Living Systems® Process Suite

Modeling

Living Systems Process Suite Documentation

3.1
Tue Jan 12 2021

Whitestein Technologies AG | Hinterbergstrasse 20 | CH-6330 Cham
Tel +41 44-256-5000 | Fax +41 44-256-5001 | http://www.whitestein.com

Copyright © 2007-2021 Whitestein Technologies AG
All rights reserved.

Copyright © 2007-2021 Whitestein Technologies AG.

This document is part of the Living Systems® Process Suite product, and its use is governed by the corresponding license agreement. All rights reserved.

Whitestein Technologies, Living Systems, and the corresponding logos are registered trademarks of Whitestein Technologies AG. Java and all Java-based trademarks are trademarks of Oracle and/or its affiliates. Other company, product, or service names may be trademarks or service marks of their respective holders.

Contents

1 Business Process Modeling **1**

1.1 Running Process Design Suite . . . 2

1.2 Checking of Errors in Process Design Suite . . . 3

2 Designing Models **5**

2.1 Model Structure . . . 6

2.1.1 GO-BPMN Projects . . . 6

2.1.1.1 Creating GO-BPMN Projects . . . 6

2.1.1.2 Closing GO-BPMN Projects . . . 7

2.1.1.3 Referencing Projects . . . 7

2.1.2 GO-BPMN Modules . . . 8

2.1.2.1 Creating GO-BPMN Modules . . . 8

2.1.2.2 Importing Modules . . . 10

2.1.3 Definitions and Configurations . . . 13

2.1.3.1 Creating Definitions and Configurations . . . 14

2.1.3.2 Opening Resource Files . . . 14

2.2 Generic Modeling Mechanisms . . . 15

2.2.1 Writing Expressions . . . 16

2.2.1.1 Testing Expressions . . . 16

2.2.2 Defining Properties . . . 17

2.2.3 Disabling Elements . . . 19

2.2.3.1 Defining Formatting of Disabled Elements . . . 22

2.2.4 Working with Diagrams . . . 22

2.2.4.1 Creating a Diagram . . . 24

2.2.4.2 Inserting Element Views 24
2.2.4.3 Aligning Element Views 25
2.2.4.4 Matching Element Views Size 26
2.2.4.5 Spreading Diagram Element Views 26
2.2.4.6 Changing the Process Element Type 26
2.2.4.7 Deleting Elements from Diagrams 27
2.2.4.8 Snapping to Grid 27
2.2.4.9 Customizing the Palette 27
2.2.4.10 Displaying and Hiding Page Borders in Diagrams 28
2.2.4.11 Limiting Diagram Frame Nesting Level 29
2.2.4.12 Inserting Hyperlinks 29
2.2.4.13 Switching Iconic and Decorative Notation 30
2.2.4.14 Hiding and Displaying Compartments of Diagram Elements 31
2.2.4.15 Changing Line Style 31
2.2.4.16 Formatting Settings 33
2.2.4.17 Diagram Printing 37
2.2.4.18 Applying Automatic Layout in Diagrams 41
2.2.5 Comparing Resources 42
2.2.5.1 Comparing Two Resources 42
2.2.5.2 Comparing Three Resources 44
2.2.5.3 Comparing Version-Controlled Resources 46
2.2.5.4 Merging 46
2.2.6 Modeling Status 46
2.2.6.1 Setting a Modeling Status of an Element 47
2.2.6.2 Defining a Modeling Status 47
2.2.6.3 Exporting and Importing a Modeling Status 48
2.2.6.4 Disabling Presentation of a Modeling Status 48
2.2.7 Todo and Task Markers 49
2.2.8 Validation 51
2.2.8.1 Configuring Validation 52

2.2.8.2 Validating Old Modules 53
2.2.8.3 Hiding Validation Markers 53
2.2.9 Search 53
2.2.9.1 Searching for GO-BPMN Entities and their Usage 54
2.2.9.2 Searching for Elements 55
2.2.9.3 Searching for Element Usages 55
2.2.9.4 Searching for Dependent Tasks 55
2.2.9.5 Searching for Call Hierarchies 55
2.3 Processes 56
2.3.1 Creating a Process 56
2.3.2 Defining Process Parameters 57
2.3.3 Defining Process Variables 58
2.3.4 Modeling a Process 58
2.3.4.1 Changing Task Type 58
2.3.4.2 Removing Invalid Task Parameters 58
2.3.4.3 Extracting Process Elements into a Reusable Sub-Process 58
2.3.4.4 Linking Goal to Goal Diagram 59
2.4 Variables 59
2.4.1 Global Variable 60
2.4.2 Process, Forms, and Sub-Process Variable 61
2.4.3 Local Variable 61
2.5 Organization Models 61
2.5.1 Creating an Organization Model 61
2.5.2 Assigning a Role or Organization Unit to a Person 62
2.6 Documents 63
2.6.1 Defining a Document 63
2.6.2 Navigate Away from Document 64
2.6.2.1 Creating a Model Instance and Navigating to its To-Do on Document Submit . . 65
2.7 Forms 66
2.7.1 Form Execution Levels 68

2.7.2 Event Processing 69

2.7.2.1 Events 72

2.7.2.2 Listeners 80

2.7.3 Creating Forms 81

2.7.3.1 Defining Form Parameters 82

2.7.3.2 Defining Form Variables 82

2.7.3.3 Designing Form Content 82

2.7.3.4 Validating Form Data 90

2.7.3.5 Reusing Forms 95

2.7.3.6 Modifying Presentation of Components 105

2.7.3.7 Creating Mobile Forms 111

2.7.4 Form Components 112

2.7.4.1 Container Components 112

2.7.4.2 Input Components 119

2.7.4.3 Output Components 133

2.7.4.4 Action Components 160

2.7.4.5 Special Form Components 161

2.7.4.6 Text Annotations and Associations 165

2.7.4.7 Deprecated Components 165

2.7.5 Form Patterns 168

2.7.5.1 Editable Table 168

2.7.5.2 Table with Derived Values 170

2.7.5.3 Calendar with Adding Entries Functionality 172

2.7.5.4 Pop-up with Save and Cancel Buttons 176

2.7.6 Enabling Error Reporting on Components 180

2.8 Localization 180

2.8.1 Creating Localization Identifiers in the Localization Editor 181

2.8.2 Creating and Calling Localization Identifier in the Expression Editor 182

2.8.3 Calling Localization Identifiers 182

2.8.4 Searching for Usages of Localization Identifiers 183

2.8.5 Identifying Unlocalized Strings . . . 183
2.9 Functions . . . 183
2.9.1 Defining Functions . . . 184
2.9.2 Calling Functions . . . 185
2.10 Queries . . . 185
2.10.1 Standard Queries . . . 186
2.10.1.1 Filtering Results in Standard Queries . . . 187
2.10.1.2 Ordering in Standard Queries . . . 188
2.10.1.3 Generating Queries for Shared Records . . . 190
2.10.2 Native Queries . . . 191
2.10.3 Calling Queries . . . 192
2.11 Data Type Model . . . 192
2.11.1 Creating a Record . . . 192
2.11.2 Creating a Record Field . . . 193
2.11.3 Creating a Record Subtype . . . 194
2.11.4 Creating a Record Relationship . . . 195
2.11.5 XML Mapping in a Data Type Model . . . 198
2.11.6 Creating a Record Composition . . . 198
2.11.7 Comparing Records . . . 199
2.11.7.1 Defining Fields Used in Record Comparisons . . . 199
2.11.7.2 Comparing Records with Fields on Related Records . . . 200
2.11.8 Presentation of Record Diagrams . . . 200
2.11.8.1 Displaying and Hiding Record Compartments . . . 200
2.11.8.2 Viewing Record Hierarchy . . . 200
2.11.9 Importing Data Types from an XSD File . . . 201
2.11.10 Validating Record Values . . . 201
2.11.10.1 Defining Constraints . . . 202
2.11.10.2 Defining Constraint Types . . . 203
2.11.10.3 Filtering Constraints using Tags . . . 204
2.12 Persistent Data . . . 205

2.12.1 Defining Database Properties . . . 205
2.12.1.1 Extracting Affixes from Data Model . . . 206
2.12.2 Generating a Data Model from a Database Schema . . . 207
2.12.3 Creating a Shared Record . . . 208
2.12.4 Creating a Shared Record Field . . . 209
2.12.5 Locking on a Shared Record . . . 210
2.12.6 Data Relationships Between Shared Records . . . 211
2.12.6.1 Setting Fetching . . . 211
2.12.6.2 Defining Indexes . . . 212
2.12.6.3 Defining a Shared Field with a Foreign Key of a Related Record . . . 213
2.12.7 Auditing: Shared Record Versioning . . . 214
2.12.7.1 Setting up Auditing . . . 215
2.12.7.2 Auditing a Shared Record . . . 216
2.12.7.3 Excluding a Shared Field or Record from Auditing . . . 217
2.12.7.4 Excluding a Relationship End from Auditing . . . 218
2.12.7.5 Working with Record Revisions . . . 218
2.12.8 Caching Shared Records . . . 219
2.12.8.1 Defining Cache Regions . . . 219
2.12.8.2 Disabling Cache Regions . . . 219
2.13 Constants . . . 220
2.14 Webservices . . . 220
2.14.1 Web Service Server . . . 221
2.14.1.1 SOAP Webservice Server . . . 222
2.14.1.2 Creating SOAP Web Service Server from Scratch . . . 222
2.14.1.3 Creating SOAP Web Service Server from WSDL . . . 223
2.14.2 Web Service Client . . . 225
2.14.2.1 Creating SOAP Web Service Clients . . . 226
2.15 Monitoring . . . 230
2.15.1 Defining Monitoring Data . . . 230
2.15.1.1 Creating Reports . . . 231

2.15.1.2 Creating Widgets 231
2.15.2 Business Activity Monitor 232
2.15.2.1 Customizing Dashboard 233
2.16 Debugging 235
2.16.1 Debugger Implementation 235
2.16.2 Setting up Debugger 236
2.16.2.1 Debugger on the PDS Embedded Server 237
2.16.2.2 Debugger on the SDK Embedded Server 239
2.16.2.3 Debugger on a Remote Server 241
2.16.3 Debugging a Model 243
2.16.3.1 Adding and Removing Breakpoints 245
2.16.3.2 Defining a Breakpoint Condition 246
2.16.3.3 Resuming Breakpoints 246
2.16.3.4 Enabling and Disabling Breakpoints 246
2.16.4 Tracker 247
2.17 Profiling 247
2.17.1 Profiling with LSPS Profiler 248
2.17.2 Profiling with Trace Logger 249
2.18 Sharing Resources 250
2.18.1 Resource Export 250
2.18.1.1 Export Configurations 250
2.18.1.2 Exporting with General Export 251
2.18.1.3 Exporting with GO-BPMN Export 252
2.18.1.4 Exporting to XPDL 254
2.18.2 Resource Import 255
2.18.2.1 Importing Model Packages to Workspace 256
2.18.2.2 Importing XPDL 257
2.18.2.3 Importing File System Structure 257
2.18.3 Libraries 257
2.18.3.1 SharePoint and Exchange Client Libraries 258
2.18.3.2 Importing a Library to GO-BPMN Projects 258
2.18.3.3 Exporting Modules as Libraries 258
2.18.3.4 Removing a Library from a Project 259
2.19 Model Presentation 259

3 Module Upload and Execution 261

3.1 Server Connection from PDS . . . 261

3.1.1 Connecting PDS to an LSPS Server . . . 262

3.1.2 Connecting to an LSPS Server using a Proxy Server . . . 263

3.1.3 Connecting to the PDS Embedded Server . . . 263

3.1.4 Loading and Resetting Database of the PDS Embedded Server . . . 264

3.2 Model Upload . . . 264

3.2.1 Uploading a Module from the Modeling Perspective . . . 265

3.2.2 Uploading a Module from the Management Perspective . . . 266

3.3 Model Instantiation . . . 267

3.3.1 Instantiating a Model from the Management Perspective . . . 267

3.3.2 Instantiating a Model from the Modeling Perspective . . . 268

4 Updating Model Instances 271

4.1 Model-Update Processes . . . 272

4.2 Transformation . . . 273

4.2.1 Record Transformation . . . 274

4.2.2 Variable Transformation . . . 275

4.2.3 Asynchronous Modeling Elements Transformation . . . 275

4.2.3.1 Transformation Strategies . . . 275

4.3 Performing Model Update . . . 276

4.3.1 Creating a Model Update Configuration . . . 277

4.3.1.1 Copying Model Update Data . . . 277

4.3.2 Editing Model Update Configuration . . . 278

4.3.2.1 Displaying Matching Data in Model Update Configuration . . . 279

4.3.2.2 Changing Element Mapping in Model Update Configuration . . . 280

4.3.2.3 Changing Model Update Settings . . . 280

4.3.2.4 Defining Transformation . . . 281

4.3.2.5 Creating a Model Update Process . . . 284

4.3.3 Updating Model Instances . . . 286

4.3.3.1 Updating a Model Instance from PDS . . . 286

4.3.3.2 Aborting Model Update . . . 288

4.3.3.3 Model Update Logs . . . 289

4.3.3.4 Downloading Model Update Configuration . . . 289

5 Model Update Examples 291

5.1 Updating a Variable Value . . . 291

5.2 Updating a Task Parameter . . . 294

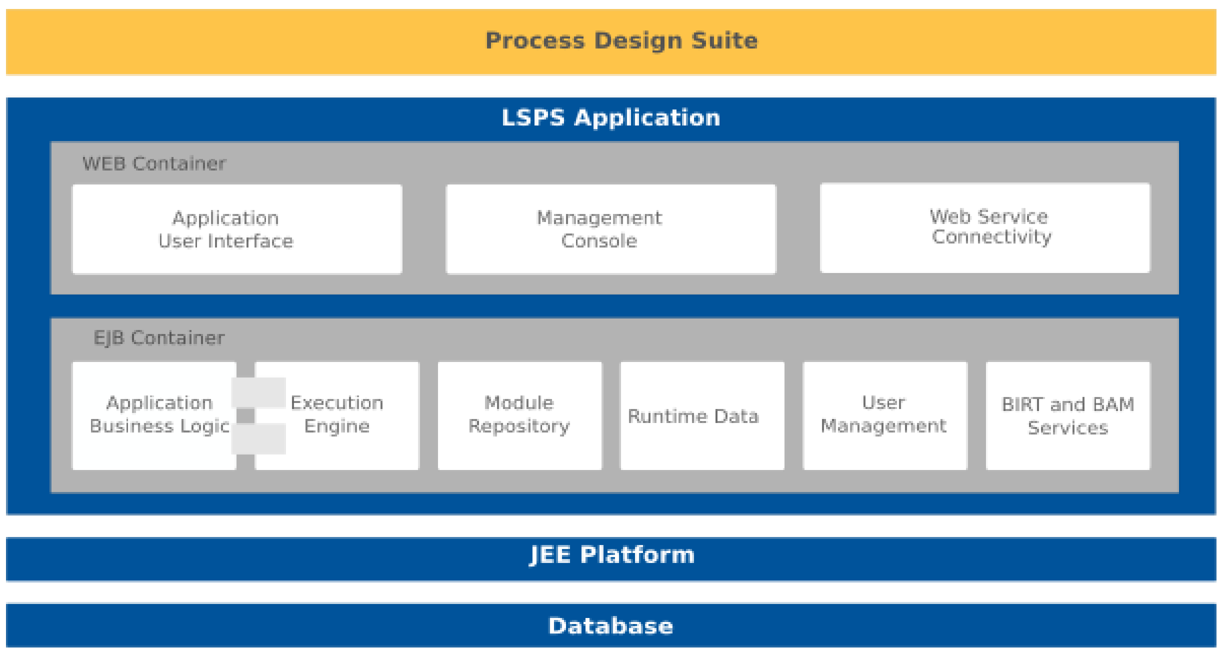
5.3 Updating an Event Type . . . 300

5.4 Updating a Data Type . . . 303

Chapter 1

Business Process Modeling

You will design your models and related resources for the Living Systems® Process Suite, in the integrated development environment called Living Systems® **Process Design Suite** (PDS), a programming platform based on Eclipse. The environment provides the tools for modeling of business processes, and communication with an L↩ SPS Application. This allows you to create your models, deploy, manage, and monitor the execution from a single environment.



PDS comes with a built-in local application server with the LSPS Application, called the **PDS Embedded Server**, and additional support for the management of the server so you can run your Models directly from PDS and on your machine.

> **Note:** If you install the SDK component in the PDS, you can generate the LSPS Server EAR with sources of the Application User Interface and modify them. For further information, refer to the `development guide`. and for instructions on how set up your own application server with the LSPS Application, refer to the `deployment guide`.

This guide focuses on the features related to the PDS environment and generally does not contain information on customization of the LSPS Application User Interface, the concepts related to `GO-BPMN`, or the `Expression Language`: refer to the respective guides for such information.

1.1 Running Process Design Suite

After you have installed the PDS Enterprise Edition, run the Process Design Suite:

1. Go to `<LSPS_HOME>`.
2. Run the `lsps-design` binary for your platform:
   - `lsps-design.exe` and on Windows with 32-bit architectures
   - `lsps-design_64.exe` on Windows with 64-bit architectures
   - `Process Design Suite` on Mac
   - `lsps-design` on Linux
3. In the Workspace Launcher dialog box, choose a workspace directory and click OK.

   If the chosen workspace folder does not exist, it will be created. A workspace is a folder, where the resources stored during the session are located. Its content is reflected in the workbench.

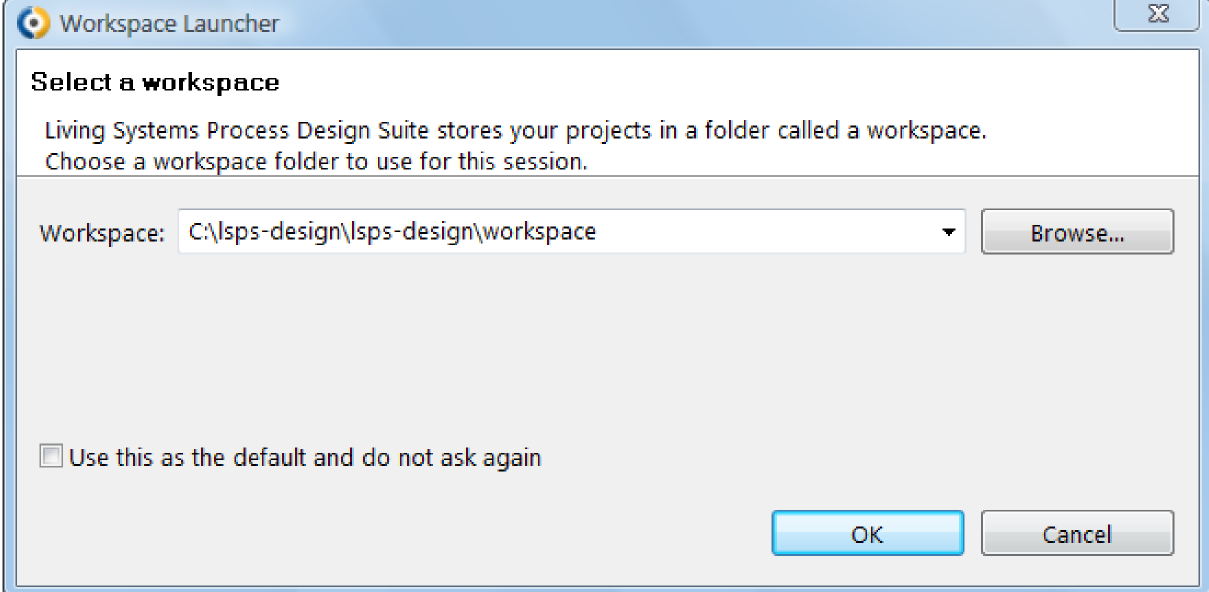


Figure 1.1 Workspace Launcher dialog

If you are starting PDS in a new workspace, PDS displays the Welcome page with links to perspectives, documents, and other PDS resources.

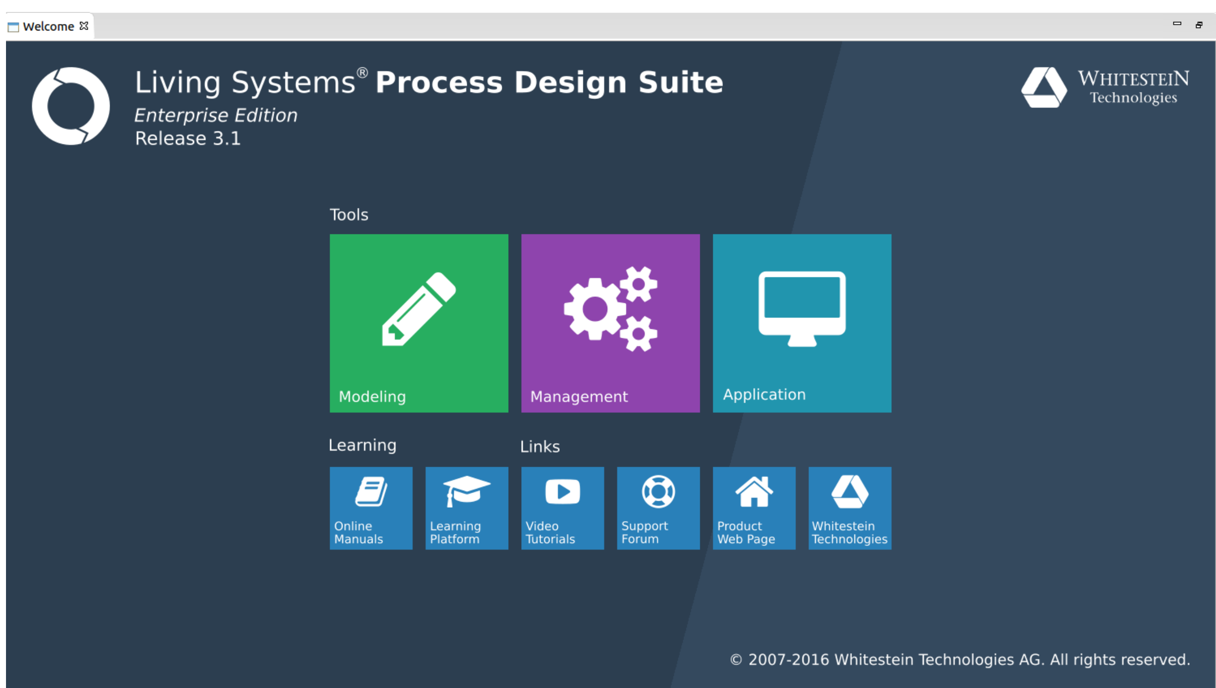


Figure 1.2 Welcome page

You can display it at any point from the main menu Help > Welcome item.

> **Note:** If the *Welcome* command is disabled, open a Process Design Suite perspective: go to Window > Open Perspective > Other and in the Open Perspective dialog box, click a Process Design Suite perspective.

1.2 Checking of Errors in Process Design Suite

Errors thrown by the Process Design Suite and other environment plug-ins are available in the **Error Log** view. Each error entry shows information about the plug-in the error was generated by, and date and time when it occurred.

Double-clicking an error entry opens Event Details dialog box with detailed information on the respective error message including the date and time of its occurrence, its severity, error message, its stack trace, and session data.

Error Log can contain also entries of the server if these were caused by a Process Design Suite feature, which appear primarily in the Console view.

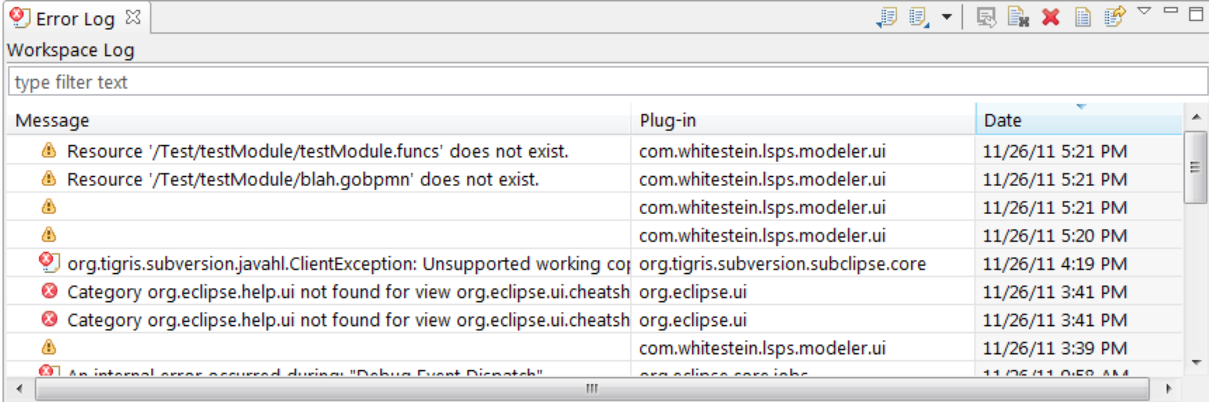


Figure 1.3 Error Log view

Chapter 2

Designing Models

When designing your models and related resources for the Living Systems® Process Suite, you will do so in the integrated development environment Living Systems® Process Design Suite (PDS), a programming platform.

To create a model, first you need to create the model structure: projects with modules. Then you can create the content of your model in dedicated definition files. The definition files will hold the elements and their properties needed by your model.

You model can contain and make use of the following:

- BPMN and GO-BPMN Processes,
- global variables,
- organization models and use their units to involve correct users in an activity
- documents used as permanent content in the front-end application,
- forms used as content of documents and to-dos,
- localization identifiers to have your texts displayed in the required language in the front-end application
- function definitions,
- definitions of queries to databases,
- data types,
- definitions of persistent data types,
- definitions of constants.

In addition, you can create Processes that serve web service requests and perform web service calls, define additional content for the BAM application. debug your models, check performance and import and export the resources.

Your resources will rely on the resources of the `Standard Library`. Apart from the Standard Library, you can use other out-of-the-box libraries.

Once your model is ready, you can present it in the Business Modeling perspective.

2.1 Model Structure

When designing your models, you will rely on projects and modules:

- GO-BPMN projects are the top-level containers; they resembles a directory and as such can contain other directories. But primarily they will hold your modules. Unlike modules, projects are not deployed to the server and do not influence module deployment or execution.
- Modules hold the definition files with model resources used for execution, such as, processes, data types, variables, etc. and are deployed to the server.

Note: We often use the term *model* and *model instance*: a model is the object that is deployed to the server, more specifically to the LSPS Execution Engine, and serves as a blueprint for model instances. On design time, it is an executable modules with all its resources including any imported modules.

Model instances are runtime versions of a model. For basic information on Projects, Modules and Models, refer to the Quickstart guide.

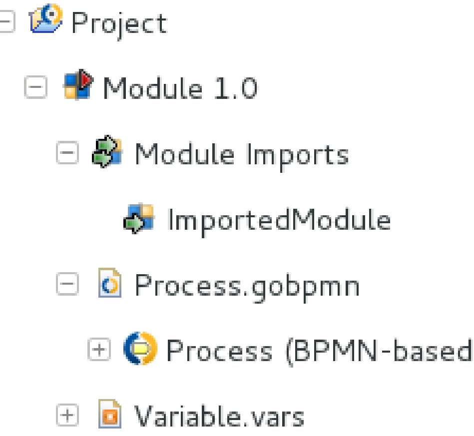


Figure 2.1 GO-BPMN Project Structure

2.1.1 GO-BPMN Projects

GO-BPMN projects are folders that can hold Modules (also library modules) and a few definition files which require different workspace resources, such as, export configuration(Export Configurations).

Project content does not depend on other resources and therefore it cannot access content in other projects. If you require in your project content of another project, you will need to reference it).

2.1.1.1 Creating GO-BPMN Projects

To create a GO-BPMN project, do the following:

1. Click File > New GO-BPMN Project.

   Alternatively, open the context menu in the GO-BPMN Explorer and click New > GO-BPMN Project.
2. In the New GO-BPMN Project dialog box, in the Project name text box, enter the project name.

Note: The GO-BPMN Libraries box displays the libraries and their modules included in the project. The Standard Library is included by default.

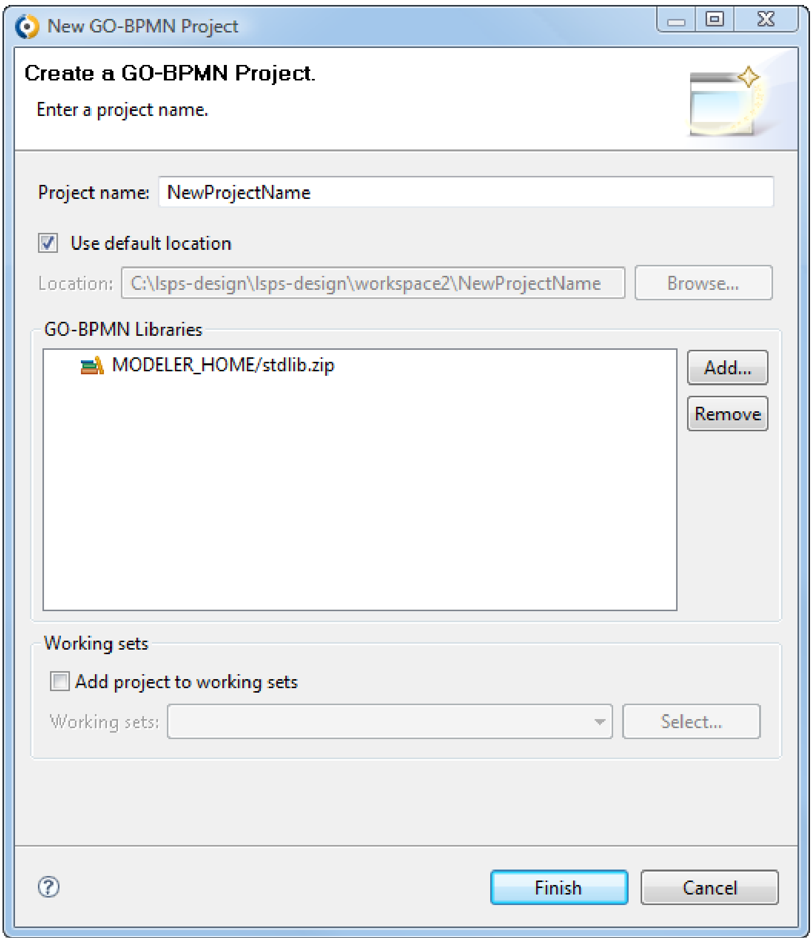


Figure 2.2 Creating a new GO-BPMN project

3. Click **Next**.
4. On the *Project References* page, select the project, which the new project can reference.
5. Click **Finish**.

2.1.1.2 Closing GO-BPMN Projects

Closing GO-BPMN projects makes projects and their content “invisible” for tools and that including the validation tool.

To close a project, right-click it in the GO-BPMN Explorer and select Close Project. To open a closed project proceed analogously.

2.1.1.3 Referencing Projects

Project content does not depend on other resources and therefore it cannot access content in other projects. If you require in your project content of another project, for example, you want to import its modules, into you module, you

can reference it: Note that referencing will copy the resources into your Projects and the module imports will be copies of the original modules, not references.

If a project references another project, it can use its content (import its Modules). Projects may reference each other.

To reference a project:

1. In the GO-BPMN Explorer, right-click the project, in which you want to reference another project and select Properties.
2. In the left pane of the Properties dialog box, select Project References.
3. In the Project References page, select the projects to be referenced.

   Note: When referencing projects, make sure the modules in the project have different names.

You may import modules of the referenced projects into modules of the parent project.

2.1.2 GO-BPMN Modules

GO-BPMN Modules are reusable units that hold definition and configuration files and can represent a Model. Similarly to packages in Java, they serve to organize resource files to logical bundles that can import each other.

2.1.2.1 Creating GO-BPMN Modules

To create a GO-BPMN module:

1. Go to File -> New -> GO-BPMN Module.

   You may also use the context menu of the respective GO-BPMN project.
2. In the New GO-BPMN Module dialog box, select the parent GO-BPMN project.
3. In the Module name text box, type the `module name`.
4. Select or unselect `executable module` checkbox.

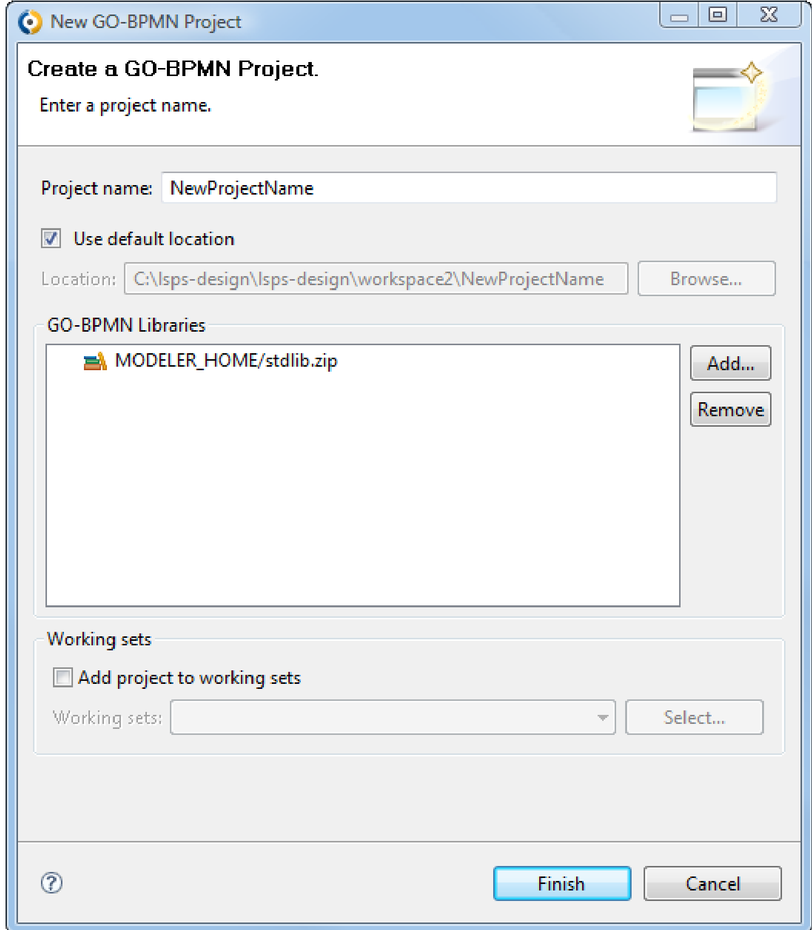


Figure 2.3 New GO-BPMN Module dialog box

5. Click Finish.

   Click Next and define module imports, libraries or modules, if required.

Note that the Module has its properties set to default values: its version set to 1.0 and there is no terminate condition.

2.1.2.1.1 Specifying Module Properties

To specify a module version, terminate condition, and a free-text description:

1. In the GO-BPMN Explorer view, right-click the module.
2. In the context menu, select Properties.
3. In the Properties dialog box, click GO-BPMN in the left part of the dialog box.
4. In the General tab, define the Module properties:
   - Module version: a module with a particular version is considered a different module on upload (modules with different versions can coexist in the server Module repository)
   - Executable module: if true, the Module can be instantiated as a Model
     Only a module that is executable can become a model instance

- Create process log: if selected, the process logs its runtime data into the database

  If you disable the setting, runtime data of the Model instance will not include Module data (for example, no data on process instances nor their diagrams will be available). This setting is intended for production environments.

  Note that the setting can be overridden by the CREATE_PROCESS_LOG setting in the database.
- Terminate condition: a condition which has to be true throughout the entire life of the model instance (condition in module imports are ignored)

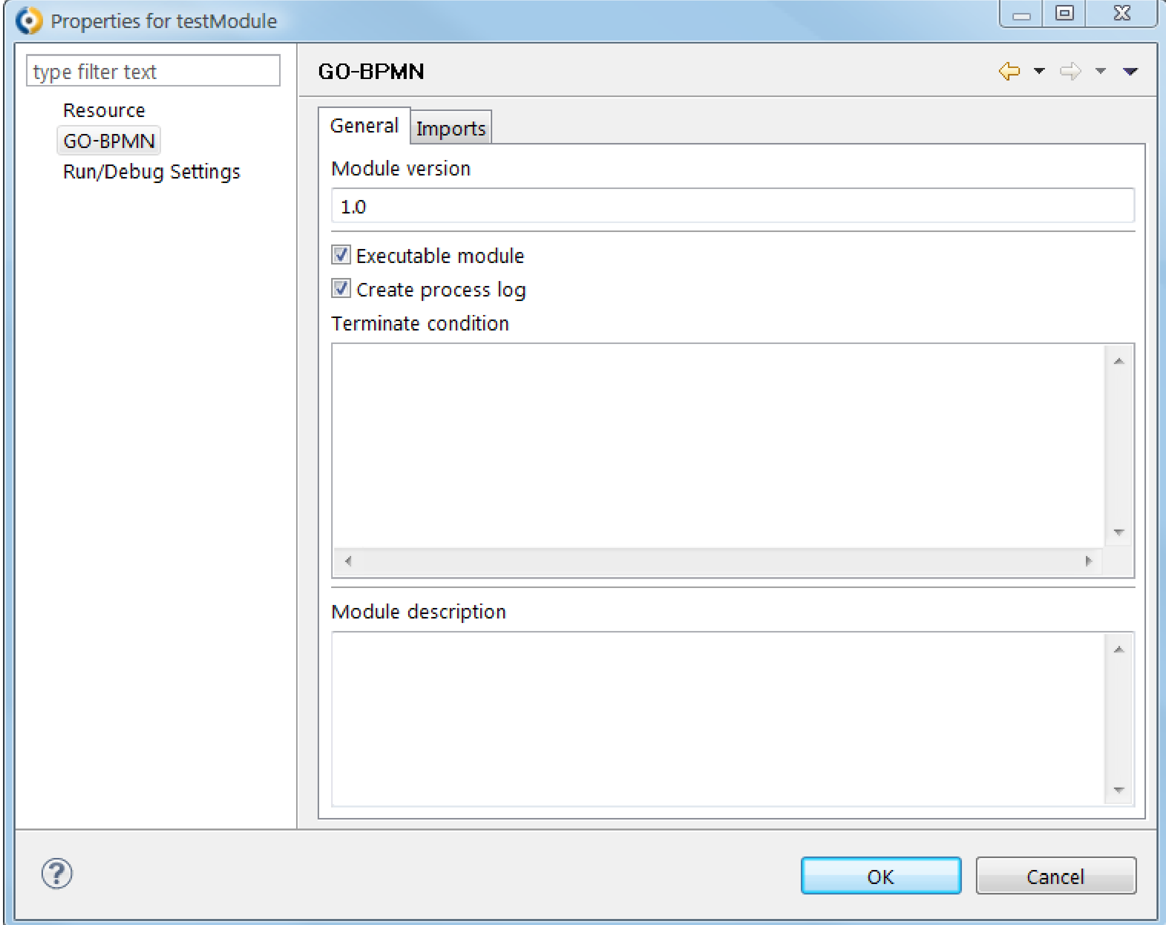


Figure 2.4 General tab of the Properties dialog box

Alternatively, you can modify the module properties in its Properties view (in GO-BPMN Explorer, select the module and edit the data in the Properties view).

2.1.2.2 Importing Modules

Module import allows a module to use resources of other modules.

To import a module of the same or a referenced project, do the following:

1. In the GO-BPMN Explorer, right-click the target module you wish and click Module Imports.

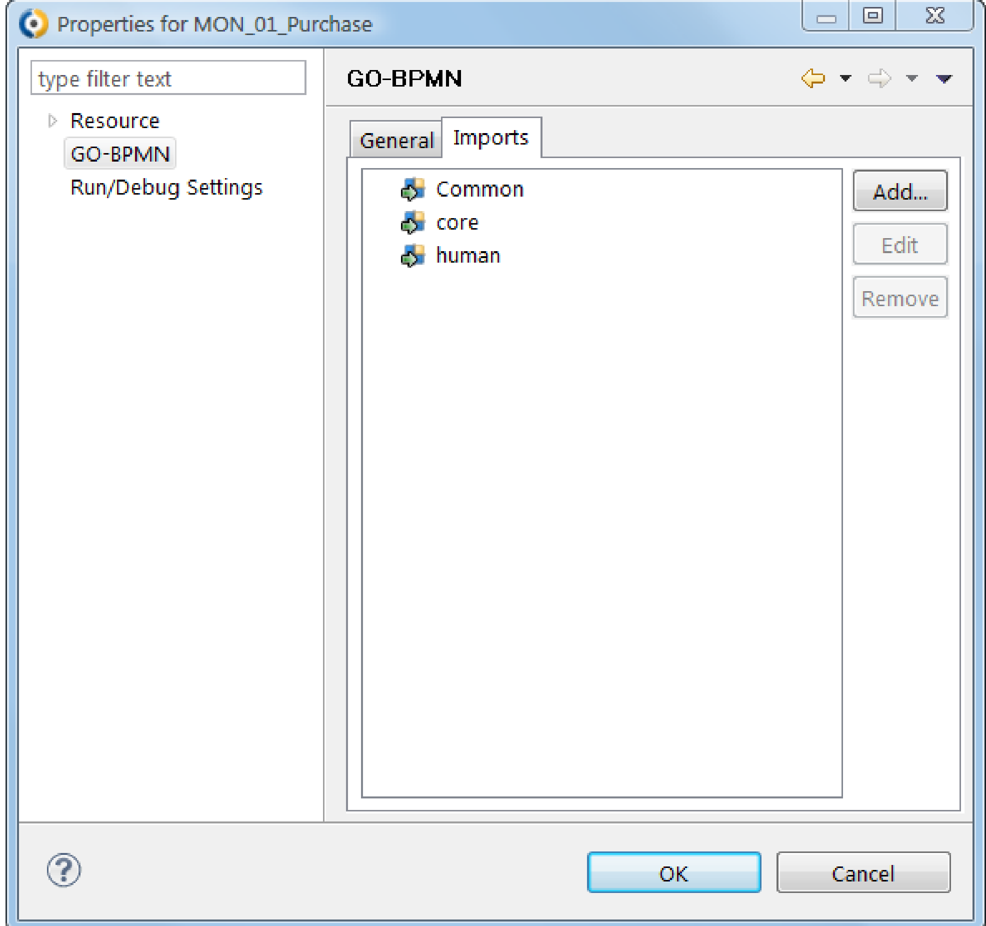


Figure 2.5 Imports tab with a list of imported modules

2. On the Imports tab, click Add.
3. In the Add New Import dialog box, in the Select a module to import box, select the module to be imported (expand the tree if necessary).

   To add several modules, double-click every module.
4. Click OK.
5. Back in the Properties dialog box, click OK.

To remove a module import, in step 2 select the module and click **Remove**.

2.1.2.2.1 Viewing Module Dependencies

When referencing projects and importing modules, the relationships between them can become relatively complicated and difficult to follow in the GO-BPMN Explorer. To allow you to view such relationships in a comprehensive way, the modeling tool provides the *Module Dependency View*. The view shows a diagram of projects and modules and their relationships.

To display the view, go to Window > Show View Module Dependency View.

To simplify the module relationships depiction, hide the transitive relationships by clicking the Show Transitive Reduction and pick the Module you are interested in in the *Display dependencies of module* drop-down box.

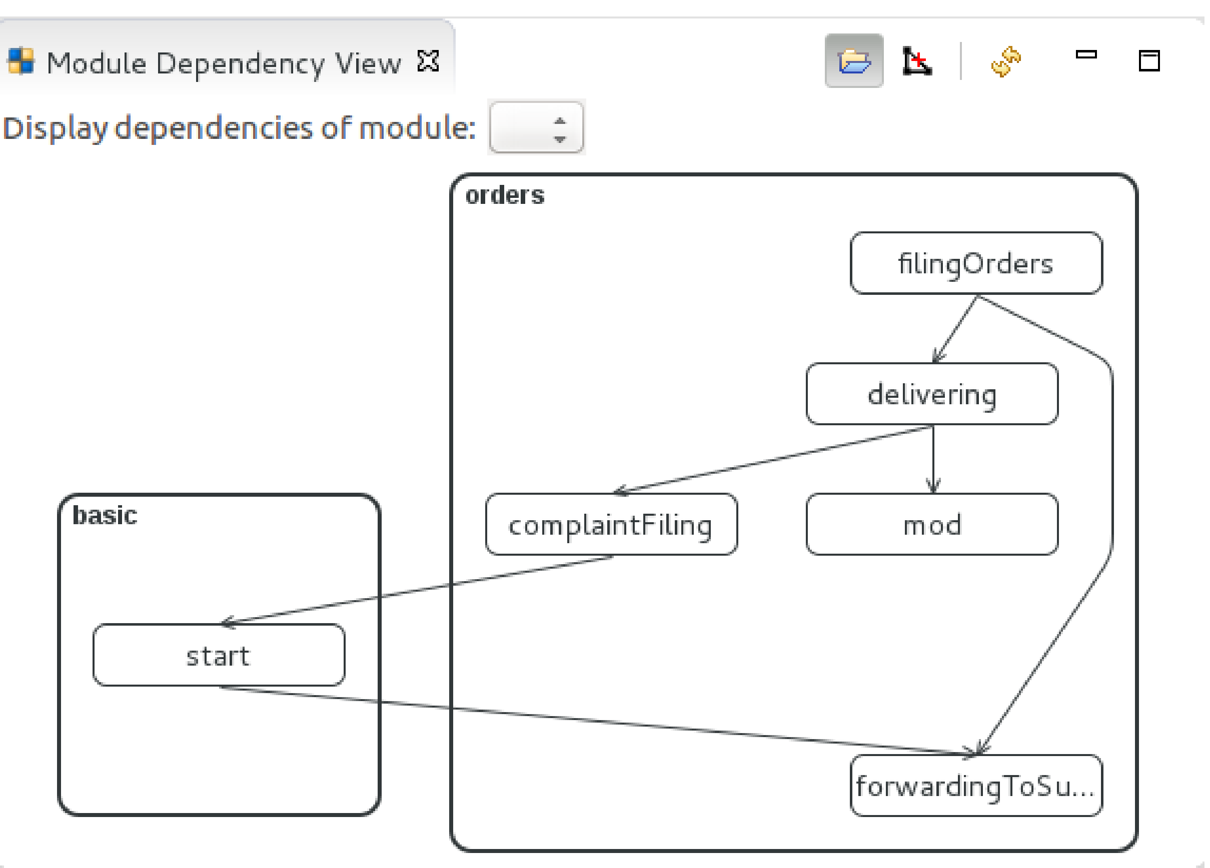


Figure 2.6 Module Dependency View with Transitive Reduction Activated and Deactivated

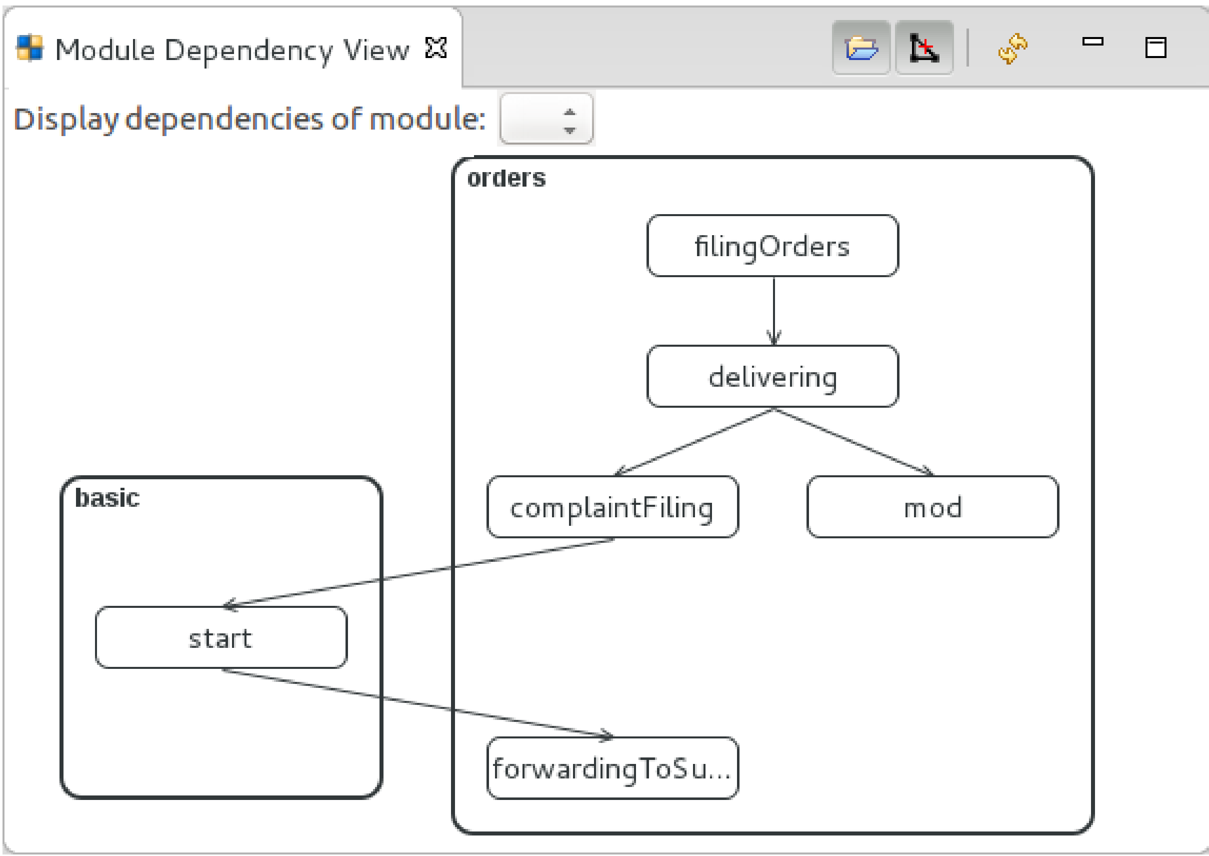


Figure 2.7 Module Dependency View with Transitive Reduction Activated and Deactivated

2.1.3 Definitions and Configurations

Definitions and configuration files, or resource files, contain definitions of resources used by the Module, such as, variables, data types, processes, organization, etc. The resources are always created in a definition or configuration file of a particular type: global variables can be defined only in a variable definition file; a process can be created only in a process definition files.

> **Note:** Model update configurations (`.muc`) and export configurations (`.export`) exist in GO-BPMN Projects, not Modules, since .muc requires definition of an old and a new model and export can define projects and modules.

Some files can contain `diagrams`, which visualize the content of the file:

Data type definition with the extension `.datatypes` hold data type models with custom data types called Records.

Process definition with the extension `.gobpmn` hold *One* GO-BPMN or BPMN Process

Organization definition with the extension `.orgmodel` organization model holds Organization Units and Roles

For further information on Diagrams as well as the depicted elements, refer to `GO-BPMN Modeling Language`.

2.1.3.1 Creating Definitions and Configurations

To create a definition or configuration file, do the following:

1. In the GO-BPMN Explorer view, right-click the respective module or project and click New.
2. Select the desired type of definition file.

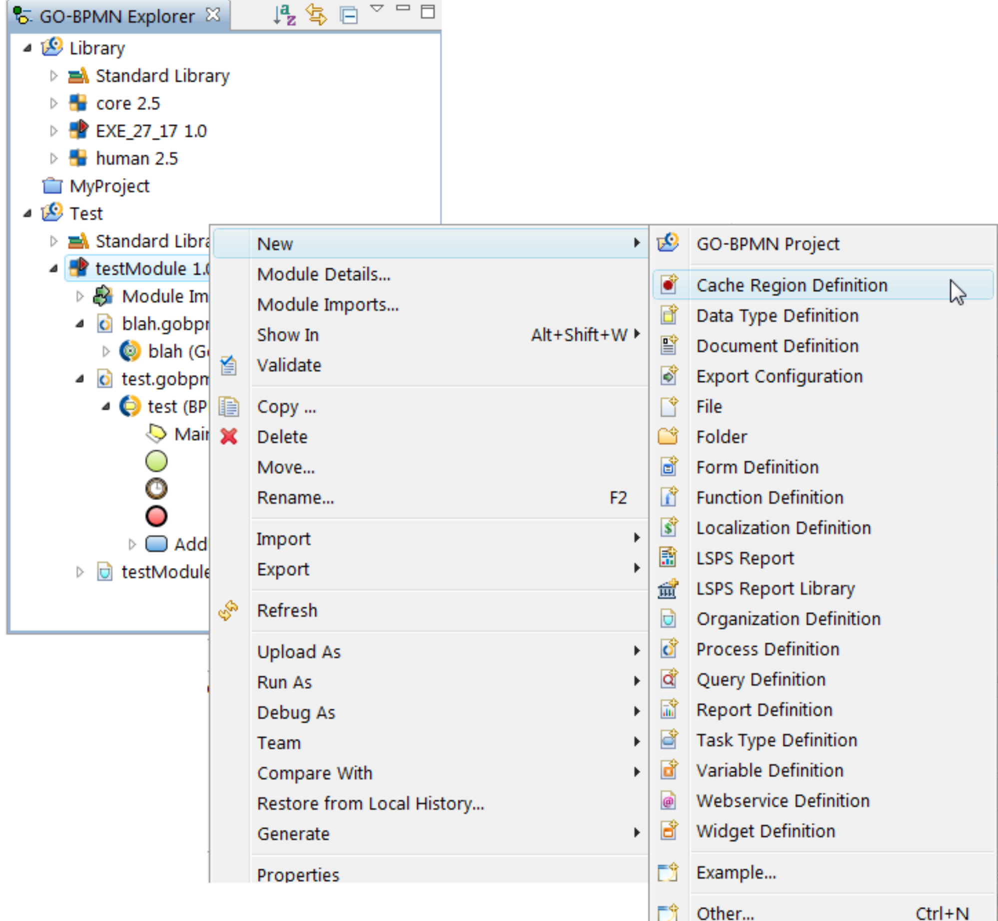


Figure 2.8 Creating a definition file

3. Check the location and enter the name of the definition or configuration file.

   When creating some definitions, for example, a process definition, the system will prompt you to enter further properties for the definition file (refer to sections on individual files for information on the properties).
4. Click Finish.

2.1.3.2 Opening Resource Files

By default the system opens a definition or configuration file with the appropriate graphical PDS editor. However, you can open any file with any internal or external editor.

To open a file with the default editor, double-click the file or any of its children in the GO-BPMN Explorer.

To use a custom editor, right-click the file in GO-BPMN Explorer and select **Open With** and select the respective editor. Alternatively, select **Other**, and in the Editor Selection dialog box, select Internal editors or External programs and double-click the respective editor.

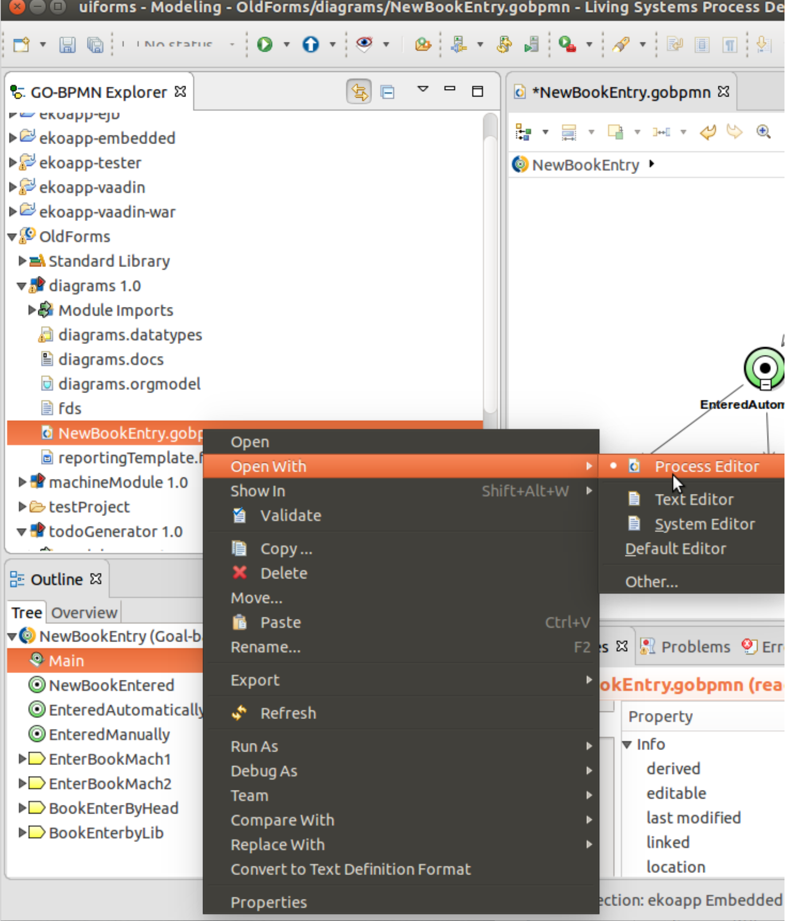


Figure 2.9 Selecting an editor

2.2 Generic Modeling Mechanisms

These are generic mechanisms applicable to the entire environment or all resources:

- Defining expressions
- Defining element properties
- Disabling (commenting out) elements
- Adjusting layout of elements in Diagrams
- Comparing resources
- Defining modeling status of elements
- Using Todo and Task markers
- Validating of the workspace data
- Searching for elements

2.2.1 Writing Expressions

Properties of items in Definition and Configuration files, such as, initial values of variables, assignments of process elements, Form elements in the Expression component, etc. are defined as expressions in the `Expression Language`. The expressions are evaluated **on runtime**.

When editing expressions, you can use the following features:

- Content Assistant (auto-complete): displays possible values including available data types, language constructs variables, function calls, etc. To display Content Assist, press **CTRL + space**.

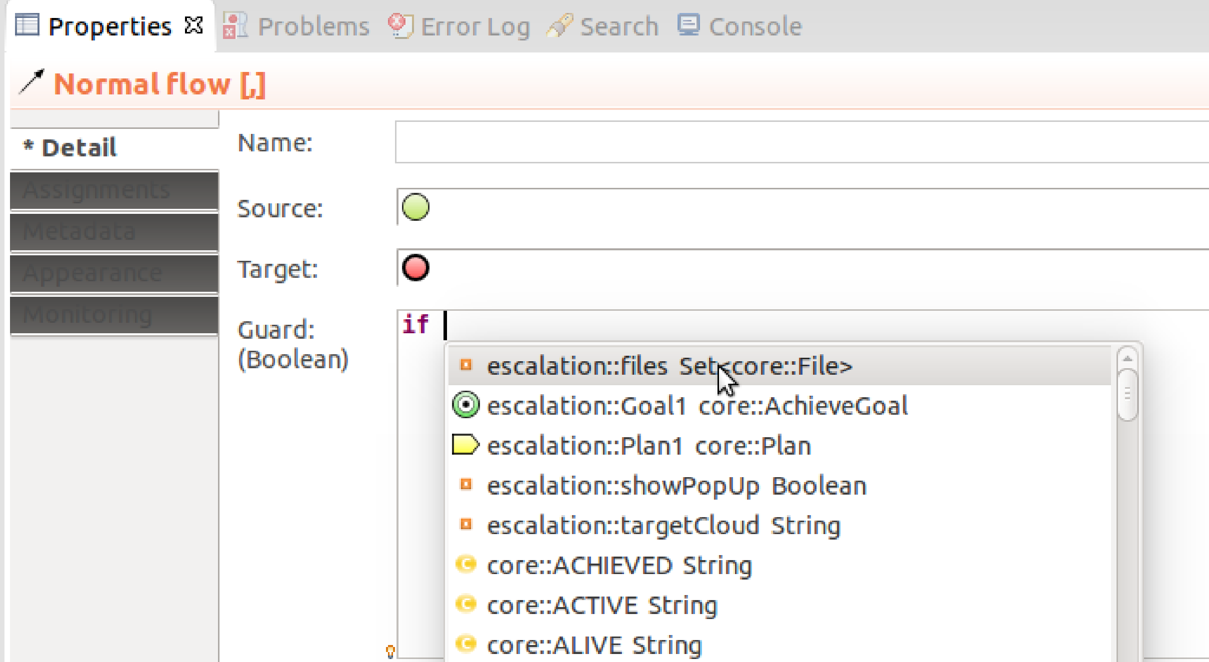


Figure 2.10 Content Assist menu

- Hyperlinking to definitions: opens the definition of the item usage in an expression

  To go to the definition of an item, press **CTRL + left-click**.

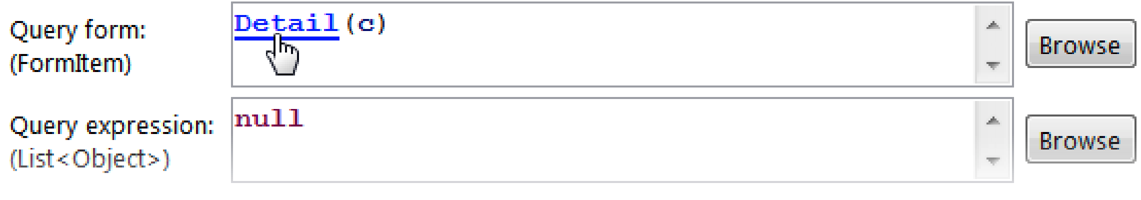


Figure 2.11 Link to definition

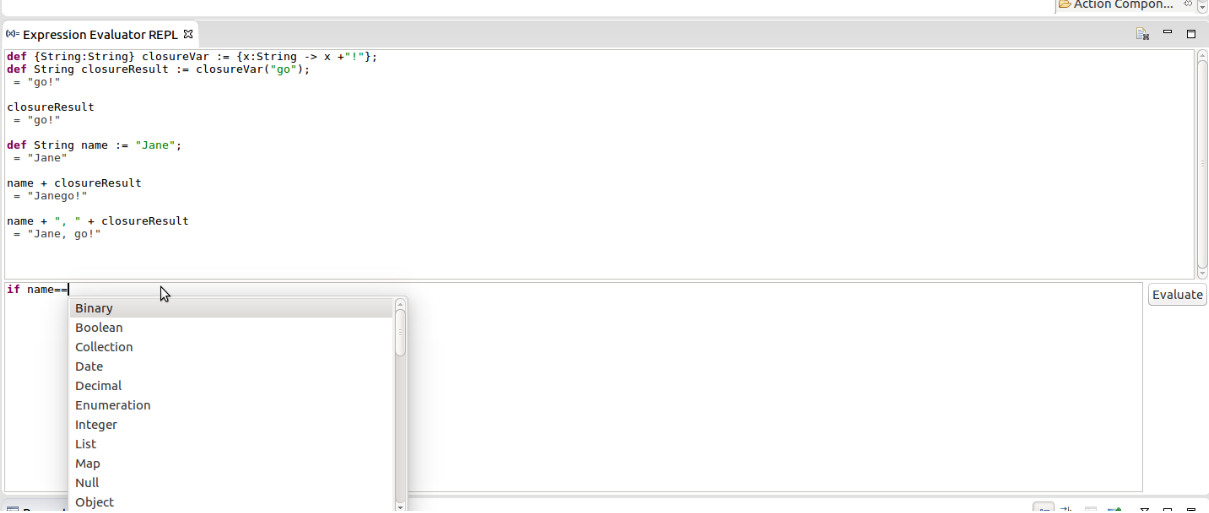
2.2.1.1 Testing Expressions

You can test expressions in the *Expression Evaluator REPL* view.

By default, the view is not displayed: To display it, go to **Window** > **Show View** > **Other** and in the displayed dialog search for the view.

In the view, you can enter expression and have the system return the resulting value: write an expression and click Evaluate or press *CTRL + Enter* (Enter will make a new line in your expression).

Note that you can list the history of expression by pressing the arrow-up key and display auto-completion options by pressing *CTRL + Space*. The view does not support calls to library resources, such as functions.



To erase the context data you created with your expressions in the view, click the clear button.

If you want to test your expressions in a context of a model instance and use resources of libraries, use the `Expression Evaluator` view.

2.2.2 Defining Properties

Every element or item with a semantic execution value, be it a Goal, Task, global variable, a Form component, etc. needs to define some set of properties related to its execution or behavior. Generally you can edit the properties in the *Properties* view, in the editor intended for the element, or in a dedicated popup editor.

Below each property name, you can see the data type of the expression or its return value.

For example, the Properties view below contains the Visible and Root property:

- *Visible* property accepts a Boolean value: you can insert any expression that will return a Boolean value on runtime.
- *Root* property accepts a Collection of TreeItems (the TreeItem type is defined in the ui module) as stated by `(Collection<ui::TreeItem>)` below the property name.

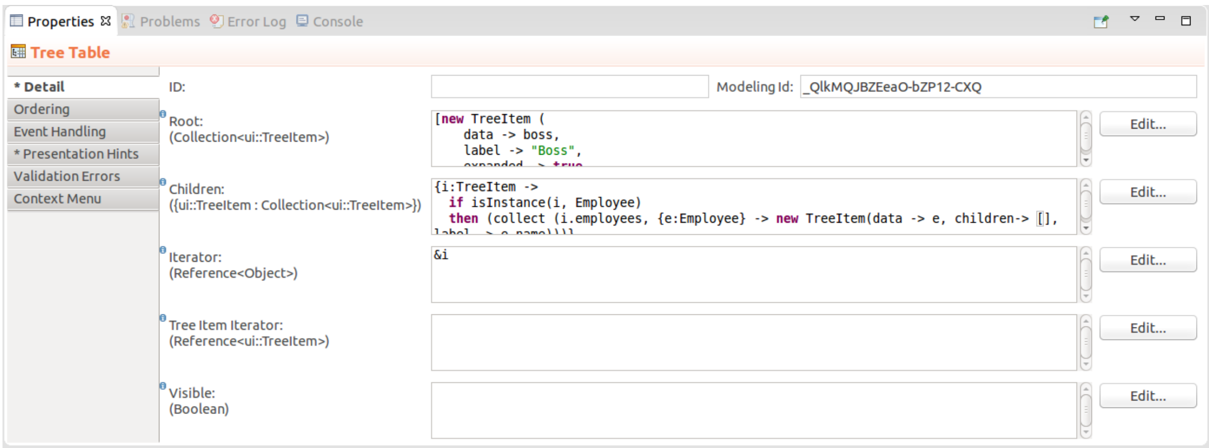


Figure 2.12 Properties view of a form component

The Save action is a closure that takes Todo as its first and SavedDocument as its second parameter and returns an Object value:

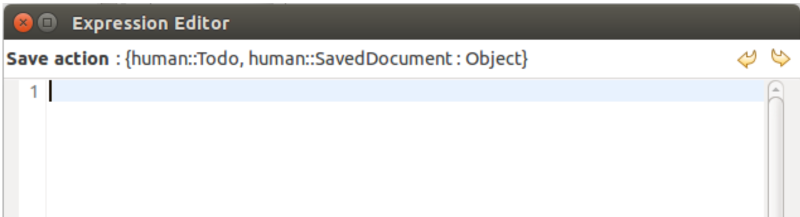
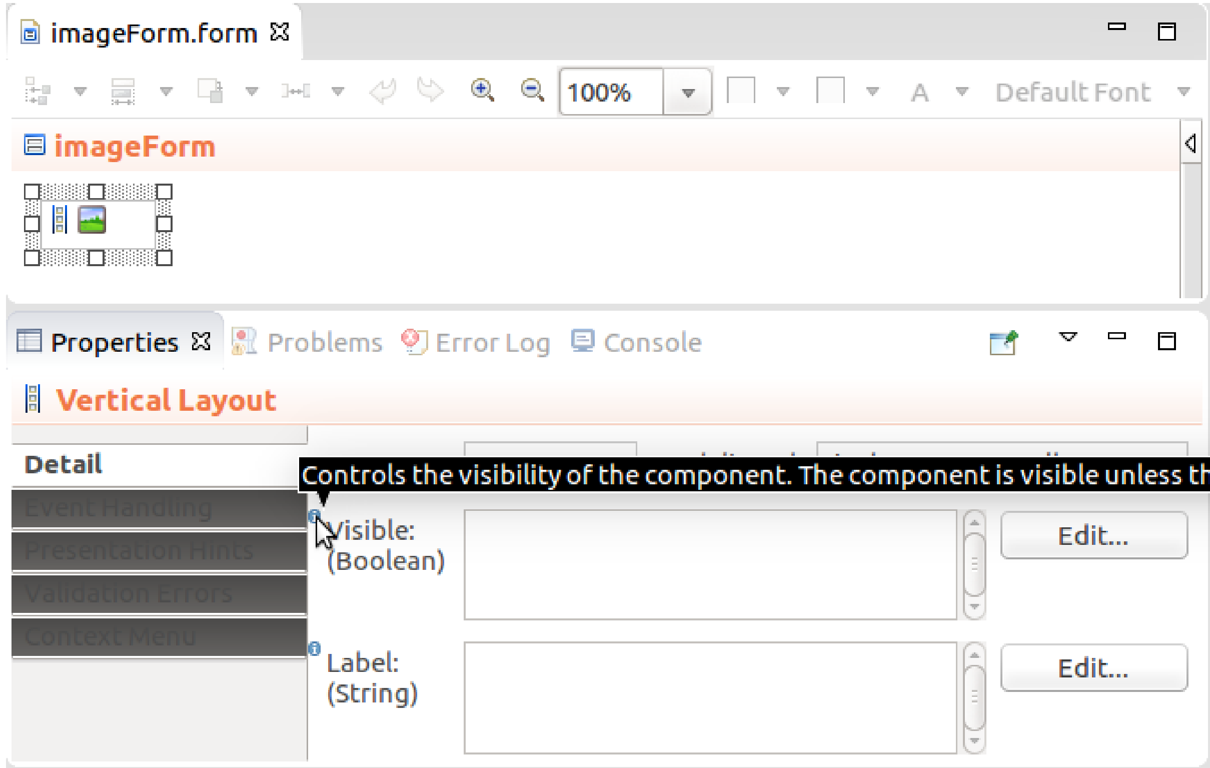


Figure 2.13 Editing Save action property

Some properties have a tooltip with useful information.



Note that any tab with changes to the property values is marked with an asterisk.

2.2.3 Disabling Elements

The Disabling mechanism serves to exclude elements from validation and execution: you can disable any element, such as, cache region, variable, process element, etc.

If a diagram element is disabled, all its diagram views are shown as disabled if applicable. and their representations in both the GO-BPMN Explorer and editors indicate the disabled status. If it has any incoming and outgoing Flows or contains any nested modeling elements, all Flows and nested elements are automatically disabled as well.

Reflections of disabled elements, such as, the record types and functions, must be disabled manually.

You can disable elements using their context menu:

- in the editors (on diagrams or in the lists with definitions) or
- in the GO-BPMN Explorer (right-click the element and click Disable)

  Note that parameters of elements, such as queries, functions, tasks, etc. can be disabled only in GO-BPMN Explorer. It is not possible to disable and enable parameters from editors.

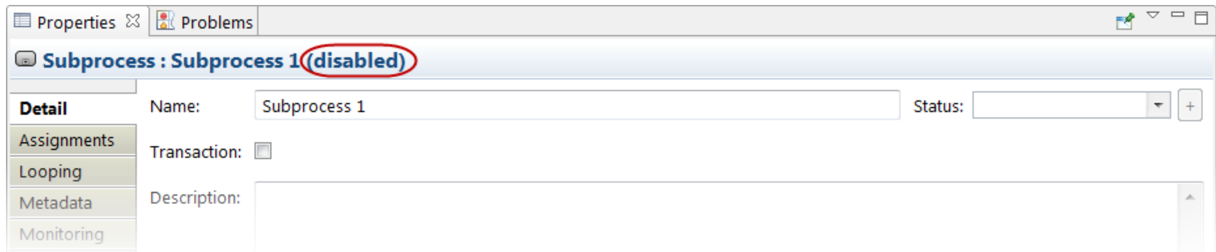


Figure 2.14 Disabled elements in the Properties view

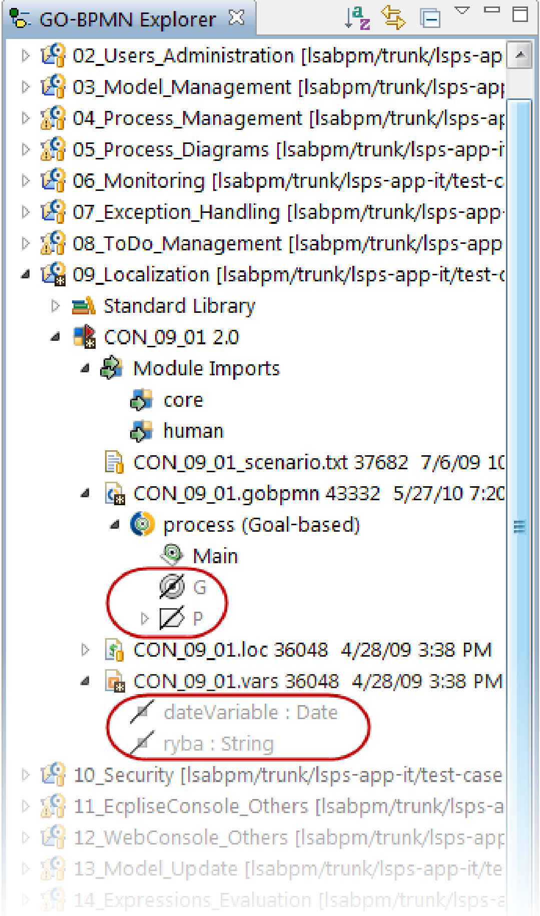


Figure 2.15 Disabled elements in GO-BPMN Explorer

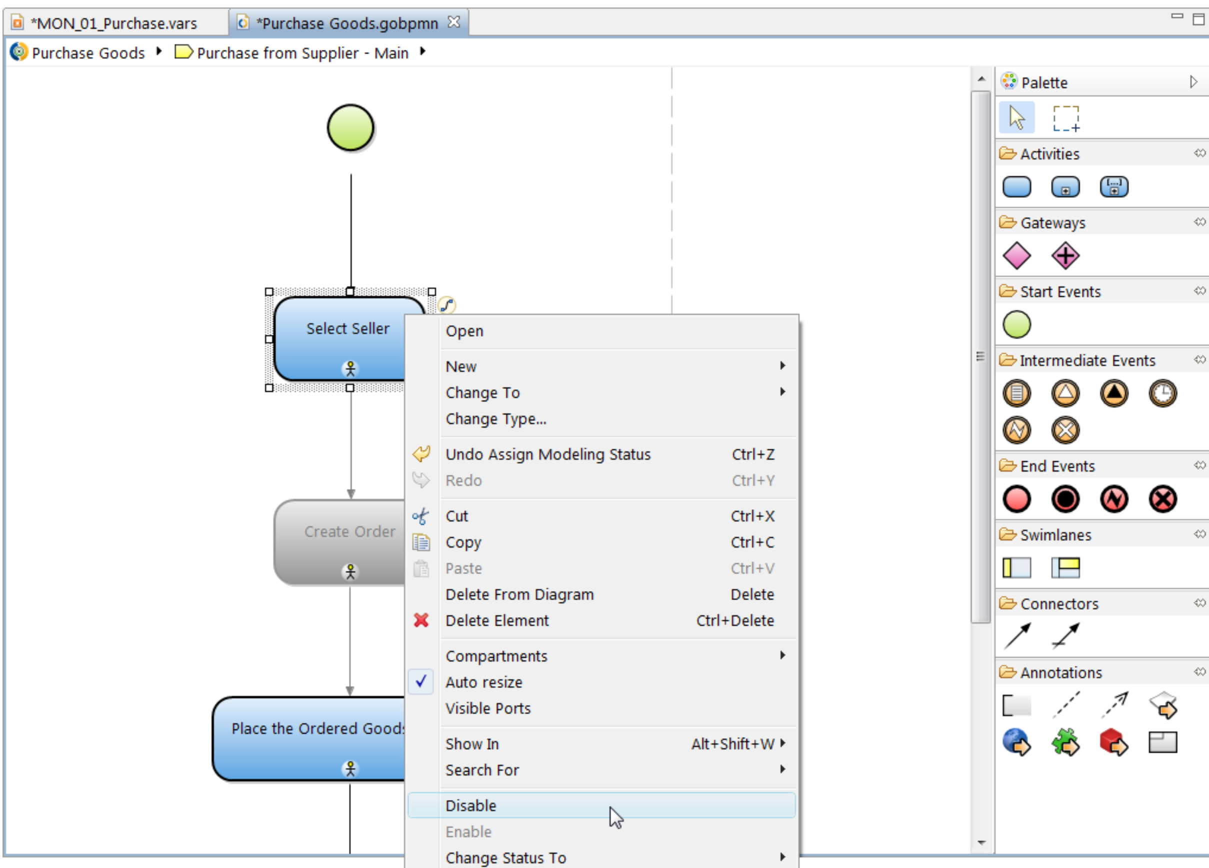


Figure 2.16 Disabling an element in a diagram editor

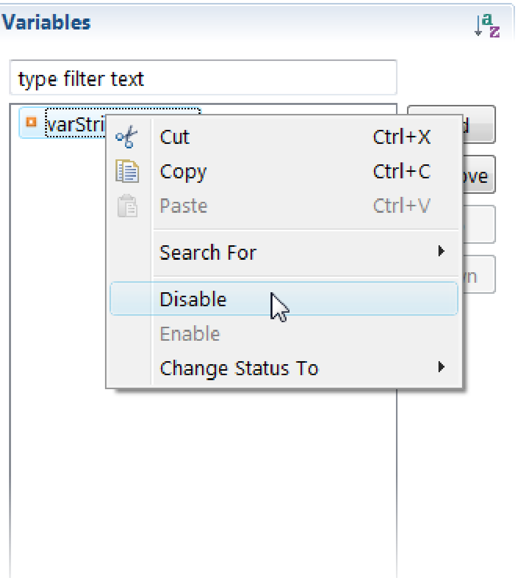


Figure 2.17 Disabling an element in a form-like editor

If you want to disable an entire project, you can use the Eclipse close feature: right-click the project and click **Close**.

2.2.3.1 Defining Formatting of Disabled Elements

You can set the style applied to elements and their views when they are disabled as follows:

1. Go to Window > Preferences.
2. In the left part of the Preferences dialog box, expand Process Design Suite > Modeling > Appearance.
3. On the Appearance page on the right, modify the style in the Formatting of Disabled Elements.

2.2.4 Working with Diagrams

The process, organization, and data type definition files can contain diagrams which visualize their content. To edit the content of these definition files, you can use diagram editors: In these editors, you can edit the content of the definition file on a canvas typically by dragging-and-dropping elements onto the canvas.

> **Note:** An element of such a definition does not have to appear in a diagram: it can exist in the definition without its Diagram representation. And one element can appear in multiple diagrams but is still the same element that exists in the definition only once. Hence diagrams can contain the following:
>
> - one view of an element: graphical representations of elements from the definition;
>
>   For example, a diagram in an organization definition can contain one view of any Role or Organization Unit in the organization definition.
> - diagram elements such as annotations; such elements do not have any semantic value for execution but serve to provide information to the user.
>
> For more information on the Diagram types, elements, and content, refer to `GO-BPMN Modeling Language Specification`.

When you create a resource that supports Diagrams, the system automatically creates a default Diagram in the resource file and displays it in the respective diagram editor.

You can apply the following global settings to your diagram editors available in the main toolbar:

- Settings on displayed elements:
  - *Show Diagram Annotations*: if unselected, all Annotation elements are hidden (that is Annotations, Associations, hyperlinks, and diagram frames).
  - *Show Validation Errors/Warnings*: if unselected, all error and warning markers are hidden.
  - *Show Modeling Statuses*: if unselected, the properties applied on the diagram element due to their modeling status are cancelled.
  - *Show Monitoring*: if unselected, monitoring markers on elements with monitoring expressions are hidden.

    Monitoring expressions hold expressions that store monitoring data. Such data can be then used, for example, in reports.
  - *Show Tooltip*: if unselected, tooltips with descriptions of elements are not displayed on mouse hover.
  - *Show Translations*: if unselected, the form editor displays the localization identifier call instead of the default value of the identifier.

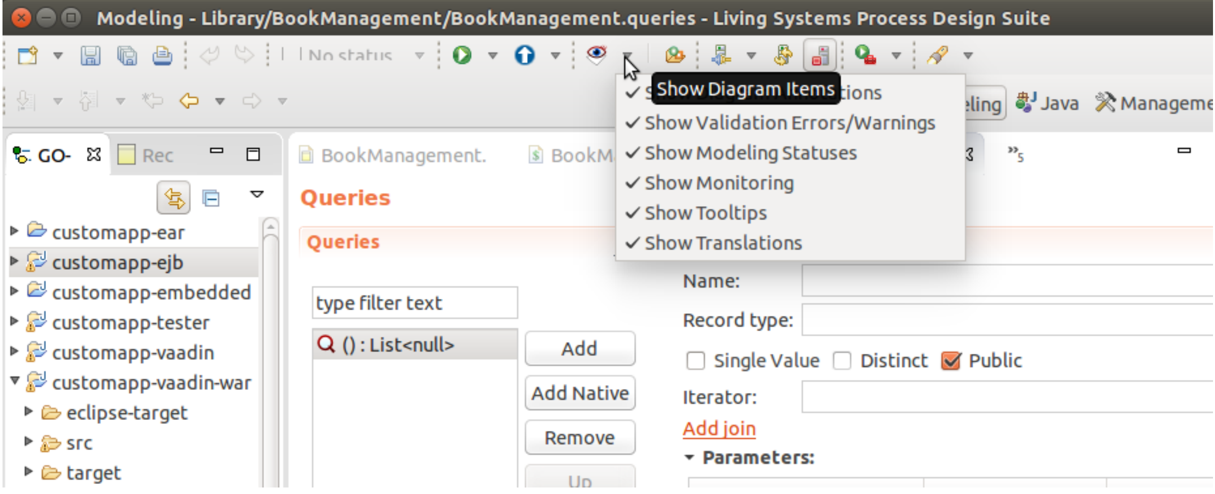


Figure 2.18 Menu with Global Diagram Editor Settings

- General settings such as line style, task visualization, etc. Go to **Windows** > **Preferences**, then **Process Design Suite** > **Modeling** > **Appearance**.

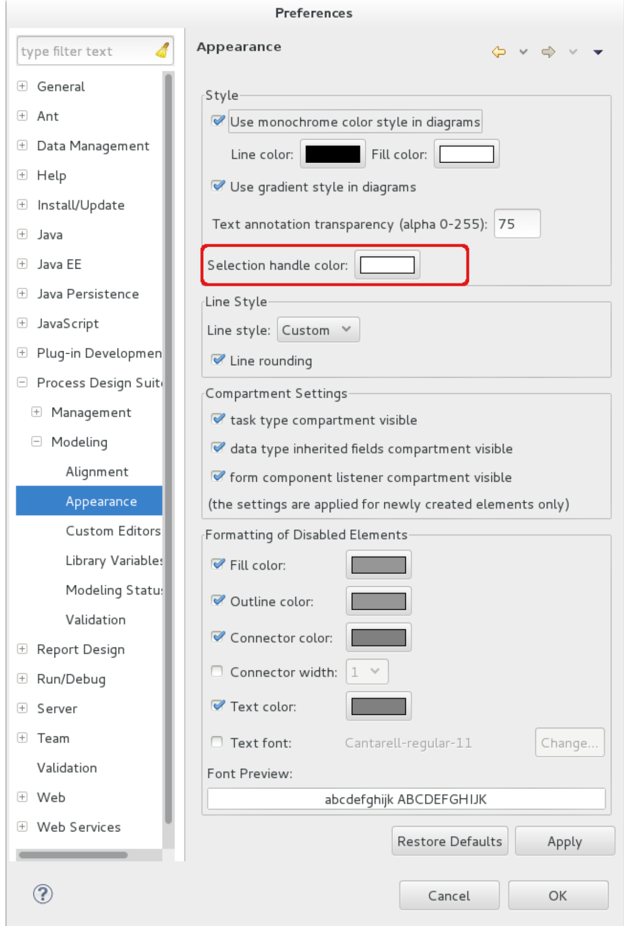


Figure 2.19 General settings

2.2.4.1 Creating a Diagram

After you have created a definition, which supports diagrams (process, organization, or data type definition, or a process in the model update configuration), the system creates a diagram in the file and opens it in the respective diagram editor. However, you can decide to create multiple diagrams in one file with different element views. To create a new diagram, do the following:

1. In the Outline view, right-click the root node of the tree.
2. Select New -> Diagram.

2.2.4.2 Inserting Element Views

Any view of an element with a semantic value that you insert into a Diagram is automatically created in the definition file, for example, if you insert a Role view from the palette or from the context menu of the canvas into an Organization Diagram, the system automatically creates the Role element in the definition file.

To create a new element and insert its view into a diagram:

1. Click the element on the palette.
2. Click the area on the canvas, where you want to place the element view.

Alternatively, right-click empty space in the canvas and select the element from the context menu.

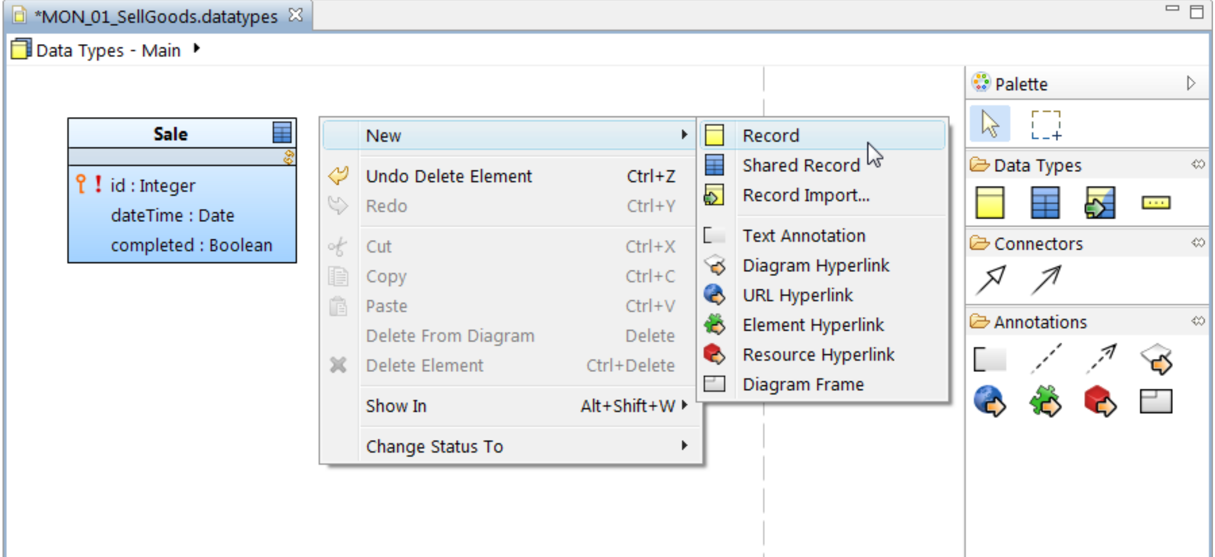


Figure 2.20 Inserting a new element through the context menu

For quick inserting of new elements onto the canvas, use the quicklinker:

1. Select a diagram element.
2. Drag the quick linker symbol to the target element view, or to an area, where you want to create the target element view; and in the context menu, select the element.

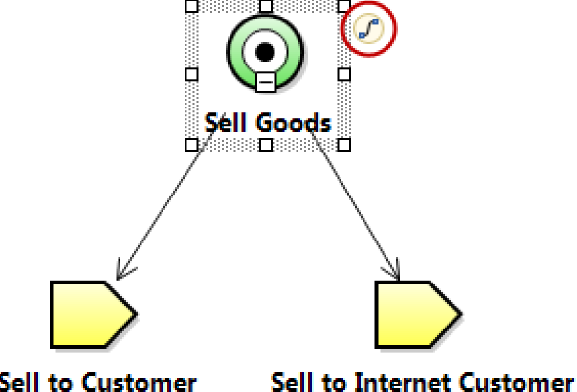


Figure 2.21 Quicklinker

You can insert an element view in between two elements connected with a flow by placing the view on the Flow element: the Flow will be split in two Flows with the element in between.

2.2.4.3 Aligning Element Views

To align element views in a diagram editor according to one of the elements, do the following:

1. Select the elements on the canvas: Either drag a select box around the elements or use the Ctrl + left-click.

   The align is performed according to the primary element. The primary element is the element in the selection with white border points. To mark another element as the primary element of the selection, press and hold the Shift key and left-click the element.

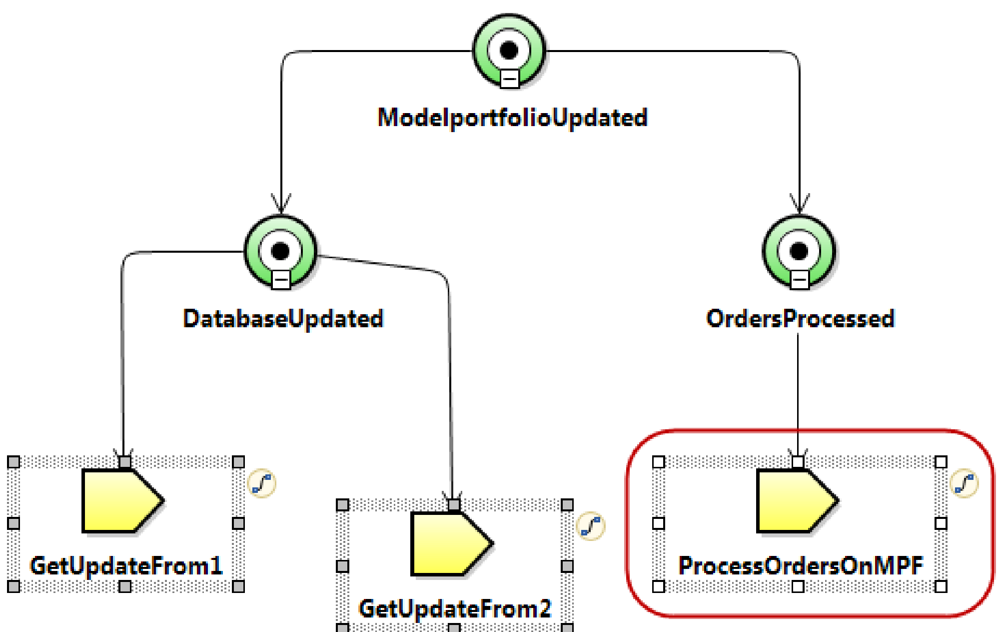


Figure 2.22 Primary element view in a selection

2. Click the Alignment ( ) button on the editor toolbar and pick the align type.

2.2.4.4 Matching Element Views Size

To match sizes of several diagram elements to the size of one element, do the following:

1. Select the elements on the canvas
2. Select the primary element: it is the properties, that is, width and height, of this elements that are applied to the other selected elements.

   The primary element is the element in the selection with white border points. To mark another element as the primary element of the selection, press and hold the Shift key and left-click the element.
3. On the main toolbar, click the Size ( ) button, and select the resizing strategy:
   - Match Width to resize the elements' width according to the primary element
   - Match Height to resize the elements' height according to the primary element. To restore the original size of a diagram element, right-click it and select Auto Resize.

2.2.4.5 Spreading Diagram Element Views

When spreading elements, the selected element are rearranged in such a way that the space they occupy remains unchanged while the "gaps" between the elements become equal.

To spread diagram elements in a diagram editor evenly, do the following:

1. Select the desired elements on the canvas: Either drag a select box around the elements or use the Ctrl + left-click.
2. Click the Spreading ( ) button on the editor toolbar and pick the spreading strategy:
   - Horizontal Spread Evenly spreads the elements horizontally.
   - Vertical Spread Evenly spreads the elements vertically.

2.2.4.6 Changing the Process Element Type

To change the type of a Process diagram element, do the following:

1. Right-click the element view.
2. In the context menu, click
   - *Change To* to change the element type: the original parameters will be dropped.
   - *Change Type* to change the type of the task: Original parameter values will be reused and, if incompatible, the system will detect an error. You can perform task parameter clean-up to remove such parameters. When changing other element types, the original parameters are dropped.

2.2.4.7 Deleting Elements from Diagrams

When deleting an element view in a diagram, you can either delete the element with all its view or delete only the view and preserve the respective element.

> **Note:** You will not be able to add a breakpoint to elements with no diagram view (icon representing the element) when debugging.

To remove an element or its view:

1. In the respective diagram (opened in the respective diagram editor), locate the desired element view.
2. Right-click the element view:
   - Click Delete From Diagram to delete only the element view from the diagram;
   - Click Delete Element to delete the element (as a consequence all its diagram views are removed).

2.2.4.8 Snapping to Grid

The grid provides helping lines and allows you to align element views on diagrams automatically.

To set and display the grid, do the following:

1. On the main menu, go to Window > Preferences.
2. In the Preferences dialog box, expand Process Design Suite > Modeling, and click Alignment.
3. On the Alignment page, set the grid properties:
   - Snap to grid: automatically aligns diagram elements to the grid;
   - Show grid: displays the grid;
   - define the grid size in pixels;
   - select Snap to geometry to snap element to the grid guidelines when moved
     When moving a diagram view or element, guidelines indicate a border (top, bottom, left, or right) of another diagram icon.
4. Click OK.

2.2.4.9 Customizing the Palette

You have activated a diagram editor.

1. Right-click anywhere on the palette (apart from its title bar) and select **Settings**.

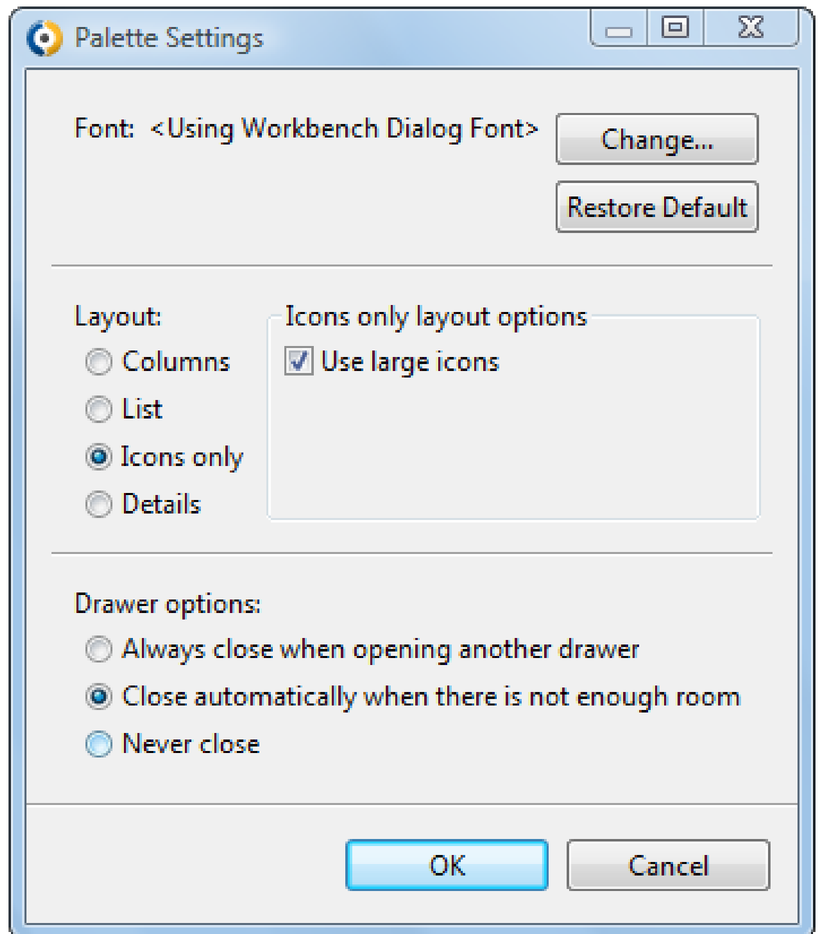


Figure 2.23 Customizing Palette

2. In the dialog box, define the palette properties.

3. Click OK to apply the settings.

2.2.4.10 Displaying and Hiding Page Borders in Diagrams

To display page borders in diagrams, do the following:

1. Click Window > Preferences.

2. In the Preferences dialog box, expand Process Design Suite.

3. In the left pane, click Modeling.

4. Under Show Page Borders on Diagram, select the respective option.

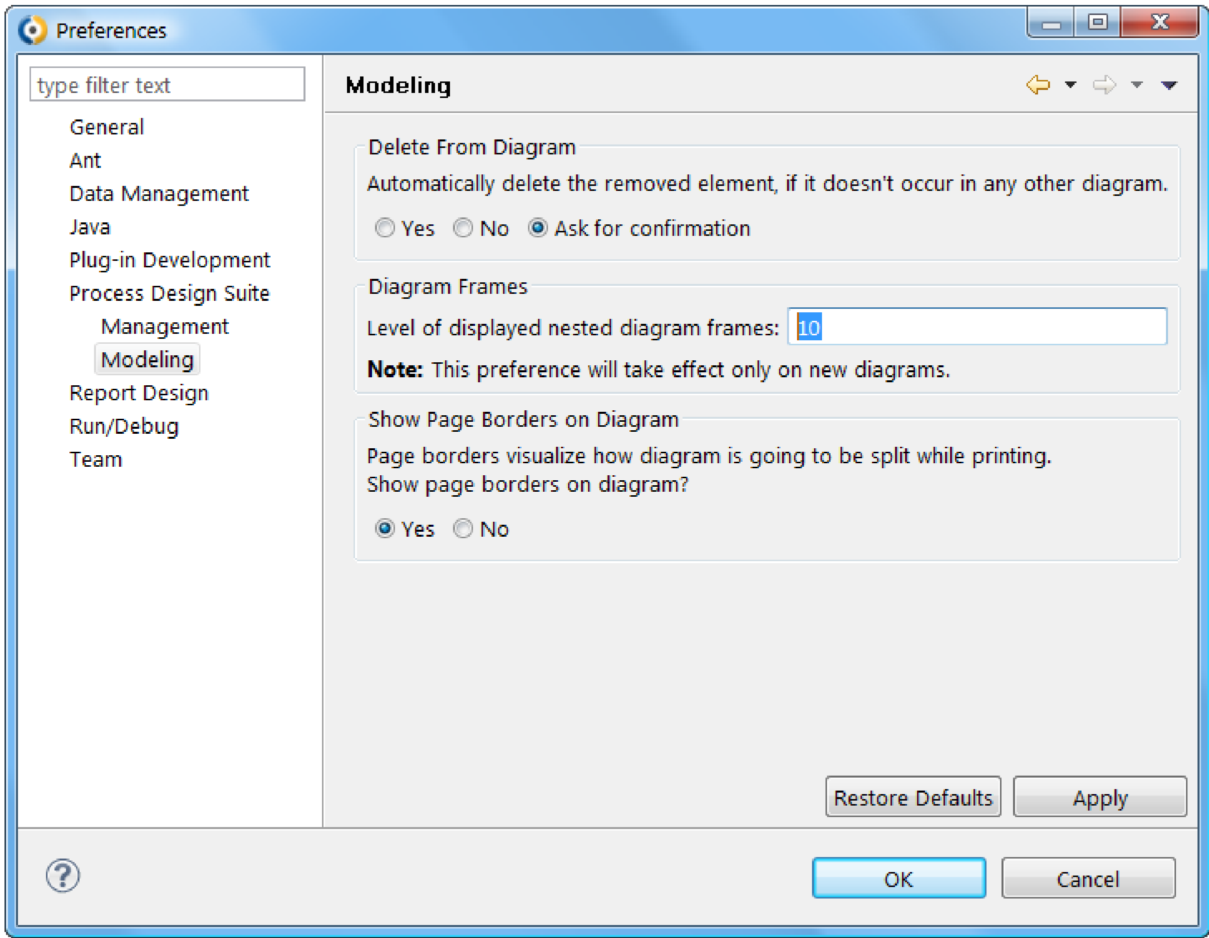


Figure 2.24 Setting the borders

2.2.4.11 Limiting Diagram Frame Nesting Level

You can restrict the level of allowed diagram frame nesting in **Window** > **Preferences** and then **Process Design Suite** > **Modeling**.

2.2.4.12 Inserting Hyperlinks

To insert a hyperlink, do the following:

1. Open the respective diagram in the diagram editor.
2. On the palette, click the respective hyperlink button.
3. Click the canvas area, where you want to place the hyperlink.

Alternatively right-click anywhere on the canvas, and select New and the desired Hyperlink or drag the respective element onto the canvas (if applicable, select Create Hyperlink in the displayed menu).

2.2.4.13 Switching Iconic and Decorative Notation

Some diagram views can be displayed in two notation variants:

- Iconic notation is compact and used for element notation by default
- Decorative notation is descriptive and contains further details about the particular element.

Notation variants (iconic and decorative) are available for the following element views:

- goals;
- plans;
- roles;
- organization units.

To change notation of a diagram element, do the following:

1. Open the diagram with the desired element view.
2. Right-click the element view and select Notation and the type of notation (Iconic/Decorative).

When applying decorative notation, you may chose the details, which should be shown: in the context menu of the element icon select Compartments and select the compartments to be displayed.

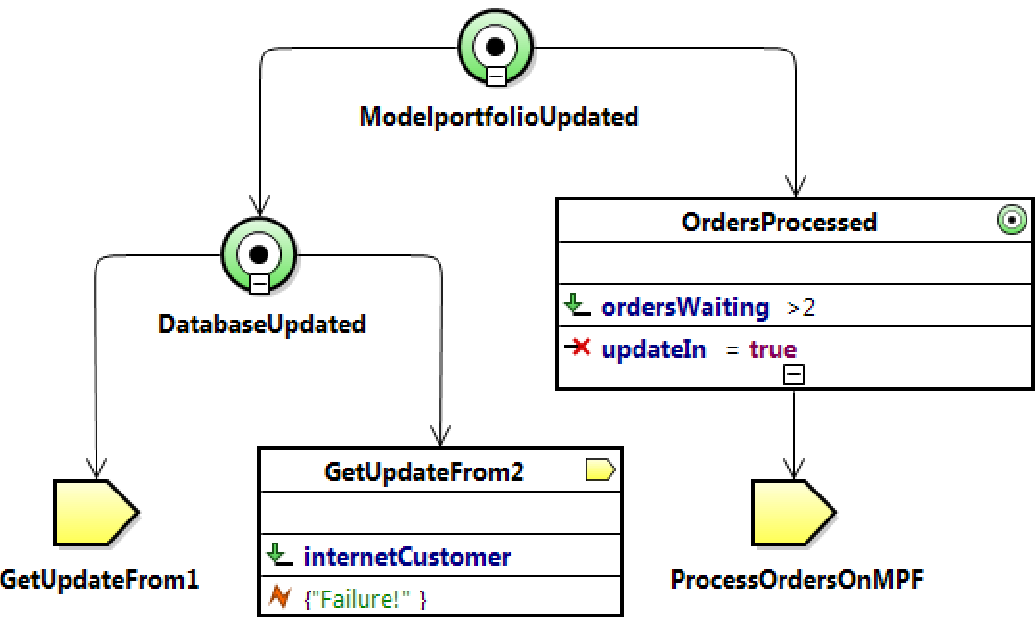


Figure 2.25 Goal hierarchy with views in decorative and iconic notation

2.2.4.14 Hiding and Displaying Compartments of Diagram Elements

You can hide the task type compartments of the task elements, inherited fields in a record, etc.

To hide or display the compartments on a diagram element, do the following:

1. Right-click the element on the canvas.
2. In the context menu, select Compartment and select the compartment to hide or show.

Note that the element settings can be overridden by global settings: to access the Settings go to **Window** > **Preferences** and in the dialog go to **Modeling** > **Appearance**. The setting are in the **Compartment Settings** section.

2.2.4.15 Changing Line Style

The line style defines the behavior of line elements and their bending.

The following line styles are available:

- Direct: No bend points are allowed (only straight lines are used).

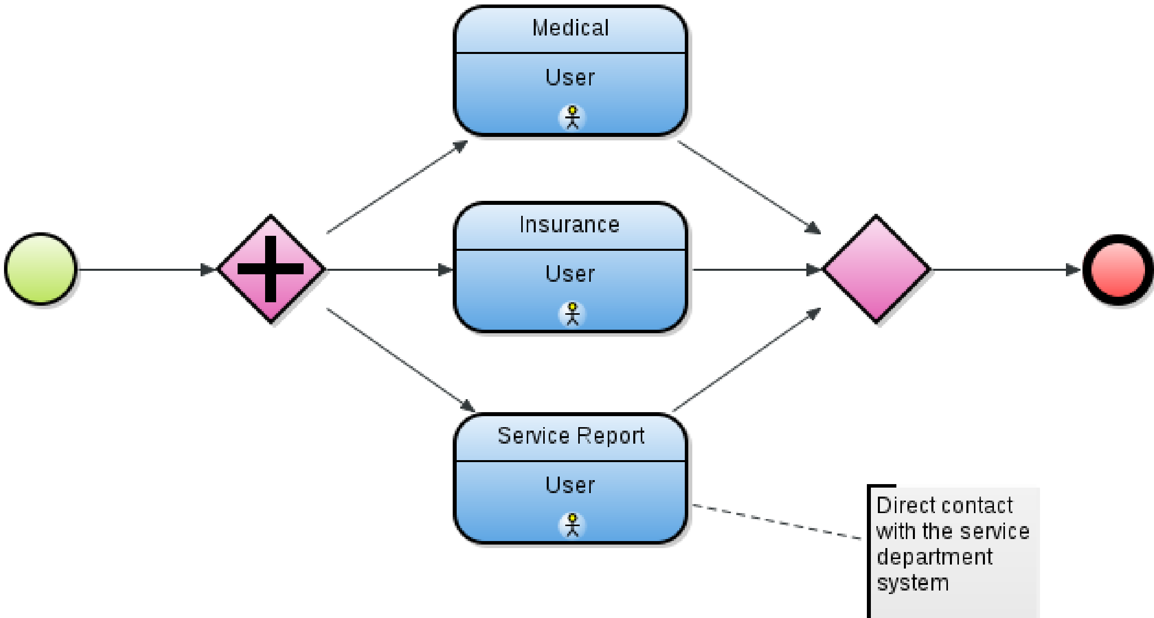


Figure 2.26 Diagram with direct line style

- Tree: Lines are bent only automatically and only at the right angle.

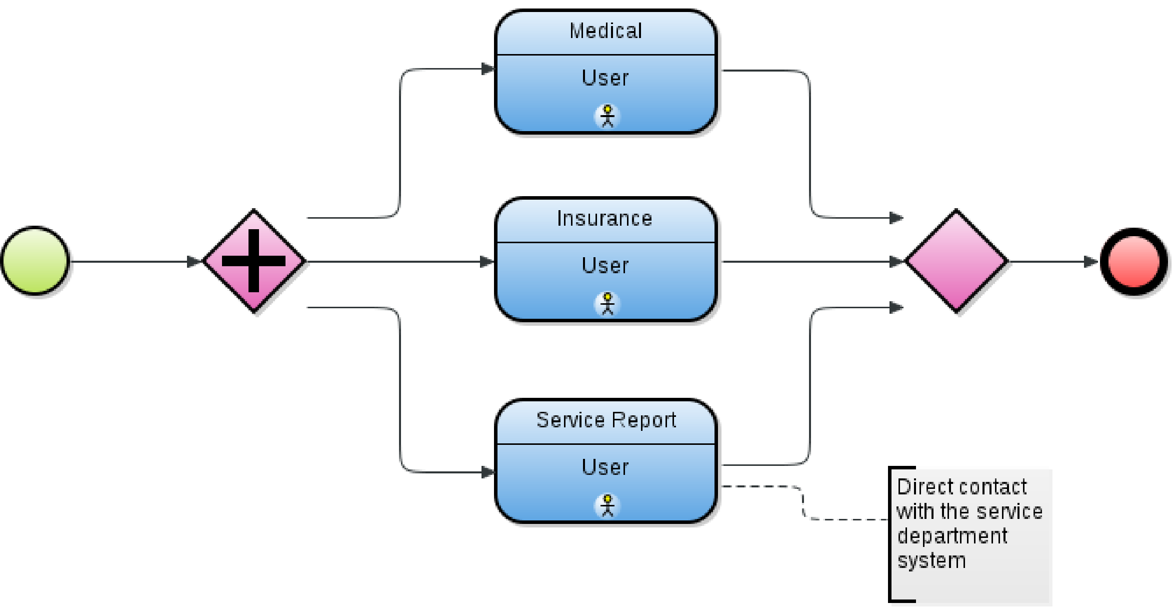


Figure 2.27 Diagram with tree line style

- Custom: Lines are direct and the possibility to add bendpoints

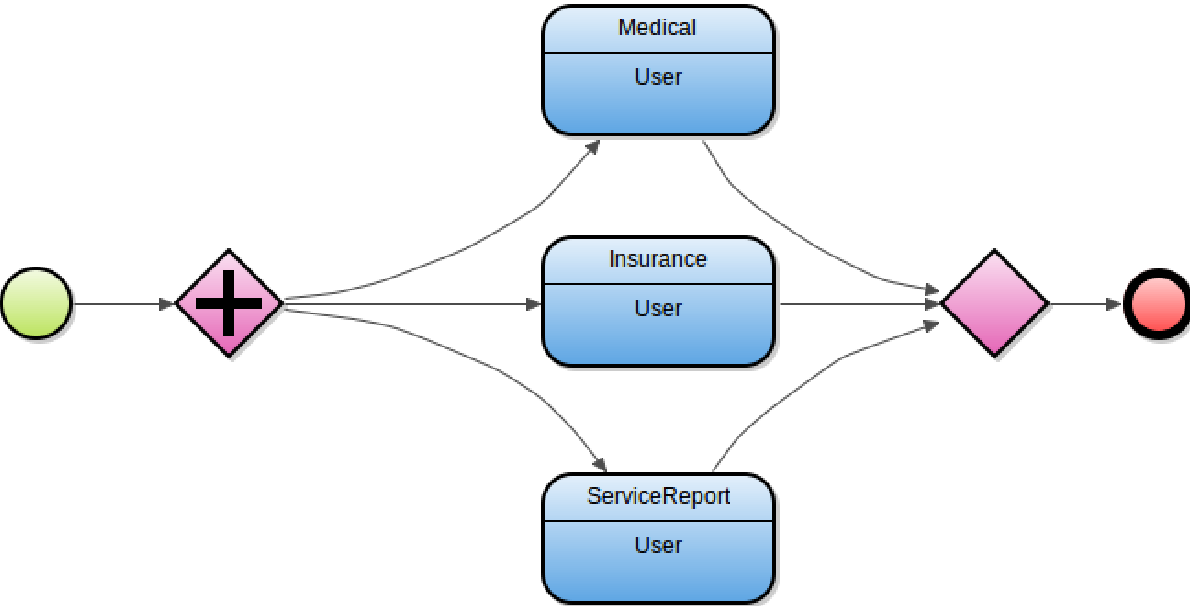


Figure 2.28 Diagram with custom line style

To change the style of line, do the following:

1. Select the line element in the diagram.

2. In the Properties view, click the Appearance tab.

3. In the Line style drop-down menu, select the style to apply on the line element.

Note that the default line style is defined by global settings: to access the Settings go to **Window** > **Preferences** and in the dialog go to **Modeling** > **Appearance**. The setting are in the **Line Style** section.

