figure 38: Execute Phase Helper – Static View 78
Figure 39: Execute Phase Triggered by Operation `Execute<BPO>` - Call Sequence 79

Figure 40: Asynchronous Execution via RFC – Overall Call Sequence 79
Figure 41: Asynchronous Execution via RFC – Detailed Call Sequence 80
Figure 42: BPON Facade - Static View 80
Figure 43: Processing of Incoming Asynchronous Confirmations – Call Sequence 81
Figure 44: Service Implementation Layers 82
Figure 45: Overview of the Services Provided by a POT 82
Figure 46: POT Standard Synchronous Inbound Service Implementation - Static View 83
Figure 47: Synchronous POT Standard Inbound Service (Request/Confirmation) – Call Sequence 84
Figure 48: POT Inbound Services (Asynchronous) static view 85
Figure 49: Synchronous POT Standard Inbound Service (Request/Confirmation) – Call Sequence 86
Figure 50: Back-End Related Asynchronous Inbound Service Implementation – Static View 87
Figure 51: Back-End Related Asynchronous Inbound Service – Call Sequence (Success) 88
Figure 52: Back-End Related Asynchronous Inbound Service – Call Sequence (Failure) 88
Figure 53: Successful Retry - Call Sequence 89
Figure 54: Bulk Splitter – Static View 90
Figure 55: Back-End-Related Synchronous Outbound Service – Static View 91
Figure 56: Execution of Back-End-Related Synchronous Outbound Service - Call Sequence 92
Figure 57: Asynchronous Outbound Service – Static View 93
Figure 58: Asynchronous Outbound Service – Call Sequence 94
Figure 59: ABAP Channel - static view 96
Figure 60: Main Page of a Generated Editing UI 97
Figure 61: Architecture of the Generated Editing UI 97
Figure 62: Initial Screen of the Editing UI 99
Figure 63: Main Page of an Editing UI 100
Figure 64: Subpage of an Editing UI 100
Figure 65: Feeders for Search – Static View 103
Figure 66: Application Controller – Static View 103
Figure 67: Form and List Feeders – Static View 104
Figure 68: Search – Call Sequence 105
Figure 69: Search Result Retrieval – Call Sequence 106
Figure 70: Custom Feeder Definitions – Call Sequence 106
Figure 71: Display Data on the Editing UI – Call Sequence 107
Figure 72: Flush After End of Roundtrip – Call Sequence 108
Figure 73: Saving Data from the Editing UI – Call Sequence 108
Figure 74: Handling Application Level Events – Call Sequence 109
Figure 75: Handling Button Status on Application Level – Call Sequence 110
Figure 76: Handling Feeder Events – Call Sequence 110
Figure 77: POT Transaction Control – Static View 112
Figure 78: POT Exception Hierarchy – Static View 113
Figure 79: Private Status of a BPO 115
Figure 80: POT Status Calculator – Static View 116
Figure 81: Order of Statuses for Overall Status Calculation 116
Figure 82: Raising of Events - Static View 117
Figure 83: Event Mediator - Static View 118

Figure 84: Cross-Reference Handling – Static View 119
Figure 85: Cross-Reference Check Helper – Call Sequence 119
Figure 86: Process Observer Integration – Static View 120
Figure 87: Process Observer Integration into Outbound Service Implementation – Call Sequence 121
Figure 88: PMA Events on Status Change of BPO – Call Sequence 121
Figure 89: PMA BAdI for UI Navigation 122
Figure 90: POT Archiving – Static View 123
Figure 91: Archiving Run – Call Sequence 124
Figure 92: Archivability Check (Standard, Custom and BAdI) – Call Sequence 125
Figure 93: Check of Residence Time – Call Sequence 125
Figure 94: Archive Search UI 126
Figure 95: Instance Data as XML 127
Figure 96: Phase Implementation 129
Figure 97: POL-specific Pattern for Raising Exceptions 131
Figure 98: Activity Flow of *Create* Phase Code Slot 142
Figure 99: Activity Flow of *Check* Phase Code Slot 150
Figure 100: Check BAdI 153
Figure 101: BRFplus Check Wrapper and Mapper 155
Figure 102: BRFplus Function `CHECK_BPO` 157
Figure 103: Accessing Input Data in BRFplus 157
Figure 104: Raising an Error Message in BRFplus 158
Figure 105: Activity Flow of the *Execute* Phase Code Slot (Synchronous Service) 160
Figure 106: Activity Flow of the *Execute* Phase Code Slot (Asynchronous Service) 164
Figure 107: Activity Flow for the Post-Processing of Asynchronous Messages 168
Figure 108: Bulk Splitter 170
Figure 109: Create External Definition in ESR 177
Figure 110: Select WSDL-file 178
Figure 111: Creation of service interface for WSDL-based service 178
Figure 112: Complete ESR counterpart for WSDL-based service 179
Figure 113: GUIBBs and Feeders - Basic Principles 186
Figure 114 Data Model With Assigned List and Form Feeder 211
Figure 115 Data Model: Form Feeder With Omitted Corresponding List Feeder 212
Figure 116 Data Model: Omitted List Feeder 214
Figure 117 Data Model: List Feeder not Intended for Manual Navigation 215