2.2.4.16 Formatting Settings

Formatting settings define how the diagram elements are depicted and organized in the diagram.

Every diagram element defines a set of its formatting properties, such as its fill color, outline color, text font, displayed compartments etc. (refer to Defining Custom Format of Elements and Compartments of Diagram Elements). Apart from the element formatting properties, diagram editors define global formatting settings, which override the local settings and define additional formatting properties.

2.2.4.16.1 Using Custom Format Style

To get and apply a custom format style on an element view, do the following:

1. On the canvas, select the element with the format you wish to apply to another element.

2. In the main toolbar, click the Get Style button ( ).

3. On the canvas, select the element to apply the format style to.

4. In the main toolbar, click the Apply Style ( ) button.

The format style is applied to the element. The custom format style may not be visible since custom formatting is overridden by the monochrome and status formatting style.

2.2.4.16.2 Activating Monochrome Style

Monochrome style is a two-color style applied to diagram elements. By default the style is set to black-and-white.

It overrides any custom style (if applied to diagram elements with custom color style, the elements loose their coloring). However, it does not influence the status coloring.

To activate or deactivate and modify the monochrome style, do the following:

1. Go to Window > Preferences.

2. In the left part of the Preferences dialog box, expand Process Design Suite > Modeling > Appearance.

3. On the Appearance page on the right, select or unselect Use monochrome color style in Diagrams.

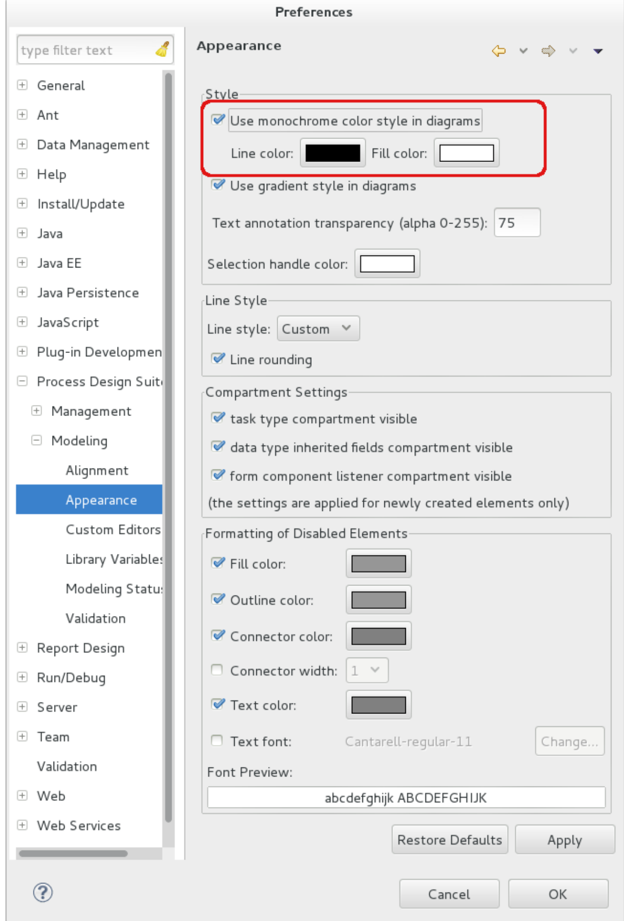


Figure 2.29 Appearance page of the Preferences dialog box

Optionally, customize the colors used by the style in the Monochrome Style Details area below (available if the Use monochrome style in diagrams checkbox is selected).

2.2.4.16.3 Activating Gradient Style

To deactivate or activate the gradient style of diagram elements:

1. Go to Window > Preferences.
2. In the left part of the Preferences dialog box, expand Process Design Suite > Modeling > Appearance.
3. On the Appearance page on the right, select or unselect Use gradient style in diagrams.

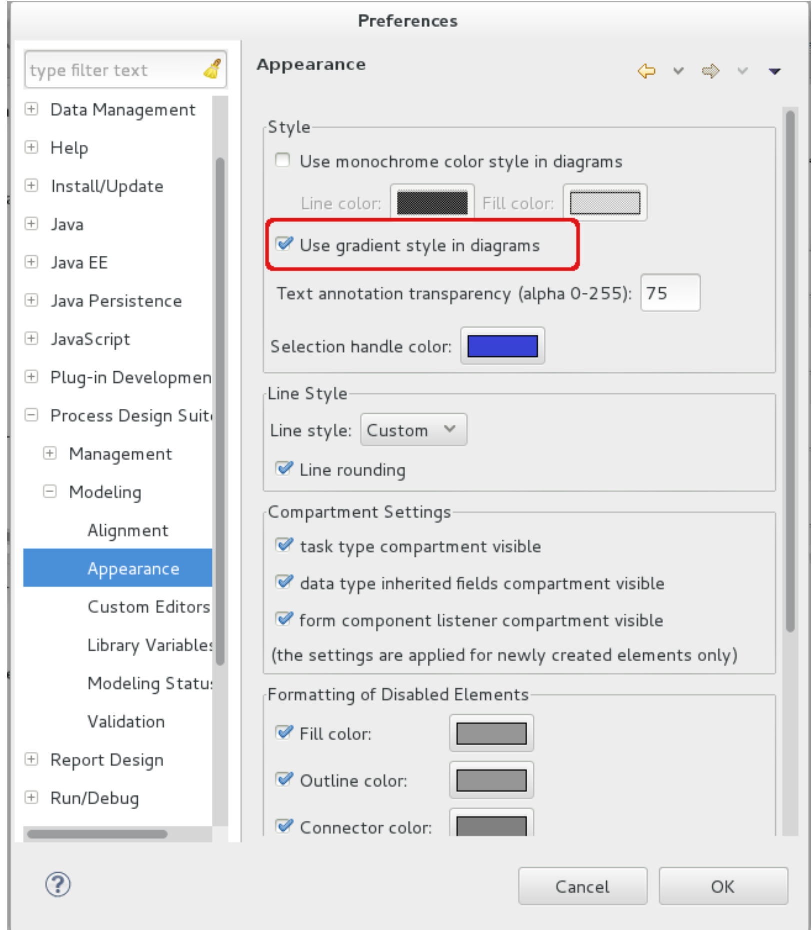


Figure 2.30 Appearance page of the Preferences dialog box

2.2.4.16.4 Activating Transparency on Text Annotations

To deactivate or activate and define transparency property of Text Annotations:

1. Go to Window > Preferences.
2. In the left part of the Preferences dialog box, expand Process Design Suite > Modeling > Appearance.
3. On the Appearance page on the right, select or unselect Text annotation transparency (alpha 0-255) and enter a valid value (255 stands for no transparency).
4. Click **Apply**.

2.2.4.16.5 Changing Primary Element Color

To change the color of the primary element, do the following:

1. Go to Window > Preferences.
2. In the left part of the Preferences dialog box, expand Process Design Suite > Modeling > Appearance.
3. On the Appearance page on the right, click the Selection handle color box and select a color.

2.2.4.16.6 Changing Default Line Style

The line style defines the style how the line elements bent.

To change the default line style, do the following:

1. Go to Window > Preferences.
2. In the left part of the Preferences dialog box, expand Process Design Suite > Modeling > Appearance.
3. On the Appearance page on the right, select the default line style in the drop-down menu. You can define enable or disable the line rounding using the checkbox below.
4. Click **Apply**.

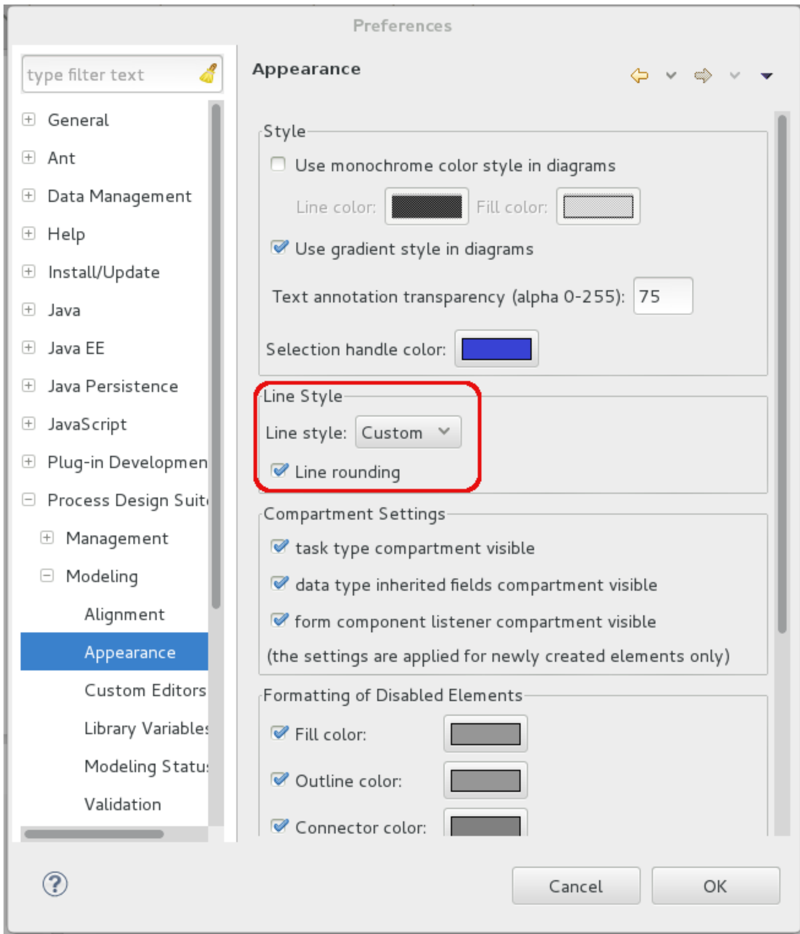


Figure 2.31 Select with the Primary Element in custom color

Note that the default line style is applied only on new line elements. The style of existing line elements remains unchanged.

2.2.4.16.7 Changing Formatting of Element Views

You can define the formatting of every diagram element view. Such custom formatting is however overridden by the monochrome and modeling status format.

To customize the formatting of a diagram element, do the following:

1. Select the element on the canvas.
2. In the Properties view, display the Format tab.
3. On the Format tab, select the formatting attribute and choose the formatting property. Alternatively, use the format buttons available in the editor main toolbar.

2.2.4.17 Diagram Printing

When printing Diagrams, the system follows the global page setup by default. However, you can override its setting in a local setup defined per Diagram.

You can export and import local page setups so you can share them with other users and print your Diagrams with the same settings.

2.2.4.17.1 Defining Global Page Setup

The setting is applied on every diagram printing and PDF export unless the local page setup is applied.

To define the global page setup, go to File > Page Setup.

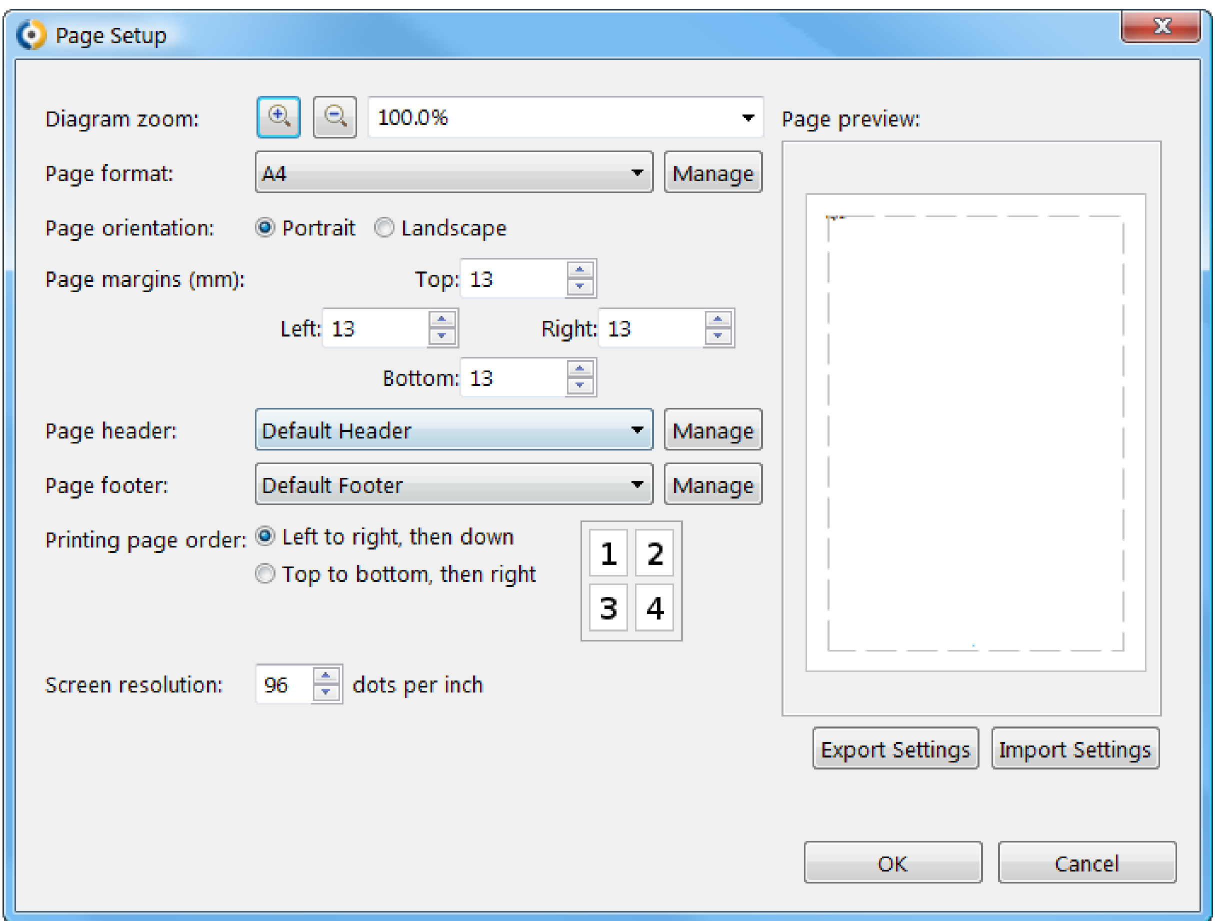


Figure 2.32 Page setup

2.2.4.17.2 Defining Local Printing Setup

To define printing setup for a Diagram, set the printing properties in the Printing tab of the Properties view. Mind that this setting is applied whenever the diagram is included in a PDF or image export and you need to disable the local page setup to re-apply the global page setup on the diagram.

2.2.4.17.3 Defining Header and Footer

To define header or footer in a global or local page setup, do the following:

1. Go to the respective page setup (for global page setup: File > Page Setup; for local page setup: Printing tab of the Properties view).
2. Click the Manage button next to the Page header/Page footer drop-down box.
3. In the displayed dialog, click the respective tab label (Page Headers or Page Footers).
   - To add a new definition, click the Add button.
   - To edit an existing definition, select the definition and click Edit.
4. In the displayed dialog (Page Header or Page Footer) define:
   - Name of the definition
   - Font to be used (click Change to define font settings)
   - Number of lines
   - Content of the left, middle, and right part of the header or footer in the areas below: Click the respective button above, to enter variable data.

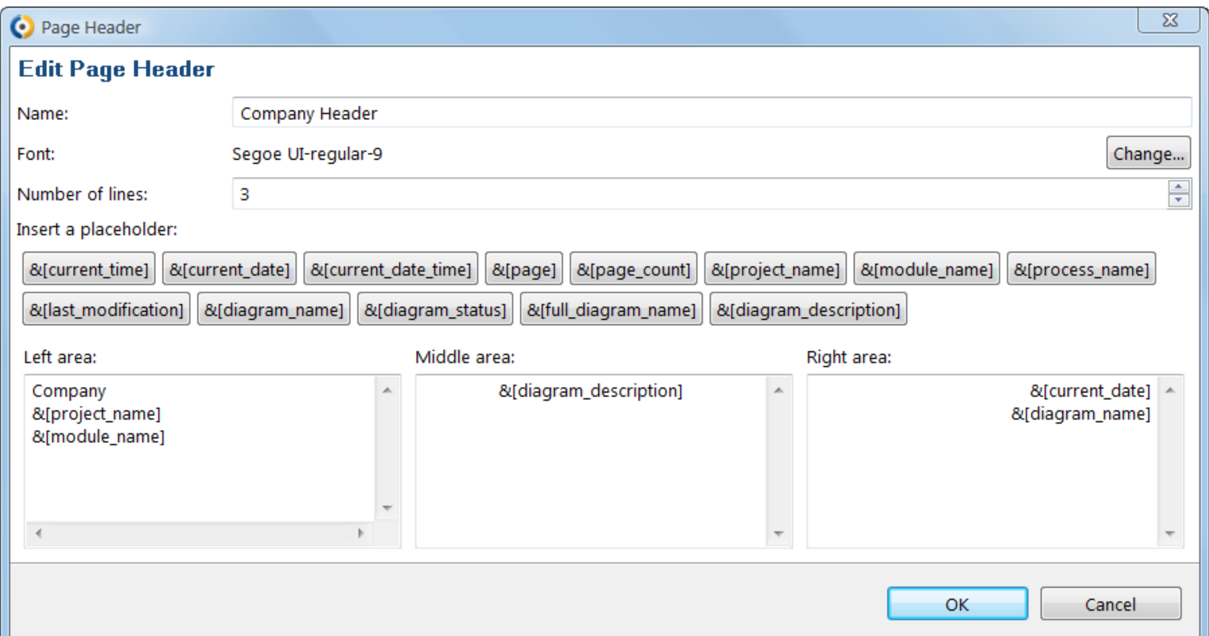


Figure 2.33 Page header setup

2.2.4.17.4 Exporting and Importing Page Setup

To export or import the global page setup, do the following:

1. Go to File > Page Setup.
2. In the Page Setup dialog box, click the Export or Import button.

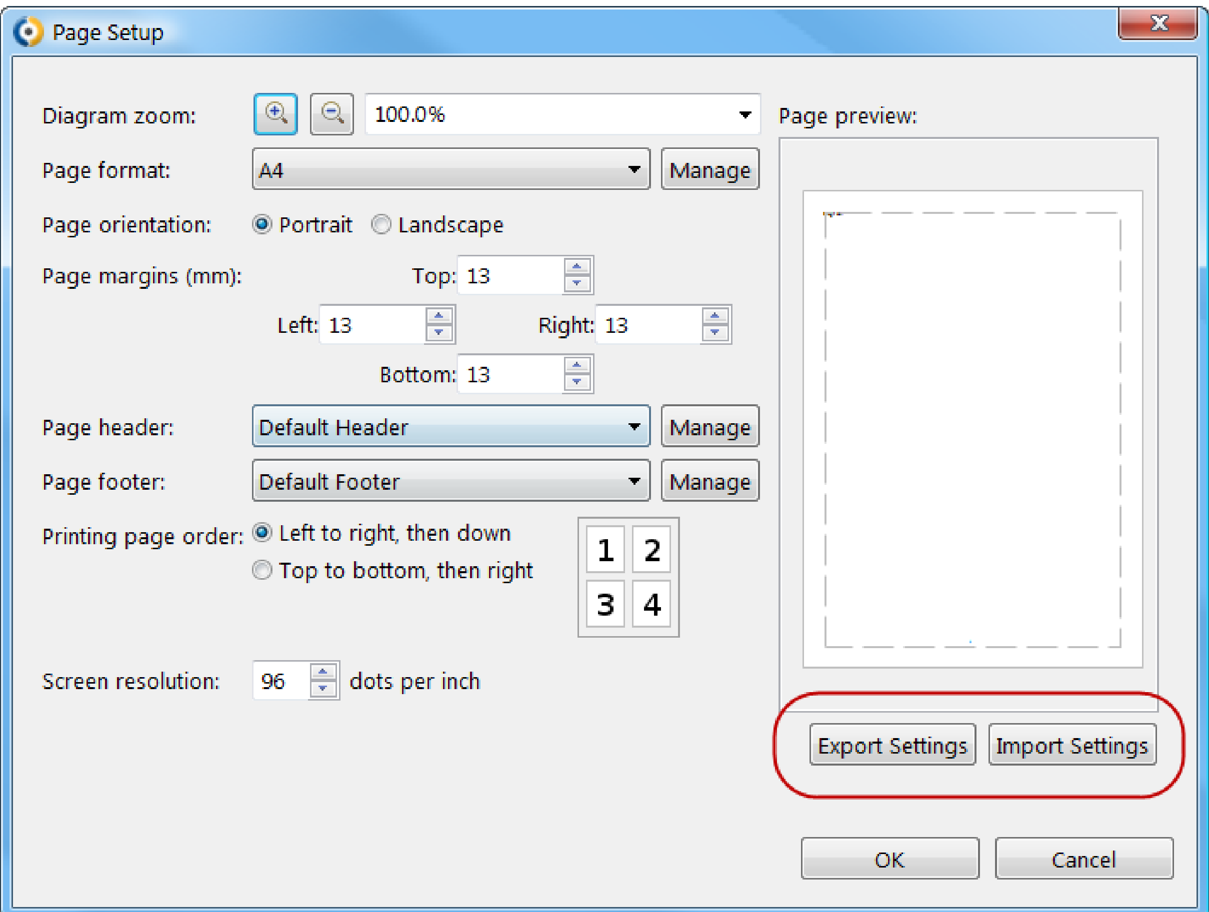


Figure 2.34 Exporting/Importing page setup

3. In the displayed dialog box, define the location and name of the export/import file.

2.2.4.17.5 Exporting Diagrams as Images

The Process Design Suite allows exporting diagrams as images, and groups of diagrams as a structured PDF document.

A diagram can be exported into the PDF format and as an SVG, PNG, JPG, or BMP image.

> **Important:** When exporting a diagram to SVG, make sure the diagram font is installed on your system. The default font used in diagrams is Arial, which is installed on MS Windows by default. Hence this applies especially to users of other operating systems.
>
> Alternatively you can change the diagram font for PDS: go to Window > Preferences and in the Preferences dialog, go to General > Appearance > Colors and Fonts, and under the Process Design Suite, select Diagram Font and click Edit.
>
> Note that the exported SVG diagram has the clip attribute on the <svg:text> Element. The attribute might cause the text to be cut off. Delete the attribute in your SVG to resolve the issue.

2.2.4.17.6 Printing Diagram Element Selection

To print a selection of diagram elements:

1. Go to File > Print.
2. Select the Print the selected elements option.

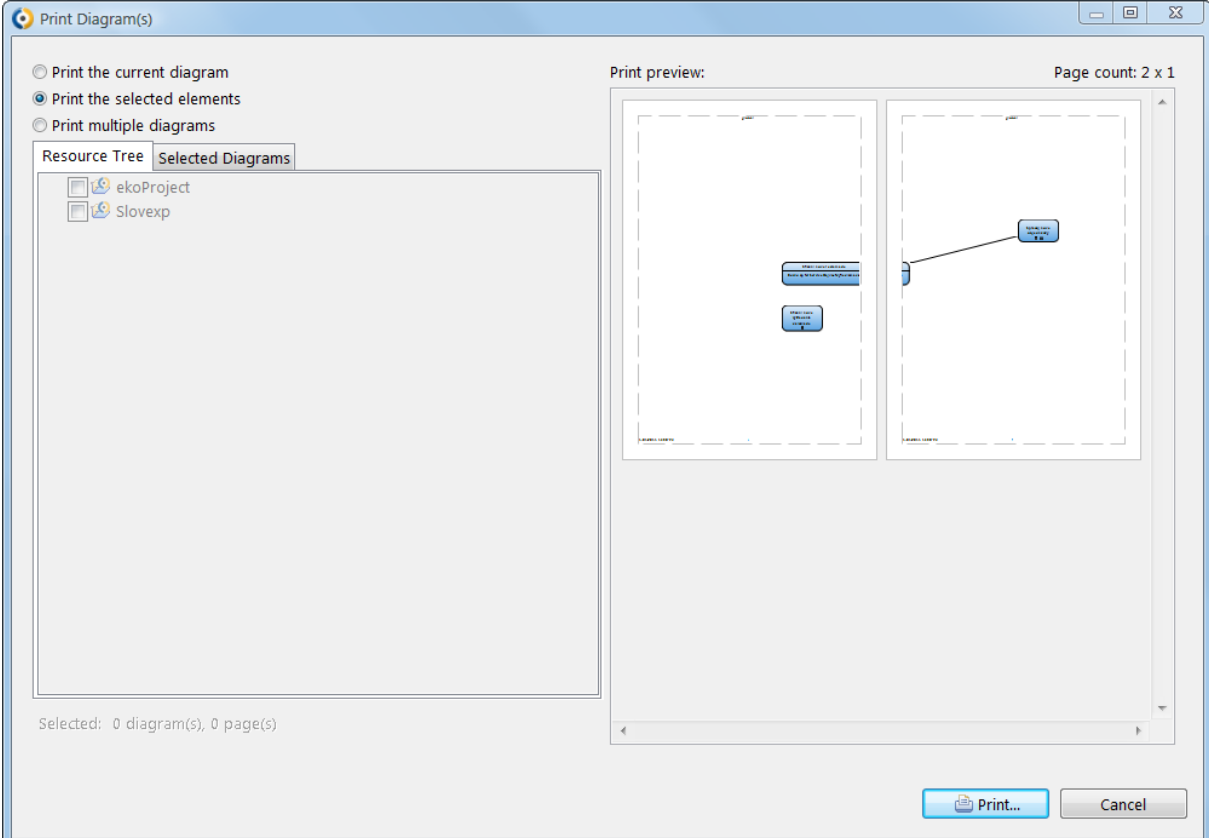


Figure 2.35 Print Diagrams dialog with the Print the selected elements selectedCaption text

3. Check the print preview on the right.

2.2.4.17.6.1 Printing Multiple Diagrams

Note: The page setup applied to individual diagrams is applied also on its printing (that is, the global page setup is applied by default; if a diagram defines a local page setup, for printing of this diagram the local page setup is applied, see Page Setup).

To print several diagrams from a project or Module, do the following:

2.2.4.18 Applying Automatic Layout in Diagrams

The automatic formatting of diagram views, does the following:

- Arranges elements in hierarchy levels
- Resolves any overlapping elements

To automatically adjust the element layout in your diagram, right-click into the diagram canvas and select **Layout**.

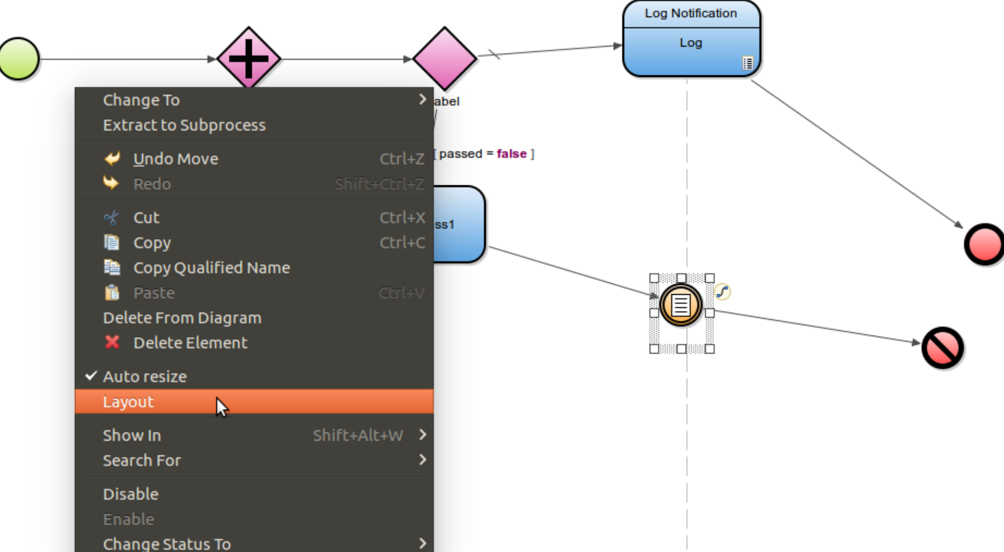


Figure 2.36 Performing autolayout

Auto layout automatically adjusts position of the diagram elements in a vertical manner.

> **Note:** If an element displayed in a process diagram contains a boundary event, the layout feature is disabled.

If you want to adjust the layout of the elements in a Diagram whenever the diagram is opened, do the following:

1. Open the definition file for editing (double-click it in the GO-BPMN Explorer)
2. Select the Diagram in the Outline view.
3. On the *Detail* tab in the Properties view of the diagram, select *Auto layout*.
4. Close the editor with the diagram and open it anew.

The diagram elements will be laid out automatically anew whenever the diagram is opened: Note that the editor with the diagram will be marked as dirty (with the asterisk ∗ sign).

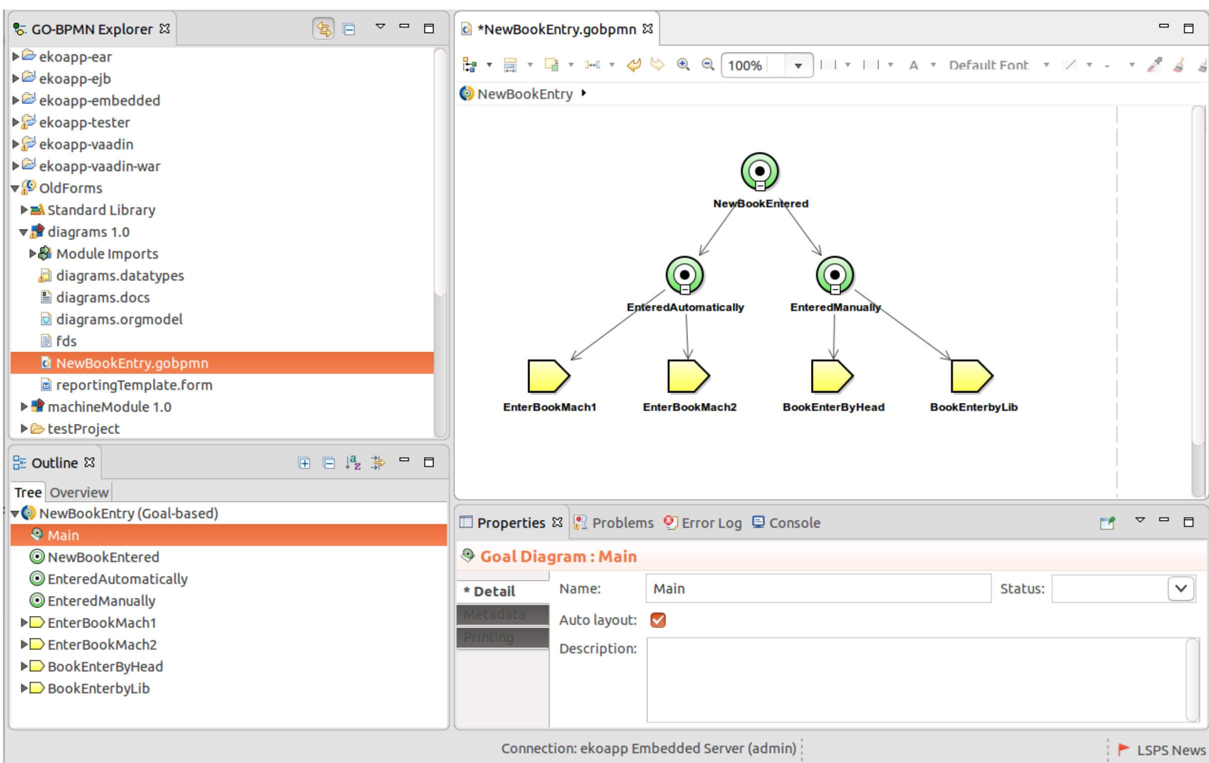


Figure 2.37 Goal diagram opened after Auto-layout was selected

2.2.5 Comparing Resources

You can compare the resources you create in PDS with a dedicated Comparing tool that visualizes the detected differences and allows you to merge the differences among the compared resources. It detects changes with semantic significance, such as, a removed task, as well as other changes, such as changes in element layout, size, etc.

You can compare two or three resources of the same type, and that, projects, modules, or configurations and definitions.

2.2.5.1 Comparing Two Resources

To compare two resources, do the following:

1. Select the resources (Ctrl + left-click to select) in the GO-BPMN Explorer.
2. Right-click one of the selected resources and click **Compare With** > **Each Other**.

   When comparing Modules or projects, you will see the files with detected differences. To display the differences on the files, click the file: in the displayed diff, use the buttons in the toolbar to navigate through the differences.

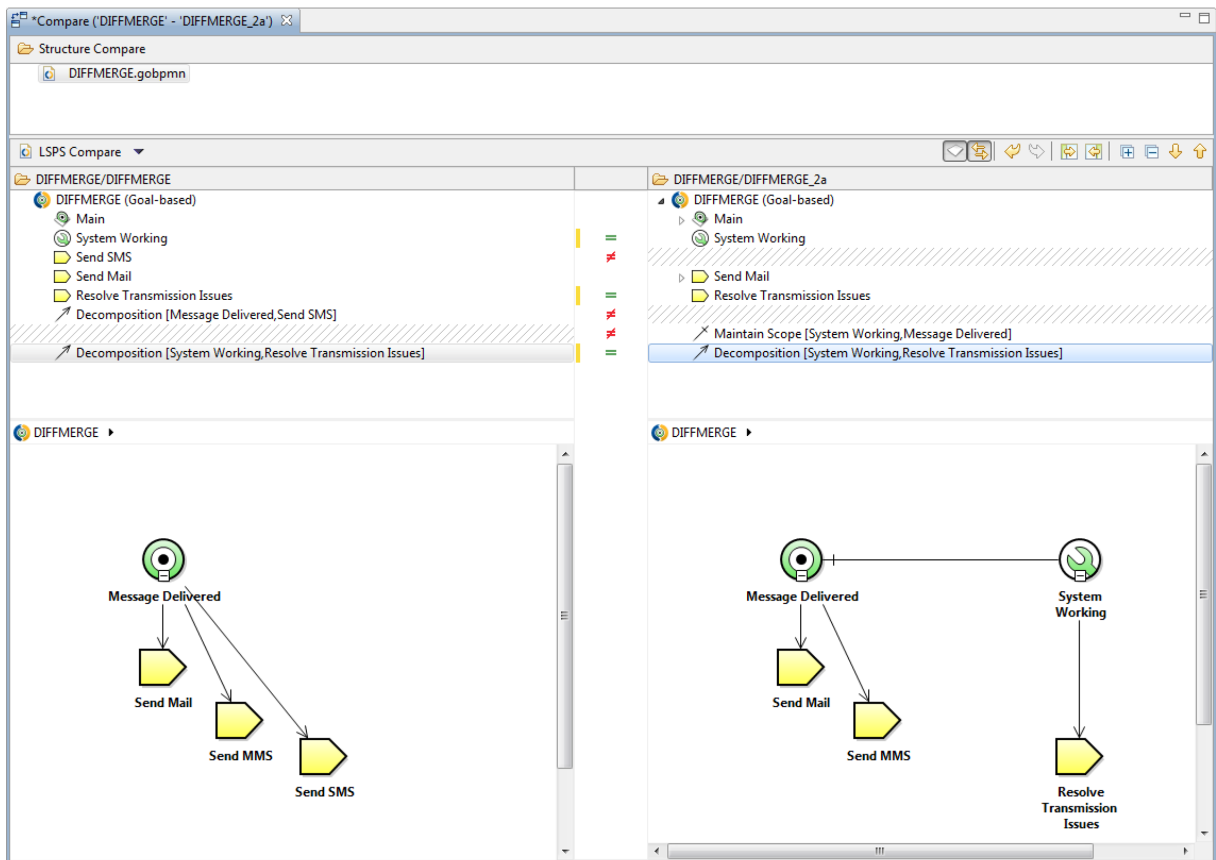


Figure 2.38 Comparing goal-based Processes

3. To apply a difference on either resource, right-click the difference and select the required action from the context menu.

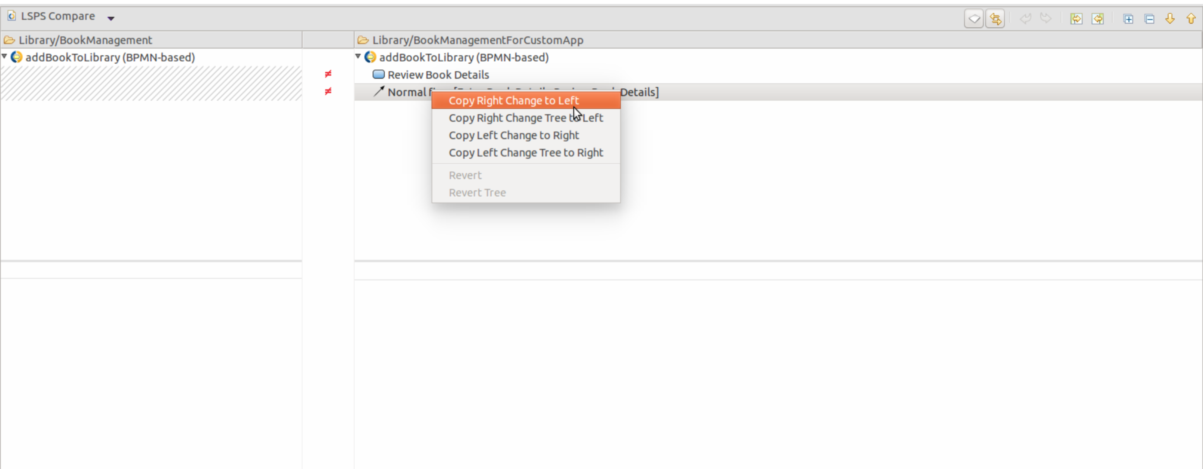
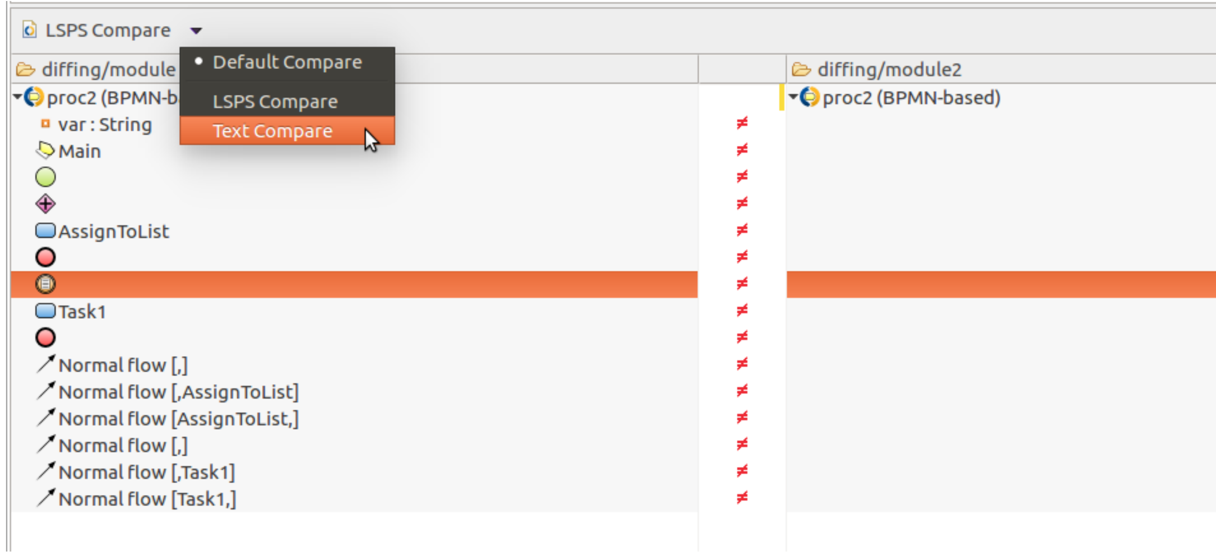


Figure 2.39 Resolving difference

To switch from the LSPS Compare to a text compare, click the arrow on top of the comparison table and click Text Compare.



2.2.5.2 Comparing Three Resources

To compare three resources, do the following:

1. In the GO-BPMN Explorer, select three resources of the same type.
2. Right-click one of the selected resources, and select Compare With > Each Other.
3. In the Select Common Ancestor dialog box, select the common ancestor, the original resource, and click OK.

   When comparing three resources, the following takes place:

   - The ancestor resource and the first modification are compared.
   - The ancestor and the second modification are compared.
   - Results of both previous comparisons are compared and visualized in the table of the Comparison Editor.

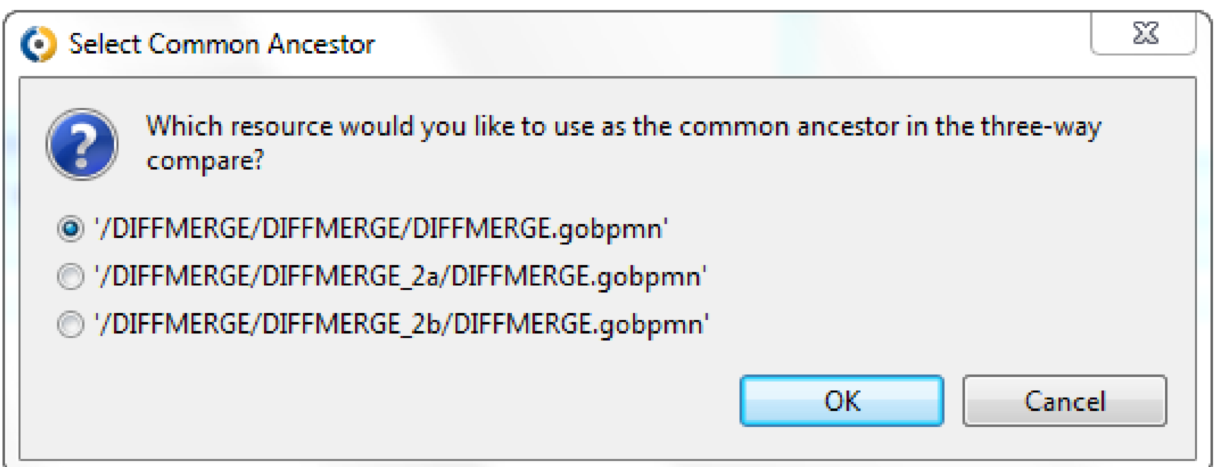


Figure 2.40 Defining the ancestor resource

When comparing Modules or projects, the editor will display files with differences. To display the differences, click the file.

Note that changes that are not in either of the resources are marked with ≠ while identical changes are marked with = .

4. Merge required changes into the respective modifications (You cannot merge a difference into the ancestor resource).

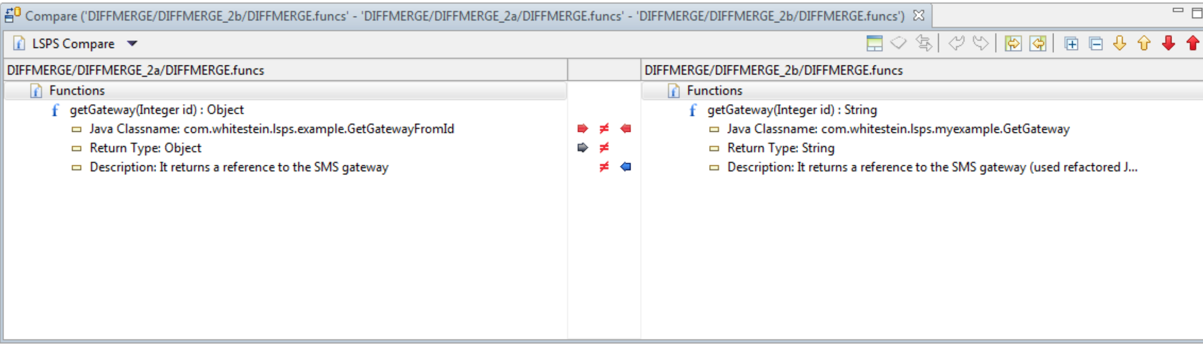
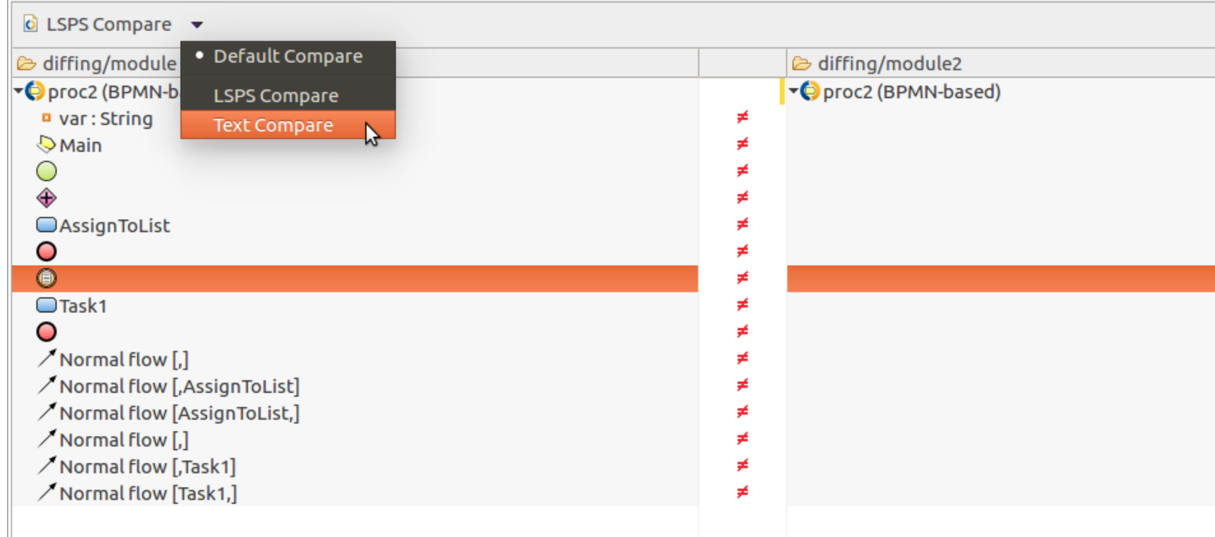


Figure 2.41 Comparing three resources

To switch from the LSPS Compare to a text compare, click the arrow on top of the comparison table and click Text Compare.



Marker	Description
	The left modification adds an element to the ancestor. This change is not in conflict with those ones in the right modification.
	The left modification removes an element from the ancestor. This change is not in conflict with those ones in the right modification.
	The left modification updates an element of the ancestor. This change is not in conflict with those ones in the right modification.
	The left modification adds an element to the ancestor. However, this change is in conflict with those ones in the right modification.
	The left modification removes an element from the ancestor. However, this change is in conflict with those ones in the right modification.

Marker	Description
	The left modification updates an element of the ancestor. However, this change is in conflict with those ones in the right modification.
	The right modification adds an element to the ancestor. This change is not in conflict with those ones in the left modification.
	The right modification removes an element from the ancestor. This change is not in conflict with those ones in the left modification.
	The right modification updates an element of the ancestor. This change is not in conflict with those ones in the left modification.
	The right modification adds an element to the ancestor. However, this change is in conflict with those ones in the left modification.
	The right modification removes an element from the ancestor. However, this change is in conflict with those ones in the left modification.
	The right modification updates an element of the ancestor. However, this change is in conflict with those ones in the left modification.

2.2.5.3 Comparing Version-Controlled Resources

The comparing procedure can vary depending on your version control system. Generally, do the following to compare local changes with the current resource version:

1. In the GO-BPMN Explorer, select the resource.
2. Click Compare With -> Latest from Repository.
3. Synchronize your changes with the Repository if applicable: Right-click the resource and select Team -> Synchronize with Repository.

2.2.5.4 Merging

You can merge the differences in diagram resources just like you would when comparing resources in the Comparison Editor: You may accept or reject the detected differences from any of the resources (baseline or modifications) to create the required merge result. The Comparison Editor is activated and contains comparison results automatically.

A yellow indicator appears on the side when a difference has been merged into one of the resources.

2.2.6 Modeling Status

Modeling Status is a property set for a modeling element, such as, a Process, global variable, constraint, organization role, etc. that serves to clearly annotate an element during the designing phase, for example, as a draft element, or an element that is probably not required without influencing the semantics of the element.

A modeling status definition must define its name and can define presentation properties of elements in that modeling status.

Task priority of a modeling status is implicitly set to Todo so elements with such a modeling Status can be displayed in the Tasks view (refer to Todo and Task Markers).

2.2.6.1 Setting a Modeling Status of an Element

Modeling status of an element is a special formatting indicating the stage of its design development.

To specify modeling status of an element:

1. Activate the respective element Properties view (for example, double-click the desired element in the GO-BPMN Explorer).
2. In the Properties view, on the Detail tab in the Status drop-down box:
   - select one of the available statuses;
   - type a custom status name.

     Tip: Alternatively, select one or several element views in the diagram or in the Outline view and use the Status drop-down box on the toolbar or their context menu.

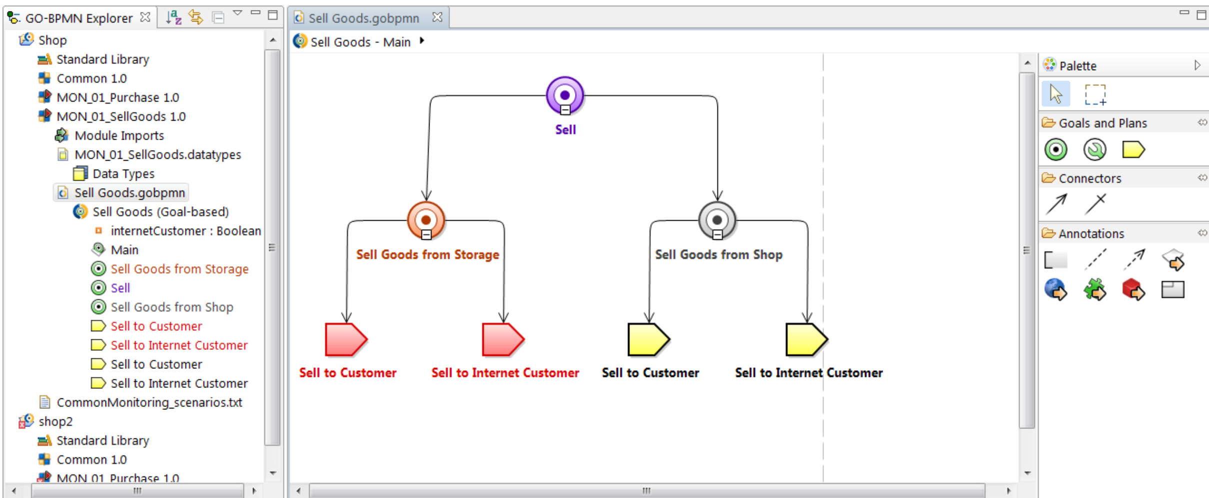


Figure 2.42 Elements with various modeling statuses

The modeling status and the respective formatting is applied to all element views in Diagrams and the element **name** in the Outline view.

2.2.6.2 Defining a Modeling Status

To define a new modeling status, do the following:

1. Go to **Window** -> **Preferences**.
2. In the Preference dialog, go to **Process Design Suite** -> **Modeling** -> **Modeling Status**.
3. Click Add to create a new Modeling Status.

2.2.6.3 Exporting and Importing a Modeling Status

To export or import modeling status definitions:

1. Click Window > Preferences.
2. In the left pane of the Preferences dialog box, expand Process Design Suite and Modeling.
3. Click Modeling Status.
4. Click the Export/Import button on the right.
5. Specify the location and the file name.
6. Click Save/Open.
7. In the Preferences dialog box, click Apply/OK.

2.2.6.4 Disabling Presentation of a Modeling Status

You can disable the properties applied to the presentation of elements by a modeling status in the Diagrams as well as in views, such as, Outline view.

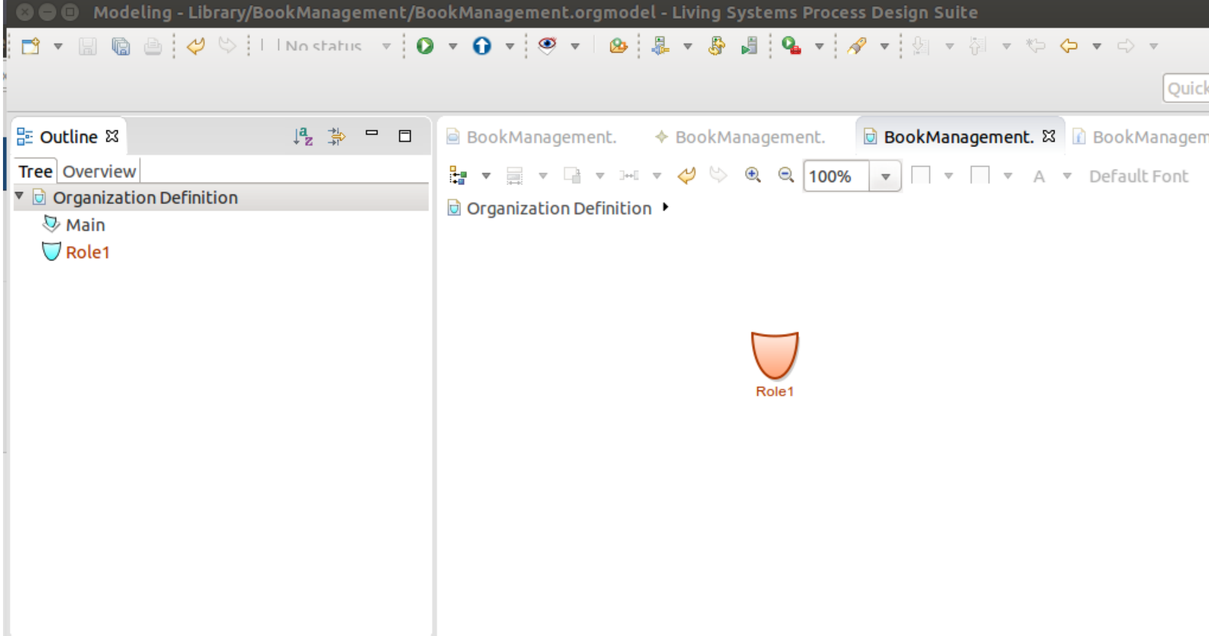


Figure 2.43 Modeling status applied on the element visible in diagrams and in views

To hide the presentation properties:

- in all diagrams, click the **Show Diagram Items** button in the main toolbar and in the context menu, unselect the **Show Modeling Statuses** item.
- in all views, go to **Window** -> **Preferences** and then **Process Design Suite** -> **Modeling** -> **Modeling Status** and unselect **Show modeling statuses in views**.

2.2.7 Todo and Task Markers

To track incomplete expressions and modeling elements, use the //TODO marker. You can parametrize the marker in the form /*TODO <TEXT>*/. The <TEXT> is kept as the marker's description.

You can view all such Markers in the Eclipse *Tasks* view. To open the view, go to Windows > Show View > Tasks.

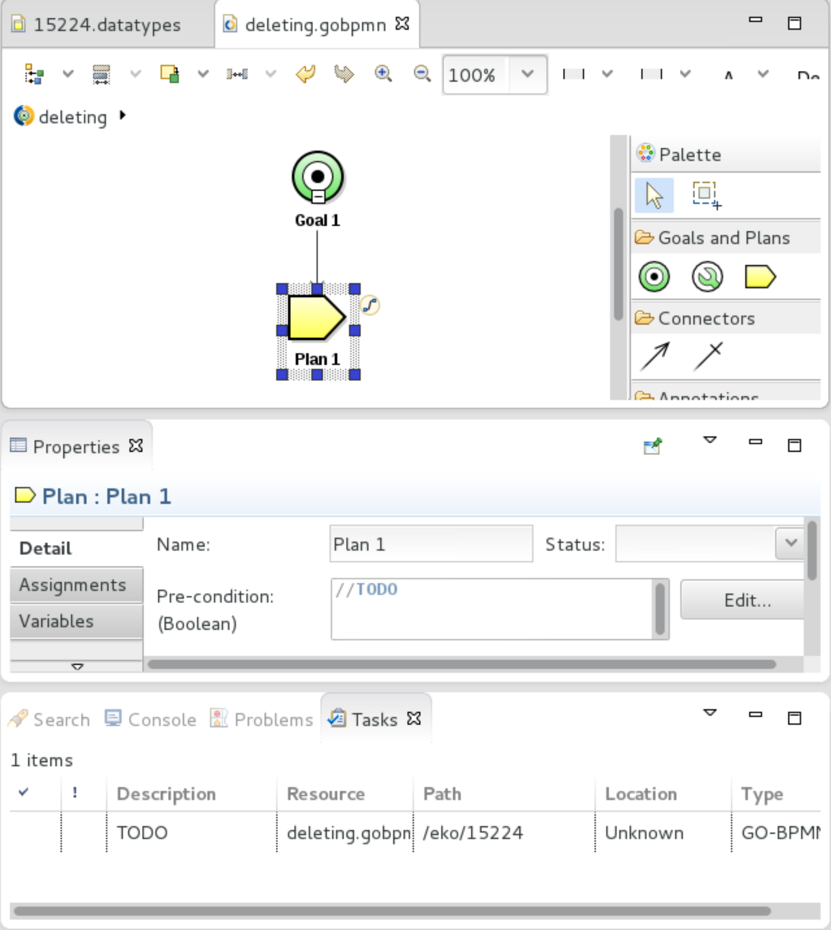


Figure 2.44 The TODO Marker

Alternatively, to track incomplete modeling elements, you can apply a modeling status with a priority to the element: The priority is interpreted as a task marker.

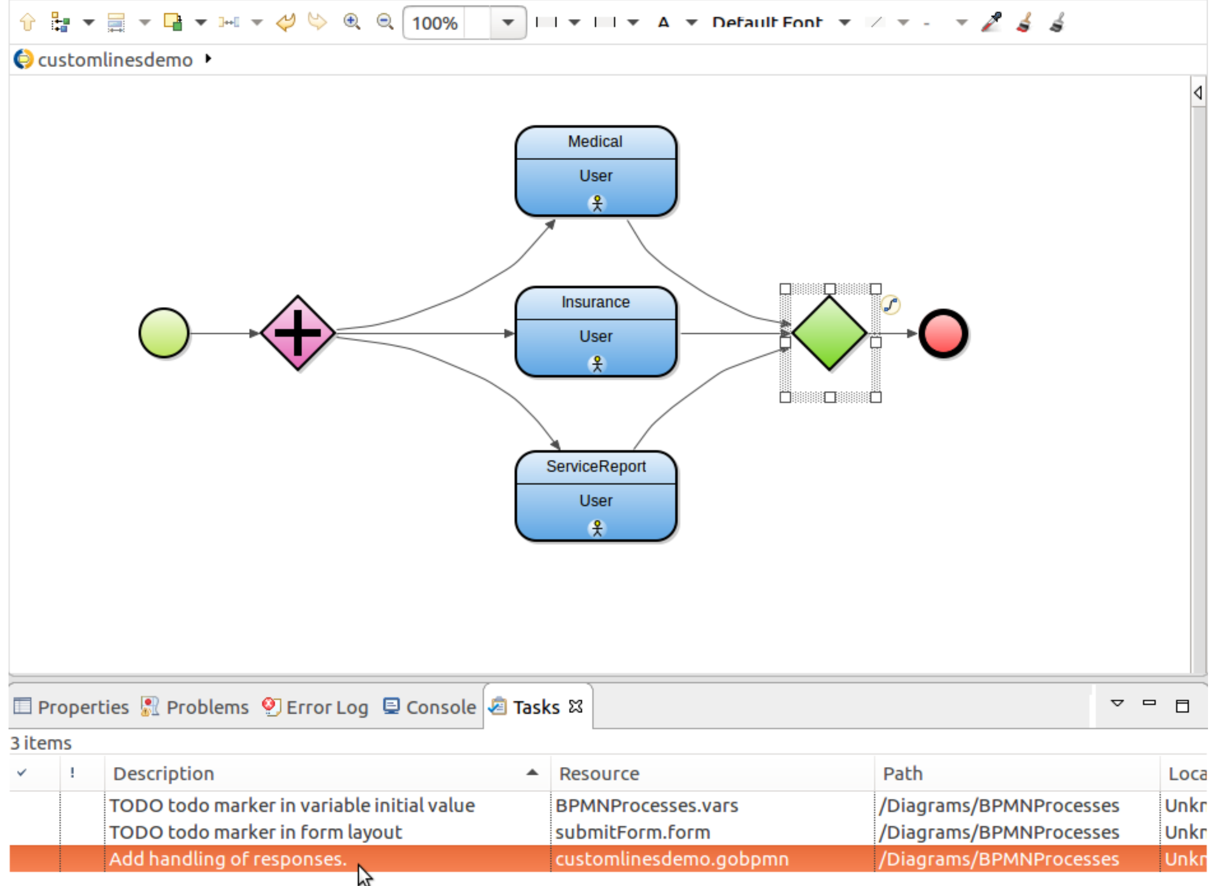


Figure 2.45 Modeling status with the normal priority on an element with its entry in the Tasks view

If elements in a modeling status with a priority are not displayed, check the filtering of the view: in the Task view, click the View Menu ( ) button and select *Configure Contents*. Make sure the filtering is set correctly and set the required priority.

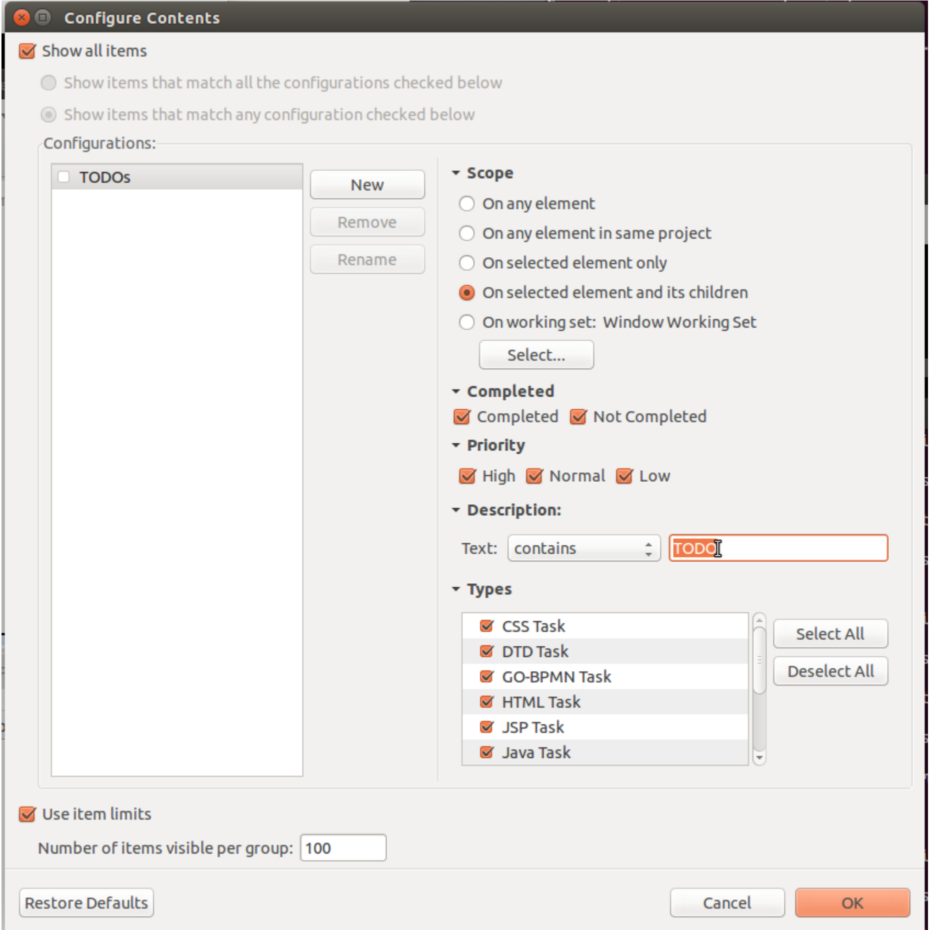


Figure 2.46 Setting contents properties for todo tasks

2.2.8 Validation

Validation in PDS checks the language and modeling correctness of the saved workspace data.

When validation detects a problem, it assigns it severity and renders the problem marker on all visualizations of the element and all its parent modeling elements:

- Errors represent severe mistakes that prevent the module being deployed.
- Warnings are issues that may affect execution and cause undesired behavior but are generally harmless; for example, a missing value of a parameter, which is interpreted automatically as null.
- Infos are notifications of non-standard situations, which influence neither the validity nor further deployment. Infos occur, for example, if a modeling element is present in the process definition, but does not appear in any Diagram.

The severity of a detected problem is generally assigned automatically but for in some cases you can define the severity yourself.

Your workspace is by default validated *automatically* on each save and before module upload. To turn automatic validation off on a project, on the main menu, go to **Project** and unselect **Build Automatically**.

You can trigger validation also *manually*: right-click the node in the GO-BPMN Explorer and click *Validate* in the context menu.If your resource has validation problems even though the expressions are correct, try cleaning the project: on the main menu, go to **Project** -> **Clean**

To analyze the detected problems, use the Problems view: The Problems view shows a list of validation problems detected in the opened GO-BPMN projects (closed project are not validated).

Non-Executable Processes and contents of Non-Main Pools are not validated.

> **Note:** The type of individual problems displayed in the view may differ, though mostly it has the "↩ GO-BPMN Problem" value; for information about other problem types refer to www.help.eclipse.org or documentation of the respective plug-in.

2.2.8.1 Configuring Validation

You can configure the severity of some of the problem that validation checks:

1. Click **Window** > **Preferences**.
2. In the Preferences dialog box in the left pane, expand Process Design Suite.
3. Expand Modeling and click Validation.

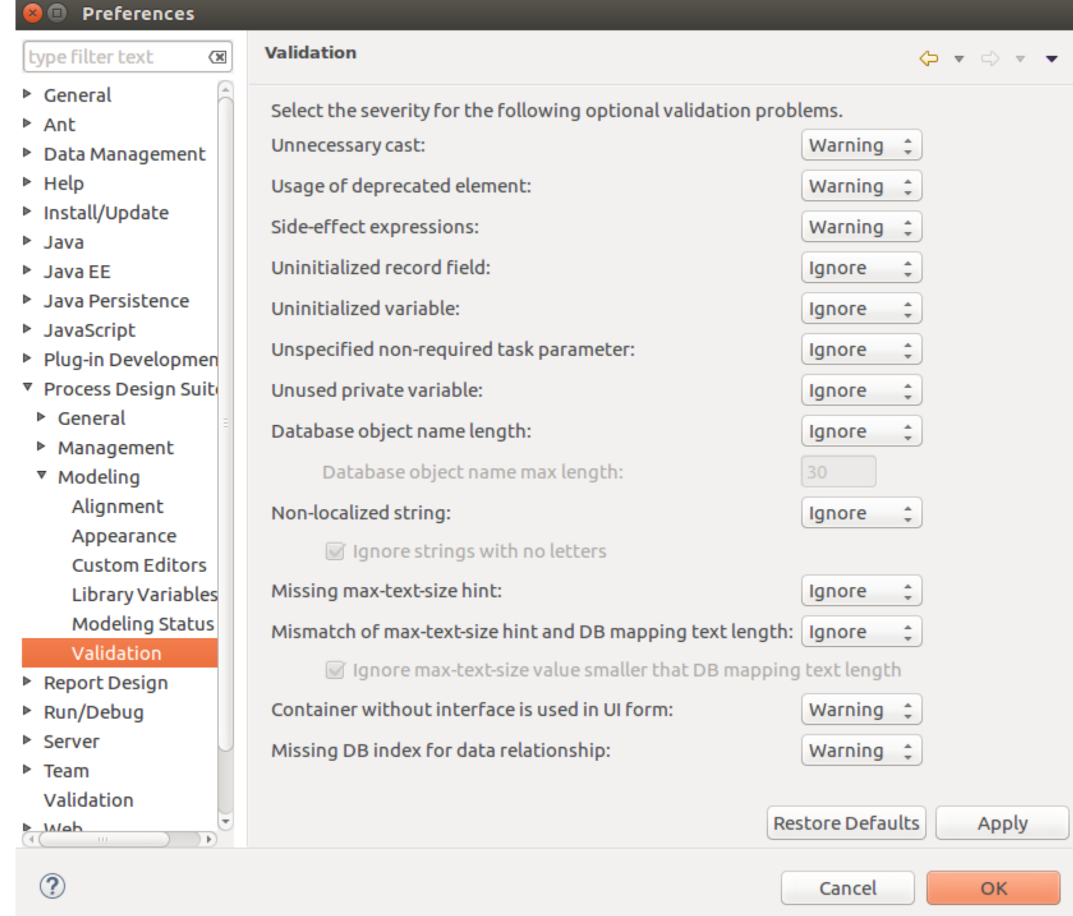


Figure 2.47 Validation page with predefined settings

Most of the settings are self-explanatory. Let us describe some of the more cryptic ones:

- An expression with a side effect does one of the following:
  – modifies a variable outside of its scope;
  – creates a shared record;
  – modifies a record field;
  – calls an expression that causes a side effect.
- Usage of deprecated element: the expression uses an element with the flag *Deprecated*

4. Click the respective severity level for the described problem.
5. Provide the maximum length allowed for a database object (if exceeded, by some data types, the validation detects a problem).
6. Click Apply or OK.

2.2.8.2 Validating Old Modules

Imported GO-BPMN Modules created in older versions of the Process Design Suite may not be validated automatically. To enable validation of older modules, do the following:

1. Click **Project** -> **Configure GO-BPMN Validator**.
2. Click OK.

Any old GO-BPMN modules in the workspace are added to the validation scope.

2.2.8.3 Hiding Validation Markers

To hide problem markers in a diagram editor (markers remain visible in the GO-BPMN Explorer and other views), do the following:

1. Click Show Diagram Items ( ) button on the main toolbar.
   The button is available only when a diagram editor is focused.
2. Unselect Show Validation Errors/Warnings.

2.2.9 Search

You can search in your workspace using any of the available Eclipse searches; however, to search the GO-BPMN Project content, we recommend the following:

- To search for elements or strings, use the GO-BPMN Search;
- To search for usages of a definition, use the usage search of elements.
- To search for Tasks that influence a Goal condition, search for dependent Tasks of the Goal.
- To search for calls and acquire entire call hierarchies, that is, the entire chains of the calls, use the call hierarchy search.

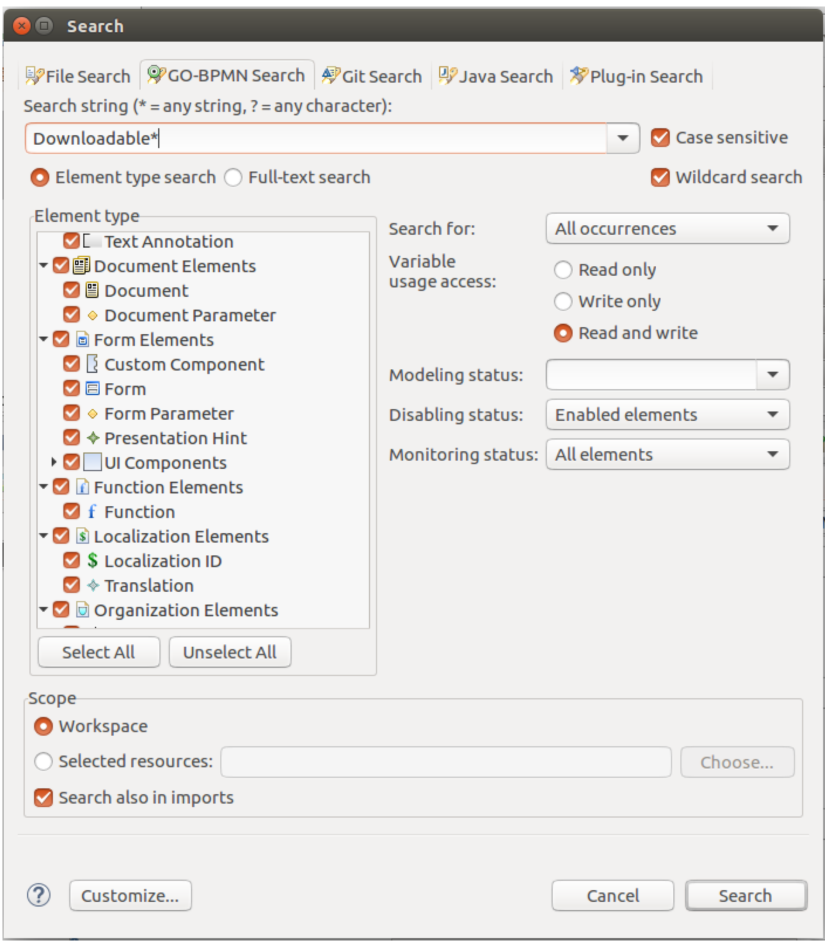
2.2.9.1 Searching for GO-BPMN Entities and their Usage

The GO-BPMN search is the generic LSPS search and allows you to search for entities based on their name, type, or status.

To define and perform such a search, do the following:

1. Go to **Search**> **GO-BPMN**.
2. In the search dialog, define the search criteria
   - Search string: string to search for in elements
     - Case sensitive: upper and lower cases are distinguished.
     - Wildcard search: wildcards are allowed (∗, ?).
   - Type of search:
     - Element type search: returns elements meeting the defined criteria.
     - Full-text search: returns any entities containing the defined string.
   - Scope:
     - Workspace: search is performed within the active workspace.
     - Selected resources: search is performed within the defined resources (projects, modules, libraries).
   - Element Type: select the element types to include in the search.

The results will be displayed in the Search view.



2.2.9.2 Searching for Elements

The Open Element feature is a search that locates any elements containing the provided string.

To search for an element:

1. Go to Navigate > Open Element
2. In the Open Element dialog, type the element name or its part. img{openElement.png, The Open Element dialog with search results}
3. In the area with search results below, double-click the element to open its definition.

2.2.9.3 Searching for Element Usages

To search for usages of GO-BPMN elements, such as, functions, variables, records, record fields, process elements, etc. right-click the definition or declaration and in the context menu, go to **Search For** and select **Usages** or the relevant type of occurrence: on variables, you can search for read or write accesses separately.

2.2.9.4 Searching for Dependent Tasks

Dependent Tasks searches for Tasks, which influence goal conditions (pre-condition or deactivate condition of achieve goals; maintain condition of maintain goals).

To search for such tasks, do the following:

1. In the GO-BPMN Explorer or in a goal diagram, locate the goal.
2. Right-click the goal, and click **Search For** > **Dependent Tasks**.
3. In the *Search for Dependent Tasks* dialog box, specify the search scope, and click **Search**.

Search results are displayed in the Search view.

2.2.9.5 Searching for Call Hierarchies

To search and open the call hierarchies over a definition, right-click the definition in the editor or the Outline view and click **Open Call Hierarchy**.

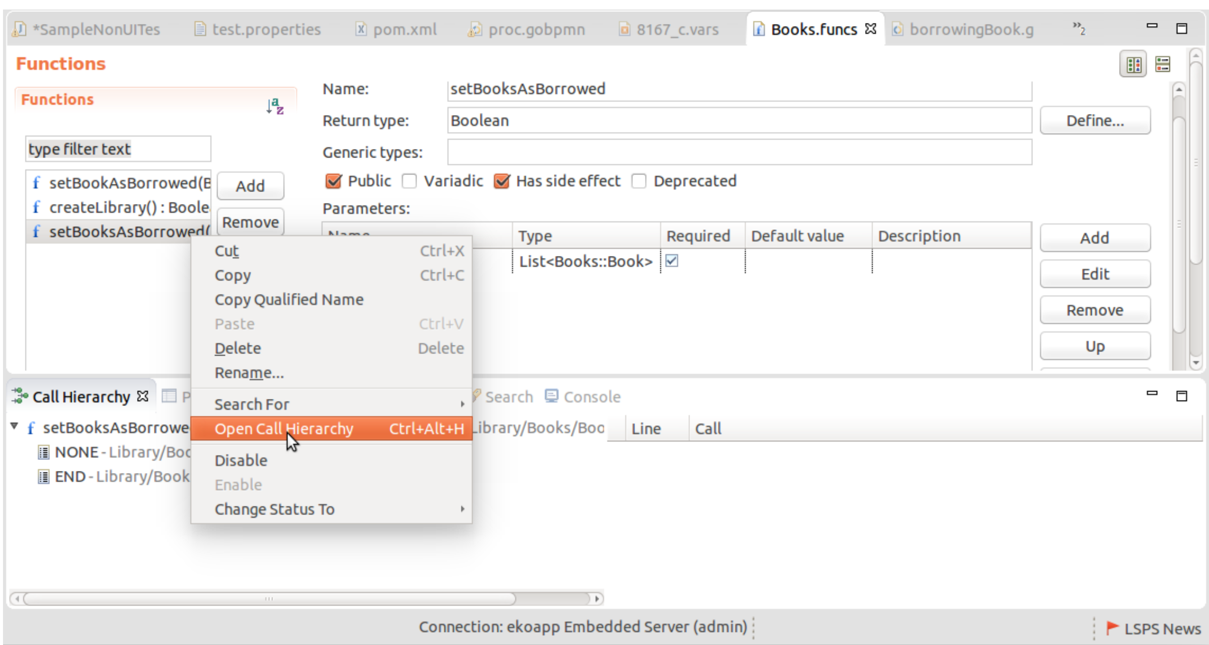


Figure 2.48 Opening call hierarchy of a function definition

2.3 Processes

A Process is defined in a process definition file, which can hold one BPMN or GO-BPMN Process. When the process is executed, it is executed directly based on the definition: no compilation or code translation takes place.

- A process represents a namespace, and as such, can define process variables. It can also define parameters.

For further information on Processes and their elements, refer to GO-BPMN Modeling Language Guide.

2.3.1 Creating a Process

To create a Process, do the following:

1. In GO-BPMN Explorer, right-click the Module and go to **New** > **Process Definition**.
2. In the dialog box, define the properties of the Process:
   - **Container**: parent Module
   - **Type**: whether the Process is goal based or a standard BPMN process without the GO-BPMN extension
   - **Visibility**: if **Private**, the Process content will not be accessible from importing modules. Also, you will not be able to use such a process in a reusable process.
   - **Executable**: when selected the process is instantiated as part of the model instance and can be used as a Sub-Process or as the Activity parameter of the Execute task.
     The flag marks a process that serves only for documentation purposes.
   - **Instantiate Automatically**: if selected, the Process is instantiated when its parent Model is instantiated and that even if it is imported in a non-executable Module.

3. Now you can define the workflow of the Process.

   Note: Modeler is a diagram editor: you can editing the content of the Process definition via a diagram, hence consider getting familiar with the concept of diagrams and diagram editors.

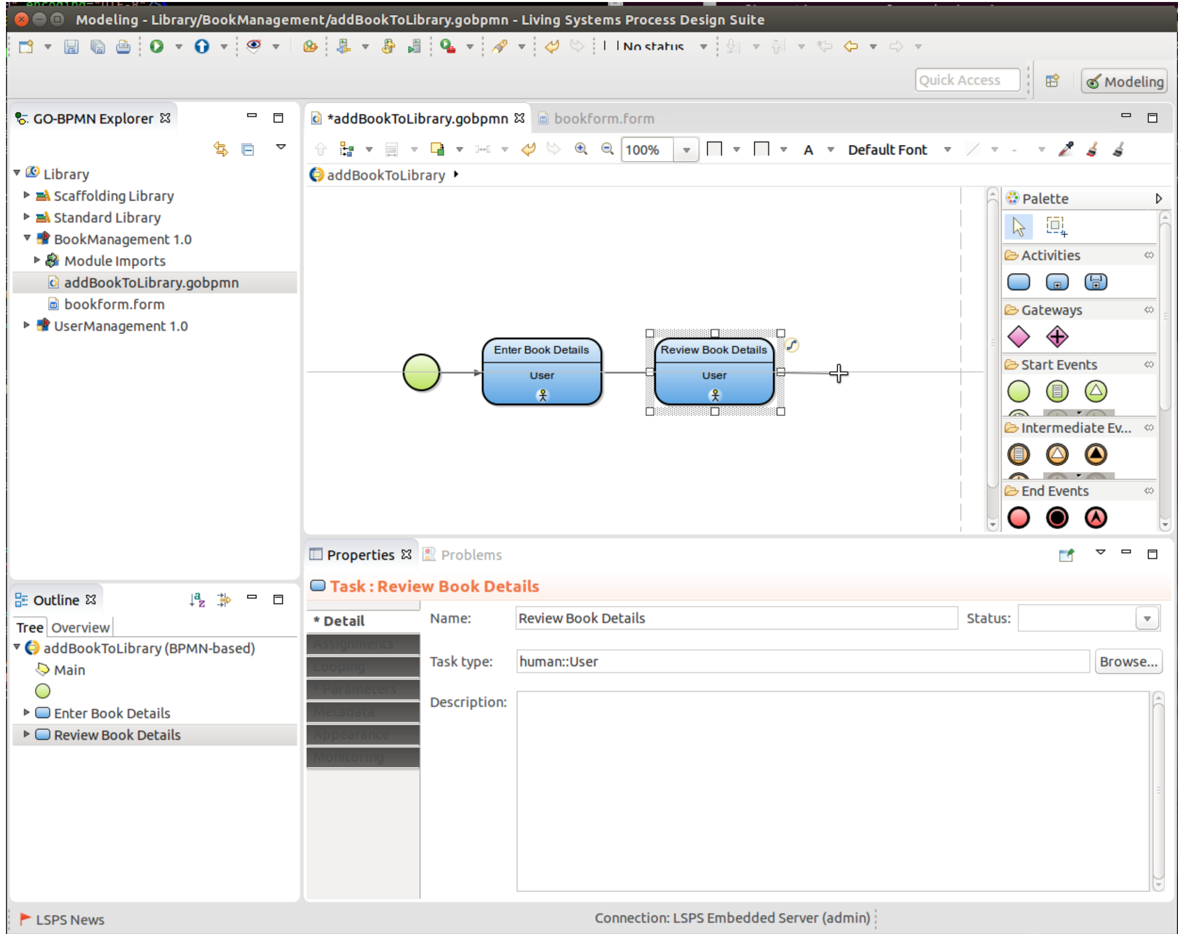


Figure 2.49 Creating Process workflow

2.3.2 Defining Process Parameters

You can instantiate your process with parameters if it is instantiated with a function call or by a reusable sub-Process, etc. If a process is instantiated automatically by its model it cannot take any input parameter.

To define process parameters, do the following:

1. Make sure the process is open in the process editor.
2. In the Outline view, right-click the process and select **New** > **Parameter**. Make sure to select the parameter as *Required* if it cannot be **null**: if not required and the process is instantiated without the parameter, the parameter value is **null**.

   Alternatively, you can open the process properties in the Properties view and click **Add** on the Parameters tab.
3. In the Property view, define the parameter properties. In the case of reusable-Subprocesses, the parameter values are defined on the Parameters tab of its Properties view.

Example process instantiation

```
applicationProcess(user -> admin, requestedHardware -> Hardware.ssd)
```

2.3.3 Defining Process Variables

Process variables are accessible from within the process context and are initialized when the process instance is created.

To define local process variables do the following:

1. Make sure, you have the process resource opened in the Process editor.
2. In the Outline view, right-click the process and go to **New** > **Variable**.
3. In the displayed Properties view, define the variable properties.

2.3.4 Modeling a Process

After you have created a process, you can model its content: the way you design your process and the element you use are primarily determined by the process type you have chosen, that is, whether your process is BPMN- or GOBPMN-based.

Process content is defined using the Modeler, which is a dedicated diagram editor: when you open a process for editing, the editor opens a `diagram`. As you insert new element views onto the diagram, the elements are created in the process and displayed in the Outlook view.

2.3.4.1 Changing Task Type

To change the task type of an existing Task, right-click the Task in the canvas and click **Change Type**. In the dialog box with task types select the new Task Type.

Note that any parameter value you have already defined are lost unless the new Task Type defines the same parameter.

2.3.4.2 Removing Invalid Task Parameters

To remove the invalid parameters and keep only the parameters that are relevant for the new Task Types, go to **Project** -> **Task Parameter Cleanup**: Task type parameter cleanup automatically removes irrelevant parameters and parameter values.

2.3.4.3 Extracting Process Elements into a Reusable Sub-Process

To nest one or multiple Process elements into a Sub-process, select the elements, right-click the selection, and in the context menu, click **Extract to Subprocess**.

2.3.4.4 Linking Goal to Goal Diagram

To continue designing a goal sub-tree in a new diagram (that is, to create an implicit hyperlink on a goal):

1. In a goal diagram, right-click the goal you wish to decompose further on a new diagram.
2. Click **Create Goal Diagram**.

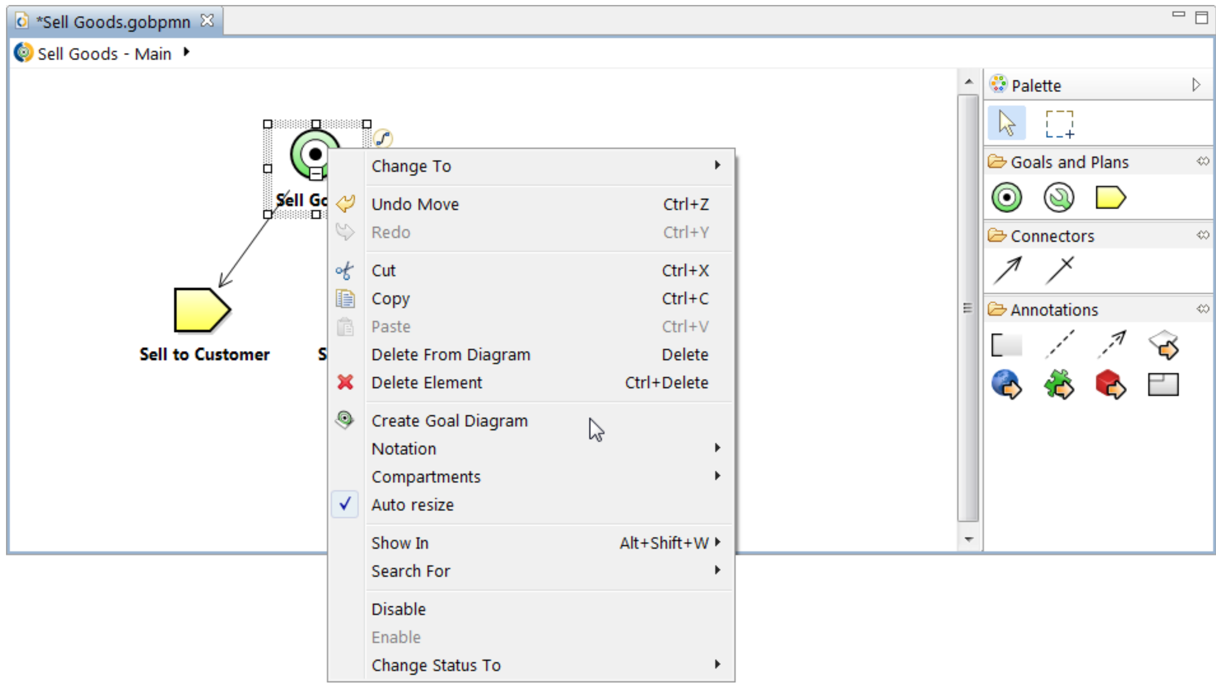


Figure 2.50 Creating a new goal diagram with an implicit goal hyperlink

Canvas with a new diagram with a hyperlink view of the selected goal opens. Both views of the goal (one in the original diagram, one in the new diagram) are provided hyperlink markers. Double-clicking a goal view with an implicit hyperlink opens the diagram with the connected goal view.

2.4 Variables

Variables are defined for a particular element, such as, module, process, sub-process, expression, form, etc. Depending on this element, we distinguish the following:

- global variables defined in a module
- variable defined in other elements that represent a context, such as, processes, forms, sub-processes
- local variables defined for an expression or an expression block

2.4.1 Global Variable

Global variables, or module variables are available throughout the entire model, that is, anywhere from its parent module and from any module that imports it, unless the access is restricted by their visibility.

They are defined in a variable definition file, a module resource file. One module can contain multiple variable definition files; however, the variable names even if in different files must not be identical.

To define a new global variable, do the following:

1. In the GO-BPMN Explorer, double-click the respective variable definition.
2. In the activated form-like Variable Editor, click Add.
3. In the Name text box, type the variable name.
4. In the Type text box, type the data type of the value that the variable is going to store.
5. Define the variable visibility:
   - Select the Public checkbox to make the variable available to the importing modules.
   - Clear the Public checkbox to make the variable private (available only within the Module).
6. In the Initial Value text box specify the initial value of the variable: Type the initial value as an expression in the Expression Language directly into the Initial Value field. If no initial value is specified, the value is set to `null`.

   Click **Edit** to open the Value Dialog dialog box.
7. To use the variable in an expression, enter the variable name; for example, to assign a value to a variable: `myGlobalVariable := "string variable value"`

Important: Global variables are instantiated in the order they are defined in. Therefore, if you want a variable to use another variable from the same file in its initial value, make sure the variable is defined below the initialization variable.

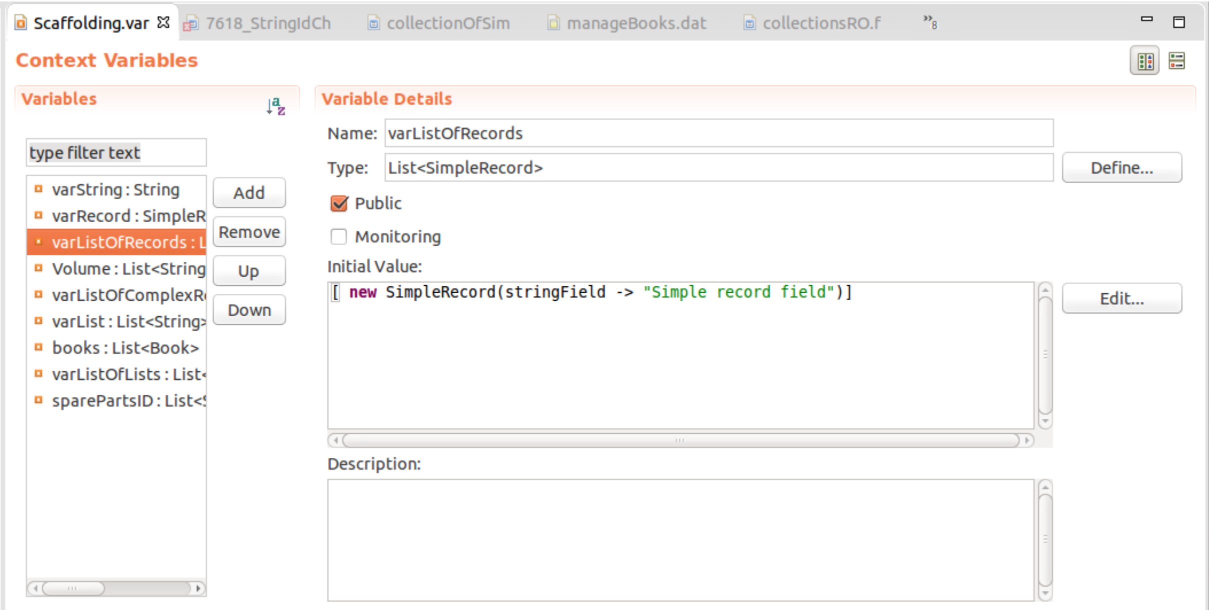


Figure 2.51 Editing variable definitions

2.4.2 Process, Forms, and Sub-Process Variable

These variables are defined on Forms, Processes and some of their elements, that create their own context, that is Processes, Plans, Sub-Processes, forms, and model update Processes.

To define such local variables, right-click the element in the Outline view, and go to **New** -> **Variable**.

2.4.3 Local Variable

In expressions, you can define local variables as `def <TYPE> <VARNAME>`. Since each property definition, such as, goal condition, event duration, initial value, etc. represent an expression block, it can define its `local variables`.

2.5 Organization Models

The organization model defines roles, units, and their relationships and serves to group users who interact with the process.

Individual roles and units can be assigned to users so as to define their role in the business process: When a Application User Interface user, a person, has a particular Role or belongs to an Organization Unit, the system includes them when the persons with the role or in the unit are requested.

To make the system return all persons with a particular role, you can use the `<ROLE_OR_UNIT>(<PARAMETERS_MAP>)` call. For example, to set the **performers** parameter of a User Task, you could use `SoftwareEngineer(["foe" -> "PM1", "lang" -> "Java"])`. This means that the To-Do of the user task will appear in the to-do list of all persons with the role *SoftwareEngineer* and the specified parameters.

For further information on organization elements, their properties, and impact on work distribution, refer to `GO-BPMN Modeling Language Guide`.

2.5.1 Creating an Organization Model

> **Important:** When creating an organization model for your Model, we recommend that you consider creating a model that reflects the organization of people who participate in the execution of the model, not the real organization structure of the enterprise.

Organization models are create with the organization, which is a dedicated diagram editor. Make sure to make yourself acquainted with the concept of `diagrams` before designing your organization model.

To design an organization structure, do the following:

1. In the GO-BPMN Explorer, right-click the Module and go to **New** > **Organization Definition** or double-click your organization definition or organization diagram.

   The Organization Editor with a diagram of the definition appears in the editor area.

2. Design the organization model: click an element in the palette and click on the canvas to place the element on the diagram and create it in the definition file. You can then create further related elements with the quick-linker.

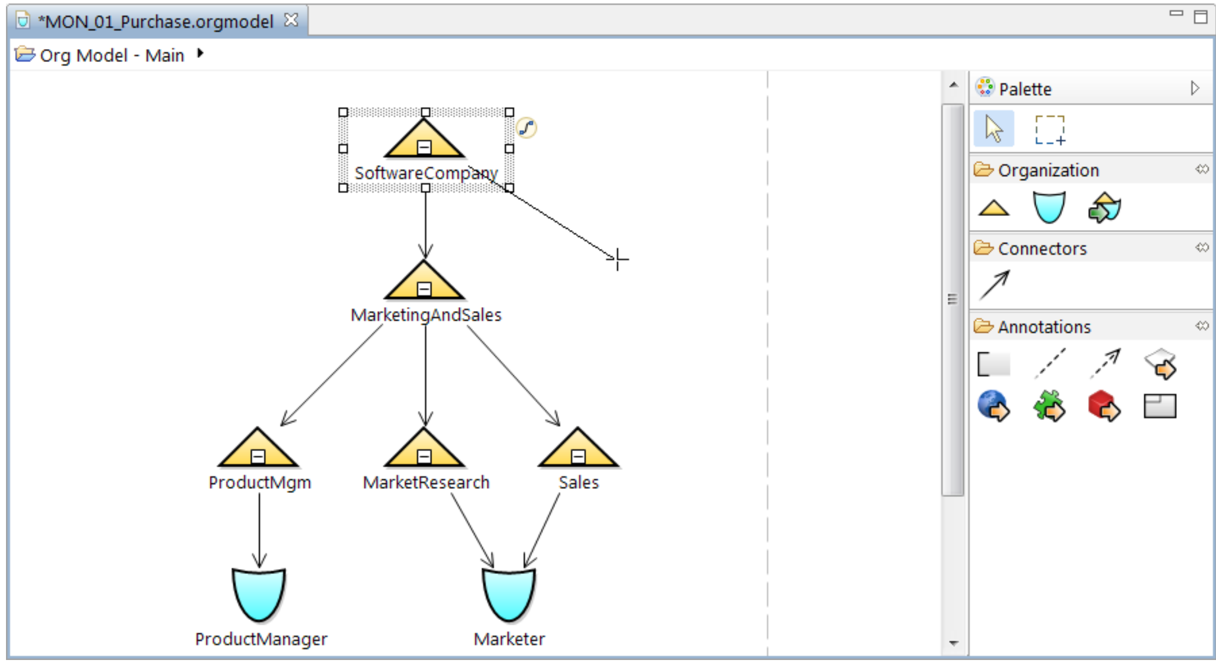
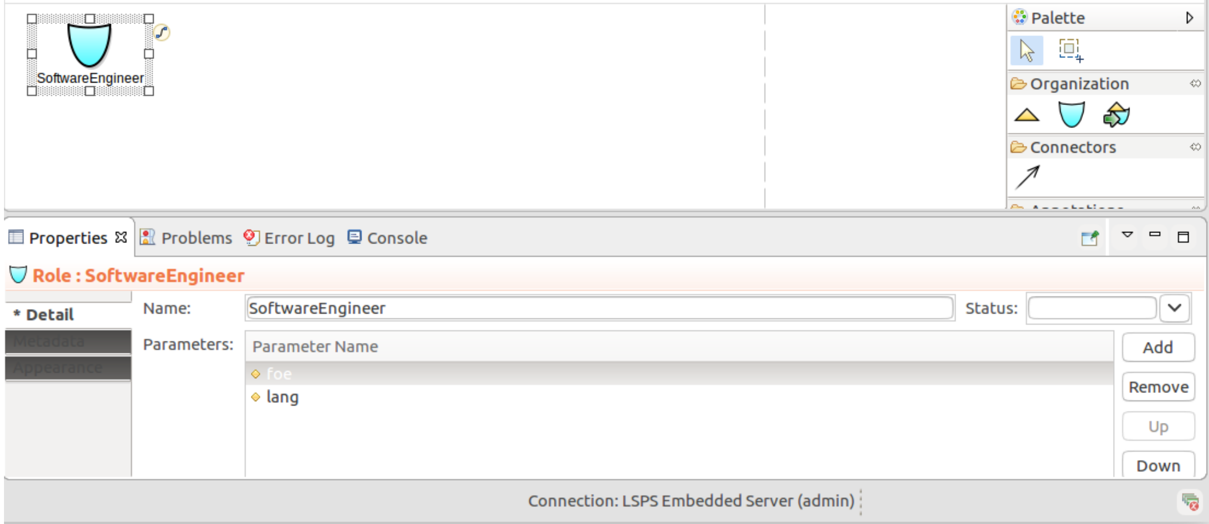


Figure 2.52 Editing the Main diagram of an organization definition file

3. For units and roles, optionally define parameters in their properties.



4. When finished, click **File** -> **Save** to save the file or press **Ctrl + s**.

2.5.2 Assigning a Role or Organization Unit to a Person

You can add a person to a role or organization unit in the following ways:

- with the `addPersonToRole()` function
- on runtime, from the `management perspective` or `web management console`.

2.6 Documents

Documents serve to interact with the user via pages with business data that are not dependent on a process: documents are available as long as their definition is on the server.

Documents are defined in the document definition: you can create one or multiple Documents in one document definition. Typically, you want to create your documents in a dedicated Module to allow you to unload the Module without side effects later if necessary. Once you have created a Module with document definitions, all you need to do is upload the Module: The documents will be available in the Document tab of the users as long as the Module remains on the server.

The default application displays the list of Documents available on the server on the Documents tab:

- When the user opens a document, the system creates a Model instance with all the context data and renders and populates the document data.
- When the user submits the document, the document data is saved, typically in the underlying DB via shared Records, and the model instance ceases to exist. Note that the document will remain available in the list of documents since the list contains the types of documents, not their instances: Unlike To-Dos, documents availability does not depend on a Process Task; they are available as long as the definition of the document is on the server and their instances are created on request.

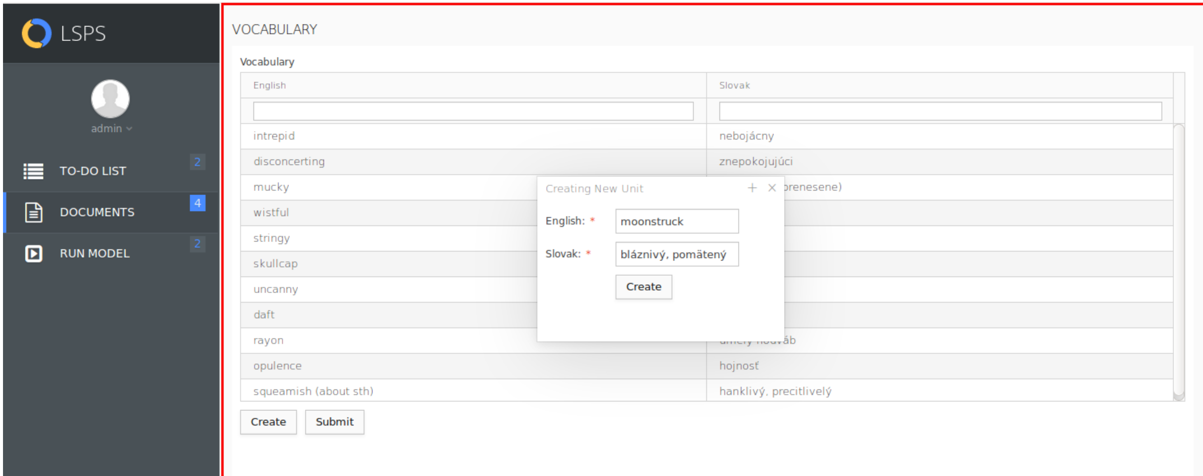


Figure 2.53 Form with a table in a document

2.6.1 Defining a Document

To define a new document, do the following:

1. Create a document definition file:
   (a) Right-click your module.
   (b) In the context menu, go to New > Document Definition
   (c) In the New Document Definition dialog, define the definition file properties: check its location and modify its name.
2. Open a document definition file.

3. In the Documents area of the Document Editor, click **Add**.
4. In the right part, define the properties of the document.
   - **Name**: name of the document unique in the Module
   - **Title**: document title displayed in the web application
     The title is a String expression, which is re-evaluated on document refresh (for details on the refresh mechanism refer to Application User Interface Forms User Guide).
   - **Parameters**: list of input arguments of simple data types used by the document
     Parameters are meant to pass data to the document. Note that documents with parameters are not visible in the Documents list in the application.
   - **UI definition**: expression that returns the UI definition, typically a form call
   - **Access rights**: boolean expression that defines the condition that must be true to allow the front-end user to access the document
   - Navigation: where to go when the user submits the document
   - **Navigation Factory**: function that will return the Navigation object to the Document and takes the same parameters as the document
5. If applicable, upload your Module and check the Document on the Documents tab of the Application User Interface.

2.6.2 Navigate Away from Document

To define where to navigate when the user submits a document, define the **Navigation** property closure: The closure gets as its input parameter all To-Dos that were created when the document form called the `createModelInstance()` or a synchronous `sendSignal()` functions.

For example, let's assume a document that references a form with a Button component. The Button component defines an ActionListener with the handle expressions

- `createModelInstance(true, findModels("MyModel", 1.0, true) [0], [->])` and
- `sendSignal(true, findModelInstances("MyModel", 1.0, true, [->]), "Consider roll-back action.")`

The To-dos created by the calls are input of the Navigation closure so you can navigate to one of the to-dos.

You can navigate to to-dos, application pages, external URLs, documents. Here are some examples:

- to-do

  ```
  {todos:Set<Todo> ->
    new TodoNavigation(
      todo -> getTodosFor(getPerson("admin"))[0],
      openAsReadOnly -> false)
  }
  ```

  To navigate to a to-do created from the document, use the input parameter of the navigation closure that holds to-dos created by a `createModelInstance()` or a synchronous `sendSignal()` functions: to navigate to the first to-do created by the document form, define Navigation as

  ```
  {todos:Set<Todo> ->
    new TodoNavigation(
      todo -> todos[0],
      openAsReadOnly -> false)
  }
  ```

- application page:

```
//navigating to the Documents page:
{ s:Set<Todo> -> new AppNavigation(code -> "documents")}
```

- document:

```
//navigating to a document (creates a new document instance):
{ s:Set<Todo> -> new DocumentNavigation(documentType -> myDoc())}
```

- saved document:

```
//navigating to a saved document;
//getMySavedDoc() is a query that returns a SavedDocument):
{ s:Set<Todo> ->
  new SavedDocumentNavigation(savedDocument -> getMySavedDoc())
}
```

- url:

~~~~~~ //navigating to an external URL: { s:Set<Todo> -> new UrlNavigation(url->"www.whitestein.com")} ~~~~~

You can also define the **Navigation Factory** function so you can use it then Navigation property so you do not have to explicitly instantiate the Navigation object with parameters.

Example Navigation Factory

```
getDateFormNavigation(1)
//instead of
//new DocumentNavigation(
//  documentType -> dateForm(),
//  parameters -> ["id" -> 1]
//  )
```

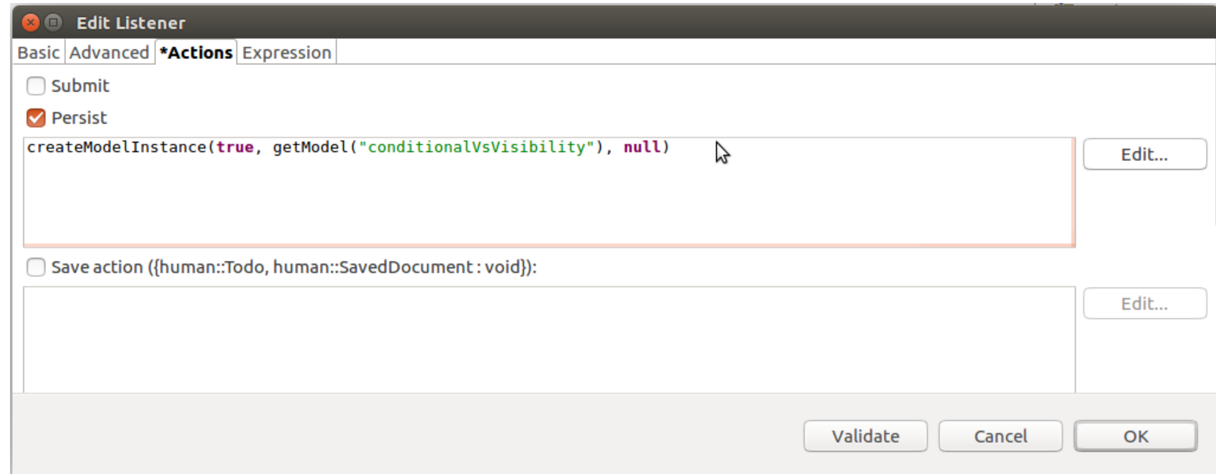
2.6.2.1 Creating a Model Instance and Navigating to its To-Do on Document Submit

To create a model instance from a document and then navigate to one of its To-dos, do the following:

1. Define a listener that will create the model instance:

   (a) Attach a listener of the required type to a component.

   (b) Create the model instance in its persist action, for example, `createModelInstance(true, getModel("myModelName"), null)`


~~~~~~

2. Define a listener with the Submit action (it represents the moment when you want to submit the data and navigate away from the document).

3. In the document definition, define the Navigate property so it returns TodoNavigation to the required To-do.

```
\\navigates to the first to-do generated by the document:
{todos:Set<Todo> -> new TodoNavigation(todo -> todos[0], openAsReadOnly -> false)}
```

2.7 Forms

Important: The ui module with forms implementation makes use of the Vaadin and Vaadin Charts frameworks. Make sure to purchase the Vaadin Charts license if you are designing charts. The users of your Application User Interface and Management Console, do not require any additional licenses.

A *form* defines the content displayed in the front-end application. It can be displayed either by a document or by a to-do:

- When displayed by a document, the form is used as content of a page that is permanently available to the user.