Table of Tables

Table 4-1: Values of <BPON>StatusCode 40
Table 6-1: Database tables of a POT 67
Table 6-2: Pushbuttons on the Global Application Toolbar 101
Table 6-3: Pushbuttons on the GUIBBs 102
Table 6-4: Private Statuses of BPO set by BPON API 115
Table 7-1: Read and Write Accesses in the Phase Implementation 134
Table 7-2: Important GDTs 135
Table 7-3: Values of the Creation Instruction Code 136
Table 7-4: Methods of the Phase Helper 137
7-5: Methods of the Phase Correlation Helper 137
7-6: Methods of the PIMPL Correlation Helper 138
Table 7-7: Methods of the Log Helper 139
Table 7-8: Methods of the FSL Message Helper 140
Table 7-9: Methods of the FSL Message Header Helper 141
Table 7-10: Interfaces for Internal Types 175
Table 7-11: Important BPO Types 176
Table 7-12: Important BPON Types 176
Table 8-1: Scenarios for Extending the Editing UI 184
Table 8-2: Methods Called During Feeder Initialization 184
Table 8-3: A Round Trip of the Editing UI 185
Table 8-4: Methods of the Custom Controller 190
Table 8-5: Custom Data Helper – Read Methods 190
Table 8-6: Custom Data Helper – Set Methods 191
Table 8-7: Feeder Helper – Read Methods 192
Table 8-8: Feeder Helper – Set Methods 192
Table 8-9: Methods of the Helper for UI Messages 193

Table of Listings

Listing 7-1: Source Code Example 128
Listing 7-2: Transferring Back-End Errors to the Provider Log 131
Listing 7-3: Writing an Error Message into the Process Log 132
Listing 7-4: Raising a Custom Exception 133
Listing 7-5: Data Declarations of the *Create* Phase 142
Listing 7-6: Code Slot of the *Create* Phase 145
Listing 7-7: Code Slot of the *Create* Phase with CIC Overwrite 148
Listing 7-8: Data Declarations of the *Check* Phase 150
Listing 7-9: Code Slot of the *Check* Phase 152
Listing 7-10: Private Details in the *Check* Phase 153
Listing 7-11: Code Slot of the *Execute* Phase with Synchronous Service 162
Listing 7-12: Call of PIMPL Correlation Helper 163
Listing 7-13: Code Slot of the *Execute* Phase with Asynchronous Service 166
Listing 7-14: Code Slot of an Asynchronous Confirmation 168
Listing 7-15: Call of Correlation Helper 169
Listing 7-16: Code Slot of a Bulk Splitter 171
Listing 7-17: Additional Parameters for Archiving 172
Listing 8-1: Mapping Additional Search Criteria 181
Listing 8-2: Mapping Additional Search Result Fields to the Search Feeder 182
Listing 8-3: Feeder Initialization 194
Listing 8-4: Mapping Model Data to Feeder Data (Form Feeder) 194
Listing 8-5: Mapping Model Data to Feeder Table (List Feeder) 196
Listing 8-6: An Empty Code Slot Implementation 197
Listing 8-7: Mapping Data from a List Feeder to the Model 198
Listing 8-8: Adding a Custom Event 201
Listing 8-9: Create New BPON Based on an Existing One 203
Listing 8-10: Changing the Activation Status of an Event 206
Listing 8-11: Handling the CHECK_EXECUTE Event 209
Listing 8-12: Setting Parameters for the “Check-Execute” Button 210
Listing 8-13: Automatically set the Lead Index for Shuttle Bookings - Flight 213
Listing 8-14: Automatic Data Creation for Flight Bookings 214
Listing 8-15: Enforce Automatic Navigation and Data Creation for Shuttle Booking 216