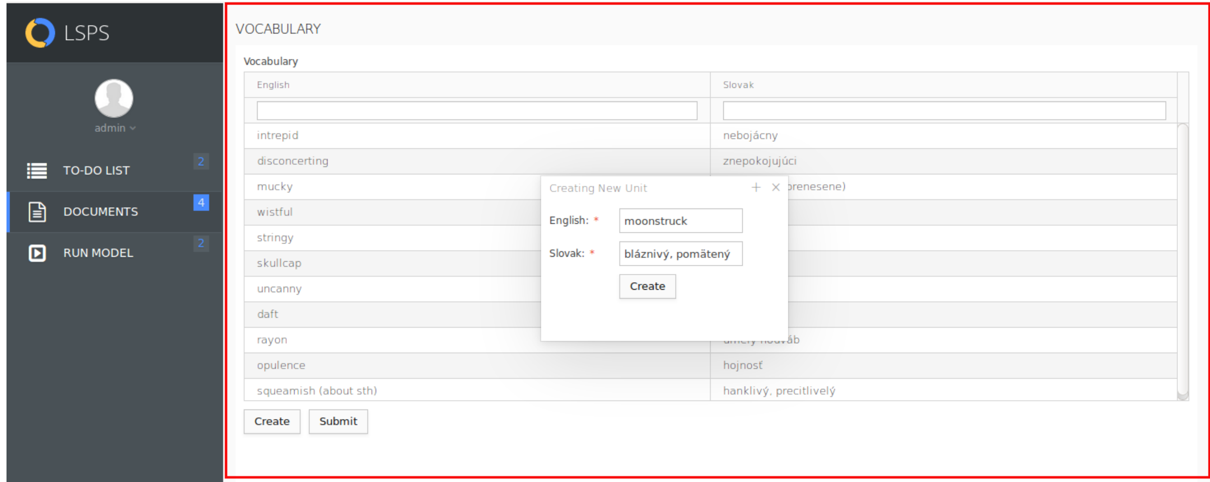


Figure 2.54 Form with a table in a document

- When displayed by a `User Task` in a `to-do`, the form is used as the content of the to-do which is part of a process instance and disappears once the user submits it.

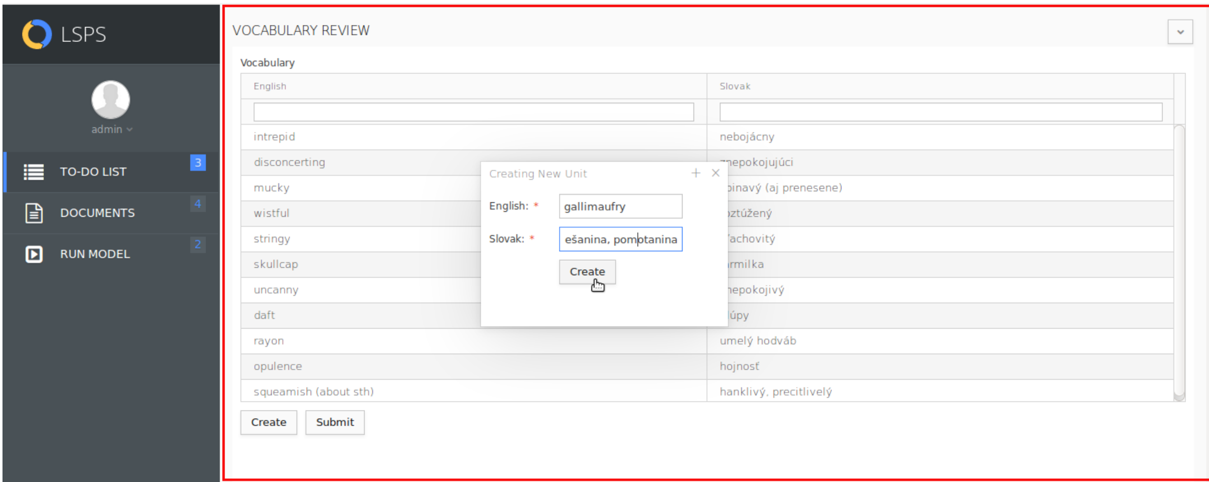


Figure 2.55 Rendered to-do with the form

The form comprises a tree of form components, such as, input fields, radio lists, etc. Each component produces events whenever something happens. These events can be caught by listeners. When a listener catches its event, the event is queued and processed in the event-processing cycle. The way the event is processed is defined by the listener.

For example, if the user clicks a button, the button fires an event, called ActionEvent. The event is caught by the ActionListener attached to the button component. The listener defines what the event causes, so the event can refresh form components, persisting the data, submitting the to-do or document, etc.

When the user opens a to-do or document, the following happens:

1. The system creates an context level, called *screen context*.
2. The system creates copies of the data from the model context in the *screen context*.
3. The system initializes form local variables and builds the form.
4. The form tree is rendered as web content and populated with the data from the *screen context*.

This allows you to change data in the form without instantly influencing the data in the "real" **base context**.

A form can take input parameters and define its local variables.

You can create also custom components and add them to the form-editor palette. For further instructions, refer to the `Software Development Kit Guide`.

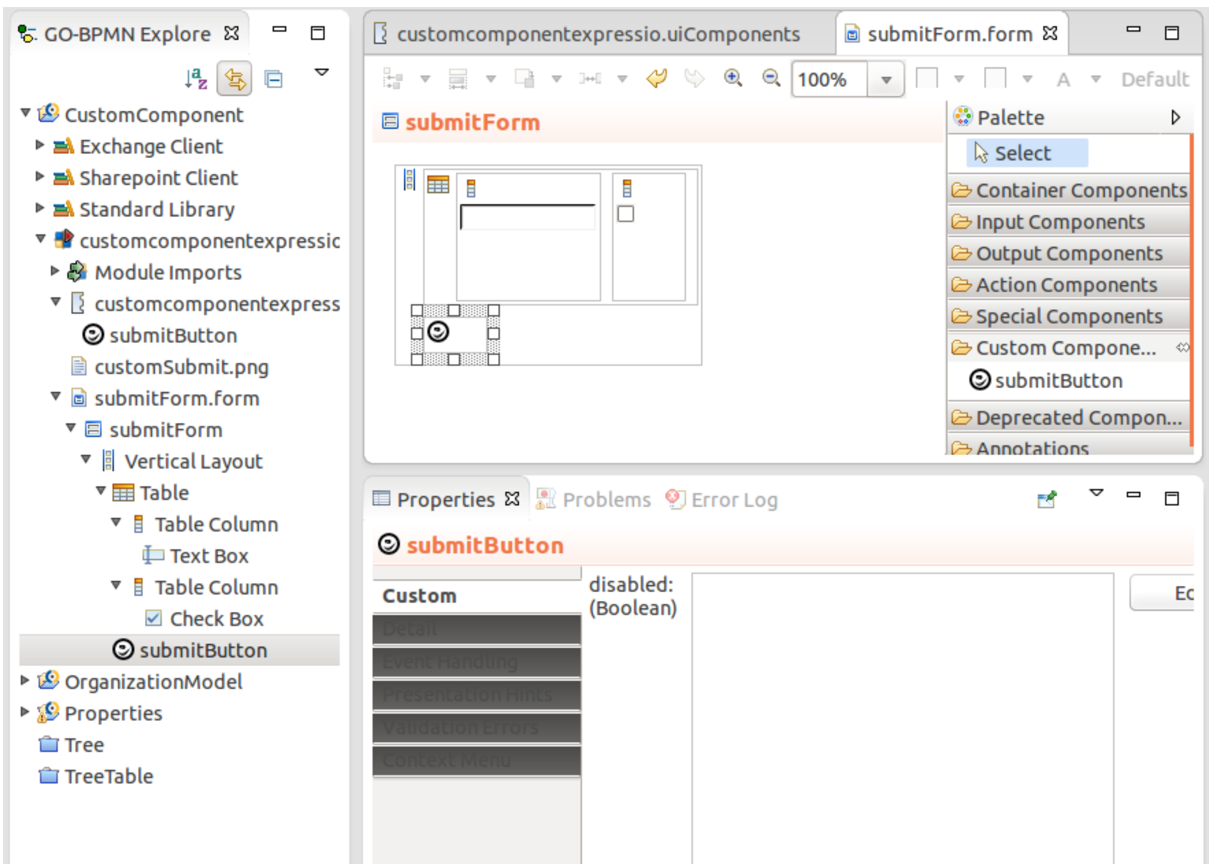


Figure 2.56 Form Editor with a Custom Submit Button

2.7.1 Form Execution Levels

A model instance state is defined by its properties and data in its `contexts`. When the model instance creates a form, the context hierarchy is created on another execution level to isolate the form data. As a result, the changes of such form data are applied only on the "real" data only when explicitly required: **to-dos or documents with a ui form are created on their own level that overlays the model instance level**.

So while model instances exist on the **base context level**, when a model instance hits a user task or when a document of model instance is created, the system creates a new level for the form of the document or task. This context level is a copy of the base context level and is referred to as the **screen context level**: When the user changes the data in the form, the changes are performed on the context of the screen level.

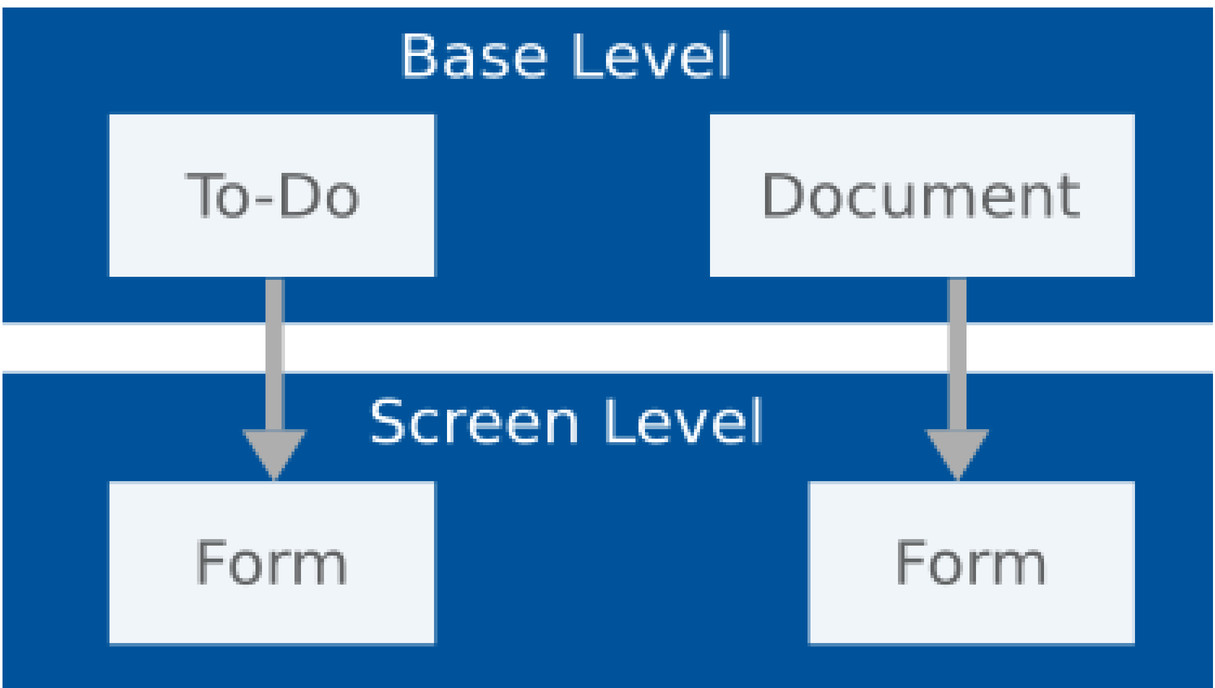


Figure 2.57 Schema with execution levels of a model instance, and a to-do and form

Note that sublevels always **hold only the differences to the context hierarchy in the superlevel**.

To apply the changes, the user must merge the screen level into the *base level*: this happens when the form is persisted. Persisting is performed as part of event processing: hence, to merge the screen level into the base level, you need to create a listener with the **persist** or **submit** action.

If you need to create another execution level over the screen level to isolate data within your form, you can do so with a View Model component: the component creates another execution level referred to as an **evaluation level**.

Continue to:

- Creating Forms
- Form Components
- Form Patterns
- Enabling Error Reporting on Components

2.7.2 Event Processing

When the user or the system performs an action on a form component, the form component produces an event of a particular type. If the form component has a listener for the type of the event, the event is usually immediately processed in the event-processing cycle, the exception being the non-immediate ValueChangeEvents).

The **event-processing cycle**, or request-response cycle, processes the event in a set of steps that gradually perform actions defined on the listeners of the events.

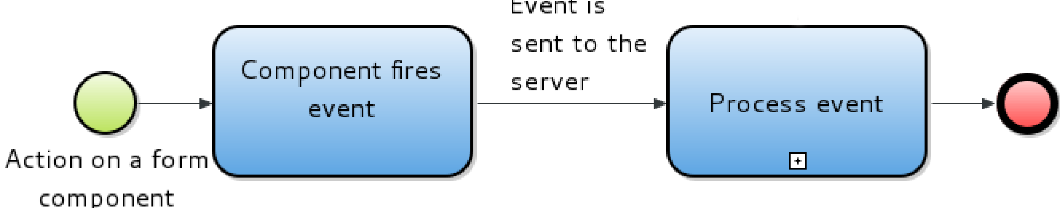


Figure 2.58 Form event firing

You can find a detailed schema of the cycle here.

The event-processing cycle performs the following:

- Updates of internal values (References to internal values are defined as bindings on form component.)
- Validates values
- Executes listener logic as defined by their handle expression
- Fires application events
- Merges or clears any View Models data
- Evaluates the Navigation expression

If a listener requests the *submit* action, the form is submitted and the event processing cycle finishes. If submit was not requested and a processed listener fired an application event, the event-processing cycle is repeated and the newly fired application events are processed. Otherwise, the components are refreshed and any produced ApplicationEvents are processed. If no such events exist, the event processing cycle finishes.

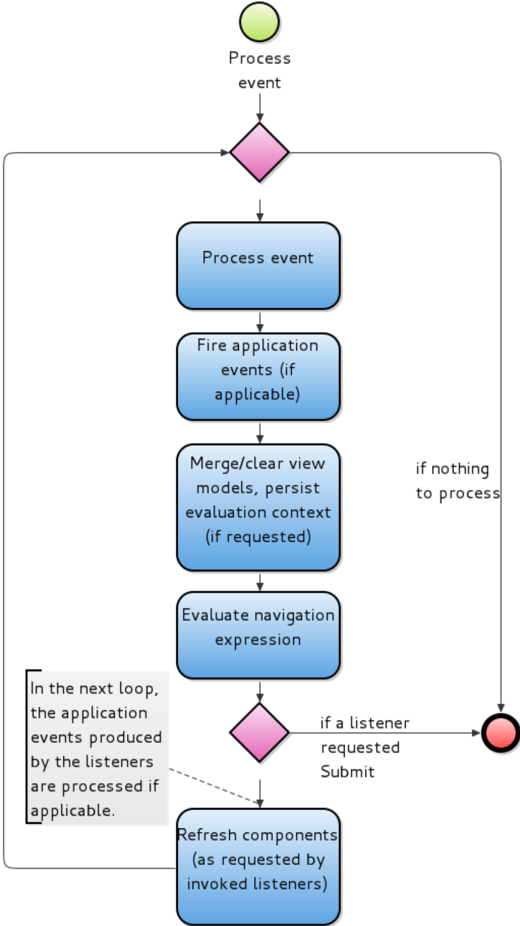


Figure 2.59 Event processing cycle

When the event processing finishes, the cycle executes the listener actions if requested, that is, the save, submit, and navigate actions:

- **Submit**: defines if the document or to-do is submitted as part of the event handling;
- **Persist**: if enabled, the relevant data in the processed components is persisted as part of the event handling; unlike on submit action, if the form is used by a to-do, the to-do does not become finished; if part of a document, the document remains open; persist is performed after the merge to the screen level before the transaction is committed.
- **Save action**: defines if the to-do or document is saved;
- **Navigation**: defines the location where to navigate after the event is processed; you can navigate to a to-do or document, URL, custom application page (refer to the descriptions of the data types defined in the human.navigation.datatypes resource in the Standard Library)

The event processing and the listener actions are executed within a single database transaction so that if the processing fails at any stage, the database is rolled back to a consistent state.

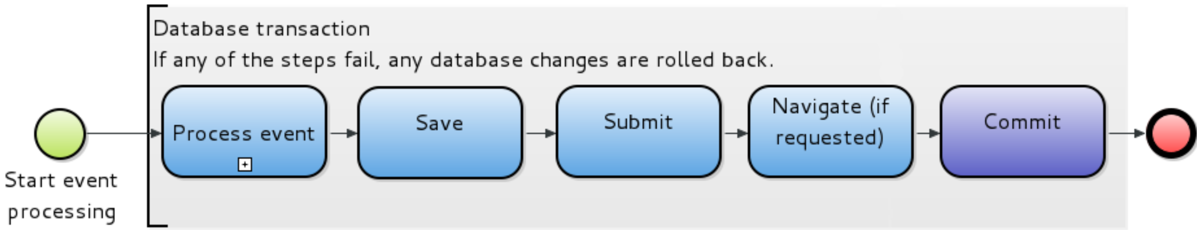


Figure 2.60 Database transaction on event processing and listener actions

Events Listeners

2.7.2.1 Events

When the user performs an action on a form component, such as, changing a value, clicking a button, moving a widget, etc. the component produces an event of a type: the type depends on the action. The produced event holds data about the actions.

If the component has a listener that is listening for an event of the given type, it catches the event. These caught events are add to the event queue. Most events, called immediate events, trigger the request-response cycle and hence are processed immediately along with all the events in the event queue. If the event is not immediate, it waits in the queue until an intermediate event enters the queue and triggers the request-response cycle.

Some event types are immediate by default; other event types are non-immediate and can be set as immediate explicitly if required.

All queued events are processed simultaneously: the way the events are handled and the actions they cause are defined by the listeners that caught them; for example, a listener can request refreshing of form components, data validation, etc.

Events have their properties stored in their fields: for example, the ChartClickEvent contains fields with details on where exactly on the chart the click occurred so the listener can define different actions depending on the location where the user clicked.

Form components produce events of different types depending on the action that created the event. Here is a brief summary:

- All form components
  - **InitEvent** produced when a component is initialized (a form or a previously hidden component is shown)
  - **ApplicationEvent** produced by any event as part of the event processing
    Application events can be fired by an event of any type and is broadcast to all form components.
- All input components
  - **ValueChangeEvent** produced when the user changes a value in an input component

  The time when the change-value event is processed depends on the setting of immediate mode.
- Button and Action Links
  - **ActionEvent** produced when the user clicks an Action Component
- File Download

  - **FileDownloadEvent** produced when the user clicks a file-download button of a file-download component

- File Upload
  - **FileUploadEvent** produced on file-upload completion
- Text Box and Text Area
  - **AsynchronousValueChangeEvent** produced on every key stroke
- All Chart components
  - **ChartClickEvent** produced when the user clicks a chart
- Widget
  - **WidgetChangeEvent** produced when the user adds a widget to the dashboard, resizes, moves, or hides a widget
- Calendar
  - **CalendarCreate event** produced when the user selects a time period by clicking and dragging
  - **CalendarEdit event** produced when the user clicks a calendar entry
  - **CalendarReschedule event** produced when the user drags a calendar entry
- Geolocator
  - **Geolocation event** produced when the Geolocator component acquires user's location
- Map Display
  - **MapClickedEvent** produced when the user clicks on the map
  - **MarkerClickEvent** produced when the user clicks on a map marker
  - **MarkerDraggedEvent** produced when the user drags a map marker
- Table of the Paged Type
  - **TablePageSizeChangeEvent** produced when the user changes the size of the table page
- Tree
  - **TreeEvent** produced when the user expands or collapses a tree item, be it in a tree or a tree table component
- Popup
  - ∗∗PopupCloseRequestEvent∗ produced when the user clicks the close button in a popup
- Context menu
  - **MenuEvent** produced when the user clicks an item in the context menu

2.7.2.1.1 InitEvent

The InitEvent is fired when a component is displayed, be it for the first time or after it was saved or when it became visible.

Table with event properties

Property	Data Type	Description
source	UIComponent	Component that produced the event
isFirstLoad	Boolean	true for a component displayed for the first time (the property is true also when a hidden component is displayed for the first time)
isFirstLoadAfterSave	Boolean	true for a component displayed for the first time or for the first time after save

2.7.2.1.2 ValueChangeEvent

The ValueChangeEvent is fired by input components and that when the user changes a value in an input component and changes the focus or on when they select an option.

> **Note:** When ValueChangeEvent takes effect depends on whether its input component is immediate, which is defined by the Immediate property of the component.
>
> If the component has the Immediate property set to `true`, the event is processed as follows:
>
> - If on a Text Box or Text Area, the event is produced and handled when the component loses focus, that is, after you edit the value and click another component or press Enter.
> - If on Lists, Check and Combo Boxes, the event is produced and handled when an item is selected.
>
> If the property is set to `false`, then the event is produced at the same time but remains unhandled, that is, the request-response cycle is not triggered: Instead it is added to an event queue. The queued events are processed on the next run of the request-response cycle, that is, when a ValueChangeEvent with activated immediate mode or an event of another type is handled. This typically happens when the user confirms the value changes by clicking a button (a listener for the ActionEvent must exist).

When a ValueChangeEvent occurs, the new value undergoes conversion validation before the events are actually processed. On conversion validation, the server checks, if the new value has the correct form. If this validation fails, the component is marked as invalid and the ValueChangeEvent, any ApplicationEvents and ActionEvents are not processed any further.

Note that when you change a value in an input component, the value change is applied on the bound entity, such as, a variable, even if no listener for the value change event exists. To prevent this behavior, use nested contexts.

Table with event properties

Property	Data Type	Description
source	UIComponent	Component that produced the event
oldValue	Object	Object with the value before change
newValue	Object	Object with the value after change

2.7.2.1.3 AsynchronousValueChangeEvent

The AsynchronousValueChangeEvent is fired and processed immediately when the user changes a value in a Text Box or Text Area component.

The event is fired asynchronously: a new one can be fired even if the old one is still being processed.

Table with event properties

Property	Data Type	Description
source	UIComponent	Component that produced the event
text	String	String with the input

2.7.2.1.4 ActionEvent

The ActionEvent is fired when an action or image component is clicked, when a file upload is started, and when ENTER is pressed on the TextBox component.

Table with event properties

Property	Data Type	Description
source	UIComponent	Component that produced the event

2.7.2.1.5 FileDownloadEvent

The FileDownloadEvent is fired when the File Download component is clicked.

Table with event properties

Property	Data Type	Description
source	UIComponent	Component that produced the event

2.7.2.1.6 FileUploadEvent

The FileUploadEvent is fired when file uploading finishes. Note that on upload start, an ActionEvent is fired.

Table with event properties

Property	Data Type	Description
source	UIComponent	Component that produced the event
uploadedFiles	Set<Files>	Set of uploaded files
errorMessage	String	Error message returned if the upload fails

2.7.2.1.7 ChartClickEvent

The ChartClickEvent is fired when a data point of a chart element is clicked.

The event contains the data series, key, and values of the clicked chart location. This allows you to implement, for example, data drill-down.

Table with event properties

Property	Data Type	Description
source	UIComponent	Component that produced the event
series	String	Label of data series that was clicked
key	Object	Key value for the data point
value	Decimal	First value defining the data point
value2	Decimal	Second value defining the data point
payload	Object	Payload of the data point

2.7.2.1.8 WidgetChangeEvent

The WidgetChangeEvent is fired when a widget is added, removed, resized, moved, or hidden.

Table with event properties

Property	Data Type	Description
source	UIComponent	widget component that produced the event
widgetId	String	ID of the widget that produced the event set in the Widget ID parameter
configuration	WidgetConfiguration	Widget configuration with details about the widget visualization properties

2.7.2.1.9 CalendarCreateEvent

The CalendarCreateEvent is fired by a calendar component when the user clicks and drags over a period in a calendar. The event holds the selection data as its payload and the data can be used to create a new calendar entry.

Table with event properties

Property	Data Type	Description
source	Calendar	Calendar component that produced the event
from	Date	Start date of the selected period
to	Date	End date of the selected period
allDay	Boolean	If the entry is a whole-day event (if selected across days, the entry is an allDay entry; if the selected area is across hours, the allDay property is false and the exact hours are included)

2.7.2.1.10 CalendarEditEvent

The CalendarEditEvent is fired by a calendar component when a calendar entry is clicked. Note that the event has as its payload the business object of the calendar entry that was clicked.

Table with event properties

Property	Data Type	Description
source	Calendar	Calendar component that produced the event
data	Object	Business object of the calendar entry that was clicked

2.7.2.1.11 CalendarRescheduleEvent

The CalendarRescheduleEvent is fired by the calendar component when a calendar entry is dragged-and-dropped to a different date. Note that the event has as its payload the business object of the rescheduled calendar entry.

Table with event properties

Property	Data Type	Description
source	Calendar	Calendar component that produced the event
from	Date	Start date of the new period
to	Date	End date of the new period
data	Object	Business object of the calendar entry that was rescheduled

2.7.2.1.12 GeolocationEvent

The GeolocationEvent is fired by the Geolocator component after the component has acquired the geographical position of the user or when the request for location times out.

Table with event properties

Property	Data Type	Description
source	Geolocator	Geolocator component that produced the event
position	Geoposition	Data on position including latitude and longitude, speed, altitude, etc.
failure	GeolocatorError	Type of error if locating failed

2.7.2.1.13 MapClickedEvent

The MapClickedEvent is fired by the Map Display component when the user clicks into the map.

Table with event properties

Property	Data Type	Description
source	MapDisplay	Map Display component that produced the event
point	GeographicCoordinate	point that was clicked

2.7.2.1.14 MarkerClickedEvent

The MarkerClickedEvent is fired by the Map Display component when the user clicks a marker.

Table with event properties

Property	Data Type	Description
source	MapDisplay	Map Display component that produced the event
makerData	Object	underlying marker business object (the respective object of the set defined in the Markers property of the Map Display component)

2.7.2.1.15 MarkerDraggedEvent

The MarkerDraggedEvent is fired by the Map Display component when the user drag-and-drops a marker.

Table with event properties

Property	Data Type	Description
source	MapDisplay	Map Display component that produced the event
makerData	Object	underlying marker business object (the respective object of the set defined in the Markers property of the Map Display component)
newLocation	GeographicCoordinate	new marker position after dropped

2.7.2.1.16 MenuEvent

The MenuEvent is fired when the user clicks an item in the context menu.

Table with event properties

Property	Data Type	Description
source	UIComponent	Component with the context menu
id	Object	id of the clicked MenuItem object

2.7.2.1.17 TreeEvent

The TreeEvent is fired when the user expands or collapses a tree item, be it in a tree or a tree table component.

Table with event properties

Property	Data Type	Description
source	UIComponent	Parent Tree or TreeTable of the treeItem
treeItem	TreeItem	TreeItem object that produced the event

2.7.2.1.18 TablePageSizeChangeEvent

The TablePageSizeChangeEvent is fired when the user changes the size of a paged table.

Table with event properties

Property	Data Type	Description
source	Table	Parent Table
pageSize	Integer	page size after the change

2.7.2.1.19 PopupCloseRequestEvent

The PopupCloseRequestEvent is fired when the user clicks the close button in the caption of a popup component.

Table with event properties

Property	Data Type	Description
source	Popup	Popup that requested close

2.7.2.1.20 ApplicationEvent

The ApplicationEvent can be produced by any listener: it is the only event, the user can define out-of-the-box↩ : When a listener catches its event, it can fire an ApplicationEvent as part of its logic. The event can be caught by any ApplicationEvent listener in the form, and that including listeners on any hidden components and reused forms.

Table with event properties

Property	Data Type	Description
eventName	String	Custom name of the ApplicationEvent
payload	Object	Custom event data

For example, let's assume a form for placing orders. It contains multiple nested reusable forms: one contains customer details, another ordered items, and the last one invoicing details. All three reusable forms need to be validated before the order can be placed. The Place Order button located in the parent form, will fire an Application↩ Event that will be handled by ApplicationEventListeners on the reusable forms. The listeners will trigger validation and any other actions needed as part of the event handling.

To distinguish application events, use the event fields.

> For example, let us assume an application form with personal details, application details has Reject and Accept buttons, and a Reusable form with additional data on reject. If the user rejects the application, the provided data must be validated and the reject form must appear and they must provide the relevant rejection data. If they accept the application, no comment is required. However, all other components must be still validated.

The underlying form will define the two buttons with listeners that throw the OnSubmit ApplicationEvent. However, the Reject button will throw an ApplicationEvent with the REJECT value as its payload field, while the Accept button throws an ApplicationEvent with the ACCEPT value as its payload field. The input component for the comment must therefore define an ApplicationListener that will check that the ApplicationEvent payload is ACCEPT or if the payload is REJECT, it will check that the comment component is not empty.

2.7.2.2 Listeners

Listeners define how an event of a particular type from a particular component is processed in the event-processing cycle when it occurs.

When the component produces an event of the expected type, the listener catches the event and sends it to the event-processing cycle. The event is processed as defined by the properties of the listener.

Listeners defines the following:

- Basic properties determine what events they catch and how they are handled:
  - **Listener type**: the type of event the listener handles
    A listener catches only events with the same type as the listener type. You can use the Generic Listener if you want it to handle any event on the component.
  - **Refresh components**: comma-separated list of component IDs that are refreshed, that is, their content is recounted and the components are re-rendered.
  - **Validators**: validation expression
    The event is handled only if the validation expressions are `true`.
  - **Data validation**: validation expressions that validate against Constraints
    When using data constraints, the expression should include a `validate()` call that will trigger the constraint validation.
  - **Execute if other validations failed**: select to ignore failed validation on other than ValueChangeEvents (refer to chapter on the settings)
  - **Execute even if invalid components**: enforces execution of the listener logic even if the form contains incorrect data, such as, "aaa" instead of an integer, to allow operations as refresh or reset on click.
  - **Handle expression**: expression that is executed
    The expression represents the action the listener performs to handle the event.
  - **Event identifier**: identifier of the received event
    You can use the identifier in the Handle expression to acquire the event fields with event details, such as the component that produced the event, uploaded files, etc.; refer to Filtering Events on Listeners.
- Advanced properties define the context of the execution and actions as well as handling pre-condition:
  - **Process components**: comma-separated list of component IDs; only events produced by the components are included in the event-processing cycle
    The property allows you to ignore queued event from the specified components of the form, typically, if you have a form with multiple tabs and you want to ignore events with their validation on another tab, or to prevent validation on all components when resetting the content of one component.
  - **Execution context**: execution context of the listener
    All expressions defined in the listener properties, such as, its precondition, validation, etc. are evaluated in this context level.
    * default: the context level of the component with the listener
    * top level: the screen context of the form
    * component: ID of the component which represents the target execution context

Important: The execution context setting does not influence how View Models are merged: this is set by the merge type property on the View Model.

- **Execute only if visible components**: comma-separated list of component IDs; the event handling takes place only if all specified components are displayed in the form.
- **Clear/merge view model components**: IDs of view models (their contexts are cleared or merged to the target execution level).
- **View model init**: expression executed right after the merge or clear of view model components

  The expression serves as a hook after the merge-clear phase.
- **Precondition**: pre-condition for event handling

  If the pre-condition evaluates to false the event is not handled.

• Actions properties define what actions are performed on the to-do or document after the event is processed:

- **Submit**: defines if the document or to-do is submitted as part of the event handling;

  If enabled, the relevant data in the components is persisted and the to-do becomes finished or the document closes.
- **Persist**: if enabled, the relevant data in the processed components is persisted as part of the event handling; unlike on submit action, if the form is used by a to-do, the to-do does not become finished; if part of a document, the document remains open; persist is performed after the merge to the screen level before the transaction is committed.

  You can define a persist action, which is performed immediately after the persist.
- **Save action**: defines if the to-do or document is saved;

  The save action saves the current state of the to-do or document for later editing, which includes saving the provided data; it is identical to clicking the Save button on a to-do or document in the LSPS Application User Interface; the save action does not persist the data; therefore make sure to define the Persist property if required. If a newer version of the saved to-do or document is available, the application displays a notification.

  Note: A saved document is persisted in the system database as the *SavedDocument* record. If saved repeatedly, the same record is overwritten; hence only the last saved document version is available. On submit, the persisted document is removed. For more information on the record type and related functions, refer to the Standard Library documentation.
- **Navigation**: defines the location where to navigate after the event is processed; you can navigate to a to-do or document, URL, custom application page (refer to the descriptions of data types defined in the human.navigation.datatypes resource in the Standard Library)

  Navigation takes place after persisting, which allows you to use the persisted data in the navigation expression.

  Important: If the form is used in documents, the form navigation is overridden with the document navigation.
- **Fire application event**: ApplicationEvent the listener produces

• Expression tab allow you to define the entire listener as an expression: when you select the **Listener is defined by expression** option, the properties defined on the other tabs are reflected in the generated expression

2.7.3 Creating Forms

When creating forms, you will typically do the following:

- Create a form definition.
- Create form parameters and variables.
- Design the form-component tree and its behavior.

2.7.3.1 Defining Form Parameters

To define form parameters, do the following:

1. Make sure the form is open in the form editor.
2. In the GO-BPMN Explorer, right-click the form and select **New Parameter**.
   Alternatively, you can open the form properties in the Properties view and click Add on the Parameters tab.
3. In the Property view, define the parameter properties.

When the form is then called, the call can provide arguments that can be used inside the form.

```
applicationForm(user -> admin, requestedHardware -> Hardware.ssd)
```

2.7.3.2 Defining Form Variables

Form variables are initialized when the form is displayed, that is, when an InitEvent is fired. In your forms you should prefer form variables to global variables so as to keep your presentation Data separated from the business data.

To define a form variable, do the following:

1. Make sure you have the form resource opened in the Form editor.
2. In the Outline view, right-click the form and go to **New** > **Variable**.
3. In the displayed Properties view, define the variable properties.

2.7.3.3 Designing Form Content

To create the content of your form, open the form definition file (double-click it in the *GO-BPMN Explorer*). Then either click the required component in the palette and then click into the canvas to insert it, or right-click the canvas, go to *Insert Component* and select the component from the context menu.

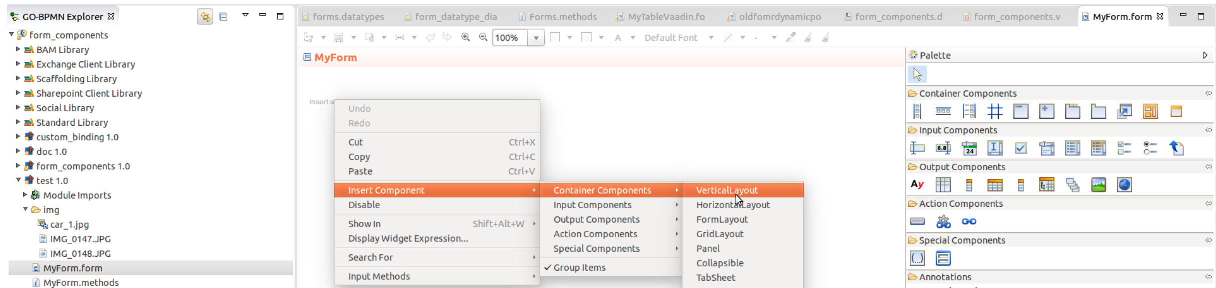


Figure 2.61 Inserting a component into the form from the context menu

Note: You can create forms also with the `dynamic-gui` functions, such as *createAndAdd()*. However, these functions are not maintained and will not be enhanced.

Once you have created your form or while doing so, you can do the following:

- define basic behavior of the form and its components
- define validation
- reuse form components
- modify presentation properties of components
- define mobile forms

2.7.3.3.1 Inserting a Parent Component

You can select one or multiple form Components and wrap them with another components in the Form graphical editor.

To insert such a parent form component over another component, do the following:

1. In the Form editor, right-click the component.
2. In the context menu, go to Insert Parent and select the component to use as the wrapper component.

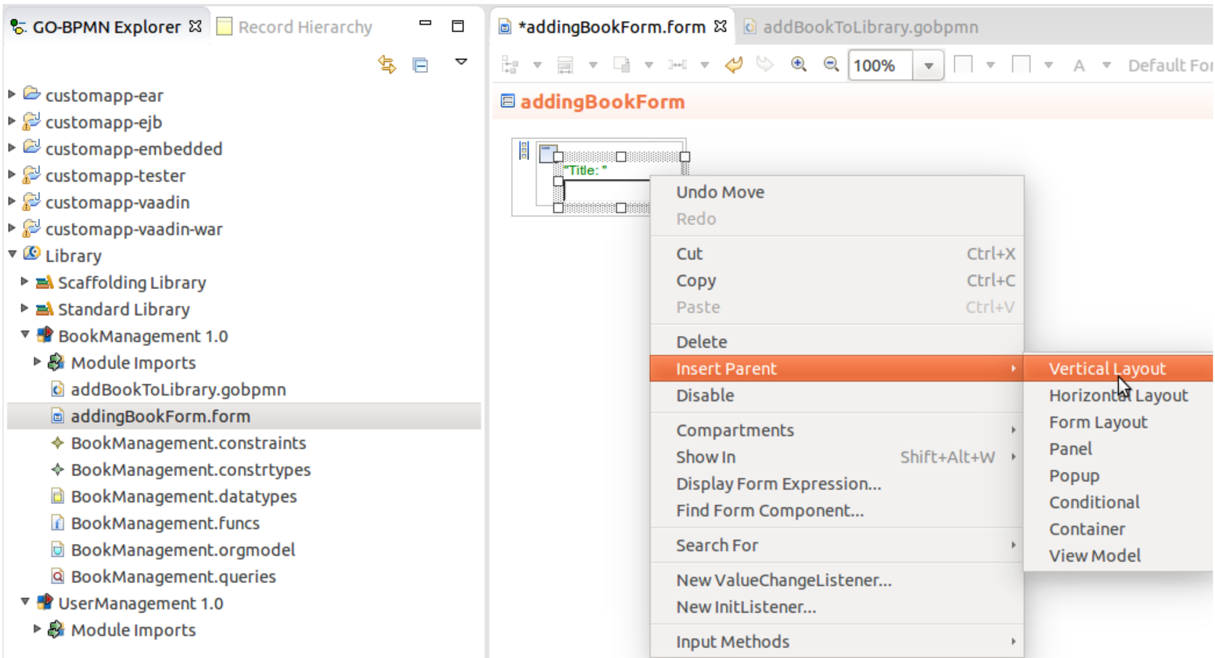


Figure 2.62 Wrapping component in a layout component

2.7.3.3.2 Deleting a Parent Component

To delete a only a parent component and preserve its child components, right-click the parent component and click **Shift Children Up**.

Note that the option is only available if the children can be accommodated in the form after the deletion; for example, it is not possible to delete a Vertical Layout with multiple child components if it has a Panel component as its parent.

2.7.3.3.3 Previewing Forms

While creating a form, you can display its preview in the browser.

Make sure a local LSPS Server is running.

When displaying a form preview, the server runs a persisted instance of the model that contains the form definition (global variables are initialized, the Form parameters are initialized, the form screen context is created). The model instance is persisted.

To display preview of your form in your web browser, do the following:

1. In the GO-BPMN Explorer, right-click the form.
2. Select Run As -> Form Preview and on the displayed application page in your browser, log in to the application.

In the Management perspective, you can delete any model instances that you triggered as form previews in the Model Instances view: right-click anywhere into the view and click Remove All Form Preview Model Instances.

2.7.3.3.4 Displaying Form Source

Since forms are defined as functions and their components as variables of the respective record type on the level of the Expression Language, you can display their source in the Expression editor:

1. In the Form editor, right-click any form component.
2. On the context menu, select Display Form Expression.

2.7.3.3.5 Searching for a Form Component

If an error on a form component occurs during runtime, the server returns an error message with the modeling ID of the form component. To find the form component, do the following:

1. Go to Search Find Form Component.
2. In the displayed search dialog, enter the modeling ID.

2.7.3.3.6 Defining a Context Menu

Every form component can define its context menu. The menu and its content is defined as a component property. When the user right-clicks the component, the context menu with its items is displayed. The user can then click a context menu item. On click, a MenuEvent is produced. The MenuEvent has the id of the clicked menu item as its payload so the form can define the MenuListener that will trigger the respective actions.

The context menu can be defined as static or dynamic: A static context menu is calculated only once before the form is rendered, while a dynamic context menu is recalculated on every right-click. This might cause performance issues since it could require accesses to server on every right-click.

If you define both a static and a dynamic context menu on right-click, the displayed context menu will contain the static menu items on top and the dynamic menu items below.

To define a context menu on a component, do the following:

1. Select the component in the form definition.
2. In the Properties view, open the Context Menu tab.
3. On the tab define either a static context menu or a dynamic context menu.

```
[new MenuItem(
            caption -> "Open the Detail",
            htmlClass -> "contextmenu",
            id -> 0,
            submenu -> [new MenuItem(
                    caption -> "Open in a New Tab",
                    htmlClass -> "contextmenu",
                    id -> 1),
                    new MenuItem(
                    caption -> "Open in This Tab",
                    htmlClass -> "contextmenu",
                    id -> 2)
                    ]
            )
]
```

2.7.3.3.7 Defining a Listener

You can define listeners on any form component; however, every component type allows only listeners for events it can actually produce; for example, a Button component can define only an Action listener, which is fired when the user clicks a button. It cannot define a ValueChange listener since there is no value to be changed on a Button. On the other hand, a component might produce various events: A button produces just like all components an InitEvent when it is created: An ActionListener on the Button will ignore the event: it catches only the events of a particular type on its component.

Note: To listen for an event on another component, you can do one of the following:

- To listen for events from child reusable forms or to listen on events from parent forms, use Container interface with public listeners and registration points.
- Produce an application event.

To create a listener on a form component:

1. In the Form editor with your form definition, select the form component.
2. In the Properties view of the component, click the *Event Handling* tab.
3. In the *Private Listeners* section, click **Add**.
4. In the Add Listener view, define the Listener properties: you can do so by gradually going through the *Basic*, *Advanced*, and *Actions* tabs or you can go to the *Expression* tab and define your listener in an Expression: if you have already defined some properties on the other tabs, these are reflected in the listener expression when you select the **Listener is defined by an expression** option.

Note that if a tab contains a change, its tab label is bold.

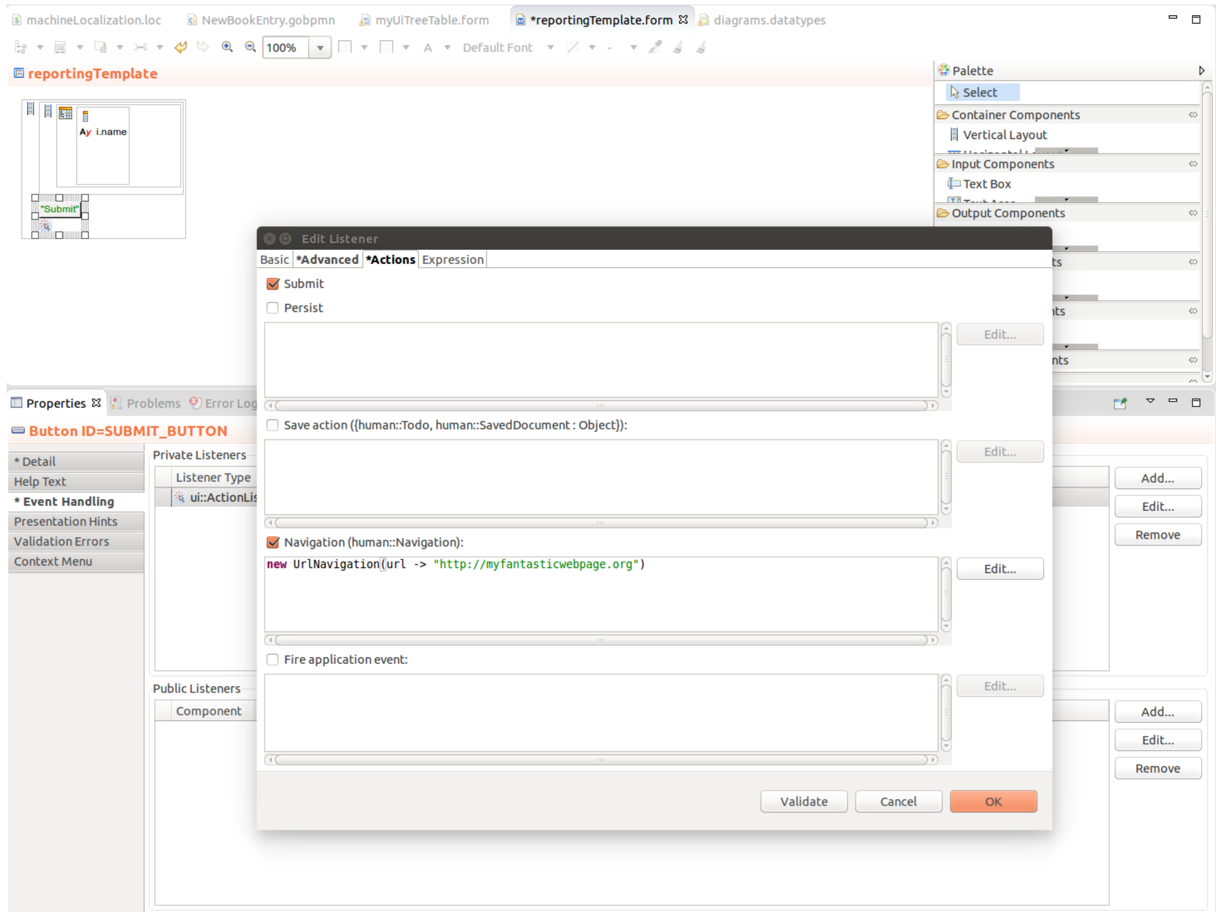


Figure 2.63 Defining an ActionEvent listener on a submit Button component

2.7.3.3.7.1 Filtering Events on Listeners

To filter out an event so the listener does not catch it, use one of the following listener properties on the Advanced tab:

- **Execute only if visible components**: The event is processed by the listener only if the component that produced the event is visible.
- **Precondition**: The event is only processed if the precondition evaluates to `true`.

```
if ((_event as ApplicationEvent).payload as Integer) = 1
  then true
  else false
end
```

- **Event name** (available only for ApplicationEventListeners): An application event is only handled by the listener, if the name of the event matches the defined event name.

Note: An event might not be processed also if the validation process fails.

2.7.3.3.7.2 Filtering Queued ValueChangeEvents

If you want to ignore some queued ValueChangeEvents when a listener is triggered, define on the listener the *Process component* property: the property defines a list of form components whose ValueChangeEvents are included in the event-processing cycle. Any other ValueChangeEvents remain in the queue.

For example, let's assume a table with column A and column B. Both columns contain input components that are not-immediate and buttons used to submit the values from the given column. When the user changes values, ValueChangeEvents from both columns are kept in the event queue. Then the user clicks the submit button in column A. The value change listener must define as its Process component only column A. If it defines as its Process component column B as well, all the ValueChangeEvents will be processed.

2.7.3.3.7.3 Disabling a Listener

To disable a listener, open its properties and select *Listener is disabled* on the Basic tab.

2.7.3.3.7.4 Refreshing Form Components

To refresh the content of form components, do the following:

1. Define IDs on the component you want to refresh.
2. Create a private listener on the component that should cause the refresh.
   (a) In the Properties view of the component, click the *Event Handling* tab.
   (b) In the *Private Listeners* section, click **Add**.
   (c) In the Add Listener view on the Basic tab, define the Listener properties:
      - Listener type: the type defines the event type the listener handles.
      - Refresh components: define IDs of the components that should be refreshed when the event is handled.

Alternatively, you can call the *refresh()* function from the handle expression (for example, `refresh([MYTABLE, MYPOPUP])`).

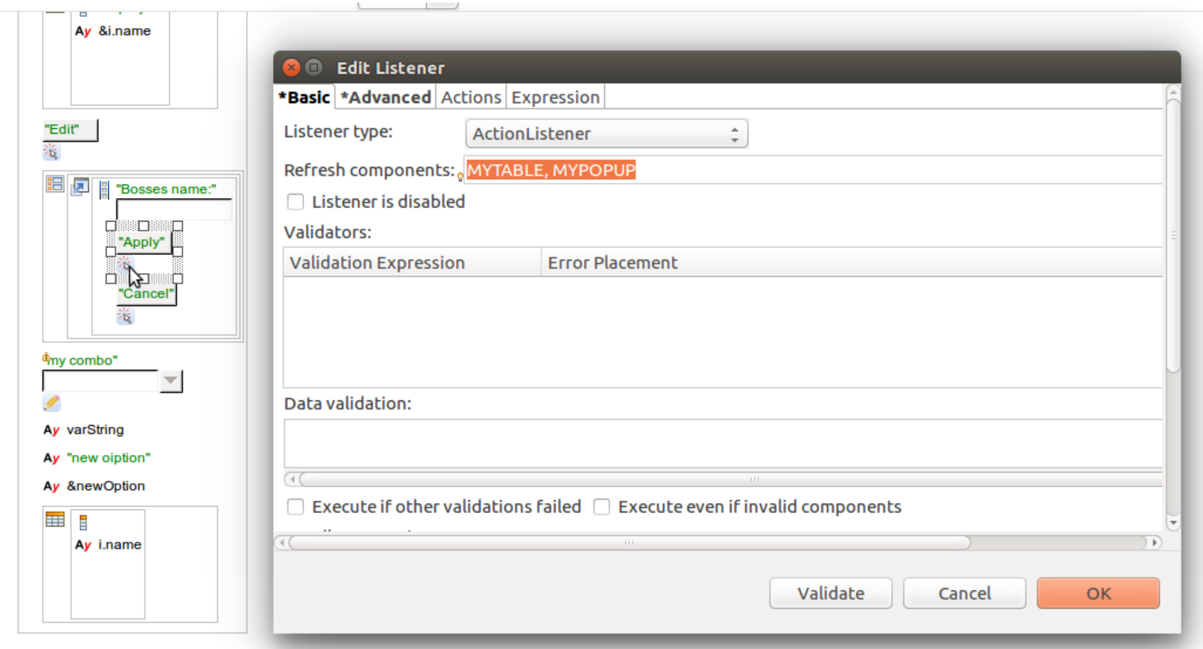


Figure 2.64 Listener that refreshes multiple components

2.7.3.3.7.5 Persisting Form Data

To persist the base context of a document or to-do base as part of the event handling, enable the Persist action on the respective listener: the data in the process components and the to-do or document remains open with the underlying process remaining in unchanged status.

Note that the scope of data that is persisted depends on the **Process components** setting: if this setting defines a set of form components, only the data of these components is persisted.

The Persist action is performed after the merge to the screen level before the transaction is committed.

Below the **Persist** checkbox, define a persist action, which is performed immediately after the persist, if applicable.

Alternatively, you can call the *persist()* function from the handle expression.

2.7.3.3.7.6 Saving Documents and To-Dos

To save the state of the to-do or document for later editing, define the *Save* action on a listener. The action is identical to clicking the Save button on a to-do or document in the LSPS Application User Interface. Note that the save action does not persist the data, therefore make sure to activate the Persist action if required.

> **Note:** A saved document is persisted in the system database as a **SavedDocument** record. If saved repeatedly, the same record is overwritten; hence only the last saved document version is available. On submit, the persisted document is removed.

The *Save action* expression is a closure that has the saved to-do or document as its input parameter. It serves to further process the saved document or to-do; for example, store the to-do or document in your own data source.

To implement this feature, you will need to define the following:

- a shared Record that is in the 1:1 relationship to the human::SavedDocument record
- a document or a todo that will contain a form with a listener that will save the data as your SavedDocument subrecord in its Saving Action expression
- implementation that will recover the saved documents, for example, another dedicated Document with a navigation
- possibly the option for deletion of sa

Note that by default, the save action does not save Column states (width and collapse state). To save these, process as described in Saving Column Width and Collapsed State.

2.7.3.3.7.7 Submitting

When the user submits the document or to-do, the data is saved, typically in the underlying DB via shared Records, and; in the case of documents, the model instance ceases to exist, and, in the case of to-dos, the user task finishes.

Note that the document will remain available in the list of documents since the list contains the types of documents, not their instances: Unlike To-Dos, documents availability does not depend on a Task; they are available as long as the definition of the document is on the server and their instances are created on request.

To define, when the document or to-do is submitted, do one of the following:

- on the component that should submit the document, create a listener of the required type and on the Action tab in the listener properties, select Submit.

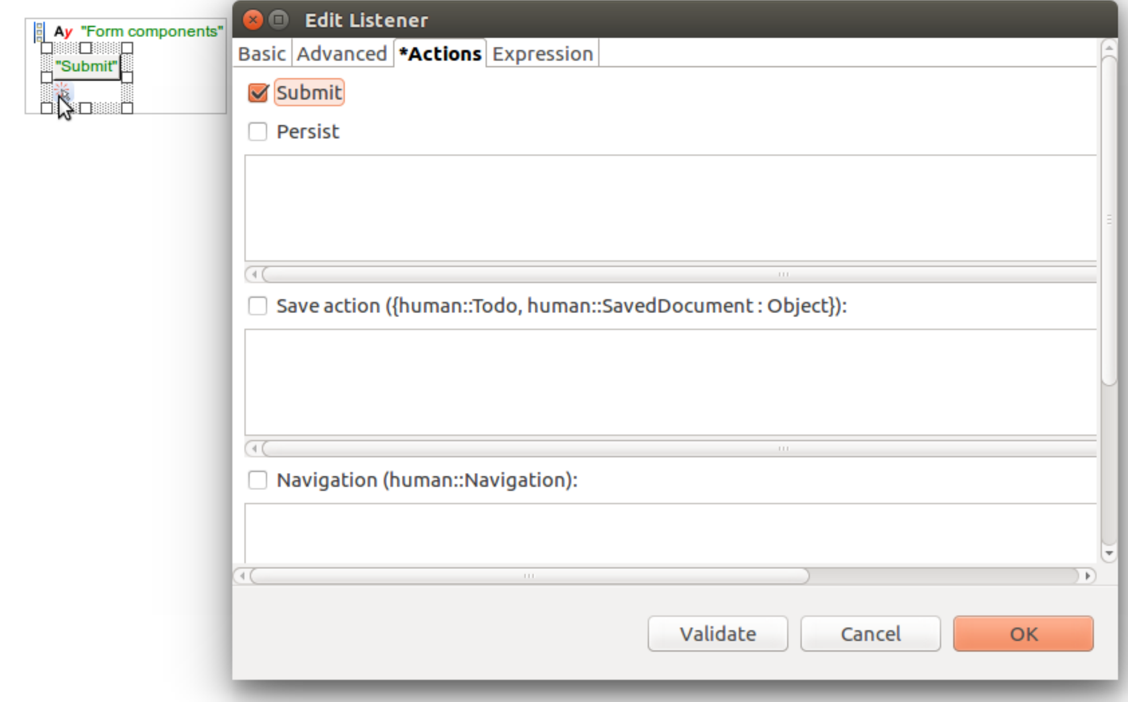


Figure 2.65 Submit action on a click listener

- on the component that should submit the document, create a listener of the required type and call the *requestSubmit()* or *requestSubmitAndNavigate()* from the Handle expression.

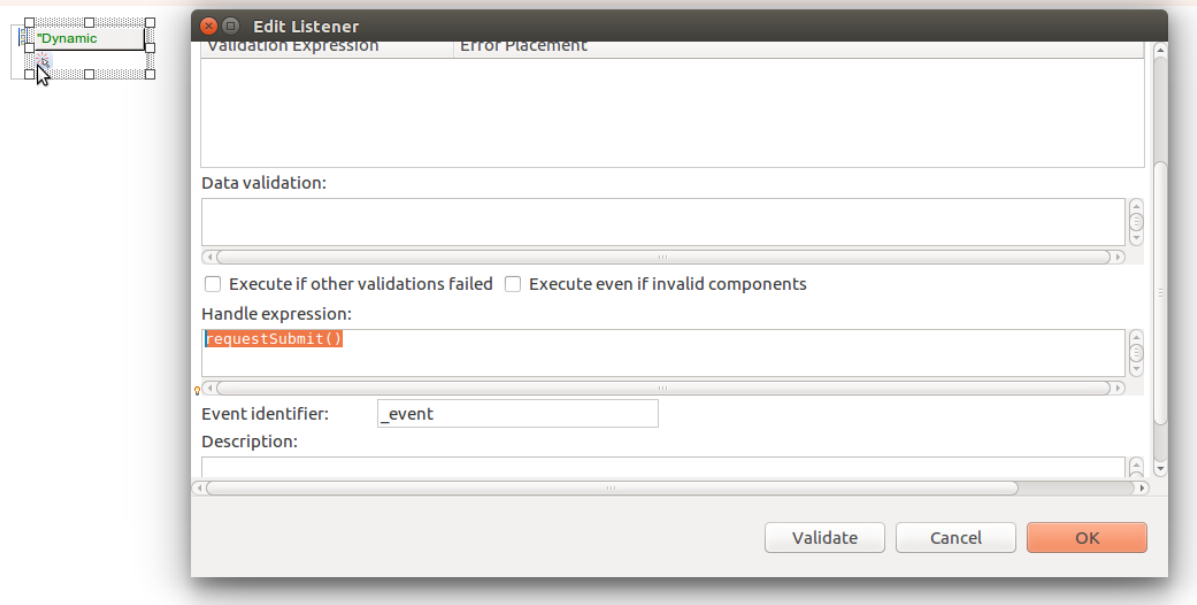


Figure 2.66 Submit call from a listener handle expression

2.7.3.3.7.8 Navigating Away

To navigate to a location when the form is submitted or saved, define the *Navigation* on the Action tab of the listener: you can navigate to a to-do or document, URL, custom application page (refer to the descriptions of data types defined in the human.navigation.datatypes resource in the Standard Library).

> **Note:** You can define a Navigation also on the document definition: This Navigation overrides the Navigation defined on the form (that is, if you define a button that submits the form, the form will navigate as defined in the Navigation closure on the document; not in the navigation of the submit component).

Navigation expression

```
//redirect to the document Confirmation when the event is handled:
new DocumentNavigation(documentType -> confirmationDocument())
```

> **Note:** Navigation expression is evaluated right after persisting, which allows you to use the persisted data in the navigation expression. However, the action is taken only after the submit or save action is performed.

2.7.3.4 Validating Form Data

You can validate data in your form with the following mechanisms:

- automatically triggered validation that is performed as part of the event processing directly on listeners as one of the following:

- **Validators**: set of expressions that check if values in the form are valid and return a String with an error message if they are not

  The String is displayed as an error message either on the current component or the component defined in the *Error placement* property.
- **Data validation**: expression that returns a set of constraint violations rendered automatically on the components with the violations

• explicitly triggered validation that you can perform from the Handle expressions of your listeners

  You will need to collect a set of constraint violations, typically, by calling the `validate()` function on the respective Record, and then call the `showConstraintViolations()` function to display the violations in the form. For further information, refer to the documentation on Record validation.

2.7.3.4.1 Defining Validation in Listener Handle

When validating a Record in the Handle expression of a Listener, do the following:

1. Collect a list with constraint violations; typically you will use the `validate()` function on the record or property.
2. Call `showConstraintViolations()` function to display the error messages of the violation on the respective form components.

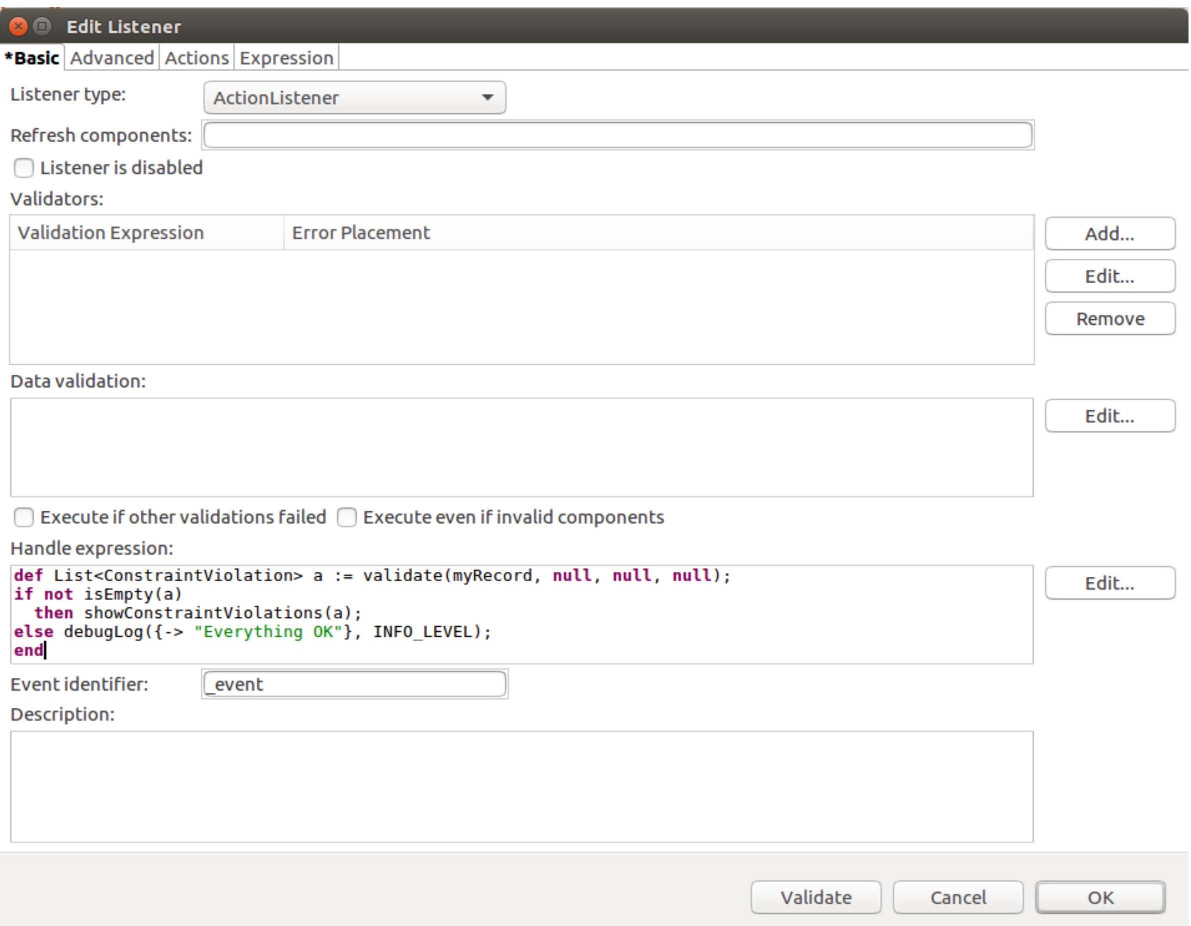


Figure 2.67 A record validated as part of a listener's Handle

2.7.3.4.2 Defining Automatic Form Validation

You can define both the validators and the data validation on listeners on their *Basic* tab:

- Validators with validator expressions that return a String when they fail, for example, `if searchVar!=null and length(searchVar)>0 then null else "Provide search string!" end`.

  The String is displayed as an error message either on the current component or the component defined in the *Error placement* property.

- Data Validation as an expression that returns a list of ConstraintViolations: to obtain such a list, use the validate() function.

  If you need to display a constraint violation on a particular component, create a constraint violation with record or property set to the same binding as your component; for example, `[new ConstraintViolation(payload -> null, record -> selectedRecord, property -> LookupValue.displayName, guid -> null, id -> null, message -> "Incorrect value.")]`

The automatic validation passes only if:

- all validator expressions from the entire form return `null` and
- data validation expressions from the entire form do not return any validation constraint violations.

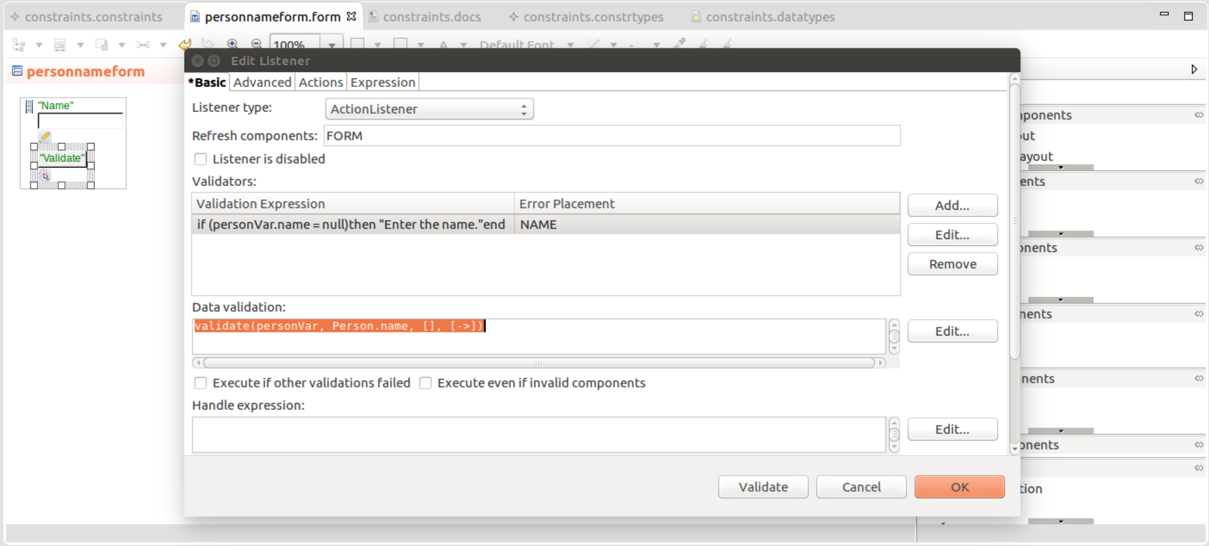


Figure 2.68 Listener with validators and data validation

2.7.3.4.3 Validation of non-ValueChangeEvents

The validation of ValueChangeEvents is governed by their own validation mechanism: they are executed always when *their* validation passes regardless of other failed validations on other listeners. Also their *Execute if other validations failed* setting is not taken into account.

In addition, validity of ValueChangeEvents has no impact on the general validity of non-ValueChangeEvents.

Consider the form below: the ISBN text box is in non-immediate mode and its ValueChangeListener defines a validator for the entered text: it must be at least 18 characters long. The Submit button has an ActionListener with the Execute if other validations failed property set to `false` and executes submit (the Submit property is selected).

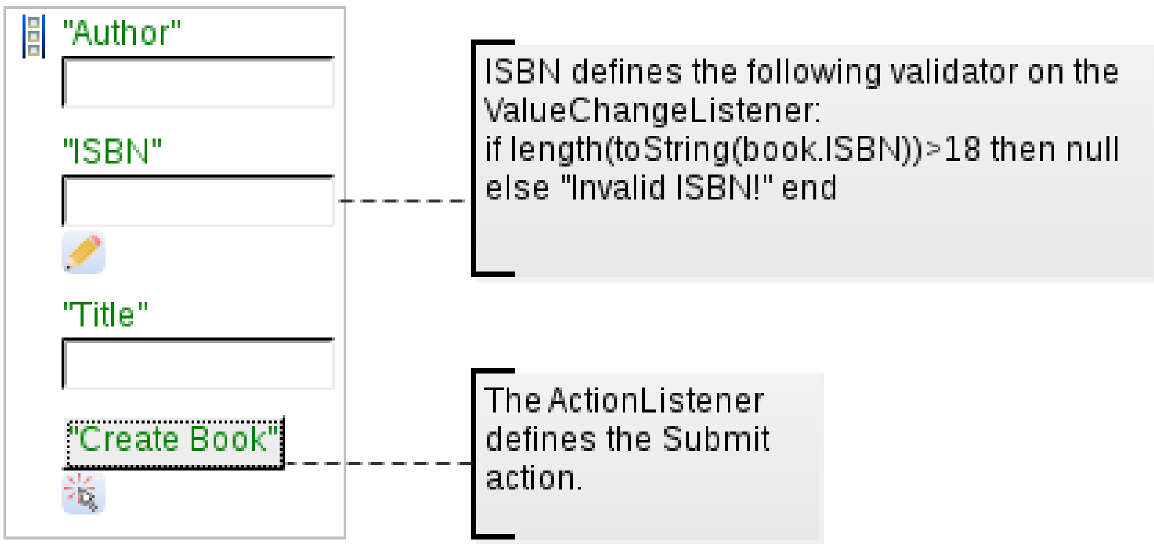


Figure 2.69 Book Create Form

If the user enters less than 18 characters in the ISBN text box and hits Submit, the invalid value is persisted and the form is submitted: the ActionListener on the Submit button ignores that the validation of the ChangeValueListener failed.

Impact of Listener Validations on their Execution:

Listener	Listener Type	Listener Validation Outcome	Execute if Other Validations Failed	Executed
A	Non-ValueChangeEvent	fail	false	no
B	Non-ValueChangeEvent	fail	true	no
C	Non-ValueChangeEvent	pass	true	yes
D	Non-ValueChangeEvent	pass	false	no
E	ValueChangeEvent	fail	true	no
F	ValueChangeEvent	pass	false	yes
G	ValueChangeEvent	fail	false	no

We assume that the listeners in the table exist in one context:

Listener A and B Validation on Listeners A and B failed and hence they are not executed regardless of their Execute if Other Validations Failed setting.

Listener C Validation on Listener C passed and its Execute if Other Validations Failed setting is true; hence it ignores the failed validation on A and B and it is executed.

Listener D Validation on Listener D passed. However, since its its Execute if Other Validations Failed setting is set to false and validation on other non-ValueChangeListeners (A, B) failed, it is not executed.

Listener E Listener E is a ValueChangeListener: its validation failed and hence it is not executed. The Execute if Other Validations Failed setting is ignored.

Listener F Listener F is a ValueChangeListener: its validation passed and hence it is executed. The Execute if Other Validations Failed setting is irrelevant.

Listener G Listener G is a ValueChangeListener: its validation failed and hence it is not executed. The Execute if Other Validations Failed setting is irrelevant.

2.7.3.4.4 Handling an Event When Validation Failed

If necessary, you can force the system to execute a particular listener even after the total validation had failed: Ignoring failed validations on other listeners allows you to perform an action even if the form contains invalid values.

It is then enough that the validation on that particular listener passes for the event to be handled: failed validations on other listeners are ignored.

This behavior is activated by the *Execute if other validations failed* property of the listener.

2.7.3.4.5 Filtering Validation Errors

When using validation of constraints on form components, by default, validation errors are displayed on the form component that contains the value with the error. However, you can define explicitly additional validation errors displayed on the component or hide such validation errors on the component:

- Excluding validation errors is intended for filtering of errors, so that only a subset of the errors is displayed on the form component.
- Including validation errors defines additional validation errors displayed on the component.

  These could be errors that would otherwise remain hidden. For example, when editing multiple record fields, which would result in an overall record constraint violation, the error for the entire record would not be displayed since there is no form component that references the entire record.

You can define the excluded or included validation errors on the Validation Errors tab of the form component properties.

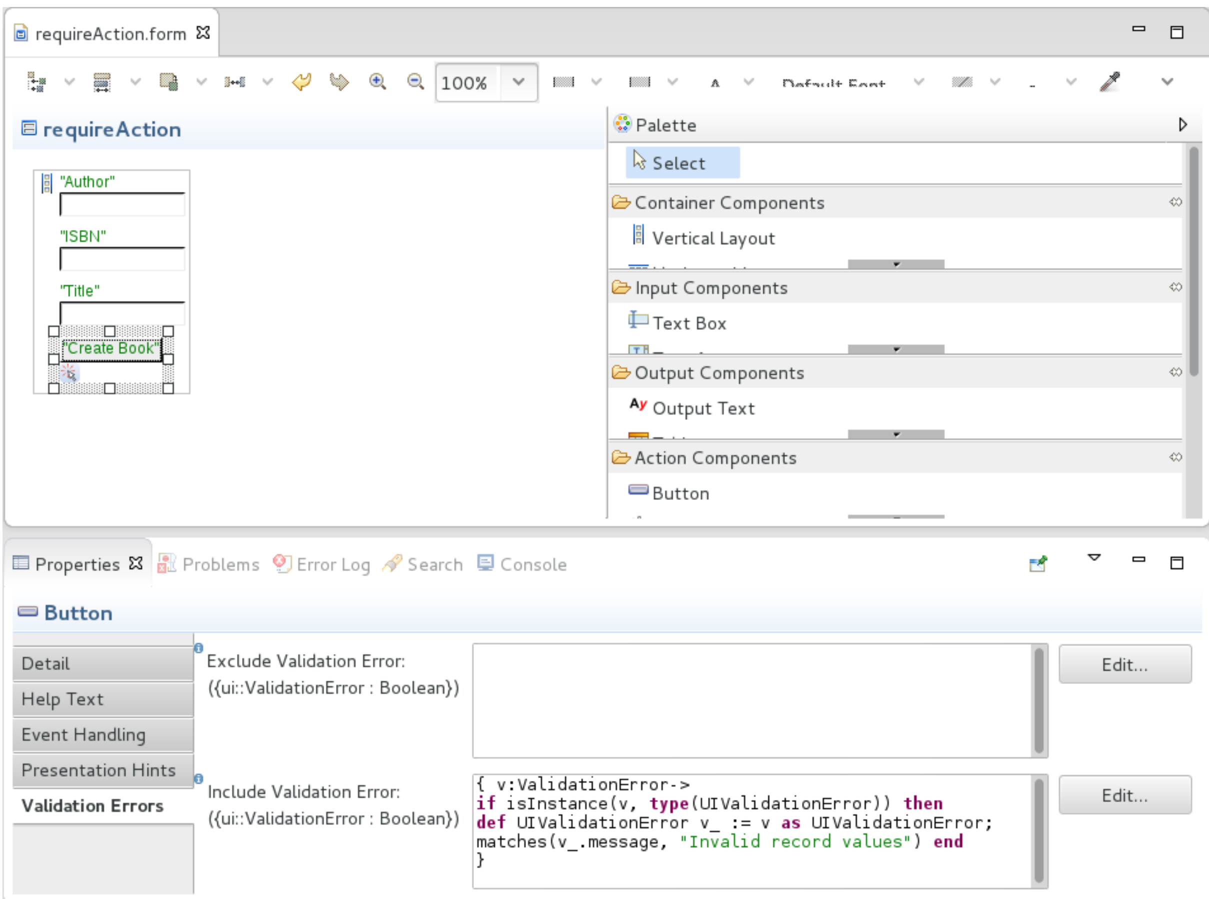


Figure 2.70 Including a Validation Error on a Form Component

If you want to ignore invalid data in a form and enforce processing of an event, select the **Execute even if invalid components** setting for the given listener: the listener will be always executed as part of the next event processing.

2.7.3.4.6 Validating Initialized Forms

To validate initialized forms, design the forms as follows:

1. On the form, define an InitListener which fires an ApplicationEvent.
2. Define an ApplicationEventListener that listens for the ApplicationEvent.
3. On the ApplicationEventListener, define the validation expression.

2.7.3.5 Reusing Forms

To reuse an already existing form, insert into your current form the Reusable Form component and set it to reference the existing form: On runtime, the Reusable Form inserts the referenced form into your form tree and renders it as its part.

2.7.3.5.1 Events in Forms with Reusable Forms

It is important to realize that events of the forms referenced by the Reusable Form component are not visible to the parent form and vice versa. To enable them to handle each other's events, you will need to define interface elements between them using the Container component:

- **To catch an event produced by the parent form and handle it in the reused form:**

  1. Wrap the reused form content in a Container component.
  2. Define a Public Listener that will listen for the event in the parent form.
  3. Add the Public Listener to the parent form.

  Detailed instruction are available in Sending Events to a Reused Form.

  > **Note:** If a public listener needs to listen on another higher parent context, you can register the listener as a delegate listener on the mediating Container.

- **To catch an event produced by a reused form and handle it in the parent form:**

  1. Wrap the reused form in a Container component.
  2. Define a Registration Point that will expose its component.
  3. Define a Private Listener on the parent form that will be registered with the Registration point and listen for the event.

  Detailed instruction are available in Receiving Events from a Reused Form.

  > **Note:** Registration points are implemented as references to the set of listeners defined on the Container component. If an event of the given type is fired within the form, it is handled as defined by the registered listener of the parent form.

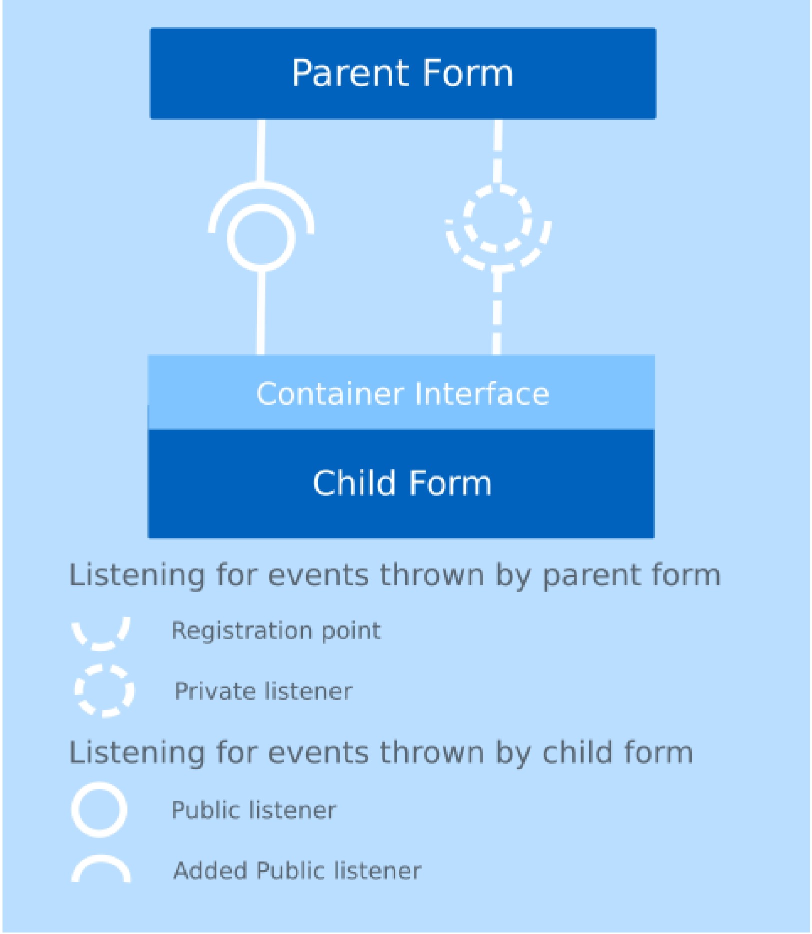


Figure 2.71 Schema of listener exposure in reusable forms with indications on how the events flow

Another way to handle events with listeners that are not defined on the component is to use Application Events: Application Events can be created by a listener of any type and handled by one or multiple listeners of any form component and that even if the component is hidden or located in a reusable form. Therefore it is not necessary to define any interfaces to handle Application Events. Note that this mechanism can result in an involute form definitions which are difficult to debug; therefore it is recommended to use Application Events sparingly.

2.7.3.5.2 Receiving Events from a Reused Form

If a form needs to handle an event that occurs in the form of its Reusable Form component, you need to expose the event using a registration point on the form referenced by the Reusable Form component and define a private listener with the required Actions on the Reusable Form component.

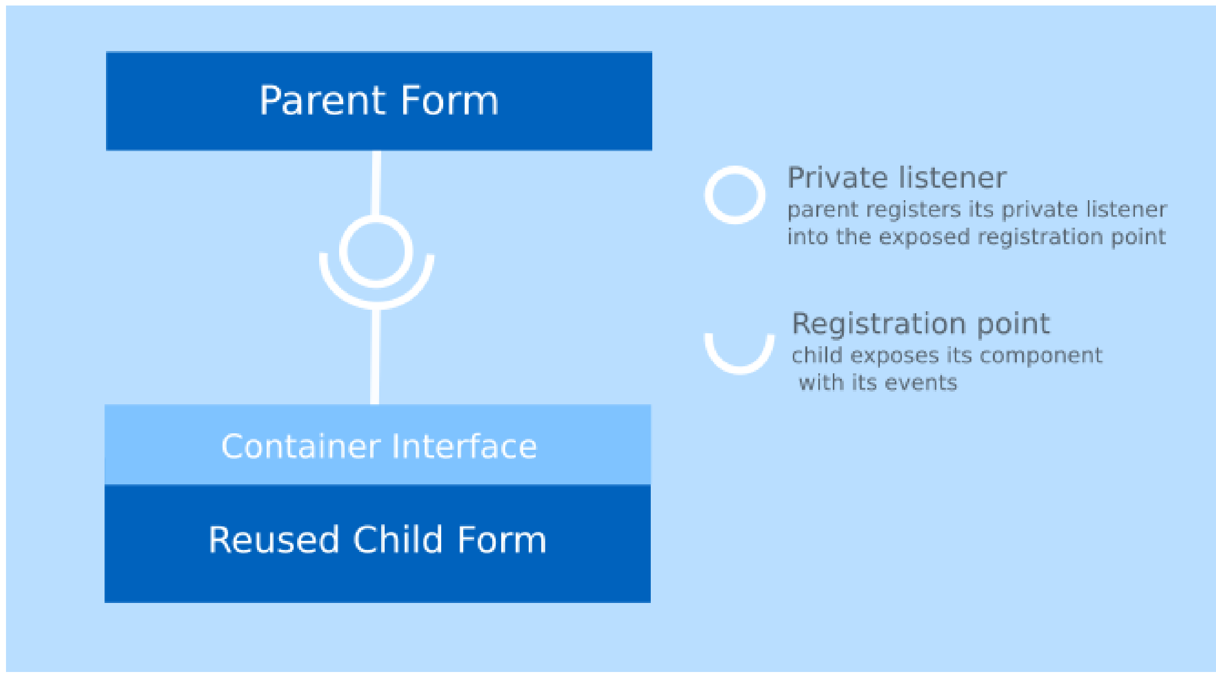


Figure 2.72 Listening for events that occur in a Reusable Form component

Note that if you need to expose an event that occurs in a form that is included via multiple Reusable Form components, you need to mediate the registration points into upper forms.

To define such event handling, do the following:

1. Create or open the child form that will be referenced by the Reusable Form component of a parent form:

   (a) Insert the Container component as the root component of the form.

   (b) On the Container, define the registration point that will register the listener of the parent form:

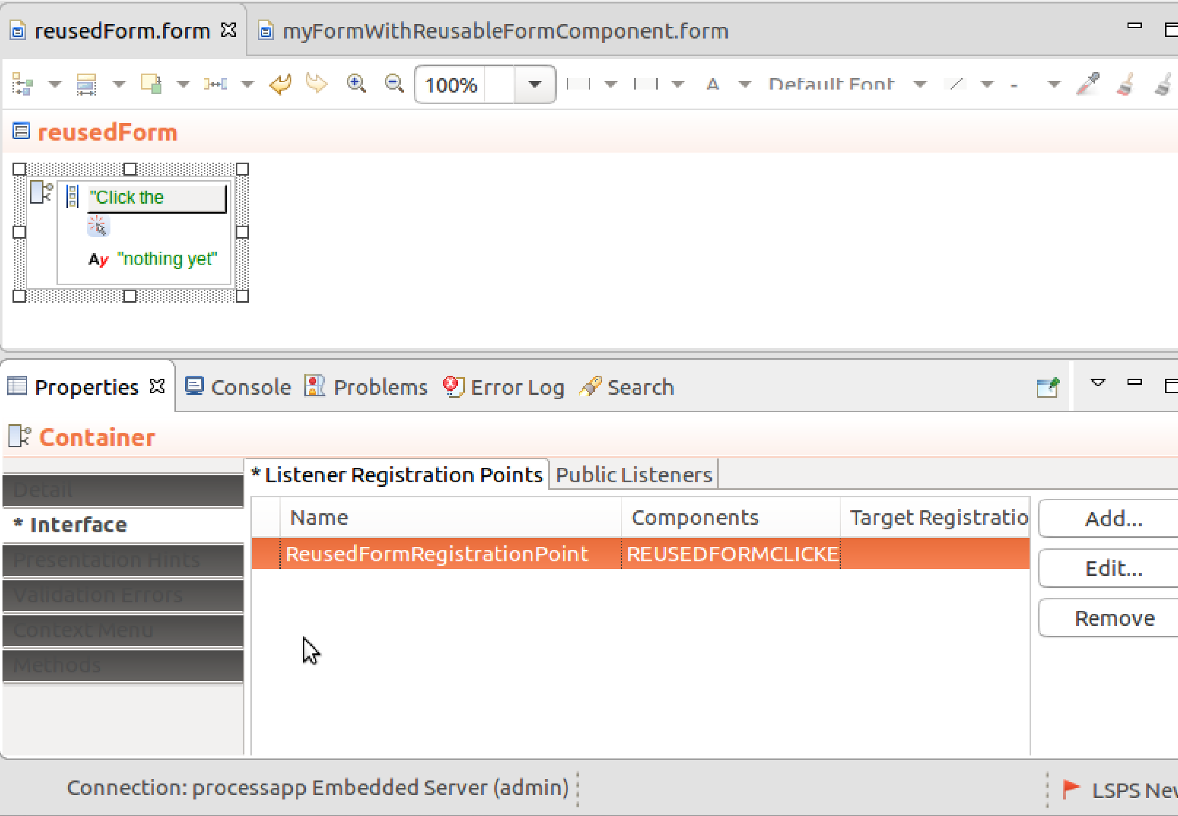
      i. Select the Container.

      ii. Go to the Properties view.

      iii. In the Interface properties, click the *Listener Registration Points* tab and click **Add**.

      iv. In the Add Listener Registration Point dialog box, define the registration point properties:

         - Name: name of the listener defined on the Reusable Form component
         - Components: IDs of the components the listener listens on



2. Create or open the form definition that will reuse the form you just created:

   (a) Insert the Reusable Form component available under the Special Components and define its properties:

   (b) As its *Form* property, set the form from the previous step.

   (c) On the *Event Handling* tab in the *Private Listeners* section, create a private listener with the following properties:

      - **Listening on registration point**: the registration point you defined on the Container of the reused form (use autocompletion)

      - Any other relevant listener properties.

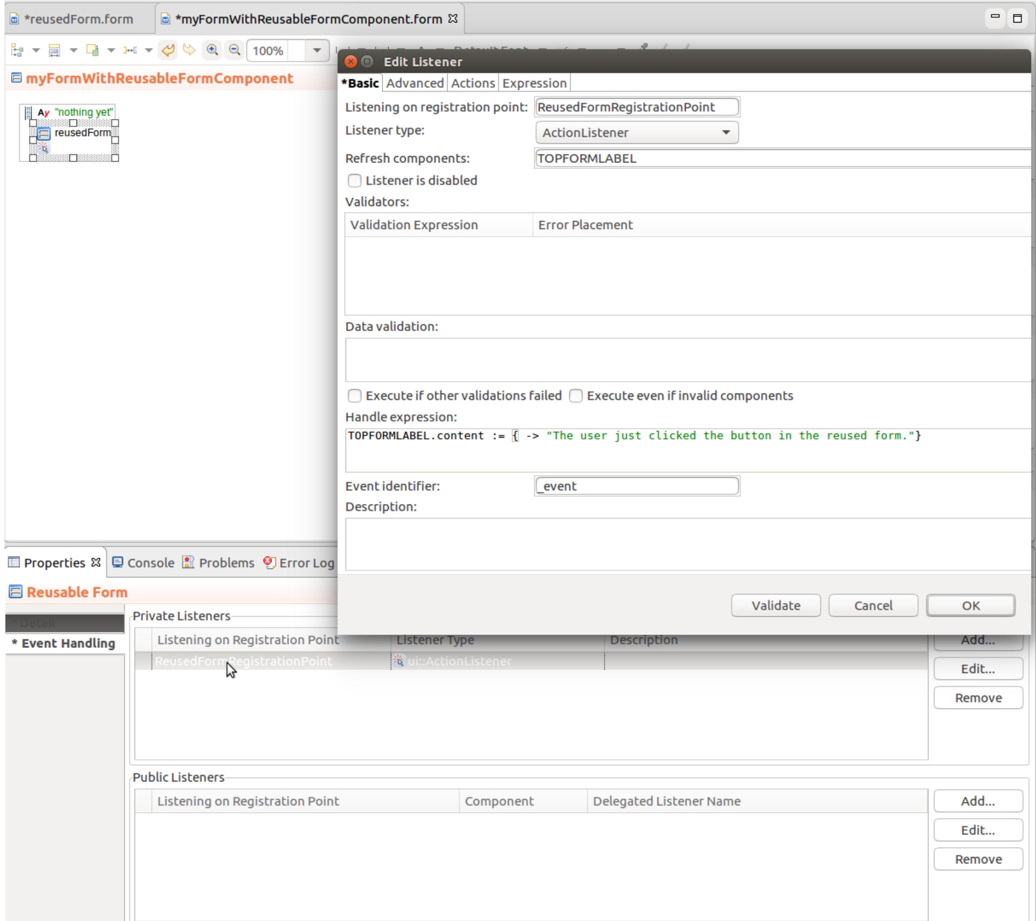


Figure 2.73 Reusable form component with a private listener for a registration point

3. Test your form on the Embedded Server: right-click the form definition, go to **Run As** > **Form Preview**.

2.7.3.5.2.1 Receiving Events from a Reused Form across Multiple Reusable Forms

If a form needs to handle an event that occurs in a Reusable Form of another Reusable Form component, you will need to delegate the Registration Point through the mediating Reusable Forms:

1. Create the child form that will be referenced by multiple Reusable Form components.

   The form must have the Container component with the registration point as its root component.

2. Create or open the mediating form:

   (a) Insert the Container component as the root component.

   (b) Insert the Reusable Form component that references the reused form step 1. Make sure to define its ID.

   (c) On the Container component, create Registration Points that will mediate the Registration point from the reused form:

      i. Open the Properties view of the Container.

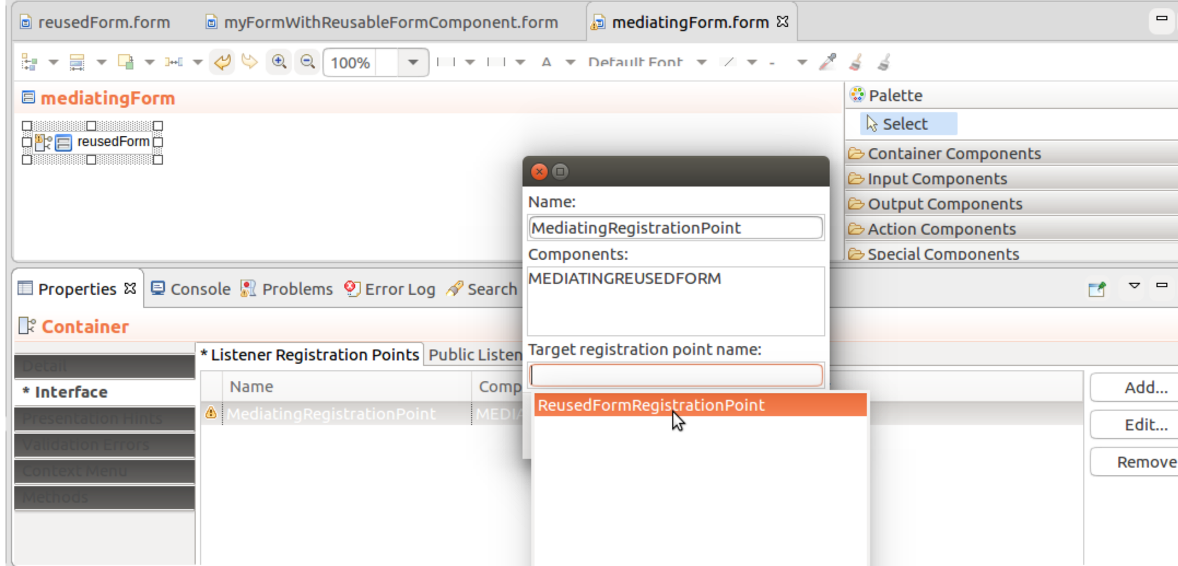
      ii. On the *Interface* tab, click the *Listener Registration Points* tab and click **Add**.

      iii. In the dialog box, define the registration point properties:

         • Name: name of the registration point

         • Components: ID of the Reusable Form component

- Target registration point name: registration point of a child Reusable Forms (use autocompletion)
  It is this property that connects the form to the Registration Point in the child Reusable Form.



3. Create the form that will handle the mediated event: insert the Reusable Form component that references the respective mediating form and create a private listener on the component.

2.7.3.5.3 Sending Events to a Reused Form

If a Reused Form needs to handle an event from other form components, you need to insert a public listener into the Reusable Form component that will listen for the event.

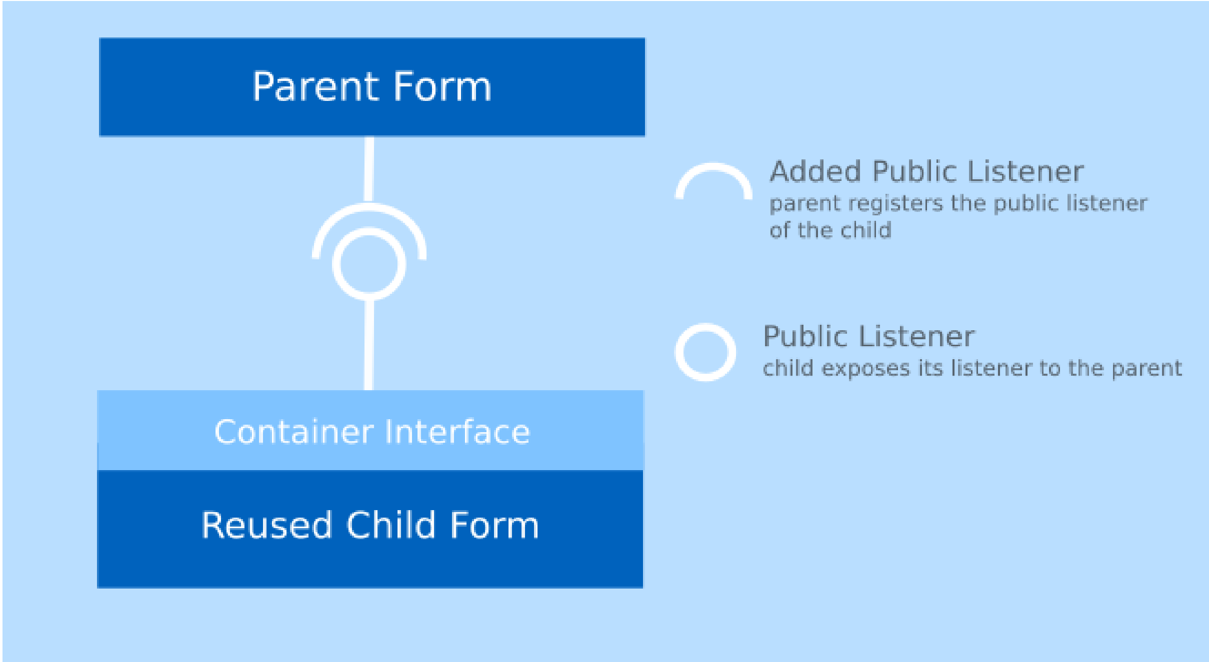


Figure 2.74 Public listener and its counterpart: added public listener on the parent

Note that if you need to send an event via multiple Reusable Form components, you need to expose the public listeners using delegated listeners

To define such event handling, do the following:

1. Create the form you want to use as the Reusable Form (this form will listen to an event on its parent form):
   (a) Create the form.
   (b) Insert the Container component as the root component of the form.
   (c) Define a public listener on the Container component:
      i. On the Interface tab of the Container properties, select the *Public Listeners* tab.
      ii. Click **Add** next to the New Listeners section.
      iii. Define its properties: make sure to select the correct listener type.

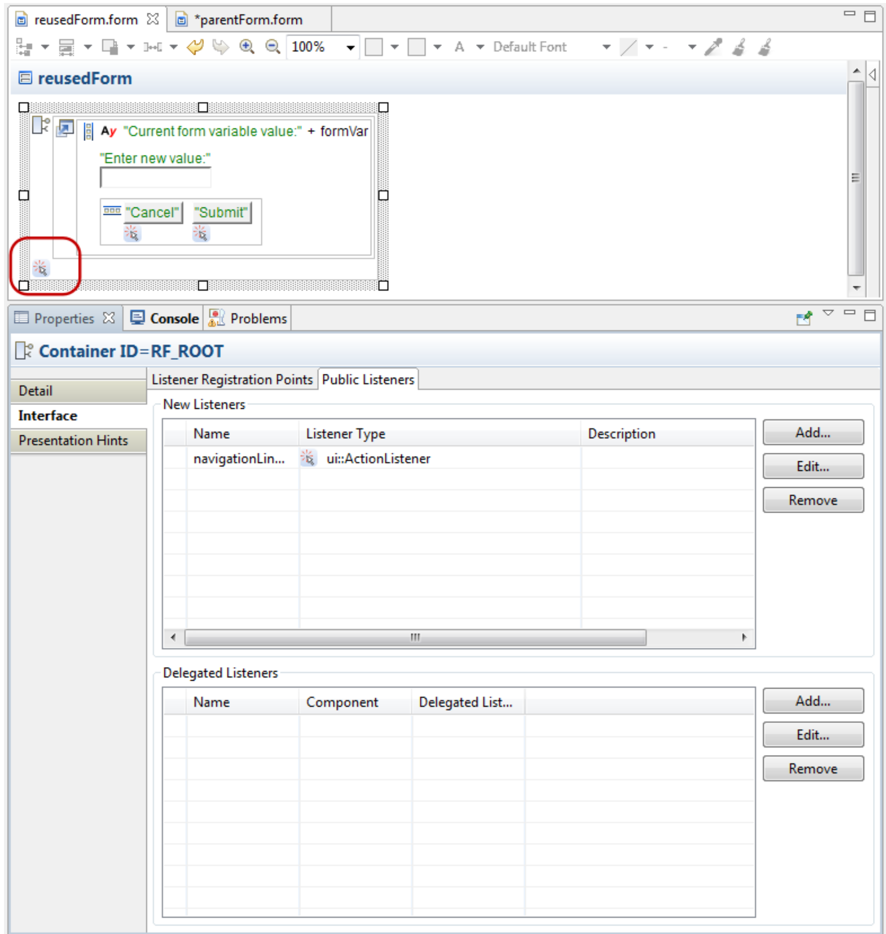
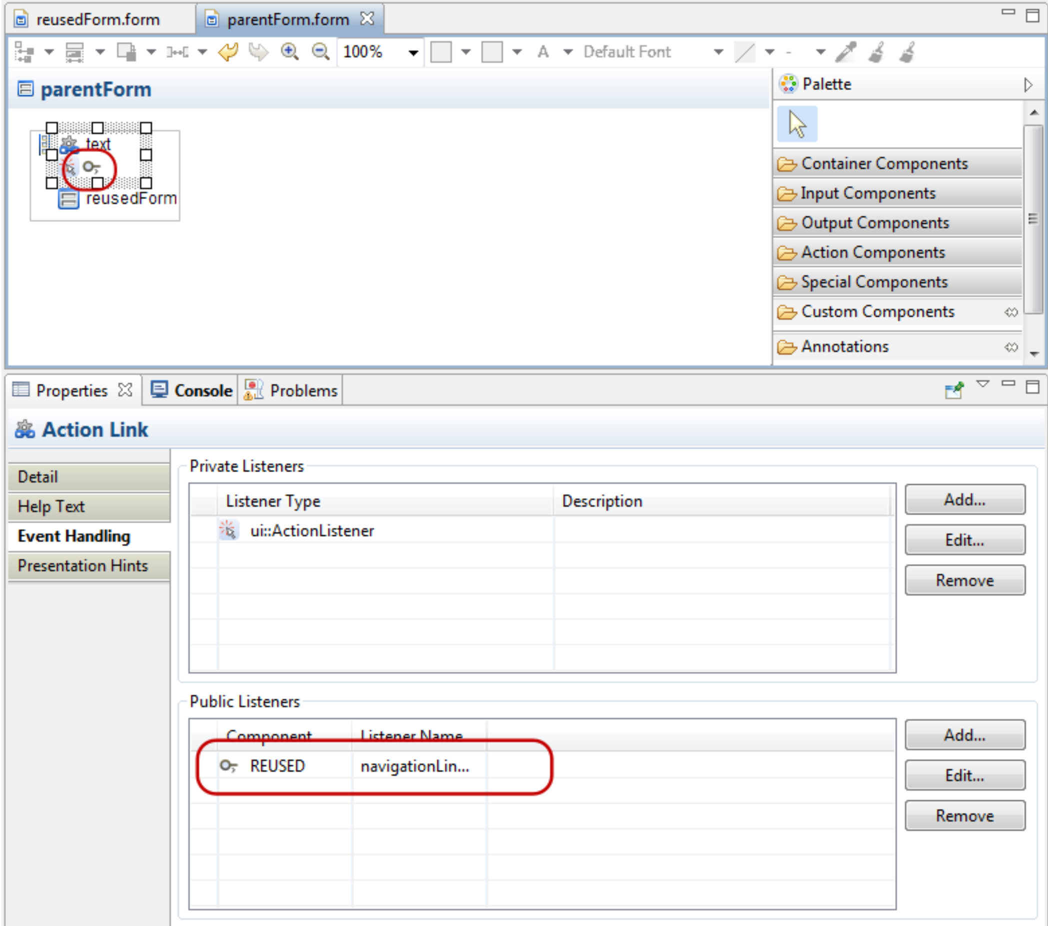


Figure 2.75 Container with the public listener navigationLinkClick

2. Open or create the form that will use the created form in the Reusable Form component:
   (a) Insert the Reusable Form component available under the Special Components and define its properties: Make sure to define its ID and the referenced form.
   (b) Create the component that will throw the event you will process in the reused form.
   (c) Register the listener on the component:
      i. Display its Event Handling tab in the Properties view.
      ii. Click **Add** in the Public Listeners section and define the properties of the public listener:
         - Component: the Reusable Form component that should process the event
         - Listener name: name of the listener you defined on the form in the previous step (use auto-completion)

If applicable, consider setting the *Immediate* property on the input component to `true`.



2.7.3.5.3.1 Sending Events to a Reused Form across Multiple Nested Forms

To expose a listener of a reused form across multiple parent forms, define a delegated listener in the mediating forms: This will allow the child reusable form to listen for events that occur in other than the immediate form.

1. Create the form you want to use as the Reusable Form with the Container component with a public listener.
2. Delegate the public listener through the Container interface of other forms with Reusable Forms components:
   (a) Create or open the mediating form with the Container component.
   (b) Insert the Reusable Form into the Container. Make sure to define its properties including its ID and the referenced form.
   (c) Delegate the public listener of the Reusable Form:
      i. Select the Container component, and click the *Interface* tab in its Properties view.
      ii. Select the *Public Listeners* tab and click the Add button in the **Delegated Listeners** section.
      iii. Specify the delegated listener parameters:
         - Name: name of the delegated listener
         - Component: the Reusable Form ID
         - Listener name: name of the listener of the reusable form (the one you are mediating; use autocompletion)

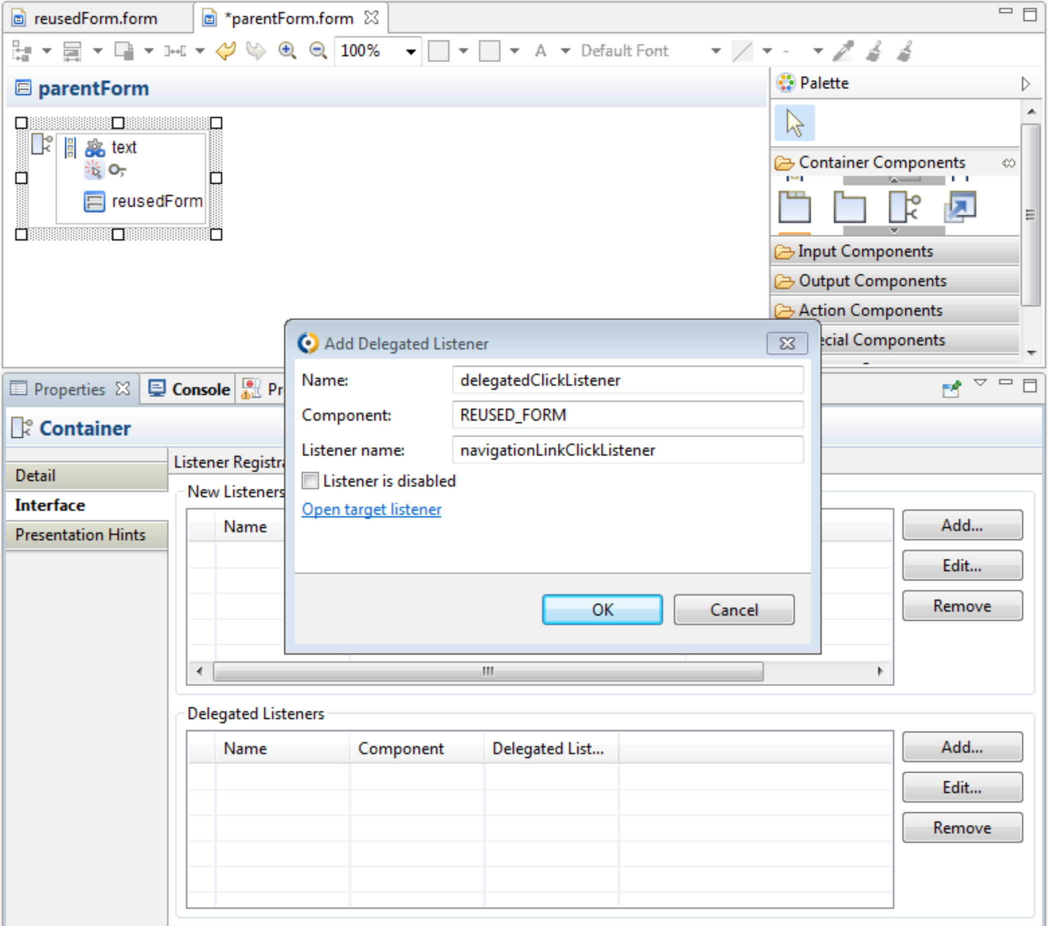


Figure 2.76 Defining properties of a delegated listener

3. To delegate across further Reusable Form components, repeat the previous step.
4. Open or create the form that will use the created form in the Reusable Form component and create a public listener on the Reused Form.

2.7.3.5.4 Broadcasting an Event

To broadcast an event across the entire form, throw an ApplicationEvent: an ApplicationEvent is thrown as part the event-processing lifecycle based on a listener definition. You can then define ApplicationEventListeners on the form to catch and process the event.

To define an application event that will be thrown by the event-processing cycle, do the following:

1. Create a listener on the component that should give rise to the event.
2. Open the properties of the listener (on the Event Handling tab of the Properties view, double-click the listener).
3. In the Edit Listener window, go to the Actions tab.
4. In the lower part of the tab, check the Fire application event checkbox.
5. In the area below, enter the expression that creates the application event.

```
new ApplicationEvent(eventName -> "idCreateRequest", payload -> idInfo)
```

You can now define the ApplicationEventListeners for the event.

2.7.3.6 Modifying Presentation of Components

To modify presentation properties of a form component, such as, its size, position, alignment, or possibly properties of a custom form component, use *presentation hints*. In the Application User Interface, hints translate into classes of the element.

You can use the hints defined in the Standard Library or you can define your custom hints.

> **Important:** Any hint assigned to a component is inherited by its child components. Note that some hints are defined implicitly and are not visible in the Form editor; for example, layout components have their width set to 100% by default and their children therefore have the width set to 100% as well. To ignore inherited hint values, override the hint value on the child component with another value or the `null` value if you want to erase the value.

2.7.3.6.1 Standard Library Hints

The ui module comes with a set of default presentation hints stored in the in the `ui::ui.hints file in the Standard Library`.

2.7.3.6.1.1 Assigning Hints From the Standard Library

To assign a form component a hint from the Standard Library, do the following:

1. In Form editor, select the form component.
2. In its Properties view, select the Presentation Hints tab.
3. Display the Hint Table tab.
4. Click the Add button to create a new hint or select an existing hint in the table and click Edit.
5. In the Edit dialog window, define the hint properties:
   (a) In the Hint name text field, select a predefined hint in the drop-down menu. You can also enter a hint name manually.
   (b) Either select a predefined hint value in the Predefined option drop-down or define its value in the Expression text area.

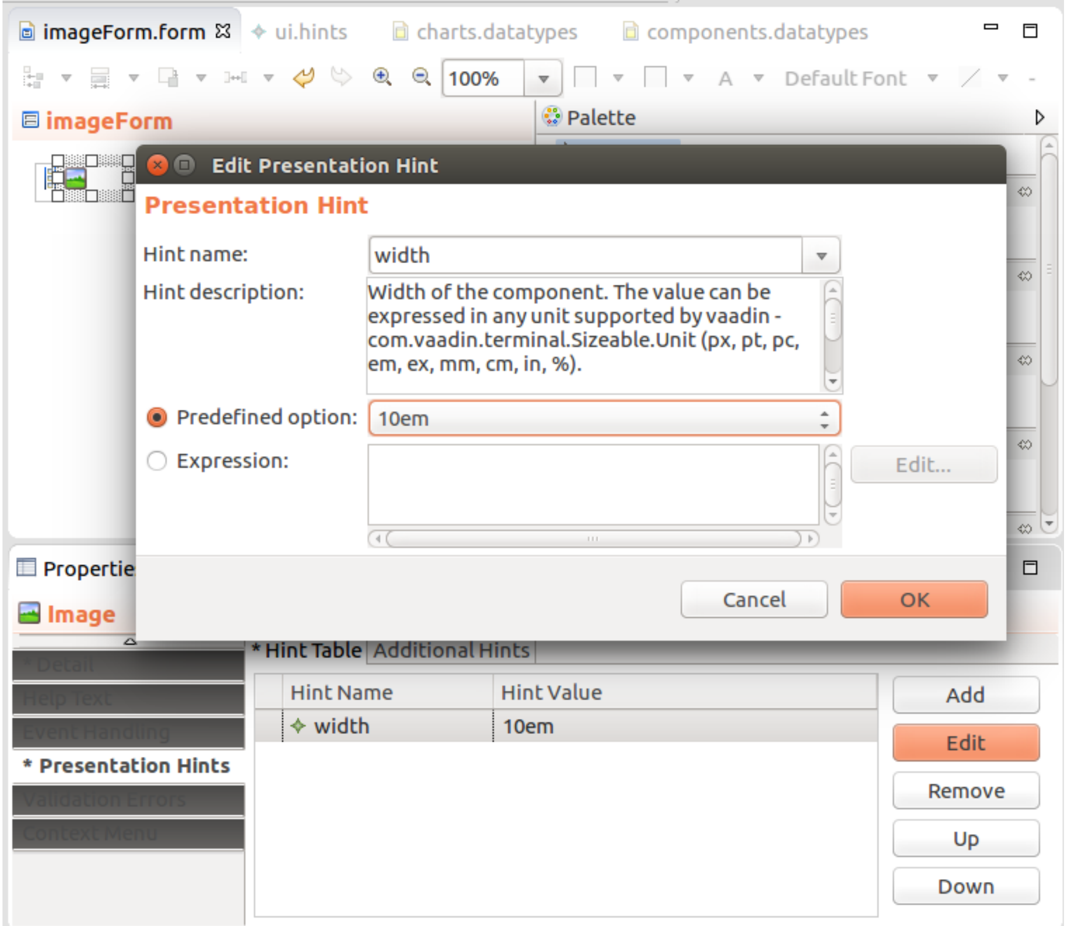


Figure 2.77 Editing a presentation hint

2.7.3.6.2 Custom Hints

Hints can be defined either directly on a form component or in a hints definition file. The hints defined in a hints definition file are available in the entire parent module.

Make sure that the component implementation accepts and processes the hint. Otherwise, the hint is ignored.

To define hints that can be used by all forms in the module, do the following:

1. Create a hint definition file (go to File New Hint Definition, and select the parent module and provide the definition file name).
2. In the GO-BPMN Explorer, double-click the hint definition file.
3. In the Hint Editor, click the **Add** button.
4. On the right, define the hint properties:
   - Name: hint name displayed in the Hint name drop-down menu.
   - Component: comma-separated list of form components that can use the hint
   - Hint options: value options available for the hint in the Predefined option (the label holds the displayed name of the option and the expression holds the value it is translated to when rendered)

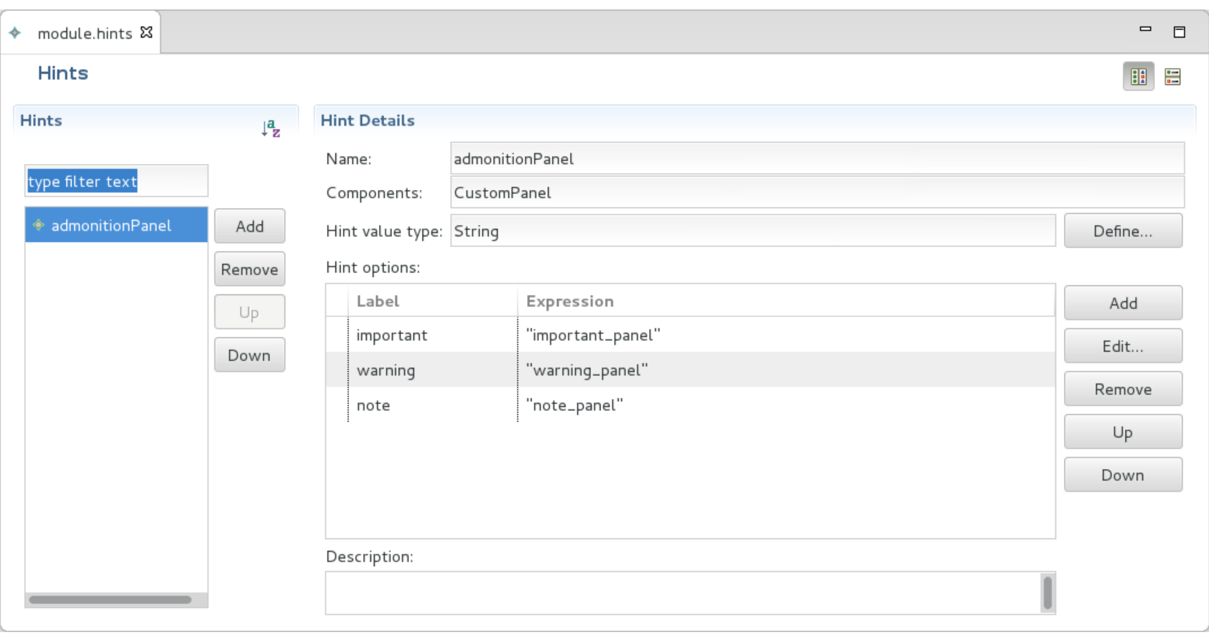
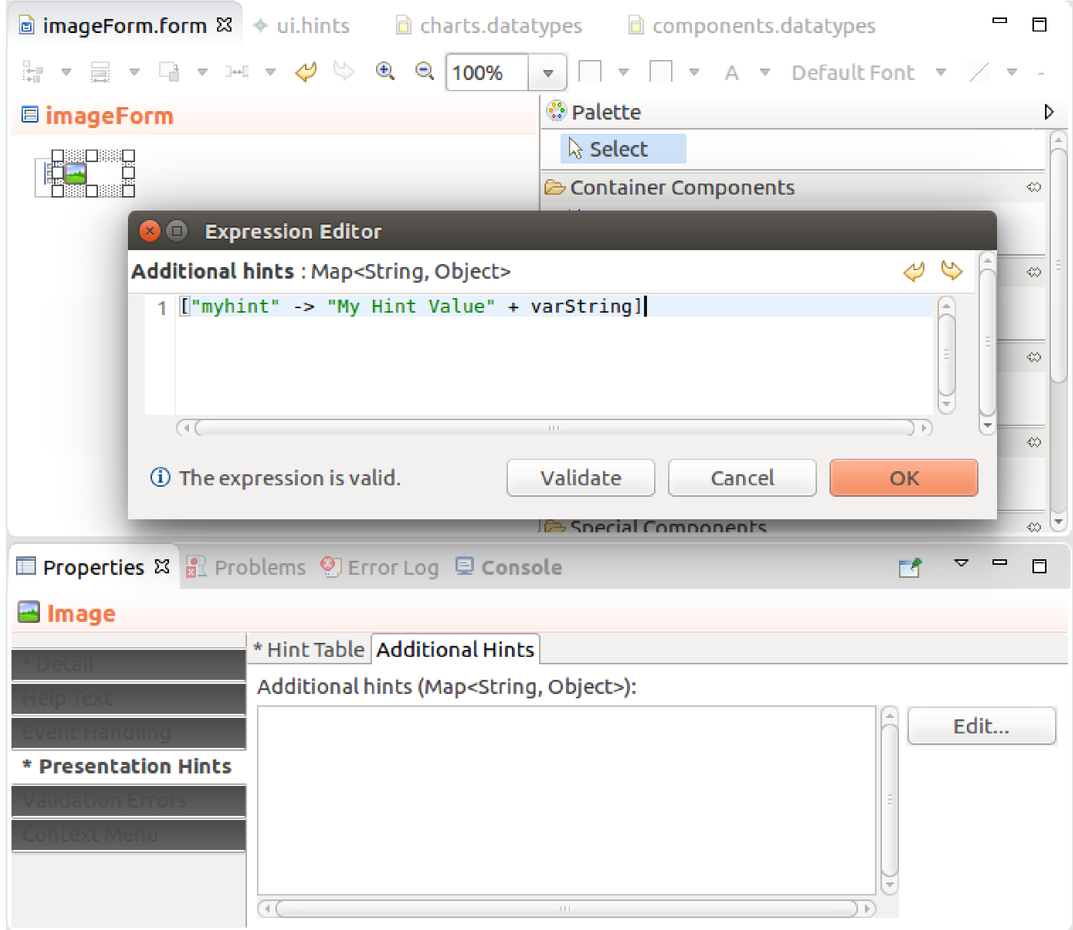


Figure 2.78 Hints editor with a hint definition

2.7.3.6.2.1 Assigning Custom Hints

To assign a form component a custom hint or a standard-library hint, do the following:

1. In the form editor, select the form component.
2. In its Properties view, select the Presentation Hints tab.
3. On the Additional Hints tab click Edit.
4. Define the hint name and values as a map of a String and Object (for example, `["myhint" -> "My Hint Value" + varString]`).



2.7.3.6.3 Using Hints

2.7.3.6.3.1 Aligning Form Components

To define alignment of child components in the Vertical, Horizontal, and Grid Layout, and table Column, use the `align hint`. Note that the *top*, *middle*, and *bottom* align settings are simply ignored.

On a table Column, the alignment is applied also on the column header. If necessary, you can override the setting with the *header-align* hint.

2.7.3.6.3.2 Resizing Form Components

Component size is defined by the height and width component hints. Where applicable, the size hints are defined implicitly. However, you can be override them with an explicit hint definition.

Hint values can be defined in the following units:

- font-relative units: size definition relative to the font size
  - em: size in factors of the letter `m` size
  - ex: size in factors of the letter `x` size
- absolute size: component size in pixels (px), picas (pc), points (pt), mm, cm, or inches (in)
- relative size to the parent: component size in relation to the size of the immediate parent

  You can use the predefined values:

  - `fillparent`: identical to `100%`,
  - `wrapcontent`: component size is set to the sum of sizes of children components

- `expand`: Sets the expand ratio for a component. If there is only one component with an expand ratio, the component will be expanded to the maximum possible size within its parent. This hint can only be applied to a table Column or the direct child of a Vertical or Horizontal Layout. Form Layout is unsupported.

2.7.3.6.3.3 Default Size Hint Values

Form components define implicitly presentation hints with default values. These values are applied unless you define another value for the hint.

Note that the default values might be in conflict. As a result, the rendered form might not meet your requirements or render at all. For example, consider a table in a vertical layout:

- Vertical layout has the default width hint defined to `wrapcontent` so it checks the size of its child components and sets its width to the maximum child width.
- A table has the default width hint value defined as `fillparent` so it uses the parents' width.

Hence there is no setting for the table or the layout and you need to explicitly set to one of the widths to a required value.

Default Widths and Heights

Component	Default Width	Default Height
Container Components	fillparent	wrapcontent
Grid Layout	fillparent	wrapcontent
Form Layout	fillparent	wrapcontent
Text Box	wrapcontent	wrapcontent
Text Area	wrapcontent	wrapcontent
Check Box	wrapcontent	wrapcontent
Combo Box	wrapcontent	wrapcontent
Lazy-Loading Combo Box	wrapcontent	wrapcontent
Form Layout	fillparent	wrapcontent
Single-Select List	wrapcontent	wrapcontent
Multi-Select List	wrapcontent	wrapcontent
Check-Box List	wrapcontent	wrapcontent
Radio-Button List	wrapcontent	wrapcontent
File Upload	wrapcontent	wrapcontent
Output Text	fillparent	wrapcontent
Table	wrapcontent	wrapcontent
Calendar	fillparent	fillparent
Action Button	wrapcontent	wrapcontent
Browser Frame	fillparent	wrapcontent
Button	wrapcontent	wrapcontent
Cartesian Chart	fillparent	400px
Gauge Chart	fillparent	400px
Pie Chart	fillparent	400px
Polar Chart	fillparent	400px
Conditional	fillparent	wrapcontent
Dashboard	fillparent	fillparent

Component	Default Width	Default Height
Dashboard Widget	0, 0, 200, 300	overrides DashBoard
File Download	wrapcontent	wrapcontent
Geolocator	invisible	invisible
Horizontal Layout	wrapcontent	wrapcontent
Image	wrapcontent	wrapcontent
Lazy Table	wrapcontent	wrapcontent
Tree Table	wrapcontent	wrapcontent
Map Display	fillparent	400px
Message	fillparent	wrapcontent
Navigation Link	wrapcontent	wrapcontent
Panel	fillparent	wrapcontent
Popup	fillparent	wrapcontent
Repeater	wrapcontent	wrapcontent
Tabbed Layout	fillparent	wrapcontent
Table Column	counted by the table	counted by the table
Tree	wrapcontent	wrapcontent
Vertical Layout	fillparent	wrapcontent
View Model	fillparent	wrapcontent

2.7.3.6.3.4 Defining Common Presentation Properties

With the *html-class* hint, you can define the presentation of a component, such as, border rendering, highlighting, emphasis, overflow, disabling of text, etc. Refer to the `hints documentation in the Standard Library documentation`.

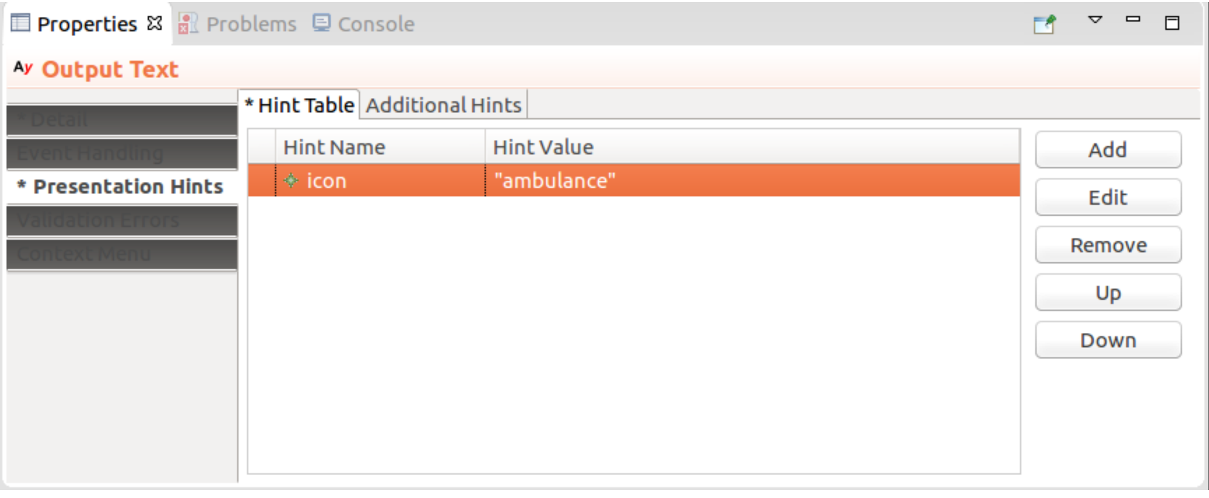
2.7.3.6.3.5 Adding a CSS Class to a Form Component

To add a class attribute to the rendered component, use the `html-class` hint.

Note that the class must be defined in the Application User Interface. For instructions on how to add new style sheets, refer to `Software Development Guide`.

2.7.3.6.3.6 Adding a Font Icon to a Form Component

To add an icon from the Awesome font to your component class, assign the `icon` presentation hint to your component and insert the name of the icon as its value.



2.7.3.6.3.7 Setting the Maximum Text Size on a TextBox and a TextArea

The TextBox and TextArea components can define the max-text-size hint, which defines the maximum length of the text the user can enter. If the component is bound to a shared record field, the size must take into account the size of the underlying database column, that is the column to which the field is mapped.

You can hence generate the hint for TextBox and TextArea components that are bound to shared record fields. Note that the hint generation is applied on all open GO-BPMN projects of your workspace. Close any projects you want to exclude.

To generate or adjust the max-size-hint for all TextBox and TextArea components in all open GO-BPMN projects, do the following:

1. Go to **Project** > **Update max-text-size hint**.
2. In the Update max-text-size hints dialog, select the applicable choices:
   - Add missing max-text-size hint to generate the hint on any TextBox and TextArea components bound to a shared record field if the hint is missing
   - Update existing max-text-size hints to overwrite any defined values of the hint with the database column sizes

     Important: If a max-text-size hint contains a non-integer value, the hint is invalid.
   - Ignore if value of max-text-size hint is smaller than DB length to prevent overwriting of the field length that are smaller than the length of the underlying database column size.

2.7.3.7 Creating Mobile Forms

LSPS provides support for creating mobile forms that can be used in mobile LSPS applications.

> **Important:** A mobile LSPS application is by no means intended to substitute the desktop application and should be considered an extension of the desktop application.
>
> When creating mobile LSPS applications, make sure to follow the common guidelines for creating your application: avoid complex forms; keep the forms intuitive.

The specific support for mobile forms in LSPS includes the following:

- mobile presentation hint for the vertical and horizontal layout component

2.7.3.7.1 Guidelines

- Use horizontal layout with the mobile hint only to hold a set of buttons.
- Wrap more complex forms either in the vertical layout component with the mobile hint or in the Tabbed container component. Every Tab component can then contain a part of the form. In the Tab component, use preferably icons instead since the used font is rather small.
- Do not nest a layout component with the mobile hint into another layout component with the mobile hint. The exception is a horizontal layout component with buttons.
- Adapt your forms to different screen sizes and resolutions: use Conditional components with different "versions" of the form.

  To detect the screen size on runtime do the following:
  - Create the custom `getBrowserWindowWidth()` that returns an Integer. The function is implemented as the `getBrowserWindowWidth()` method in `com.whitestein.lsps.app.vaadin_touchkit.touchkit.MobileUtils` and returns the width of the displayed page in DP(density independent pixel).
  - Define how to handle the ApplicationEvent with the ID `internal_windowresize`. The event is produced when the width of the browser window changes and therefore also when the device is rotated. It bears the window size in DP as its payload (held by the `ui::Dimension` record instance).

> **Important:** If the user provides invalid values in the form and the window width changes, the invalid values are not transferred to the refreshed form. Consider handling such situations in your form.

2.7.4 Form Components

When you design the component of a form, you insert various types of form components to the form content. The components constitute a tree with each component defining

- listeners for the type of events that can occur on the component and
- properties required to render the component, such as, data to display, presentation styles, size, etc.

The following are the properties that are common to all components:

- **ID**: ID of the component on design time (Only capital letters, digits, underscores and dollar signs are allowed.)

  ID can be used by other form components and listeners in the form.
- **Modeling ID**: ID of the component used on runtime

  Modeling IDs are populated with a random unique value. You can change the value manually if required. If an error occurs in runtime, the server returns an error with the modeling IDs of the involved components. The feature is disabled by default. You can enable it with a parameter.
- **Visible**: visibility of the component and its child components

  If the property expression is `true` or `null`, the component is visible. If `false`, the component is not visible. The default value is `true`. Note that invisible components and their child components are not recalculated on refresh.

  Components with an expression in the *Visible* property are marked with the icon .

  > **Important:** The **Visible** property substitutes the **Show** property, which was deprecated and removed. However, if your forms still contain the **Show** property, it will be handled as expected. If both of the properties are defined, the Show property is ignored.
- **context menu**: menu displayed when the user right-clicks the component
- **component-specific properties** that define the behavior of the component (refer to sections on individual components)
- presentation hints define the presentation style of the component

Properties and events specific to components or the group they belong to, are described in the respective sections:

- Container Components
- Input Components
- Output Components
- Action Components
- Special Form Components
- Text Annotations and Associations
- Deprecated Components

2.7.4.1 Container Components

Container components nest components and define the form structure.

They can fire only InitEvents with the exception of the Dashboard Widget, which can fire a WidgetChangeEvent as well.

2.7.4.1.1 Vertical Layout

The Vertical Layout component ( ) is a container component that can hold multiple form components, which are arranged in a vertical manner.

Specific Vertical Layout properties:

- **Label**: object that is displayed as the title of the layout; label is rendered by the parent component; hence if the layout is the root component, Label is not rendered.

2.7.4.1.2 Horizontal Layout

The Horizontal Layout component ( ) is a container component that can hold multiple form components, which are arranged in a horizontal manner.

Specific Horizontal Layout properties:

- **Label**: object that is displayed as the title of the layout; label is rendered by the parent component; hence if the layout is the root component, Label is not rendered.

2.7.4.1.3 Form Layout

The Form Layout component ( ) is a container component that can hold multiple form components arranged in two columns. The child form components are rendered with their labels placed in the left column and any other component parts rendered in the right column.

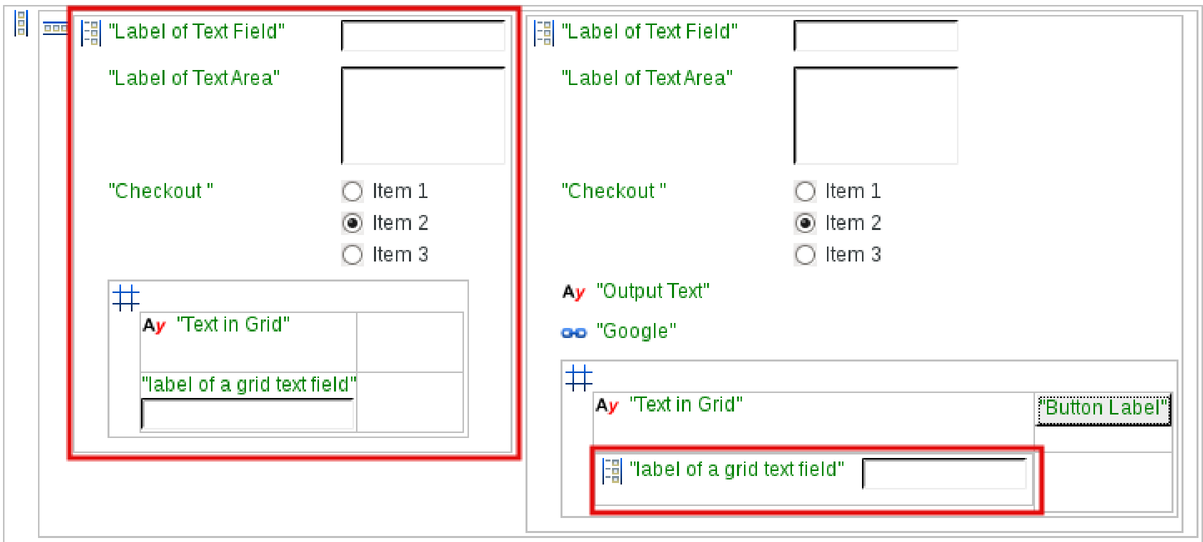


Figure 2.79 Form with multiple Form Layouts

2.7.4.1.4 Panel

The Panel component ( ) is a container component that serves for adding a frame and a caption to one, possibly a layout, component.

A scrollbar is rendered in panel child component automatically if applicable.

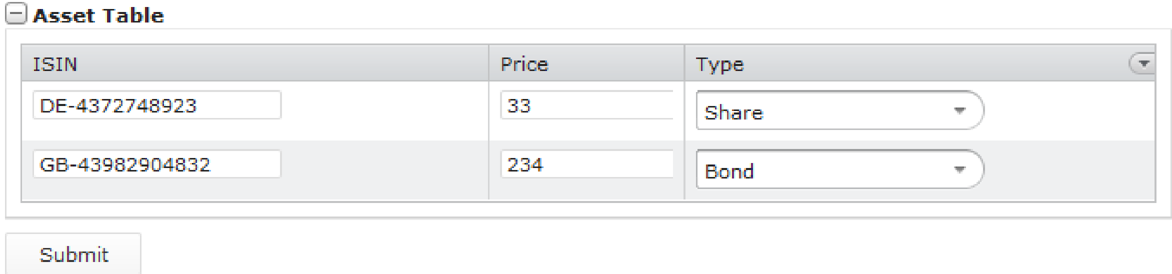


Figure 2.80 Asset Table nested in a collapsible panel (note the collapse box next to the table label)

The Panel component has the following properties:

- **Title**: label displayed on the panel The Title property supports HTML content.
- **Collapsed**: if set to true, the panel is rendered as collapsible

2.7.4.1.5 Grid Layout

The Grid Layout component ( ) is a container component that can hold multiple form component elements, which are arranged in a grid manner, that is in a table-like way. You can add rows and columns to the grid using the Add ( ) and Delete ( ) buttons on the grid component.

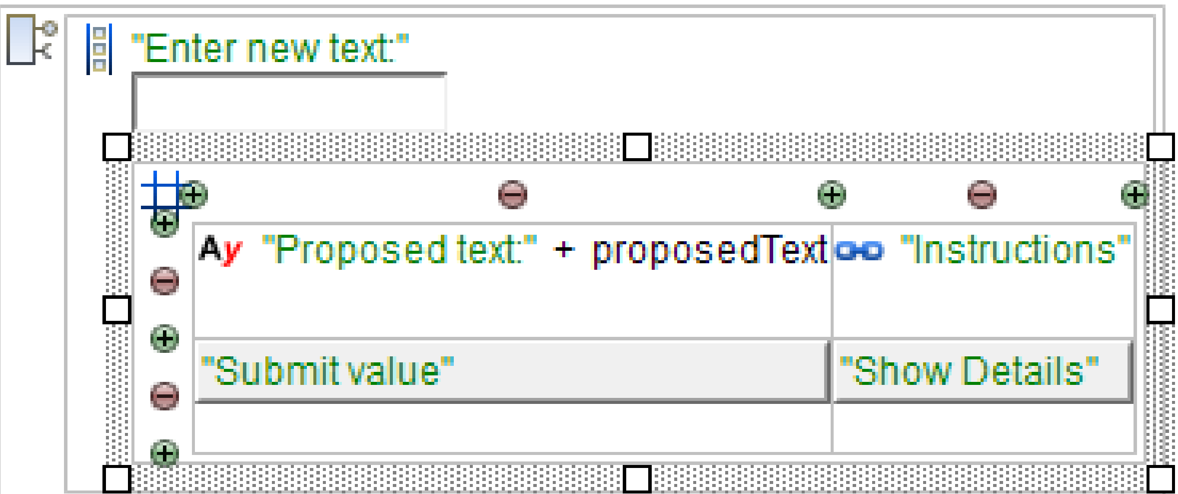
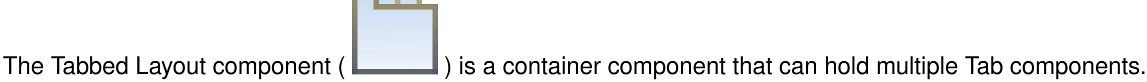


Figure 2.81 Grid Layout component

2.7.4.1.6 Tabbed Layout

The Tabbed Layout component () is a container component that can hold multiple Tab components.

2.7.4.1.6.1 Tab

The Tab component ( ) is a component rendered as a tab. It that can contain one form component and its parent component must be the tabbed layout component.

A tab content and its children are initialized along with their parent. However, a tab produces an init event and is refreshed when displayed.

The Tab component has the following properties:

- **Title**: tab label
- **Visible**: boolean expression defining the tab visibility

  If the visible expression evaluates to true or undefined on init or refresh, the tab is displayed.

Note that you can add, remove, and select a Tab component programatically from listener handle expression using the addTab(), removeTab(), selectTab() functions.

2.7.4.1.7 Container

A Container component ( ) is a form component that allows you to define an interface for a form or a form component and its child components. The interface provides a mechanism for listening for events either on nested forms or on parent forms (for details refer to Container Interface).

2.7.4.1.8 Popup

The Popup component ( ) is rendered as a pop-up window with a maximize/minimize and close button in its caption. It can be modal: when a modal popup is displayed, you cannot work with the underlying form until it is closed.

Since Popups are included directly in the form, they are calculated when the form is initialized. For large popups with a lot of data, this can result in poor form performance since these popups are part of the form all the time even if you might not need them at all. In such a case, consider using a dynamic popup which is created only when requested.

For an example popup design and usage, refer to Pop-up with Save and Cancel Buttons.

2.7.4.1.8.1 Dynamic Popup

Dynamic popups are created at the moment they are requested. This prevents existence of potentially unnecessary Popups in the form hierarchy and saves you some headache over the performance of your form.

Typically create a dynamic popup from a Listener handle expression or as part of a reusable form:

1. Define the popup and its content (for example `new ui::Popup p; p.child := ...`)
2. Create the popup with the call `createAndShow()`.
3. Define the logic of the popup, possibly using `clear()` and `merge()` functions.
4. To remove the popup from the component tree, call the *hideAndDestroy()* function on the popup.

```
//This is a dynamic Popup (possibly created in a Reusable form or a component listener handle
def ui::Popup p := new ui::Popup (
  visible -> { ->
    true
  },
  listeners -> {
    new PopupCloseRequestListener(
      handle -> { e ->
        hideAndDestroy(p)
      }
    )
  }
);
```

```
//p.visible := { -> true};
def TextBox tx := new TextBox(binding -> &varString);


p.child := new VerticalLayout(
      children -> [ tx ]
);
createAndShow(p);
```

More dynamic functions are available from the *dynamics.funcs* file in the *ui* module, for example, to search for components higher or lower in the hierarchy, such as, `findTopmostComponents()`, `findTopmostContainers()`, `getChildren()`, etc.

2.7.4.1.8.2 Closing a Popup

You can close a popup by setting its visibility to `false` and refreshing it from any listener so it disappears or with the *hideAndDestroy()* function call on the popup.

If you want to close your popup with an X button in its caption, define the behavior in the popups *PopupCloseRequestListener*.

```
def ui::Popup p :=
  new ui::Popup (
    visible -> { -> true },
    listeners -> { new PopupCloseRequestListener(
      handle -> { e ->  hideAndDestroy(p) }
      )
    }
  );
```

2.7.4.1.9 Dashboard

The Dashboard component is a container component that can hold multiple widget components.

When working with a rendered dashboard in the client application, use the plus (+) button on the dashboard to display a widget. Visualized widgets can be positioned anywhere within the dashboard, resized, and visualized as necessary. Only one instance of the given widget can be visualized on the dashboard.

Note that dashboard has a toolbar, which can contain any form element apart from a widget.

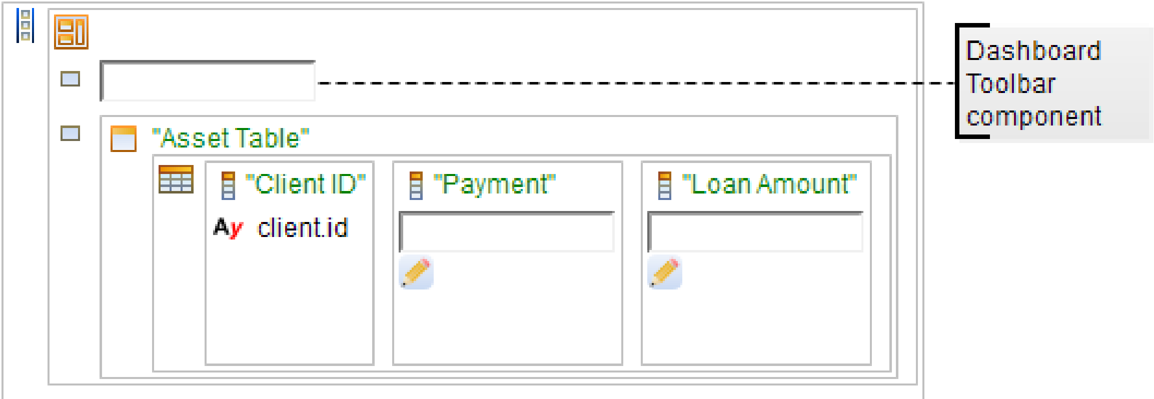


Figure 2.82 Form with a Dashboard component

Note: By default, dashboard size adapts to the requirements of the visualized widgets with the minimum size 650x490 px. To alter the size, use presentation hints on the dashboard (refer to the Standard Library document).

2.7.4.1.9.1 Dashboard Widget

The Dashboard Widget component is a form component that is rendered as a widget on a dashboard: Therefore it must have the Dashboard component as its parent. A Dashboard Widget can hold one form component.

It produces events of the following types:

- *InitEvent* when the component is initialized or displayed if previously hidden
- *WidgetChangeEvent* when the user adds a widget to the dashboard, resizes, moves, or hides a widget

  The event contains information on the widget properties in the WidgetConfiguration field.

By default, when a dashboard is visualized, it remains empty. The user can then add the available widgets to the dashboard using the plus (+) button on the dashboard. To visualize a widget immediately when a dashboard is displayed, define the widget's Configuration property: the property is defined as its WidgetConfiguration property.

For example, if the Configuration property is set to `new WidgetConfiguration(visible -> true, width -> 300, height -> 300, top -> 300, left -> 350)`. The widget will be visible when the dashboard is displayed; its size will be 300x300 pixels; the coordinates of the upper left corner will be 300:350 (its upper left corner will be positioned 300 pixels below the top of the dashboard and 350 pixels to the right).

The Dashboard Widget component has the following properties:

- **Title**: widget title
- **WidgetID**: ID used in the WidgetChangeEvent
- **Configuration**: WidgetConfiguration object defining the visualization properties of the widget when the dashboard is visualized

  WidgetConfiguration defines the following:

  - `visibility`: boolean value that defines whether the widget is displayed by default when the dashboard is displayed

    In the default Application User Interface, previously closed or invisible widgets can be visualized by clicking them under the + plus button of the dashboard.
  - `width`: default width of the widget
  - `height`: default height of the widget
  - `top`: distance of the top widget border from the dashboard top border
  - `left`: distance of the left widget border from the dashboard left border
  - `zIndex`: vertical stacking order of the widget

    When the widget produces a WidgetChangeEvent, that is whenever it is clicked, moved, or resized, the zIndex is raised so that the widget is on top of any other widgets in the dashboard.
  - `maximized`: boolean value that defines if the widget is maximized
  - `minimized`: boolean value that defines if the widget is minimized

    In the default Application User Interface, minimized widgets are available at the bottom of the dashboard.

Note: Note that widgets keep its last size value separate from its maximized and minimized status↩
: if you set a widget to a particular size, then maximize it, and minimize it, when restored it will be maximized. If you then double-click the widget caption, it will be restored to the original size. If you want to move a widget to the foreground, click the widget.

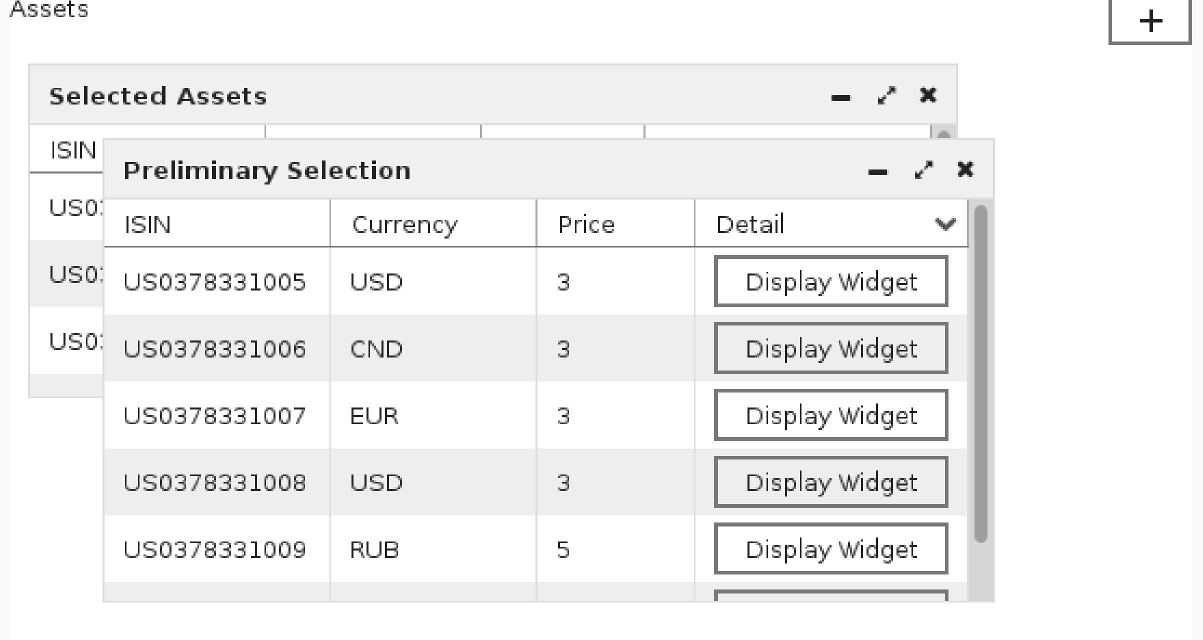


Figure 2.83 Dashboard with Widgets

2.7.4.2 Input Components

Input components serve to acquire input from the user: When the user provides the input, the component produces a value change event, which can be caught by a value change listener. The listener and the component define how to process the event during the next response-request cycle.

2.7.4.2.1 Text Box

The Text Box component ( ) is rendered as single-line text field to allow the user to provide text input.

It produces events of the following types:

- *InitEvent* when the component is initialized or displayed if previously hidden
- *ValueChangeEvent* when the component has changed content and loses focus (to process it also on focus loss; set the Immediate property to true)
- *ActionEvent* when the user presses ENTER while the Text Box is focused
- *AsynchronousValueChangeEvent* whenever the user inserts or removes a character

  Note that the binding of the Text Box does not have to change immediately due to possible validation; hence binding of the Text Box might appear to be out-of-date.

The Text Box component has the following properties:

- **Label**: visible text label
- **Placeholder**: input prompt (text displayed in the text box if the value of the Binding reference is null)
- **Required**: compulsoriness of the component; If true, a mark indicating that the value is required is rendered in the component.

  > **Important:** The property **does not provide any validation** whether the field contains a value and the user will be able to submit a null value unless a additional validation mechanism is defined. Such a validation can be implemented on a listener, possibly as a validation expression.

- **Binding**: reference to a slot that holds the text value (for example, a form variable or global variable)

  If a Text Box binds to a Date type, it is rendered as the Date Picker: for the Date Picker you can define the format of the date the Date Picker will accept in the **Format** property. The available formats are defined in the `additional formats` hint.

- **Format**: required input format, such as, date, integer, etc.

  The format can be specified as defined in java.util.Pattern, java.text.SimpleDateFormat, and java.text.DecimalFormat

  If the value in the field is not of the defined format pattern, a validation mark is displayed next to the Text Box. Note that the property does not provide any validation. The validation must be implemented on the respective listener possibly as a validation expression.

- **Read-only**: editability of the text

  If true, the text renders as grayed out and cannot be edited.

- **Immediate**: setting of the Immediate mode

  If true, the immediate mode is active: the value change event triggers request-response cycle on focus change.

- **Help text**: tooltip text

2.7.4.2.1.1 Defining Suffix on a Text Field

To add a suffix to a Text Field component, such as, `PCS`, define on the Text Field the *suffix* Hints with the value of the suffix in a String. You can do so on the *Presentation Hints* tab of the Properties view.

2.7.4.2.2 Text Area

The Text Area component ( ) renders as multi-line text input area to allow the user to provide longer text input.

It produces events of the following types:

- *InitEvent* when the component is initialized or displayed if previously hidden
- *ValueChangeEvent* when the component contains changed content and loses focus (to process it also on focus loss; set the Immediate property to true)
- *AsynchronousValueChangeEvent* whenever the user inserts or removes a character

  Note that the binding of the Text Area does not have to change immediately due to possible validation; hence binding of the Text Box might appear out-of-date.

The Text Area component has the following properties:

- **Label**: visible text label
- **Required**: compulsoriness of the component; If true, a mark indicating that the value is required is rendered in the component.

  > **Important:** The property **does not provide any validation** whether the field contains a value and the user will be able to submit a null value unless a additional validation mechanism is defined. Such a validation can be implemented on a listener, possibly as a validation expression.

- **Binding**: reference to a slot that holds the text value (for example, a form variable or global variable)
- **Read-only**: editability of the text
- **Immediate**: setting of the Immediate mode

  If true, the immediate mode is active: the value change event triggers request-response cycle on focus change.

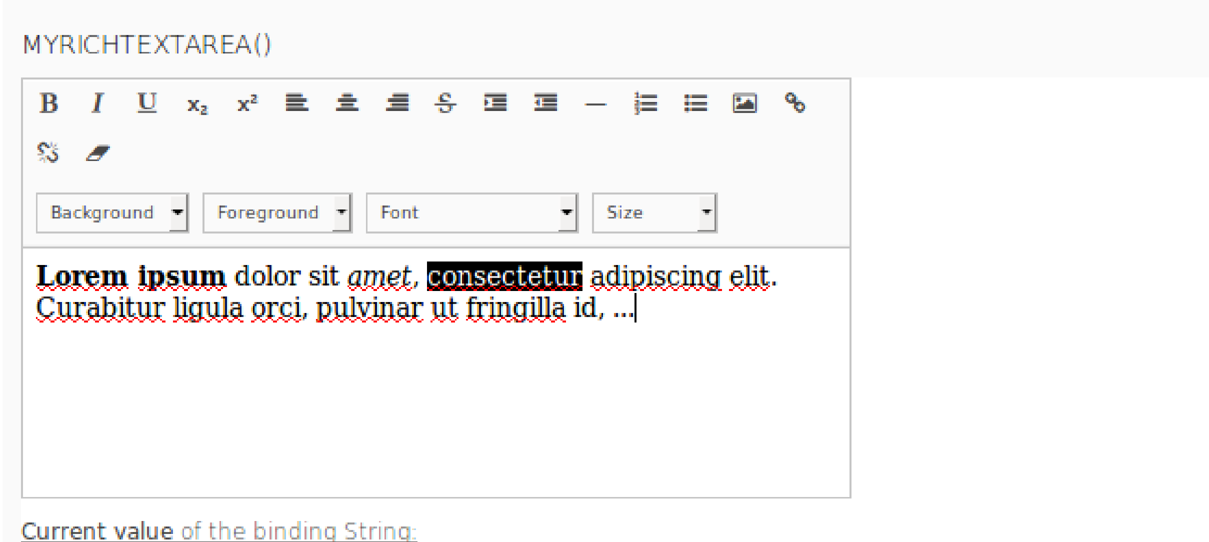
- **Placeholder**: input prompt (text displayed in the text area if the value of the Binding reference is null)

  If true, the text area is grayed out and cannot be edited.

- **Is Rich Text**: if true, the Text Area is rendered with a formatting toolbar

  The input text is translated into html.

- **Help text** (on the Help Text tab): tooltip text



Lorem ipsum dolor sit <i>amet</i>, consectetur adipiscing elit. Curabitur ligula orci, pulvinar ut fringilla id, ...

2.7.4.2.3 Check Box

The Check Box component ( ) renders as a check box with a label and allows the user to provide a Boolean value.

It produces events of the following types:

- *InitEvent* when the component is initialized or displayed if previously hidden
- *ValueChangeEvent* when the component was selected or unselected

The Check Box component has the following properties:

- **Label**: visible label displayed next to the checkbox
- **Required**: The property is not applicable for Check Box components.
- **Binding**: reference to a slot that holds the Boolean value (for example, a form variable or global variable)
- **Read-only**: editability of the check box

  If true, the check box renders as grayed out and cannot be edited.
- **Immediate**: setting of the Immediate mode

  To process the ValueChangeEvent at the moment the user clicks the Checkbox, set the Immediate property to 'true'.
- **Help text**: tooltip text

2.7.4.2.4 Combo Box

The Combo Box component ( ) renders as a single-line text field with a drop-down list of options with the possibility to insert a non-listed option.

It produces events of the following types:

- *InitEvent* when the component is initialized or displayed if previously hidden
- *ValueChangeEvent* when the user selects an option

The Combo Box component has the following properties:

- **Label**: visible label above the combo box
- **Placeholder**: input prompt (text displayed in the text field if the value of the Binding reference is null)
- **Required**: compulsoriness of the component; If true, a mark indicating that the value is required is rendered in the component.

  Important: The property **does not provide any validation** whether the field contains a value and the user will be able to submit a null value unless a additional validation mechanism is defined. Such a validation can be implemented on a listener, possibly as a validation expression.

- **Binding**: reference to a slot that holds the selected option value (for example, a form variable or global variable)
- **Read-only**: editability of the combo box

  If true, the combo box renders as grayed out and no option can be displayed or selected.

- **Immediate**: setting of the Immediate mode

  To process the ValueChangeEvent at the moment the user select an option, set the Immediate property to 'true'.

- **Options**: list of options

  The list items are displayed in the drop-down. Make sure the defined expression resolves to an object of the type List<ui::Option>.

  Note that the Option data type has the value and label field so that you can define the label displayed in the drop-down list that represents the given value.

```
collect(
  literals(type(AssetType)),
  { e ->
    new ui::Option(
       value -> e,
       label -> literalToName(e)
    )
  }
)
```

- **Create new option**: closure that is called if the user enters a custom value

  When the user enters a custom value in the text field of the combo box, the closure is called with the value as its parameter. The closure returns an object that is set to the binding value.

  Important: A event that creates a new option is created only when the user presses ENTER after they input the new value.

- **Help text**: tooltip text

2.7.4.2.5 Lazy-Loading Combo Box

The *Lazy-Loading Combo Box* component renders as a single-line text field with a drop-down option list. However, the options in the drop-down are paged and queried as the user enters parts of the option so the component is useful if you need to select one item from many options.

It produces events of the following types:

- *InitEvent* when the component is initialized or displayed if previously hidden
- *ValueChangeEvent* when the user selects an option

When creating a Lazy-Loading Combo Box component, define the following properties:

- **Label**: visible label
- **Required**: compulsoriness of the component; If true, a mark indicating that the value is required is rendered in the component.

  Important: The property **does not provide any validation** whether the field contains a value and the user will be able to submit a null value unless a additional validation mechanism is defined. Such a validation can be implemented on a listener, possibly as a validation expression.

- **Binding**: slot that holds the selected option value, such as a form or global variable
- **Read-only**: editability of the combo box

  If true, the combo box renders as grayed out and is disabled.

- **Immediate**: setting of the Immediate mode

  To process the ValueChangeEvent at the moment the user clicks an option, set the Immediate property to 'true'.

- **Options**: list of options

  The closure returns a List<Object>, where each list object represents an option. What is actually displayed as a label for the option is defined in the Formatter property.

  The closure has the following input parameters:

  – value in the text field (the value is a String and can serve to filter out the options in the drop-down list)

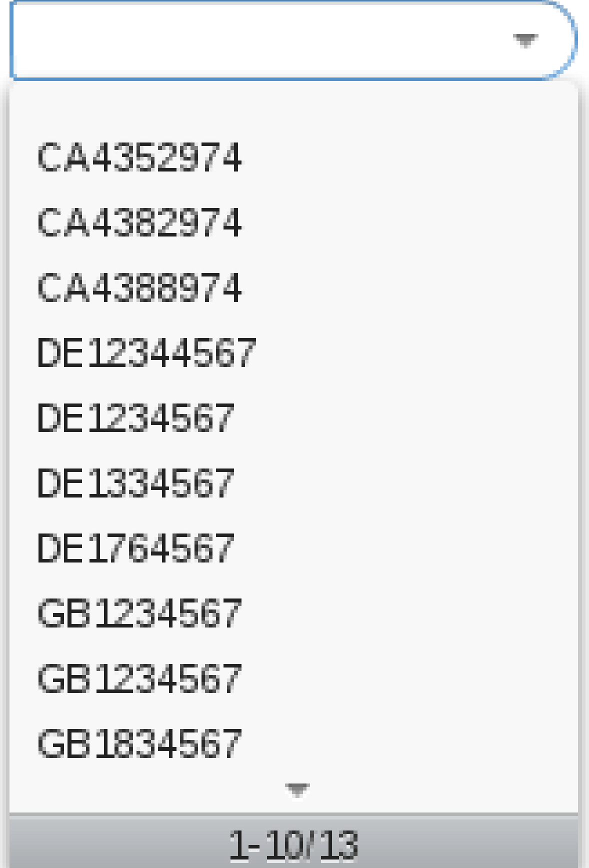


Figure 2.84 Lazy-Loading Combo Box with no value in the text field

  – index of the first option (options of items starting from this index can be displayed)
  – displayed option count (the number of options from the first option to be displayed)

- **Option count**: count of displayed options

  Make sure the defined closure returns an Integer value. Typically you want to use a count query.

```
{code -> getCurrencies_count(code, null, null)}
```

- **Formatter**: closure that returns a string that is displayed in the drop-down list

  The string represents the respective object from the Options.

```
{c:Currency -> c.currencyCode}
```

- **Create new option**: closure that is called when the user enters a custom value

  If the user enters a custom value in the combo text field of the combo box, the closure accepts the value as its parameter and returns it to the binding entity.

- **Placeholder**: input prompt (text displayed in the text field if the value of the Binding reference is null)
- **Visible**: whether the component is displayed (if not, the component is not initialized)
- **Help text**: tooltip text in the *Help Text* tab

2.7.4.2.5.1 Creating a Lazy-Loading Combo-Box

Typically, a lazy-loaded combo box offers the available options as the user inputs characters: in the background, the options need to be loaded in batch required by the combo box.

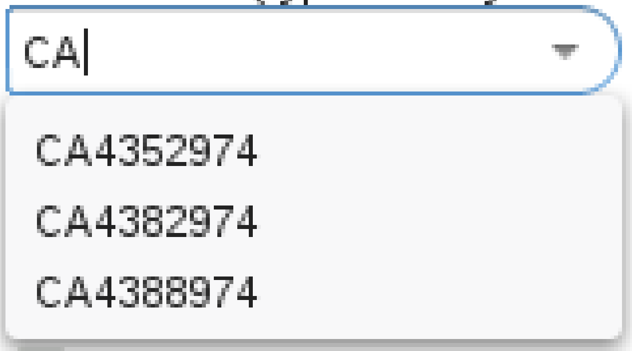


Figure 2.85 Lazy-Loading Combo Box with options for the user input

To create such a lazy-loading combo box, do the following:

1. Define the *Binding* property with the target slot for the selected option.
2. Define the *Options* property so that the closure returns the results for what the user typed into the combo box. The user input is passed as the first closure parameter.

   ```
   {
   userInput, firstPageItem, batchSize ->
      getCurrenciesContaining(userInput, firstPageItem, batchSize)
   }
   ```

   In the example above, the query is defined so as to return:

   - only results that contain the user input, for example, you can define a query with a condition, such as, `currentCurrency.code like userInput +"*"`
   - and that in batches defined by the closure input parameters `firstPageItem` and `batchSize`
3. Define the *Options count* property so that the closure returns the number of options returned by the *Options* property.

   ```
   {userInput -> getCurrenciesCountForFiltered(userInput)}
   ```

   Consider using the count query to obtain the options count.
4. Define the *Formatter* property so that it returns the string displayed in the drop-down menu.

   ```
   { c:Currency -> c.code }
   ```
5. Define the *Placeholder* property with the string that should be displayed if no option is selected. Alternatively, in the *Formatter* property handle the object for null value.

2.7.4.2.6 Single Select List

The Single Select List component ( ) is a form component that displays a list of options and allows the user to select exactly one option.

It produces events of the following types:

- *InitEvent* when the component is initialized or displayed if previously hidden
- *ValueChangeEvent* when the user selects an option (to process it also, set the Immediate property to true)

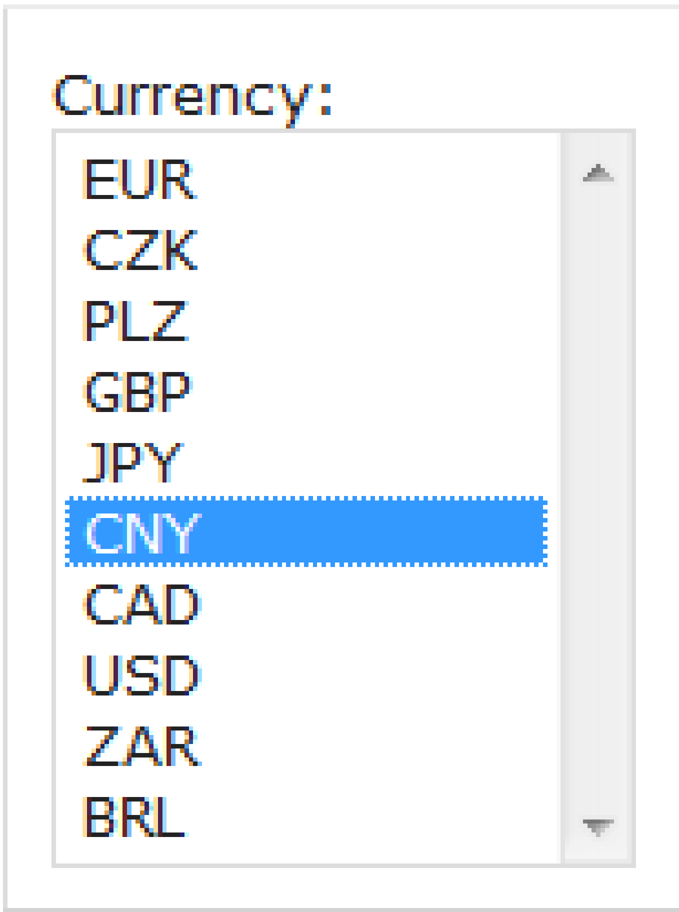


Figure 2.86 Single Select List in a Panel

- **Label**: visible label
- **Required**: compulsoriness of the component; If true, a mark indicating that the value is required is rendered in the component.

  Important: The property **does not provide any validation** whether the field contains a value and the user will be able to submit a null value unless a additional validation mechanism is defined. Such a validation can be implemented on a listener, possibly as a validation expression.

- **Binding**: reference to a slot that holds the selected option value (for example, a form variable or global variable of the respective type)

- **Options**: list of options

  The options are displayed as a vertical list. Make sure the defined expression resolves to an object of the type List<ui::Option> object.

  Note that the Option data type has the value and label field so that you can define the label displayed in the drop-down list that represents the given value.

- **Read-only**: editability of the Single Select List

  If true, the list renders as grayed out and no option can be selected.

- **Immediate**: setting of the Immediate mode

  To process the ValueChangeEvent at the moment the user selects an option, set the Immediate property to 'true'.

- **Help text**: tooltip text

2.7.4.2.7 Multi Select List

The Multi Select List component ( ) is a form component that displays a list of options and allows the user to select multiple options.

It produces events of the following types:

- *InitEvent* when the component is initialized or displayed if previously hidden
- *ValueChangeEvent* when the user selects an option (to process it also, set the Immediate property to true)

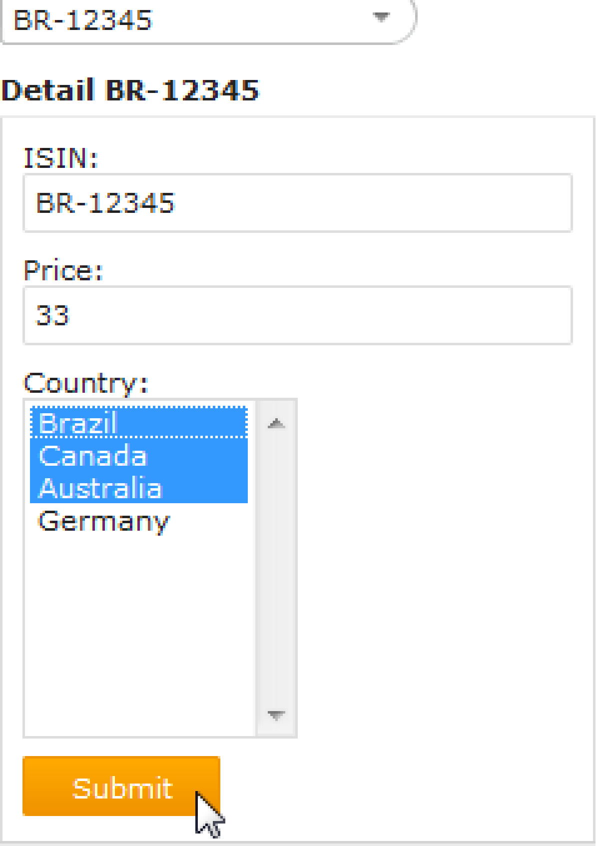


Figure 2.87 Multi Select List

- **Label**: visible label
- **Required**: compulsoriness of the component; If true, a mark indicating that the value is required is rendered in the component.

  Important: The property **does not provide any validation** whether the field contains a value and the user will be able to submit a null value unless a additional validation mechanism is defined. Such a validation can be implemented on a listener, possibly as a validation expression.

- **Binding**: reference to a set of objects that holds the selected values
- **Options**: list of options

  The list is displayed in the drop∗down menu. Make sure the defined expression resolves to an object of the type List<ui::Option>.

- **Read-only**: editability of the Multi Select List

  If true, the list renders as grayed out and no option can be selected.

- **Immediate**: setting of the Immediate mode

  To process the ValueChangeEvent at the moment the user selects an option, set the Immediate property to 'true'.

- **Help text**: tooltip text

2.7.4.2.8 Check Box List

The Check Box List component is a form component that displays a list of options with check boxes and allows the user to select multiple options using the check boxes.

It produces events of the following types:

- *InitEvent* when the component is initialized or displayed if previously hidden
- *ValueChangeEvent* when the user selects an option (to process it also at this point, set the Immediate property to true)

Figure 2.88 Check Box List in a Panel component

- **Label**: visible label
- **Required**: compulsoriness of the component; If true, a mark indicating that the value is required is rendered in the component.

  Important: The property **does not provide any validation** whether the field contains a value and the user will be able to submit a null value unless a additional validation mechanism is defined. Such a validation can be implemented on a listener, possibly as a validation expression.

- **Binding**: reference to a set of objects that holds the option values
- **Options**: list of options

  The list is displayed as entries in the check list. Make sure the defined expression resolves to an object of the type List<ui::Option> object.

- **Read-only**: editability of the Check Box List

  If true, the list renders as grayed out and no option can be selected.

- **Immediate**: setting of the Immediate mode

  To process the ValueChangeEvent at the moment the user selects an option, set the Immediate property to 'true'.

- **Help text**: tooltip text

2.7.4.2.9 Radio Button List

The Radio Button List component is a form component that displays a list of options and allows the user to select exactly one option by clicking a radio button.

It produces events of the following types:

- *InitEvent* when the component is initialized or displayed if previously hidden
- *ValueChangeEvent* when the user selects an option (to process it also at this point, set the Immediate property to true)

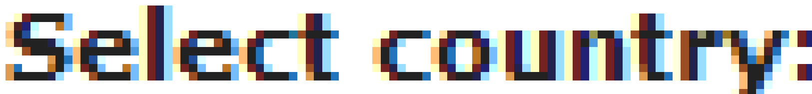


Figure 2.89 Radio Button List in the Panel component

- **Label**: visible label
- **Required**: compulsoriness of the component; If true, a mark indicating that the value is required is rendered in the component.

  Important: The property **does not provide any validation** whether the field contains a value and the user will be able to submit a null value unless a additional validation mechanism is defined. Such a validation can be implemented on a listener, possibly as a validation expression.

- **Binding**: reference to a slot that holds the selected option value (for example, a form variable or global variable of the respective type)

- **Options**: list of options

  The options are displayed as a vertical list with radio buttons on the left. Make sure the defined expression resolves to an object of the type List<ui::Option>.

- **Read-only**: editability of the Radio Button List

  If true, the radio button list renders as grayed out and no option can be selected.

- **Immediate**: setting of the Immediate mode

  To process the ValueChangeEvent at the moment the user selects an option, set the Immediate property to 'true'.

- **Help text**: tooltip text

2.7.4.2.10 Tree

The Tree component serves as a picker of exactly one option (node). Once the user selects a node, the selection is stored in the Binding slot.

It produces events of the following types:

- *InitEvent* when the component is initialized or displayed if previously hidden
- *ValueChangeEvent* when the user selects an option

The Tree ( ) component renders as a tree of nodes. The nodes are expandable unless they are leaf nodes, that is, they do not have any children. Whenever a node is expanded its child nodes are lazy-loaded. The Tree component calls the closure defined in the Children property to acquire child nodes if children in a Root element are null. The closure returns a collection of <TreeItem> objects.

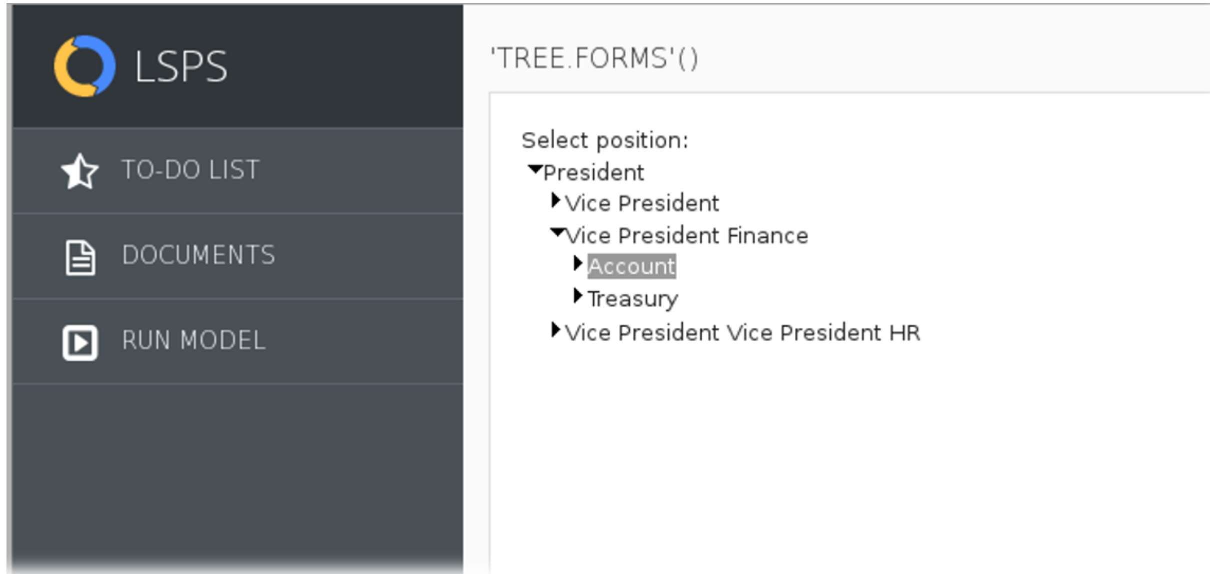


Figure 2.90 Tree with Multiple Nodes Expanded

The Tree component has the following properties:

- **Label**: visible label

- **Required**: compulsoriness of the component; If true, a mark indicating that the value is required is rendered in the component.

  Important: The property **does not provide any validation** whether the field contains a value and the user will be able to submit a null value unless a additional validation mechanism is defined. Such a validation can be implemented on a listener, possibly as a validation expression.

- **Binding**: reference to a slot that holds the selected tree option value (for example, a form variable or global variable)
- **Root**: collection of TreeItem that are root items of the tree
- **Children**: closure that returns a list of tree items which are displayed when a node is expanded

```
{ parent:Position, depth:Integer ->
  compact(
    collect( positions, { p:Position ->
      p.parent = parent ? new TreeItem (data -> p, expanded ->false, label -> p.name ) : null
      }
    )
  )
}
```

- **Read-only**: editability

  If true, the tree renders as grayed out and no option can be selected. The selection uses the value of the binding slot.

- **Immediate**: setting of immediate mode

  To process the ValueChangeEvent at the moment the user selects a node, set the Immediate property to 'true'.

- **Help text**: tooltip text

2.7.4.2.11 File Upload

The File Upload component allows the user to upload a file to the server. It renders as an input field for a file path and a Browse and upload button.

It produces events of the following types:

- *InitEvent* when the component is initialized or displayed if previously hidden
- *ValueChangeEvent* when the user inserts a file name
- *ActionEvent* when the file is uploaded
- **FileUploadEvent** produced on file-upload completion

Since file upload is an asynchronous process, on runtime, the component produces an ActionEvent when the upload button is clicked and an FileUploadEvent when the uploading finishes.

Figure 2.91 File Upload

- **Label**: visible label
- **Required**: compulsoriness of the component; If true, a mark indicating that the value is required is rendered in the component.

  Important: The property **does not provide any validation** whether the field contains a value and the user will be able to submit a null value unless a additional validation mechanism is defined. Such a validation can be implemented on a listener, possibly as a validation expression.

- **Binding**: reference to a set of objects that will hold the files
- **Read-only**: editability of the File Upload

  If true, the component is disabled and grayed out and no file can be selected.

- **Immediate**: setting of the Immediate mode

  To process the ValueChangeEvent at the moment the user selects a node, set the Immediate property to 'true'. In immediate mode, the Browse button is no longer be available and the user is prompted to select the file when they click the Upload button. When they select the file, the file is uploaded immediately.

- **Button text**: label of the upload button
- **Multiple**: setting for enabling upload of multiple files

  Important: This option is currently not supported since native HTML forms do not allow selecting multiple files in one window.

- **Upload to memory**: If true, the file is uploaded to memory. If false, the file is uploaded to LSPS_BINARY_↩ DATA table in the application database.
- **Delete temp data**: if true, the files from the LSPS_BINARY_DATA are deleted when the respective HTTP session finishes (applicable only if Upload to memory is set to false)
- **Help text**: tooltip text

2.7.4.3 Output Components

Output components serve to display data in a certain way and include the following:

- Output Text
- Tabular Components
- Repeater
- Image
- Navigation Link
- File Download
- Pie Chart
- Gauge Chart
- Cartesian and Polar Chart
- Browser Frame
- Calendar
- Map Display

2.7.4.3.1 Output Text

The Output Text component is rendered as a read-only, single-line text field.

- **Label**: label of the output box
- **Content**: object that is displayed in the output box
- **Format**: formatting applied on the object defined in the Content property

  The applicable format depends on the data type of the object from the Content property:
  - Format for a **Date** object: as java.text.SimpleDateFormat, for example `"EEE, MMM d, ''yy"`
  - Format for Integer and Decimal: as java.text.DecimalFormat, for example `"#0.00"`
  - Format for a **String** can be set to:
    - `html`: HTML tags are supported.
    - `plaintext`: Line breaks in the Content object are ignored.
    - `preformatted`: Line breaks are respected.
  - Objects of **other types** are transformed with the `toString()` function with no format options.

2.7.4.3.2 Tabular Components

Tabular components include the Table and Tree Table components which share common features: they serve to display data in a tabular manner and have the Column components as their immediate children. Both work with the same types of data sets and support ordering and filtering of their content.

2.7.4.3.2.1 Table

The **Table** component displays the data of its data set in the child components of the table's columns: The system iterates through individual objects of the data set and each object becomes the *data iterator* value for a row. The iterator is then used by the child components of table columns to access the data object.

It supports ordering, filtering, and grouping of the data.

The way the data set is acquired and the way the table is rendered is defined by the table type:

- **simple**: The table is on one page and the data is obtained in a single request.
- **lazy**: The table is rendered with a particular size and with a scroll. The data is loaded as the user scrolls through the table.
- **paged**: The table is rendered with a page navigation at the bottom. The data is acquired per page.

The data set itself can be defined as the following:

- **Type**: shared record or a shared record field

  The system fetches all shared record instances from the database.
- **Query**: query that returns a collection of objects

  The table iterates through the collection objects.
- **Collection**: collection of the data object

  The table iterates through the collection objects.
- **Data**: parametric closure that returns the collection of data objects

  The first parameter holds the index of the first entry; The second parameter is the count of entries per load or page. You will use this option typically when integrating with other systems, for example, `{currentIndex, count -> getEntryBatch(currentIndex, count)}`.

  This data kind requires the **Data count** property, expression that returns the total amount of entries to be loaded.
- **Generic**: an Object that results in any of the above on runtime

  The setting is used for generic reusable tables when the user wants to fill the same table with different data queried in different ways, typically when reusing the form in other forms.

2.7.4.3.2.2 Creating a Simple Table

A simple Table is rendered as a table on a single page with all its items.

To create a simple Table, do the following:

1. Insert the Table component in a Form.
2. On the *Details* tab of the Properties view, define the following:
   - Data Kind: data source type and the related data
   - *Data Iterator*: reference to a local form variable of the same type as your Data objects.

     Important: The data iterator can be used solely as the table iterator: Using the iterator out of the Table only when creating the table dynamically. The practise is generally discouraged and results in a validation problem.
   - *Type*: `TableType.simple`
3. Insert Table Columns into the Table.
4. Into each column, insert the required components and bind them to the iterator data.

2.7.4.3.2.3 Creating a Paged Table

A paged table data is rendered on pages with the number of rows defined by the *initial-page-size* presentation hint and page navigation at the bottom. It can be set to load a particular page using the *Show Index* property.

A paged table produces the **TablePageSizeChangeEvent** when the user changes the size of the table page.

To create a paged Table, do the following:

1. Select the Table component in a Form.
2. On the *Details* tab of the Properties view, define the following:
   - Data Kind: data source type
   - *Type*: `TableType.paged`
   - *Data Iterator*: reference to a local form variable of a type that can hold your Data objects

     Important: The data iterator can be used solely as the table iterator: using the iterator out of the Table results in a validation Error.
   - *Show index*: the index number of the page that should be opened on load.
   - *Index Iterator*: reference to an object that will hold the index of the row where the current Data object is used
3. Set the size of the page in the `initial-page-size` presentation hint.
4. Insert Table Columns into the Table.
5. Into each column, insert the required components and bind them to the iterator data.

2.7.4.3.2.4 Creating a Lazy-Loaded Table

A lazy table is rendered as scrollable table with the number of rows defined by the *initial-page-size* presentation hint.

To create a Table, do the following:

1. Select the Table component in a Form.
2. On the *Details* tab of the Properties view, define the following:
   - Data Kind: data source type
   - *Type*: `TableType.lazy`
   - *Data Iterator*: reference to a local form variable of a type that can hold your Data objects

     Important: The data iterator can be used solely as the table iterator: using the iterator out of the Table results in a validation Error.
   - *Show index*: the index number of the page that should be opened on load.
   - *Index Iterator*: reference to an object that will hold the index of the row where the current Data object is used
3. Insert Table Columns into the Table.
4. Into each column, insert the required components and bind them to the iterator data.

2.7.4.3.2.5 Defining Grouping on a Table

Groups serve to group the items in a table on multiple levels according to their properties in a tree-like way, for example, a list of songs according to their interpret and then according to the album.

Note that grouping takes place *on the data set* and renders the table type setting irrelevant.

The groups are defined as a subtype of the abstract `GroupSpec` record and that as one of the following:

- **PropertyGroupSpec**: the groups are defined as Record properties;

```
[
//first-level grouping:
new PropertyGroupSpec (
    label -> "Surname",
    groupBy -> Author.surname),
    //second-level grouping:
    new PropertyGroupSpec
          (label -> "First name",
          groupBy -> Author.firstName)
]
```

- **ClosureGroupSpec**: the groups are calculated on runtime in a closure;

```
[new ClosureGroupSpec(
   label -> "Surname",
   groupBy -> {a:Author -> a.surname}
   )
]
```

- **OptClosureGroupSpec**: the groups are calculated on runtime in a closure in optimized manner;

```
[new OptClosureGroupSpec(label -> "Surname",
      groupBy ->
        { c:Collection<Author > ->
        def Set<String> surnames := toSet(collect(c, {a:Author -> a.surname}));
        //creates map with surname as the key and collection of authors as value
        map(surnames, {surname:String -> select(c, {a:Author -> a.surname = surname})})}
    )
]
```

Alternatively, if you are using the Type data kind, you can infer grouping: The system will "guess" the applicable groups. The Group Spec then becomes a collection of all bindings of child components of all columns. Therefore, grouping is inferred only on columns with one Input component with binding set to a simple data type entity.

To define grouping on a table, do the following:

1. In your form definition, select the table component.
2. In the Properties view, open the *Grouping* tab.
3. Uncheck Disable Grouping.
4. Define the grouping:
   - For your own grouping:
     (a) Define the GroupSpec and optionally Grouping that holds the currently applied grouping.
     (b) Optionally, on individual table columns, define the Group value on their Grouping tab. The value is used as a group header in the line that serves to expand and collapse the group.
     `{values:Collection<Author>->values[0].gender}`

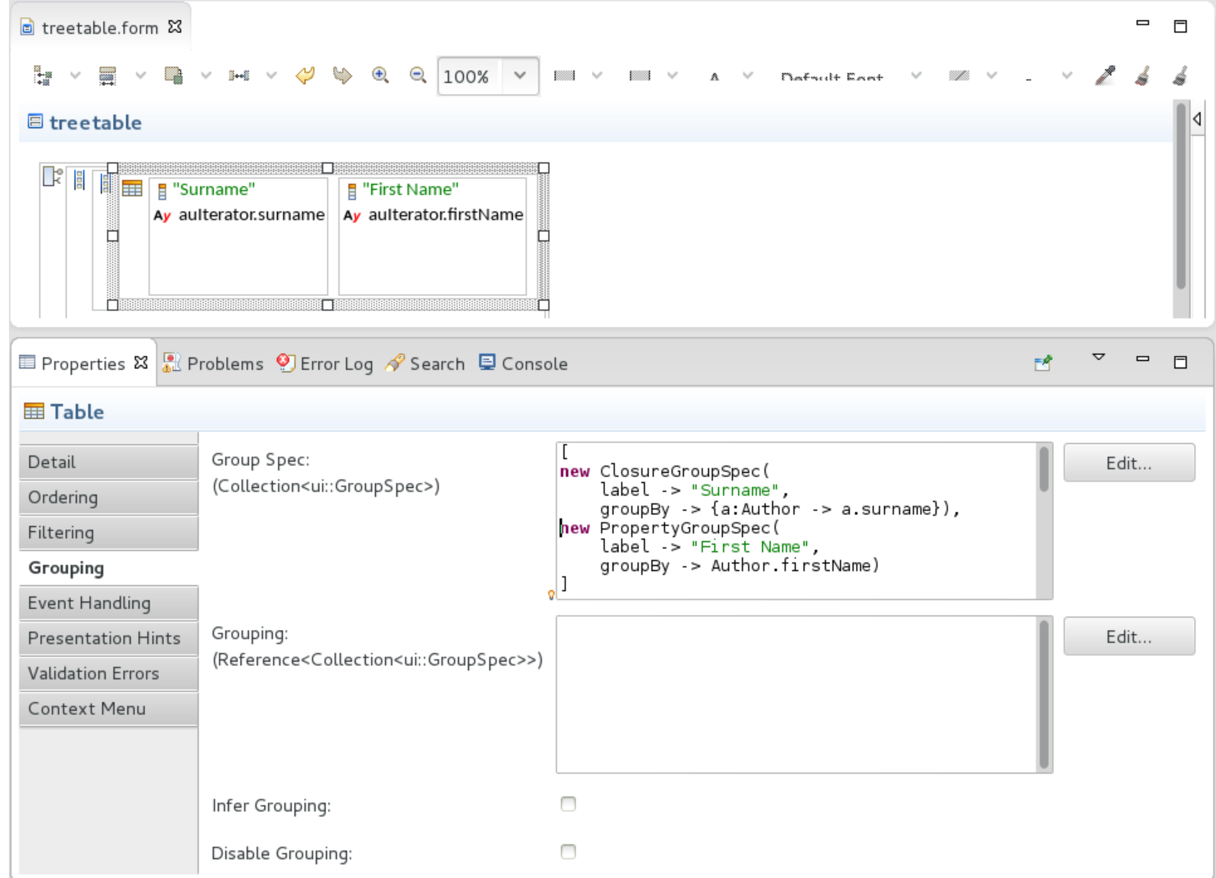


Figure 2.92 Form with Table and its Grouping Definition

- To infer grouping, select **Infer Grouping**.

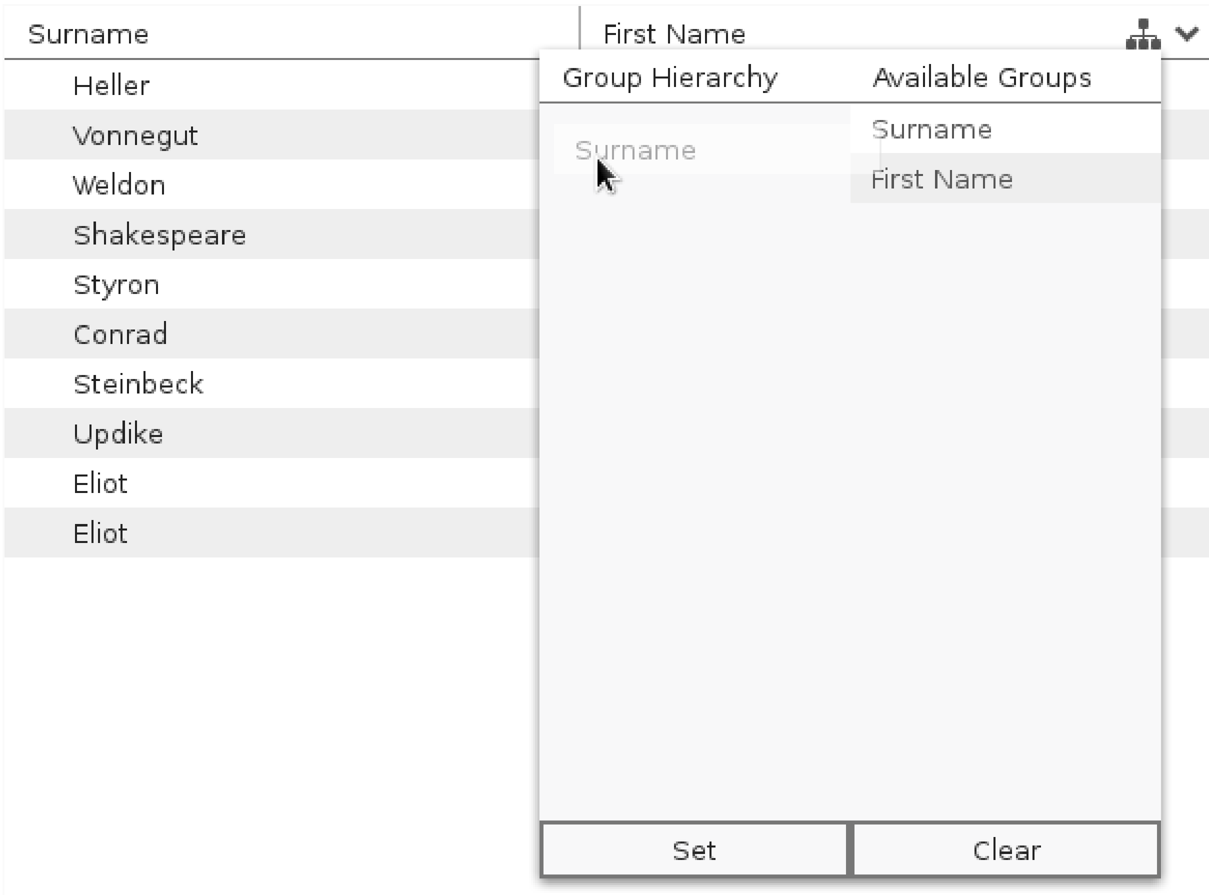
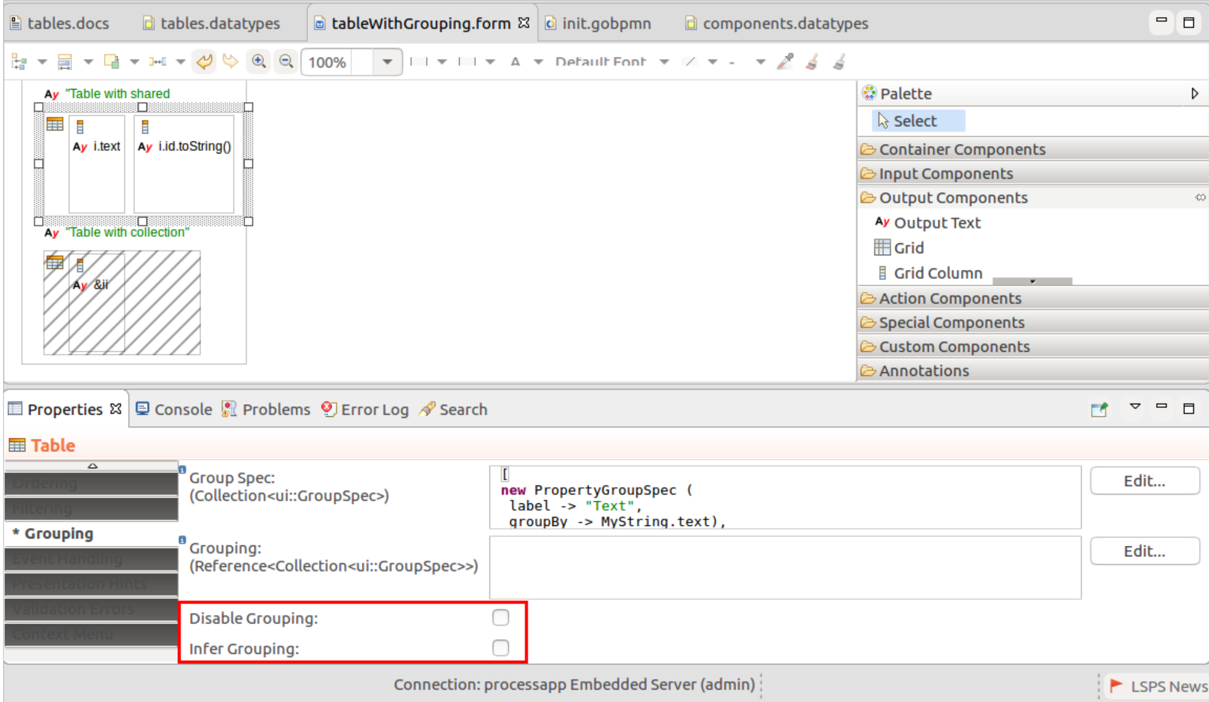


Figure 2.93 Setting Grouping

2.7.4.3.2.6 Disabling Grouping on Tables

To disable grouping on a table, do the following:

1. In the Properties view of the Table component, go to the Filtering, Ordering, or Grouping tab.
2. Unselect the applicable grouping option:
   (a) Unselect **Disable Grouping** to disable grouping altogether.
   For the Columns with inferring of grouping, the inferring will be disabled. Other Grouping settings will be applied.
   (b) Unselect **Infer Grouping** to disable inferred grouping.



2.7.4.3.2.7 Tree Table

The *Tree Table* ( ) component renders as a table with rows organized in a tree structure. The rows of the collapsible tree nodes are loaded on expand.

Individual nodes are represented by *TreeItem* objects, which hold the business data and information about the tree node and its child elements: The tree table component iterates through a collection of the root <TreeItem> objects. If a root object defines children, it iterates also through the child elements. Every element becomes the object for a row.

Tree Tables support ordering and filtering of their content.

The TreeItem objects define the following:

- data: business object the tree table item operates over (business data of a row)
- label: label you can use as component content
- expanded: whether the node is expanded by default
- parent: parent TreeItem object (element "above")
- children: child TreeItem object (elements "inside")

You can access the data through the *iterator* or *tree item iterator*: the iterator holds only the business data object, while the tree item iterator holds its TreeItem object.

Figure 2.94 Tree Table with expanded nodes

The Tree Table component has the following properties:

- **Root**: collection of root TreeItem elements
- **Children**: closure that returns a list of tree table items that are loaded when a node is expanded

  The closure is called when the children in the Root expression are null; note that when *children* is an empty collection, the node is considered a leaf and the closure is not called.
- **Iterator**: reference to an object of the business data type

  The iterator will hold the *data* object of the given row so it has to be able to hold any business data type used in the Tree Table: if the table uses different records, consider using the *Record* type or another parent data type.

  > **Important:** The data iterator can be used solely as the table iterator: using the iterator out of the Table results in a validation Error.
- **Tree Item Iterator**: reference to an object of the TreeItem type

  It will hold the TreeItem object of the current row.

  > **Important:** The iterator can be used solely as the table iterator: using the iterator out of the Table results in a validation Error.

2.7.4.3.2.8 Creating a Tree Table

To create a Tree Table, do the following:

1. Insert the Tree Table component in a Form.
2. Define the *Root* property with the collection of the root nodes and possibly their children.

```
[new TreeItem(
//business object
    data -> boss,
    label -> "Boss",
    expanded -> true,
    parent -> null,
    children -> [
```

```
            new TreeItem(
                data -> employee,
                label -> "Employee",
                expanded -> true,
                parent -> null,
                //if children is null, like below, then the Children property is used to retrieve
                children -> null)
        ]
    )
]
```

3. Define the Children property (the closure that returns the child objects for root objects with null value in *children*).

```
{ treeitem -> collect(getAllChildren(treeitem), convertToTreeitems())}
```

4. Define the Iterator and the TreeItem iterator: typically these are references to local form variables.

5. Insert Table columns into the Tree Table.

6. Insert the required components into the Columns, typically, *OutputTexts*, *Text Boxes*, etc.

7. Define the content of the Column components using the iterators where applicable.

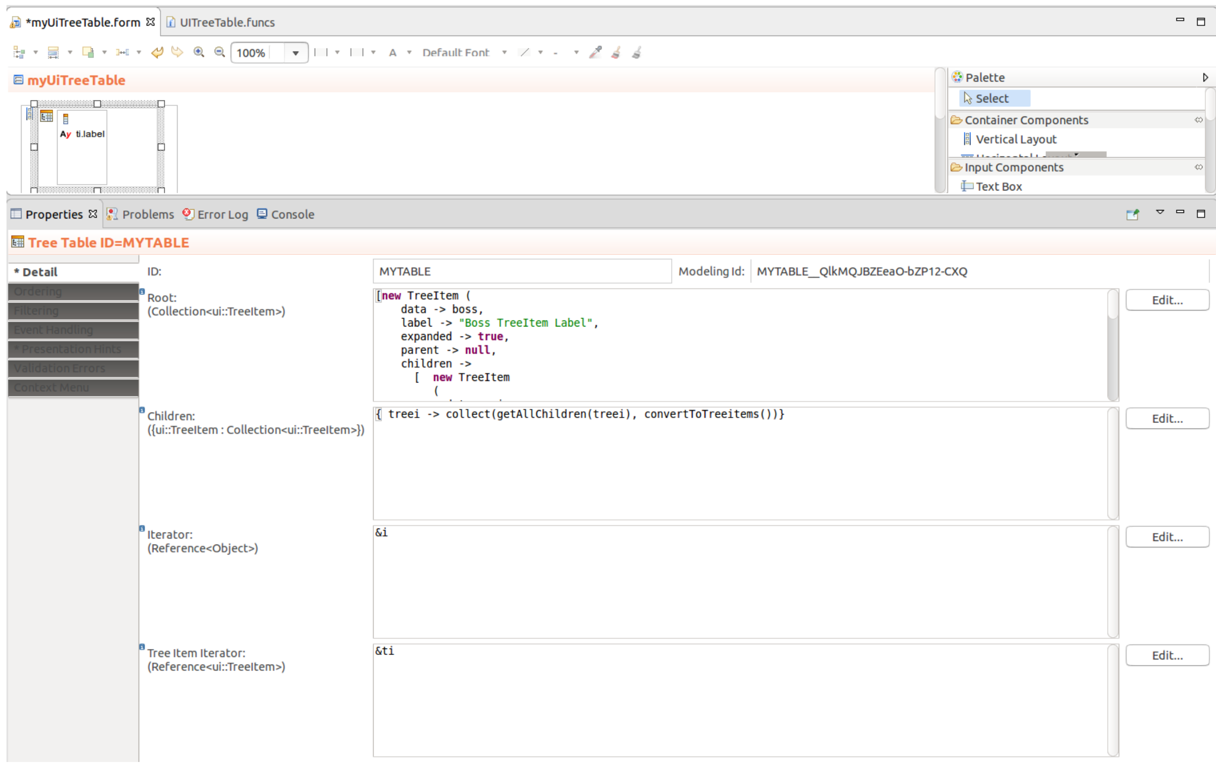


Figure 2.95 Tree Table form with its Detail properties

2.7.4.3.2.9 Table Columns

The *Table Column* component can represent column of a Table or a Tree Table. It defines table column properties, including ordering and filtering options. Its children are components that can operate over the Table business data.

To define how children of a column are aligned use the `align` presentation hint: note that the hint is applied on the header as well.

> **Warning:** Note that refreshing a table column does not refresh its underlying data set: only its header, hints, and any data not related to the data set, such as global variable references, are recalculated. This is justified by the fact that if you managed to refresh only the data set for a column, the table could be populated with inconsistent data. To refresh the underlying data set of a column, either refresh the parent table or the content of the column child components.

Table Column Properties

The Table Column component has the following properties:

- **Header**: column header
- **Visible**: visibility of the table column

  If false, the column is not visible or available in the column picker.

  The expression is evaluated whenever the column's parent component is refreshed.

2.7.4.3.2.10 Collapsing a Table Column

To hide a Table Column, that is to display it collapsed, set the `hidden` presentation hint to true: this setting will be applied when the form is initialized.

2.7.4.3.2.11 Saving Column Width and Collapsed State

To save the column width in a table and the collapsed state of a table when a document or a to-do is saved, call the `getColumnStates(<MYTABLE>)` function and store the returned column states. When applicable, call the `restoreColumnStates(<MYTABLE>, <COLUMNSTATE>)` function to restore the column widths and collapsed states.

2.7.4.3.2.12 Ordering and Filtering of Tables and Tree Tables

Tables and Tree Tables support

- ordering: sorting of data in an ascending or descending order when the user clicks a column label
- filtering: filtering of data so that only entries that meet the filter requirements are displayed

The features are enabled on the Table and Tree Table components; however, since their logic applies to Columns, that is where they are actually defined.

All the features allow their inferring, that is, setting up the feature automatically as best effort for the given Column data.

2.7.4.3.2.13 Defining Ordering

Ordering is defined on the Column components in Tables or Tree Tables. By default, Columns are set to infer ordering so the feature is automatically set up. However, inferring is applicable only on Columns with exactly one input component and with their binding set to a simple data type. For columns with multiple or other than input components, the inferring of ordering is not supported. Also, if the data set is obtained as Data, you will need to define the ordering manually.

> **Note:** Ordering cannot be applied on columns with enumeration values. If you want to order a table according to an enumeration, use a query to acquire the data set already sorted. You can then define the enumeration as the sorting property on table columns.

To define ordering on a table, do the following:

1. Define the ordering properties on the Ordering tab of the Table or Tree Table:
   - **Ordering**: reference to a variable that holds the ordering applied to the table at the given moment
   - **Infer ordering**: if checked, inferring of ordering on the table is enabled (Columns can infer filtering).
     Ordering can be inferred only on tables that define a shared Record type as its data kind and that only on columns that contain exactly one input component (the infer guess is performed based on the binding reference of the component).

   Once you have defined ordering properties on the table, you will need to define ordering on table columns.
2. Define the ordering properties on individual columns: On the Ordering tab in the column Properties view, select the Ordering kind and define the respective Order By expression:
   - Property: the expression must return a record field of a simple data type. The returned type must be present in the row scope type. The table data will be sorted according to the values of this field in row scopes when the user clicks the column header.
     ```
     //Author is a shared record that is part of the row scope and surname is its field.
     Author.surname
     ```
   - Expression: the closure expression must return a field of a simple data type. The input closure parameter is the row scope. The returned type must be present in the row scope type. The table data will be sorted according to the values of this field in row scopes when the user clicks the column header.
     ```
     //a incoming parameter is the row scope, in this case, the Author instance; the closure returns the
           instances surname
     {a:Author -> a.surname}
     ```
   - Enumeration: the name of the ordering enumeration. This setting applies only to data acquired using non-native *query* and defines entered ordering enumeration.
   - Infer: no Group By expression applies. Inferring is applied only on the particular column.
     If setting the value to **Infer**, make sure inferring of ordering is enabled on the parent table or tree table.
   - Disabled: ordering is disabled on the particular column.

2.7.4.3.2.14 Tracking Current Ordering on Tables and Tree Tables

To track and programatically change the ordering of a Table or Tree Table, define the Ordering expression on the Ordering tab of the component properties. The value is a reference to a map and will be set to values in the form `[ORDER_BY -> ORDER_DIRECTION]`, for example `[MyForm::MyRecord.id -> ui::↩ OrderDirection.Ascending]`.

Other model resources can use the variable value to detect the current ordering. Also, if you set the reference to an expression, the expression will be evaluated when the component is visualized in the front-end application and used as default ordering on the Table or Tree Table.

2.7.4.3.2.15 Defining Filtering

Filtering on Tables and Tree Tables is by default enabled and Columns are set to infer the filtering settings: by default your Tables and Tree Tables have filtering set up: However, inferring is applicable only on Columns with exactly one input component with binding to a simple data type. For columns with multiple or other than input components, you need to define filtering manually.

To define filtering on a table, do the following:

1. Define the filtering properties on the Filtering tab of the Table or Tree Table:
   - **Filtering**: currently applied filtering expression
   - **Infer Filtering**: if checked, inferring of filtering on the table is enabled (Columns can infer filtering).

     Note: Filtering can be inferred only on table columns that contain exactly one input component. The guess is performed based on the binding reference.
2. Define the filtering properties on individual columns: On the Filtering tab in the column Properties view, do the following:
   (a) In the Filter UI field, define the filter type (for example, `new ui::SubstringFilterUI(substring -> "default substring"), new ui::RegExpFilterUI(regexp -> "default regex")`).

   (b) Select the Filtering type and define the respective Filter By expression:
   - **Property**: the expression must return a record field of a simple data type. The returned type must be present in the row scope type. The table data will be filtered according to the values of this field in row scopes.
     ```
     //Author is a shared record that is part of the row scope and surname is its field.
     Author.surname
     ```
   - **Expression**: the closure expression must return a field of a simple data type. The input closure parameter is the row scope. The returned type must be present in the row scope type. The table data will be sorted according to the values of this fields in row scopes when the user clicks the column header.
     ```
     //a incoming parameter is the row scope
     // in this case, the Author instance; the closure returns the instances surname
     {a:Author -> a.surname}
     ```
   - **Custom**: applicable only on Data (paged collection) custom filter definition (Filter must be of PropertyFilter, ClosureFilter, or CustomFilter)
     ```
     new CustomFilter(ui -> new TextBox(  binding -> &filterBinding),
          filterText -> {->"Surname Filter"},
          popup -> false)
     ```
   - **Infer**: no Filter By expression applies. Filtering is applied only on the particular column.
     If setting the value to **Infer**, make sure Inferring of Filtering is enabled on the parent component.
   - **Disabled**: filtering is disabled on the particular column.
3. On the Table, consider defining the `no-data-message` presentation hint: the hint value will be displayed in the Tables when it is empty.

2.7.4.3.2.16 Disabling Filtering and Ordering on Tables and Tree Tables

To disable filtering, or ordering on a table, do the following:

1. In the Properties view of the component, go to the Filtering, Ordering, or Grouping tab.
2. Select the *Disable* option:
   (a) To disable any filtering and ordering, select **Disable Filtering** or **Disable Ordering**.

   (b) To disable only the inferred filtering or ordering, unselect **Infer Filtering** or **Infer Ordering**. For the Column with inferring of filtering, ordering, or grouping, the inferring will be disabled. Other Filtering, Ordering, or Grouping settings will be applied.

2.7.4.3.3 Repeater

The *Repeater* component is a Form output component that renders its child component multiple times.

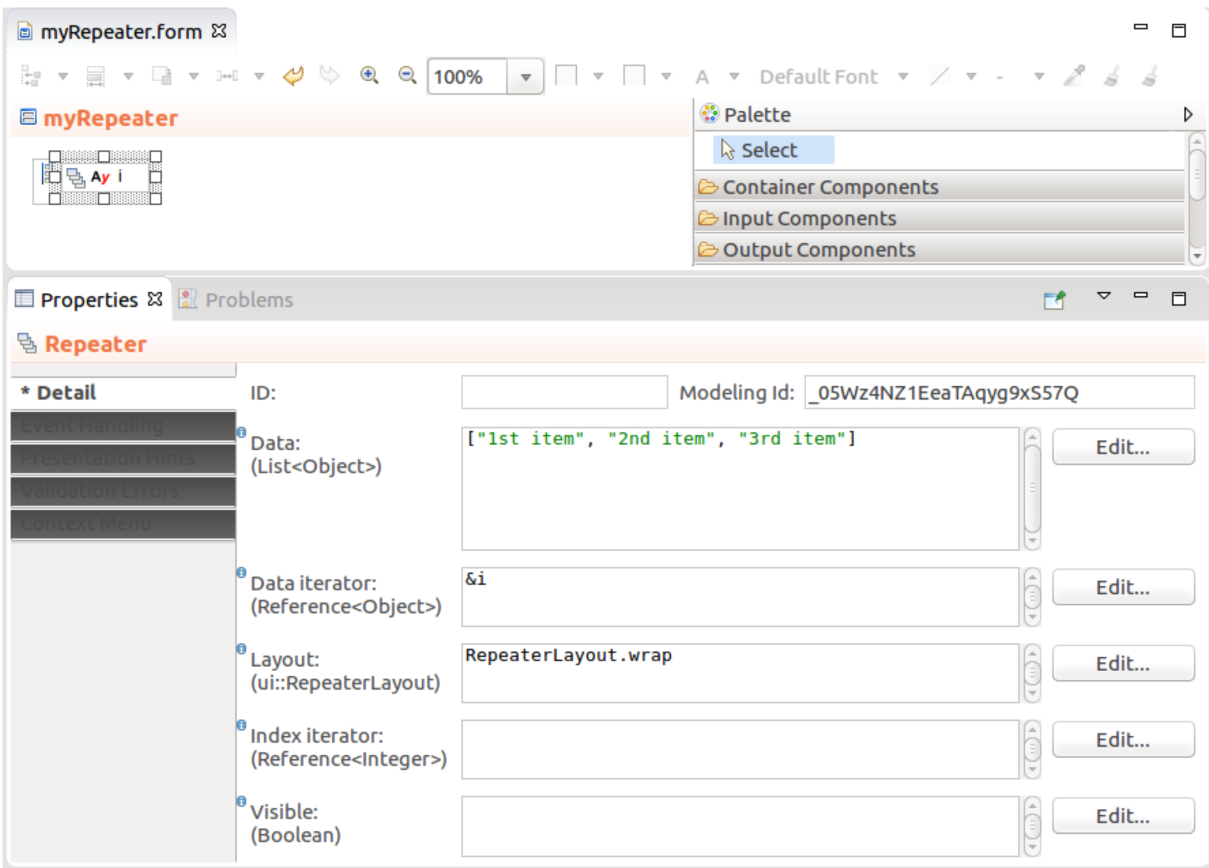
- **Data**: list of objects to iterate through
- **Data iterator**: reference to the iterated object (Since an iterator holds the value of the object for the current iteration, the referenced object must be of the same type as the Data list objects)

  Important: If you define a reference to a variable as iterator, make sure the variable is used solely as the repeater iterator.

- **Layout**: a RepeaterLayout that defines the layout applied on child elements
  - `RepeaterLayout.wrap`: repeated items are arranged horizontally; if their content is larger than the width wrapping, the lines overflow.
  - `RepeaterLayout.horizontal`: repeated items are arranged horizontally. Their size is ignored.
  - `RepeaterLayout.vertical`: repeated items are arranged vertically. Their size is ignored. If some of the child elements are not displayed in the repeater, set the child components width to `Wrap Content`.

    Important: The Layout property is not recalculated on refresh.

- **Index iterator**: reference to an Integer variable that holds the index of the currently iterated object



2.7.4.3.4 Image

The Image component is a Form output component that renders an image File object.

- **Content**: the image object to be displayed; you can use the `getResource()` function; for example, `getResource("myModule", "picture.jpg")`

  Note: The File object cannot be created directly over a file in the file system. Therefore, you need to import the image into your project (File > Import > General > File System) and create the File object over the imported image.

- **Text**: image caption
- **Help text**: tooltip text; you can define the Help text on the Help Text tab in the Properties view.

2.7.4.3.5 Navigation Link

The Navigation Link component renders as a hyperlink; when clicked, it redirects the user to navigation target.

- **Content**: the Navigation object to be used for navigation

  ```
  new UrlNavigation(url -> "http://www.whitestein.com")
  ```

- **Text**: text displayed in the navigation link
- **Disabled**: whether the link is disabled (rendered as grayed out and not clickable)
- **Help text**: tooltip text

  Note: You can define the Help text on the Help Text tab in the Properties view.

2.7.4.3.6 File Download

The *File Download* component renders as a hyperlink: when clicked, it produces the FileDownloadEvent and downloads a file.

It produces events of the following types:

- InitEvent when the component is initialized or displayed if previously hidden
- FileDownloadEvent when the user clicks the file-download button

File Download defines the following properties:

- **Content**: File object with the file
- **Text**: text of the download hyperlink
- **Help text**: tooltip text

2.7.4.3.7 Pie Chart

The *Pie Chart* component renders as a circular chart that depicts data values as sections. When clicked, it produces a *ChartClickEvent* with data about what was clicked so the system can process the click as required.

It produces events of the following types:

- InitEvent when the component is initialized or displayed if previously hidden
- ChartClickEvent when the user clicks a chart

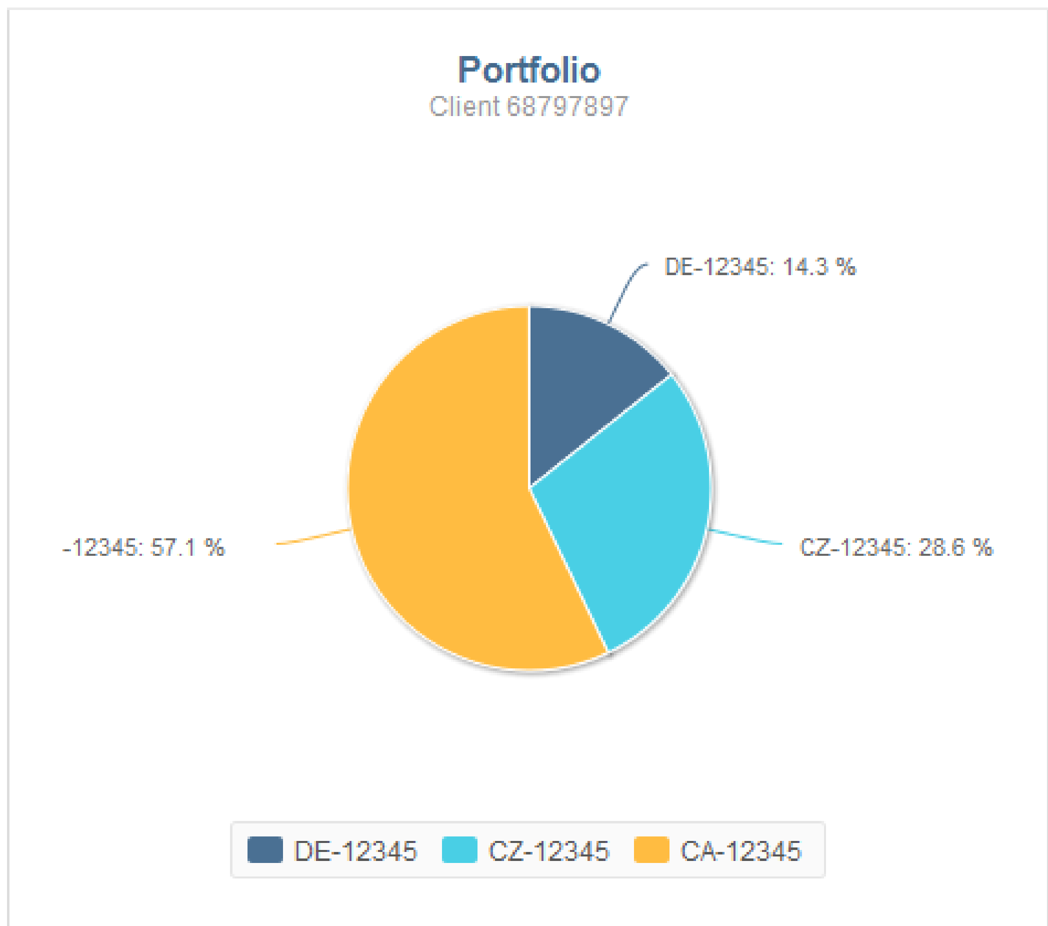


Figure 2.96 Pie chart with legend

The Pie Chart component has the following properties:

- **Title**: pie chart main title
- **Subtitle**: pie chart subtitle
- **Slices**: pie chart sections (a list of slices displayed in the pie chart)

  The slices define their label and value: the Slice value is defined as a decimal value and represents the mutual ratio of individual slices.

```
collect(
  getCurrentAssets(),
  {asset ->
    new ui::PieSlice(
      label -> asset.ISIN,
      value -> asset.currentAmount)
  }
)
```

- **Show legend**: visibility of the chart legend (if true, the legend is visible)

2.7.4.3.8 Gauge Chart

The Gauge Chart component renders as a chart with a circular Y-axis and a rotating pointer.

It produces events of the following types:

- InitEvent when the component is initialized or displayed if previously hidden
- ChartClickEvent when the user clicks a chart with information about the clicked data series

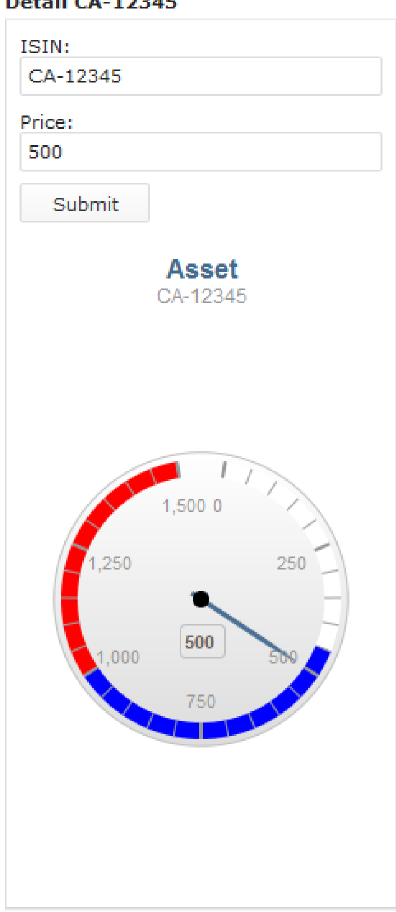


Figure 2.97 Form with the Asset gauge chart displaying asset price

Gauge Chart Properties

- **Title**: gauge chart main title
- **Subtitle**: gauge chart subtitle
- **Value**: value displayed by the pointer (needle)
- **Value name**: value name displayed in the pointer tooltip along with the Value
- **Axis**: gauge axis that defines the gauge chart scale, label, properties of the scale, bands, and chart position: The gauge axis constructor takes the following arguments:
  - min and max: minimal and maximal values on the gauge scale
  - label: label displayed directly on the gauge chart
  - opposite: position of the axis (if true, the axis is displayed on the edge of the gauge circle; if false, the axis is displayed inside the gauge circle)
  - bands: plot bands (shown white, blue, and red in the gauge chart figure above; CCS color definitions are supported)
  - startAngle and endAngle: start and end angle of the gauge axis
  - centerY: horizontal positioning of the gauge chart middle in percent ("50" places the chart directly under its title with no gap in between)

Example Axis definition

```
new ui::GaugeAxis(
    min -> 0,
    max -> 80,
    label -> "axis label",
    opposite -> null,
    bands -> [
        new PlotBand(from -> 0, to -> 20, color -> "#FFFFFF"),
        new PlotBand(from -> 20, to -> 40, color -> "blue"),
        new PlotBand(from -> 40, to -> 50,  color -> "red")],
    startAngle -> 300,
    endAngle -> 420,
    centerY -> 120)
```

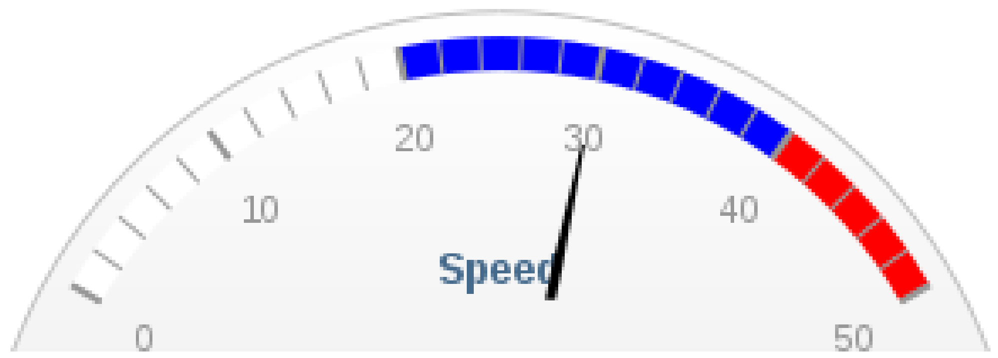


Figure 2.98 Rendered gauge chart with the axis as defined above

- **Show legend**: visibility of the chart legend

This property is not applicable for the gauge chart component.

2.7.4.3.9 Cartesian and Polar Chart

The Cartesian Chart renders as a multi-dimensional chart with an arbitrary number of x and y axes. The rendering of the x axis depends on the given data series, while the y axis displays the value connected to the x value defined as a data point.

The Polar Chart is a variation of the Cartesian chart. It is rendered as a circle with the x axis on its circumference and its radius is the y axis. Just like the Cartesian chart, it is multi-dimensional chart, that is, it can have n arbitrary number of x and y axes. However, though this option is functional, the y axes are overlaid over each other in a single radius.

The charts produce events of the following types:

- InitEvent when the component is initialized or displayed if previously hidden
- ChartClickEvent when the user clicks into the chart

  The event holds data about the clicked data point: a listener can use this data, for example, to drill down into the chart or visualize details related to the clicked data point.

Cartesian and Polar Chart Properties

- **Title**: chart main title
- **Subtitle**: chart subtitle
- **Series**: a set of data series displayed in the chart (see data series)
- **X axes**: list of x chart axes

  You can define multiple x axes. The axes are arranged underneath or next to each other depending on their orientation.
- **Y axes**: list of y chart axes

  You can define multiple y axes. The axes are arranged underneath or next to each other depending on their orientation.

  Axes for Cartesian and Polar chart define the following properties:

  – min and max: minimal and maximal values on the axis
  – label: label displayed directly on the chart
  – opposite: position of the axis (if true, the axis is displayed as the opposite axis, that is x is displayed as y and vice versa
  – bands: plot bands (any CCS colors are supported)
- **Rotate axes**: Boolean value that defines whether the x and y axes are rotated (for example, if true, chart bars can be displayed horizontally)

  This option is not available for the Polar Chart component.
- **Show legend**: visibility of the chart legend (if true, the legend is visible)

Data series defines a set of data points that are displayed as values in the chart. It also defines general properties of the data series:

- `label`: the data series label in the legend and tooltip of the plotted data series
- `options`: plotting options
- `xAxisIndex`: index of the x axis the data series uses

- `yAxisIndex`: index of the y axis the data series uses

  Since chart axes are defined as lists, the xAxisIndex and yAxisIndex are defined as integers with the first defined axis being indexed `0`.

Important: The ui::DataSeries record is abstract: Define the Data Series as one of its sub-types.

One chart can display multiple data series of different types. The type of a data series defines the way its x axis renders (note that a chart may contain multiple x or y axes):

- **ListDataSeries**: the x axis values are integers.

  Values are defined as a list of data points (**List**<**DataPoint**>) and are distributed evenly as depicted below.

Figure 2.99 Cartesian chart with ListDataSeries ("Company A" and "Index") and the series definition (further below)

```
def List<DataPoint> companyAPriceList :=
[
    new DataPoint(value -> 40),
    new DataPoint(value -> 60),
    new DataPoint(value -> 35),
    new DataPoint(value -> 10),
    new DataPoint(value -> 30),
    new DataPoint(value -> 30)
];

[
new ListDataSeries(
    label -> "Company A",
    options -> null,
    xAxisIndex -> null,
    yAxisIndex -> null,
    values -> companyAPriceList
    ),
new ListDataSeries(
    label -> "Index",
    options -> new PlotOptionsLine(
        color -> "#005398",
        showLabels -> null,
        stacked -> null,
        marker -> null,
        lineStyle -> LineStyle.solid,
        lineWidth -> 3,
        spline -> true),
    xAxisIndex -> null,
    yAxisIndex -> null,
    values -> companyBPriceList
    )
]
```

Figure 2.100 Cartesian chart with ListDataSeries (“Company A” and “Index”) and the series definition (further below)

- **CategoryDataSeries**: the x axis values are arbitrary string values.

Values are defined as a map of Strings and DataPoints (**Map**<**String, DataPoint**>): the String is used as the value on the x axis and the data point defines the values on the y axis.

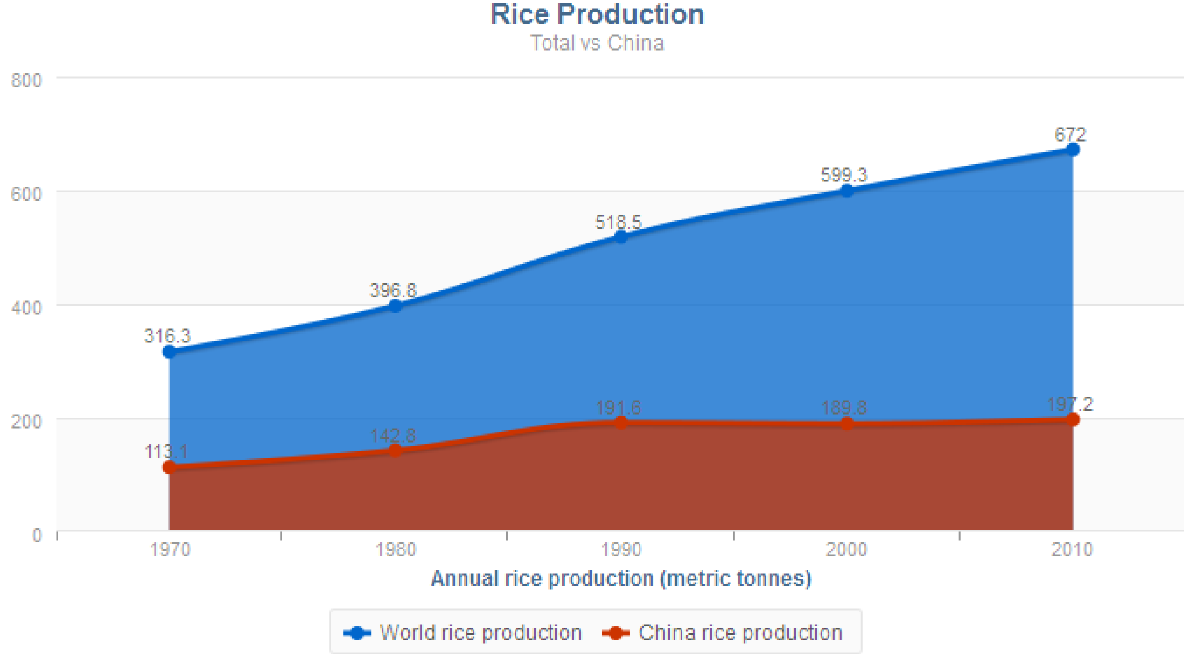


Figure 2.101 Cartesian and Polar charts with CategoryDataSeries (“World rice production” and “China rice production”) and years as String values; The definition of the “World rice production” CategoryDataSeries (further below)

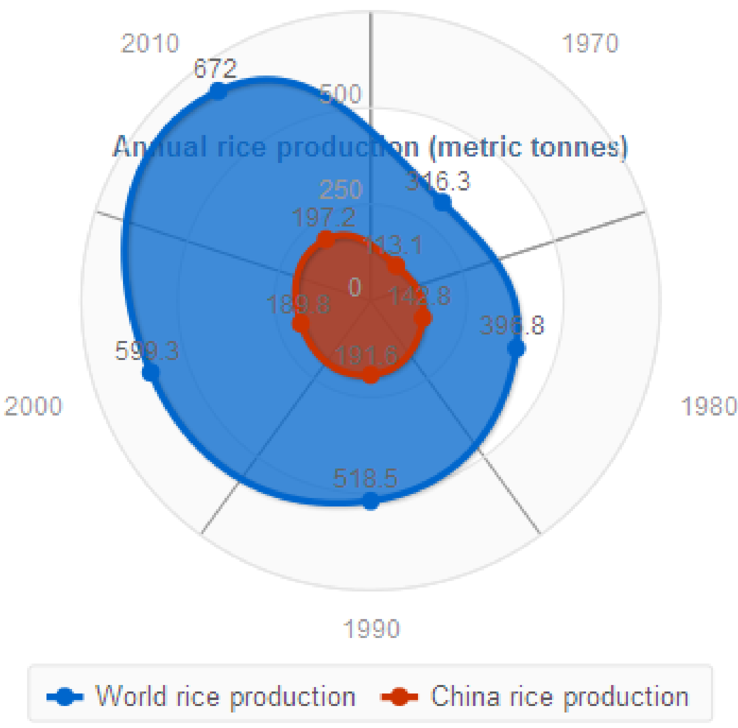


Figure 2.102 Cartesian and Polar charts with CategoryDataSeries ("World rice production" and "China rice production") and years as String values; the definition of the "World rice production" CategoryDataSeries (further below)

```
def Map<String, DataPoint> worldData :=
[
"1970"-> new DataPoint(value -> 316.3),
"1980"-> new DataPoint(value -> 396.8),
"1990"-> new DataPoint(value -> 518.5),
"2000"-> new DataPoint(value -> 599.3),
"2010"-> new DataPoint(value -> 672)
];

[
new CategoryDataSeries(
    label -> "World rice production",
    options -> new PlotOptionsArea(
        color -> "#0066CC",
        showLabels -> true,
        stacked -> false,
        marker -> Marker.circle,
        lineStyle -> LineStyle.solid,
        lineWidth -> 3,
        spline -> true,
        range -> null,
        opacity -> null),
    xAxisIndex -> null, yAxisIndex -> null, values -> worldData), ...
```

Figure 2.103 Cartesian and Polar charts with CategoryDataSeries ("World rice production" and "China rice production") and years as String values; the definition of the "World rice production" CategoryDataSeries (further below)

- **TimedDataSeries**: the x axis values are points in time.

Values are defined as a map of Dates and DataPoints(**Map**<**Date, DataPoint**>): the date is used as the value on the x axis and the data point defines the values on the y axis.

Figure 2.104 Cartesian chart with TimedDataSeries ("Train delay"); the definition of the "Train delay" TimedDataSeries (further below)

```
def Map<Date, DataPoint> trainDelay :=
[
    date(2012, 1,1)->new DataPoint(value -> 50),
...
];

[
    new TimedDataSeries(
        label -> "Train delay",
        options -> new PlotOptionsScatter(color -> "#0066CC",
        showLabels -> true,
        stacked -> false,
        marker -> Marker.circle),
    xAxisIndex -> null,
    yAxisIndex -> null,
    values -> trainDelay)
]
```

Figure 2.105 Cartesian chart with TimedDataSeries ("Train delay"); the definition of the "Train delay" TimedDataSeries (further below)

- **DecimalDataSeries**: the x axis values are decimal numbers.

Values are defined as a map of Decimals and DataPoints (**Map**<**Decimal, DataPoint**>): the decimal is used as the value on the x axis and the data point defines the values on the y axis.

2.7.4.3.9.1 Plotting Options

Plotting options define how a set of data renders. They are defined in the `options` property of every data series so that every data series in a chart can be plotted differently. If undefined, the default plotting properties for the given data series are applied.

The following plotting options are available:

- `PlotOptionsArea`: the data series is rendered as an area.
- `PlotOptionsBar`: the data series is rendered as a set of bars.
- `PlotOptionsBubble`: the data series is rendered as a bubble.

  Note: This plotting option is currently not supported.
- `PlotOptionsScatter`: the data series is rendered as a set of dots (scatter).
- `PlotOptionsLine`: the data series is rendered as a line.

```
[
    new TimedDataSeries(
        label -> "Train delay",
        options -> new PlotOptionsLine(
            color -> "#0066CC",
            showLabels -> true,
            stacked -> false,
            marker -> Marker.diamond,
            lineStyle -> LineStyle.solid,
            lineWidth -> 4,
            spline -> true),
    xAxisIndex -> null,
    yAxisIndex -> null,
    values -> trainDelay)
]
```

Figure 2.106 TimedDataSeries with a PlotOptionsLine definition

2.7.4.3.10 Browser Frame

The Browser Frame renders as a view to a URL. To adjust the size of the frame, use presentation hints (refer to the Standard Library).

Figure 2.107 Browser Frame to external page in a tab component

Browser Frame defines the URL property with the URL displayed in the frame.

2.7.4.3.11 Calendar

The Calendar ( ) component is rendered as a calendar with calendar entries. The calendar entries are clickable and can be dragged-and-dropped. You can also select the calendar mode in the calendar. These actions fire the respective events.

It produces events of the following types:

- *InitEvent* when the component is initialized or displayed if previously hidden
- *CalendarEditEvent* when the user clicks a calendar entry
- *CalendarRescheduleEvent* when the user drags a calendar entry
- *CalendarCreateEvent* when the user selects a time period by clicking and dragging

The calendar component is displayed in month mode by default. To display a day schedule, click the day date in the calendar cell. Note that if a calendar entry needs to be scheduled in the day mode and keep track of exact hours, the allDay property of the entry must be set to `false`.

- **Data**: closure that returns a set of business data for the calendar (the data contains information about individual calendar entries)

```
{
    x, y -> def Set<Object> result:={};
    foreach Note n in notes do
        if n.notetype = NoteType.MEETING
        then
          result:=add(result, n)
         end;
    end;
    result
}
```

- **To item**: closure that transforms the data from the set defined in the **Data** property to CalendarItem

```
{ mynote:Note -> new ui::CalendarItem(
                caption -> mynote.description,
                description -> "Imported MEETING note",
                from -> mynote.time.from,
                to -> mynote.time.to,
                allDay -> false,
                style -> null)}
```

- **Initial Date**: date that is selected in the calendar when first opened (By default, the current date is selected.)
- **Mode**: calendar display mode

  The property determines the way a calendar is displayed initially and after refresh. The possible values are `daily`, `weekly`, `monthly`.
- **Read only**: calendar renders as read-only and cannot be edited

  If read-only, CalendarRescheduleEvent, CalendarCreateEvent are not fired. CalendarEditEvent is fired to allow the form to display an event details.

2.7.4.3.12 Map Display

The Map Display component renders as an OpenStreetMap with the defined center location, zoom, and optionally also markers. You can drag the map to change the visualized area and zoom it in and out. The map markers can be draggable.

It produces events of the following types:

- InitEvent when the component is initialized or displayed if previously hidden
- MapClickedEvent when the user clicks the map
- MarkerClickEvent when the user clicks a map marker
- MarkerDraggedEvent when the user drags a marker

Map Properties

- **Center**: coordinates of the center of the rendered map

- **Zoom**: default zoom on initialization and refresh defined as an integer with value 0-18 (0 being the lowest zoom with the entire Earth displayed)
- **Markers**: set of business data that are used as map markers

```
{ [new Meeting(
          title -> "Whitestein Meeting",
          location -> whitesteinHeadquarters,
          can_reschedule -> false,
          attendees -> ["Vladimir", "Estragon"]
          ] }
```

- **To marker**: expression that converts the business data from Markers to MapMarker objects

```
{ x:Meeting -> new ui::MapMarker
(
  title -> x.title,
  location -> x.location,
  popup -> x.title + "<BR/>Atendees: " + x.attendees,
  draggable -> x.can_reschedule
  )
}
```

2.7.4.4 Action Components

Action components produce an action event when clicked.

2.7.4.4.1 Button

The *Button* component renders as a button. On click the button produces an action event. The component can have an ActionListener that defines how the event should be handled.

It produces the events of the following types:

- *InitEvent* when the component is initialized or displayed if previously hidden
- *ActionEvent* when the user clicks the component

Figure 2.108 Tab with the Submit note Action Button

The Button has the following properties:

- **Text**: text on the button
- **Disabled**: availability of the button
  If true, the button is grayed out and cannot be clicked.
- **Help text**: tooltip text

  Note: You can define the Help text on the Help Text tab in the Properties view.

2.7.4.4.2 Action Link

The Action Link component renders as a clickable link. On clicking the link produces an action event. The component can have an ActionListener that defines how the event should be handled. The handling typically involves navigation action.

It produces the events of the following types:

- *InitEvent* when the component is initialized or displayed if previously hidden
- *ActionEvent* when the user clicks the component

Figure 2.109 Tab with the Link to note Action Link

The Action Link has the following properties:

- **Text**: link text
- **Disabled**: whether the link is disabled (rendered as grayed out and not clickable when true)
- **Help text**: tooltip text

  Note: You can define the Help text on the Help Text tab in the Properties view.

2.7.4.5 Special Form Components

2.7.4.5.1 Message Form Component

The Message component displays validation messages of failed validators in the chosen Form location. It is useful if you do not want to display these in the components with failed validation.

2.7.4.5.2 Expression Form Component

The Expression component defines an expression that returns a component. The expression is evaluated when the screen context is created and cannot be recalculated later.

2.7.4.5.3 Reusable Form

Reusable Form component references a form: on runtime, the form is inserted into the tree of components and becomes a part of the form tree. The form is called and resolved when the screen context is created.

Note that if you want to work with events of such injected reused form in other form components or process events from other nodes inside the reused form, you will need to explicitly allow such event distribution.

2.7.4.5.4 Conditional Form Component

The Conditional component is a form component that defines the visibility of its child components: if the Visibility property evaluates to `false` and the parent component is visualized or refreshed then the child components are not displayed; the children do not exist at all. Therefore it is not possible to operate over the child components unless the Conditional component defines them as visible.

2.7.4.5.5 View Model Component

The *View Model* component serves to isolate changes on form data so that the changes can be applied at once or thrown away without compromising the underlying data later. The content of the view model exists in its own execution context on an evaluation level.

This is helpful, for example, when creating pop-ups: the user enters data into the popup without influencing the data in the rest of the form.

A View Model creates a new context on a higher evaluation level which overlays the current level:

- if you insert a view model into a form and it has no other view model as its parent, the view model creates a context on the first evaluation level.
- if you insert a view model into a form and it has a view model as one of its parent elements, the view model creates a context on the evaluation level overlaying the evaluation level of the parent view model.

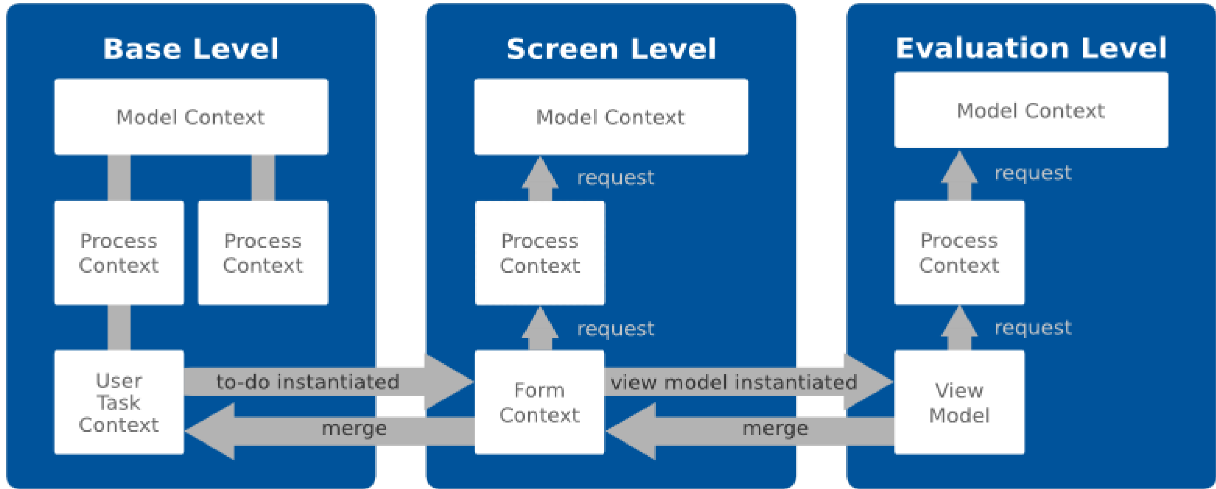


Figure 2.110 Execution levels and contexts in Model instances

When the user changes the data in components of a view model, the data on the level below remain unchanged until explicitly requested to be merged.

Note that there might be multiple evaluation contexts on the same evaluation context level.

You can either apply the changes to the context in the underlying level or discard them. The level to which the changes are applied is defined by the merge type of the view model:

- `MergeType.oneLevel`: the view model context is merged into the immediate underlying level
- `MergeType.screenLevel`: the view model context is merged into the screen level

For a view model use case, refer to Pop-up with Apply and Cancel Buttons.

2.7.4.5.5.1 Isolating Transient Data Using View Model

To isolate transient component data use the View Model component: The component provides a "commit mechanism" by creating a context on another evaluation level. A context on an evaluation level overlays the original context so the components inside the View Model work with data in their own space. You can use a view model, for example, to implement a cancel action when editing data: the user will edit the data in the View Model and the view model will be discarded or merged on a button click.

Generally you will proceed as follows:

1. Insert the *View Model* component into your Form. Make sure to define its name and merge type.
2. Into the View Model, insert Input Form components that will allow the user to modify the data.
3. In the View Model, create components with listeners that will merge or clear the data from your View Model: On the listener's Advanced tab:
   - To apply the data changes from a View Model, enter the View Model name to the Merge view model components property.
   - To discard the data change from a View Model, enter the View Model name to the Clear view model components.
   - You might want to define the View model init expression on the listeners: the expression is executed right after the merge or clear of view models.

Alternatively, you can call the *merge()* and *clear()* functions from the handle expression.

See Pop-up with Save and Cancel Buttons for example usage.

For example, let us assume the form below.

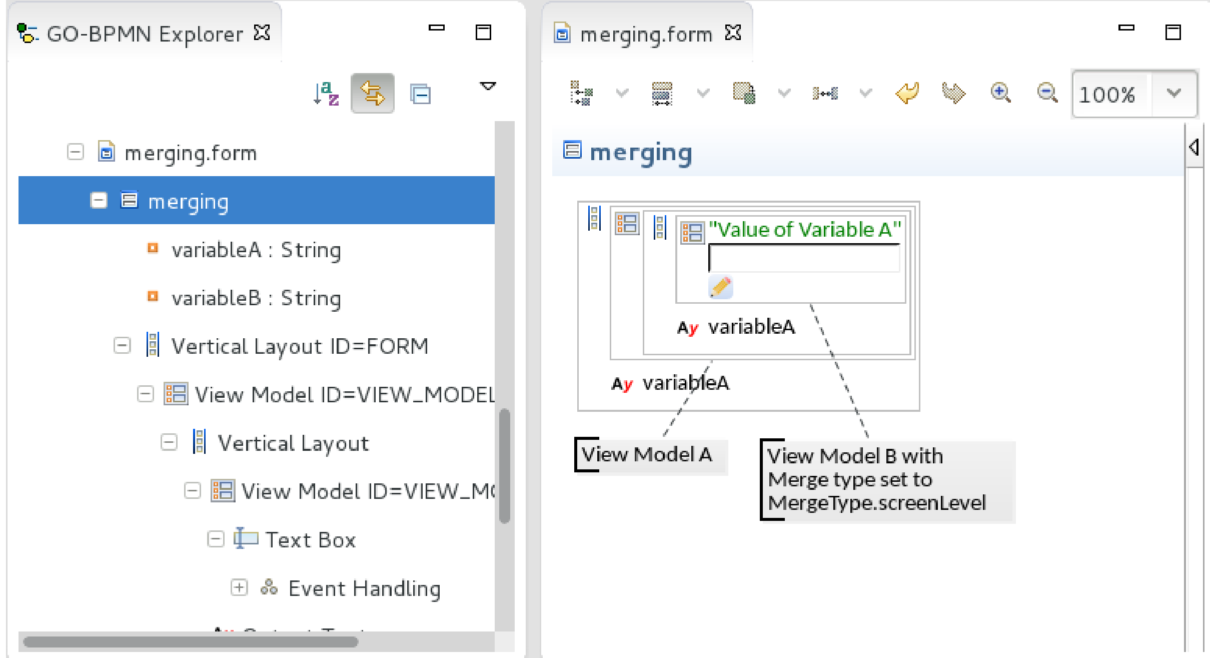


Figure 2.111 Form with multiple view models

Note the following:

- View Model B has the Merge type property set to `MergeType.screenLevel`
- The "Value of Variable A" TextBox is bound to variable A and has the Immediate property set to `true`.
- The ValueChangeListener refreshes the entire form and merges the View Model B (set in the listener properties).

On runtime, if you change the variable A value in the text box, the value will be merged to the screen context via the underlying evaluation level: hence also the context in of View Model A will have the change reflected and the displayed value will be updated.

For a view model use case, refer to Pop-up with Apply and Cancel Buttons.

2.7.4.5.6 Geolocator Component

The *Geolocator* component serves to acquire user's location. The location is detected on initialization and every component refresh. Note the component is not rendered in a form and is intended to provide input data for other components such as the map component.

The location is acquired either from the Wi-Fi or BTS location with accuracy from 300 to 3.000 meters, or from GPS with accuracy from 1 to 10 meters.

> **Note:** When a form with a Geolocator component is rendered, the browser asks the user whether he wants to enable the locating.

It produces events of the following types:

- *InitEvent* when the component is initialized or displayed if previously hidden
- **Geolocation event** when the Geolocator component acquires user's location

Note that as of the time of writing, Firefox version 24 and later do not support geolocation.

The Geolocator component has the following properties:

- **Detect**: enables or disables the location detection (If `false` the location is not detected on component refresh; this feature allows the user to disable the location detection, for example, via an action button.)
- **Position Options**: options of the location detection (for details, refer to *PositionOption* in `ui::components.datatypes`)

2.7.4.5.6.1 Acquiring Location

To acquire location of the user in your form, do the following:

1. Insert a *Geolocator* component into your form.
2. On the Geolocator component, create the GeolocationListener.
3. To work with the received location, in the Handle expression, handle the event's position property.

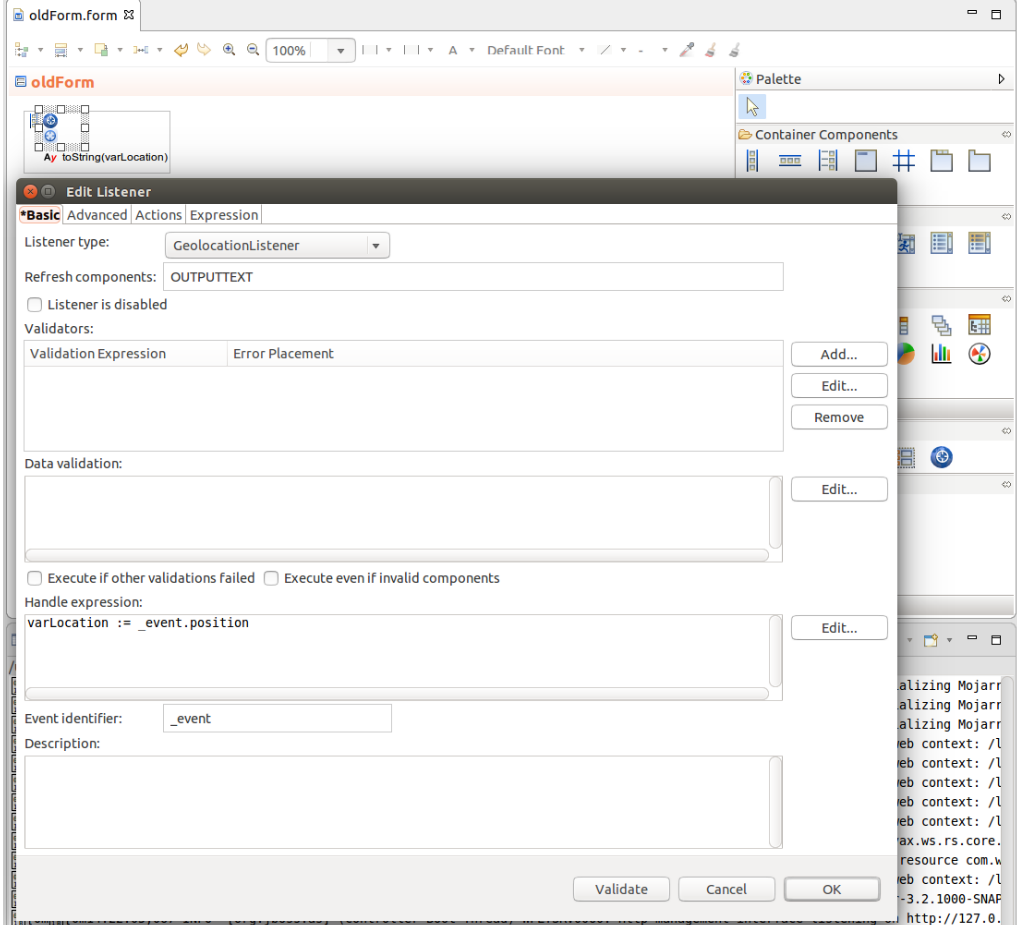


Figure 2.112 Handling of Geolocation event on the Geolocator component

2.7.4.6 Text Annotations and Associations

Annotation components serve to document the form: they display information about the and have no semantic value.

The Text annotation holds description or comments and can be connected to the related form component with a directed or non-directed association, which is a line, again, with no semantic value. You can use also a directed association, which is rendered as an arrow.

2.7.4.7 Deprecated Components

The components described in this section are deprecated and will be removed in the next release.

2.7.4.7.1 Tree

The Tree ( ) component renders as a tree of nodes. The nodes are expandable unless they are leaf nodes, that is, they do not have any children. Whenever a node is expanded its child nodes are lazy-loaded. To acquire the child nodes, the Tree component calls the closure defined in the Children property. The closure returns a List<TreeItem> objects.

The TreeItem objects define the following:

- label (String): arbitrary text displayed as the node label
- expanded (Boolean): whether the node is expanded by default
- data (Object): business object the tree item operates over

The Tree component serves as a picker of exactly one option (node). Once the user selects a node, the selection is stored in the Binding slot.

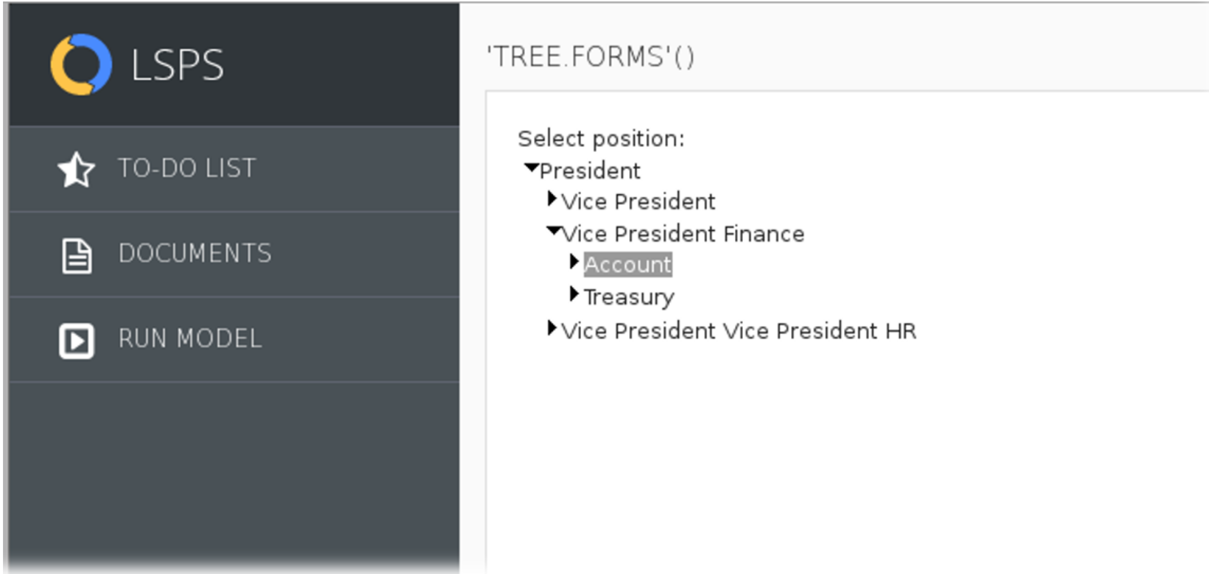


Figure 2.113 Tree with multiple nodes expanded

The Tree component has the following properties:

- **Label**: visible label
- **Required**: compulsoriness of the tree component

  If true, a mark indicating that the value is required is rendered in the component. > **Important:** The property **does not provide any validation**: the user > will be able to submit a null value even if it is set to `true`. To prevent > the submit, define a form validation on a listener.
- **Binding**: reference to a slot that holds the selected tree option value (for example, a form variable or global variable)
- **Children**: closure that returns a list of tree items which are displayed when a node is expanded

```
{ parent:Position, depth:Integer ->
    compact(
      collect(
        positions,
        { p:Position ->
          p.parent := parent ? new TreeItem (
            data ->p,
            expanded ->false,
            label -> p.name) : null
         }
      )
   )
}
```

- **Read-only**: editability

  If true, the tree renders as grayed out and no option can be selected. The selection uses the value of the binding slot.

- **Immediate**: setting of immediate mode

  If true, the Immediate mode is active: any value changes are processed immediately.

- **Help text**: tooltip text

  Note: You can define the Help text on the Help Text tab in the Properties view.

2.7.4.7.2 Tree Table

The Tree Table component renders as a table with rows organized in a tree structure with rows as tree nodes. Hence, child rows can be collapsed.

The component works similarly to the Table component. However while a table operates directly over business data types, the tree table component wraps the business data in the TreeTableItem data type objects. The object holds the along with business data also additional information. Based on the additional information, the tree table either expands the node that represents the business data or keeps it collapsed.

The tree table component iterates through a List<TreeTableItem> object. The TreeTableItem objects content can be then further processed (for example, displayed or provided for editing) in the table's columns.

The TreeTableItem objects define the following:

- expanded (Boolean): whether the node is expanded by default
- data (Object): business object the tree table item operates over (business data you want to work with in the tree table)

Figure 2.114 Tree Table with multiple nodes expanded

The Tree Table component has the following properties:

- **ID**: component ID unique in the form (Only capital letters, digits, and underscores are allowed.)
- **Children**: closure that returns a list of tree table items that are displayed when a node is expanded
- **Iterator**: reference to an object of the business data type

  The object is used as iterator for the table tree items.

  > **Important:**The iterator is not of the TreeTableItem type but of the type of your business data↩ : the List<TreeTableItem> holds TreeTableItem objects, while the TreeTableItems objects hold the business objects in **data** and serve to provide the additional **expanded** property.

- **Help text**: tooltip text

  > **Note:** You can define the Help text on the Help Text tab in the Properties view.

2.7.5 Form Patterns

This chapter contains patterns with happy-flow examples.

2.7.5.1 Editable Table

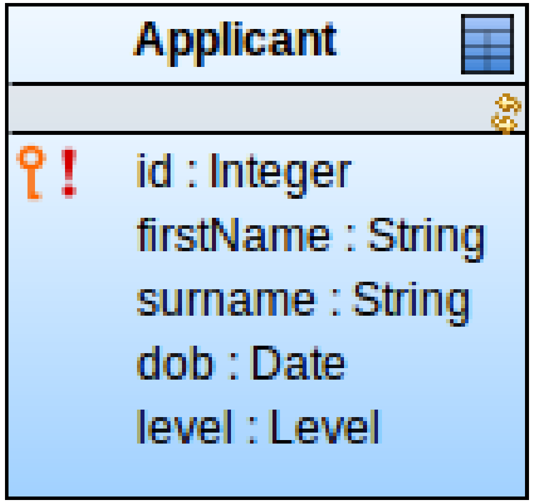
Required result:

- Table with columns with editable values.
- One of the columns contains a drop-down list with the possible options. The options are based on an enumeration.
- The table values are persisted when you click the **Submit** button.

Figure 2.115 Resulting form

To create a document or a to-do with such a table, you need to do the following:

1. Create the data type model with a shared record for the persisted entity and the enumeration.



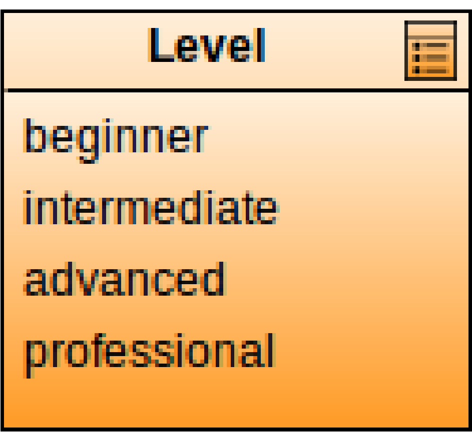


Figure 2.116 The underlying data type hierarchy

2. Create the form definition.

   (a) Create a form variable of the shared record type (Applicant): The table will use the variable as its iterator.

   (b) In the form, add the Table component and define its properties on the Detail tab:

      i. Set **Data Iterator** as the reference to the form variable.

      ii. Set Data Kind.

      iii. Define the Data expression.

   In this pattern, we assume you are using the Data Kind set to Data with the Data expression defined as a closure with two input parameters:`{x, y -> getAllApplicants()}`

   (c) In the table component, insert the Table Column components.

   (d) In the columns, add the Input components.

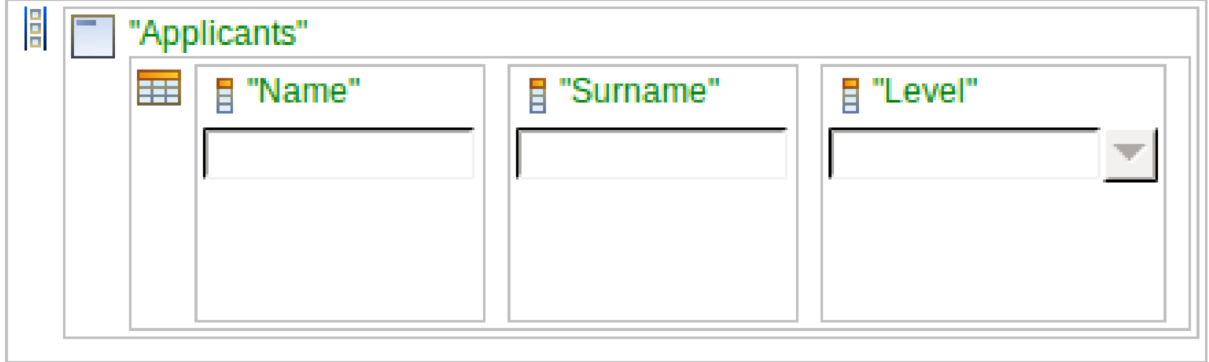


Figure 2.117 Asset table with columns with two text boxes and one combo box

In the example, we inserted two Text Boxes and one Combo Box:

i. On Text Boxes, define the binding to the reference to the iterator fields, for example, `&i.surname`.
ii. On the Combo Box component, define the binding to the iterator field and the options to be displayed in the drop-down area.

To bind options to the enumeration, convert the enumeration literals to options. You can do so using the `collect()` and `literalToName()` functions.

```
collect(literals(type(Level)),
       {e -> new ui::Option(value -> e,
                             label -> literalToName(e))})
```

(e) Define the Submit button:

i. Insert the Button component into the form.
ii. Create ActionListener on it.
iii. On the listener properties, select the Submit action on the Actions tab.

(f) Optionally, set the text that should be displayed in the table if it contains no entries: on the Presentation Hint tab of the table properties, add the `no-data-message` hint.

3. Create a document or a process with a to-do that uses the form.

2.7.5.2 Table with Derived Values

Required result: A table with a column with a value derived from another column value: One column value is persisted; the derived value is transient. The column values depend on each other and adapt to each other when either is changed.

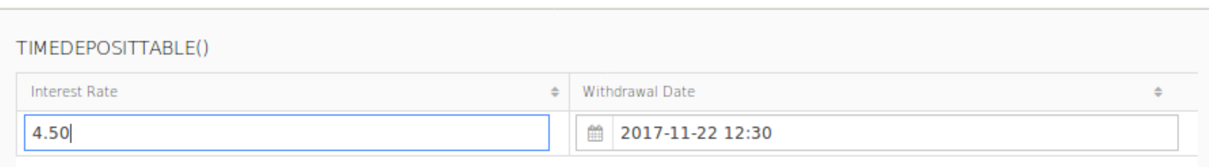


Figure 2.118 When you change Interest Rate, the Withdrawal Date changes. Withdrawal Date is not persisted.

1. Create the underlying data type hierarchy with the *base shared record* and a *non-shared record with fields for the derived values*:

   (a) Create or import the base shared record.

(b) Create a record with the derived field.

(c) Define an association between the records: the derived record is the target of the relationship.

Important: In such scenarios, you **cannot use the supertyping mechanism** since a shared record is involved:

- If you had a derived non-shared record that is the supertype of the base shared record, the derived record would include the fields of the base shared record but the shared record itself could not be recovered efficiently.
- If you decided to define the base shared record as the supertype of the wrapper non-shared record, whenever you decide to refresh the table with the record data, new shared record instances would be created and written in the database.

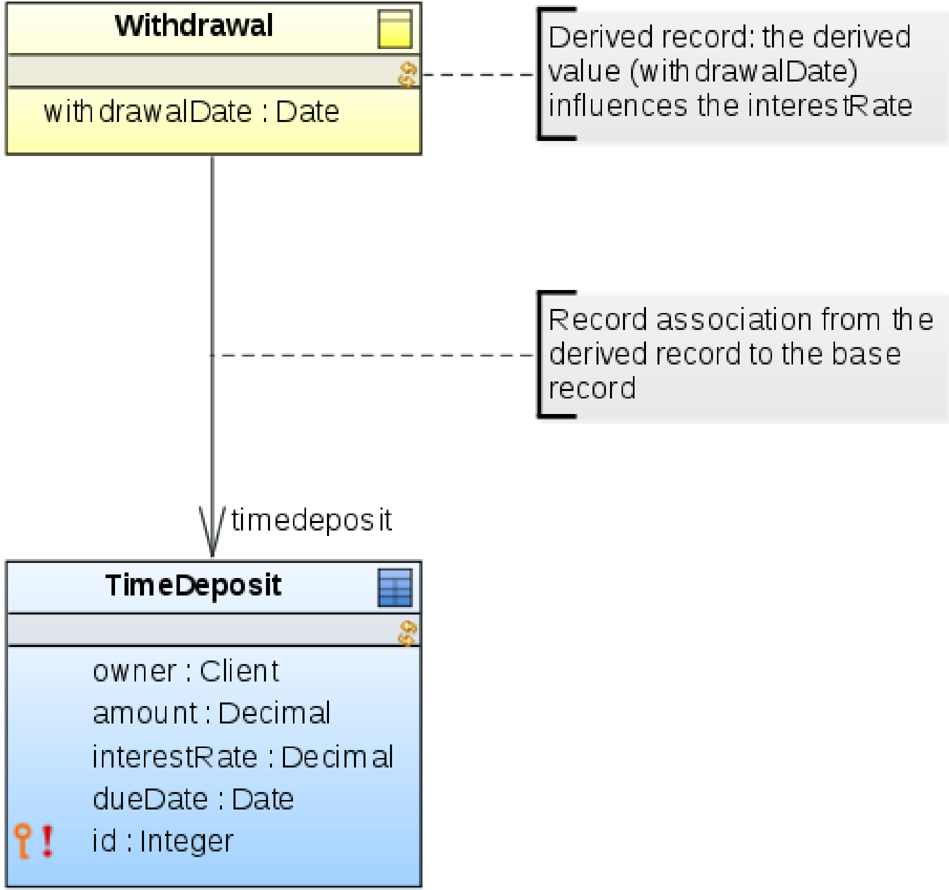


Figure 2.119 Base shared record TimeDeposit associated with the derived non-shared record Withdrawal

1. Create the form definition.

(a) Create a local variable of the derived record type.

The variable will serve as the iterator variable for the table.

(b) Create a local variable of the collection type with the derived records (for example, List<Withdrawal>), and initialize it so it holds the available Withdrawal object, for example, with the `collect()` function.

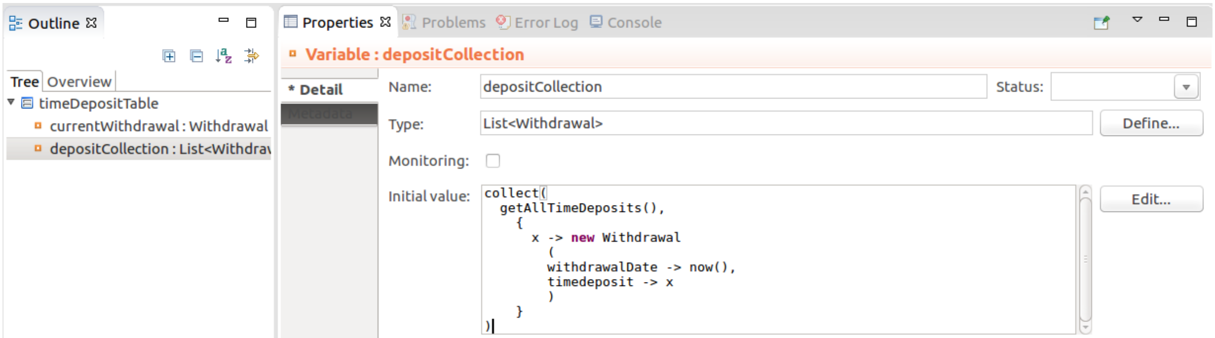
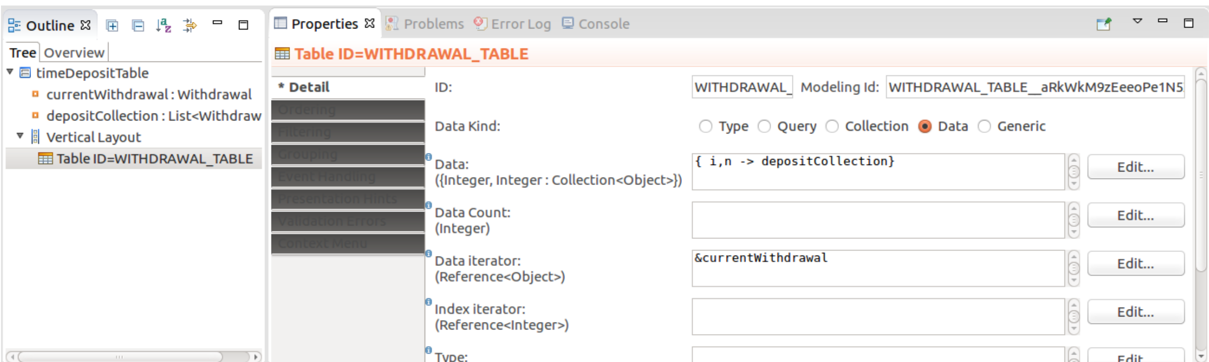


Figure 2.120 The collection form variable with the initial value

(c) In the form, insert the Table component and define its properties:

- *Data Kind* as **Data**
- *Data* as a closure that returns the local variable with data.
- *Data Iterator* as reference to the iterator variable



(d) In the table component, insert Table Column components and input components as their child components: define their ID and the binding of the input components to the respective field of the iteration variable (in the example, `¤tWithdrawal.timedeposit.interestRate` and `¤tWithdrawal.withdrawalDate`).

(e) On each input component define the following:

i. Create ValueChangeListeners: as the component to refresh, define the other input component and as Handle expression, define the new value of the iterator field, for example, using a function. Do not define the column as the component to be refreshed. Columns do not support the refresh action.

```
currentWithdrawal.withdrawalDate:= countWithdrawalDate(currentWithdrawal.timedeposit.interest)
```

ii. Set the Immediate property to `true` otherwise change of a value will not trigger change of the other value: the change would take place only after another event triggers the processing.

When set to `true`, the value changes are processed whenever the user clicks out of the input component or presses Enter.

2. Create a document or a process with a to-do that uses the form.

2.7.5.3 Calendar with Adding Entries Functionality

Required result: Form with a calendar into which you can add entries by selecting days in the calendar: entries details are defined in a pop-up dialog.

Do the following:

1. Create or import the shared record for your calendar entries.

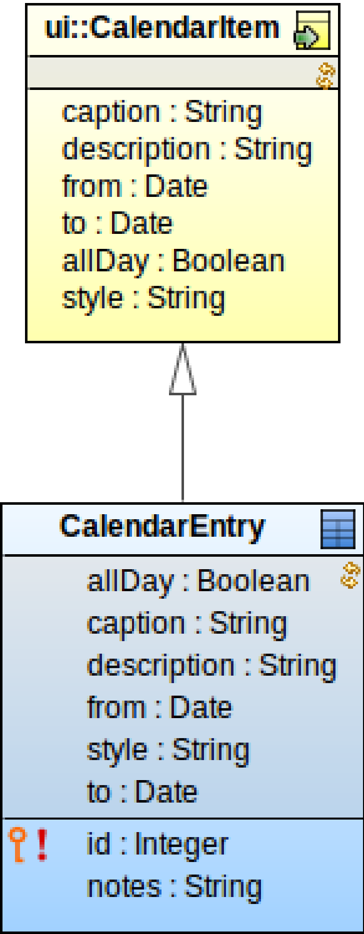


Figure 2.121 Shared record for calendar entries derived from the CalendarItem record

2. Create a form definition, open it and insert a Vertical Layout component.

3. Create a local variable of the calendar entry type.

   The variable will hold the data about a new calendar entry. For the example above, the variable will be of the `CalendarEntry` type.

4. Create the calendar:

   (a) Insert the Calendar component into the vertical layout.

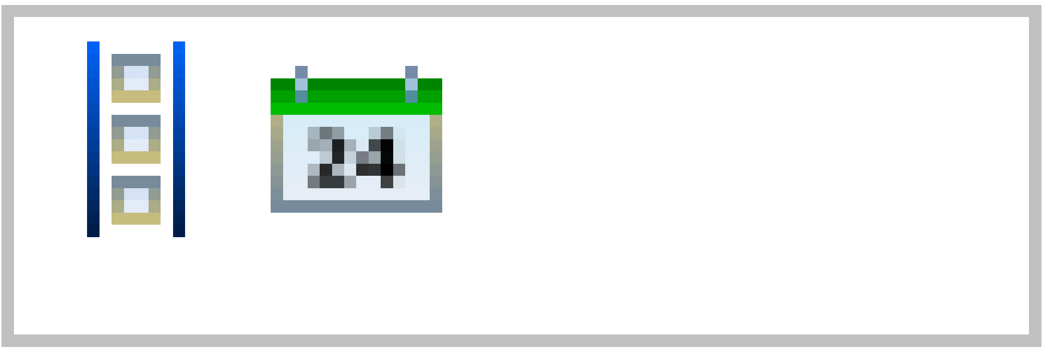


Figure 2.122 Vertical layout with calendar component

(b) Define the properties of the calendar:

- **Data**: closure that returns all calendar entries (The closure is called on calendar initialization and refresh: After you add a new calendar entry to the database, the calendar needs to be refreshed so as to load and render the new calendar entry.)

```
{ a, b -> (toSet(findAll(type(CustomCalendarItem)))) }
```

- **To item**: transformation of the data object to CalendarItem so the Calendar component knows how to display them; in this case, transformation of the *CalendarEntry* to *ui::CalendarItem*.

```
{ calItem:CalendarEntry -> new CalendarItem(caption -> calItem.caption, description -> calItem.description,
       from -> calItem.from, to -> calItem.to, allDay -> calItem.allDay, style -> calItem.style)}
```

5. Create the popup:

(a) In the form, insert the pop-up component and define its properties:

- **ID**: although component ID is not required, you will need it when displaying the pop-up (on button click, the visibility variable will be set to true the pop-up component will be refreshed so as to have it rendered).
- **Visible**: enter a name of a Boolean variable that holds the visibility of the popup.
  You can define a Boolean form variable; make sure to set its initial value to `false`.

(b) Nest the pop-up component in a *View Model*: right-click the popup and selects **Insert Parent** > **View Model**. Define its ID.

Note: The view model component isolates the data in the pop-up component from the data in the form context: it creates an evaluation context over the screen context. You will initialize the calendar entry variable when the pop-up is displayed and get the dates the user selects in the pop-up, all this will take place in the new evaluation context.
If you don't nest the pop-up in a view model component, the initialization of the variable will create a shared record with incomplete data in the screen context. When nested in the view model, the data is written into the screen context only after it is submitted or persisted by a listener.

6. Create the content of the popup: insert the *Form Layout* component and into it input components so the user to provide the other details for the *CalendarEntry*. Make sure the input components are bound to the correct field of the `CalendarEntry` variable.

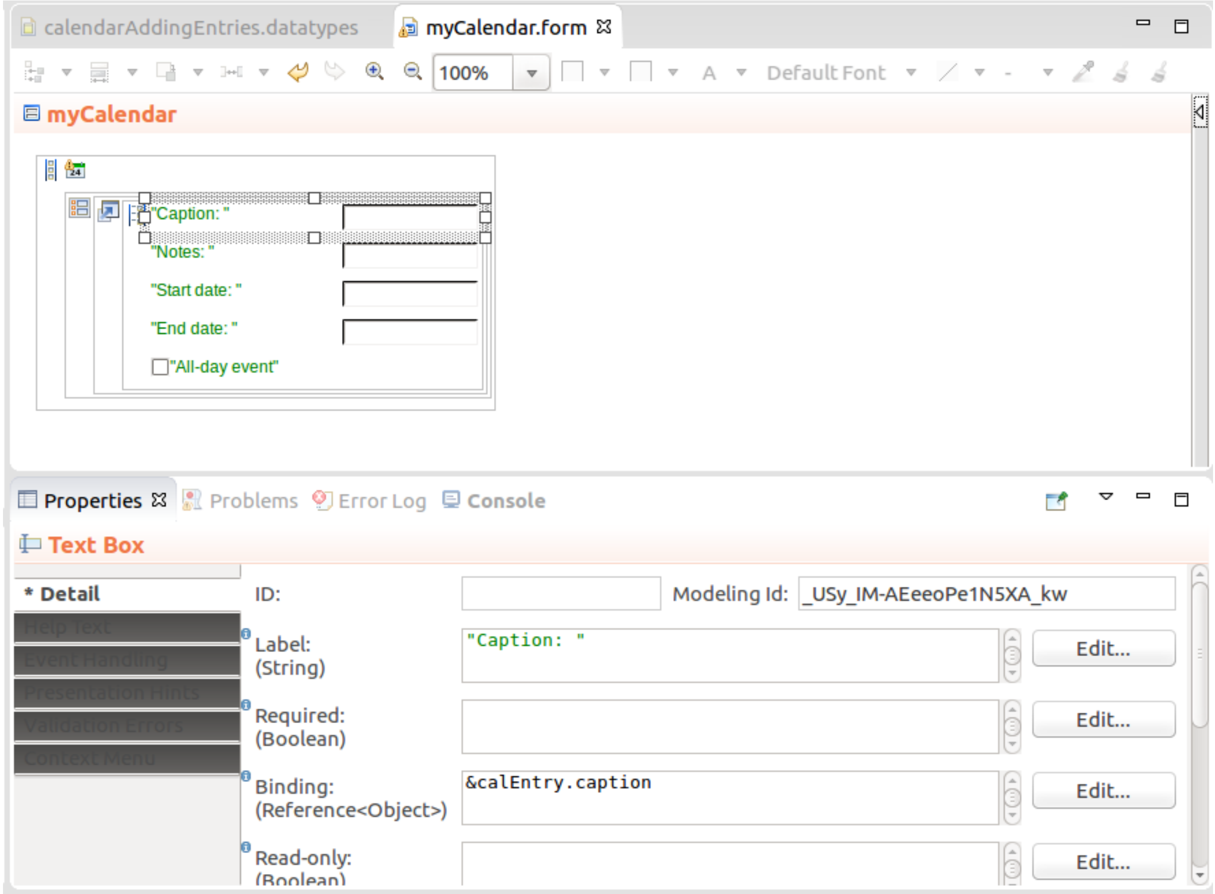


Figure 2.123 Calendar form

7. On the calendar component, create a *CalendarCreateListener* that will display the popup with the selected dates, when the user selects a time period by clicking and dragging:

   - Set its visibility to true and refresh it:
     - On the Basic tab, enter the **pop-up ID as the Refresh components** value.
     - On the Basic tab, define the **handle expression** so it sets the variable with the **pop-up visibility to true**.
   - Initialize calendar entry with the clicked dates: on the Basic tab, in the Handle expression, extract the dates from the event into the *CalendarEntry* variable:

```
calEntry:=new CalendarEntry(
        from -> _event.from,
        to -> _event.to
        )
```

8. Define the submit button in the pop-up that will persist the provided data and close the pop-up:

   - In the pop-up component of the form, insert the button component and define its properties.
   - Create the ActionListener on the button with the following:
     - Handle expression hides the pop-up.
     - Refresh the pop-up and the calendar.
     - Merge the view model (On the *Advanced* tab, enter the ID of the view model in the **Merge view model components** property)
     - Persist to save the new event in the database so it is picked up by the `findAll()` call on calendar refresh.

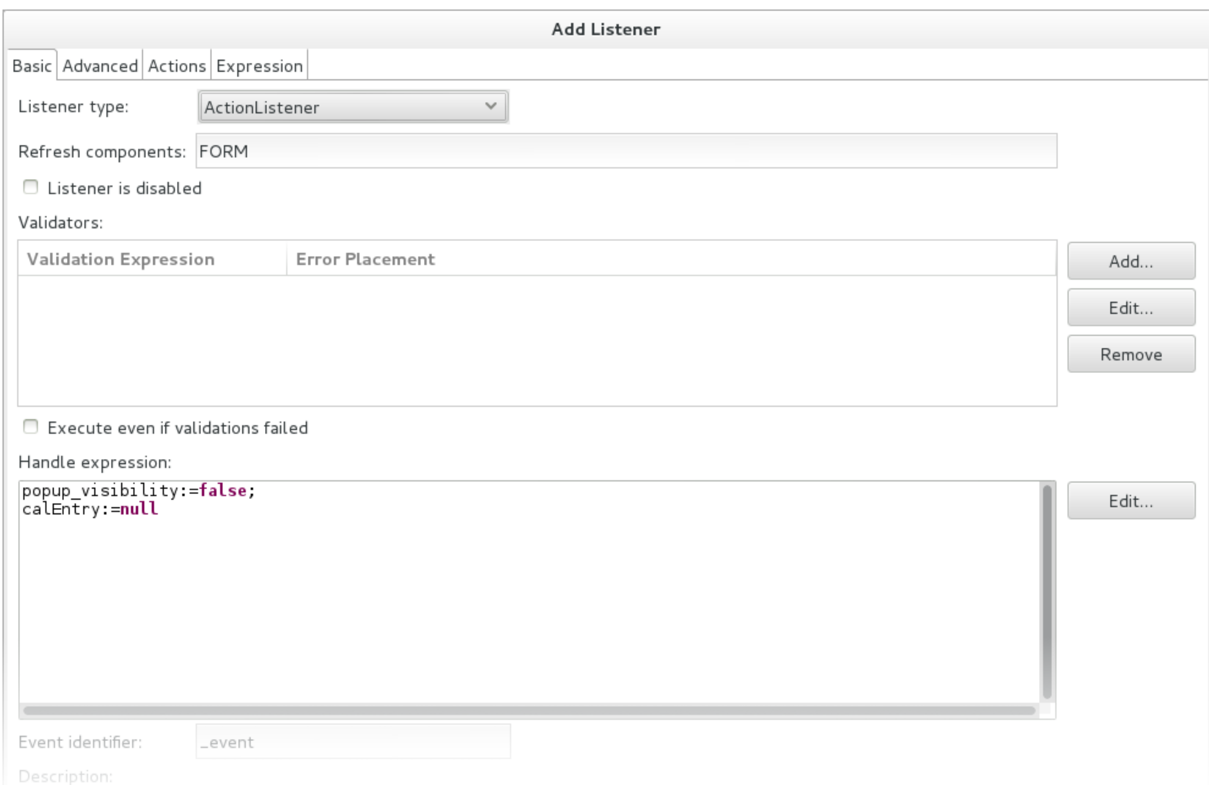


Figure 2.124 Listener on the submit button

2.7.5.4 Pop-up with Save and Cancel Buttons

Required result: When you click a button in your form, a popup where you can edit the form data is displayed. The pop-up contains an Save and a Cancel button. When you click the Save button, the data in the pop-up is applied to the data in the form and the pop-up closes. When you click Cancel, the data in the pop-up is discarded and the pop-up closes.

POPUP()
User Details
First name: John
Surname: Doe
Email: John@doe.com
Edit
Editing User Details
First name: Jane
Surname: Doe
Email: John@doe.com
Save
Cancel

To create a pop-up window with a Save and Cancel button, do the following:

1. Open the form with the data you want to edit in the popup.

   In the example, the data already exists and is stored in a form variable. If you want to create new data from the popup, make sure to initialize the data in the View Model we create in the next step.

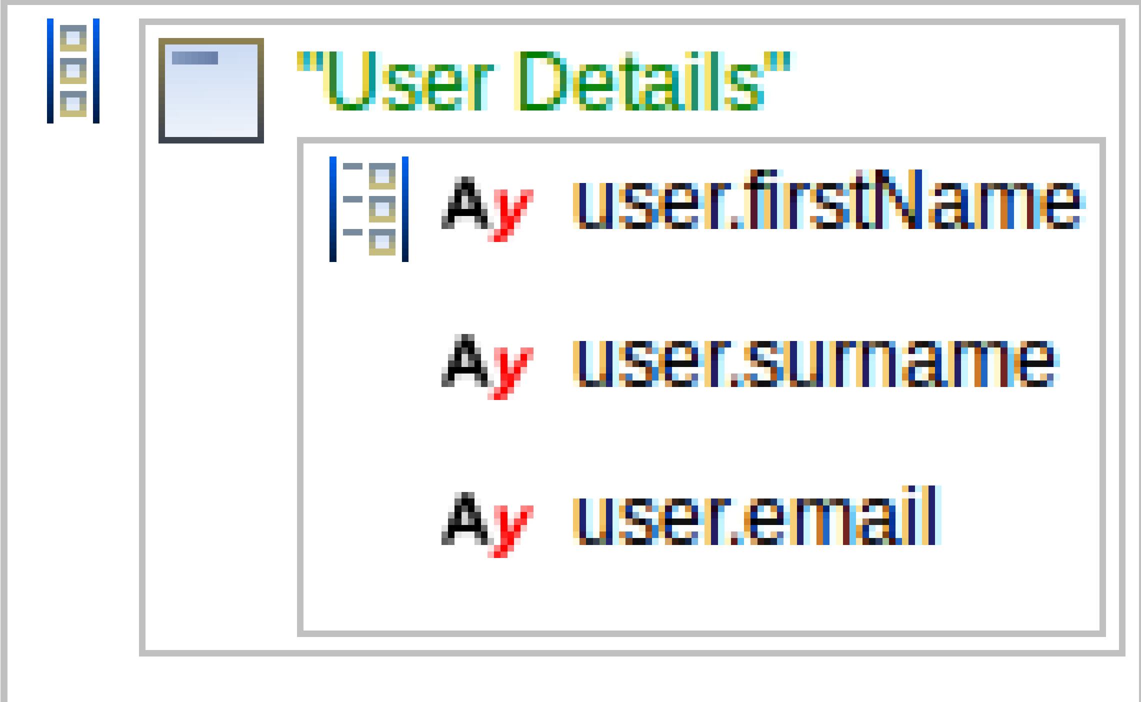


Figure 2.125 Form with user details

2. Insert the View Model component into the form and define its ID.

   The view model creates a context on an overlay execution level: this context will hold the differences to the form context on the screen level and will allow us to discard or save the differences in one step by clicking a button (for more details on how it works, refer to View Model Component).

3. Insert the Popup component into the View Model component.

   If you plan to create a complex component tree in the popup, consider using the dynamic popup to prevent performance issues: with dynamic popup the form is calculated and populated only when the popup is created, while the modeled popup is created when the form is initialized.

4. Set up the popup displaying:

   (a) Create a Boolean local variable with the initial value to `false` and set the variable as value in the **Visible** property of the popup.

   (b) Create the logic that will open the popup, for example, insert a Button with an ActionListener that sets the visibility variable to true and refreshes the popup.

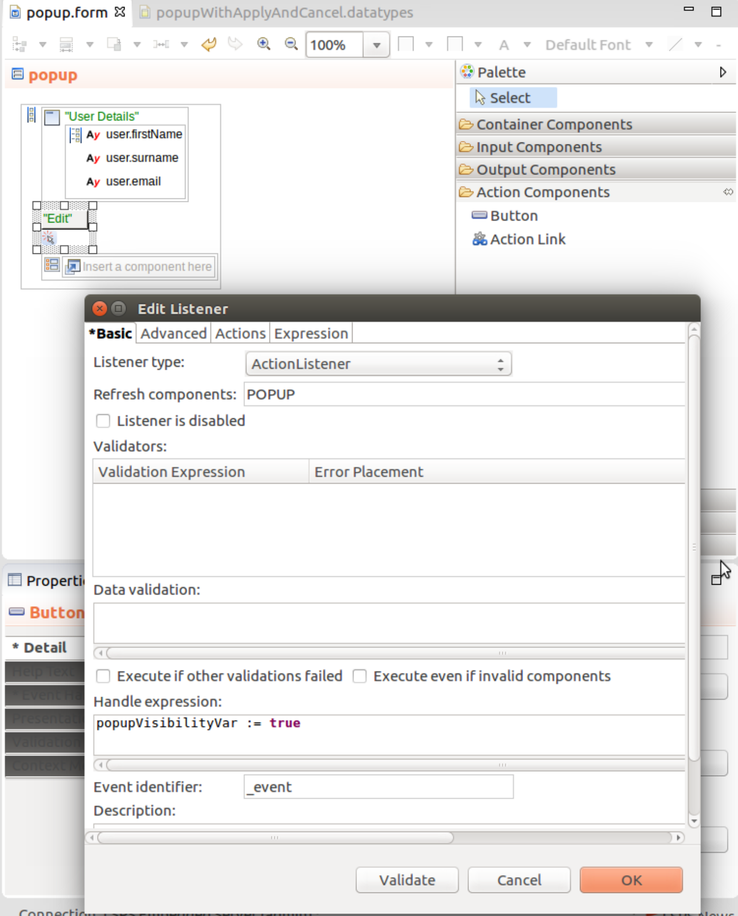


Figure 2.126 Setting Popup visibility for the *Save* button click

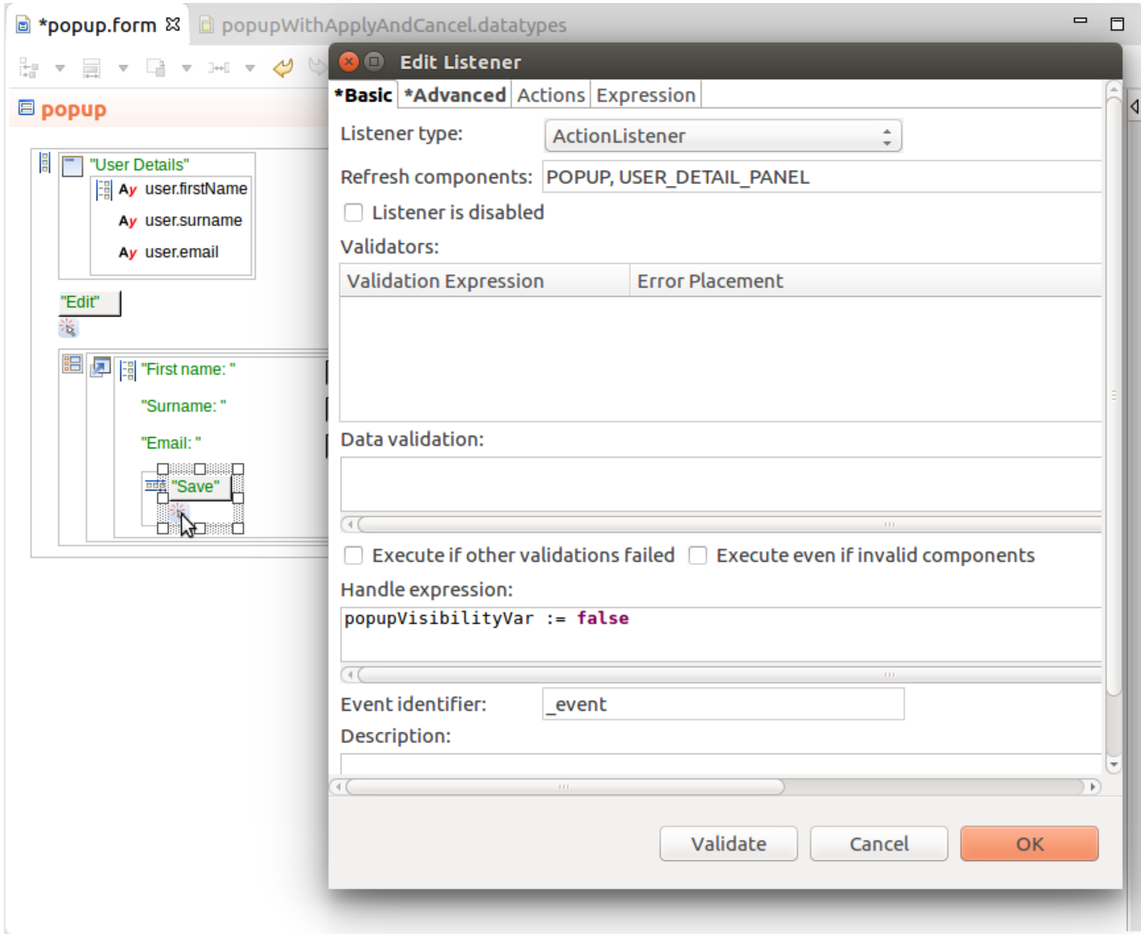
5. Create the popup content:

   (a) Insert a layout component and input components into the Popup component.

   (b) Bind input components to the local variable and define the labels.

   (c) Insert the Button component for the **Save** button and attach to it an ActionListener that will execute the following:

      - Merge the changed data to the form context: On the *Advanced* tab in the **Merge view model components** property, insert the ID of your view model.
      - Close the popup: on the Basic tab in the Handle expression, set the popup visibility to `false` and in the Refresh components, insert the ID of the popup component.
      - Refresh the data in the form (outside of the view model): in the Refresh components, insert the IDs of the components.



(d) Insert the Button component for the **Cancel** button and attach to it an ActionListener that will execute the following:

- Close the popup: on the Basic tab in the Handle expression, set the popup visibility to `false` and in the Refresh components, insert the ID of the popup component.
- Discard the changes in the View Model: On the Advanced tab in the *Clear view model components* property, enter the name of your view model.
- Set the listener to execute in the form context: On the Advanced tab, set the *Execution context* property to **Top level**.

If left set to default, the listener would execute in the execution context created by the view model. Since we are discarding the data from the view model, the visibility setting would be discarded as well and the popup would remain open.

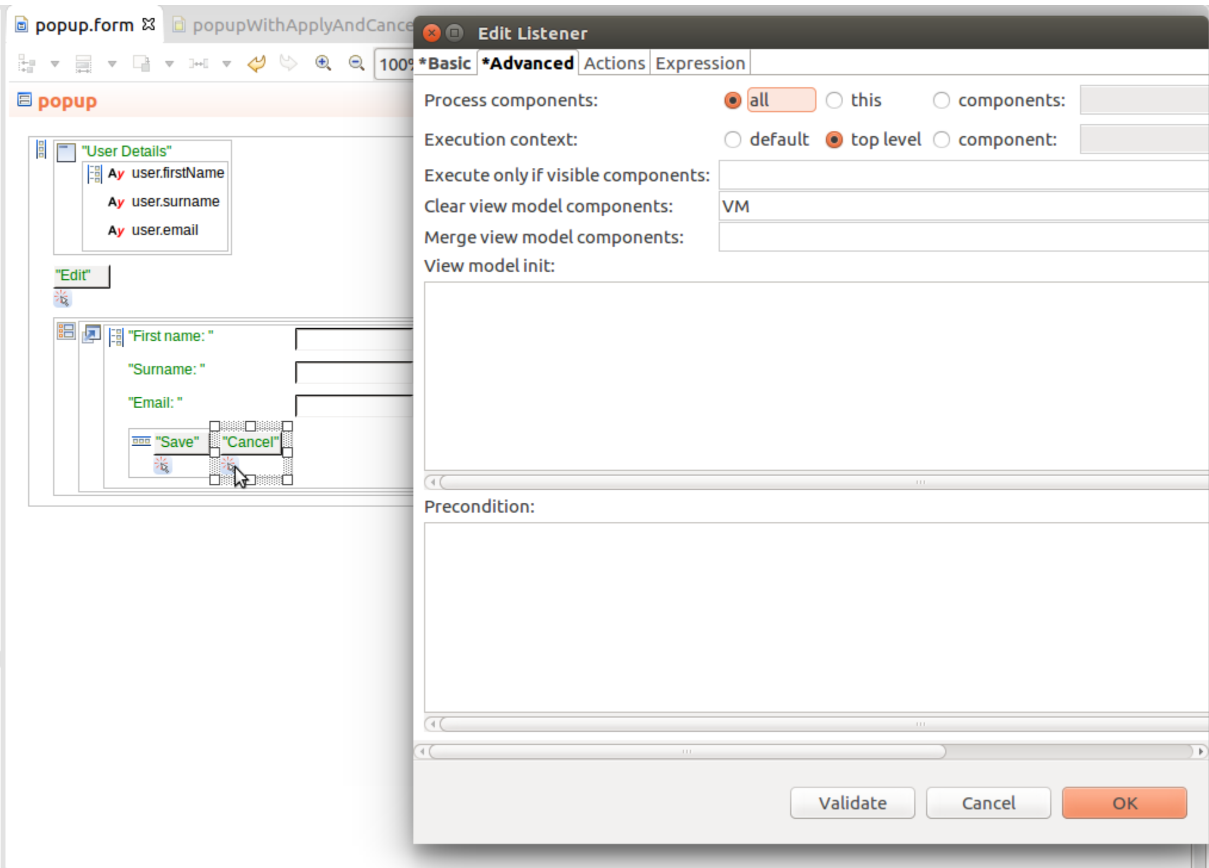


Figure 2.127 Setting cancel as View Model action for the Cancel button click

6. Run the Form Preview and check the functionality.

2.7.6 Enabling Error Reporting on Components

By default, runtime error reports do not contain information on which form component caused the error. Run your server with the `-Dcom.whitestein.lsps.vaadin.ui.debug=true` system property to include the modeling ID of your components in the reports (if you are running your server from PDS, go to **Runtime Connection** -> **Runtime Connection Settings**, select the Connection and click **Edit**).

You can then use the search to find the form component with the modeling ID: go to **Search**

Find Form Component.

2.8 Localization

The *LSPS Application User Interface* supports switching of locales so that the GUI can be displayed in another language. To display also the texts that originate from your Models in the correct language, use localization identifiers in the underlying expressions: Localization identifiers are special functions that return the localization of the string in the current locale. They support position text parameters so you can pass values to the localization text on runtime.

Note: In the Default Application User Interface, you can switch locales, on the *Settings* page. Note that the default application supports the English, German, and Slovak locale and localization identifiers in other languages will not be used unless you add a new locale to the application. For instructions on how to add a new locale to the application, refer to Software Development Kit Guide.

You can define a localization identifiers in the Localization Editor or from the Expression Editor.

2.8.1 Creating Localization Identifiers in the Localization Editor

To create a localization identifier with the Localization Editor, do the following:

1. From the *GO-BPMN Explorer*, create a localization definition (right-click a Module and select **New** > **Localization Definition**).

   Important: It is considered good practice to define localization definitions with all identifiers used by the given module to keep modules self-contained.

2. In the editor, select the Default Language in the *Default Language* combo box.

   Localization in this language will be used if the localization version requested by the application is not available.

3. In the *Localization Identifiers* area in the left part of the editor, click **Add**.

4. In the *Localization Details* area on the right, do the following:

   - Enter the identifier name into the *Identifier* field: you will use the name to use the identifier.
   - In the *Number of parameters* field, enter the number of position parameters the identifier will take.
   - Select the *Public* check box if you need to use the identifier in importing Modules.

5. In the *Translations* area, click **Add** and provide a translation of the string:

   (a) In the Language drop-down box, select the language.

   (b) In the Text area, type the translation: use the syntax %<position_number> to insert values of the parameters.

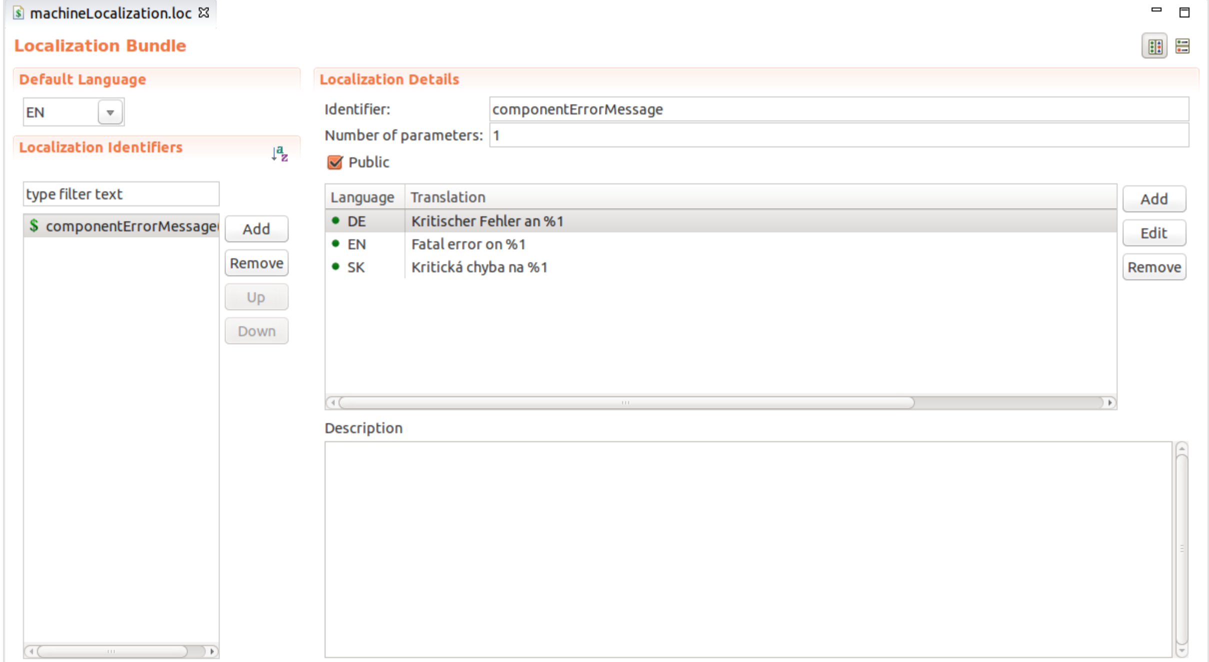


Figure 2.128 Identifier with parameter

2.8.2 Creating and Calling Localization Identifier in the Expression Editor

To define a localization identifier directly from the Expression Editor, do the following:

1. In the Expression Editor, type the name of the identifier as a String value and select it.

   To create an identifier with parameters, define the entire value with concatenation, such as, `"You requested the following book " + getBookName()` and select the entire expression: any concatenation elements that are not string literals will be interpreted as position parameters.

2. Press Shift + Alt + L.

   To change the shortcut, go to Window > Preferences, then General > Keys, search for Localize and set the new shortcut in the Binding field.

3. In the Localize dialog, define the identifier details.

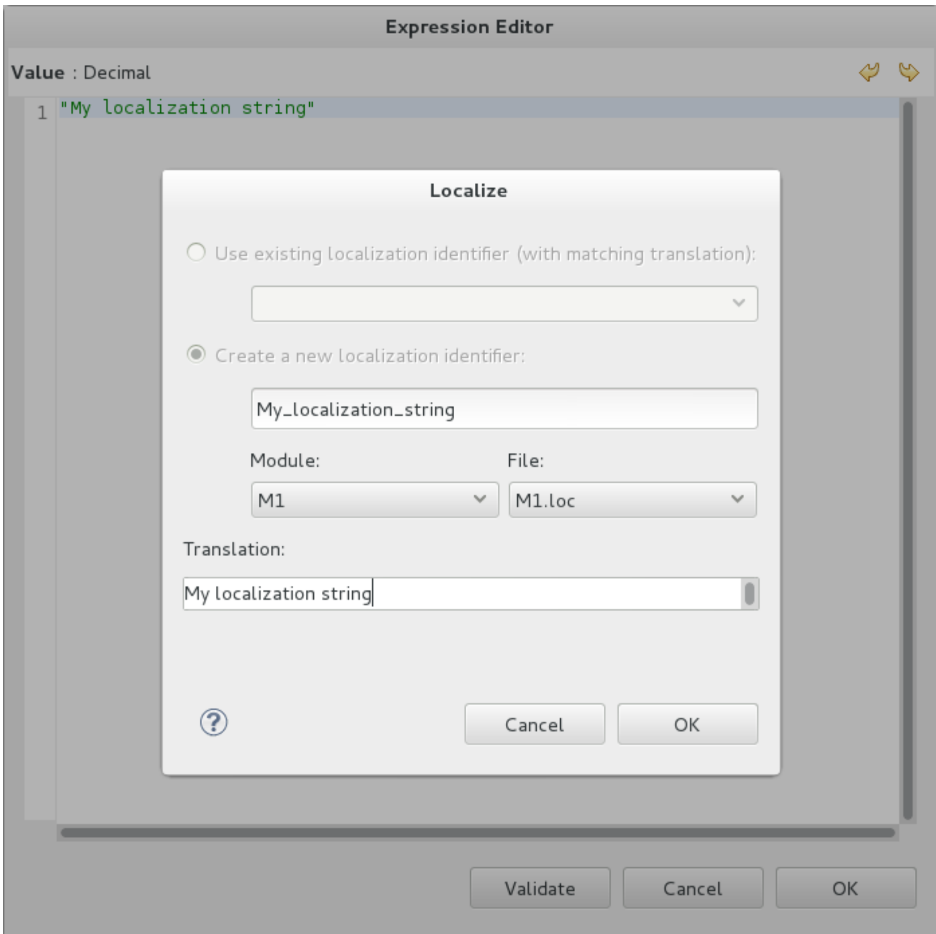


Figure 2.129 Localization dialog box with an identifier definition

Based on the data in the dialog box, the system creates a localization identifier in the selected localization definition with the translation and substitutes the string or the selected expression with a call of the identifier into the expression.

2.8.3 Calling Localization Identifiers

To call a localization identifier from your expressions, use the syntax:

```
$<identifier>(<comma-separated_arguments>)
```

For example, to call the identifier *deviceAdjust* with two parameters, enter `$deviceAdjust("#123", Sector A)`.

2.8.4 Searching for Usages of Localization Identifiers

To search for usages of a localization identifier, right-click the definition of the localization identifier in the Localization Editor and go to **Search For -> Usages** or select the identifier definition and press Ctrl+ALT+g.

2.8.5 Identifying Unlocalized Strings

If you are using localization identifiers and you want to make sure that everything is localized, you can set the validation feature to detect unlocalized Strings. Note that you can explicitly mark a String as non-localizable with the hashtag sign (#): such Strings are excluded from the validation check.

To have unlocalized Strings detected on validation, do the following:

1. Go to *Window* > *Preferences*.
2. In the *Preferences* dialog, open the nodes *Process Design Suite* > *Modeling* > *Validation*
3. On the right pane in the *Non-localized string* item, select the validation severity.
4. Set the *Ignore strings with no letters* option as required.

2.9 Functions

Function definitions are part of the module: however, only if they are implemented in the Expression Language is the implementation part of the Module as well. If you decide to implement your function in Java, you will need to upload the implementation to the LSPS server as part of you application.

The definition of functions are stored in Function Definition files.

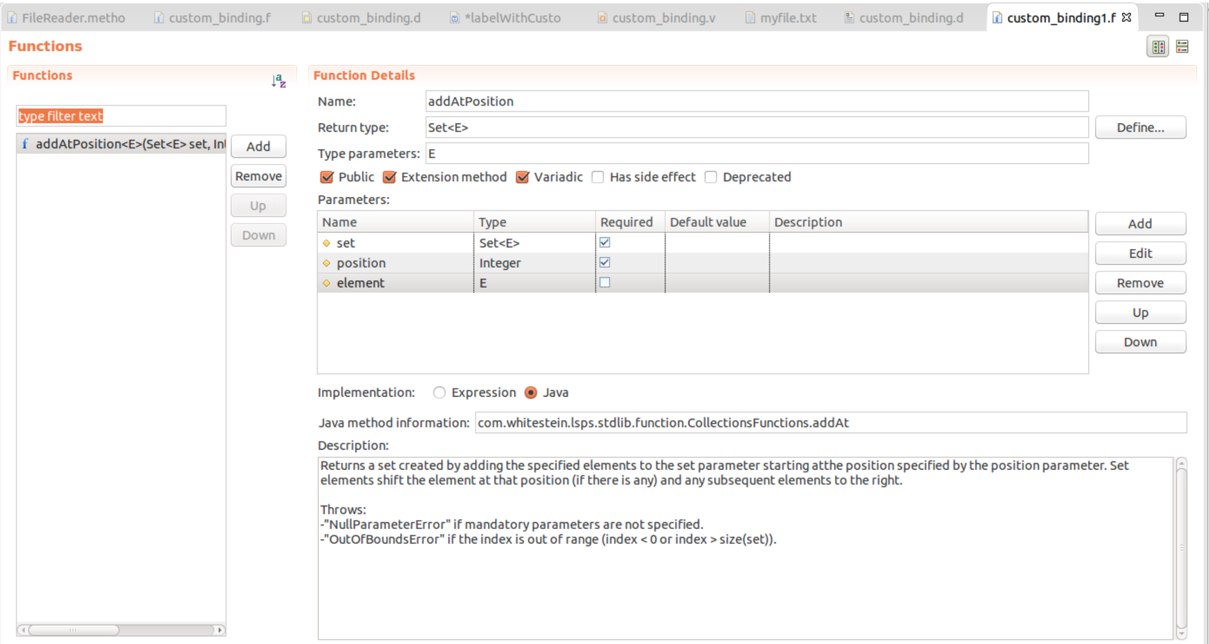


Figure 2.130 Function definition editor with a function definition file of the Standard Library

2.9.1 Defining Functions

To create a function, do the following:

1. In the Modeling perspective, create or open a function definition file.
2. Add a new function and define the function details:
   - **Name**: name used to call the function
   - **Return type**: data type of the return value
   - **Type parameters**: comma-separated list of abstractions of data types used in parameters

     Type parameters allow functions to operate over a parameter that can be of various data types, not only a single data type, in different calls. The concept is based on generics as used in Java. You can also make a generic data type extend another data type with the extends keyword. The syntax is then `<type_param_1> extends <type1>`, `<type_param_2> extends <type2>` (for details, refer to the Expression Language User Guide).
   - **Public**: function visibility
   - **Variadic**: function arity A function is variadic if its last parameter can be declared variadic, that is, zero or more occurrences of that last parameter are allowed.
   - **Has side effects**: if true, on validation, the info notification about the function having a side effect is suppressed A function is considered to have side effects if one of the following is true:
     - The function modifies a variable outside of the function scope.
     - The function creates a shared record.
     - The function modifies a record field.
     - The function calls a function that causes a side effect.
   - **Deprecated**: if true, on validation, a notification about that the called function is deprecated is displayed
3. Define the input parameters. For every parameter you need to define the following:
   - **Name**: parameter name unique within the function declaration
   - **Type**: data type of the parameter
   - **Required**: if checked, every function calls must define the parameter. The Required property does not provide any additional runtime check of the parameter value.
   - **Description**: optional description of the parameter Functions can be overloaded: functions with the same name but different parameters are considered different functions.
4. Define the implementation:
   - Select **Expression** and define the implementation.

     ```
       Name -> &newEmployee.name,
       Birthdate -> &newEmployee.birthdate,
       Salary -> "You need to set the salary",
       CurrentPosition -> getPosition(newEmployee);
     return true)
     ```
   - Select **Java** and enter the path to the method that implements the function, such as `org.example.example.TimeUtils.getWeekday`, and include the implementation in the ejb project of your LSPS Application. For further information on how to develop a custom LSPS Application, refer to the Software Development Kit Guide.
5. Enter the Description of the function in the Description field.

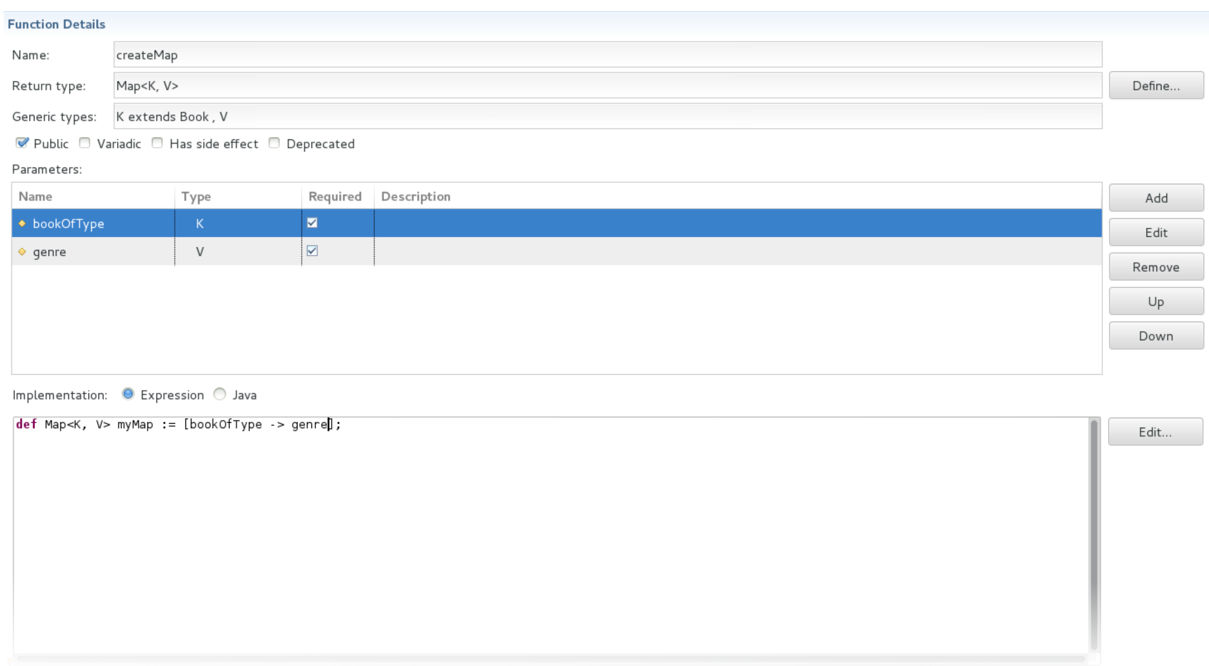


Figure 2.131 Function definition

2.9.2 Calling Functions

Functions are called by their name followed by function parameters in brackets:

```
<FUNCTION_NAME>(<FUNCTION_PARAM1>, <FUNCTION_PARAM2>)
```

2.10 Queries

Queries serve to request data from the underlying LSPS database or an external database.

You can define two types of queries:

- standard queries query the data of shared records
- native queries query data with database statements

Both query types are defined in the query definition file.

2.10.1 Standard Queries

A *standard query* returns shared Records from the database. and is defined in the Expression Language. It is simple to define but its possibilities are restricted.

A standard query defines the following properties:

- **Name**: name used when calling the query
- **Record type**: entries of the shared Record type retrieved from the database
- **Single value**: amount of the returned entries

  If true, only the first entity is returned. If false, all entities are returned.
- **Distinct**: whether to return only unique entities

  If enabled, identical entities result in a sole entity in the returned data.
- **Public**: availability of the query in Module imports.

  If not public, importing Modules cannot use the query.
- **Iterator**: object used to refer to the currently processed entity
- **Join**: joins to other tables via Record relationships
- **Parameter**: query parameters
- **Condition**: Boolean expression which must be true for the entity to be included in the result (equivalent to WHERE)
  - Only the following functions with the entity iterator as their parameter define their database statement equivalent: `toLowerCase()`, `toUpperCase()`, `trim()`, `length()`, `substring()`: no other functions can be used in the query condition:
    - `getDayOfMonth(book.published)=5` as query condition is invalid since the `getDayOfMonth()` function does not define its translation to SQL.
    - `toLowerCase(b.Author)=="joseph heller"` is a valid query condition since the toLowerCase() function does define its translation to SQL.
  - You can use the `*` and `?` wildcards. Note that it is not possible to escape these characters. If required, consider using Native Queries instead.

    Important: the condition is interpreted into SQL and the interpretation does not fully correspond to its Expression Language interpretation mainly due to the fact that the `null` value is not considered a legitimate value in databases. Hence, `!= null` is interpreted as `is not null` and `== null` is interpreted as `is null`. For example, if you use the condition `x != true` as a query condition, if x is null in the database, it will not be included in the results.
- **Join Todo List**: allows to create a join to a list of to-dos which are ALIVE

  If the shared Record has a relationship to the `human::Todo` record, you can use the join to get the to-dos related to your Record, typically, in the Condition. This mechanism makes up for the absence of the direct access to the to-do database tables.
  - *Query Todo Iterator*: object that holds the current To-Do
  - *Todo List Criteria*: criteria for the to-do entries

    ```
    new TodoListCriteria(
      person -> p,
      includeSubstituted -> true,
      includeAllocatedByOthers -> true
    )
    ```

- **Paging**: the query returns the number of entities defined by the Result size starting from the Start Index position.

  If the start index and result size are both null, the query returns all results: no paging is applied. If you keep track of the current start index, you can implement paging; you can, for example, save the current index in a variable and increment it on each request. Hence the following properties are required when paging the results:

  – *Start index*: index of the first entity returned in a result
  – *Result size*: size of the batch request (maximum number of entities returned by the query starting from the start index position)
  – *Generate count query*: if checked, the system generates a function that takes the same parameters as the query and returns the number of entities in the result set.

  By default, the count function uses the name of the query with the `_count` suffix. To set a custom name, enter it in the entry field next to the check box.

- **Dynamic Ordering and Static Ordering**:

  Sorting of the result set with the dynamic ordering definition evaluated on runtime and static ordering remaining unchanged.

- **Fetch joins**: initialize join entities with the entity's parents with a single select

  Fetch joins prevent performance problems that occur when every record is fetched with a separate database select.

2.10.1.1 Filtering Results in Standard Queries

You can filter the query results depending on the values in:

- the queried Record: define the filter condition in the condition of the query definitions
- the related Records: define a join in the query definition.

2.10.1.1.1 Defining a Join in Standard Queries

Joins in Queries enable filtering of the Record entries based on related Record data, such as, get Authors who wrote a Book in 1983: in this case you would use an inner join to author's book and check if a book was published in 1983.

You can use joins on Records that are connected to the returned Record with a Data Relationship. Make sure, the end pointing to the related Record is named. Note that joining on a query does not fetch the joined table: the query still returns only the Records of the query return type.

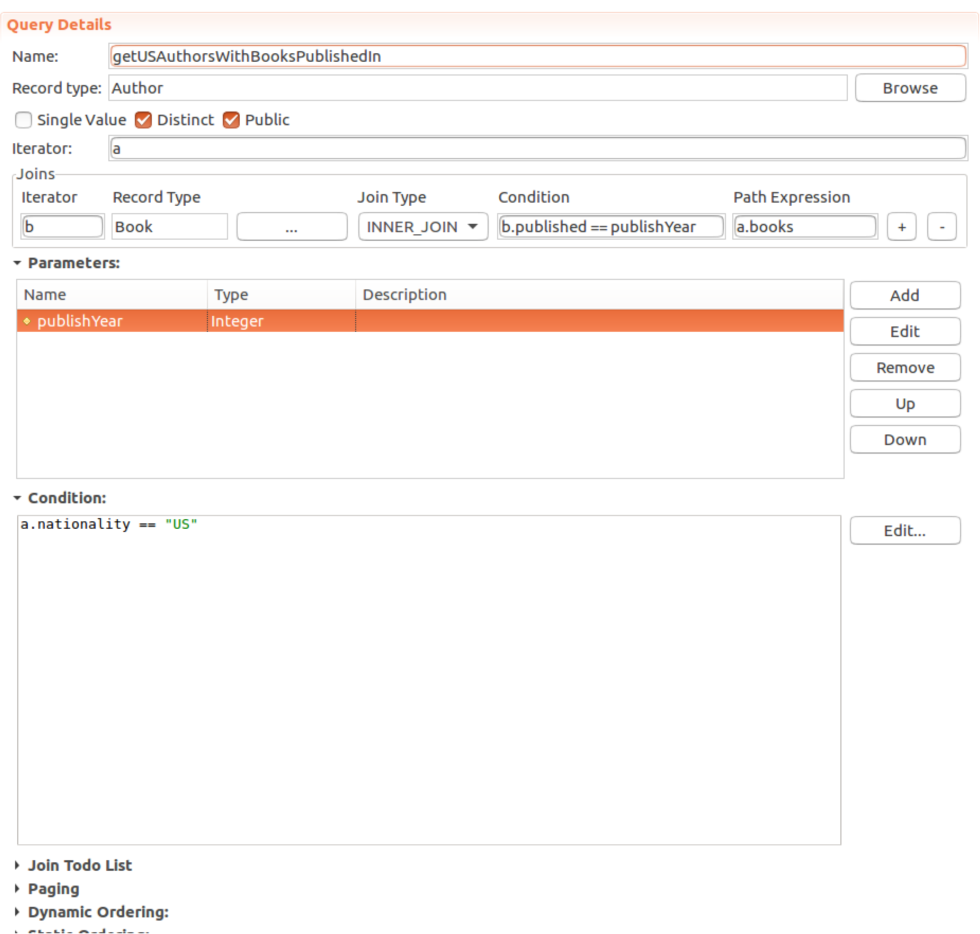
To define a join in your standard query, do the following:

1. Open the query definition.
2. Click *Add join* below the Iterator or the + button in an existing join.
3. Define the join properties:
   - **Iterator**: iterator name of the related Record
   - **Record type**: the Record type of the joined Record
   - **Join type**: the type of join

- **full**: all Records that meet the conditions
  The query will perform a cross join: it collects the Records and the joined Records, combines each Record with each joined Record, and applies the query condition on the results.
- **inner**: the query will return only the entities, which have values in all shared records (if any shared record entry is missing, the resulting entity is not returned)
- **outer**: the query will return any resulting entities with at least one missing value in any of the shared Records

  Note: Inner and outer joins are left joins.

- **Condition** (for path expression): condition applied on the join table entities
  The condition is useful when the join is an outer join, since it is checked on a smaller set of entities as opposed to being checked on all entities when defined as the query condition. Under these circumstances, the condition can improve performance.
- **Path expression**: path from the iterator Record to the Record for the join
  It must return a single instance of its type, or a list or set of instances of such a type.
  The path expression must start with one of the iterators, either a join iterator or a shared Record iterator.



2.10.1.2 Ordering in Standard Queries

To order the entities returned by a standard query, define the list of record properties used for ordering of the result entries: the entities are then ordered according to the values of the first property; if records contain the same value in the property, the second property is used for ordering, etc.

You can define ordering as dynamic or static:

- Dynamic ordering is based on an expression that is evaluated on runtime.
- Static ordering is based on a list of ordering paths.

If you define both the static and dynamic ordering, the static ordering takes precedence over the dynamic ordering: if you define static ordering of book records according to their author and dynamic ordering according to their title, the results will be ordered primarily based on their author and only on the next level ordered according to their title.

2.10.1.2.1 Defining Dynamic Ordering

Dynamic ordering defines an ordering expression which returns a list of order-enumeration values. It is evaluated for every query call.

The database query can return the records in a different order on different calls; for example, the ordering expression can use the query parameters, where the incoming parameter holds a list of ordering enumerations.

To define your query to return results ordered based on runtime data, do the following:

1. Create a query or open an existing query.
2. Expand **Dynamic Ordering**.
3. Optionally, define ordering enumeration with their ordering direction (you can then use the enumeration in the ordering expression):
   (a) In the Ordering enumeration name, define the name of the ordering enumeration.
   (b) In the table below, define the values of the ordering enumeration. Every enumeration value defines the following:
      - Name: name of the ordering value
      - Path: path to the record field that is used for ordering The path is defined as a path to the record field using the dot operator, that is, <ITERATOR_NAME>.<FIELD_NAME>, for example, book.↩ author. Every path must define its sort order as either ASC to sort the records in the ascending order or DESC to use the descending order.
      - Nulls ordering: the way the *null* values are ordered (*default*: as set in the database setting; *nulls first*: null values come before any other values; *nulls last*: null values come after any other values)
4. Define the **Ordering Expression**.

The expression can use the incoming query parameters, where the incoming parameter holds a list of ordering enumerations and must return a list of ordering enumerations.

Example ordering expression

```
//query parameter of List<Ordering_enumeration>:
queryParameter
//possible literal value:
//[OrderEnum.AssetCurASC]
```

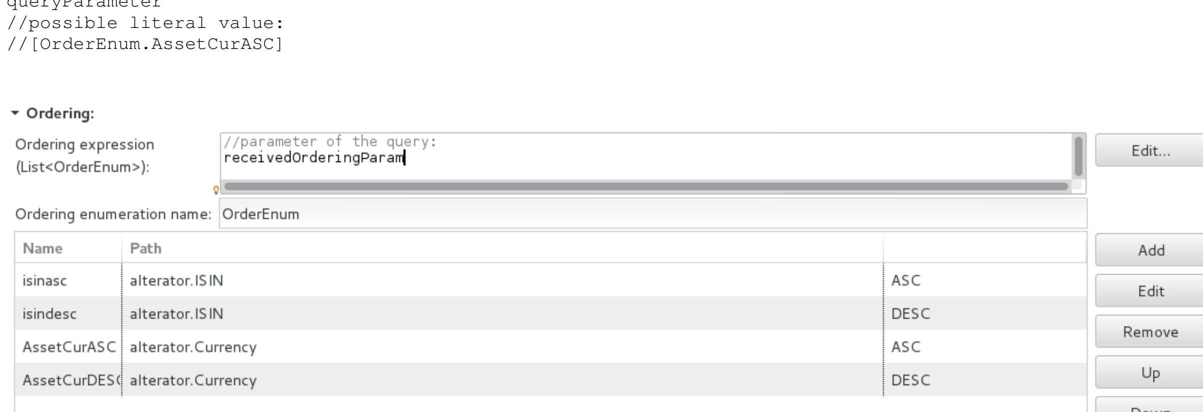


Figure 2.132 Dynamic ordering in a query

2.10.1.2.2 Defining Static Ordering

Static ordering of query output relies on a list of ordering paths: each query call uses the same paths for ordering.

To define static ordering of your query, do the following:

1. Create a query or open an existing query.
2. Expand the Static Ordering item.
3. In the Path table, define the ordering paths in the order you want to have them applied.

The ordering path must define the following:

- Path to the respective record field of the iterator in the form `<ITERATOR_NAME>.<FIELD_NAME>`, for example, `assetIterator.owner.email`
- Sort order as either ASC to sort in the ascending order or DESC to use the descending order according to the path field
- Nulls ordering: the way the *null* values are ordered (*default*: as set in the database setting; *nulls first*: null values come before any other values; *nulls last*: null values come after any other values)

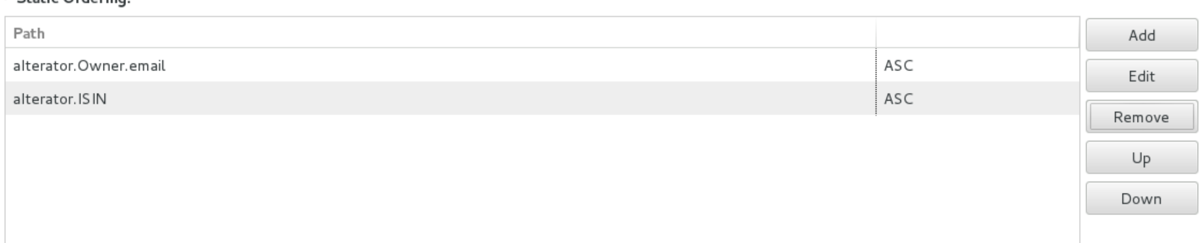


Figure 2.133 Static ordering query

2.10.1.3 Generating Queries for Shared Records

The mechanism for generating queries creates *standard queries* for shared Records of the Module.

For every shared Record, you can generate queries that return the following:

- all entries of the shared Record

  The queries are generated as `findAll<RECORD_NAME>()` queries.

- entries of the shared record with a particular ID

  The queries are generated as `find<RECORD_NAME>ById(ID)` queries. The ID Parameter has the data type of the primary key of the shared Record.

Note, that you can re-configure the name format on generation.

To generate the definitions of such queries perform the following steps:

1. Select the module containing the shared records.
2. Right-click and select Generate > Queries.

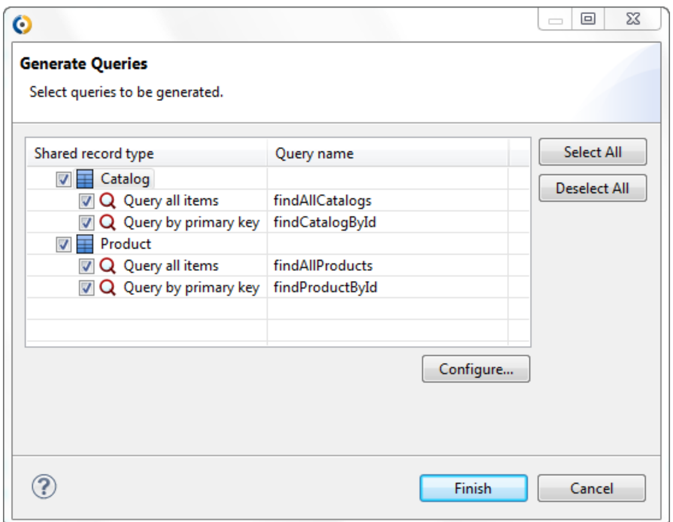


Figure 2.134 Generating queries

3. Select which queries you want to generate. To re-configure the default prefixes and suffixes of generated queries, press the Configure button.
4. Click **Finish**.

2.10.2 Native Queries

A *native query* is a named expression that defines an SQL database query. Note that, unlike standard queries, native queries do not rely upon shared records to query the underlying database. This allows you to make use of native database features and possibly secure better performance.

A native query is called from an expression in the same way as a function. When called, the query requests entities based on the defined query string. The results are stored in the defined Row type. If the query returns only the first entity, the entity is returned as an object of the row data type. If the query returns multiple entities, they are returned as a list of the row data type.

A native query defines the following properties:

- **Name**: name used to call the query
- **Result type**: type of the return value

  The type can be a primitive data type, such as a String, or a non-shared Record.
- **Single value**: amount of the returned values

  If true only the first returned value is provided as output; if false, all values are returned as a List of the return type. Note that these might be subject to paging.
- **Public**: availability of the query in importing Modules
- **Database**: JNDI name of the target database (if not defined, the default database is used)
- **Parameter**: query parameters

  Note: Parameter names must be valid identifiers unique within the query.

- **Mapping**: mapping of the fields of the Row type (define the mapping as a comma-separated list of the Row type fields; note that the order of the fields defines the mapping to the returned entity values).
- **Query**: expression resolved to a String that contains the SQL query

  Example: Mapping and query definitions

  – Mapping: `Currency, Price, ISIN`
  – Query:`"SELECT CURRENCY, PRICE, ASSET_ISIN FROM ASSET WHERE CU↩ RRENCY=:curr"`

  The CURRENCY will be mapped to Currency, PRICE to Price, and ASSET_ISIN to ISIN of the row type. Note that the order of the defined fields is preserved.

2.10.3 Calling Queries

To call a query, use the following syntax: `<query_name>(<parameters>)`

2.11 Data Type Model

A *data type model* comprises the hierarchy of all Records–user-defined data types with an inner data-type structure–with their relationships and inheritance in a Model.

They are defined in data type definition files of Modules using a graphical editor, the Data Type Editor.

Since a record holds your business data, you want to make sure that the data is valid. You can do so by defining constraints for your record.

> **Note:** Further information on Records, Relationships, and related elements and mechanisms is available in `BPMN Modeling Language Guide`. The instantiation of Records, accessing a Record instance Fields, and related mechanism are documented in the `Expression Language Guide`.

2.11.1 Creating a Record

To create a Record, do the following:

1. Open a data type definition file for editing or create one:
   - To create a data type definition file, right-click a Module and go to *New* > *Data Type Definition*.
   - To open an existing definition file, double-click the definition in the GO-BPMN Explorer.
2. Right-click into the canvas and in the context menu, select *Record*.
3. Select the inserted Record and define its basic properties on the *Detail* tab of the Properties view:
   - **Name**: name unique within the Module
   - **Supertype**: supertype Record of the Record
   - **Monitoring**: displays an icon to indicate that the Record is involved in `Monitoring`
   - **Shared**: shared record has its data persisted and hence is not dependent on the existence of its model instance
   - **Abstract**: An abstract record cannot be instantiated, but can be used as a supertype record of another Record.

- **Read-only**: if true, all Fields of the Record are read-only: The Record instance is initialized on model instantiation and can be deleted during runtime; however, neither the Record instance nor instances of its subtype can be modified during runtime.

  Read-only Records can only be targets of data relationships, but not their sources. This prevents a possible inconsistency of data.

- **System**: if true, the Record is read-only for the user

  A model instance cannot create instances of a system Record or calculate its Field values. However, it can be instantiated from your Application with the `createRecord()` on your context, possibly from a custom task or function implementation.

- **Final**: if true, the Record cannot be a supertype

- **Deprecated**: flag to denote a deprecated Record

  If the attribute is true, the validation displays an information message that the Record is deprecated.

4. Insert fields into the Record and define Relationships and their properties if applicable.

5. Optionally, define Metadata as key-value pairs on the Metadata tab.

6. Optionally, define XML Mapping.

2.11.2 Creating a Record Field

To create a Record Field in a Record, do the following:

1. Select the Record or its Field and press the Insert key to add a new Field.

2. On the canvas, select the Field and define its properties in the Properties view.

   - **Name**: name unique within the Record

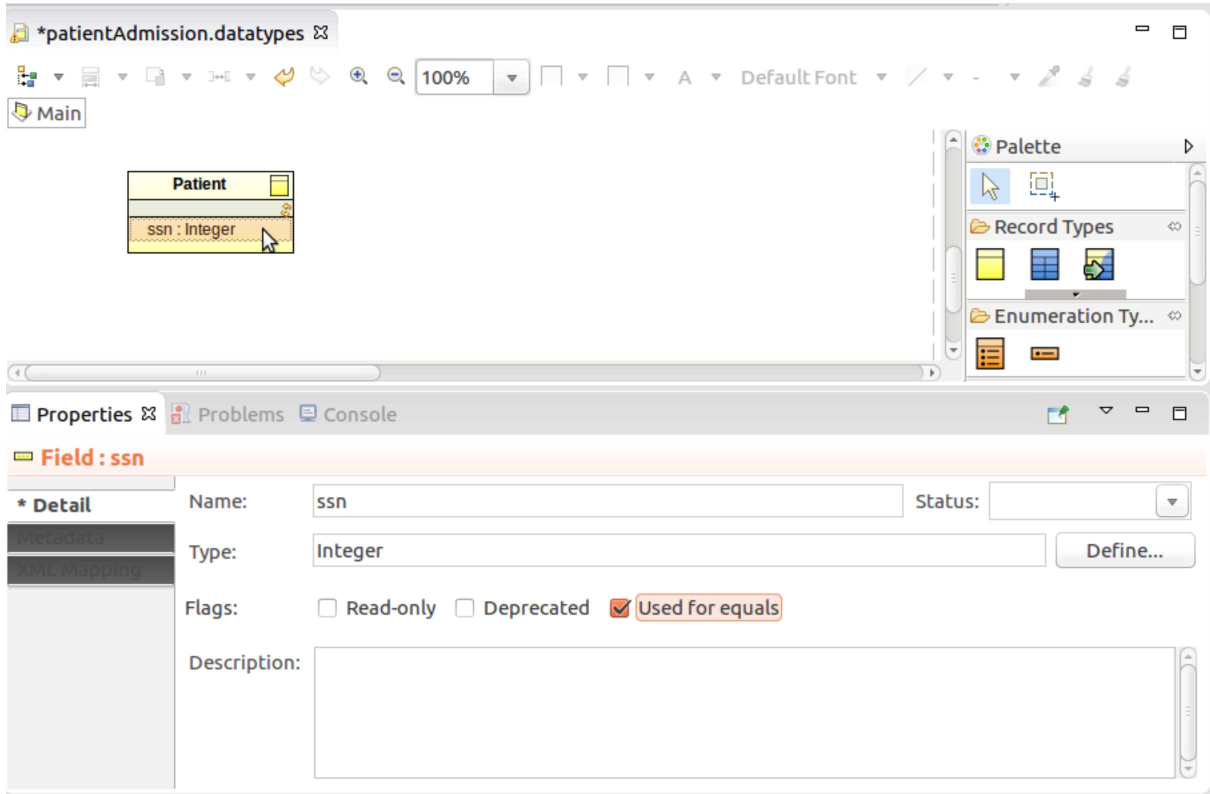
   - **Type**: data type of the record field

   - **Read-only**: if true, the field is read-only (instantiated on model instance instantiation; not modifiable after that)

   - **Deprecated**: flag to denote a deprecated Record field

     If the attribute is true, the validation displays an information message that the Field is deprecated.

   - **Used for equals**: if true, the field value is used as a comparison criterion. You can find further details in Using Fields in Record Comparisons



The additional tabs with properties serve for the following:

- **Metadata** define additional data as a map
- **XML Mapping** define the mapping used on export to XML when creating webservices, and by the `convertToXML()` and `parseXML()` functions.

2.11.3 Creating a Record Subtype

To define a Record's supertype, open its Properties view and on the Detail tab enter the name of the supertype Record. Alternatively, select the Inheritance connector in the palette and draw the connection on the canvas.

For more information on the inheritance mechanism, refer to the `GO-BPMN Modeling Language Guide`.

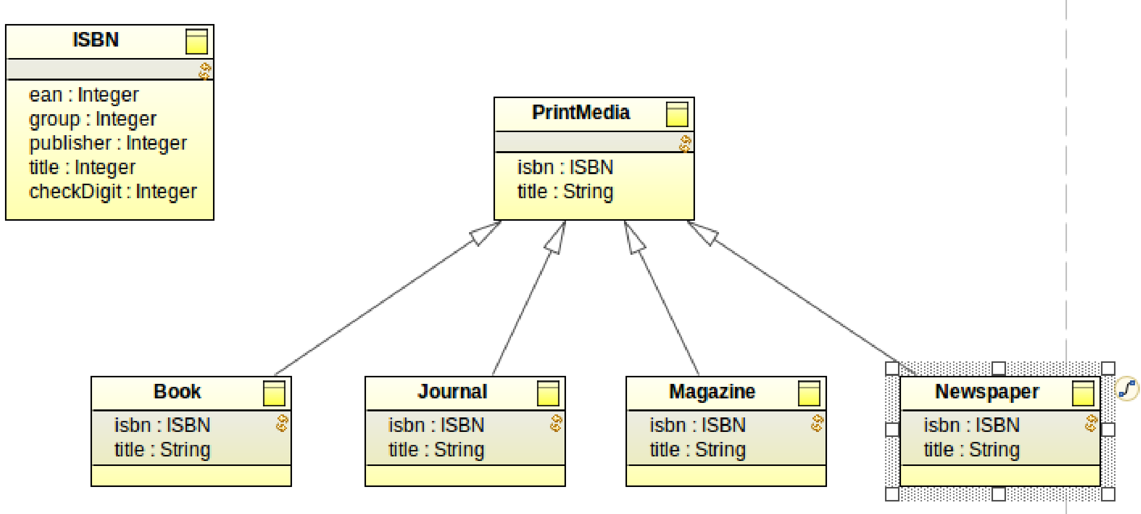


Figure 2.135 Hierarchy of Records with multiple subtypes

2.11.4 Creating a Record Relationship

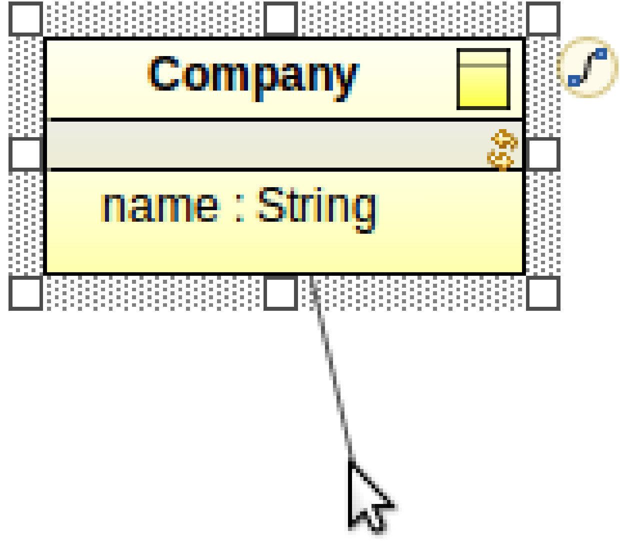
A Relationship serves to establish logical relation between Records. In the case of shared Records, it translates to relation between the Record database tables.

It is symmetrical, that is, set on both ends of the relationship. Therefore it is not necessary to make one of the Record the owner of the relationship.

Further information on the mechanisms, related elements, comparing of records and record properties is available in the `GO-BPMN Modeling Language Guide`.

To create a Relationship between two Records, do the following:

1. In the Data Type Editor, click the quicklinker icon and pull the link towards the target Record.



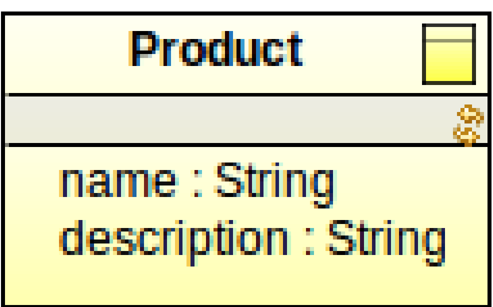


Figure 2.136 Creating Record Relationship

2. Select the Relationship and open its Properties view:

   (a) Define general properties of the Relationship on the Detail, Metadata, and Appearance tabs.

   (b) Define the properties of the Relationship ends on the Target and Source tabs.

   The ends that represent the Target and Source ends depend on the way you dragged the Relationship line: the Record you started from is considered the source. Information on the Relationship direction is at the top of the Properties view. Define the required properties of individual ends:

   - **Name**: name of the Relationship end used to navigate through to the target record

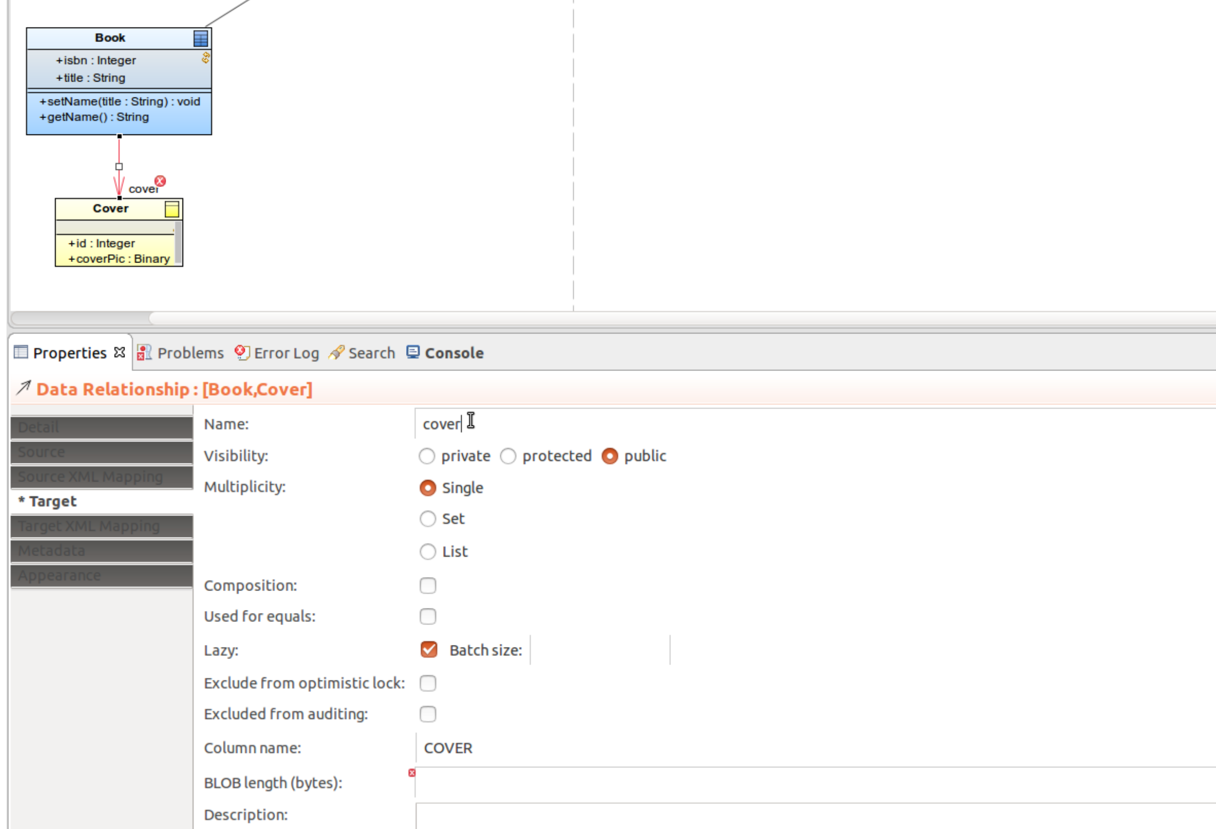


Figure 2.137 Setting the target name

- **Visibility**: visibility of the navigation
  If set to *Public*, the relationship direction is accessible from the entire model; if you unselect the option, it is private and accessible only from within its module.
- **Multiplicity**: cardinality of the relationship end
- **Composition**: establishes a "fixed" relationship direction of records
- **Used for equals**: when comparing instances of the records, the given relationship is included
- **Lazy**: whether loading of the related shared records is lazy or eager (refer to Fetching)
- **Exclude from optimistic lock**: if selected the version of the entity in the database remains unchanged when the relationship changes; hence, the entity will be changed by all commits without collision. If not selected, and the relationship end entity is changed by multiple users at once, the system returns a "Conflict on entity" error.
- **Ends of Shared Records** define in addition the following:
  - **Excluded from auditing**: if true, the relationship end is excluded from revision history keeping of shared Records.
  - **Cache Region**: name of the cache region of the relationship
  - **Foreign keys** (for each end): If such a name corresponds to the name of a primary key Field of that shared Record, the foreign key is used as primary key. The types of the primary key and of the foreign key have to match.
- **Ends leading to a non-shared Record from a shared Record** define the BLOB size.
  This is the space reserved for the non-shared Record value since the entire tree of the non-shared Record is serialized and stored as a database entry of the BLOB type.

(c) Optionally, define the XML Mapping properties of the Relationship ends on the Target XML Mapping and Source XML Mapping tabs.

2.11.5 XML Mapping in a Data Type Model

XML mapping of a data model is used on export to XML when creating webservices, and by the `convertToXML()` and `parseXML()` functions, which make use of xml mapping properties defined on the pertinent records, record fields, and relationships.

- To define the XML mapping of a record, open the *XML Mapping* tab in its Properties view and define the relevant properties:
  - **XSD type**: XSD type of the record
  - **Node name**: name of the record node in the namespace
  - **Namespace**: XML namespace used for the record node
  - **Use xsi:nil flag**: make the record schema nillable so if the instance is `null` its XML instance node is generated with the `xsi:nil="true"` attribute.
  - **Order of fields**: the order of XML nodes generated for the record fields
- To define the XML mapping of a record field or a relationship direction, open the *XML Mapping* on a field and *Source/Target XML Mapping* on a relationship Properties view and define the relevant properties:
  - **XML Transient**: exclude the field from the XML
  - **XSD type**: XSD type of the field
  - **Node name**: name of the record node in the namespace
  - **Namespace**: XML namespace use for the record node
  - **Use xsi:nil flag**: make the field schema nillable so when the field is `null`, its XML instance node is generated with the `xsi:nil="true"` attribute.
  - **Attribute**: the field is the attribute of the record element
  - **Optional**: the field is left out when null
  - **Use as content**: the field value is used as the record node content

  This setting is ignored if the Attribute property is selected.

2.11.6 Creating a Record Composition

Compositions establish a "fixed" data relationship between records in a direction, where the target record represents a part of the other record. The part cannot exist by itself and must hence always define its source value. It represents a "has-a" relationship.

To define a relationship end as a composition end, select the *composition* attribute of the data relationship end in its tab of the Properties view. For further information on composition, refer to the `GO-BPMN Modeling Language Guide`

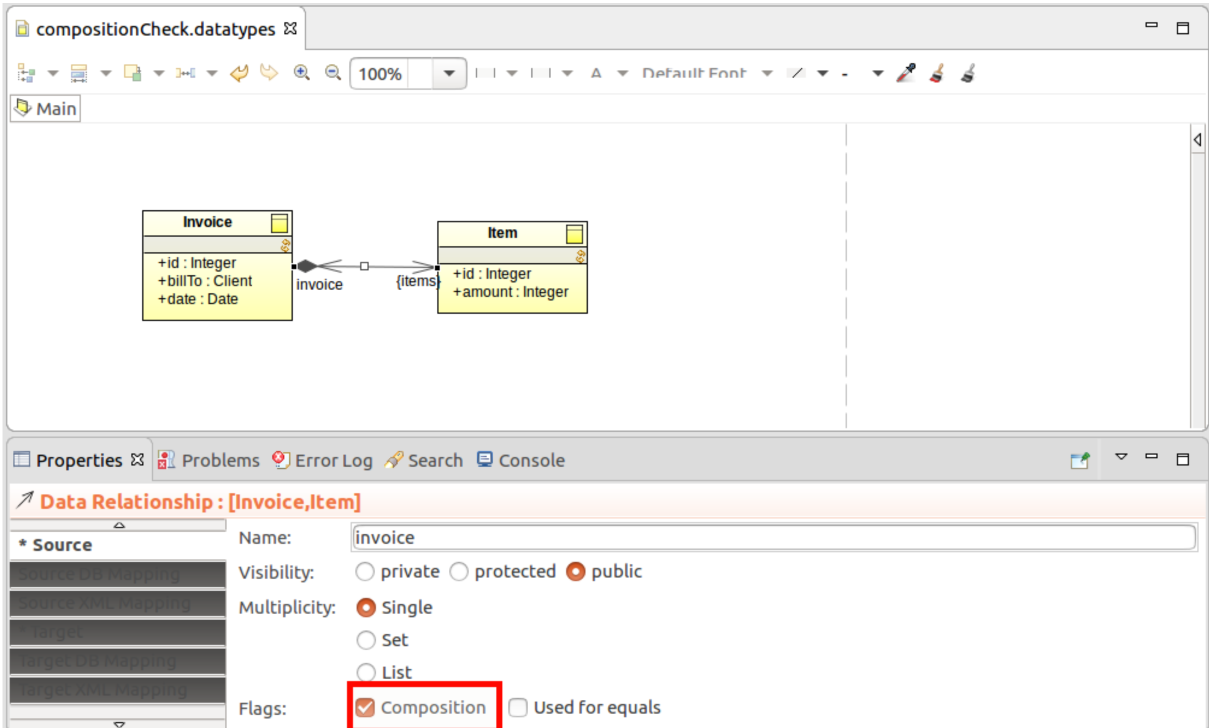


Figure 2.138 Composition setting

2.11.7 Comparing Records

A Record instance is resolved into an object identity: when you compare records, the system compares the object identities, not the Record field values. If you want to compare Records based on the values of some Fields, you will need to define, which Fields should be used.

2.11.7.1 Defining Fields Used in Record Comparisons

Consider two instances r1 and r2 of a Record, which has a String field.

```
def Record2 r1 := new Record2(property -> "string");
def Record2 r2 := new Record2(property -> "string");
r1 == r2
```

The code returns false by default since it checks whether r1 and r2 are the same record instances, which they are not.

On the other hand, if you compare fields of a basic data type, the system compares the field values to check if the records equal: Hence `r1.property == r2.property` returns true.

To compare one or multiple properties when comparing record instances instead of the record identities, set the relevant record fields to Used for equals. If you set the flag on the Record String property,`r1 == r2` would compare the strings and return `true`.

If a record has multiple properties with the *Used for equals* flag, all properties must be evaluated as equal for the record comparison to return `true`.

Note that the flag is inherited by fields of child records. For records with relationships to other records, you need to set the *Used for equals flag* on the navigation to allow inheritance of the equals flags.

> **Important:** The Used for equals flag applies only to non-shared records.

2.11.7.2 Comparing Records with Fields on Related Records

To use a Field of a related Record when comparing instance of a Record, select the *Used for equals* flag on the Relationship end.

2.11.8 Presentation of Record Diagrams

2.11.8.1 Displaying and Hiding Record Compartments

To display or hide additional compartments in Record views, such as,

Fields and inherited Fields, right-click the Record view in the Diagram, go to **Compartments** and select or unselect the compartments.

2.11.8.2 Viewing Record Hierarchy

You can view the record hierarchy of the entire workspace in a tree structure in the *Record Hierarchy* view. The view provides an overview of all data types and their inheritance relationships.

The view is not displayed by default: to display it, go to to **Window** > **Show View**, or call the view from the context menu of a data type (either in the GO-BPMN Explorer or the Data Type Editor). Alternatively, right-click a Record view in a diagram and select **Open Type Hierarchy**.

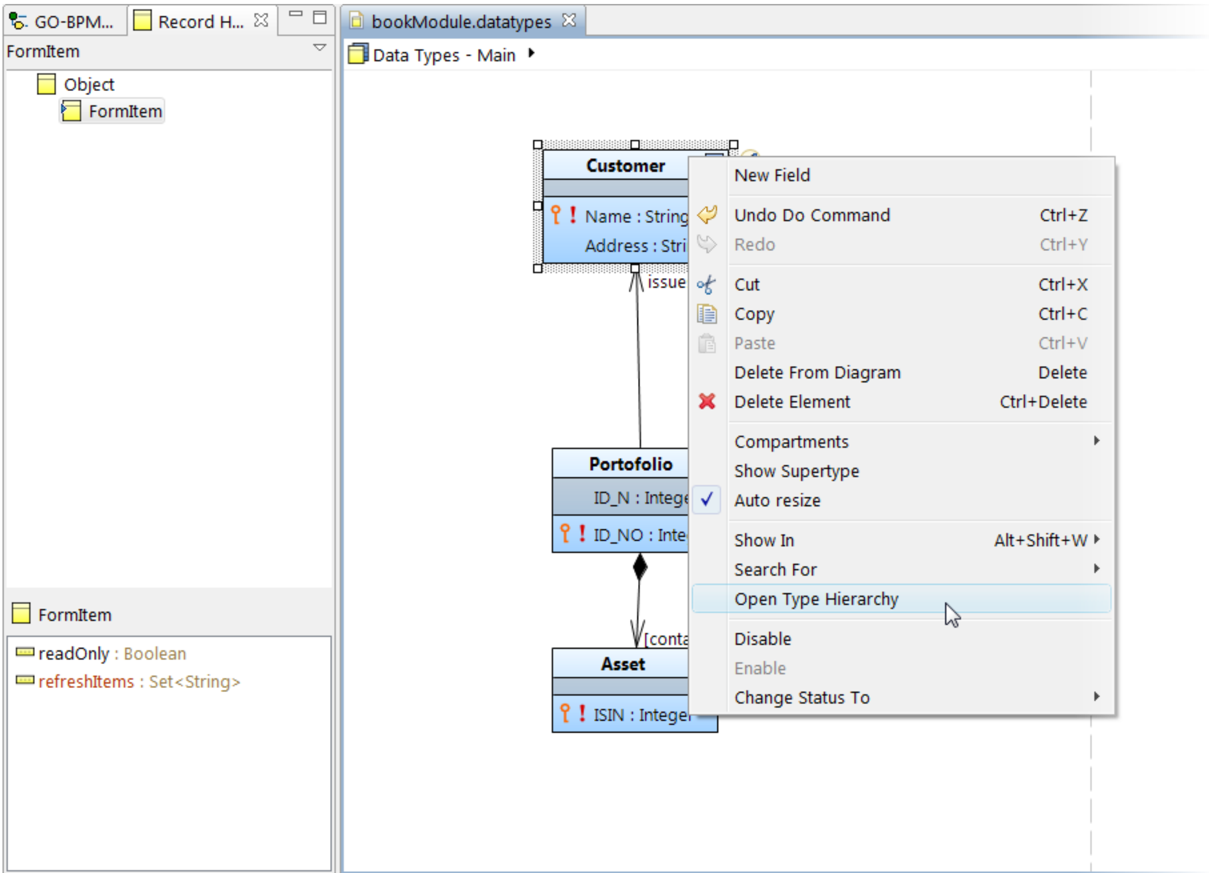


Figure 2.139 Displaying a data type in the Record Hierarchy view (using the context menu) with the Record Hierarchy view on the left

When you select a Record in the tree, its fields are displayed in the lower section. From the context menu of a Record, you can do the following:

- Open the Data Type Editor with the data type focused
- Show the Data Type in the GO-BPMN Explorer or Properties view
- Show Full Type Names for all displayed Data Types
- Focus On the selected Data Type to show only the particular Data Type and its hierarchy

2.11.9 Importing Data Types from an XSD File

To import the data type definitions from an XSD file as Records, do the following:

1. In the GO-BPMN Explorer view, right-click the GO-BPMN module.
2. Click **Generate Data Types from Xsd**.
3. In the Generate Types from Xsd dialog box, select the Xsd file from the file system.

   Some mechanisms of the XSD file, such as, reducing the set of the permitted values or allowing a choice of values for several types, are not supported.

2.11.10 Validating Record Values

Validation of record values checks the values of Record instances and their properties: Validation returns a list of validation messages for the values that do not meet the required criteria.

The criteria for values of records and their properties are defined as **constraints**. Each constraint defines its constraint type and maps to a Record or a Property. It is the constraint type that defines the semantics of the validation—for the constraint to be met, the value of the Record or property must meet the condition defined in the constraint type.

For example, you could define a **constraint** that **binds the doesNotContainDigits constraint type to the record field `givenName`**. Or you could define your own constraint type *isISBN* with a constraint expression that checks if a value is in the ISBN format and then a constraint that would bind it to such record fields as `Book.ISBN` or `Journal.ISBN`.

To check if a Record or property value meets the constraints, call the `validate()` function. The function returns a list of constraint violations: each failed constraint is added to the list as a `ConstraintViolation` with a message returned by the constraint expression of their constraint type, the record instance that failed the validation, and their constraint definition.

You can call the `validate()` function:

- over a record field instance: all constraints defined for the record field within the model are checked.
- over a record instance: all constraints on the record's fields, the record itself, and its super types are validated and that in this order.

To provide additional granularity to validation on runtime, the `validate()` call can have tags as its arguments: on validation, the constraint is applied only if the tags of the validate() call are identical to the constraint's tags expression. This applies only if any of the constraints for the record or record field define a tag expression. If the constraints do not contain any tags, they are applied on any `validate()` call.

When validating Record data from a ui::form, the violation messages can be displayed in the form with the `showConstraintViolations()` function.

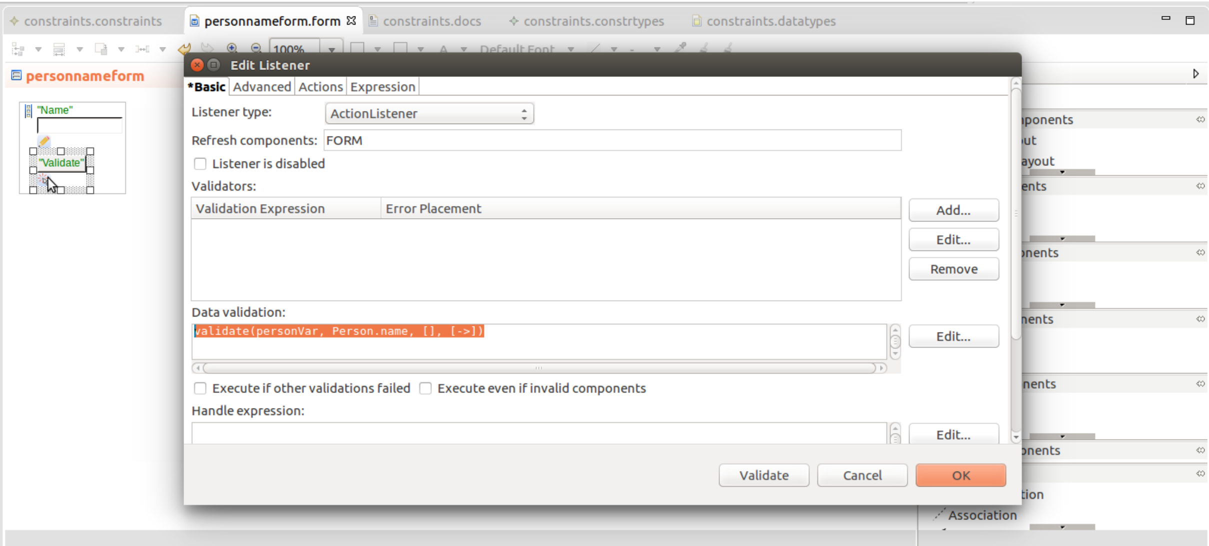


Figure 2.140 Validating record value using constraints

2.11.10.1 Defining Constraints

To define a constraint for a Record or a Record Field, do the following:

1. Open or create a constraint definition file.
2. In the constraint editor, do the following:
   (a) In the Constraint area, click Add.
   (b) In the Constraint Details area, define the constraint details:
      - **ID**: unique identifier of the constraint
        It is recommended to use IDs in the format `<RecordName>.<FieldName>.<ConstraintTypeName>`, such as, `Book.ISBN.Format`, `Book.ISBN.NotNull`, `ISIN.IsNumber`, `ISIN.HasMinLength`, etc.
      - **Record (property)**: record or record Field to which the constraint applies
        Use the dot operator to use records or record Fields on Data Relationship ends.
      - **Tags**: expression that results in a list of tags
        Tags serve to filter constraints that are applied on validation of the Record or Record Field.
        Example Tag expression

```
tag1 and (tag2 or tag3) and not tag4
```

      - **Constraint type**: constraint type that defines the constraint expression, type, parameters

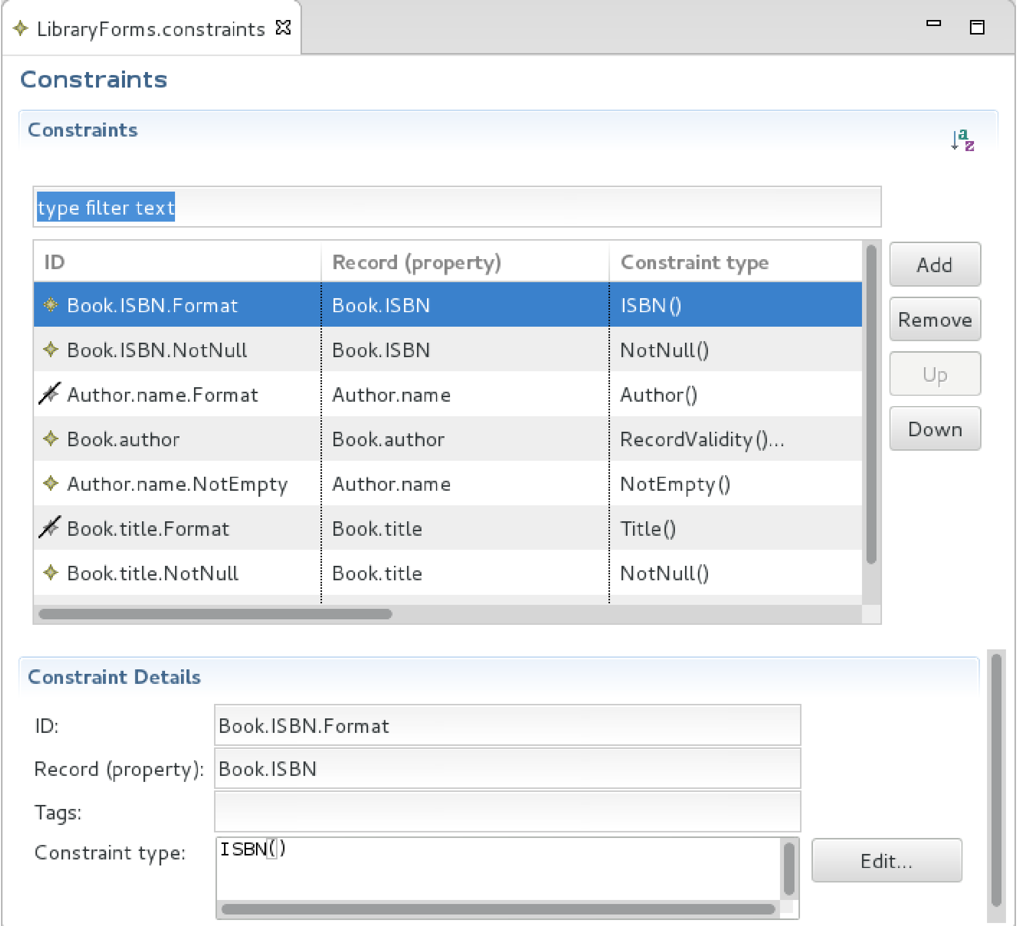


Figure 2.141 Defining constraints

2.11.10.2 Defining Constraint Types

Note: The Standard Library contains a rich set of constraint types. Hence check the available constraint types before you define custom constraint types.

To define a constraint type, do the following:

1. Open or create a constraint type definition file.
2. In the constraint type editor, do the following:
   (a) In the Constraint Types area, click Add.
   (b) In the Constraint Type Details area, define the constraint type details.

Every constraint type defines the following properties:

- **Name**: name of the constraint type

  The constraint type name should refer to what is validated, for example, IsEmail, IsNumber, HasMinLength, etc.
- **Applied to**: data type to which the constraint type can apply

- **Generic types**: Type parameters allow a constraint type to operate over a parameter that can be of different data types in different calls. The concept is based on generics as used in Java. You can also make a generic data type extend another data type with the extends keyword. The syntax is then `<type_param_1> extends <type1>`, `<type_param_2> extends <type2>`.
- **Parameters**: input parameters of the constraint type

  The following parameters are provided by default and cannot be deleted:

  – value: input object value (the value is automatically filled in based on the *Applied to* data type)

  – context: additional data that can be passed as parameters
  The context parameter can be used instead of tags. It is an object of the type Map<String, Object> and when defined by the validate() call, the call cannot define any tag arguments.

  – constraintId: ID of the constraint definition that calls the constraint
  You can define additional parameters.
- **Type**: return type of the constraint expression

  – **simple**: Expression return value is a String which represents the error message.

  – **complex**: Expression return value is a list of ConstraintViolation objects or their subtype.
  Complex types allow cascading validation when related records and records fields are validated as part of the record validation.
- **Expression**: constraint check expression

  If validation is successful, the expression must return null; if it fails, it must return
- a String when Type is set to simple or a list of ConstraintViolation objects when Type is complex.

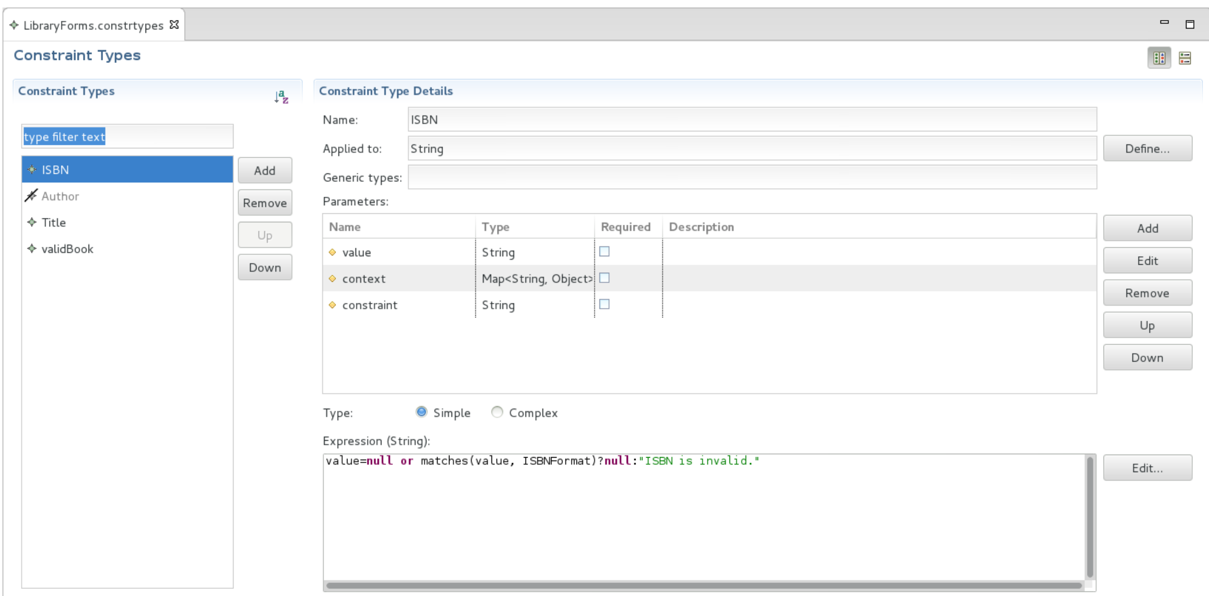


Figure 2.142 Defining constraint type

2.11.10.3 Filtering Constraints using Tags

A constraint can define a tag expression, which must evaluate to true for the constraint to be applied: The expression serves as a filter on constraints. When the validate() function is called, the constraint definition is invoked only if the validate() call contains at least one of the constraint's tags as its argument. If the constraints do not contain any tags, they are applied on any `validate()` call of the Record or Record field.

Validation tags can contain subtags. They allow you to create hierarchical tags: if a tag contains a collection of subtags, the tag is considered a union of its subtags.

For example, if a tag Car defines subtags ProductionYear and Model, on `validate()` call, the tag Car resolves to the ProductionYear and Model tags.

2.11.10.3.1 Defining Tags

To define tags, do the following:

1. Create a Validation Tag Definition (right-click your module, then click New -> Validation Tag Definition)
2. Double-click the definition to open it in the Validation Tag Editor.
3. In the Validation Tags area, click Add.
4. In the Validation Tag Details area, define the tag's properties.

2.12 Persistent Data

Under normal circumstances, record data ceases to exit as soon as the model instance ceases to exist. If you want to persist Record instances, mark their Records as shared: shared Records and their Relationships are reflected in the database as tables.

When you create an instance of a shared Record, a database entry is created in the database table: any readings, modifications, and deletions of shared Record values are reflected in the entry.

> **Note:** Note that variables of a shared Record type hold a reference to the shared record and must be fetched anew in each new transaction: You might want to consider the performance aspect (for details on model transactions, refer to the `Software Development Kit Guide`).

The persistence mechanism of shared Records makes use of Hibernate: Based on the data models, the system generates the respective tables and a single common Hibernate setting file: as a consequence, if you upload multiple versions of a data type hierarchy in multiple modules, only the last data type model is used.

If you modify the data type hierarchy, you have to decide how to handle the changes in the database mapping: this is set by the Database Schema Update Strategy (similar to the `hbm2ddl` Hibernate configuration) of your server connection. You can also enable schema update per Record.

2.12.1 Defining Database Properties

Each data type definition can specify properties that define the database where the tables are persisted. To change the properties of data types in a definition file, such as, target database, table names, foreign key names, and index name prefixes, do the following:

1. Open the datatypes definition for editing.
2. In the *Outline* view, select the root *Data Types* item.
3. In the Properties windows, change the settings:

- **Database**: JNDI name of the data source used for the database
  This allows you to store the instances of shared Records in another database accessible to your application server.
- **Table name prefix**: prefix used in the names of database tables created based on this data type definition
  It is good practise to use a prefix so you can easily find your tables. You can check the prefixed table name for individual Records and Relationship in their Properties view.
- **Foreign key name prefix**: prefix for the names of the foreign key columns created based on this data type definition
- **Index name prefix**: prefix of the index column name created based on this data type definition
- **Table name suffix**: suffix used in the name of the database tables created based on this data type definition
- **Foreign key name suffix**: suffix for the names of the foreign key columns created based on this data type definition
- **Index name suffix**: suffix of the index column name created based on this data type definition

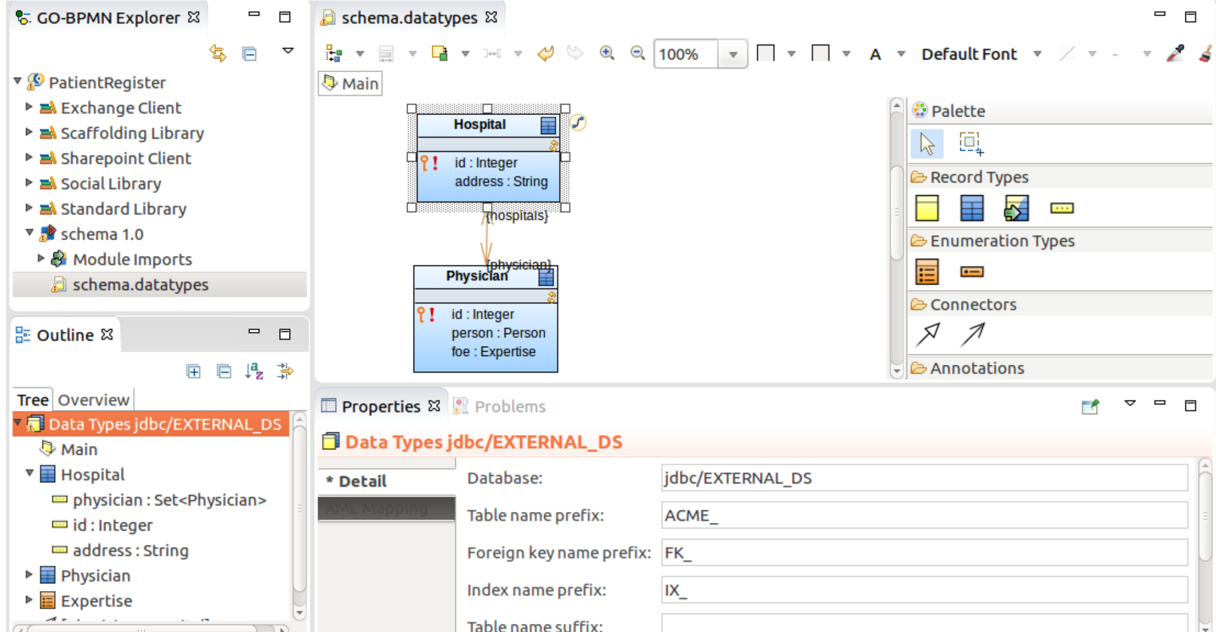


Figure 2.143 Defining database properties for a data type hierarchy

2.12.1.1 Extracting Affixes from Data Model

If a data type definition contains shared Records with a common prefix or suffix, foreign key or index names on the *DB Mapping* tab of the Properties view, you can extract the affixes so that they are set for the entire data type definition.

To extract affixes from a database with shared records, do the following:

1. In GO-BPMN Explorer, click the data types file.
2. In Outline view, click the Data Types root node.
3. In the Properties windows, open the Detail tab.
4. Click the **Extract** button.
5. Modify the affixes are required and click **Apply**.

2.12.2 Generating a Data Model from a Database Schema

To generate a data type model from a database schema, do the following:

1. Connect to the server with the data source.
2. In the GO-BPMN Explorer view, right-click the GO-BPMN module.
3. Click **Generate Types from DB Schema**.
4. In the Generate Types from DB Schema dialog box, select the data source and click Next.
   If the data source is not listed, click New and define its properties.
5. Select the database schema and click Next.
6. Select the tables and columns to be included in the data type model.

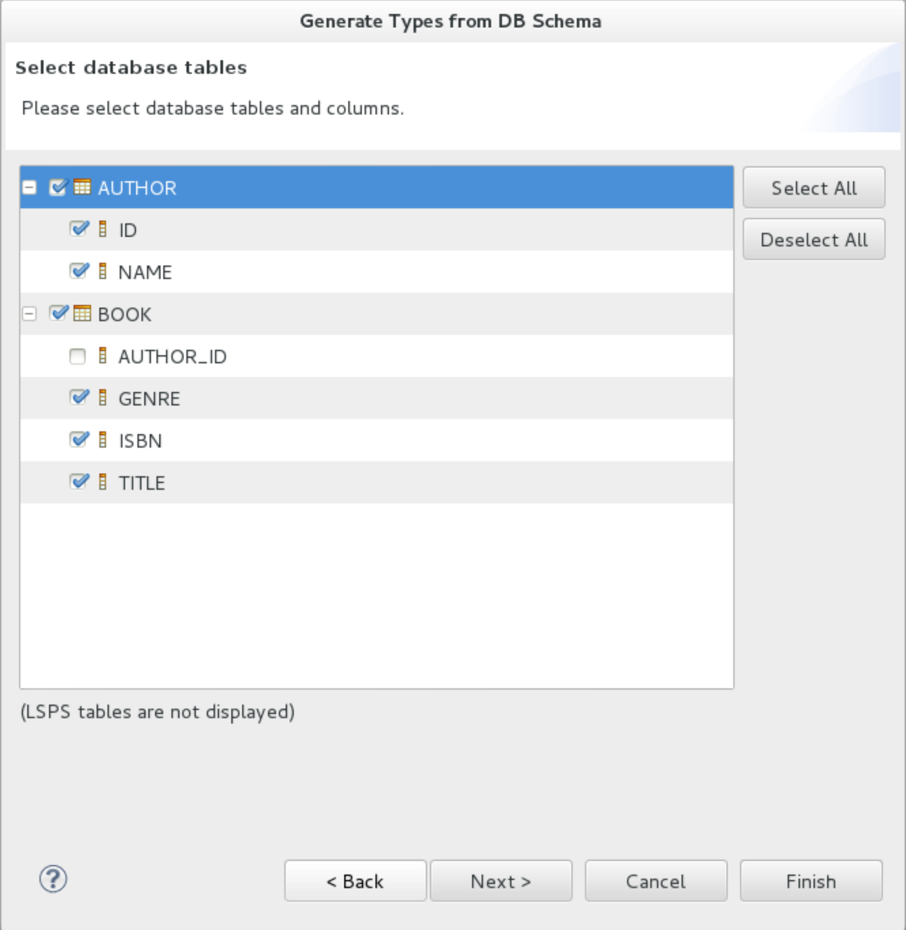


Figure 2.144 Selection of Database Entities

7. Select the target location and enter the resource name, and click **Finish**.

Note the following limitations:

- The generated data type definition does **not** generate any Diagram views of the shared Records: Drag the shared Records from the GO-BPMN Explorer onto the data type canvas if required.

- The tool fails to detect that a shared record for a table already exists in the module and generates a new record.

  Consequently, if a record that is being generated has a relationship to an already existing record, the relationship is not generated either.

- The fields of generated records are ordered randomly.

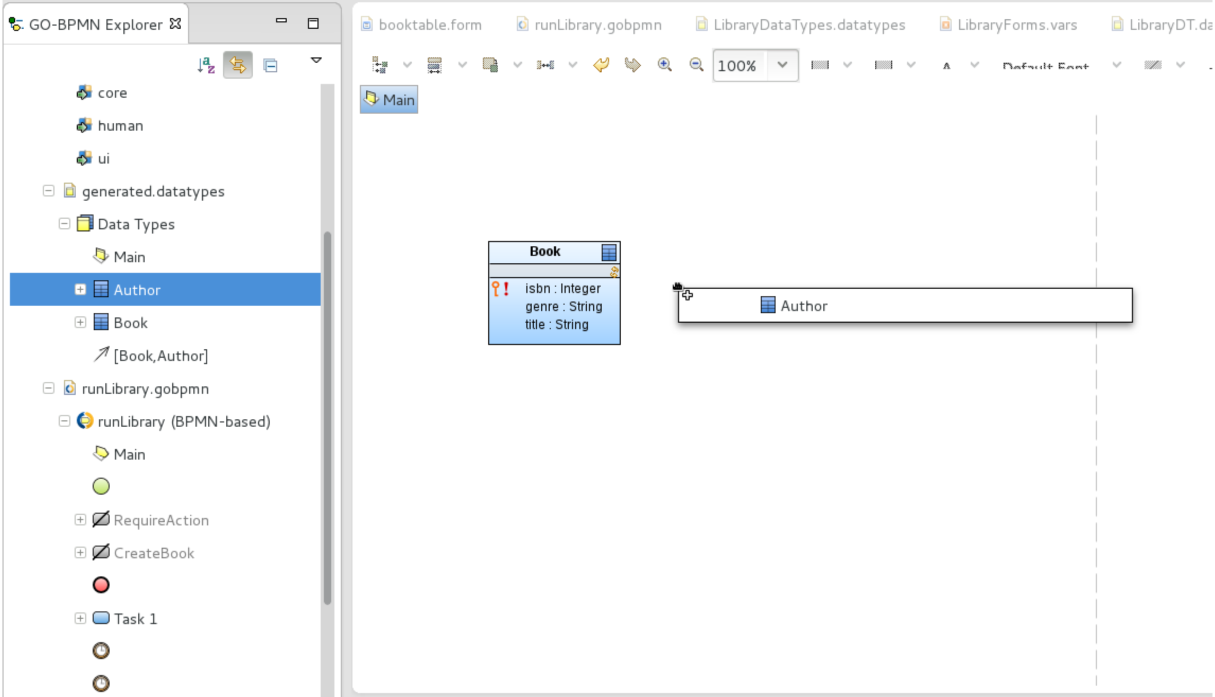


Figure 2.145 Adding a depiction of a generated shared record onto a data type diagram

2.12.3 Creating a Shared Record

Shared records are defined just like common Records with the additional *shared* flag, which is equivalent to `@Entity` in Hibernate, and the following properties related to persisting:

- **Database Mapping**:
  - *Table name*: target database table
  - *Schema*: target database schema

    Important: If multiple shared records have the same target table and schema, the shared records will be mapped to the same table. This might results in issue due to incompatible Schema incompatibilities. To prevent such issues, consider setting *Table name prefix* for your data type definition.
  - *Catalog*: target database catalog
  - *Batch size*: the number of the fetched entities when the record is accessed from its related record
  - *Cache Region*: cache region of the shared record
  - *O-R inheritance mapping*: available only if the shared record is the supertype of others.
    - *Each record to own table*: maps each shared record in the hierarchy as its own table

    * *Single table per hierarchy*: a single table that unifies properties of all sub-records is used to store the hierarchy. This option is convenient if you plan to query the database based on the Record type. For this option, you can specify the name of a DB column name that discriminates the type (the default values is TYPE_ID).
  - *Inheritance FK name*: name of the Foreign Key used on inheritance
  - *Update schema*: if true, the existing table schema is updated using the current schema

- **Indexes**: indexes on the related table of the Records

Important: When working with shared Records that are a target or source ends of a relationship, make sure to define indexes for the foreign keys on the Records and their Relationship to prevent potential performance issues. Note that you can generate the indexes in the data types file automatically: right-click the canvas of your Record diagram and select **Generate Indexes**.

2.12.4 Creating a Shared Record Field

To create a Record field in a shared Record, do the following:

1. Select the shared Record or its Field and press the Insert key to add a new Field.
2. Select the field and define its generic properties on the Detail tab of its Properties view.

   The **type** of a shared field should be set to a simple data type. If such a field is of another data type, consider creating a related shared Record to prevent performance issues due to frequent serialization and deserialization.
3. Define the database-related attributes on the *DB Mapping* tab:
   - **Column name**: the name of the mapped database column (The target database table is defined in the parent shared Record.)
   - **Text length**: the maximum length of the database entry for the String fields
   - **Precision**: maximum number of digits for the Decimal and Integer fields
   - **Scale**: number of digits after the decimal point for Fields of the Decimal data type

     Note: When writing values into the fields, Decimal fields behave like Java's BigDecimal.
   - **Unique**: When generating the database schema of the data model, the field will be translated into a column with unique values.
   - **Version**: if true, the field is used to store the version number which is bumped whenever the record changes making it subject to optimistic locking;
   - **Exclude from optimistic lock**: if selected, a change of the field never results in a locking conflict and transaction rollback.
   - **Not null**: if true, the field must not be `null`. If you try to create a record instance with the field set to `null`, the operation will fail with an exception.
   - **Primary key**: if true, the field is considered the primary key in the database table.
   - **Auto generated**: if true, the field value is generated automatically when the instance of the shared record is created.

     The attribute is available only for fields that are simple primary keys with integer values. Depending on the target database, either a sequence is generated, or auto-incrementation is used.
   - **Sequence name**: the database sequence used for the auto generated field

   If it does not exist, it is be created.

Important: When creating a new record with a specified property that is set as auto-generated over an H2 database, the system will silently ignore the specified value and use the auto-generated value. For example, if a shared Record Book defines the field ID that is auto-generated and you instantiate a new Book as `new Book (id->1)`, the ID value 1 will be ignored and the auto-generated ID will be used instead. On other databases, such code causes an exception.

2.12.5 Locking on a Shared Record

Locking prevents a change of a shared record if the record changed since it was loaded; for example, if you save a to-do with a shared record and another user changed the record from another to-do while you were editing it. In such a case, the attempt to apply the changes on the shared record will result in a transaction rollback and an exception.

Optimistic locking uses a dedicated Field of a shared record to store its *Version*: the Field has its value bumped each time the record changes. If you are changing a record with a particular version and the version changes in the meantime, the server returns an exception when you try to apply your changes. Note that you can exclude fields from optimistic locking: when such fields change, the version Field of the record remains unchanged. As a result, the field accept all its changes from all transactions without causing a transaction rollback.

If not selected, and the Field is changed by multiple users at once, the system returns a `Conflict on entity` exception.

> **Note:** The locking mechanism makes use of the optimistic locking mechanism of the underlying Hibernate framework.

To set up locking on a shared Record, do the following:

1. Add a Field to the Record that will hold the version information. The Field can be of the Integer or Date type; it is recommended to use the Integer type.
2. On the *DB Mapping* tab of its Properties view, select *Version*.

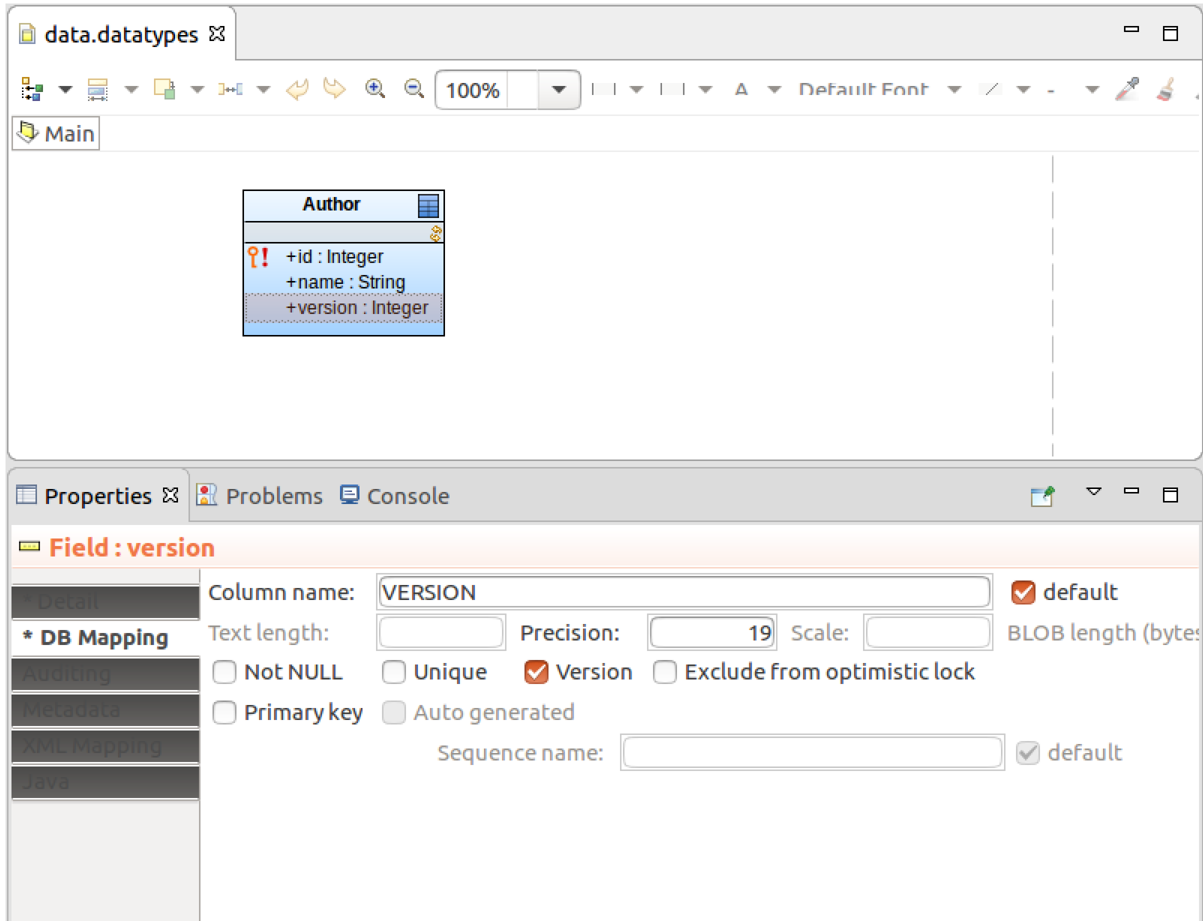


Figure 2.146 The Version field of a Shared Record

3. Optionally, exclude Fields which can be changed between their load and save:

   On the *DB Mapping* tab of the Fields' Properties view, select **Exclude from optimistic lock**.

2.12.6 Data Relationships Between Shared Records

Data Relationships between Records establish a relationship between the Record tables. Unlike in JPA, the Relationship is symmetrical (set on both ends of the relationship) and it is not necessary to make one of the tables the Owner.

2.12.6.1 Setting Fetching

While values of shared Records are fetched anew in every transaction of a model instance, when you fetch a shared Record that is related to other records, the related Records can be fetched immediately or when explicitly requested. This is determined by the fetching strategy on the data relationship ends:

- Eager: related shared records are fetched immediately when the source record is accessed.
- Lazy: related records are fetched only when explicitly accessed via their data relationship (`source.relationshipname`)
  - For to-many relationships, you need to define the batch size. This defines for how many of the parent records all the related records are fetched.

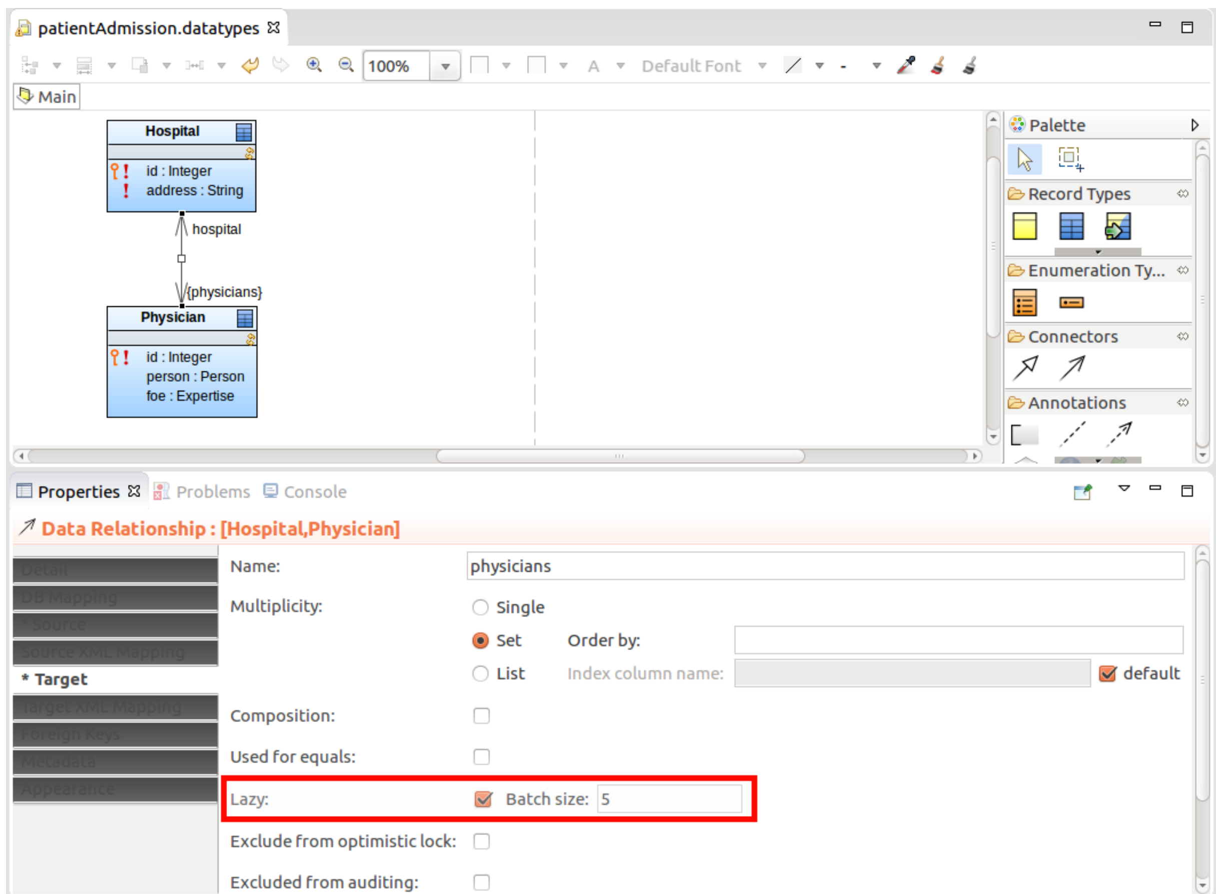


Figure 2.147 Lazy fetching configuration

2.12.6.2 Defining Indexes

To allow quick look-up of shared Records in relationships, create indexes of foreign keys for the underlying database tables: you can do so directly in the database or you can define the indexes using PDS.

> **Important:** The absence of indexes for your shared Records can cause performance issues. Make sure to define indexes to prevent slow search on your database data.

To define indexes for a table of a shared Record from PDS, do the following:

1. Display the properties of the record in the Properties view: click the record either in the GO-BPMN Explorer or in the record diagram.
2. In the Properties view, open the Indexes tab.
3. Click the Add button on the right.
4. In the Database Index dialog, select the column that should be indexed and click > to add it to the indexed columns.

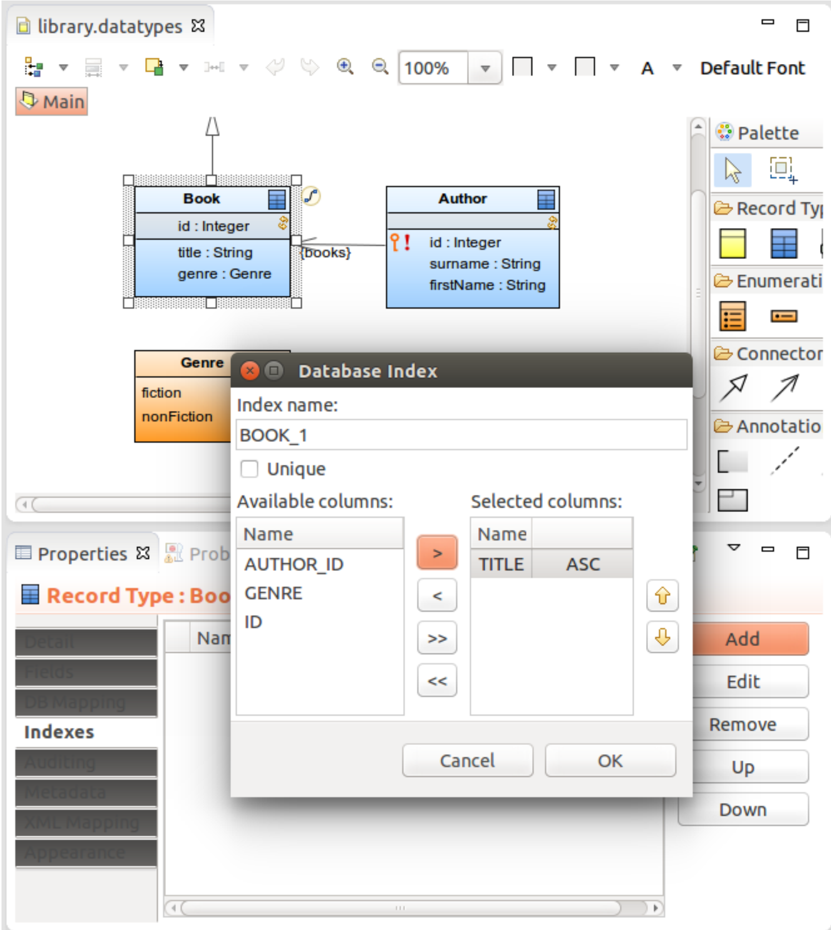


Figure 2.148 Defining index

5. Upload the Module.

2.12.6.2.1 Generating Indexes

To generate indexes on foreign keys for all shared records that are related to another record in the data types file, right-click the file in GO-BPMN Explorer and click the **Generate Indexes** button in the Properties view; alternatively you can right-click into the canvas in a Record diagram and select **Generate Indexes**.

2.12.6.3 Defining a Shared Field with a Foreign Key of a Related Record

To allow for a more efficient recovery of IDs of related shared Records, you can define the foreign key of the Relationship end as the column name of the record:

1. Name the Relationship end targeted to the related Record.
2. On the target Record, define a primary key Field.
3. On the *Foreign Keys* tab of the Relationship properties, check and possibly modify the foreign key of the primary key Field.

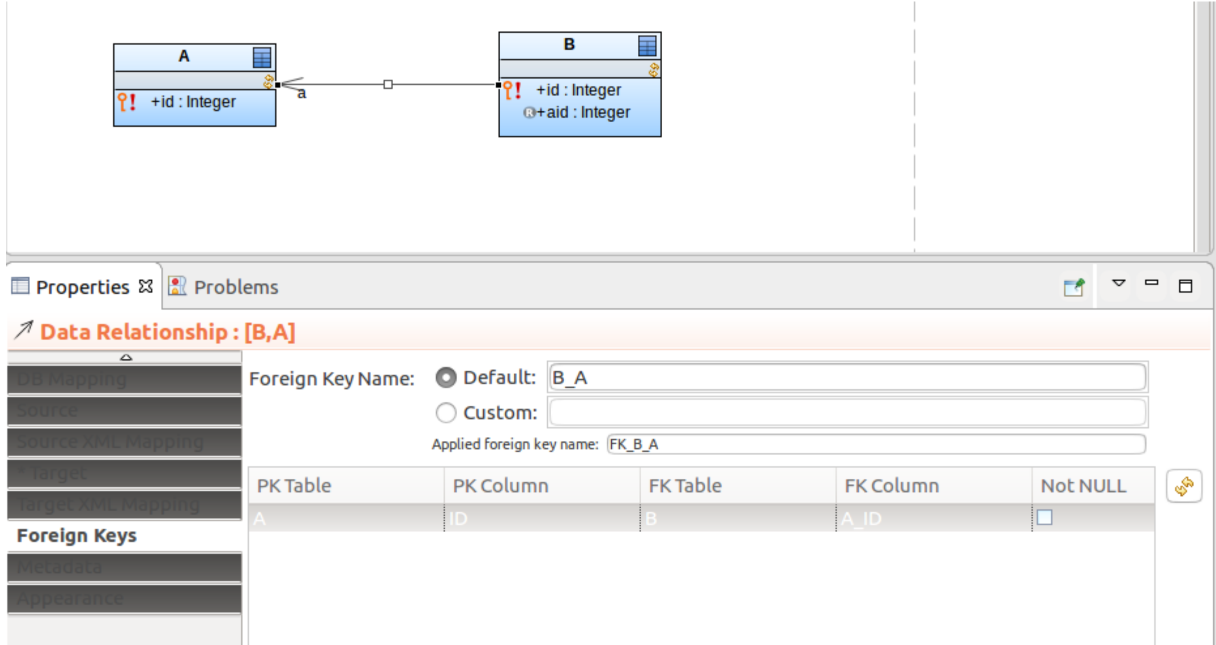


Figure 2.149 Foreign key on the Relationship end

4. On the source Record, create a read-only Field that will be mapped to the foreign key.

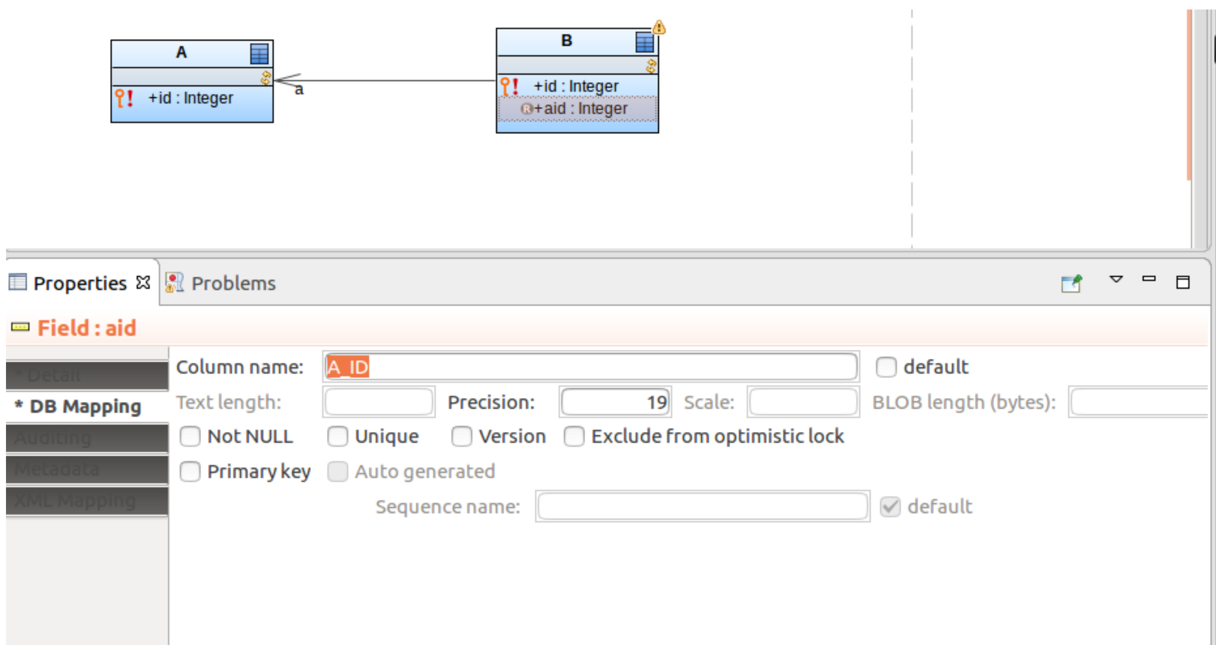


Figure 2.150 DB mapping of the Field

5. On the DB Mapping tab of the Field, insert the name of the foreign key column.

Now you can access the primary key of the related shared Record using the read-only Field.

Such foreign key fields, if set as primary keys, are set automatically when the related record is assigned. For example, if *Parent* has a relationship to *Child* and one of the *Child*'s primary-key fields is mapped to the primary key of *Parent*, the field is automatically filled with the *Parent* id:

```
def MyParent p1 := new MyParent();
def MyChild c1 := new MyChild(id -> 1, parent -> p1);
```

Note that you need to make the relationship with the parent object: **do not assign the foreign key directly** as, for example, `new MyChild(id -> 1, parentId -> p1.id);`

2.12.7 Auditing: Shared Record Versioning

The auditing of shared records refers to storing of each version or revision of a shared record which is subject to auditing: for instances of such records, as the record changes, the system stores each revision. You can then work with individual versions of the entity using `auditing functions`.

When you change records of audited Records in a transaction, the auditing mechanism creates a revision entity with an ID and inserts "snapshots" of the changed records to their auditing tables. Optionally, it enters the Record name and revision ID for each changed record into the entity name table.

Example: *Book* is an audited Record with the Field *title* and you create a new book and edit an existing one:

```
new Book(title -> "Something Happened"); // record id is 2
getBookByTitle("Catch 22").title := "Catch-22"; //record id is 1
```

Auditing will perform the following:

1. Create a revision with revision ID and optionally the timestamp as a Revision Entity, for example, with the ID `1`.
2. Record the changes on the Book instances: two entries, one for the new book and another one for the changed book with the following details:
   - record ID
   - revision ID set for both to `1`
   - type of change
   - title of the book as after change
3. Optionally, the entity name table records for each change a new entry with record type: hence two entries with record BOOK and the revision ID `1`

2.12.7.1 Setting up Auditing

To set up auditing, do the following:

1. Create a shared Record that will be used as the Revision Entity: the record represents the table that will store revisions.
   (a) On its *Auditing* tab in the Properties view, select the *Revision entity* option.
   (b) Typically you will add also the *timestamp* Field to the Record to be able to request the timestamp from the database: add a Field of the type Integer or Date and set it as *Revision timestamp* on the Auditing tab in its Properties view.

   Note: To store additional revision-related information, modify the underlying data model, that is, either add new Fields to the *Revision Entity* Record or create a related shared Record (refer to `Entity Auditing in the Software Development Kit documentation`). **Important:** Only *one* revision entity Record and modified entity name Record can exist on the LSPS server.

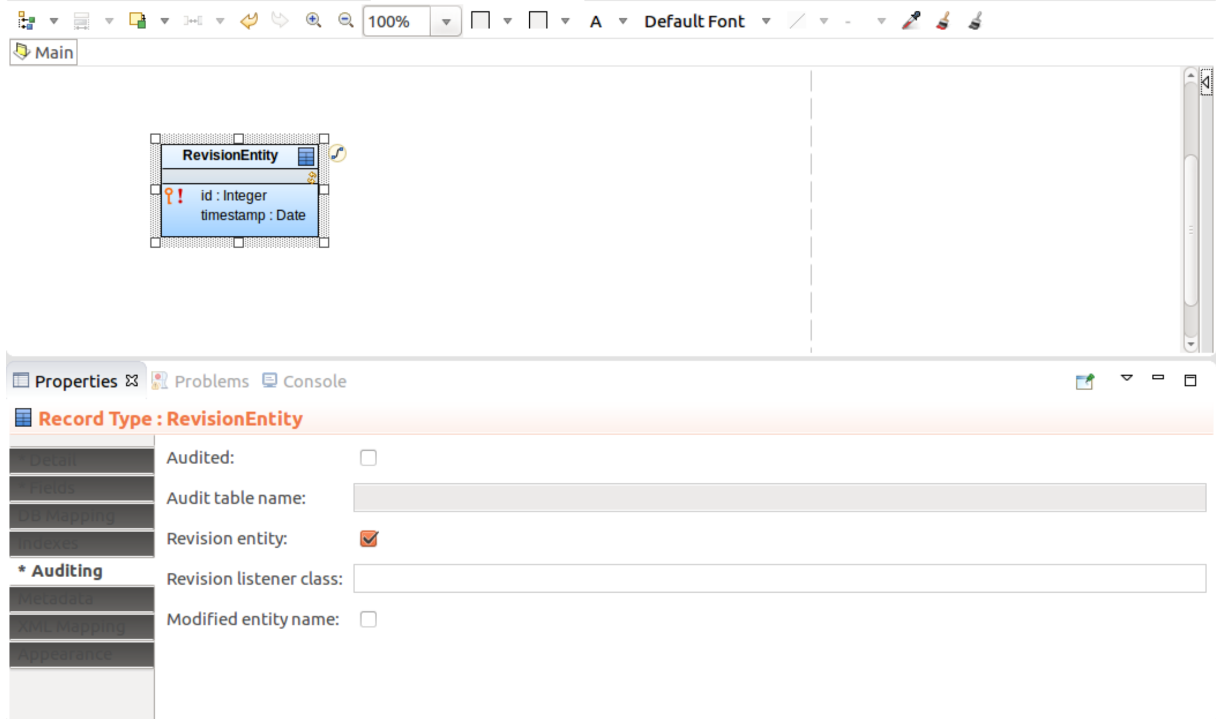


Figure 2.151 Revision entity shared Record

Note: By default the Revision Entity uses the LSPS implementation of the Revision Listener to enter revision data into the database table. The listener enters the id of the revision and optionally the timestamp into the database table. If you want the system to enter further data about the revision, you need to implement your own Revision Listener that will extend the *LSPSRevisionListener* class (refer to Entity Auditing in the Software Development Kit documentation).

2. Optionally, create a related shared Record that will hold the names of the entities changed in the given revision:

   (a) Create a shared Record related to your *RevisionEntity* Record.

   (b) On the Relationship, name **both** relationship ends and set its Multiplicity to Set.

   (c) On the *Auditing* tab of the Record Field for the name, select its *Entity name* property.

   (d) Add a String field that will hold the name of the modified entity: On the *Auditing* tab of its Properties view, select its *Entity name* property.

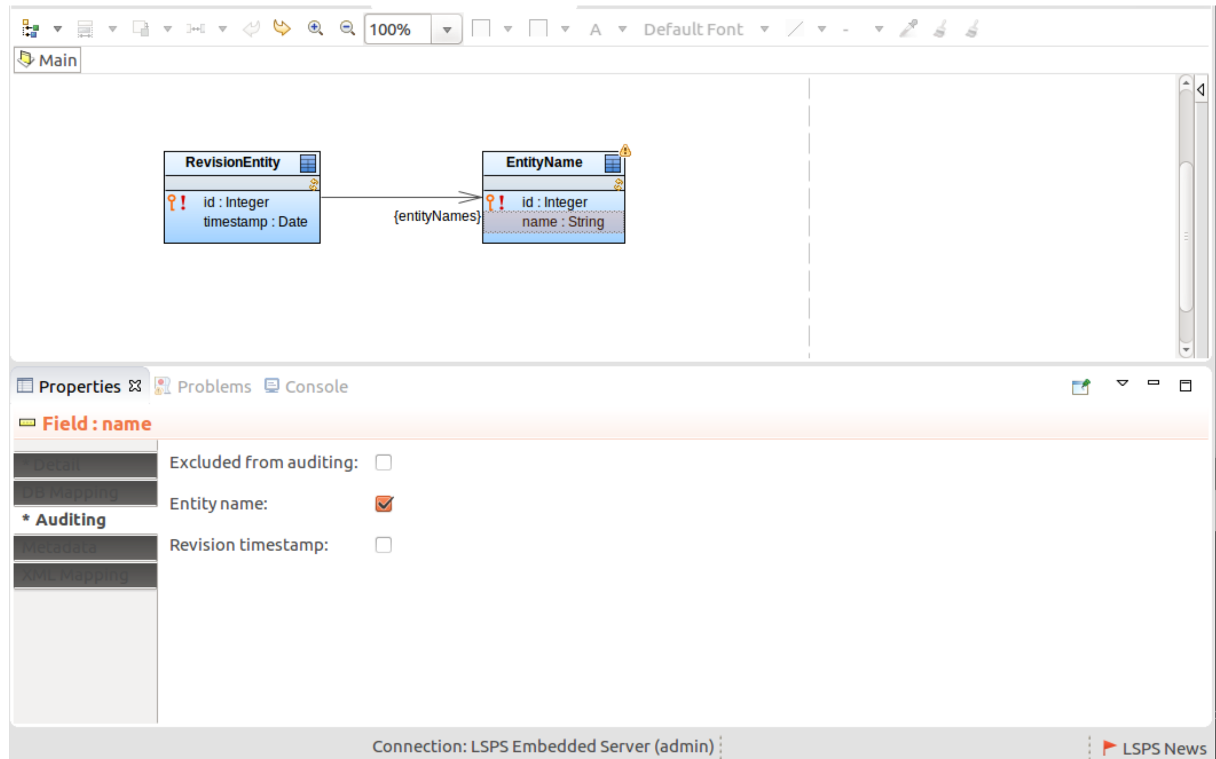


Figure 2.152 Shared Record for the modified entities of a revision

3. Upload the Module.

4. Set the required shared Records as audited: open the Record's Properties and on the *Auditing* tab select the *Audited* option and upload the Module.

2.12.7.2 Auditing a Shared Record

To enable auditing of a shared Record, do the following:

1. Open the Record's Properties view.

2. On the *Auditing* tab:

(a) Select the *Audited* option.

(b) Optionally, in *Audit table name*, enter the name of the table that will hold the Record revisions.

By default, the table name is <RECORD_NAME>_AUD.

3. Upload the Module with the data model.

Note: Make sure that you have uploaded the Revision Entity shared Record to your server: the LSPS database will contain the ENTITY_REVISION table.

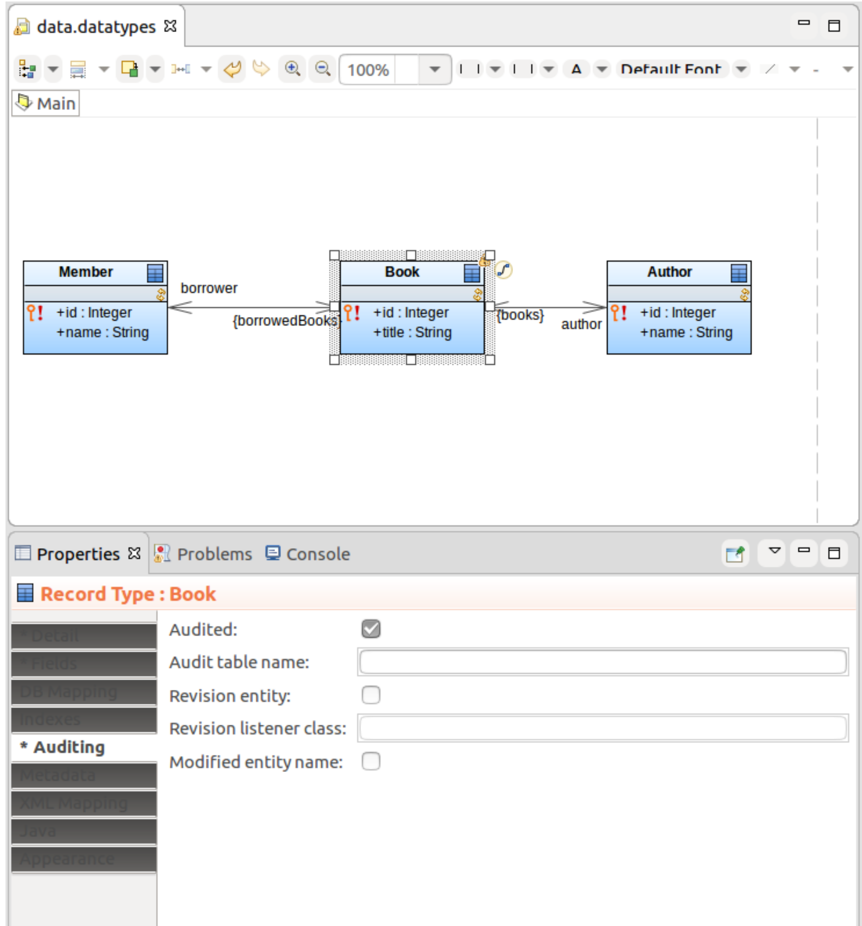


Figure 2.153 Audited shared Record

2.12.7.3 Excluding a Shared Field or Record from Auditing

If you want to exclude a Record Field from auditing, open its Properties and on the *Auditing* tab, select the *Excluded from auditing* option.

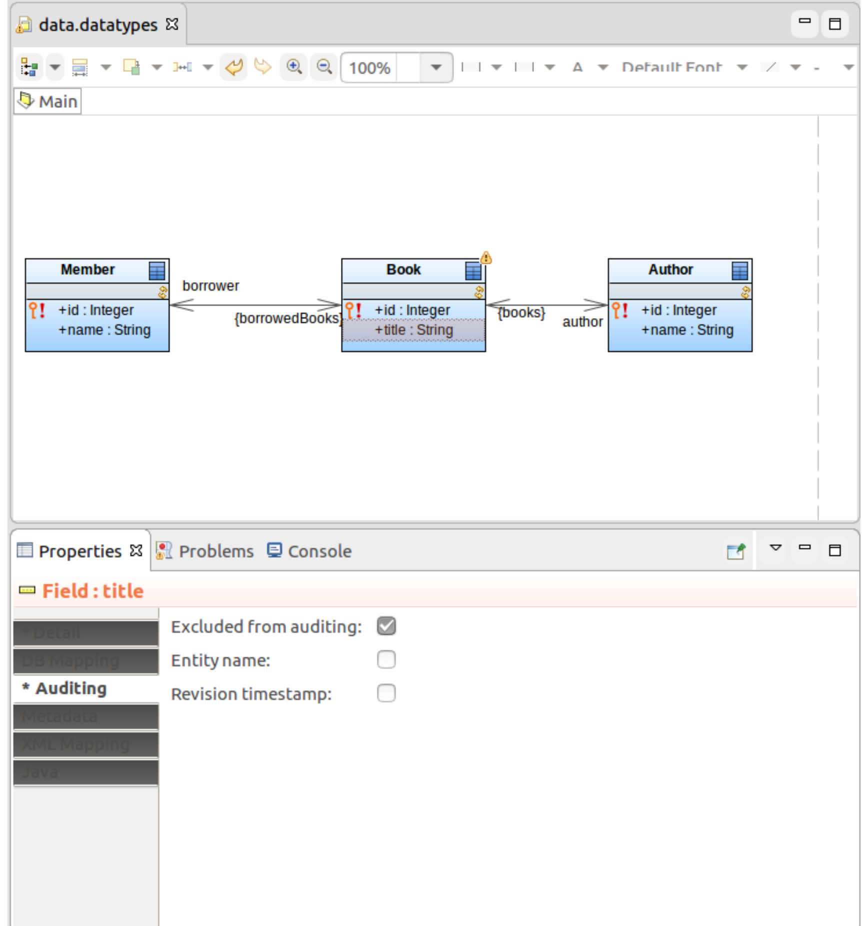


Figure 2.154 Field excluded from auditing

2.12.7.4 Excluding a Relationship End from Auditing

If you want to exclude a Record Relationship end from auditing, open the Relationship Properties and on the tab for the Relationship end, either the *Source* tab or the *Target* tab, select the *Excluded from auditing* option.

2.12.7.5 Working with Record Revisions

To work with revisions of Record instances, use the provided Standard Library functions available in the *Core* module:

- *getRevisions()* to obtain revisions of the Record instance created within a certain period of timestamps
- *findByRevision()* to obtain the Record instance in a particular revision
- *getCurrentRevision()* to get the current revision of an Entity

2.12.8 Caching Shared Records

The caching mechanisms for shared Records reduces the load on the underlying databases. It ensures that shared Records that might be required by the same model instance, user, as well as other users and transactions are kept in memory.

LSPS applies first-level caching within individual transactions, that is, any data is cached within a transactions. The cache regions implement second-level caching that is applied on shared Records. The cache exists regardless of the instance transaction or model instance life (refer to `Software Development Kit Guide` for information on transactions).

The caching is defined by cache regions, which are added to the LSPS Server cache on module upload.

2.12.8.1 Defining Cache Regions

To define a cache region, do the following:

1. In the GO-BPMN Explorer view, double-click the respective cache region definition. The Cache Region Editor is opened in the editor area.
2. In the Cache Regions area, click **Add**.
3. In the Cache Region Details area, define the cache region attributes.
   - **Name**: name of the cache region
   - **Database**: JNDI database name on which the cache region is applied (for example, jdbc/my_database) If undefined, the LSPS system database is used.
   - **Max elements in memory**: maximum number of objects to keep in the memory cache
   - **Eternal**: if true, the cached objects are not scheduled for discarding
     - **Time to idle**: time in seconds an object remains cached if not accessed
     - **Time to live**: seconds an object remains cached regardless of the accesses
   - **Overflow to disk**: if true, the file system is used to store cached objects.
     - **Disk persistent**: if true, the cache remains unchanged after the restart of the process engine.
     - **Disk expiry thread interval**: interval for the cleaning of expired cached business objects in seconds
     - **Max elements on disk**: maximum number of objects to keep in the disk cache
   - **Memory store eviction policy**: If a memory store has a limited size, the objects will be evicted from the memory when it exceeds this limit. The following strategies are available:
     - **LFU**: The least frequently used objects are evicted.
     - **LRU**: The least recently used objects are evicted.
     - **FIFO**: objects are evicted in the same order as they are cached.
   - **Description**: free text area to describe the cache region.

2.12.8.2 Disabling Cache Regions

To disable the LSPS system cache regions, define the disabled cache regions in the `<YOUR_CUSTOM_A↩ PP>-ejb/src/main/resources/cache-regions.properties` file of your custom LSPS application.

2.13 Constants

A *Constant* is a global variable with a value of a basic data type, an enumeration data type, or a map of these data types. After its value has been initialized, it remains unchanged in its context.

Constants are initialized before model variables; hence calls to model variables return the value `null`. Therefore make sure not to use any context data when initializing constants.

To define a constant, do the following:

1. In the GO-BPMN Explorer view, double-click the respective constant definition.
2. In the Constants area of the displayed editor, click **Add**.
3. In the Name text box, type the variable name.
4. In the Type text box, type the data type of the constant value.
5. Define the visibility:
   - Select the Public checkbox to make the constant accessible from importing modules.
   - Clear the Public checkbox to make the variable available only within the parent module.
6. In the Value text box specify the value of the constant:

To call your constant in an expression, write its name.

2.14 Webservices

Process Design Suite supports communication with other systems via SOAP web services: it allows you to generate Task types that can be used to design Processes that act either as a Web Service Server or a Web Service Client):

- Web-service client tasks:
  - For every operation, the system generates one task type that sends a request to the web service server, and receives the response.
- Web-service server tasks:
  - Wait task that waits for a request from a client,
  - Response task that sends a response to the client,
  - Error task that sends a fault response to the client.

2.14.1 Web Service Server

To be able to implement a Process that will serve as a Web service server, you will need to generate the following task types and create the Process using these:

1. waitForInvoke: waits for a client request
2. sendResponseToInvoke: sends the client the output data
3. Optionally sendErrorToInvoke tasks: sends fault responses

waitForInvoke The workflow enters the *waitForInvoke* task, it waits until it receives a web service client call. Once, it has received a client call, the task finishes, and the Process instance execution proceeds. The task defines the following parameters:

- **input**: input of the call from the client request
- **requestId**: reference to the ID of the call

  The ID is used by the response tasks to identify the call; the value is generated by the Execution Engine.
- **principal**: reference to a slot that holds the principal who is performing the call
- **logXmlMessage**: if true, all XML messages received by the task are logged to the database and can be viewed in the `Webservices`.

  Important: Enabling logging of XML messages can result in significant database growth. Make sure to handle such risks appropriately.
- **requestHeaders**: reference to a slot that holds the HTTP headers of the request

sendResponseToInvoke The sendResponseToInvoke Task type sends a response to the wait point of the request ID. The task defines the following parameters:

- **output**: output sent in the client response
- **requestId**: reference to the ID of the call
- **logXmlMessage**: if true, all XML messages received by the task are logged to the database and can be viewed in the `Webservices`.

  Important: Enabling logging of XML messages can result in significant database growth. Make sure to handle such risks appropriately.
- **requestHeaders**: reference to a slot that holds the HTTP headers of the request

sendErrorToInvoke For the cases when the received request call is incorrect or fails, use this task to send a fault message to the client. For error responses, the task defines the following parameters:

- **error**: error sent in the error response to the client
- **requestId**: reference to the ID of the call
- **logXmlMessage**: if true, all XML messages received by the task are logged to the database and can be viewed in the `Webservices`.

  Important: Enabling logging of XML messages can result in significant database growth. Make sure to handle such risks appropriately.
- **requestHeaders**: reference to a slot that holds the HTTP headers of the request

2.14.1.1 SOAP Webservice Server

Depending on whether you have the WSDL file for the service, you will need to do the following:

- If you do not have the WSDL file, you need to model the input and output data types, and define a WSD definition (Generating Tasks for Web Service Server). Based on the WSD definition, the system generates the task types that you can use to implement a web service server in a process.
- If you already have a WSDL file, the system generates the data types and task types for the web service server directly based on the WSDL file (refer to Generating Tasks for Web Service Server from WSDL File).

On runtime, you can check and manage Model instances that are serving a web service call in the `Webservices`.

2.14.1.2 Creating SOAP Web Service Server from Scratch

To generate task types and design a process acting as a Web service server without a WSDL, you need to do the following:

1. Create the data types for the server.
   - input: input data type received from the web service client
   - output: output type of the data sent as a response to the web service client
   - error: data types for data sent to the client as a web service error (when the client sends invalid data to the server)

   All these data types must be defined as Records even if they contain only primitive values since they will be used in the WSDL and used for XML mapping in XSD.
2. Check the XML mapping of the Records, their Fields, and Relationships on the *XML Mapping* tab of their Properties. The mapping will be applied in the XSD and WSDL files of the service. Generally, the default mapping should work fine.

   To exclude a Relationship direction from the XSD schema of your server, mark the relationship direction as *XML Transient*: in the Properties view of the Relationship, select the *Target XML Mapping* or *Source XML Mapping* tab and select **XML Transient**.
3. Create a web service definition file:
   (a) Right-click the Module, go to **New** > **Webservice Definition**
   (b) Open the file in the web service editor and define the web service properties:
   - Service name: name used as the service name and the task name in the generated task types
   - Service namespace: namespace of the Web service
   - Input type: input data type you defined above
   - Output type: output data type you defined above
   - Soap fault types: fault data types you defined above

   (c) If applicable, select *Generate optional parameters for access to HTTP headers*: if selected, the generated task type contain the properties for setting of responseHeaders.
   (d) Click **Generate**:

   The system generates the following:
   - XSD schema file, which will make use of the data types
   - WSDL file that imports the XSD schema.
   - Webservice server task types
4. Design the server Process workflow with the generated task types.

Important: Make sure to define the id parameters on the **waitForInvoke**, **sendResponseToInvoke**, and **sendErrorToInvoke**. Failing to do so, might result in transaction rollback.

5. Design the process workflow with the Webservice-Server Tasks.

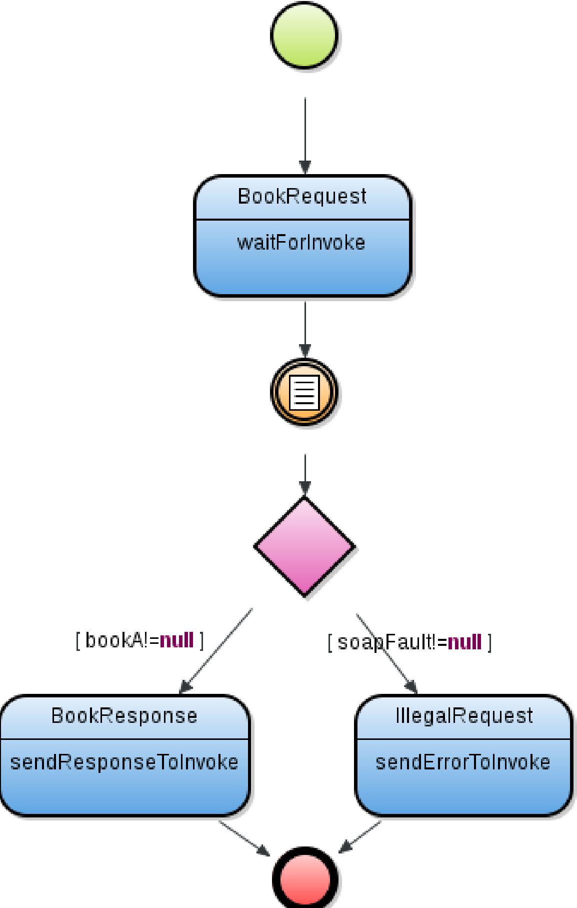


Figure 2.155 Workflow of a Process Serving as a Web Service Server

6. Expose the WSDL and XSD files as appropriate.

Important: If the timeout period elapses and the server has not sent any response, the server timeout response is sent to the client. The default time out is set to 10 seconds.

2.14.1.3 Creating SOAP Web Service Server from WSDL

To generate task types and the underlying data types for a Process that will serve as a Web service server from a WSDL file, do the following:

1. In the GO-BPMN Explorer, right-click the respective module and in the context menu click Generate > Webservice Server.
2. In the Generate Webservice Server dialog box, enter the following:
   (a) In the WSDL location text field, type the location of the WSDL, possibly a URL.

(b) Select the optional task parameters for HTTP headers. The system will create a copy of the WSDL in the Module and create the respective Records and task types. Consider global variables for the input, output, and error data types that will hold the data during the web service invocation.

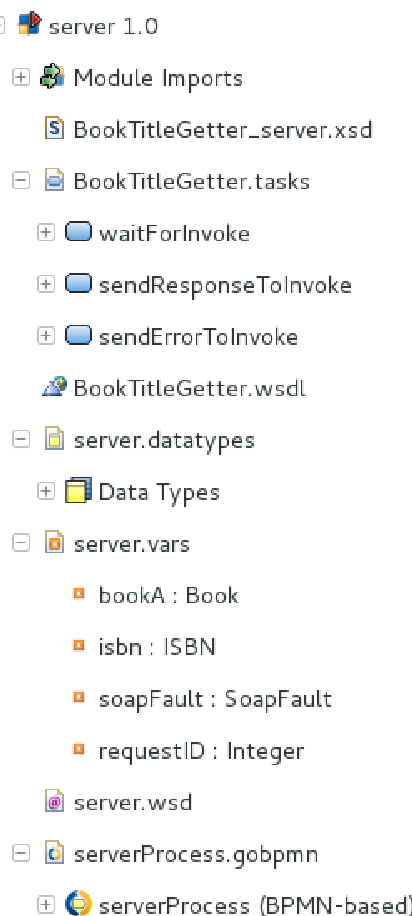


Figure 2.156 Web Server Service Module

3. Design the process workflow with the generated Webservice-Server Tasks.

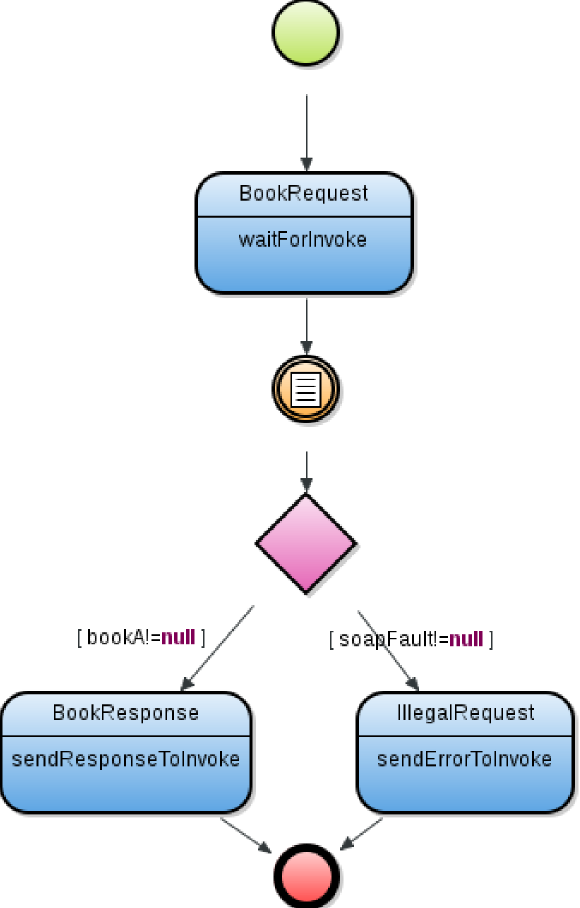


Figure 2.157 Workflow of a Process Serving as a Web Service Server

4. Expose the WSDL and XSD files as appropriate.

2.14.2 Web Service Client

A Process that acts as a web service client must contain artifacts generated on hand of the WSDL file of the target web service. The artifacts allow the Process to send a valid request to a web service server and receive and process the server response.

The generated client artifacts include the invoke task types that perform the calls, and the underlying data types:

- invoke task types The tasks of the invoke task type perform the web service operation calls:
- fault data types Fault data types are used to create variables, which can store the returned web service errors.
- data types used by the generated task types

Note: Only SOAP 1.1 document and literal style Web services are supported.

Web Service Task Types

Every generated task type represents one WSDL operation. Its messages and other properties are defined by its properties:

- isSynchronous: if true the web service call is synchronous If synchronous, the entire process execution stops until the server response is received.

  If asynchronous, other parts of the process can continue their execution (other tokens in the process continue their execution); for example, parallel branches continue, while the branch with the web service client task waits for the server response.

- input: input data sent to the web service server
- requestHeaders: custom headers of the request
- requestSoapHeaders: soap headers sent with the request
- output: reference to a slot that stores the server response
- responseHeaders: reference to a slot that stores the response headers
- responseSoapHeaders: list of references to variables

  The variables will be filled with the received soap headers if they match any of the present headers.

- endpointAddress: target webservice endpoint URL address If null, the endpoint address from metadata is used.
- logXmlMessage: database logging of messages

  If true, all XML messages received by the task are logged to the database (false by default).

  Note: Enabling logging of XML messages can result in significant database growth.

- error: If the web service server return soap: fault in web service response, fault is stored in this variable. If null, soap fault is discarded.
- readTimeout: read timeout in milliseconds After the defined time period elapses, a BPMError is thrown. You can handle the possible error with an Error Intermediate Event. If undefined the timeout is set to zero (0) and hence no timeout is applied (the timeout is infinite).
- login: login name used to access the web service HTTP basic authentication is used.
- password: password for HTTP basic authentication

2.14.2.1 Creating SOAP Web Service Clients

Important: Resources for a web service client can be generated only for SOAP 1.1 web services.

To create data types and tasks that will constitute a Process that will act as a web service client, do the following:

1. Get the respective WSDL and any other related resources. Web service WSDL is usually available on a URL provided by the web service server.

   When generating web service client for web service server Process, include the required XSD files.

2. In the GO-BPMN Explorer, right-click your Module and in the context menu, click Generate > Webservice Client.

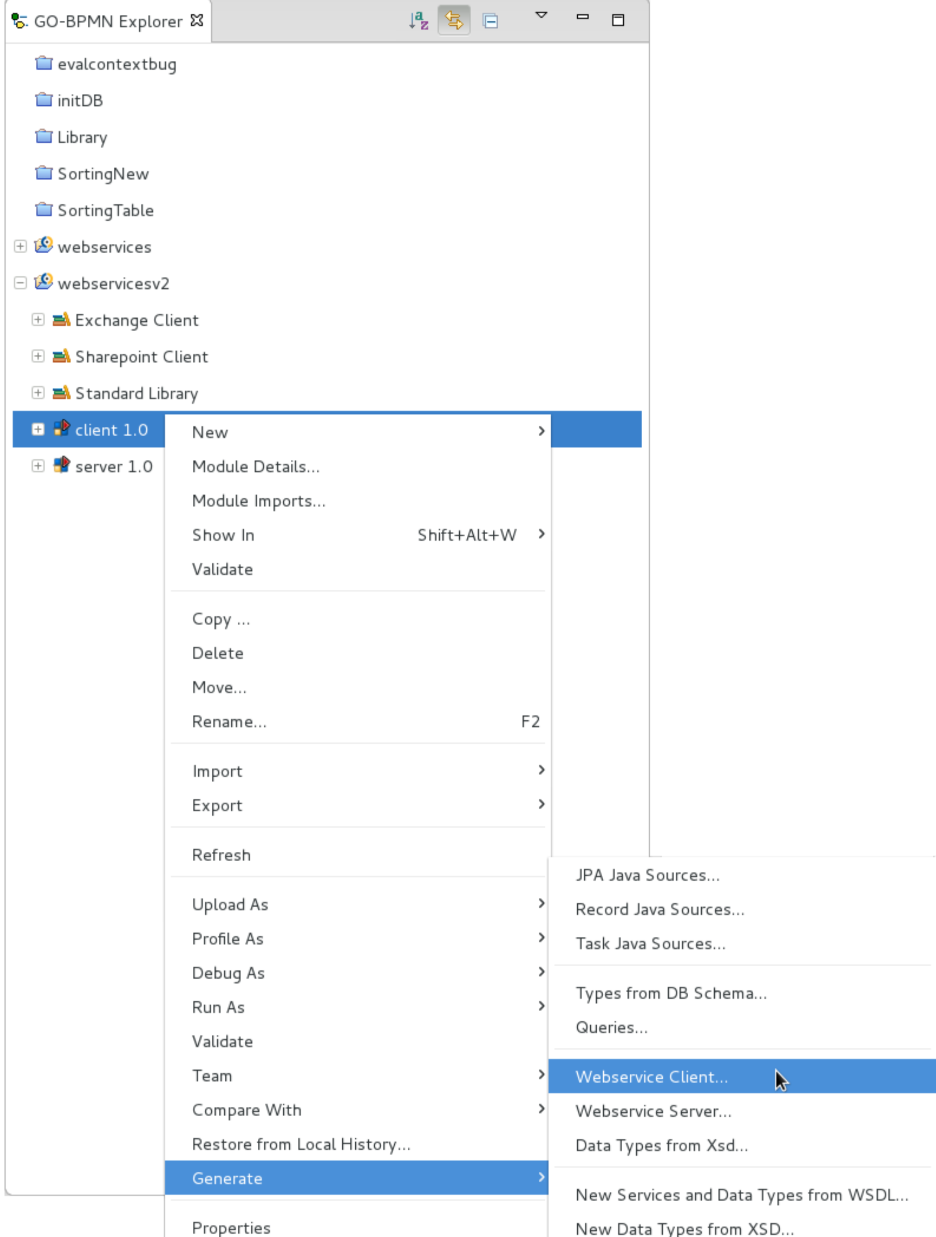


Figure 2.158 Generating Artifacts for Web Service Client

3. In the Generate Web Service Client dialog box:

   (a) Enter the path to the target Module.

   (b) In the WSDL location text field, type the URL or file system location of the WSDL.

   (c) Select the additional optional task parameters to be generated.

   (d) Click OK.

   (e) Select the web service operations you want to use in your process: for every operation, one task type and the respective data types are generated. Tasks of the task type call the webservice server and request the operation.

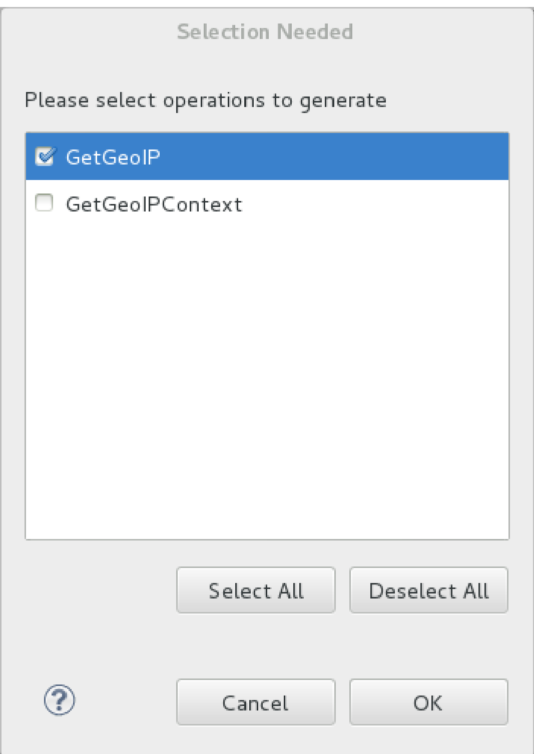


Figure 2.159 Selecting relevant operations

(f) If necessary, adjust the XML mapping of the data types: open the Properties view of the record and on the XML Mapping tab, change the mapping.

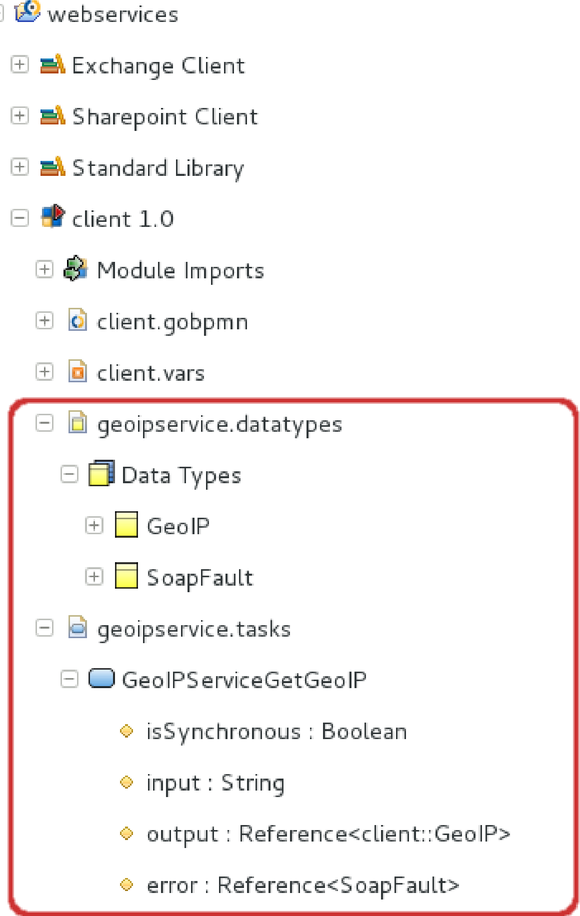


Figure 2.160 Generated Task Types and Data Types

4. Design a Process that uses the generated task types.

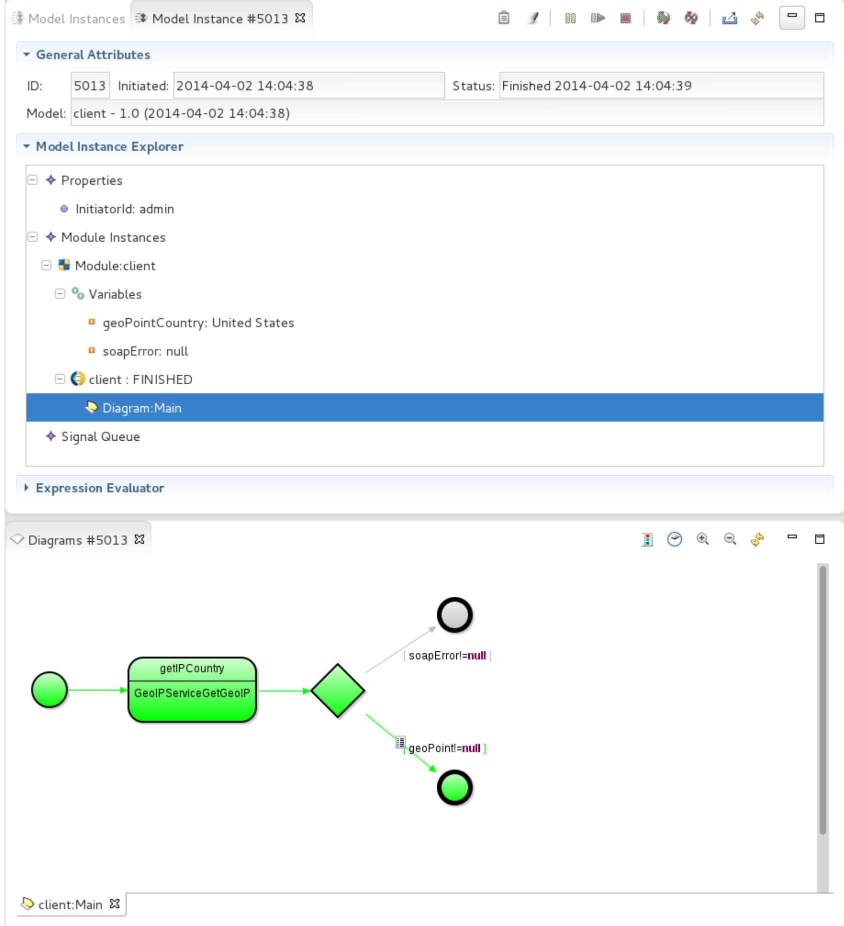


Figure 2.161 Model instance of a web service client Process

2.15 Monitoring

Monitoring is based on a special type of assignment referred to as the *monitoring assignment*. The monitoring assignment is a type of `assignment`: it defines what and where is assigned. The involved data is considered monitoring data, the metrics, and is accessible to reports and dashboard widgets of the BAM application.

The monitoring data is uploaded as part of the model and then accessed by the reports and widgets with metric from the BAM Application Monitor.

Note: Process Design Suite uses the BIRT framework for reports and widgets (for further information refer to the respective BIRT documentation).

2.15.1 Defining Monitoring Data

To define the data and their layout in the Business Activity Monitor, you need to do the following:

1. Define the monitoring data: In PDS in the Properties view of the respective process elements, specify the monitoring assignments on the Monitoring tab.

2. Define the report or widget that will operate over the monitoring data: In PDS in your module, create the report or widget definition file, and create the report or widget in the file.
3. Define the report design that will determine how the data in the widget or report are presented in the Business Activity Monitor: Switch to the Report Design perspective and create a new report design.

Note that example report designs are available: Go to **File** > **New** > **Examples** > **Common Monitoring** under Process Design Suite.

2.15.1.1 Creating Reports

A report is a static visualization of the current business intelligence data in a specific layout. The data is loaded in the moment the report is visualized.

The layout and data sources can be defined in any of the supported business intelligence tools: A report may access data from LSPS as well as any other available external business data available in a database.

> **Note:** To create a report that contains data that depend on the logged user, the report has to define the parameter currentUserLogin of the type String and set as hidden. When the report is invoked, the parameter is automatically filled with the login data of the person in the Business Activity Monitor.

To define a report, do the following:

1. In the GO-BPMN Explorer view, double-click the respective report definition.
2. In the Reports area, click Add.
3. In the Report Details area, define the report properties:
   - Name: report name
   - Title: report title
   - Design file name: design file created in a business intelligence tool with the report layout
     You can create a design file in the Report Design perspective, which is a standard BIRT perspective. For further information, refer to the BIRT documentation.
   - Performers: set of roles or performers who are allowed to access the report
   - Description: free text

2.15.1.2 Creating Widgets

A widget is a dynamic visualization component of business intelligence data: Widgets are placed on dashboards of the Business Activity Monitor application. They define a specific layout filled with runtime data loaded on every refresh, manual or automatic. The widget layout and data sources can be defined in any of the support business intelligence tools.

Widget definitions are created in a widget definition file. One Module may contain multiple widget definitions.

To define a widget, do the following:

1. In the GO-BPMN Explorer view, double-click the respective widget definition.
2. In the Widgets area, click Add.

3. In the Widget Details area, define the widget attributes.

    - Name: widget name
    - Title: widget title
    - Design file name: design file created in the business intelligence tools

    You can create a design file in the Report Design perspective, which is a standard BIRT perspective. For further information, refer to the BIRT documentation.

    - Default width: default widget width when visualized on the dashboard
    - Default height: default widget height when visualized on the dashboard
    - Refresh rate: period of data refresh
    - Performers: set of roles or performers who have the access rights to the widget
    - Description: free text

    Note: To create a widget that depends on data about the logged-in user, define the parameter current↩ UserLogin of the type String on the widget definition and set as hidden. When the widget is displayed, the parameter is automatically filled with the login data of the person currently logged in the Business Activity Monitor.

2.15.2 Business Activity Monitor

Business Activity Monitor is a web-based dashboard application for the visualization of reports and widgets with metrics. The application enables you to work with monitored data (KPIs) in real time.

It displays metrics in reports or widgets on a dashboard. Note that both reports and widgets are defined as model resources.

The web application is accessible from `http://<LSPS_DOMAIN>/lsps-monitoring`, when running on localhost `http://localhost:8080/lsps-monitoring`.

The Business Activity Monitor contains by default the following:

- Dashboard tab is a private tab displayed by default. It is located in the area where all dashboard tabs, that is, both the private and common tabs are open.
- Reports tab contains a list of available reports.
- Common tabs contains a list of common tabs.

Note that you can delete the default private Dashboard tab and create an arbitrary number of private tabs. However, at least one dashboard tab must be available.

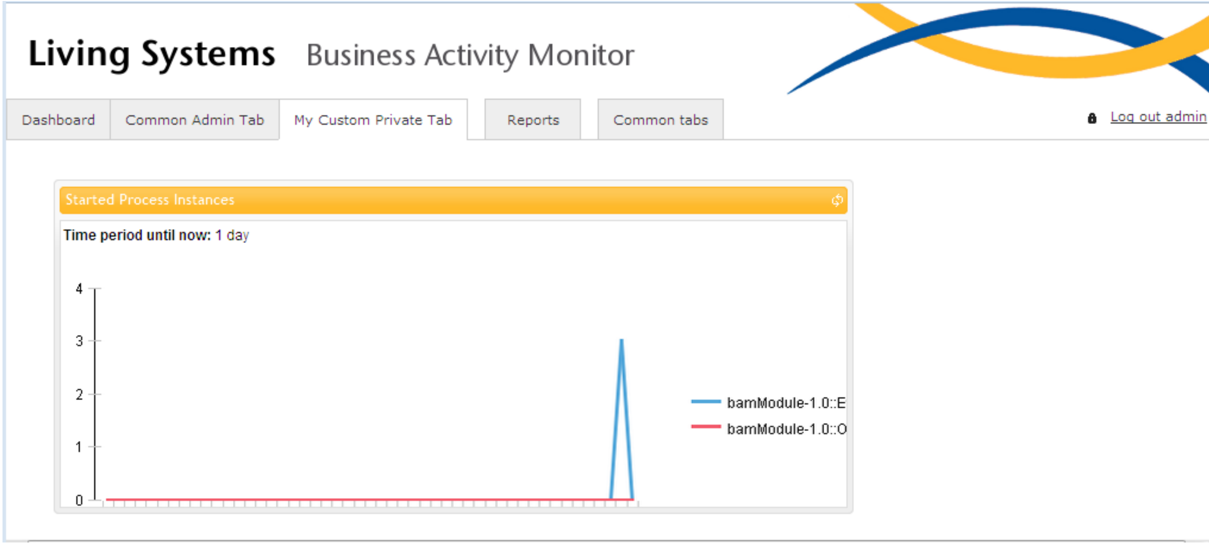


Figure 2.162 Business Activity Monitor with multiple common and public tabs

The Business Activity Monitoring application is supported on the following browsers:

- Google Chrome 23 and newer
- Mozilla Firefox 17 and newer
- Internet Explorer 8 and newer
- Safari 6 and newer

2.15.2.1 Customizing Dashboard

The Business Activity Monitor is locked by default to prevent accidental changes. To change the properties and content of dashboard tabs, click the Edit Dashboard button to enable editing of all dashboard tabs.

To save any changes made to the particular dashboard tab, click the Save Dashboard Changes ( ) button. To revert all changes on the dashboard tab since unlocking, click the Revert Dashboard Changes button ( ) button.

To lock the dashboard tab, click the Cancel Dashboard ( ) button.

2.15.2.1.1 Dashboard Tabs

The Business Activity Monitor dashboard is divided in tabs; each tab contains its set of widgets.

Depending on the accessibility of tabs, the application can contain two types of dashboard tabs:

- A private tab can be created by the currently logged-in user and is visible only to this user.
- A common tab can be created by a user with administration rights and is visible to multiple users.

2.15.2.1.2 Private Tabs

Any user of the Business Activity Monitor can create their private tabs with custom widgets and tab properties. These tabs cannot be displayed by other users.

2.15.2.1.2.1 Adding and Removing Private Tabs

Before you can add or remove private tabs, unlock the dashboard for editing.

To add a private tab, click the Add New Dashboard Tab (+) next to the tabs labels. To remove a private tab, click the Delete (✖) in the label of the tab.

2.15.2.1.3 Common Tabs

A common tab is a dashboard tab available to a set of users (for information on user management, refer to the chapter Persons in the Process Design Suite User Guide).

To be able to create common dashboards, you need to have the respective security rights.

2.15.2.1.3.1 Defining Common Tabs

A user with administration rights can create common tabs that are available in the Common Tabs of a group of users. Such a user must have the respective security role assigned.

To create a common tab, do the following:

1. Display the `Common tabs` tab.
2. Click the Add New Dashboard Tab link and define the tab configuration.
3. Click the Close button.
4. Click the Save button to confirm the creation of the common tab.

You can manage and delete the common tabs on the Common tabs tab.

Display the tab by clicking its name in the Show hidden common tabs menu (▾) and insert the required widgets.

2.15.2.1.3.2 Hiding and Showing Common Tabs

The logged-in user has the right to view one or more common tabs.

To hide and re-display a common tab, do the following:

1. Unlock the dashboard.
2. Click on the common tab you want to hide.
3. Click the Delete (✖) button in the tab label.
4. To display a hidden common tab, click its name in the Show hidden common tabs menu (▾).
5. Lock the dashboard.

2.15.2.1.4 Renaming Tabs

To change the label of a displayed tab, make sure the dashboard is unlocked, click the tab label, enter the new name, and press Enter.

2.15.2.1.5 Rearranging Tabs

To rearrange the displayed tabs, drag-and-drop the tabs by the Drag to reorder dashboard tabs ( ) button.

2.16 Debugging

To debug your models, run LSPS Server in debug mode, add breakpoints to modeling elements in your diagrams and expressions, and run the model on an LSPS Server; both the JVM and the LSPS Server must run in debug mode.

2.16.1 Debugger Implementation

To debug your models, both the Execution Engine and the underlying JVM must run in debug mode. After you start the Execution Engine and JVM in debug mode, you can enable the debug mode in PDS. At that moment, the following happens:

1. LSPS Debugger creates in the Execution Engine Java breakpoints: these serve to stop the execution when it encounters an LSPS breakpoint. You can check that the Java breakpoints are in place in the Breakpoints view. If you remove these Java breakpoints, the LSPS debugging will be disabled.
2. Any LSPS breakpoints present on the server are discarded.
3. The LSPS breakpoints defined in PDS are uploaded to the server.

When you then run a model instance in debug mode with LSPS breakpoints on its modeling elements and expressions, and the execution hits a breakpoint, in an execution step, the breakpoint condition is evaluated and if the condition evaluates to true, the LSPS Debugger notifies the Java Debugger to suspend the execution in the execution step.

An execution step is any of the following:

- Sending a token to a process element
- Changing of goal status
- Interpreting an expression
- Assigning a value to a variable
- Assigning a value to a Record property
- Creating a record instance

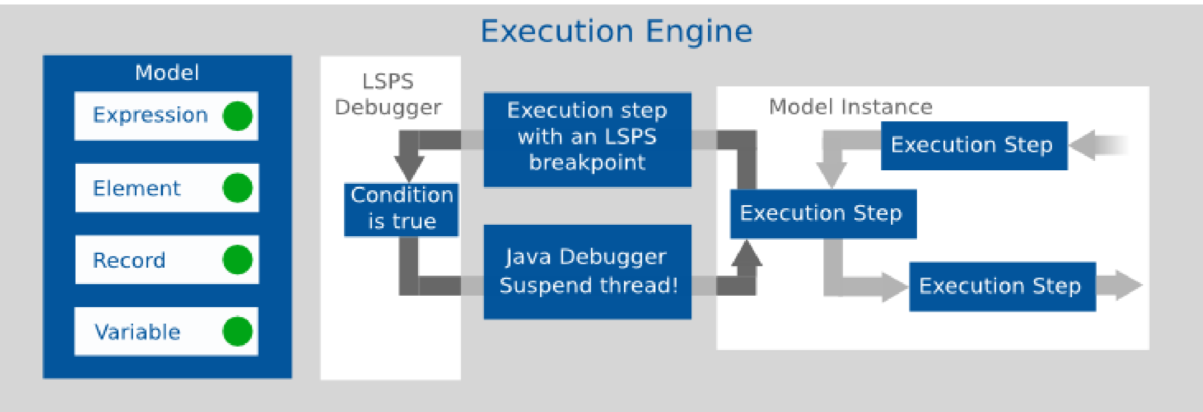


Figure 2.163 LSPS debugger implementation

2.16.2 Setting up Debugger

You can connect the LSPS Debugger and debug a model instance that runs on an Execution Engine in a local JVM or on a remote JVM. Generally, you have the following options:

- **Local**:
  - If you are debugging models and run the Default Application User Interface, debug model instances locally on the PDS Embedded Server
  - If you are developing your custom Application User Interface, debug model instances locally on the SDK Embedded Server.
- **Remote**:
  - If you need to debug on a remote LSPS server connect to the remote server.

Generally, the connection of the LSPS Debugger to a server, remote or local, is defined in a debug configuration. However, the connection to the PDS Embedded Server is integrated in PDS and it is sufficient to modify the server properties.

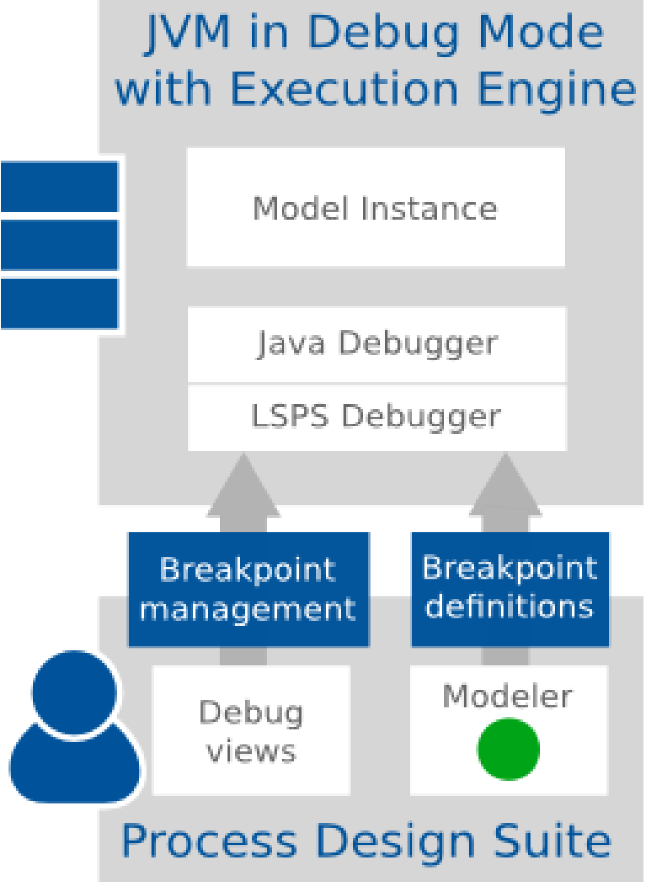


Figure 2.164 Debugger schema

2.16.2.1 Debugger on the PDS Embedded Server

To run the PDS Embedded Server in debug mode, do the following:

1. Go to **Runtime Connections** > **Runtime Connection Settings**, select *LSPS Embedded Server* and click **Edit**.
2. In the VM Arguments field, add arguments that will run the JVM and the Execution Engine in debug mode:

   ```
   -agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=4000 -DlspsDebug=true
   ```

   The `-DlspsDebug=true` argument runs the Execution Engine in debug mode; the address argument defines the target port: you might provide another port number if required.

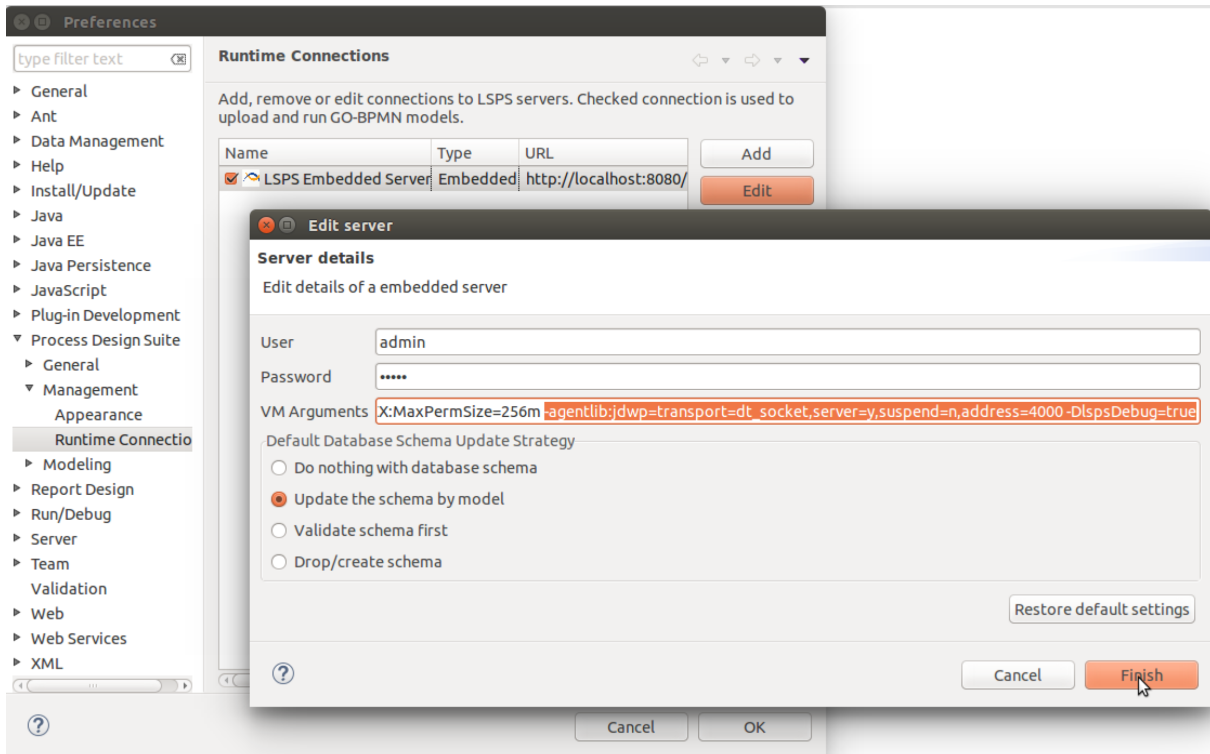


Figure 2.165 Debug parameters in the PDS Embedded Server configuration

Important: Make sure the user used to establish the connection, in the example it is the admin user, has the security right for debugging: run the server, go to the Management perspective; and in the Person view, check that at least one security role of the Person has the `Debugger↩ :Manage` permission.

3. Click **Finish** and then **OK** in the error notification dialog.

4. Start the PDS Embedded Server.

5. Create configuration and connect the Debugger:

   (a) Switch to the Debug perspective.

   (b) Go to Run > Debug Configurations > Remote Java Application.

   (c) Click the New button and on the right define the details of the server:

      - Set the host to `localhost` and port the target port (from the vm arguments above `4000`).
      - Enter a Java project name into the Project field (create an empty Java project if no other Java project is available in the workspace).

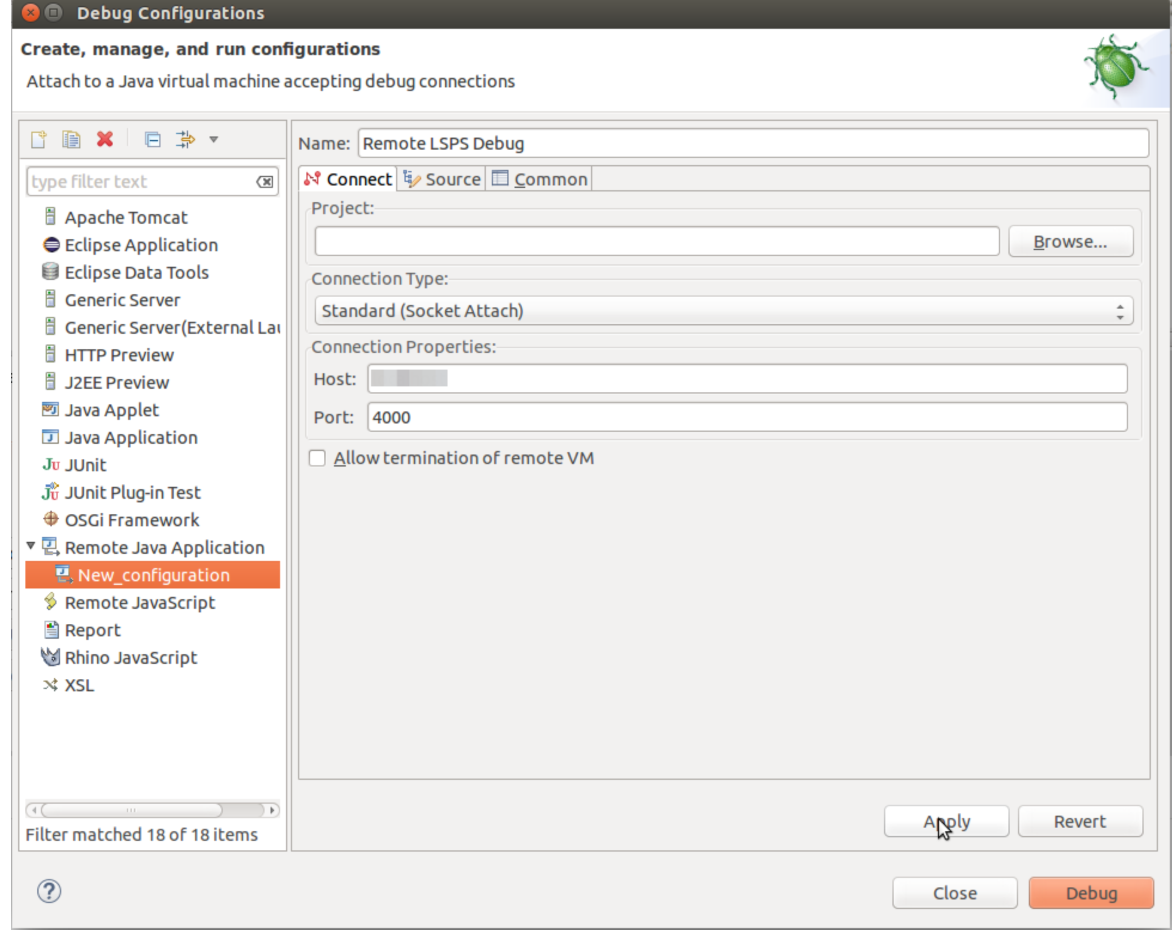


Figure 2.166 Setting debug configuration

(d) Run the debug configuration to attach the LSPS Debugger to the server.

6. Enable debugging: click the **Debug** button in the **LSPS Breakpoints** view.

7. Now you can debug the model.

2.16.2.2 Debugger on the SDK Embedded Server

To debug your Custom Application User Interface locally on the SDK Embedded Server, do the following:

1. Open the *Debug* perspective.

2. Go to Run > Debug Configurations > Java Application > <YOUR_APP_LAUNCHER> (for example, `My CustomApp Embedded Server Launcher`).

3. Edit the configuration: add the `-DlspsDebug=true` VM argument.

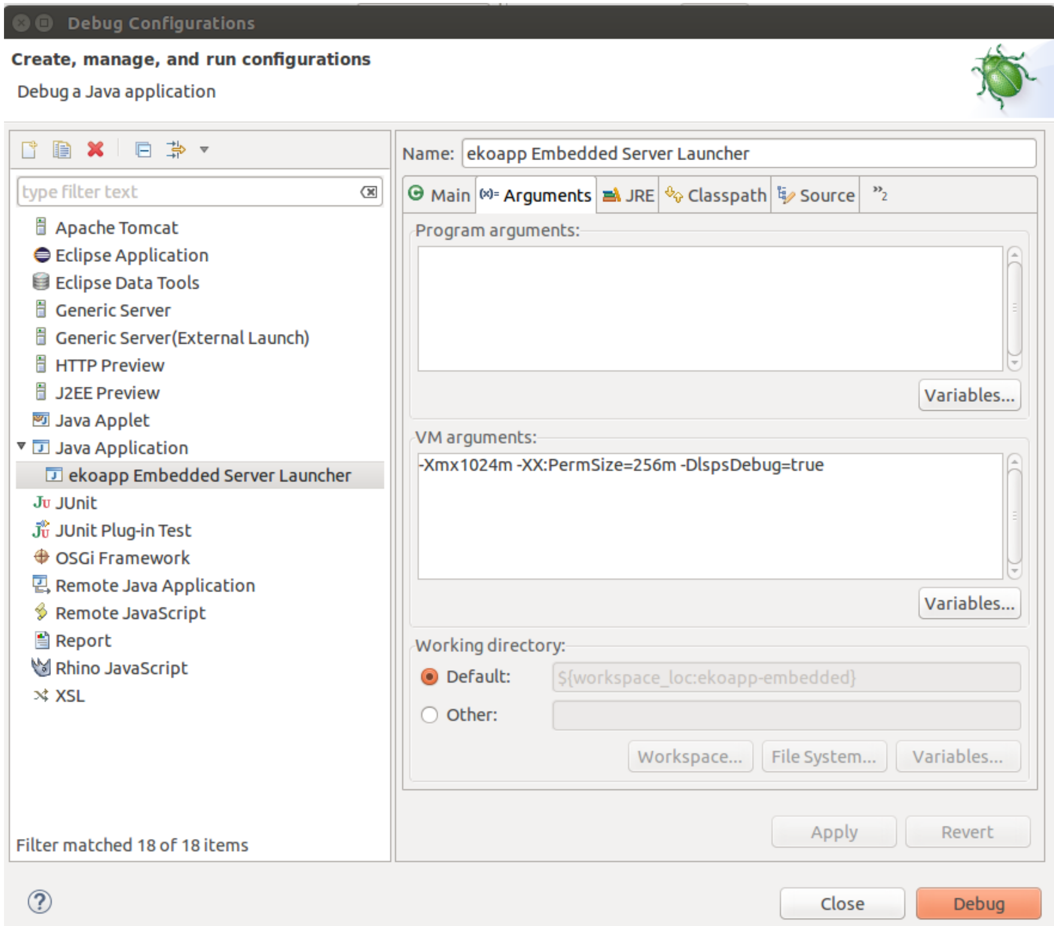


Figure 2.167 Setting application launcher configuration to debug mode

Under the hood, the connection is established on a free port automatically.

4. Run the launcher configuration in debug.

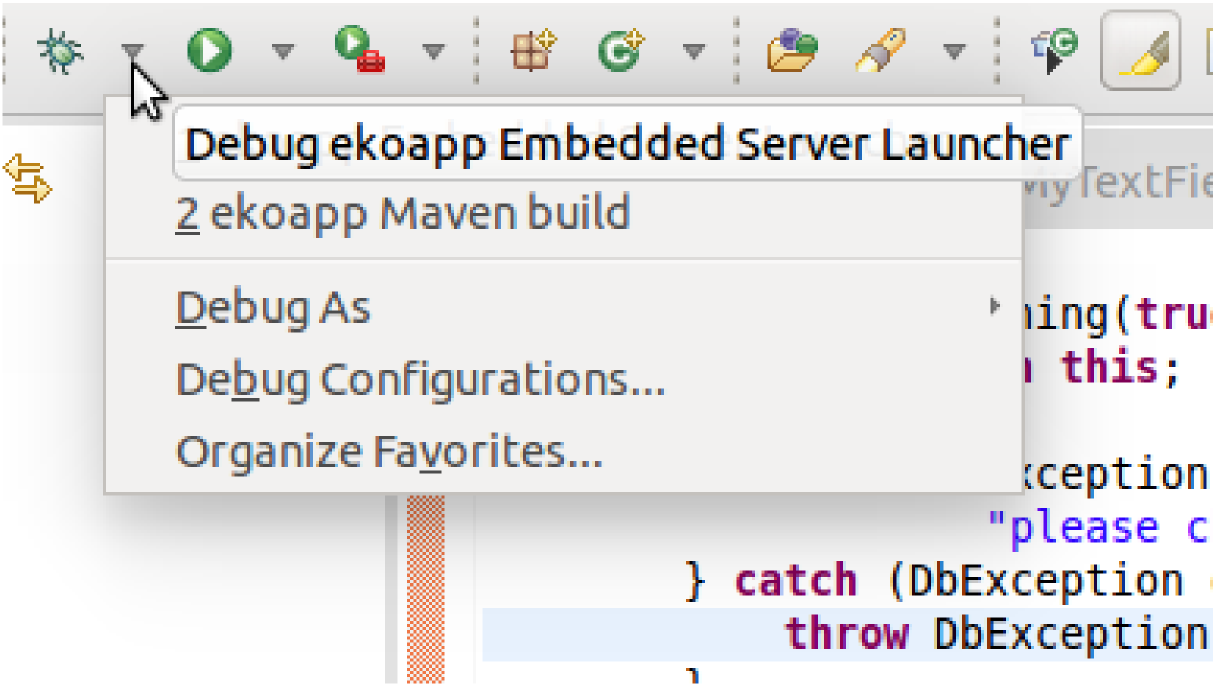


Figure 2.168 Running the launcher in debug

5. Enable debugging: go to the Debug perspective and click the Debug button in the LSPS Breakpoints.

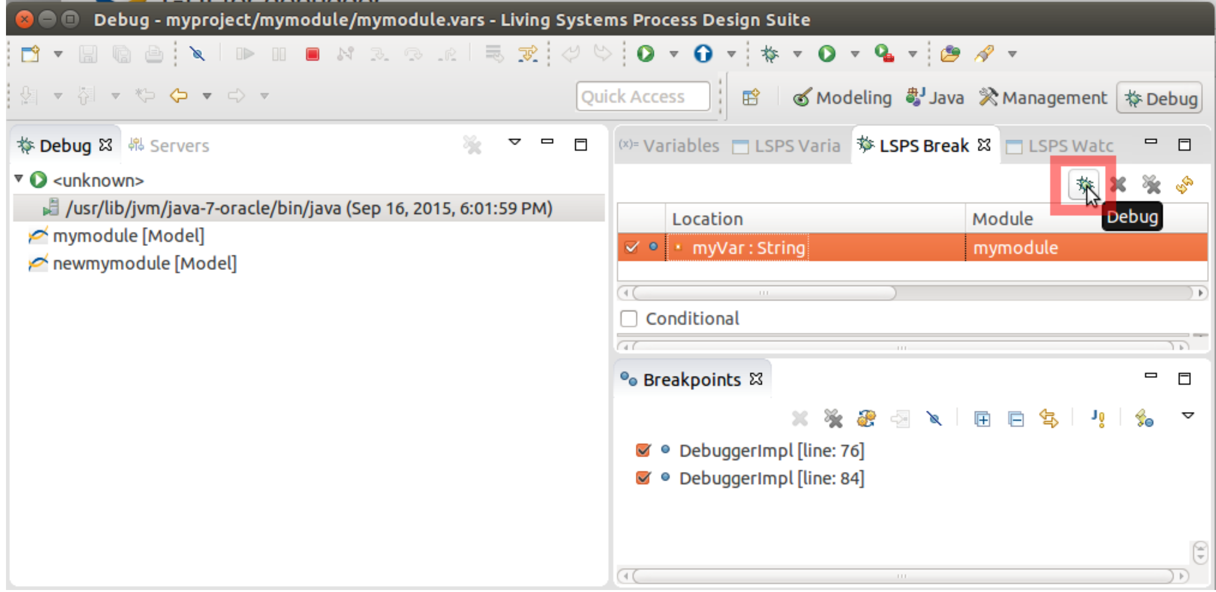


Figure 2.169 The LSPS debug is enabled: the Java breakpoints are in place.

6. Now you can debug the model.

2.16.2.3 Debugger on a Remote Server

Important: Only one user can connect to the Execution Engine in debug mode since, on connection establishing, any LSPS breakpoints on the server are discarded. If you need to prevent such a scenario,

restrict the access of users to the debug feature: remove from persons any security roles with the `Debugger` security right.

To set up LSPS debugging, do the following:

1. Make sure the server with the LSPS Execution Engine runs in debug mode with the VM arguments:

```
-agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=<PORT> -DlspsDebug=true
```

2. Open the Debug perspective.
3. Go to Run > Debug Configurations > Remote Java Application.
4. Click the New button and on the right define the details of the remote server.

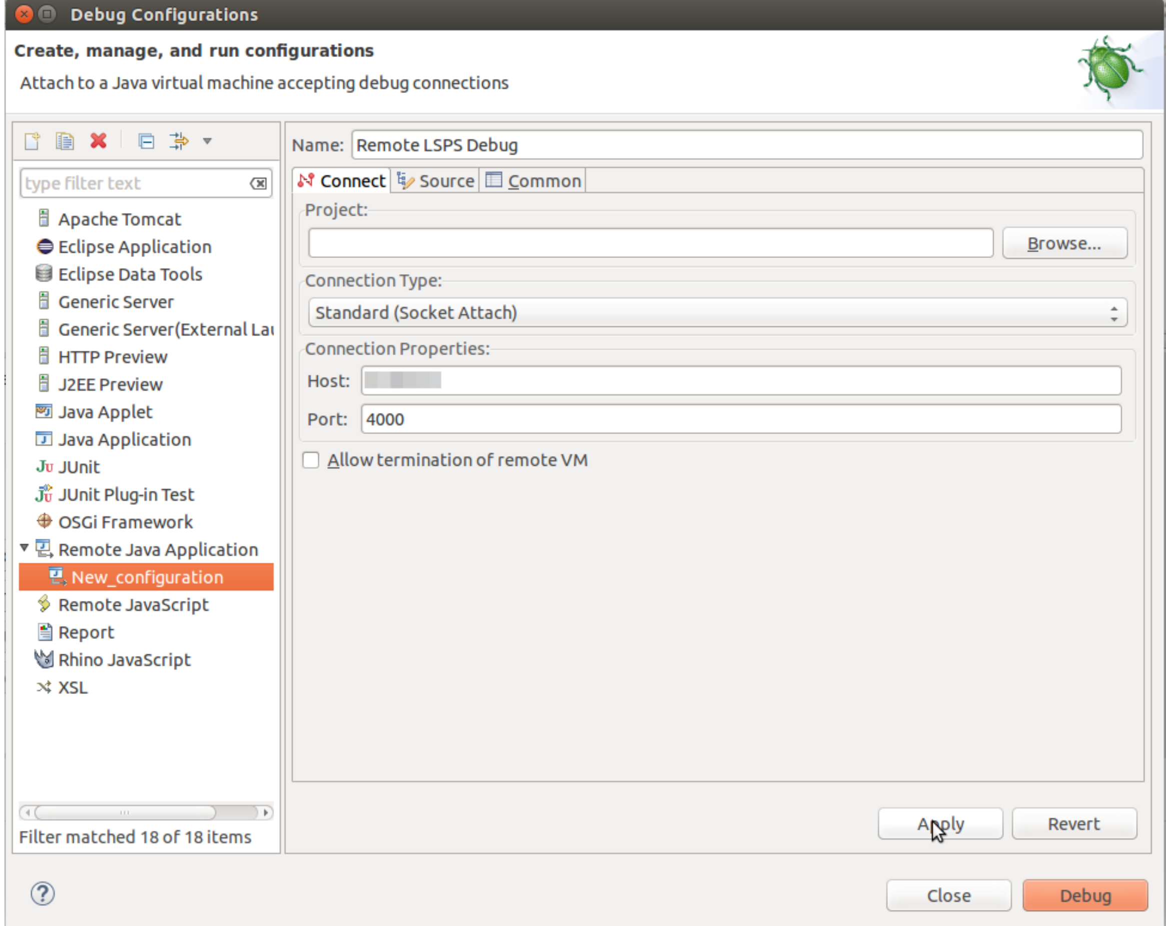


Figure 2.170 Setting connection to a remote server

5. Run the debug configuration to attach the LSPS Debugger to the server.
6. To start debugging, go to the Debug perspective and **click the Debug button in the LSPS Breakpoints** to enable the debugging.

   When debugging is enabled, the following takes place:

   (a) LSPS Debugger creates two Java breakpoints in the Execution Engine that are used to stop the execution on LSPS breakpoints.

   (b) Any LSPS breakpoints present on the server are discarded.

(c) PDS uploads all LSPS breakpoints to the server.

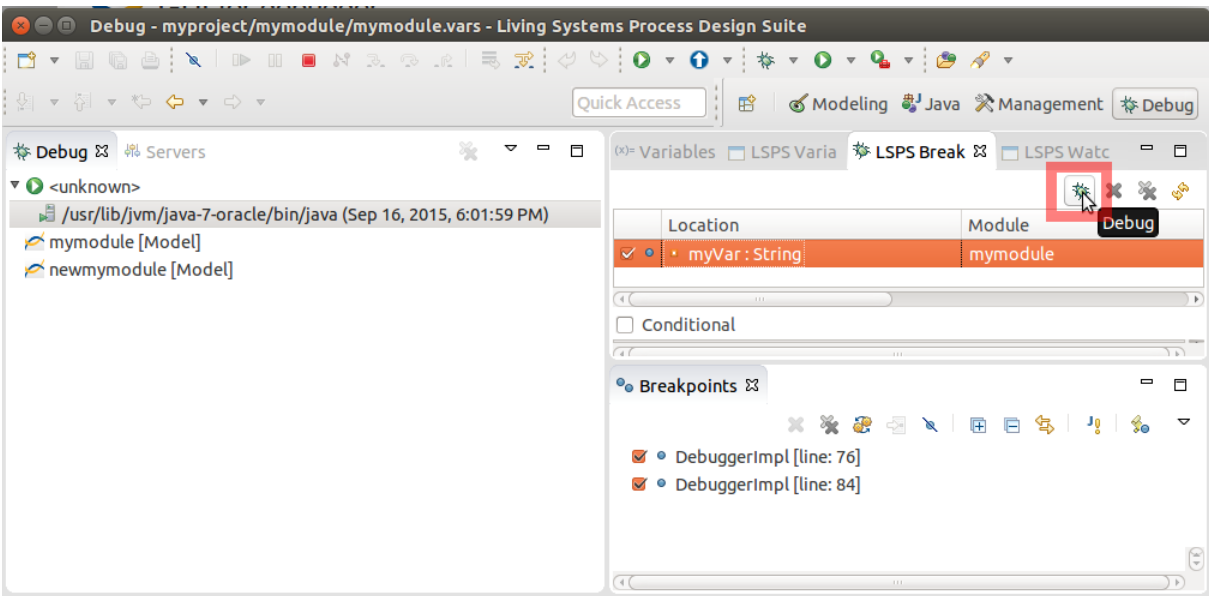


Figure 2.171 The LSPS debug is enabled: the Java breakpoints are in place.

7. Now you can debug the model: add breakpoints to the model and run the model.

2.16.3 Debugging a Model

Once you have established connection to the LSPS debugger, and enabled debugging, you can start debugging your model:

1. Switch to the Debug perspective.
2. Make sure the *Debug* view contains the server threads and the *Breakpoints* view the DebuggerImpl breakpoints.

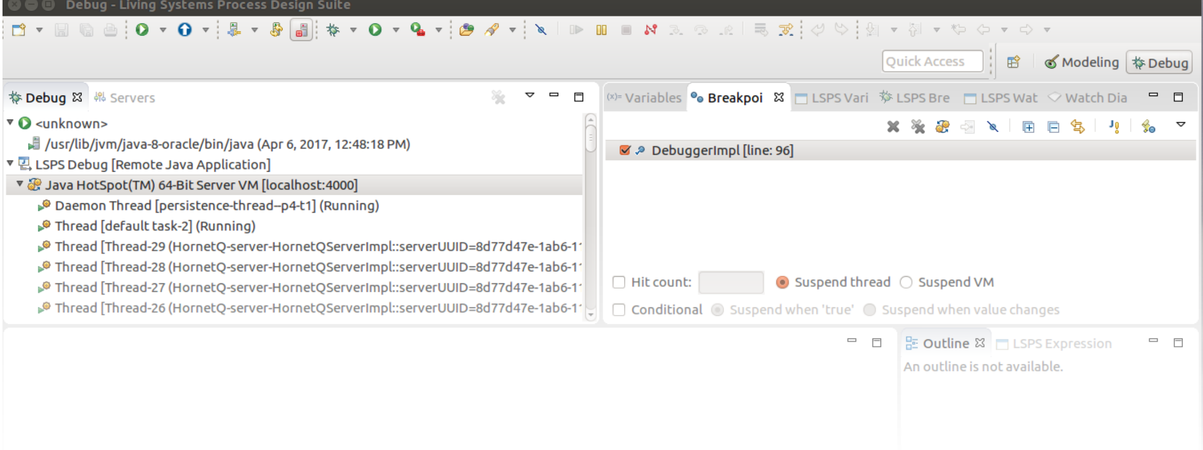


Figure 2.172 Local Embedded server in debug mode and debug enabled

3. If applicable, connect the debugger: run the remote configuration.
4. Switch to the Modeling perspective:
   (a) Add breakpoints to the module resources.
   (b) Run the Model.
5. Switch to the Debug perspective.
6. If at any point the views do not contain the correct data, double-click the first method in the current thread to refresh the views.

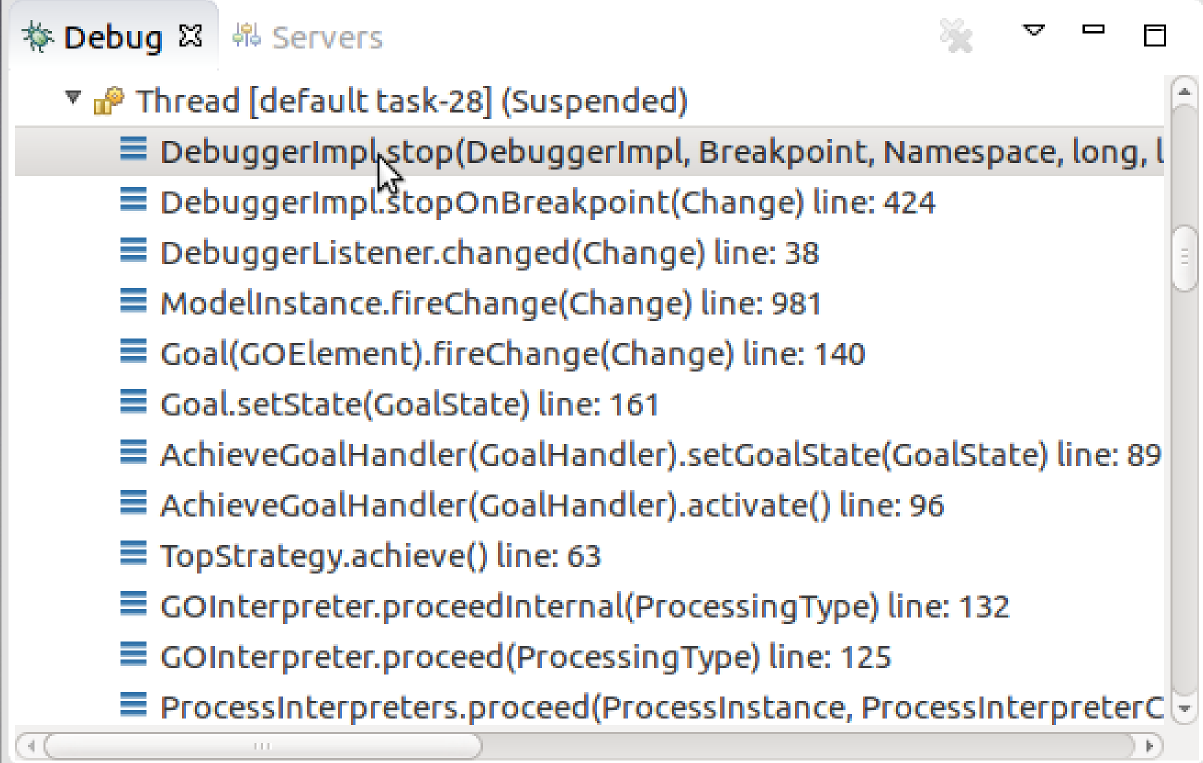


Figure 2.173 Caption text

7. Manage the debug execution with the buttons in the **LSPS Expression** view toolbar.

When the model instance hits a breakpoint, the Java thread of the Execution Engine is suspended. Note that the model instance state remains unchanged.

Once suspended, you can examine the model instance in the following views of the Debug perspective:

- *Watch Diagram*: diagram with the current breakpoint if the breakpoint is on a modeling element
- *LSPS Expression*: expression with the current breakpoint if the breakpoint is on an expression
- *LSPS Variables*: variables from the current execution context

  This includes modeling elements and their properties, variables, etc.
- *LSPS Watch Expressions*: expressions which are evaluated as the model is debugged

2.16.3.1 Adding and Removing Breakpoints

To add or remove a breakpoint to an element, do the following:

1. In the Modeling perspective, open the resource with the respective element, such as, variable definition or an expression.
2. Right-click the element and click Add Breakpoint or Remove Breakpoint.

You can add breakpoints to the following:

- expressions: right-click the expression and select **Add Breakpoint** from the context menu.
- BPMN modeling elements: right-click the element on the canvas and select **Add Breakpoint** from the context menu.
- Goals: right-click the element on the canvas and select **Add Breakpoint** from the context menu.
- variables: right-click the variable definition and select **Add Breakpoint** from the context menu.
- records or their properties: in the Outline view, right-click the record or its property and select **Add Breakpoint** from the context menu.

Note: Breakpoints on method or function signatures are ignored: if you need to stop at the function or method start, insert the breakpoint on the first expression in its body.

An added breakpoint appears in the LSPS Breakpoints view in the Debug perspective and is displayed as a dot marker in the diagram.

When the breakpoint is triggered, it is visualized either in the *Watch Diagram* or in the *LSPS Expression* view, depending on the breakpoint type.

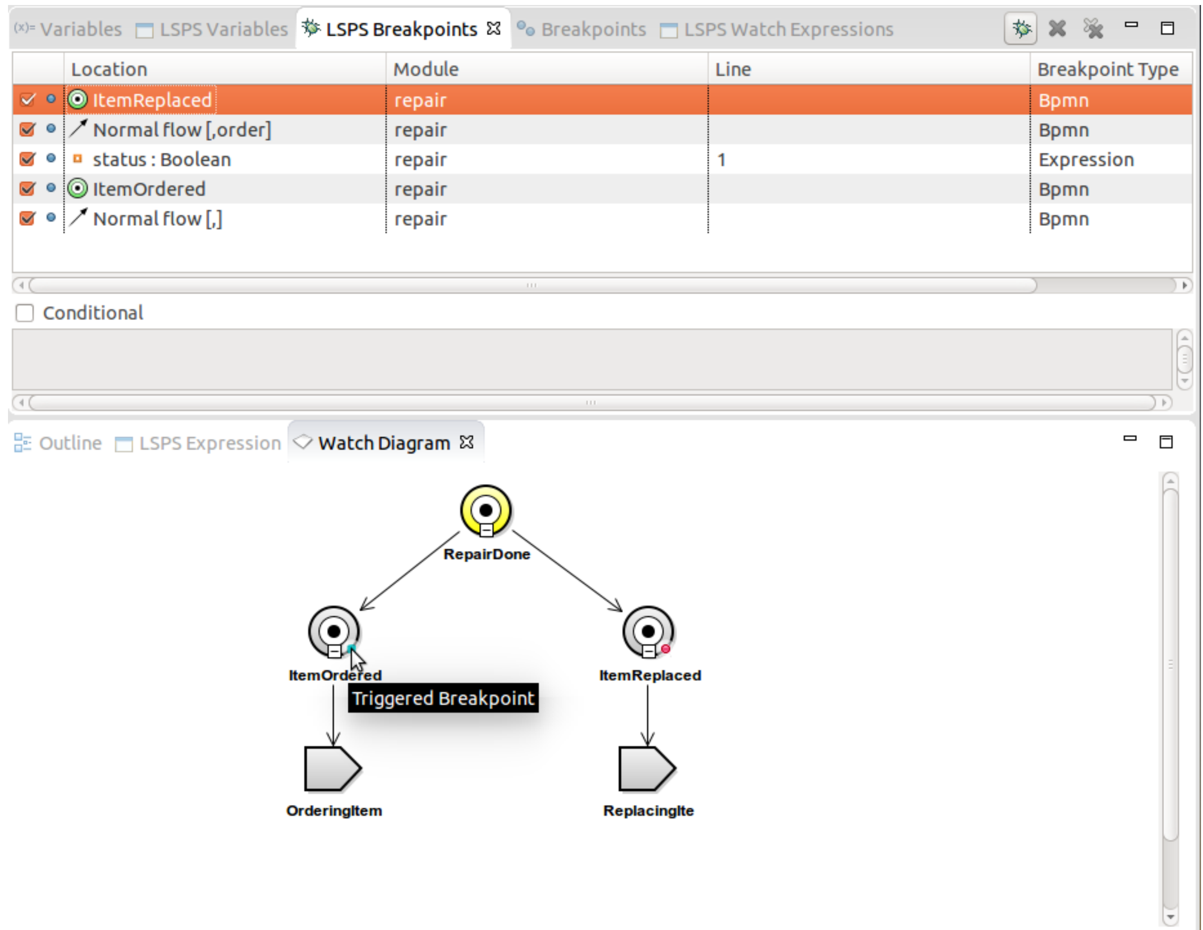


Figure 2.174 Active breakpoint in a Goal hierarchy

2.16.3.2 Defining a Breakpoint Condition

If a breakpoint defines a condition, it is activated only if the condition evaluates to true when the breakpoint is hit.

To add a condition to a breakpoint, select the breakpoint in the LSPS Breakpoints view and define the condition in the Condition area below.

> **Important:** It is not possible to define a breakpoint on a loop with a condition so as to have the breakpoint activated at a certain iteration.

2.16.3.3 Resuming Breakpoints

To resume a model instance to the next breakpoint, click the respective button, such as Resume, Step Over, etc., in the LSPS Expressions view.

2.16.3.4 Enabling and Disabling Breakpoints

You can disable and enable LSPS breakpoints as well as define breakpoint conditions in the LSPS Breakpoints view.

2.16.4 Tracker

> **Note:** Execution Tracker is deprecated. Use the Debugging feature.

To acquire information on runtime data, you can use the Execution Tracker: it provides details about actions during model instance execution; typically you will run tracking only for that part of the model execution you are interested in.

You can enable the Tracker by running the server with the `-Dcom.whitestein.lsps.lang.Tracker=true` JVM option. To start and stop tracker, go to the Model Management perspective and display the Tracker view (Window > Show View > Other; and in the Tracker View, click the Start Tracking ( ) button to start execution tracking. Perform the required steps and stop the tracking to display the tracking information.

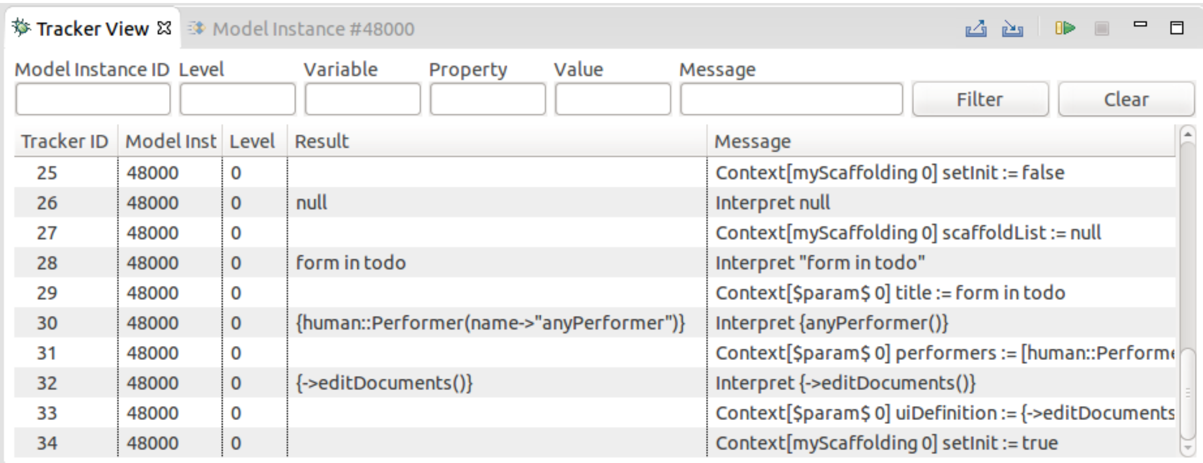


Figure 2.175 Tracker view with output

2.17 Profiling

Profiling serves to identify possible performance problems in the execution of expressions, such as, function calls, assignment, multi-instance calls, etc.

The following profiling features are available:

- Profiler that gathers a list of function and closure calls with execution longer than 1 ms.

The execution duration is counted cumulatively: when a function call takes longer than 1 ms in a node of the model instance tree, the call is included in the list (any function calls from the body of this function are considered part of the main function call).

- Trace Logger prints data about invocations with execution longer than a defined period in milliseconds into the Log view and log file. This feature can be of use in environments, where LSPS Profiler cannot be used. Also, it might be more convenient as input for analysis in other systems.

2.17.1 Profiling with LSPS Profiler

To profile with LSPS Profiler, do the following:

1. Enable profiling by running your server with the property `-DlspsProfile=true`.

   For your Embedded server, you can do so as follows:

   (a) Stop the Embedded server.

   (b) Go to **Runtime Connections** -> **Runtime Connection Settings**

   (c) In the Preferences dialog under **Process Design Suite** -> **Management** -> **Runtime Connections** edit the *LSPS Embedded Server* connection.

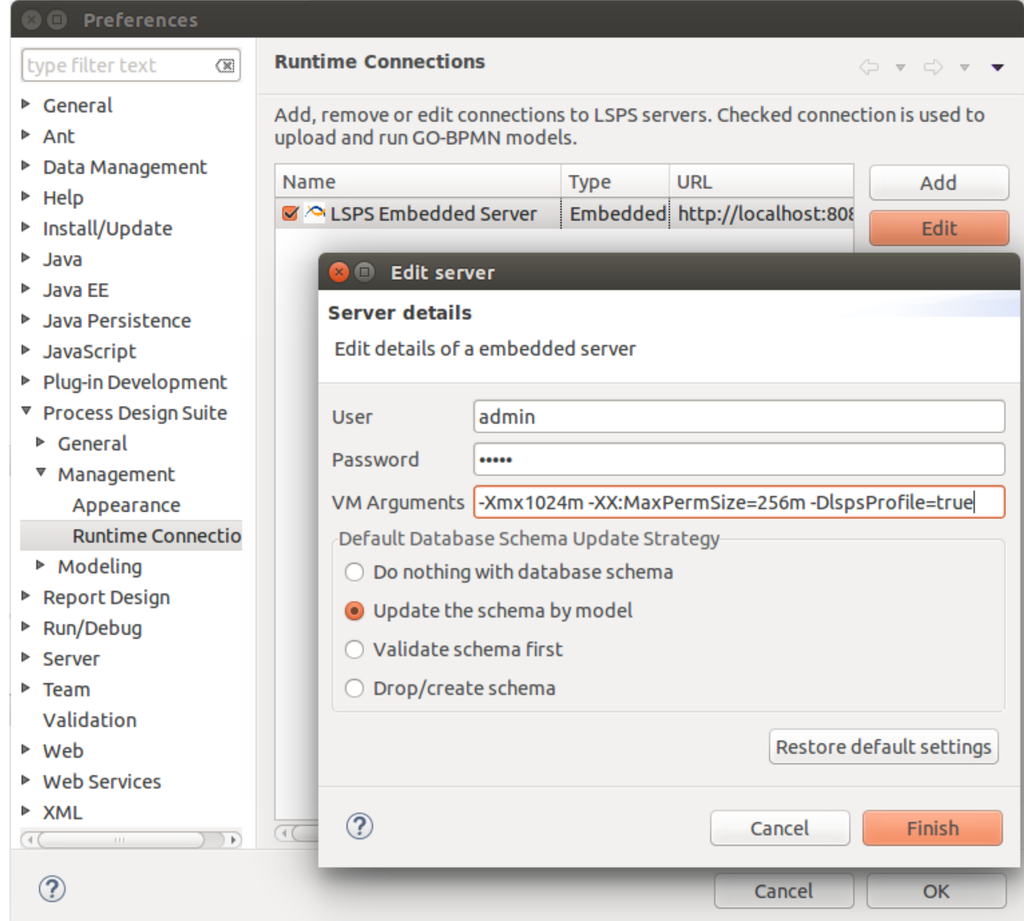


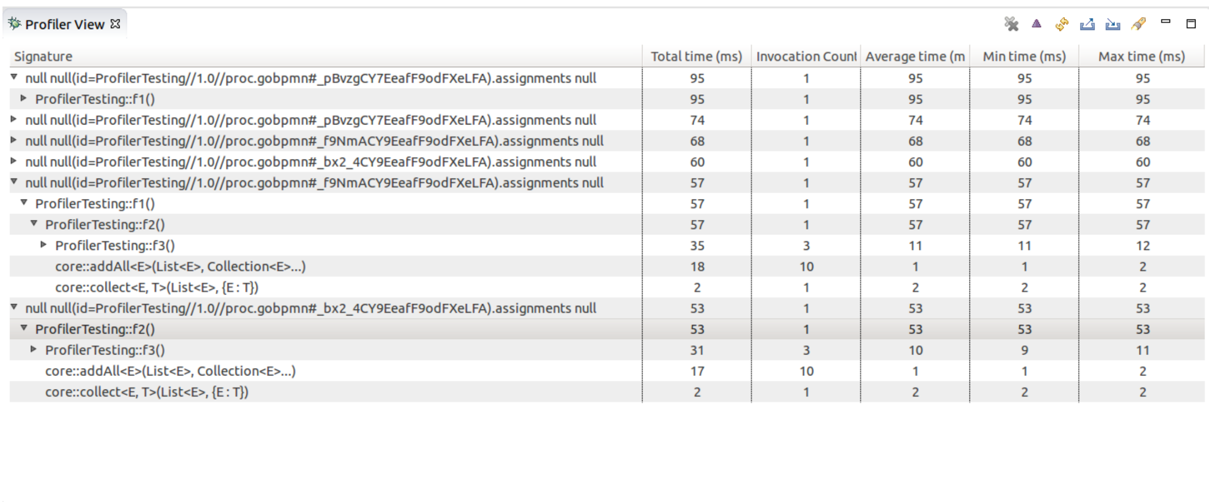
Figure 2.176 Profiler-enabling property on the Embedded Server connection setting

2. In the Management perspective, open the *Profiler view*.

3. Run the models you want to profile.

   Note that if other users are running models on the server, their model instances are be included in the results.

4. Refresh the *Profiler view* to display the profiling results: click the calls to display it in the Module if this is available in the Workspace.



Profiler View

Signature	Total time (ms)	Invocation Count	Average time (m	Min time (ms)	Max time (ms)
null null(id=ProfilerTesting//1.0//proc.gobpmn#_pBvzgCY7EeafF9odFXeLFA).assignments null	95	1	95	95	95
ProfilerTesting::f1()	95	1	95	95	95
null null(id=ProfilerTesting//1.0//proc.gobpmn#_pBvzgCY7EeafF9odFXeLFA).assignments null	74	1	74	74	74
null null(id=ProfilerTesting//1.0//proc.gobpmn#_f9NmACY9EeafF9odFXeLFA).assignments null	68	1	68	68	68
null null(id=ProfilerTesting//1.0//proc.gobpmn#_bx2_4CY9EeafF9odFXeLFA).assignments null	60	1	60	60	60
null null(id=ProfilerTesting//1.0//proc.gobpmn#_f9NmACY9EeafF9odFXeLFA).assignments null	57	1	57	57	57
ProfilerTesting::f1()	57	1	57	57	57
ProfilerTesting::f2()	57	1	57	57	57
ProfilerTesting::f3()	35	3	11	11	12
core::addAll<E>(List<E>, Collection<E>...)	18	10	1	1	2
core::collect<E, T>(List<E>, {E : T})	2	1	2	2	2
null null(id=ProfilerTesting//1.0//proc.gobpmn#_bx2_4CY9EeafF9odFXeLFA).assignments null	53	1	53	53	53
ProfilerTesting::f2()	53	1	53	53	53
ProfilerTesting::f3()	31	3	10	9	11
core::addAll<E>(List<E>, Collection<E>...)	17	10	1	1	2
core::collect<E, T>(List<E>, {E : T})	2	1	2	2	2

Figure 2.177 Profiler view with results

Note that the list with the function statistics contains the most recent function calls. To display the most expensive function calls of the profiling sessions, click the *Expensive Operations* button in the view toolbar.

To import and export profiling results in the XML format, click the export button or the import button in the Profiler view respectively.

2.17.2 Profiling with Trace Logger

To profile with LSPS Trace Logger, do the following:

1. Enable profiling by running your server with the property:

   ```
   -Dcom.whitestein.lsps.lang.InterpreterStackTraceFactory=com.whitestein.lsps.lang.TimerInterpreterStackTraceFactory
   ```

   You can set also the threshold execution time (by default 5ms) with the property `-Dcom.whitestein.lsps.lang.TimerInterpreterStackTrace.threshold`; for example, set it to 100ms with `-Dcom.whitestein.lsps.lang.TimerInterpreterStackTrace.threshold=100`

2. Run the models you want to profile.

The log file for the PDS Embedded Server is by default located in `<WORKSPACE>/.LSPSEmbedded/wildfly-<VERSION>/standalone/log/`

Note that if other users are running model instances on the target server, their model instances will be included in the results.

2.18 Sharing Resources

- Exporting

  You can export Modules with their resources:

  - as a bundle, which is ready to be deployed to the LSPS Server,
  - as a library, which can be used as a source of elements referenced by other user from their Modules,
  - as a package that other users can import and work with, or as an XPDL so you can import your Processes to other BPMN designing tools.

- Import

  You can import modules packages exported by another user with PDS, or as XPDL files.

- Libraries

  Modules exported as libraries are intended to shared by users as as resources of reusable elements.

2.18.1 Resource Export

You can export the following:

- modules with their content as a package with your model. Such a package can be used to do the following:
  - deployment to the server (refer to GO-BPMN Export)
  - reuse other project and modules as libraries (refer to Creating a Library)
  - sharing with users of PDS (refer to Exporting with General Export)
  - sharing with users of other BPMN products (refer to Exporting to XPDL)
- projects and modules (refer to Exporting with General Export)

To speed up export if you need to export the same resources repeatedly, define anExport Configuration file.

2.18.1.1 Export Configurations

Frequent export configurations can be store in an export configuration file: it holds the list of workspace resources. It serves as input for export so you do not have to define the resources you want to export all over again.

Since they are not considered part of the Model, Export configurations can be located anywhere within a GO-BPMN project, that is, in a project, Module, or a generic folder.

2.18.1.1.1 Defining an Export Configuration

To define an export configuration, do the following:

1. In the GO-BPMN Explorer, double-click the respective export configuration or create a new export configuration.
2. In the Export Configuration Editor, click Add.
3. In the Add Export Configuration Item dialog box, select the GO-BPMN projects, GO-BPMN Modules, and folders you wand to include in the configuration file.
4. Click OK.
5. Click File -> Save to save the file.

2.18.1.2 Exporting with General Export

You can export a GO-BPMN module or project to archive files or file system structure, which the users can then import. Such bundles are non-deployable and serve only to exchange resource among users when a version control system cannot be used.

To export a GO-BPMN module or project to an archive file or file system:

1. In the GO-BPMN Explorer view, right-click the GO-BPMN project or module.
2. In the context menu, go to **Export** and select:
   - *Archive File* to export as an archive file
   - *File System* to export as a folder structure
3. On the File System or Archive file page in the Export dialog box, select the checkboxes of the projects and modules.

   To specify the module or project content, click the module or project in the left pane; select the resources on the right.

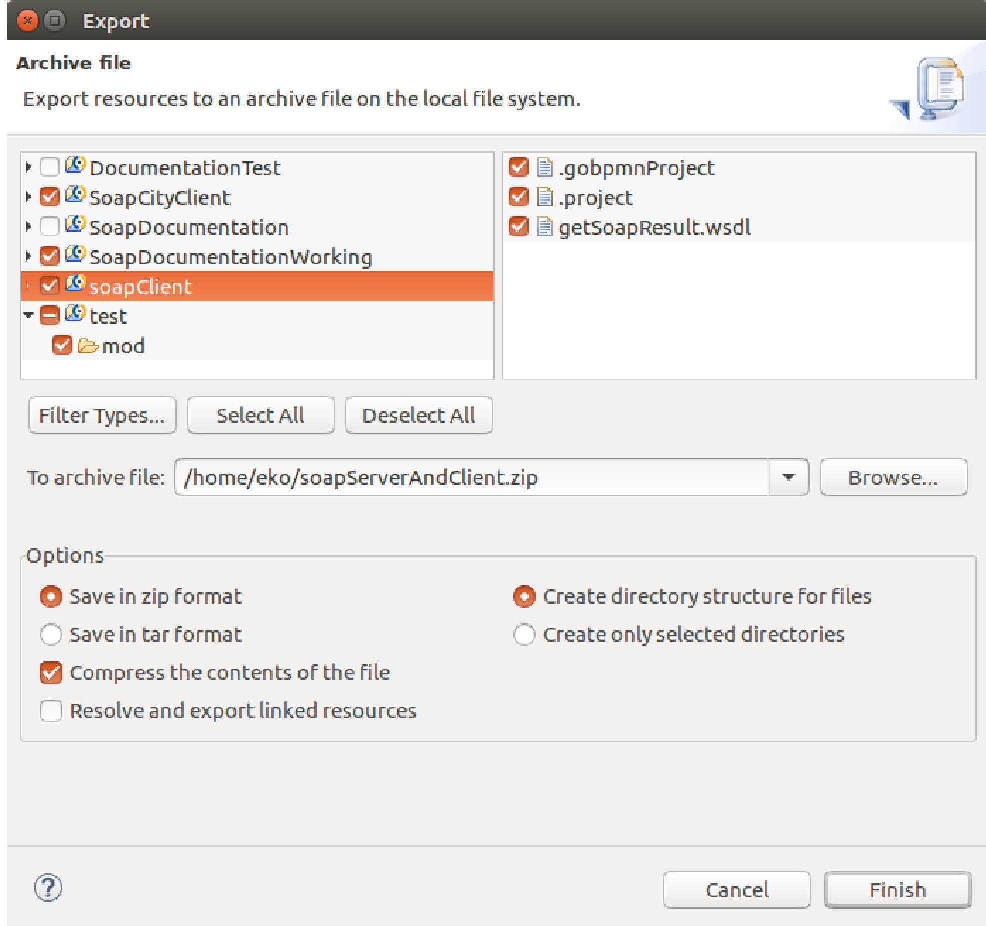


Figure 2.178 Selecting resources and files to be exported into an archive

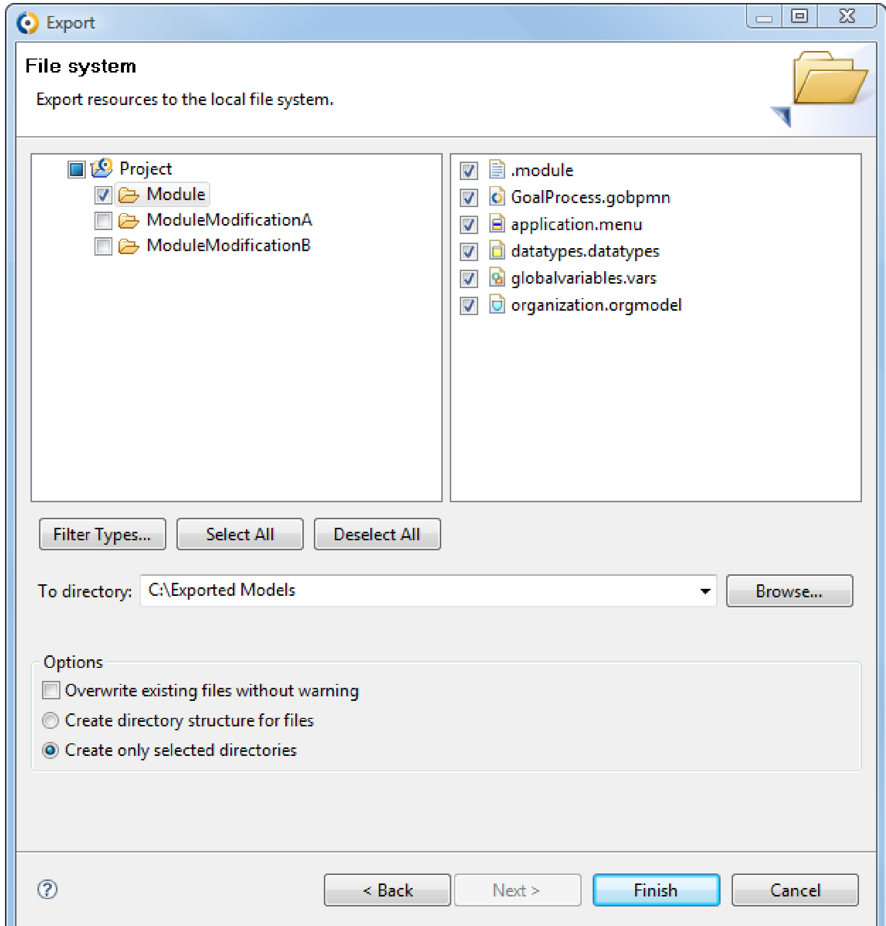


Figure 2.179 Selecting resources and files to be exported into a file system

4. In the Destination file text box, specify the path and the `archive name` of the file, for example, `archive.zip`.

2.18.1.3 Exporting with GO-BPMN Export

The `GO-BPMN Export` feature allows you to create zip files with a model, which can be deployed to the server.

To create a deployable zip file, do the following:

1. Go to File > Export.

   Alternatively right-click the project or module in GO-BPMN Explorer and select Export.

2. In the Export dialog box, expand Process Design Suite and select GO-BPMN Export.

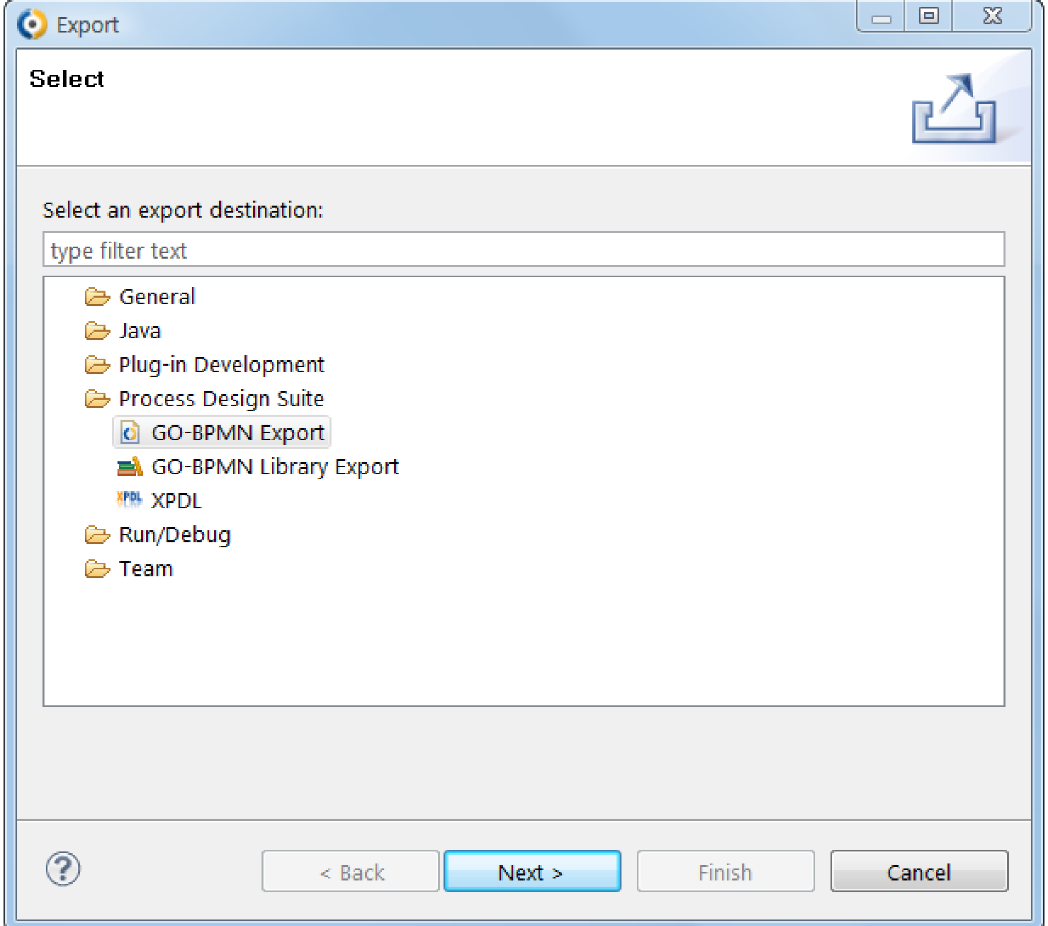


Figure 2.180 Selecting the GO-BPMN Export

3. Click Next.

4. In the GO-BPMN Export dialog box, do the applicable:

   - Under GO-BPMN Modules, select the modules to be exported.
   - Under Export Configuration, select the desired `export configurations`.

   Modules selected in the GO-BPMN Modules area are exported along with the resources specified in the selected export configuration (resources are summed up).

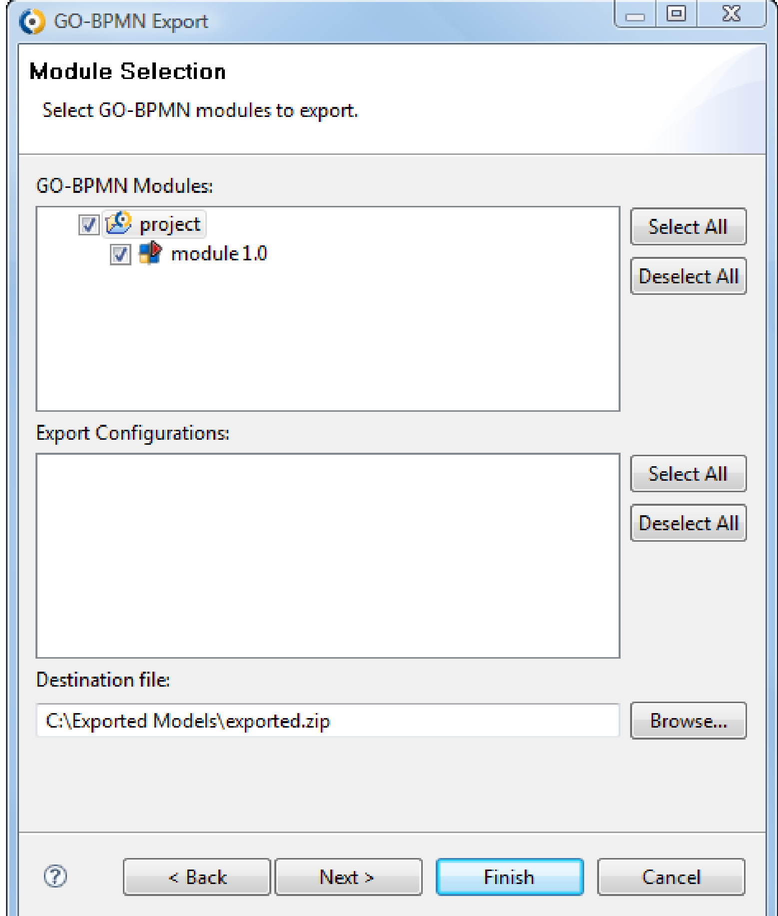


Figure 2.181 Selecting modules to be exported

5. In the Destination file text box, specify the path and the `name` file.

   Export name has to include the `.zip` extension.

6. Click Next and check the list of modules.

7. Click Finish.

Note: The GO-BPMN export feature drops the project structuring – only modules are preserved.

2.18.1.4 Exporting to XPDL

You can export your models as XPDL files with the resources supported by the format (goal-extension is not supported by XPDL).

To export your model to an `XPDL` file, do the following:

1. Go to File > Export.

   To export, you can right-click the project or module in GO-BPMN Explorer and select Export.

2. In the Export dialog box, expand Process Design Suite and select XPDL.

3. Click Next.

4. In the XPDL Export dialog box:

   - under Module to export, define the module to be exported;
   - under XPDL file, define the target file name and path;
   - in the Elements to export area, select module elements to be exported;

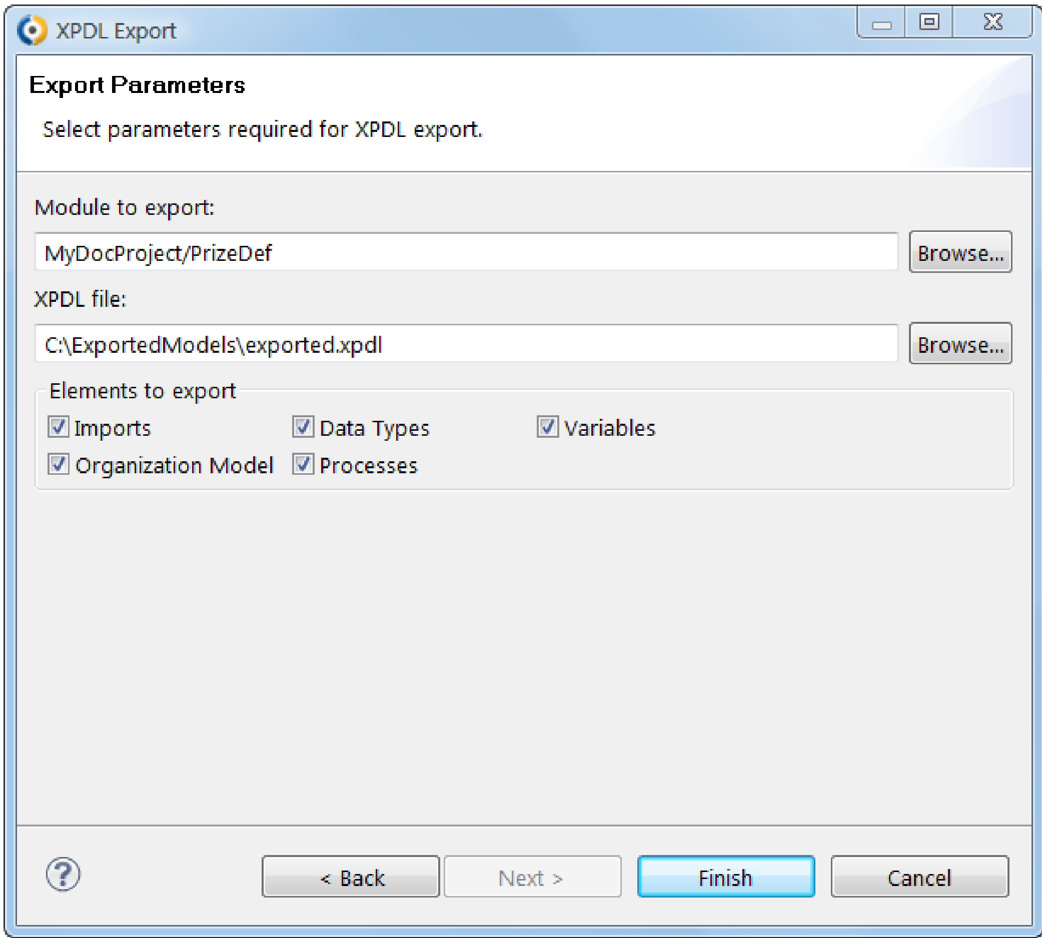


Figure 2.182 Defining XPDL Export settings

5. Click Finish.

2.18.2 Resource Import

You can import the following resources into the workspace:

- module packages
- XPDL files
- file systems with the project directory structure

Note: Resource import should not be confused with module import: importing a module that is already in your workspace tree to another module is another mechanism that allow you to reuse existing modules in your workspace (refer to Importing Modules).

2.18.2.1 Importing Model Packages to Workspace

If you want to work with external resources, typically with Models that you received from somebody else, you need to import them into your workspace.

To import a model package to your workspace, do the following:

1. Right-click anywhere inside the GO-BPMN Explorer view and select Import.

   Note: If importing a model package created using the GO-BPMN export, make sure a project, which can accommodate imported modules is created (package contains only modules).

2. In the Import menu box, select Archive File.
3. Click Next.
4. On the Archive file page in the Import dialog box:
   (a) In the From archive file text box, type the `filepath` of the model package.
   (b) Click Browse next to the Into folder text box, and in the Import into Folder dialog box, select the project to import the modules. Click OK.

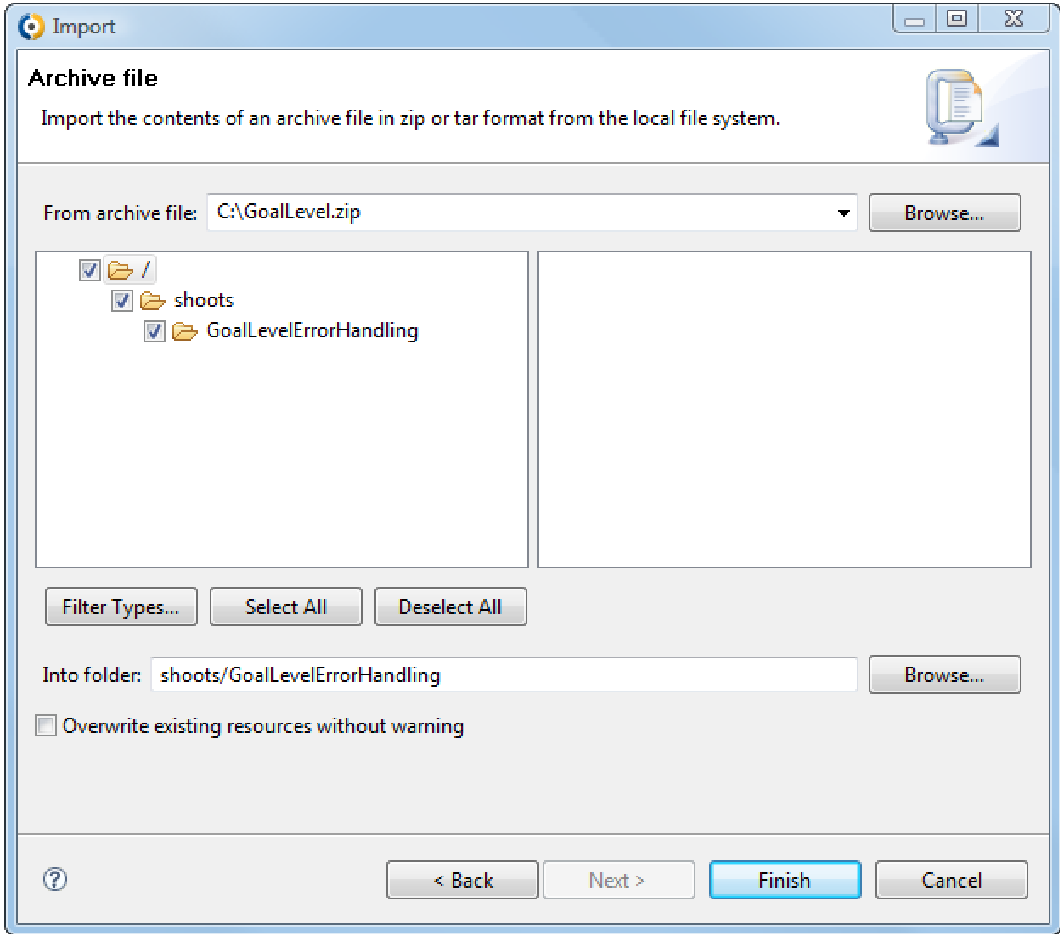


Figure 2.183 Importing an archive

5. Click Finish.

The model package content is imported to the workspace.

2.18.2.2 Importing XPDL

Import is supported for XPDL 2.1 files.

To import an XPDL file to a module, do the following:

1. Right-click anywhere inside the GO-BPMN Explorer view and click **Import**.
2. In the Import dialog box, expand GO-BPMN Modeler and select XPDL.
3. Click Next.
4. On the next page of the dialog box:
   - In the XPDL file text field provide the location of the xpdl file.
   - In the Target module provide the target module path (`project/module`).
   - In the Elements to import area, check the elements to import.
5. Click Finish.

2.18.2.3 Importing File System Structure

To import a file system structure to the workspace, do the following:

1. Right-click inside the GO-BPMN Explorer view and select **Import**.
2. In the Import menu, select File System.

   To import all GO-BPMN projects contained in a location (including subfolders) select Existing Projects into Workspace.
3. In the From directory field box, type the path to the file system. Use Browse, if necessary.
4. In the compartment below the From directory field box, select the projects and resources you wish to import.
5. In the Into folder field box, type the path to the project and module to import the file system.
6. Check and select options in the Options area.
7. Click Finish.

The structure appears in the GO-BPMN Explorer tree.

2.18.3 Libraries

LSPS comes with multiple libraries to support your modeling efforts:

The `Standard Library` modules define crucial resources required to design and execute GO-BPMN models, such as, the data types of BPMN elements, functions of models, basic task types, etc. For details, refer to the `Standard Library Reference Guide`; note that the documentation is available directly in the Standard Library modules as well.

Additionally, the following libraries are provided out-of-the-box as well:

- Exchange Client Library and SharePoint Client Library: web-service task types to communicate with Share↩ Point and Exchange Server
- Social Library: resources for forums (import the module to one of your modules, upload and check the *Social Settings* document)
- `Scaffolding Library`: resources for CRUD ui components

Related links:

- Resource Sharing
- Importing Modules

2.18.3.1 SharePoint and Exchange Client Libraries

Support form communication with Microsoft SharePoint 2010 and later and Microsoft Exchange 2010 and later is provided by the SharepointClient and ExchangeClient libraries. The libraries contain web service tasks and supporting resources to communicate with SharePoint and Exchange servers.

> **Note:** The libraries provide only tasks for a subset of the SharePoint and Exchange web services. If necessary, create your own web service task (refer to Web Service Client).

2.18.3.2 Importing a Library to GO-BPMN Projects

To import a library to a GO-BPMN project:

1. In GO-BPMN Explorer, right-click the respective GO-BPMN project.
2. Click **Add Library**.
3. In the Add GO-BPMN Library dialog box select:
   - Select absolute file path: absolute path to the library
     - Select library variable and the relative path: insert library variable and relative path (the variable defines an absolute path and the relative path defines the path relative to the variable).
     - Select a workspace file: insert the workspace path to a library zip file available in the workspace.
   - Select a built-in library: select one of the built-in libraries.

   To remove a library from a GO-BPMN project, in GO-BPMN Explorer, right-click the library and select Remove.

To work with library content in your module, import it into the module: right-click the module, go to **Module Imports** and click **Add** to select a library module.

2.18.3.3 Exporting Modules as Libraries

To export modules as a library, use the dedicated export feature: it allows you to explicitly select, which modules are included in the output zip:

1. In the GO-BPMN Explorer view, right-click the GO-BPMN project or module.
2. In the context menu, open Export.
3. In the Export menu, select GO-BPMN Library Export.
4. On the Module Selection page of the GO-BPMN Export dialog box, select the checkboxes of the modules you want to include in the library.

   You can use an export configuration by selecting it in the lower part of the dialog box.

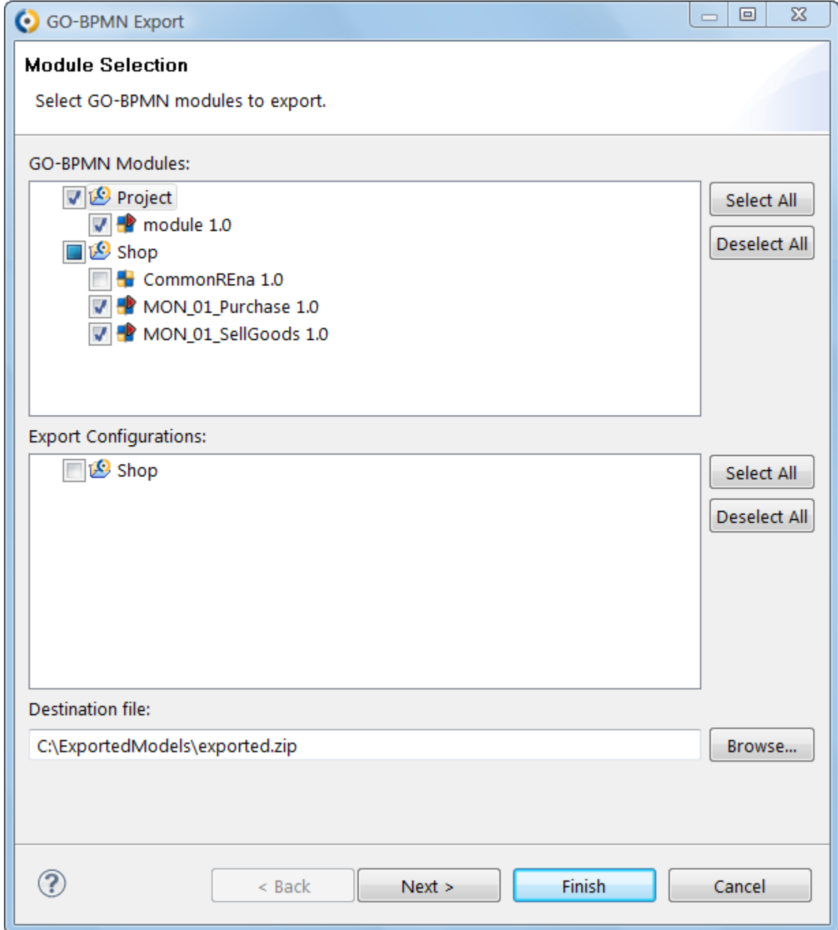


Figure 2.184 Selecting modules to be exported as a library

5. In the Destination file text box, specify the path (and the `archive name` of the file, for example, `archive.zip`).
6. Click Next to display overview of the added modules modules.
7. Click Finish.

2.18.3.4 Removing a Library from a Project

To remove a library from your project, in GO-BPMN Project Explorer, right-click the library reference in your project and click Remove Library. Note that this removes only the import of the Library, not the Library modules themselves.

2.19 Model Presentation

To present your Models to audience, use the Business Modeling perspective.

Chapter 3

Module Upload and Execution

Once you have designed a module, you can upload it to an LSPS server and run it. Note that only a module that is `executable` can be run or instantiated.

To upload modules from PDS, first connect PDS to a server with LSPS Application Once you have established the connection, you can upload a model to the server and execute it.

> **Note:** You can use also `command-line console` as well as from the `Management Console` to deploy and instantiate your model. However, you need to export your Modeling as a deployable bundle first.

When you upload a module with a `data model` that contains shared Records, tables for the related data is created in the database. If you modify the shared Records and pertinent relationships and upload the module with the same name and version again, the already existing tables need to be adjusted.

Therefore, whenever you are uploading or running a module, you need to define, how to handle such a situation:

- **Do nothing with database schema**: data-model tables remain unchanged.
- **Update the schema by model**: New tables, columns and relationships are added; no data is deleted.
- **Validate schema first**: data-model tables are validated against the new model. In case of inconsistencies, the model upload or launching fails.
- **Drop/create schema**: data-model tables are dropped and created anew.

> **Important:** The Database Schema Update Strategy setting does not impact LSPS runtime information, such as, data on uploaded models, model instances in the current execution status, persons details, etc. and the data remains unchanged regardless of the setting. Only tables based on module data models are subject to the strategy.

3.1 Server Connection from PDS

PDS connections to LSPS servers are set under **Runtime Connections** > **Runtime Connection Settings**. The connections to embedded servers are added automatically when the servers are generated (For the PDS Embedded Server, when first started and for the SDK Embedded Server, when the LSPS Application is generated). Other connections must be configured manually.

3.1.1 Connecting PDS to an LSPS Server

To connect PDS to a server with LSPS Application other than the PDS or SDK Embedded Server, do the following:

1. Set up the connection configuration:
   (a) On the menu bar, under **Runtime Connections**, click **Runtime Connection Settings**.
   (b) In the Runtime Connections part of the Preferences dialog box, click **Add** and define the connection settings:
      - Server name: arbitrary name of the connection
      - Server URL: URL of the server instance to connect to (for example, for the PDS Embedded server `http://localhost:8080/lsps-application`)
      - User and Password
      - Application URL: URL to the LSPS Application on the server
      - Default Database Schema Update Strategy: the strategy which is set by default on configurations of module upload and run. The configurations are used when uploading and running modules from the Modeling perspective and are created when the module is run or uploaded for the first time from the Modeling perspective.
   (c) Click **Test Connection** to make sure the settings are correct.
   (d) Optionally, select the connection: the selected connection is used as default: if you trigger a module upload or run a model from the Modeling perspective and PDS is not connected to an LSPS server, the selected connection is established.

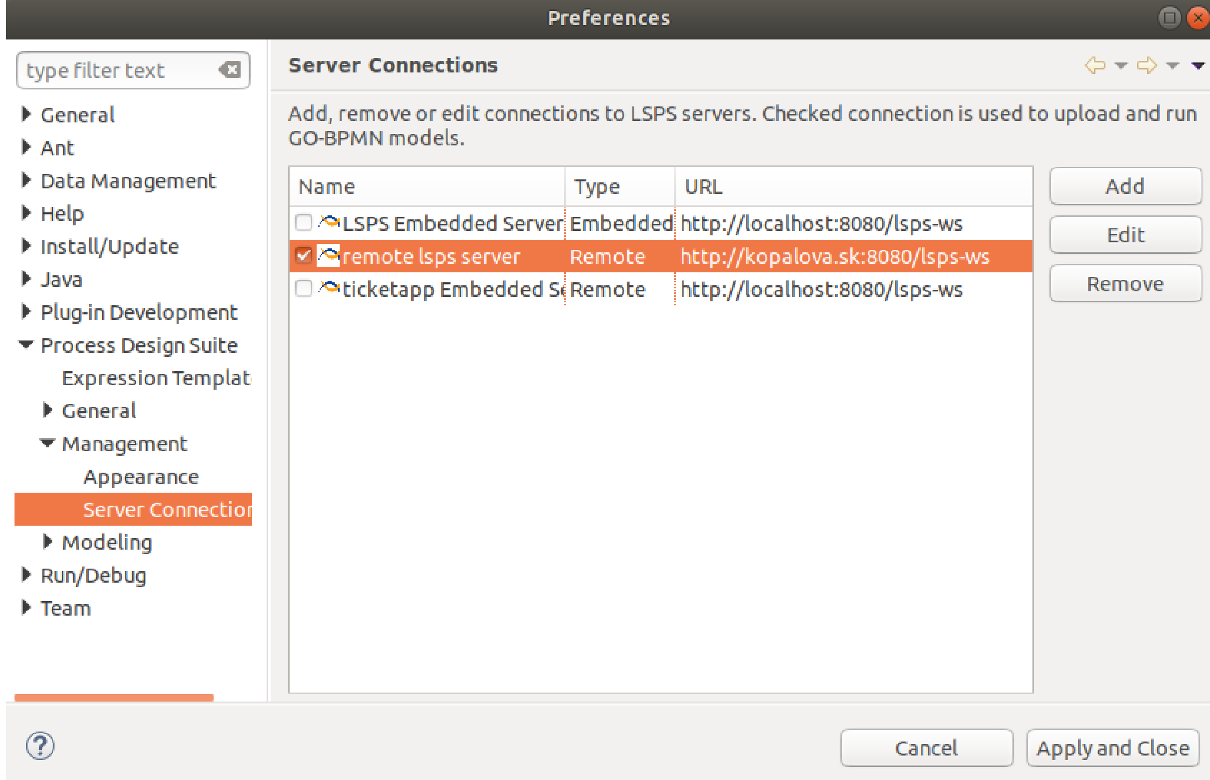


Figure 3.1 The remote connection set as preferred

2. If required, define a proxy server for your connection.
3. Go to **Runtime Connections** > **Select** and click the connection. If you selected the connection as default, simply upload or run a model.

3.1.2 Connecting to an LSPS Server using a Proxy Server

To use a proxy server to connect from PDS to an LSPS Server, do the following:

1. Go to **Window** > **Preferences**.
2. In the *Preferences* dialog, go to **General** > **Network Connections**
3. Set the Active Provider option:
   - **Direct**: no proxy server
   - **Manual**: as defined in Proxy entries below
     You can define in the *Proxy bypass* table, the hosts that should be excluded from the proxying.
   - **Native**: as defined by the operating system
4. Click apply.

3.1.3 Connecting to the PDS Embedded Server

The PDS Embedded Server is an application server with LSPS Server accessible on `http://localhost:8080`.

To launch the PDS Embedded Server and connect the Process Design Suite to the server, do the following:

1. Go to **Runtime Connections** -> **LSPS Embedded Server**.
2. Optionally, select the connection: the selected connection is used as default. If you trigger a module upload or run a model from the Modeling perspective and PDS is not connected to an LSPS server, the selected connection is established.

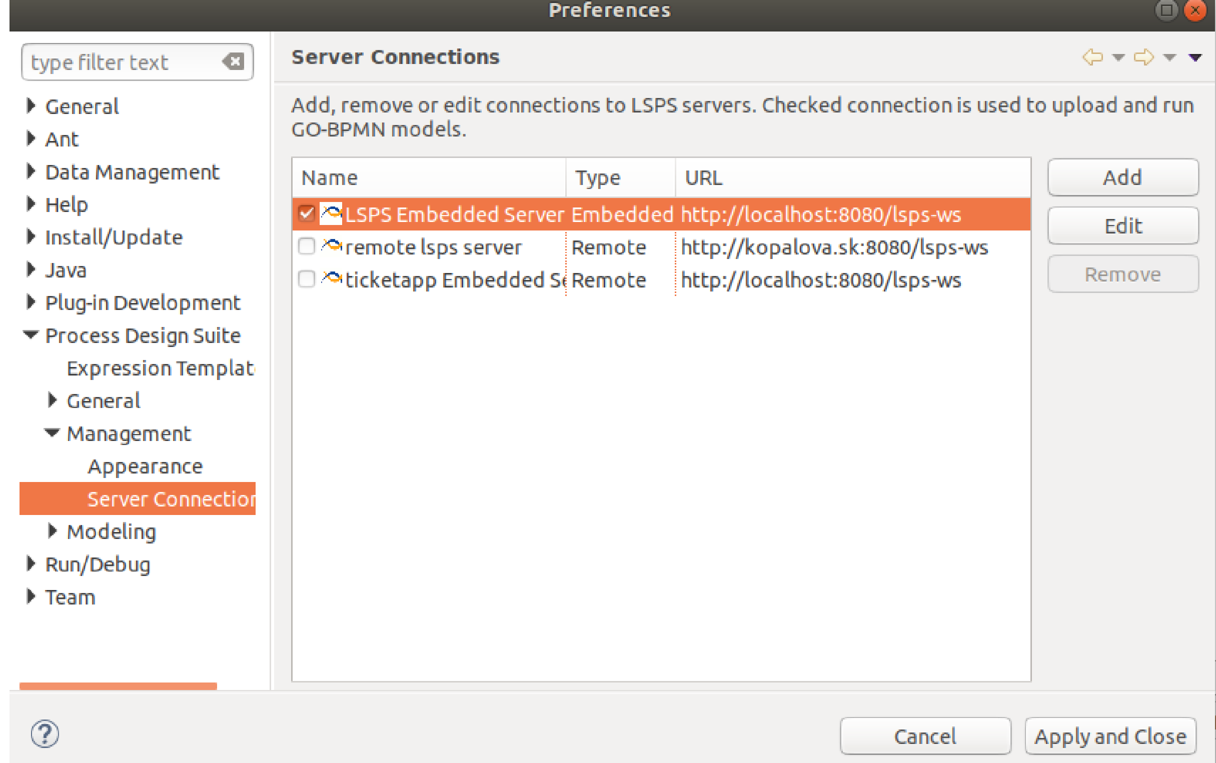


Figure 3.2 Runtime Connection page of the Preferences dialog box with connection to PDS Embedded Server as default

3. Click Start.

Alternatively, click the Start Embedded Server ( ) button on the main toolbar.

On start, if the server does not exist, it is created in the `.LSPSEmbedded` directory of your workspace. To reinstall it, delete the directory and start the server again.

3.1.4 Loading and Resetting Database of the PDS Embedded Server

The content of the LSPS database includes the runtime information, such as uploaded models, model instances in the current execution status, persons details, etc. Before you reset the database consider exporting it so you can load it if necessary. When resetting the database, all data is lost.

To save, restore, or reset the LSPS Embedded Server database, do the following:

1. Stop the Embedded Server.
2. Under **Runtime Connections**, select LSPS Embedded Server and Database Management.
3. In the Embedded Database Management dialog box:
   - Select a database name and click Load to restore a database.

     Important: If a database is restored, the current database goes lost.
   - Click Save and provide the database name in the displayed dialog box to save the current database.
   - Click Reset to erase the database.

Note: The database is stored in the current workspace in the .LSPSEmbedded folder.

3.2 Model Upload

When you upload a module, the module and all its imported modules are uploaded to the LSPS Server of the server to which PDS is connected. An uploaded module is identified by its name and version and if a module with the given name and version is already present on the LSPS Server, its is replaced.

You can upload your modules to your LSPS Server from the Modeling perspective or from the Management Perspective. The upload is performed based on the upload configuration: when you are uploading a module for the first time, PDS automatically creates an upload configuration, which is then used for upload.

3.2.1 Uploading a Module from the Modeling Perspective

> **Note:** Ensure that the server with LSPS Server is running and your PDS is connected to it (Information on the current connection is in the line at the bottom of PDS).

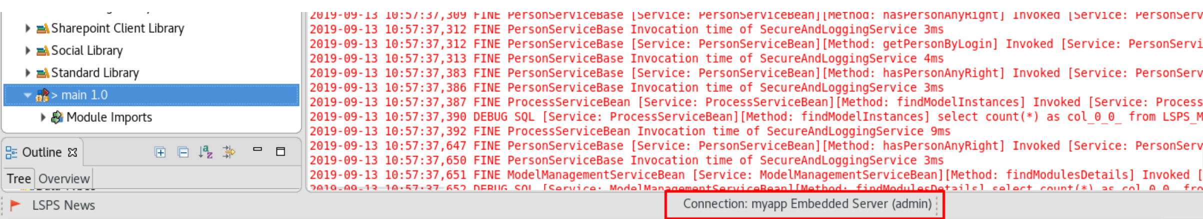


Figure 3.3 Address line with connection established

To upload a module for the first time, do the following:

1. In *GO-BPMN Explorer*, right-click the module.
2. In the context menu, click **Upload As** > **Model**.

This creates an upload and run configuration, which is used for the next upload and run of the module from the Modeling perspective: To adjust the configuration, do the following:

1. Go to **Run** > **Run Configuration** or **Upload Configuration**.
2. Select the configuration under *Model*.
3. Adjust the properties of the configuration in the tabs on the right.

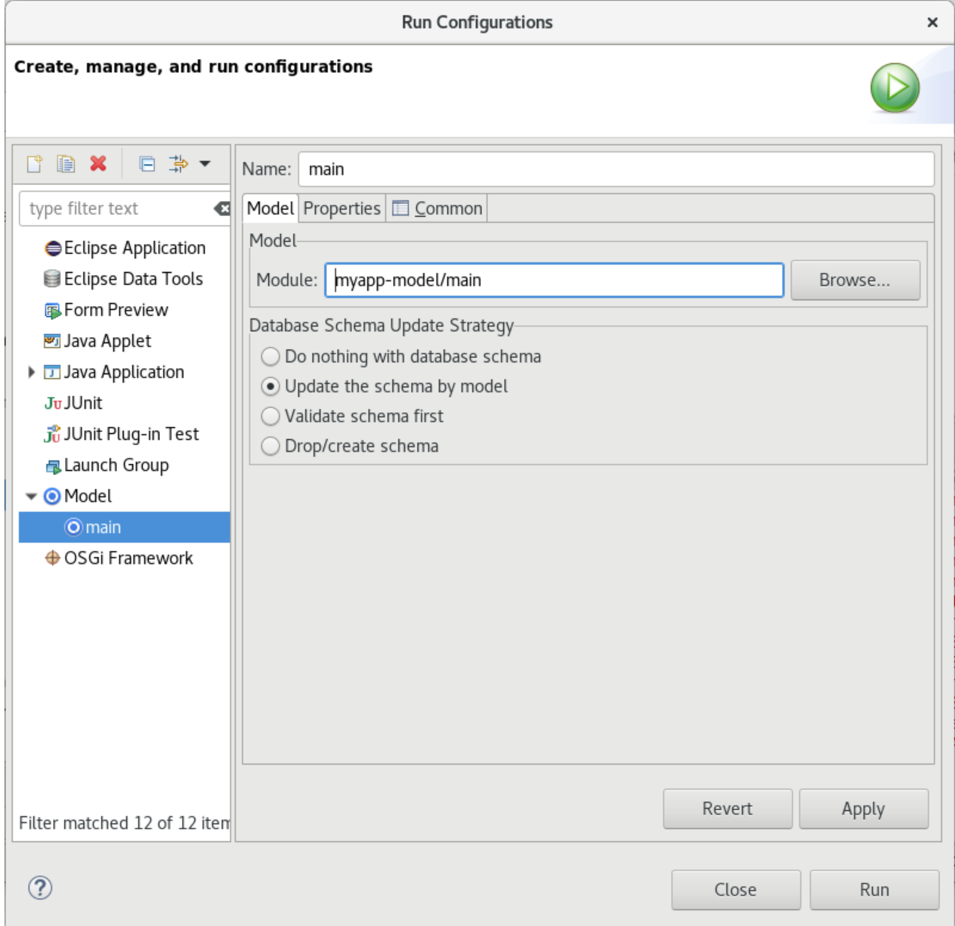


Figure 3.4 Run and upload configuration for the main module

3.2.2 Uploading a Module from the Management Perspective

Note: Ensure that the server with LSPS Server is running and your PDS is connected to it (Information on the current connection is in the line at the bottom of PDS).

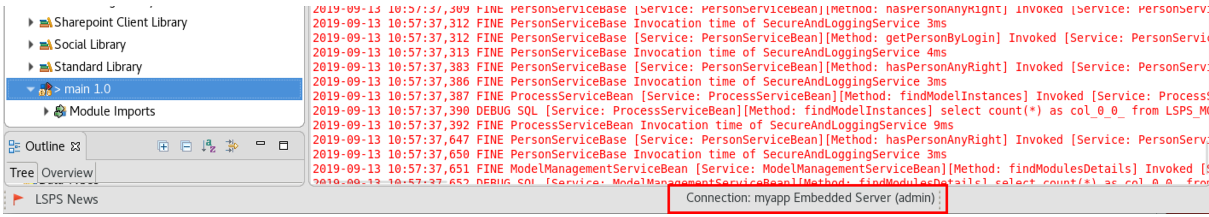


Figure 3.5 Address line with connection established

To upload a module to the LSPS Server to which your PDS is connected, do the following:

1. Display the Module Management view (in the toolbar, click the Management Views button ( ) and click **Module Management**).

2. In Module Management view, click the Upload ( ) button.
3. In the Model Upload dialog box define the model details. Every model has to be valid (the system validates it on upload); otherwise uploading fails.
4. Click OK. The selected Model is uploaded to the current LSPS Server.
5. If you selected the Do not execute schema update scripts flag, the scripts are available in the Schema Update Scripts tab of the Module Management view. Double-click the respective row to have the script displayed.

You can check the list of uploaded Modules in the Module Management view.

3.3 Model Instantiation

Once you uploaded an executable module with its module imports to an LSPS Server, you can run it as a model instance. Typically, you will instantiate Models from the Management perspective, but PDS allows you to upload and instantiate a Model directly from your workspace.

On Model instantiation, the Execution Engine of the LSPS Server creates a runtime version of a model: a model instance. Since you can create multiple model instances of a model, each model instance is created with a unique ID.

In additional, a model instance may define specific properties in the form of key-value pairs. For more information on Model instance lifecycle, refer to the GO-BPMN Modeling Language User Guide.

3.3.1 Instantiating a Model from the Management Perspective

To create a model instance of an uploaded Model from the Management perspective, do the following:

1. In the main toolbar, click the arrow in the Management Views ( ) button and select Model Instances.
2. In the toolbar of the displayed Model Instances view, click Create ( ).
3. In the Model drop-down box of the Model Instance Creation dialog box, choose a model, and click OK.
4. If necessary, in the Properties table, edit the key-value pairs attached to the model instance.

   Note: If a Model instance remains in the status Created, check the Error Log for errors. Error Log may contain also error entries of the server if these were caused by a Process Design Suite feature, which appear primarily in the Console view.

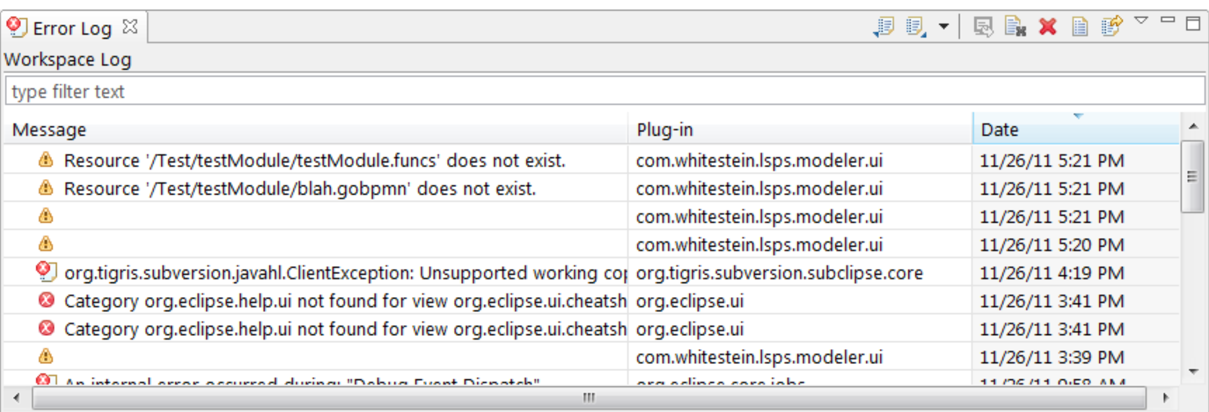


Figure 3.6 Error Log with error entries

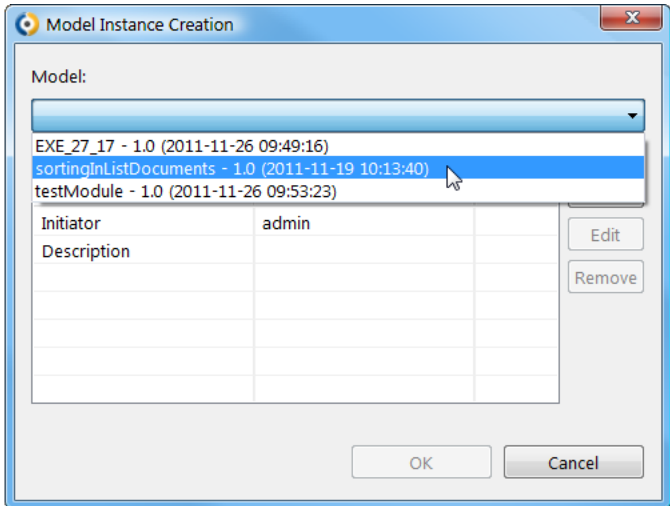


Figure 3.7 Creating a model instance based on an uploaded executable module

3.3.2 Instantiating a Model from the Modeling Perspective

Note: Ensure that the server with LSPS Server is running and your PDS is connected to it (Information on the current connection is in the line at the bottom of PDS).

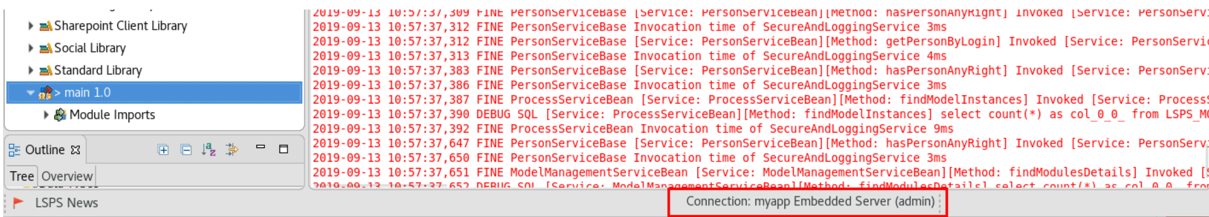


Figure 3.8 Address line with connection established

To instantiate a model of a module for the first time, do the following:

1. In *GO-BPMN Explorer*, right-click the module.
2. In the context menu, click **Run As** > **Model**.

This creates an upload and run configuration, and instantiates the module. Next time you run the module from the Modeling perspective, the created configuration is used.

To adjust the configuration, do the following:

1. Go to **Run** > **Run Configuration** or **Upload Configuration**.
2. Select the configuration under *Model*.
3. Adjust the properties of the configuration in the tabs on the right.

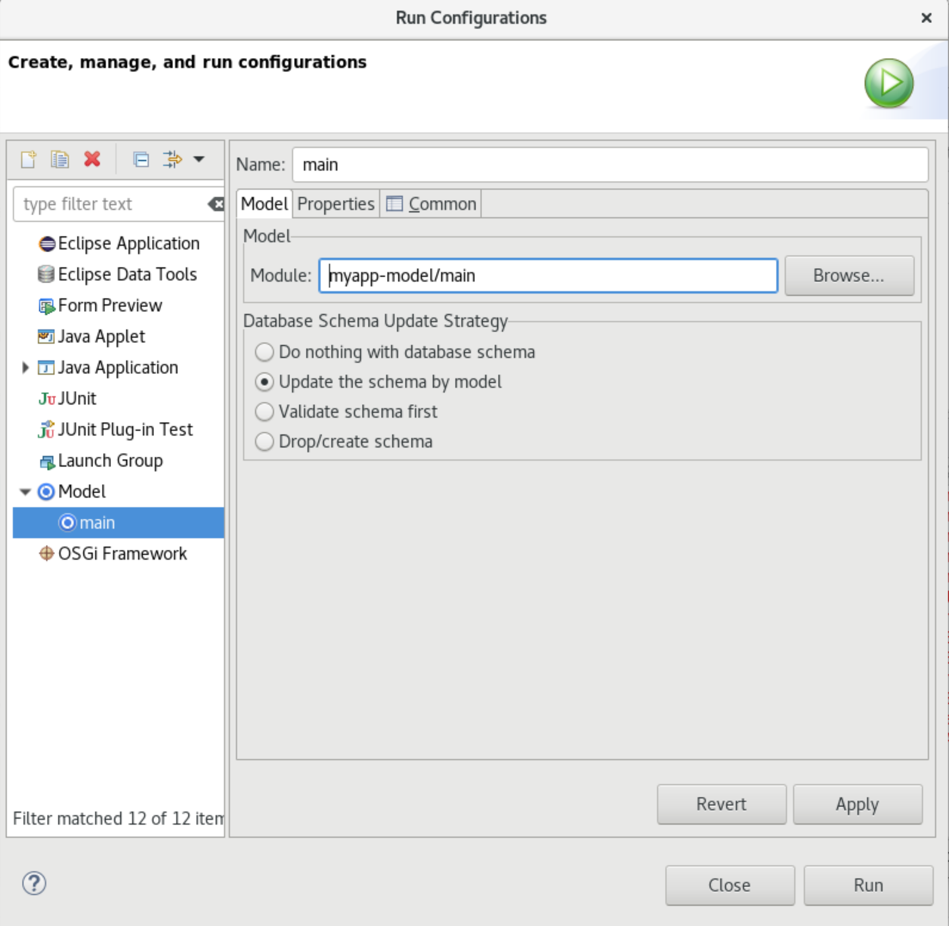


Figure 3.9 Run and upload configuration for the main module

Chapter 4

Updating Model Instances

The LSPS *model update* mechanism allows you to update model instances on runtime so they use a new version of the underlying model for their execution:

A model instance is created based on a model. The instance itself holds only runtime data, such as, which element is currently executed, values of variables, model instance ID, etc. The static data, such as the BPMN flows, remains in the model. Using the model update feature, you adapt the runtime data and "switch" the model for another model. The execution of the model instance then continues according to the new model.

Before you can perform the "switch", you need to define how to adapt the runtime data so the model instance *can* use the new model; for example, if you change the data type of a variable from String to Integer, you need to define how to convert the String value to the Integer value.

These rules are defined in a model update definition with the Model Update Editor which is based on the Comparison Editor. It is strongly recommended you get familiar with model comparing before using the Model Update Editor.

> **Important:** The model update feature is intended for models that were created in the same version of LSPS, since the muc definition file is specific for the given version and not portable. It is not possible to use model update definition created in one version of LSPS to update a model instance if its model was created in another LSPS version.

Model update is performed in the several phases:

1. **Pre-processing**

   The elements that were being executed when the model instance was Suspended are transformed as defined in their pre-process.

2. **Transformation**

   The runtime data of the model instance is transformed to follow the new model. Its data types and variables are updated to their target types and values, and instantiated model elements are transformed according to their transformation strategies.

3. **Post-processing**

   The post-processes of instantiated elements prepare the respective elements for running.

Note that on update, closures of the model instance are checked for compatibility. If an incompatible closure is detected, the system displays a dialog with details on the closure with incompatibility.

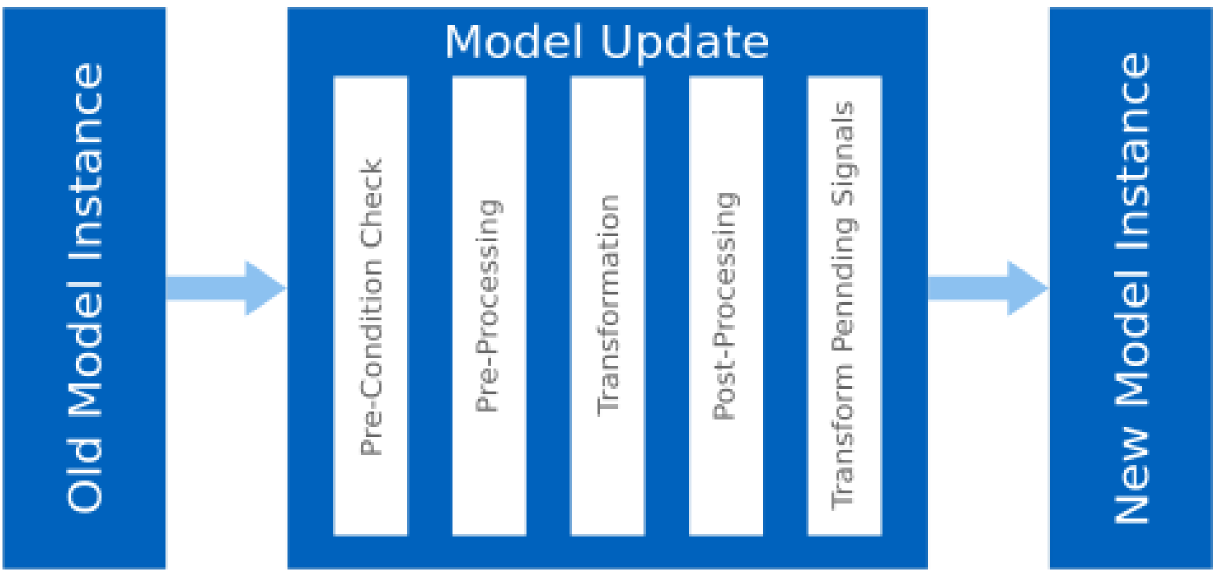


Figure 4.1 Schema of Model update process

A set of steps required to perform a model update is described in the chapter Performing Model Update.

4.1 Model-Update Processes

Model-update processes are BPMN Processes that are executed before and after the transformation of model update: They serve as hooks that prepare instantiated elements for transformation and for running after the transformation.

You can define model-update processes for **Modules, Processes, Plans, Sub-Processes, and Tasks with changes** in the new model. One such element can contain only one model-update process for each phase.

Only model-update processes of instantiated elements are executed, that is, the elements that hold the token or their parent elements:

- for pre-processes, the elements that were running when the model update was triggered;
- for post-processes, the elements that will be running when the instance continues running.

As mention above, the processes are define as BPMN processes with the following restrictions:

- They must start with a None Start Event.
- They can import modules, which are common for the model update phase. However, note that the imported modules do not have access to the old or new models and must import the models explicitly if necessary.

Model update processes exist in their own contexts, separated from the model instances that are updated. However, they have access to the following:

- Contexts relevant for their phase:
  - Pre-processes have access to the old model instance.
  - Post-processes have access to the new model instance.
- Contexts of their model element.

Defined for Element	Accessible Context and Namespace
Module	Module namespace
Process	Process and parents
Plan	Plan and parents
Embedded Sub-Process	Sub-Process namespace and parents; for multi-instance sub-processes also their iterator
Reusable Sub-Process	Parameters of the sub-process and parents; for multi-instance also sub-processes iterator
Task	Task parameters and parents; for multi-instance Tasks their iterator

If a name clash occurs, for example, if a pre-process context variable has the same name as a variable in the attached old model, the name in the namespace with higher priority takes precedence. The namespace priority from the highest to the lowest is as follows:

1. Local namespace of the model update process
2. Contexts of the element the model update process is attached to:
   - If on activity, the parameters of the activity
   - If on a module, process, plan, or sub-process, the namespace of the element
   - If on a multi-instance activity, the iterator of the activity
   - Imports of other modules and global model update variables

Model update processes do not have access to other model update processes. However, you can define context variables which are shared by update processes in the given update phase or use signals between model-updates of the same phase.

Pre-Processes

A pre-process is a model update process attached to a modeling element of the old model. It is executed in the pre-processing update phase if defined for an instantiated element and serves to prepare the old model instance for transformation.

Post-Processes

A post-process is a model update process attached to a modeling element of the new model. It is executed in the post-processing update phase if defined for an instantiated element that still exists in the new model and serves to prepare the new model element for running.

4.2 Transformation

Transformation is the main phase of model update: the record instances, variables, and asynchronous model elements that were instantiated before the model instance was suspended for model update and are changed in the new model are transformed to their version for the new model.

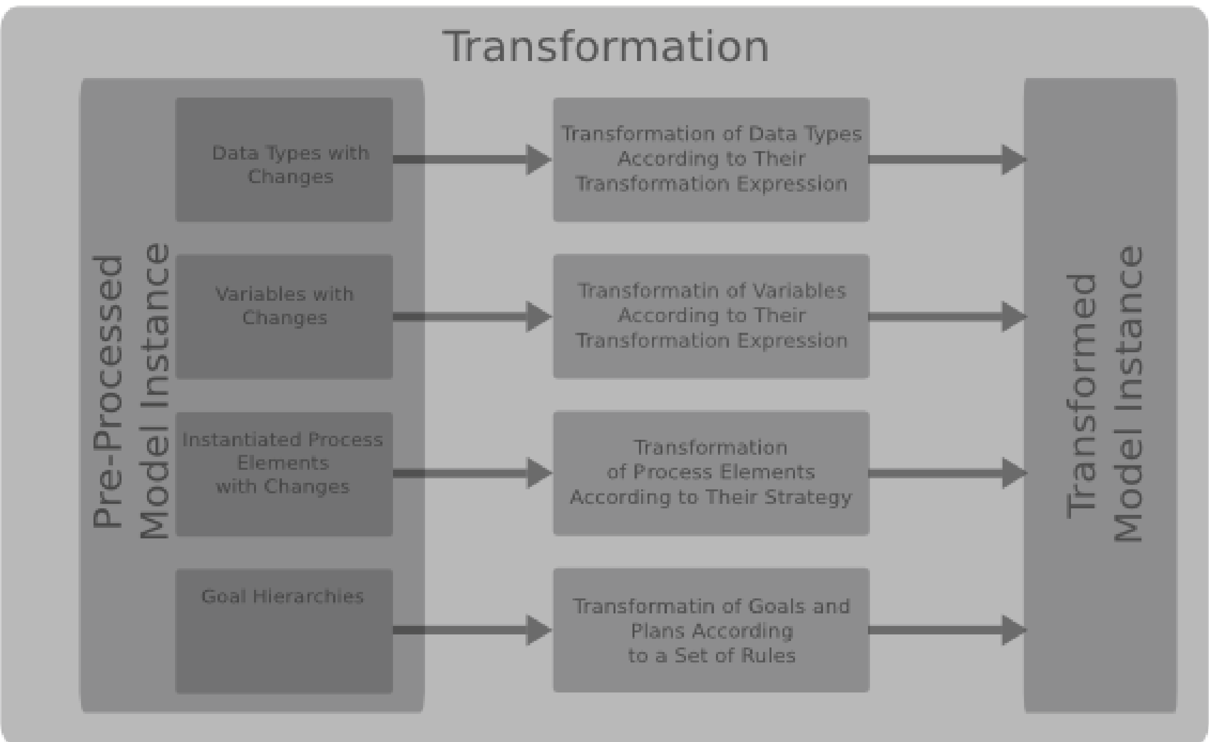


Figure 4.2 Schema of transformation

The way they are transformed is define by transformation rules in the model update definition.

For example, it is not necessary to define transformation on a function definition, since it is called and returns an output in one step. If a function is modified in the new model, the function is simply substituted with the new implementation and next function call uses the model instance the new function. Also, if you delete a variable in the new model, the variable is simply removed and no other action is required. However, if you change the type of a record field on a record, any values of variables of the record type must be transformed accordingly.

4.2.1 Record Transformation

Record transformations are defined for record fields of basic data types that have been added or changed in the new model as the transformation expression: the expression return the value for the new record field instance.

If a record has a supertype, it inherits the transformation expression. If it defines its own transformation expression, the transformation expression of its supertype is ignored.

> **Important:** It is not possible to transform the type of a Shared Record field since this requires database migration.

The evaluation is performed for each field instance in the context of the new model instance; To refer to the values in the original model instance, use the `old()` function call: the function takes a string expression that resolves into the name of the record property.

Example: The old() function call syntax

```
old(<STRING_EXPRESSION>)
```

4.2.2 Variable Transformation

Variable transformations are defined for variables that have been changed in the new model. The transformation expression returns the new variable value. Note that variable transformation can be influenced also by the transformation on the records. Therefore, on model update transformation, **the variable is transformed according to the variable transformation expression. If such an expression is not defined, it is transformed according to the record transformation.**

4.2.3 Asynchronous Modeling Elements Transformation

Asynchronous modeling elements that can be running when model update is triggered and exist in the new model must define their transformation strategies. Deleted or added asynchronous elements do not require any special actions on model update.

If you remove an asynchronous model element from the new model and the element was running when model update was triggered, the Start Event of its parent activity becomes running when the model instance resumes (the token from the deleted element is moved to the closest Start Event).

Transformation of asynchronous elements is performed as follows:

- The transformation of modified data types and variables is applied on all instances of the element.
- The transformation defined on asynchronous model elements is applied on the elements that were running before or will be running after the model update.

Goals and elements with atomic execution semantics do not define Transformation strategy since they cannot hold a token and can be only deleted or have their parameter values modified. The behavior of goal structures on model update is described in Goal Hierarchy Update.

4.2.3.1 Transformation Strategies

The transformation strategy specifies how an instantiated asynchronous element that is changed in the new model and was running when model update was triggered or will be running after the model update, is handled.

The following model elements must define their transformation strategy if modified in the new model:

- Processes
- Plans
- Activities

The strategy can be set to *restart* or *continue*:

- restart: If the element was running when the model update was triggered or will be running after the update, the context of the element is discarded and created anew.
- continue: If the element was running when the model update was triggered or will be running after the update, the context is preserved.

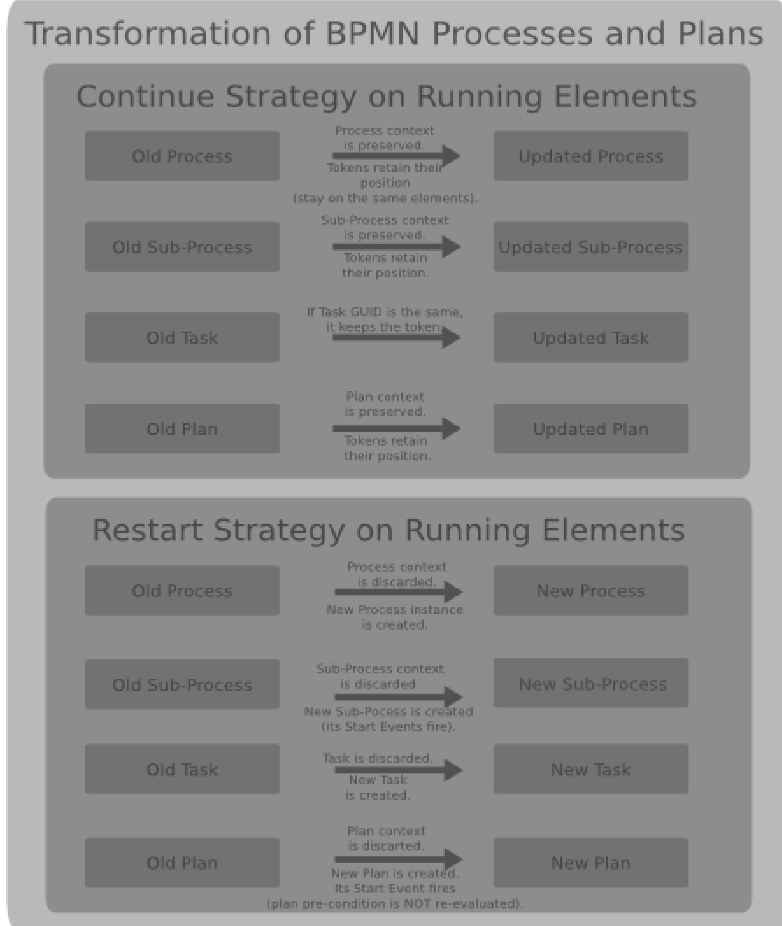


Figure 4.3 Transformation strategy semantics

The strategies are applied in a top-down manner: If a task in a sub-process has its strategy set to continue and the subprocess has its strategy set to restart, the continue strategy of the task will be never applied, since the parent subprocess will be always restarted if its content is running.

4.2.3.1.1 Goal-Hierarchy Update

Since Goal hierarchies need to be updated in a way which preserves the status of the Goals and adjusts it if necessary so that the hierarchy is consistent.

On transformation, the statuses of matched Goals are preserved and the following rules are applied:

- If a Goal is decomposed in Plans and at least one of the Plans is running, the Goal becomes running.
- If a Goal is inactive and its parent is deactivated, it becomes deactivated as well.

4.3 Performing Model Update

When updating a model instance, you proceed usually as follows:

1. Import the model you want to update into your workspace: note that you can download the model from the server either using the Management perspective or the Management Console.

2. Create a copy of the model and modify it to become your new model: make sure to keep this original model unmodified.
3. Create a model update definition file for the original and new model and define in it the instructions on how to handle differences between the models.
4. Upload the new model to the server and update the model instances.

4.3.1 Creating a Model Update Configuration

How model update is executed on a running Model instance is defined in a Model update configuration file. This definition file stores mappings between modeling elements of the old and new model as well as all the related update data.

Before creating the definition, make sure your workspace contains both: the source (old) module and the target (new) model.

To create a model update configuration:

1. Go to File > New.
2. Select Model Update Configuration.
3. In the New Model Update Configuration dialog box, type the path to the source (old) and target (new) models.
4. Click Next.
5. In the updated dialog box, select the project where to save the model update configuration.
6. In the File name text box, type the configuration name.
7. Click Finish.

Whenever you change the sources (old and new models), refresh the configuration: In the GO-BPMN Explorer, right-click the configuration and select Refresh.

> **Important:** When you generate the model update, the system attempts to map the old elements with changes to the new elements. The proposed configuration might not be accurate and it is strongly recommend to review the file thoroughly.

4.3.1.1 Copying Model Update Data

Once you have defined the update data, that is, transformations and model update processes, you can copy the data and paste it into another model update configuration.

Note that pasting works only if the Module name is the same as the original Module and the type of update data is correct: you cannot paste model update data from variables to data types.

To copy the variables, processes, or data type update data, do the following:

1. Open the `.muc` file.
2. Open the tab with the data.
3. Right-click the module and click **Copy Module Configuration**.

4. Open the target `.muc` file and the tab with the model update data.

5. Right-click the target module and click **Paste Module Configuration**.

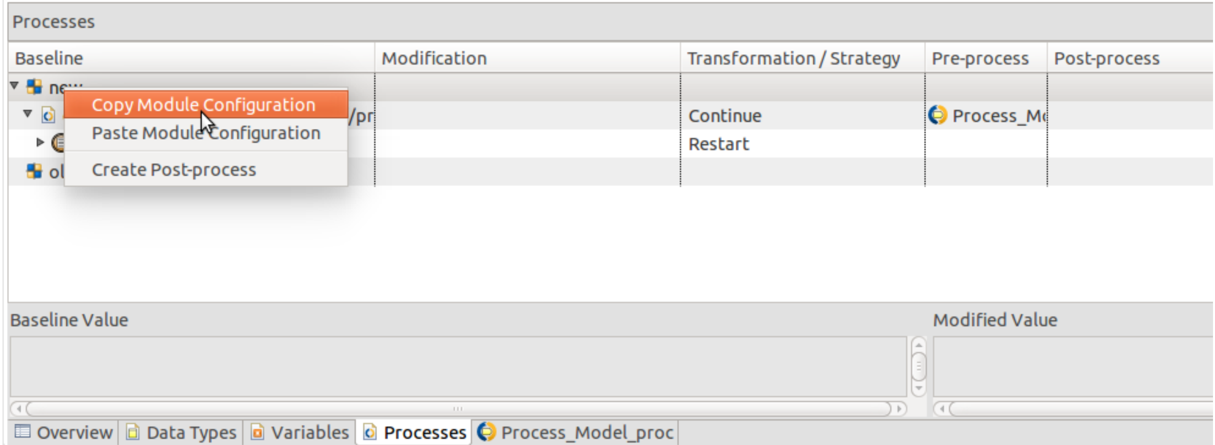


Figure 4.4 Copying model element changes on a module

4.3.2 Editing Model Update Configuration

Model Update Configurations are edited in the model update multi-page editor, which is by default associated with model update configurations (`.muc`).

The editor opens the configuration with the update rules sorted on the following pages:

- Overview with general model update information and settings
- Data Types with differences on data types and their transformation expressions
- Variables with differences on variables and their transformation expressions
- Processes with differences on processes and their elements, and their strategies as well as their model update process references

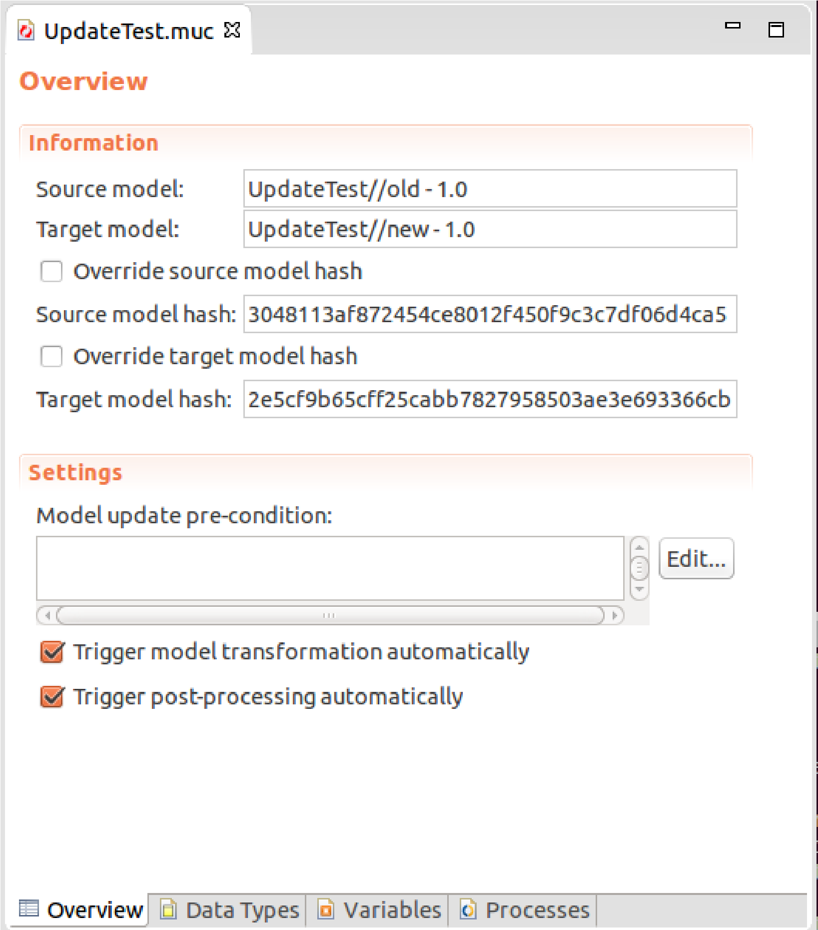


Figure 4.5 Model Update Editor

Every page contains the following:

- Baseline column with the resource with the change
- Modification column with the respective change marker and its description (see Comparison Editor);
- Transformation/Strategy column with data on how to handle the difference.

Pre-processes and post-processes are displayed in the Outline view.

4.3.2.1 Displaying Matching Data in Model Update Configuration

When the user creates a model update definition (.muc) file, the system automatically maps the elements of the old model with the elements of the new model and compares the models. Based on the comparison, the editor provides options for update rules.

To display the matched data, which is hidden by default, do the following:

1. Open your .muc file in the Model Update Editor.
2. Click the Filter button ( ).
3. In the Differences Filter dialog box, select Matched Properties and Elements Without Changes as required.
4. Click OK.

4.3.2.2 Changing Element Mapping in Model Update Configuration

When the user creates a model update definition (.muc) file, the system maps the elements of the old model with the elements of the new model and performs diff of the models. The automatic mapping might need some manual adjustment.

When changing mapping, you can match only elements with the same parent and of the same type. Recognized element types are start event, intermediate event, task, sub-process, plan, and process.

To change or cancel the mapping of an element, do the following:

1. Open your .muc file in the Model Update Editor.
2. Display the matched elements if necessary (click the Filter ( ) button, and select Elements without Changes).
3. In the Model Update Configuration Editor, right-click the element entry with the incorrect mapping.
4. In the context menu, select the applicable option:
   - Select **Change Source of Mapping** and select the source element.
   - Remove mapping with **Cancel Mapping**.

4.3.2.3 Changing Model Update Settings

Model update configuration defines general properties that are used to control the update process, such as, the source and target model, pre-condition, etc.

To change general model update settings of your model update configuration, do the following:

1. Open the model update configuration (.muc).
2. On the *Overview* page in the *Settings* area, define the model update settings:
   - Source (old) and target (new) model
     Make sure to refresh the definition whenever the source or target model change.
   - Override source/target model hash
     The source and target models are identified by their hash codes, which change whenever a Model is changed. Therefore it is not possible to apply a model update configuration on Model instances of a Model that was changed after the model update configuration was created. If you want to use another hash code, select the option and enter the Model hash code below. You can check the hash code of uploaded modules in the `Module Management` view.
   - Model update pre-condition
     When a model instance update is triggered, the server evaluates the pre-condition: If false, the model update is rejected and the execution of the model instance continues.
   - Trigger model transformation automatically enables and disables automatic start of transformation phase of the model update
     If false, the transformation phase must be triggered manually by the user.

- Trigger post-processing automatically enables and disables automatic triggering of post-processing phase after transformation

  If false, the post-processing phase must be triggered manually by the user.

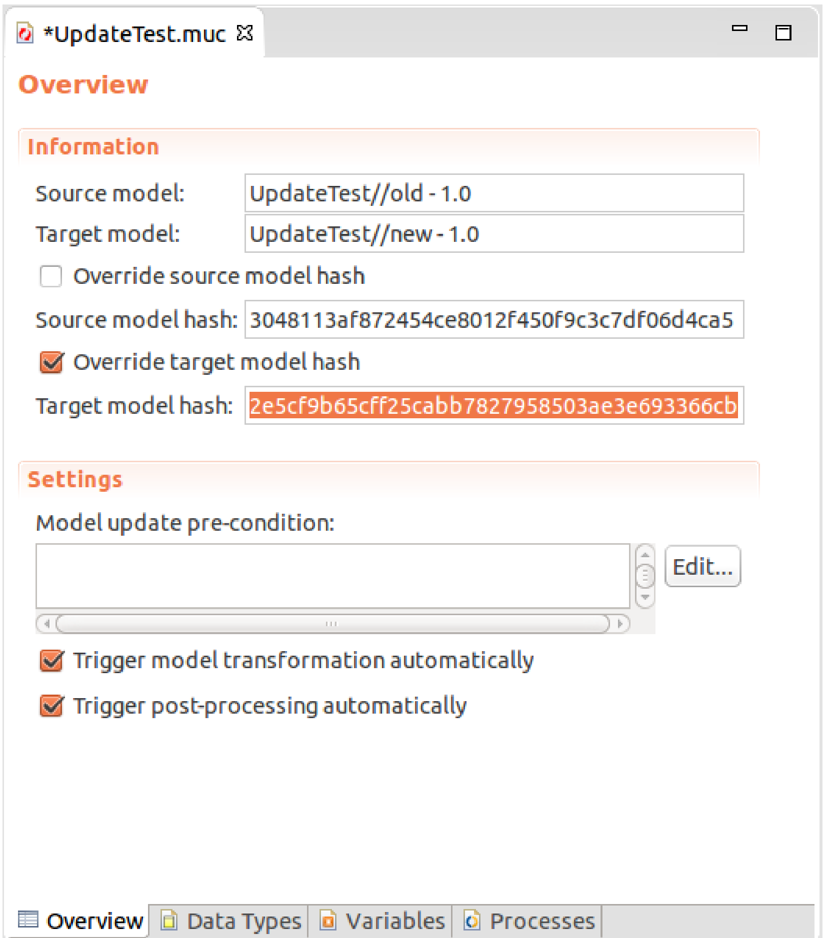


Figure 4.6 Changing hash code of the target Model

3. Click OK.

4.3.2.4 Defining Transformation

Transformation for added and changed records or fields and changed variables is defined as a transformation expression. When model update configuration is created or refreshed, the system attempts set automatic transformation rules for changed entities. If this is not possible, the entity entry is highlighted.

For details on the concepts behind transformation, refer to Transformation.

To change transformation strategy or expression, go to the entry in you muc file and click the Transformation cell.

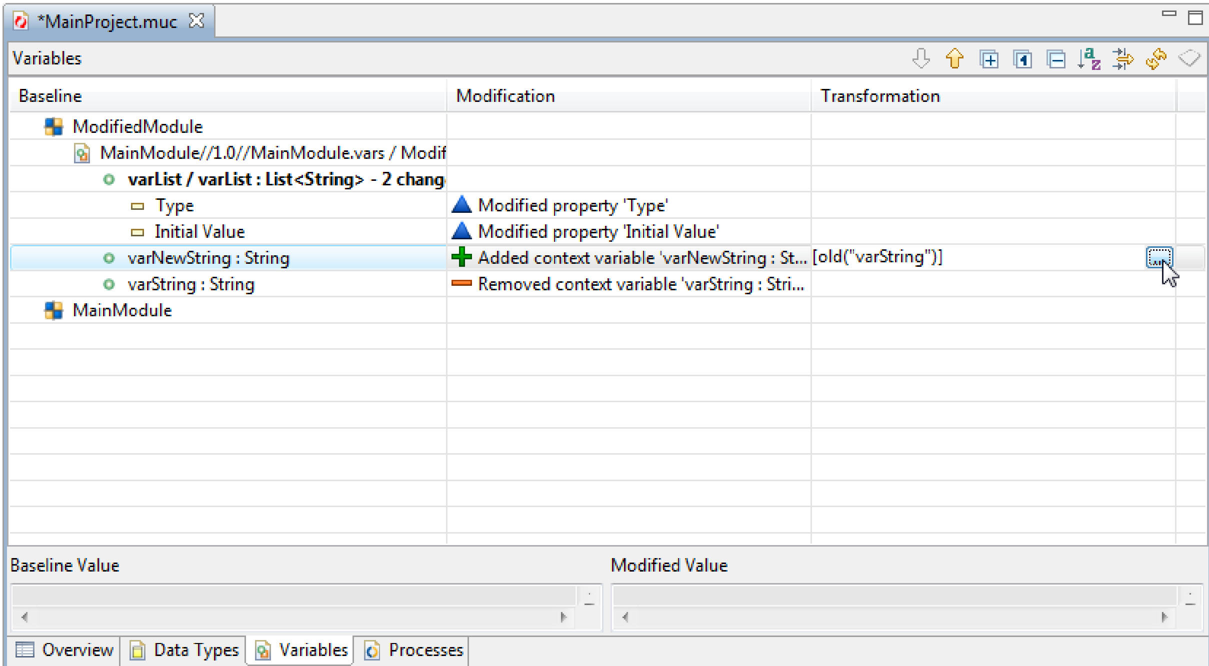


Figure 4.7 Defining variable transformation expression

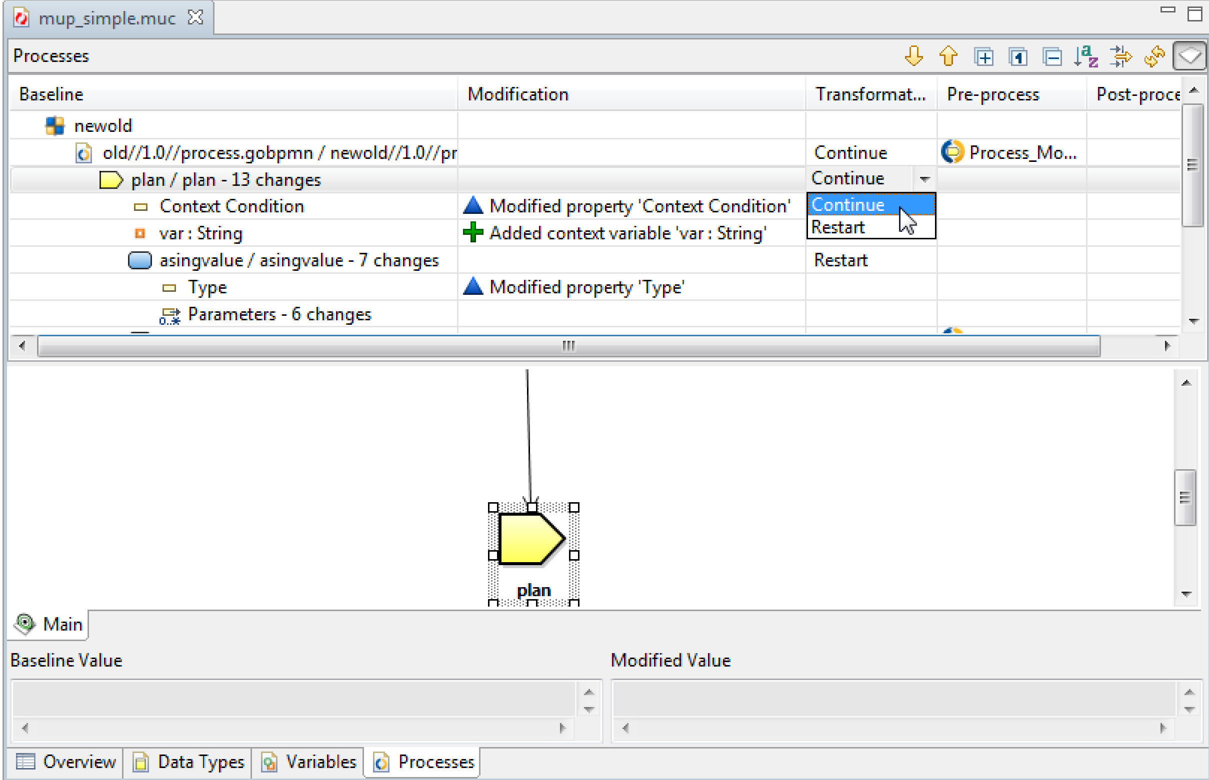


Figure 4.8 Selecting transformation strategy

4.3.2.4.1 Defining Process Element Transformation

To define transformation strategy for elements, do the following:

1. Open the model update configuration.
2. Click the Processes page title in the lower part of the Model Update Editor.
3. Click the Transformation/Strategy column in the row with the element entry and select Continue or Restart.

Note that if the strategy of a parent element is set to Restart and the child element has the strategy Continue, the Continue strategy will be never applied since the parent element will be always restarted.

4.3.2.4.2 Defining Data Type Transformation

The transformation of data types on model update is defined in the model update configuration as an expression that defines how the value of every instance of the old data type is transformed into the new data type value.

> **Important:** It is not possible to define a transformation for the field type of a Shared Record to a field of another type, since such a process requires database migration.

To define transformation expression for data types, do the following:

1. Open the Data Types page of the model update configuration.
2. Click the Transformation column in the row with an added or modified data type and then the Browse button on the right.
3. Specify the transformation expression.

   Example:

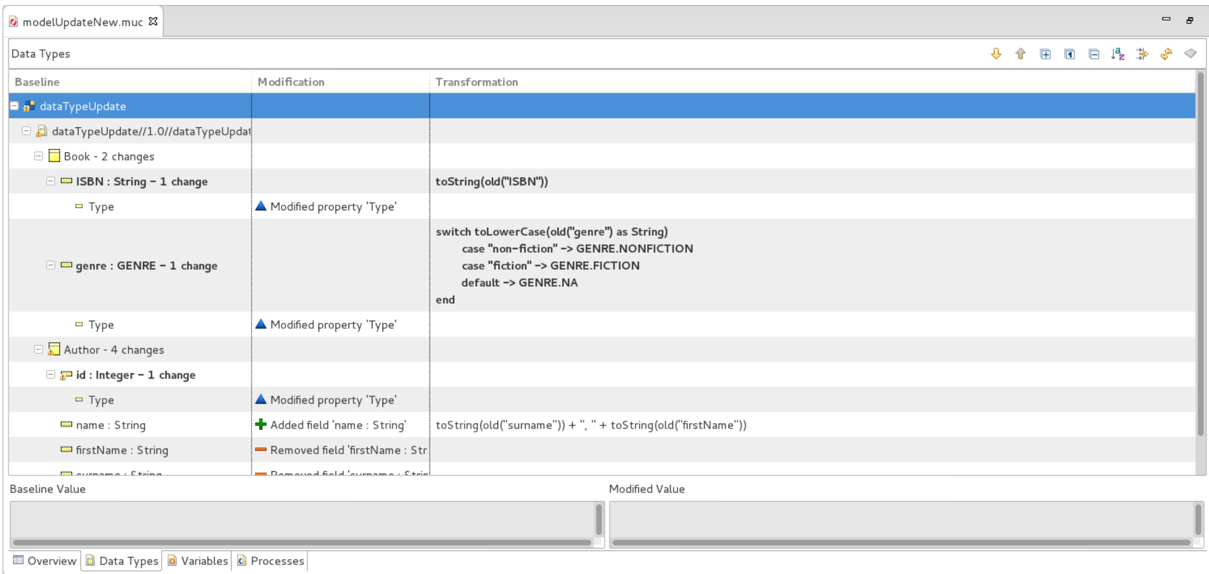


Figure 4.9 Defining data type transformation

The data type transformation is defined for a Book record:

- The `ISBN` record field has changed from Integer to String. The old ISBN instances will be transformed with the `toString(old("ISBN"))` expression.
- The field `genre` of the Book record was originally of the String type. In the new model, the field is an Enumeration. Any instances of the field will be transformed to the value with the `switch` expression.
- On the related `Author` Record, the fields `firstName` and `surname` have been removed and the record field name has been added. The new field will be populated with the concatenated version of the removed fields. Note that the function `old()` returns an Object and therefore the call must be wrapped into a `toString()` function call.

4.3.2.4.3 Defining Variable Transformation

Variable transformation is defined by a transformation expression in the model update configuration as a closure that returns a value of a basic data type evaluated within the context of the old model instance.

To define the transformation strategy for a variable do the following:

1. Open the model update configuration.
2. In the Model Update Editor, do one of the following:
   - For process variables, open the Processes tab and expand the parent element of the variable.
   - For global variables, open the Variables page.
3. Click the Transformation column in the row with the variable and click the Browse button.
4. In the Transformation Expression Editor dialog box, click Edit and specify the transformation expression in the Expression Editor.

   To use a value from the old context use the `old()` function call.

4.3.2.5 Creating a Model Update Process

Model update processes in model update definitions are defined in the model update configuration per relevant process element. They serve to prepare an instantiated element for transformation (pre-processes) and for renewed running of the model Instance after transformation (post-processes).

To create a model update process, do the following:

1. Open a model update configuration in Model Update Editor.
2. Open the Processes page.
3. Right-click the row with the element entry with the change (module, process, plan, sub-process, behavior, or task).
4. In the context menu, select **Create Pre-proces**s or **Create Post-process**.
5. Design the model update process as a common BPMN-based process.

Model update processes are accessible from the Outline view.

4.3.2.5.1 Importing a Module to Model-Update Processes

To import a module to all pre- or post-processes of a model update, do the following:

1. In the GO-BPMN Explorer, select the muc file.
2. In the Outline view, right-click the Pre-processing or Post-processing folder in Outline and click **Add Import**.

3. In the Add New Import dialog, select the module and click OK.

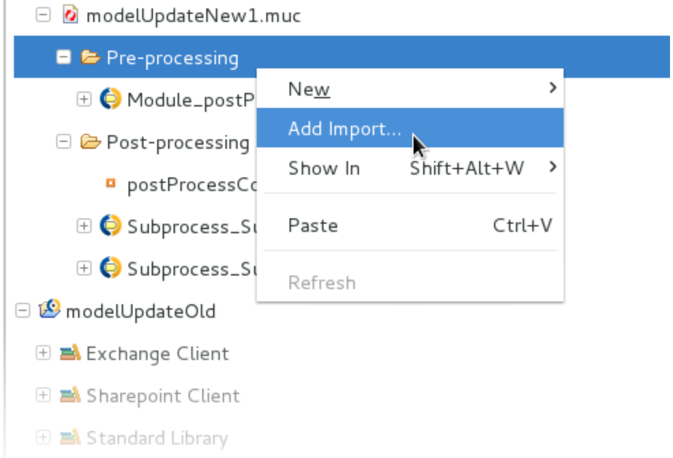


Figure 4.10 Importing Module

4.3.2.5.2 Creating a Variable in a Model Update Process

To create a variable in the pre-processes or post-processes of a model update, do the following:

1. In the GO-BPMN Explorer, select the muc file.
2. In the Outline view, right-click the Pre-processing or Post-processing directory.
3. In the context menu, select New > Context Variable.

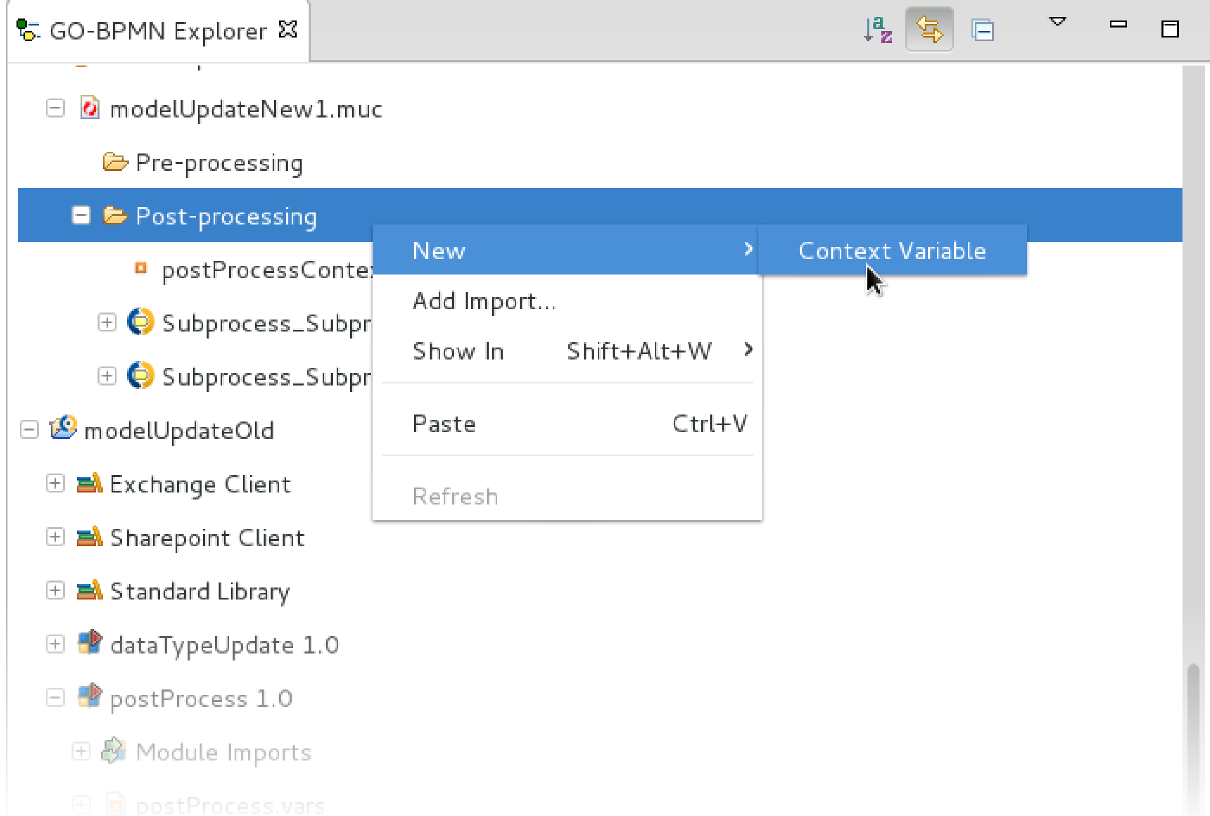


Figure 4.11 Creating variable in a model update process

4.3.2.5.3 Editing a Model Update Process

To edit a model update process, do the following:

1. In the Model Update Editor, open the .muc file and then the Processes page.
2. Right-click the row with the required element entry.
3. In the context menu, select Open Pre-Process or Open Post-Process.
4. In the Process editor, edit the Process and save.

Alternatively, double-click the model update process in Outline.

4.3.2.5.4 Deleting a Model Update Process

To delete a model update process from your muc file resource do the following:

1. In the page title bar (at the bottom of the editor), click the Processes page tab.
2. Right-click the row with the element.
3. In the context menu, select Delete Pre-Process/Delete Post-Process.

4.3.3 Updating Model Instances

Before updating model instances, make sure the new Model (target) is uploaded. Note that only model instances that are *Running*, *Suspended*, *Updated*, or *Update Aborted* can be updated. During update, the model instance is suspended and becomes read-only. Such a model instance cannot be resumed manually.

When updating Model instances their IDs are preserved and the rules defined in the Model update configuration are applied.

Note: It is not possible to update a model to a model with a different version of Standard Library.

You can update model instances either from PDS or from the command line with the LSPS cli tool.

4.3.3.1 Updating a Model Instance from PDS

To update model instances, do the following:

1. Open the Model Instances view.
2. Click the Update ( ) button in the view toolbar.
3. In the Model Update dialog box, specify the model update configuration and data mapping file to be applied:
    - Click Workspace Resource to select a configuration/data mapping file from the current workspace.
    - Click External File to select a configuration/data mapping file from other location.

4. Click Next.

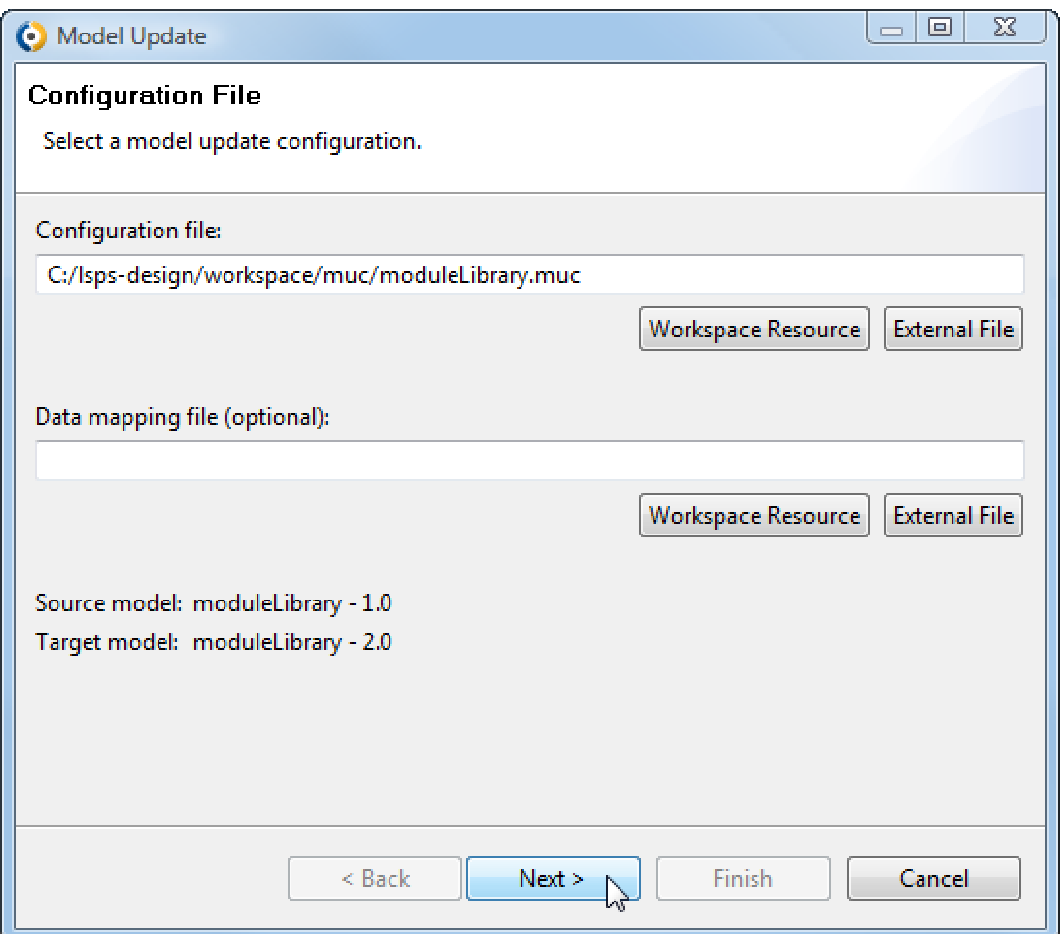


Figure 4.12 Model update dialog

5. On the Filter page, define the filtering criteria for model instances and click **Apply Filter** if applicable.

   You can apply also a previously defined filter. However, status and model criteria are ignored since the model is defined in the Model Update Configuration.

6. In the Model Instances area, select the model instances to update.

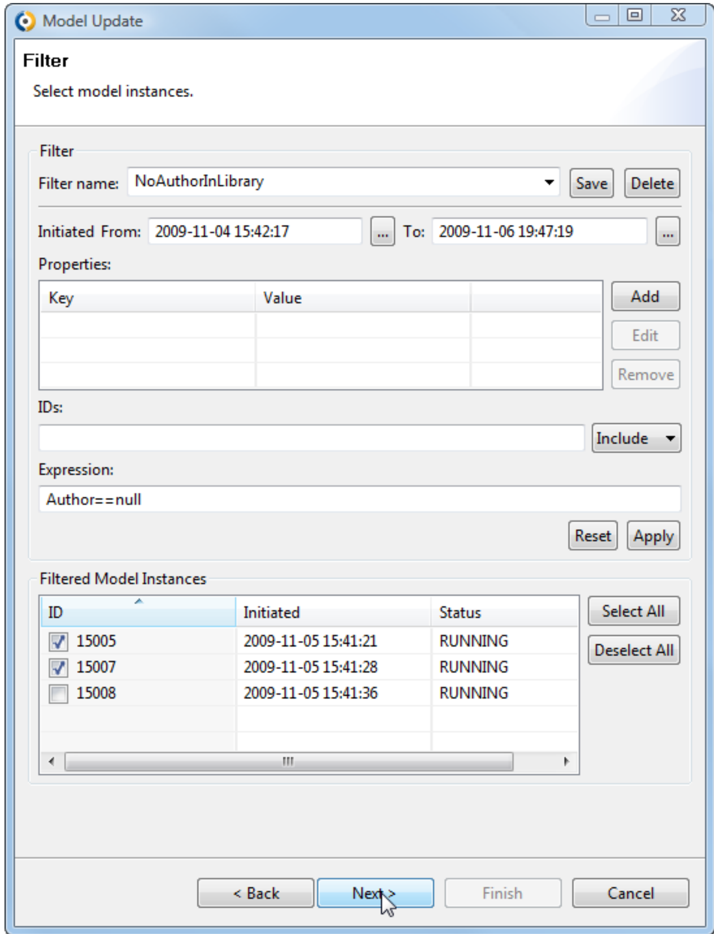


Figure 4.13 Model update dialog box with filtered Model instances

7. Click Next.
8. On the Summary page, chheck the settings. If you plan to perform the model update with the LSPS Command-Line Console, click **Show command line script** and copy the commands for later use.
9. Click Finish.

The selected instances go through the model-update process. Check their detail views and the Model Update Logs view for details. Depending on the model update settings, the update process may be waiting for manual triggering of its transformation or post-processing phase: To resume a model update of a model instance, click the Continue Model Update ( ) button in the Model Instances view in the Management perspective.

After the update has finished, the model instances remain in the status *Updated* and are still not running. To resume the instances, click **Resume** in their context menu.

To resume all model instances returned by the Filter of the view, click **Resume All Updated** in the context menu.

> **Note:** If resuming a large number of instances, make sure the resumed action was applied to all of them. Resume any instances that failed to resume.

4.3.3.2 Aborting Model Update

You can abort model updates from the Management perspective: Click the **Abort Model Update** button ( ) in the view toolbar.

The feature is available in the Management Console as well.

4.3.3.3 Model Update Logs

The logs from a model update procedure are available in the Model Update Logs view. The view displays information on source and target models and details on updated model instances (below the list of model updates).

4.3.3.3.1 Model Update Detail

The Model Update Detail view contains details about the update of a model instance.

To display the view, click the model instance ID in the Model Update Logs view.

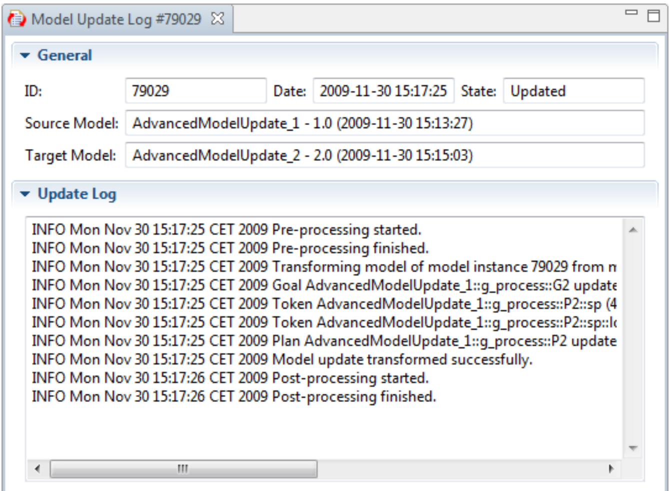


Figure 4.14 Model Update Detail

4.3.3.4 Downloading Model Update Configuration

To download a model update configuration from the LSPS Server, do the following:

1. Open the Model Update Logs view.
2. Select an update log.
3. In the view toolbar, click the Export MUC ( ) button.
4. Define the target location and file name.

Chapter 5

Model Update Examples

This section contains examples of simple model updates with the following modifications:

- Variable value change
- Task parameter change
- Event change
- Data type change

5.1 Updating a Variable Value

Required action: Update a model instance so that a variable value changes to a value derived from its original value.

In the example update, we will introduce the following changes on global variables:

- A variable will be removed.
- A variable will have its value modified to a value derived from the removed variable.

1. *Design the source model* with a process definition and a variable definition:
   (a) Create a module with a global variable definition with the following variables:
      - `varSet` of the type `Set<String>` with the initial value `{"old value 1", "old value 2"}`
      - `varString` of the type String with the initial value `"42"`

   (b) Create a process definition with a None Start Event, a Conditional Intermediate Event, and a Simple End Event.

   (c) Set the Condition parameter of the Conditional Intermediate Event to `false` to keep the model instance running so it can be updated.

Figure 5.1 Process

2. *Design the target model:*

   (a) Copy and paste the old module.

   (b) Modify the global variables

   - Modify `varSet` to have the initial value `{"new value 1", "new value 2"}`
   - Delete `varString`
   - Create `varInt` of the type Integer with the initial value `1`.

3. *Create the `.muc` file*:

   (a) Right-click the parent project, go to New > Model Update Configuration and follow the instructions.

   (b) Open the Variables page in the newly opened editor with your muc file

   (c) Adjust the mapping if necessary: Map the new variable definition file to the old variable definition file and `varInt` to `varString`. Mapping of `VarSet` should be recognized automatically after the variable definition file is mapped.

   (d) Define the transformation expressions on the new variables:

   - `varSet: {"transformation value"}`
   - `varInt: toInteger(toString(old("varString")))`

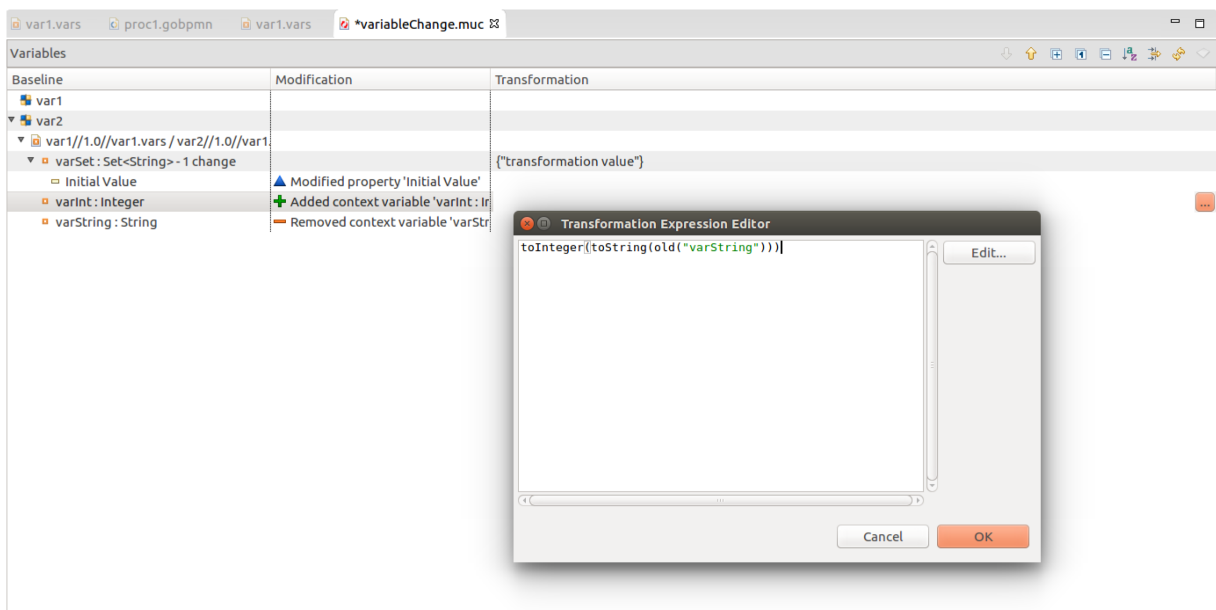


Figure 5.2 The muc file with variables and their transformation expressions

When you perform the model update, the system does the following:

1. First attempts to transform variable values according to their transformation expression.

2. If the expression does not exist, the system performs the transformation defined for the variable data type.

3. If neither the variable transformation expression nor the data type transformation exist, the variable is initialized. This typically applies to variables that were added in the new model.

Note: When updating **local variables** of processes, sub-processes, and tasks, the update is determined by the update strategy of the parent element:

- If the strategy of the parent element is continue, the parent context is preserved. The execution continues in the old transformed context: Its local variables are transformed as defined by their transformation expression.

- If the strategy of the parent element is restart, the parent context is dropped and a new context is created: any local variables are discarded and new variables are initialized. The transformation expression on the variables is not applied.

To upload the resources and perform the model update, do the following:

1. Make sure your server with the Execution Engine, possibly the PDS or SDK Embedded Server, is running and your PDS is connected to it.
2. Upload the model to the server and create its model instance: In the GO-BPMN Explorer, right-click the source module and go to Run As > Model.
3. Upload the target model to the server: In the GO-BPMN Explorer right-click the module and go to Upload As > Model.
4. Switch to the Management perspective.
5. Refresh the Module Management and Model Instances view and check that both models are uploaded and the source model is instantiated.
6. In the Model Instances view, open the detail of the source-model instance and check the values of the global variables.

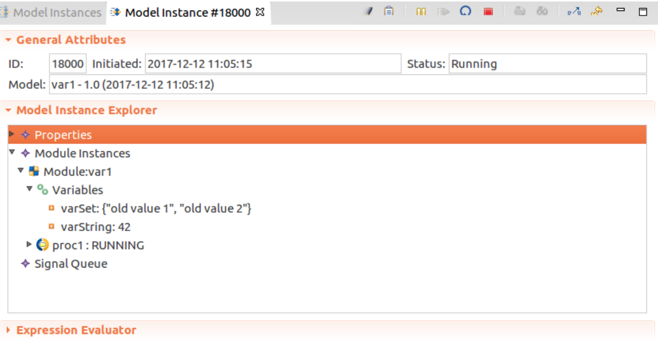


Figure 5.3 Detail of the old model with old global variables

7. Perform the update:
   (a) In the Model Instances view, click the Model Update ( ).
   (b) In the Model Update dialog window, provide the path to your muc file in the Configuration file field and click Next.
   (c) In the refreshed dialog, check that the model instance is listed and selected in the **Filtered Model Instances** section and click Next. Check the summary of the model update and click Finish.
   (d) Refresh the Model Instances view: The model instance should be in the Updated status.
   (e) Display the detail of the model instance.

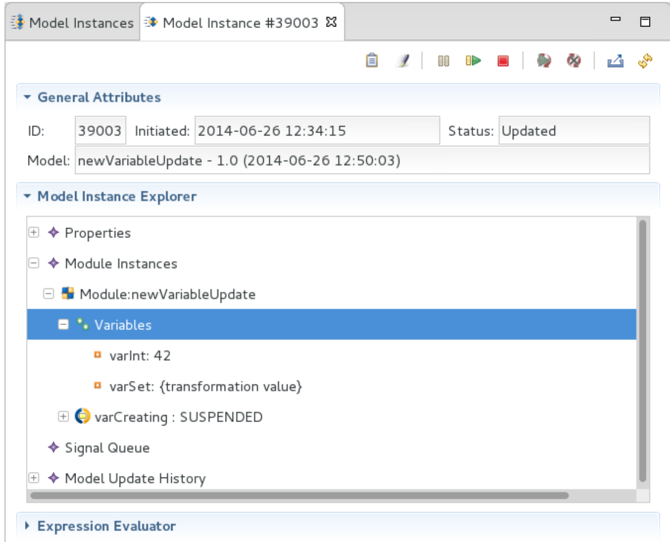


Figure 5.4 Detail View of the Updated Model Instance with New Global Variables

Note that the variables hold now the values defined by their transformation expression.

5.2 Updating a Task Parameter

Required action: Perform model update to a new model with changed task Parameters and have a post process log information about the update.

For this scenario, you should consider whether the task can be instantiated at the moment the model update is started or when the model instance is started after update:

- A task cannot be instantiated if it is **atomic** since it cannot be holding the token at that moment. Such tasks include the Log, Assign, Lock Task, etc. Changes on such tasks are considered as **a removal of the old task and adding of a new task**.
- A task can be instantiated when its task type requires **asynchronous or multi-step execution**, or waits for an event. These are tasks that can hold an execution token and become a transaction border (for information on transactions, refer to the `Software Development Kit Guide`). Such task types include the User, HttpCall, Web Service Client, and Server Tasks of the Standard Library and possibly custom tasks.

  For these tasks, you need to **define their transformation strategy** so that if such a task is running at the moment you start the model update, or it will be running after the model instance starts after model update, the task is handled according to the transformation strategy. The strategy can be either restart or continue:

  - If the strategy is set to restart, the task ignores its old context and restarts as a new task.
  - If the strategy is set to continue, the task continues in its old context.

Let us update the Performers and Form parameter of a User Task. We will change the following:

- **Perfomers**

  `{anyPerformer()}` will be changed in the new model to `{getPerson("admin")}`
- **Underlying Form**

  The form content will be changed in the new model.

We will consider the outcome of both the restart and continue strategy.

Proceed as follows:

1. *Design the old model* as a module with a process and define its form definition with arbitrary content.

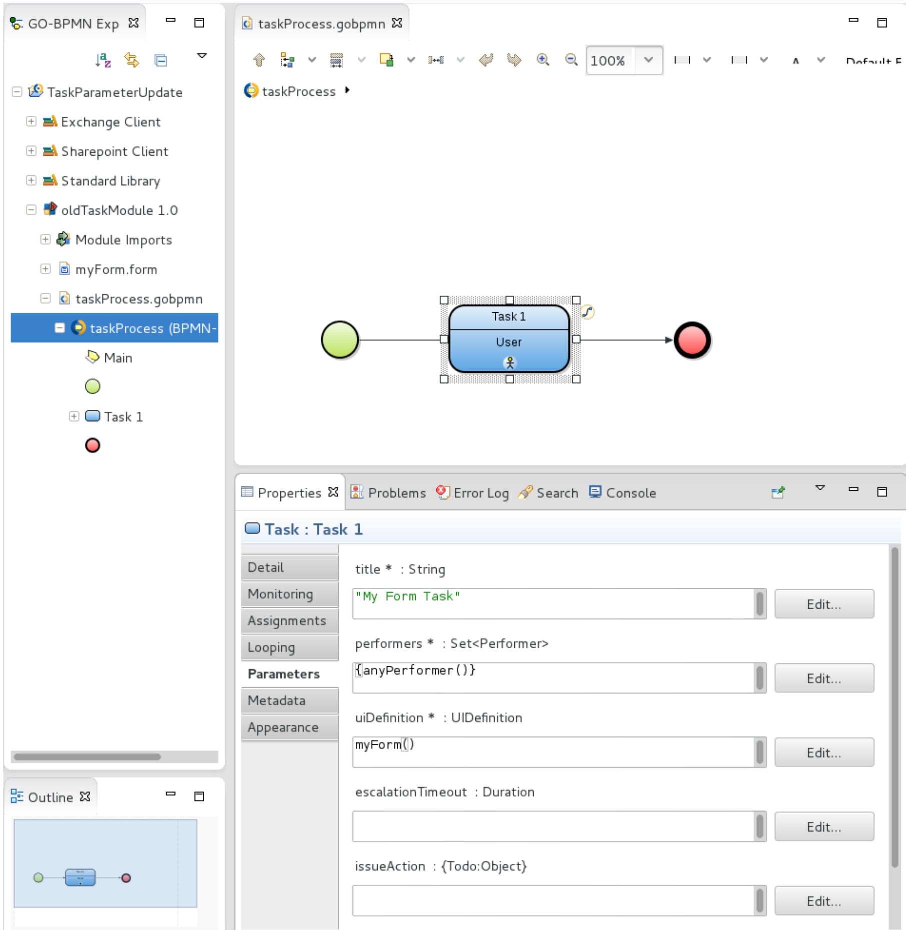


Figure 5.5 Old model

2. *Design the new model*: Copy and paste the old module and **modify the Performers parameter** of the User Task and modify the content of its **form** definition.

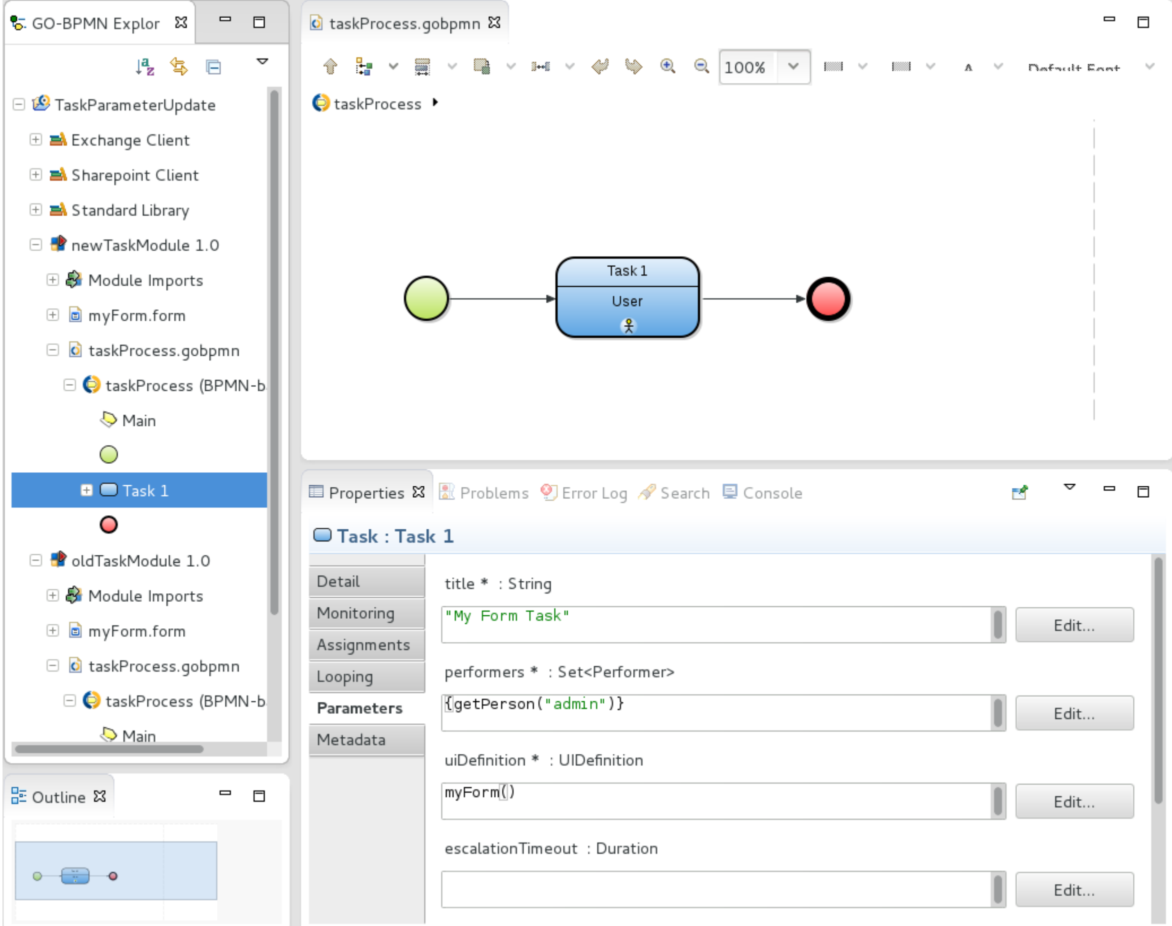


Figure 5.6 New model

3. *Create the .muc file*: Right-click the parent project, go to **New** > **Model Update Configuration** and follow the instructions.

4. Open the `.muc` file and on the Processes page locate the task parameter: The transformation strategy is set to Continue by default.

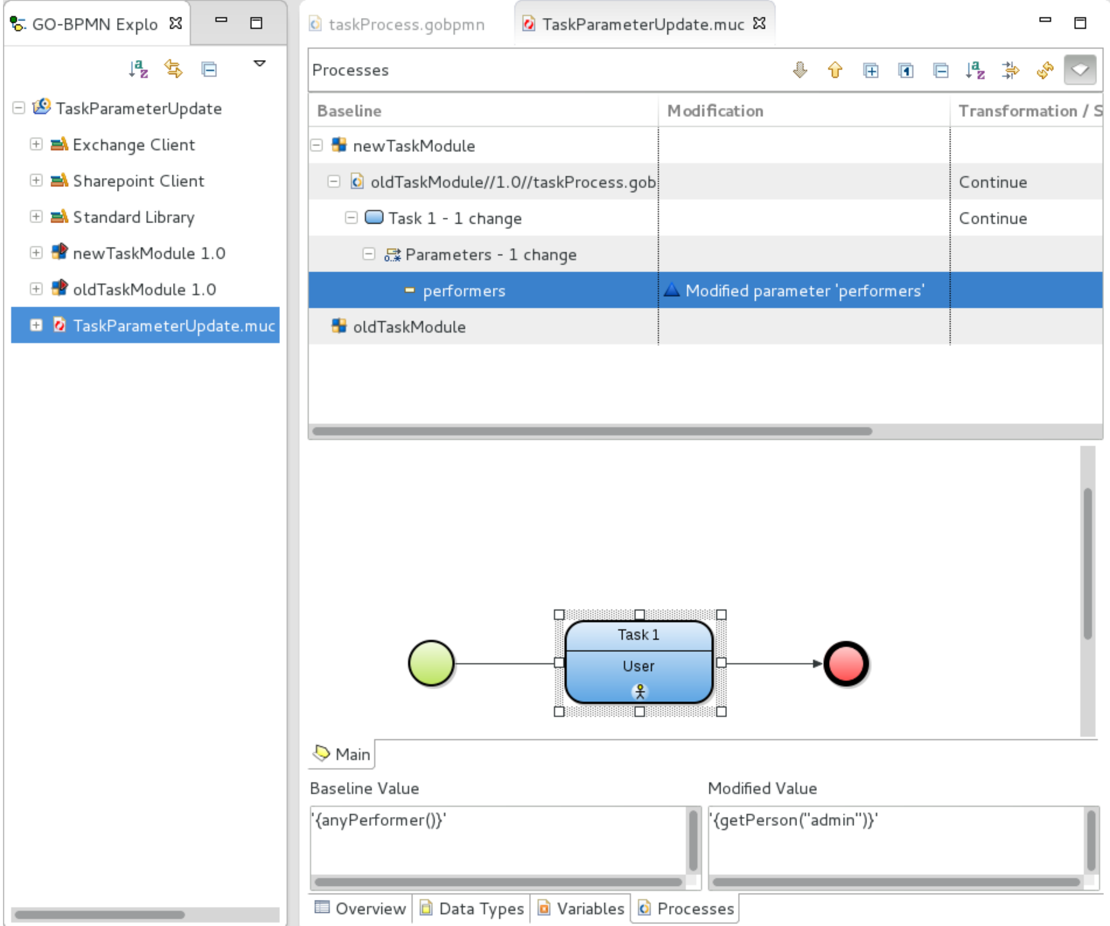


Figure 5.7 Model update configuration with the parameter change

Note that no changes on the form are detected since forms do not require any special handling on model update but are simply substituted with their new version.

5. Define a post process on the module that will log a message:

   (a) In the .muc file, right-click the new module and click Create Post-process.

      i. On the opened page, design the post process with a Log Task.

      ii. Define the message parameter of the Log Task.

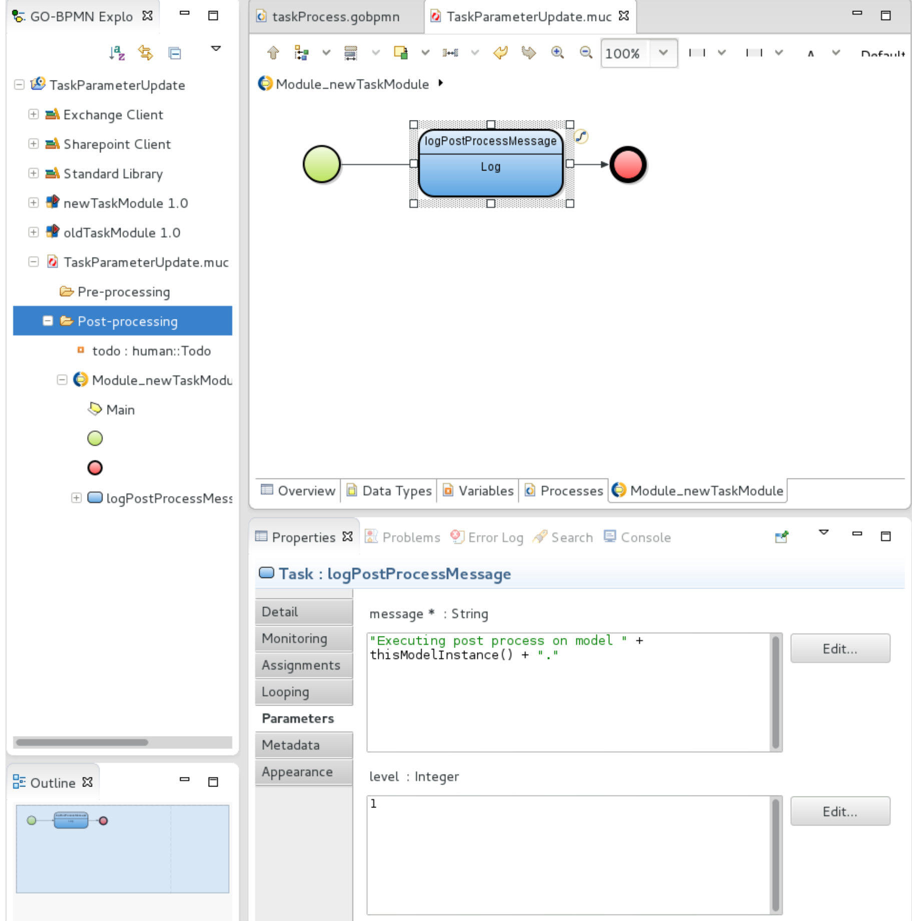


Figure 5.8 Post-Process

Make sure the transformation strategies on the module, process, and task of your muc file are set to Continue. This is the default transformation strategy.

To perform the model update, do the following:

1. Make sure your server with the Execution Engine, possibly on the PDS or SDK Embedded Server, is running and your PDS is connected to it.
2. Upload the model to the server and create a model instance of the old model: In the GO-BPMN Explorer, right-click the old module and go to Run As > Model.
3. Upload the new model to the server: In the GO-BPMN Explorer right-click the new module and go to Upload As > Model.
4. Switch to the Management perspective.

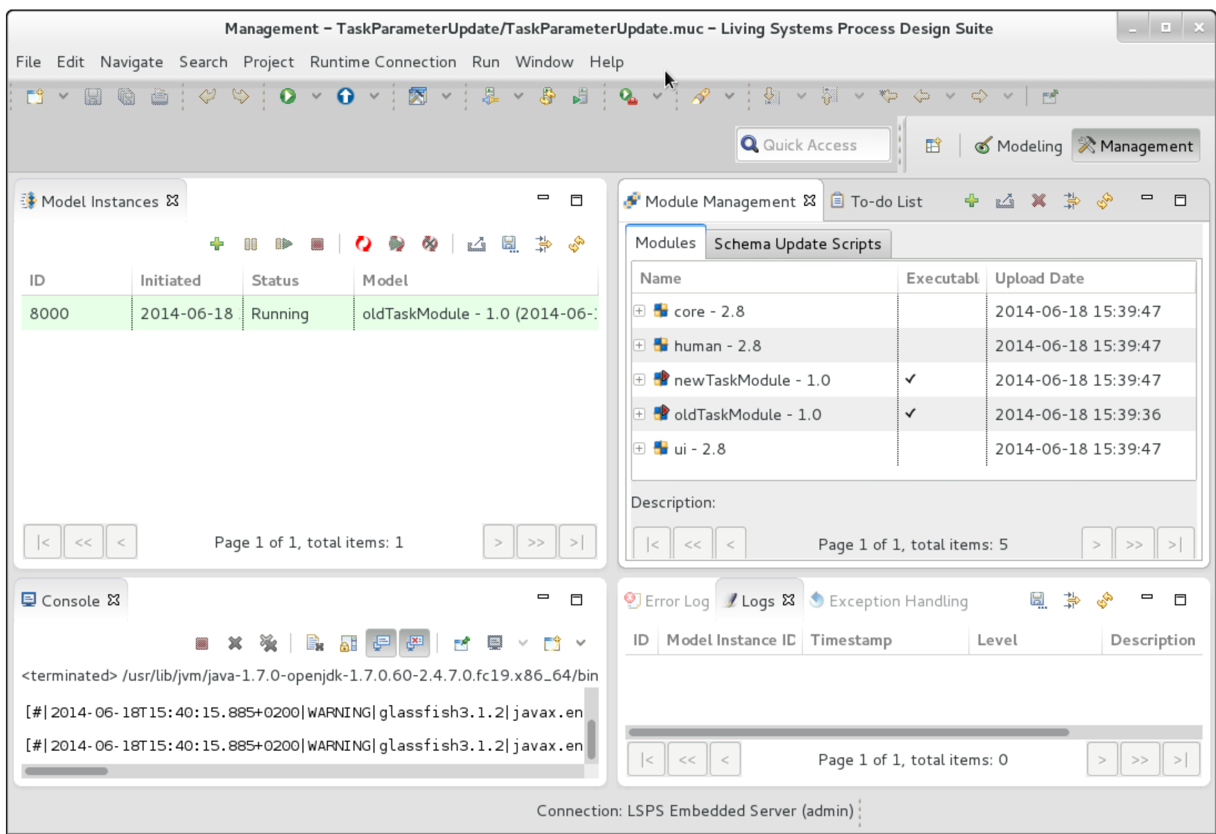


Figure 5.9 Management perspective with an instance of the old model

5. Refresh the Module Management and Model Instances view and check that both models are uploaded and the source model is instantiated.

6. Go to the Application User Interface and lock the generated to-do:

   (a) Open a browser and go to `http://DOMAIN/lsps-application/`

   (b) Log in as the user guest (By default, the password is set to guest for the guest user).

   (c) Click TO-DO LIST.

   (d) Open the to-do, which was generated by the old model instance.

   (e) With the to-do content displayed, log out, so the guest user locks the to-do.

7. Back in the Management perspective, perform the update:

   (a) In the Model Instances view, click the Model Update ( ).

   (b) In the Model Update dialog window, provide the path to your muc file in the Configuration file field and click Next.

   (c) In the refreshed dialog, check that the model instance is listed and selected in the **Filtered Model Instances** section and click Next. Check the summary of the model update and click Finish.

   (d) Refresh the Model Instances view: The model instance should be in the Updated status. If the model instance is in the Pre-processes state, hit the Refresh button again.

   If the model instance is in the Pre-processes state, hit the Refresh button again. The model instance is still suspended: If you check the to-do list of the guest user, the to-do is not available since the user task is suspended.

   (e) Check the Log view for the log message of the post-process.

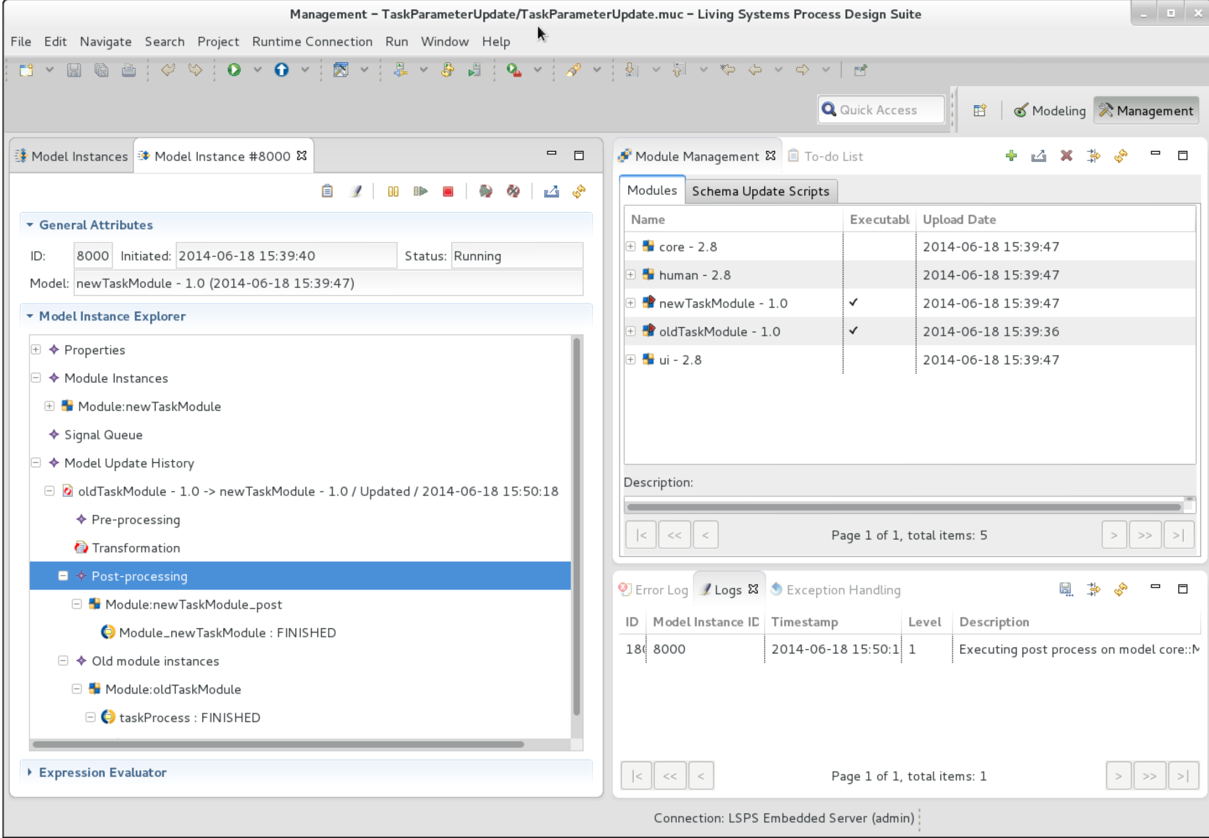


Figure 5.10 Log view with the post-process log message

(f) Select the model instance and click Resume ( ) button. The model instance becomes Running.

8. Go to the Application User Interface as the guest user. The to-do list of the guest user still contains the locked to-do in spite of the fact that the new model allows only the admin user as the to-do performer. However, its content already follows the form of the new model.

Set the transformation strategy on the `User Task` of your muc file to `Restart`. Leave the strategy on the parent process and module set to Continue and perform the model update anew. The to-do will be discarded.

Note: If you set the strategy on the parent process and module to Restart, the entire process/module will be discarded on update and a new process/module will be instantiated.

5.3 Updating an Event Type

Required action: Update a model instance so that its None Start Event is changed to a Conditional Start Event and Timer Intermediate Event changes in a Conditional Intermediate Event.

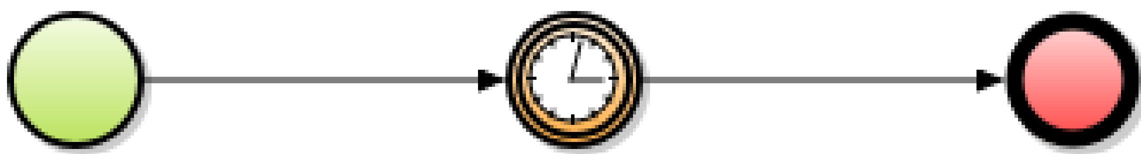
A change of an event type does not allow to define any pre- or post-processing on the event, or a transformation expression since the change is detected as a removal of the old event and addition of the new event. If required, define model-update processes on the parent modules and process.

1. *Design the old model* with a process definition:

   (a) Create a module with the old process.

(b) Create a process definition with a None Start Event, a Conditional Flow Event, and a Simple End Event.

(c) Set the Delay parameter of the Timer Intermediate Event, for example, to `new Duration(years -> 1)`.



2. *Design the new model:* copy and paste the old module, change the None Start Event to a Conditional Start Event and the Timer Intermediate Event to Conditional Intermediate Event, and set their condition, for example, to false.

3. *Create the .muc file:*

   (a) Right-click the parent project, go to New > Model Update Configuration and follow the wizard.

   (b) Open the `.muc` file and on the Process page and check the element mapping.

   The change mapping might be incorrect as shown in Model Update Configuration with Incorrect Mapping: The `newUpdateEventType` Process is recognized as a new Process, while we want it to be mapped to the `oldUpdateEventType` Process.

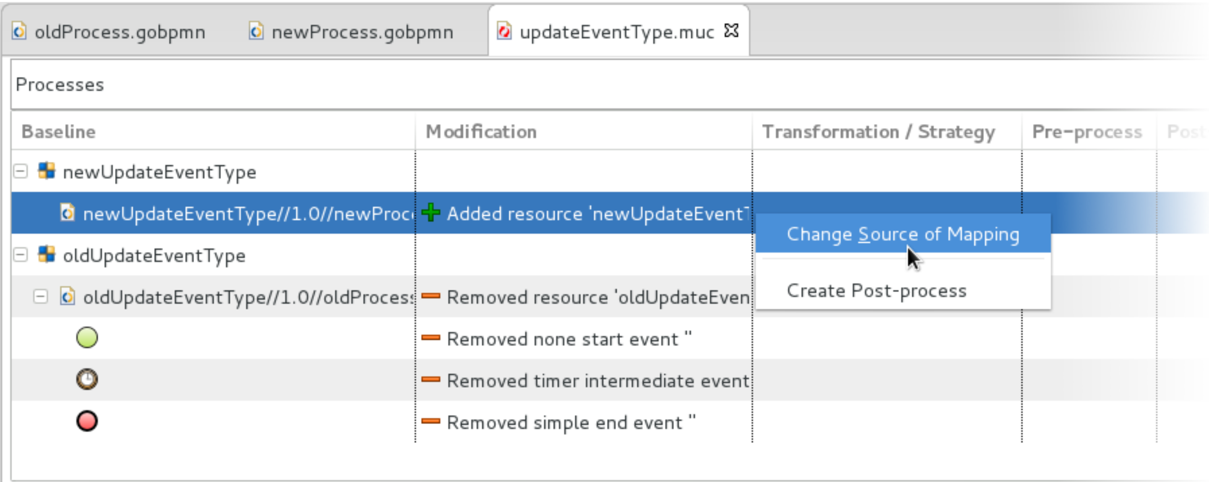


Figure 5.11 Model Update Configuration with Incorrect Mapping

   (c) Rigth-click the element and adjust the mapping if needed.

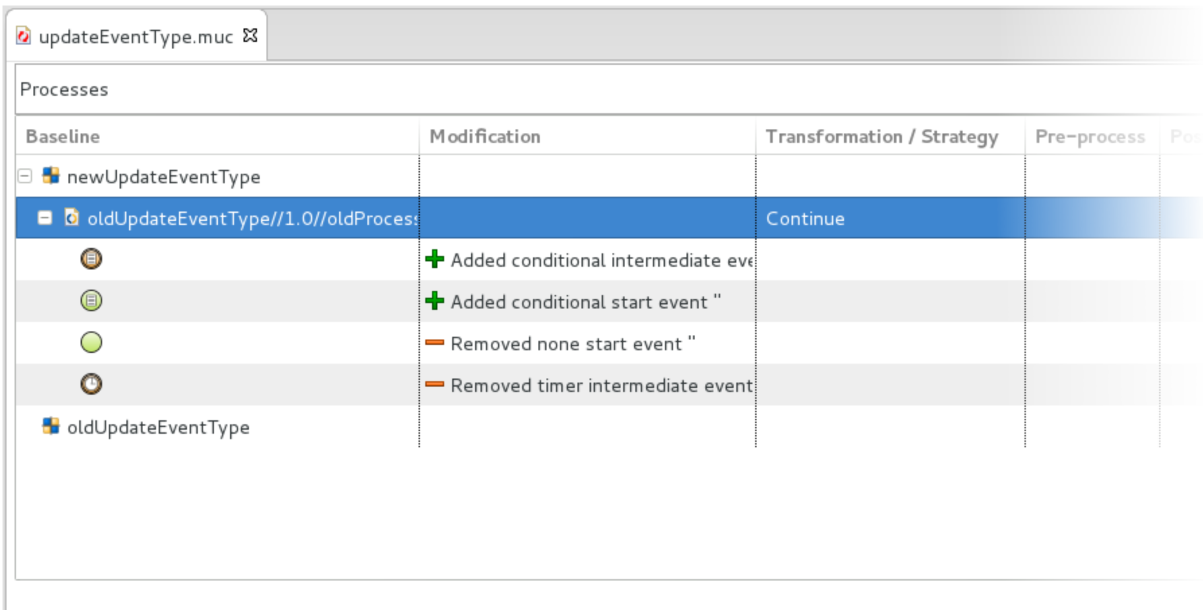


Figure 5.12 Model Update Configuration with Corrected Mapping

> The transformation strategies on the Process is set to *Continue*. This is the default transformation strategy. If we used the *Restart* strategy, the process would be restarted on update if on the element at the given moment.

To perform the model update, do the following:

1. Make sure your server with the Execution Engine, possibly on the PDS or SDK Embedded Server, is running and your PDS is connected to it.
2. Upload the model to the server and create a model instance of the old model: In the GO-BPMN Explorer, right-click the old module and go to Run As > Model.
3. Upload the new model to the server: In the GO-BPMN Explorer right-click the new module and go to Upload As > Model.
4. Switch to the Management perspective.
5. Refresh the Module Management and Model Instances view and check that both models are uploaded and the source model is instantiated.
6. Switch to the Management perspective.
7. Refresh the Module Management and Model Instances view and check that the old model is instantiated and the new model uploaded.
8. In the Model Instances view, open the detail of your old model and check the execution diagram of the process.

Figure 5.13 Execution Diagram of the Old Model Instance

9. Perform the update:

   (a) In the Model Instances view, click the Model Update ( ) button.

   (b) In the Model Update dialog window, provide the path to your muc file in the Configuration file field and click Next.

   (c) In the refreshed dialog, do not apply any filtering, just click Next so that any available instances of the old model are updated (in this case, exactly one model instance is running).

   (d) Check the summary of the model update and click Finish.

   (e) Refresh the Model Instances view: The model instance should be in the Updated status.

   (f) Click the Continue ( ) button to trigger the execution of the updated model instance.

10. Check the execution diagram of the updated model instance.

Figure 5.14 Execution Diagram of the Updated Model Instance

Note that the execution remains on the new Conditional Event.

Now set the transformation strategies on the process to `Restart` so the Process is restarted on update if on the event. Perform model update as described above: The execution remains on the new Conditional Start Event since the process instance was restarted.

5.4 Updating a Data Type

Required action: Update a model instance with a changed data type of a record property.

> **Important:** It is not recommended to update shared records via model update since changes on shared records are reflected on the database. While it is safe to add a new field to a shared record and remove not-nullable fields using model update, modifications to fields, such as modification of their data types, might result in corrupted database schema or data loss. It is recommended to migrate the database directly, not via update of shared records.

When you are updating a model instance to a model with changed non-shared record types, all record instances will be updated according to the transformation expression.

We will update a model instance's data hierarchy as follows:

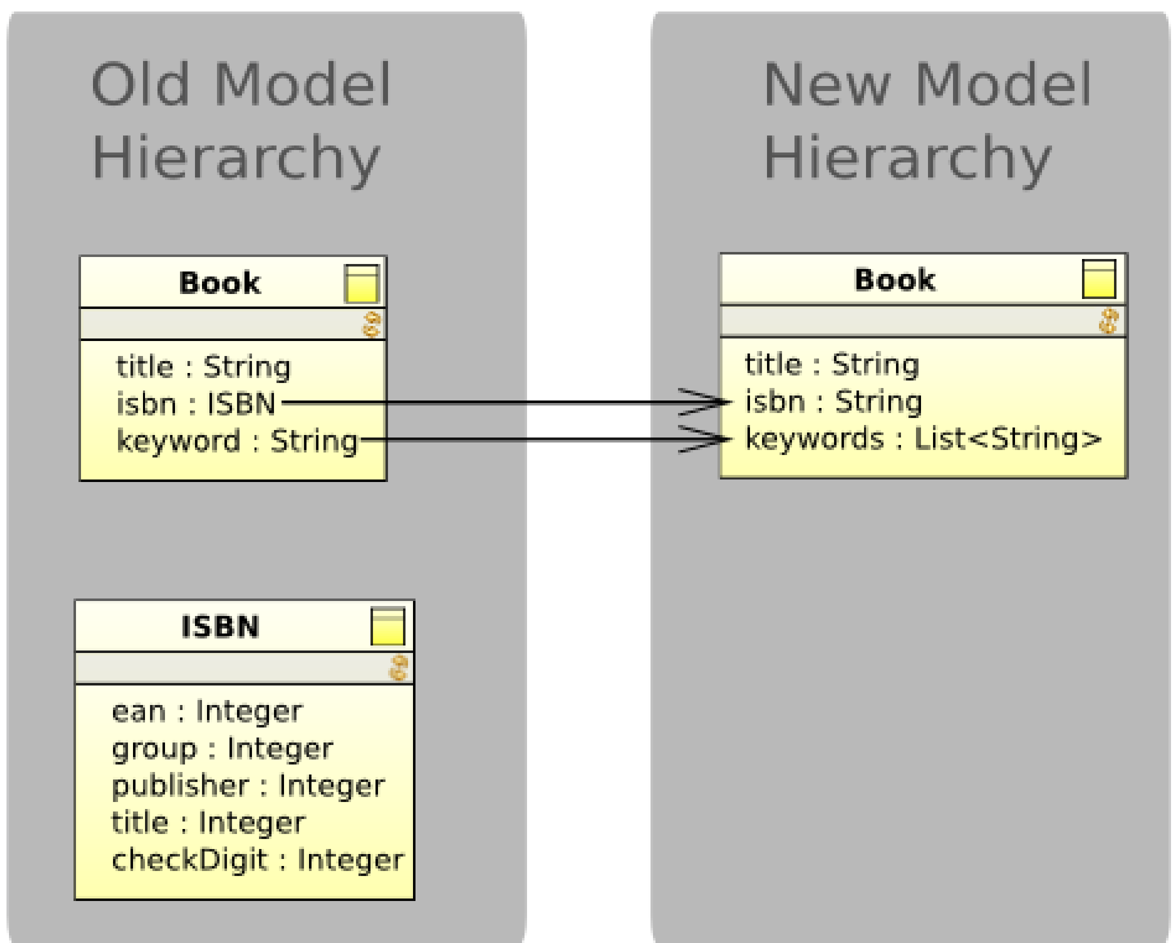


Figure 5.15 Old and new data type models

The data type update will involve the following changes:

- The ISBN record is removed: No further actions are required.
- The Book.isbn field is changed from the ISBN type to String: The new isbn field must concatenate and format the old instance of ISBN for the given Book instance.
- The keyword field is changed from a string to a list of strings and renamed to keywords: The new keywords field should import the old keyword string.

1. *Design the old model* with a process definition, variable definition, and data type definition as shown in Old model for data type update:
   (a) Create a module with the old data type hierarchy with the `Book` and `ISBN` records.
   (b) Create a global variable definition with a bookSet variable of the type `Set<Book>` and a book variable of the type `Book`.
   (c) Create a process definition with a None Start Event, a Conditional Flow Event, and a Simple End Event.
   (d) On the flow from the None Start Event define an assignment expression that creates three Book instances assigned to the bookSet global variable:

```
bookSet := {
  new Book(title -> "Brave New World",
           isbn -> new ISBN(ean -> 978, group -> 1, publisher -> 85399, title -> 393, che
           keyword -> "science fiction"),
```

```
        new Book(title -> "Catch-22",
                 isbn -> new ISBN(ean -> 978, group -> 1, publisher -> 4055, title -> 387, chec
                 keyword -> "army"),
        book := new Book(title -> "Brave New World",
                         isbn -> new ISBN(ean -> 978, group -> 1, publisher -> 85399, title ->
                         keyword -> "science fiction")
}
```

(e) Set the Condition parameter on the Conditional Flow Event to `false`. The Conditional Flow Event will hold the execution so that the process creates the record instances and then remains running.

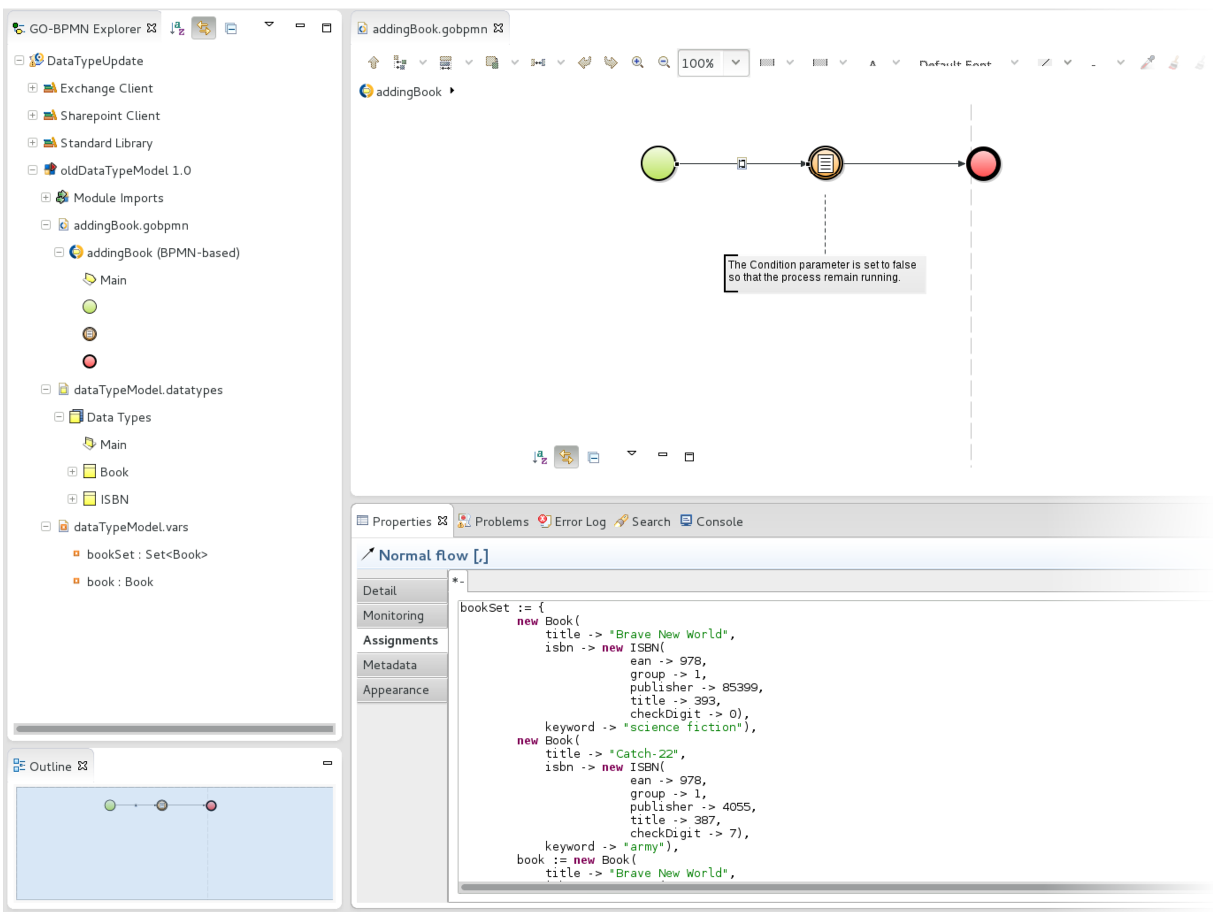


Figure 5.16 Old model

2. *Design the new model:* copy and paste the old model, modify the data type model, and remove the assignment expression on the flow.

3. *Create the .muc file*:

   (a) Right-click the parent project, go to New > Model Update Configuration and follow the wizard.

   (b) Open the .muc file and on the Data Types page and define the transformation expressions for the isbn record field and the new keywords field:

   - isbn:

     ```
     toString(old("isbn.ean")) + "-" +
     old("isbn.group") + "-" +
     old("isbn.publisher")  + "-" +
     old("isbn.title")  + "-" +
     old("isbn.checkDigit")
     ```

     This expression will take individual fields from the old record and concatenate them into a new hyphenated isbn value.

- keywords: `[old("keyword")]` This expression will take the keyword string and add it to a new list of keywords.

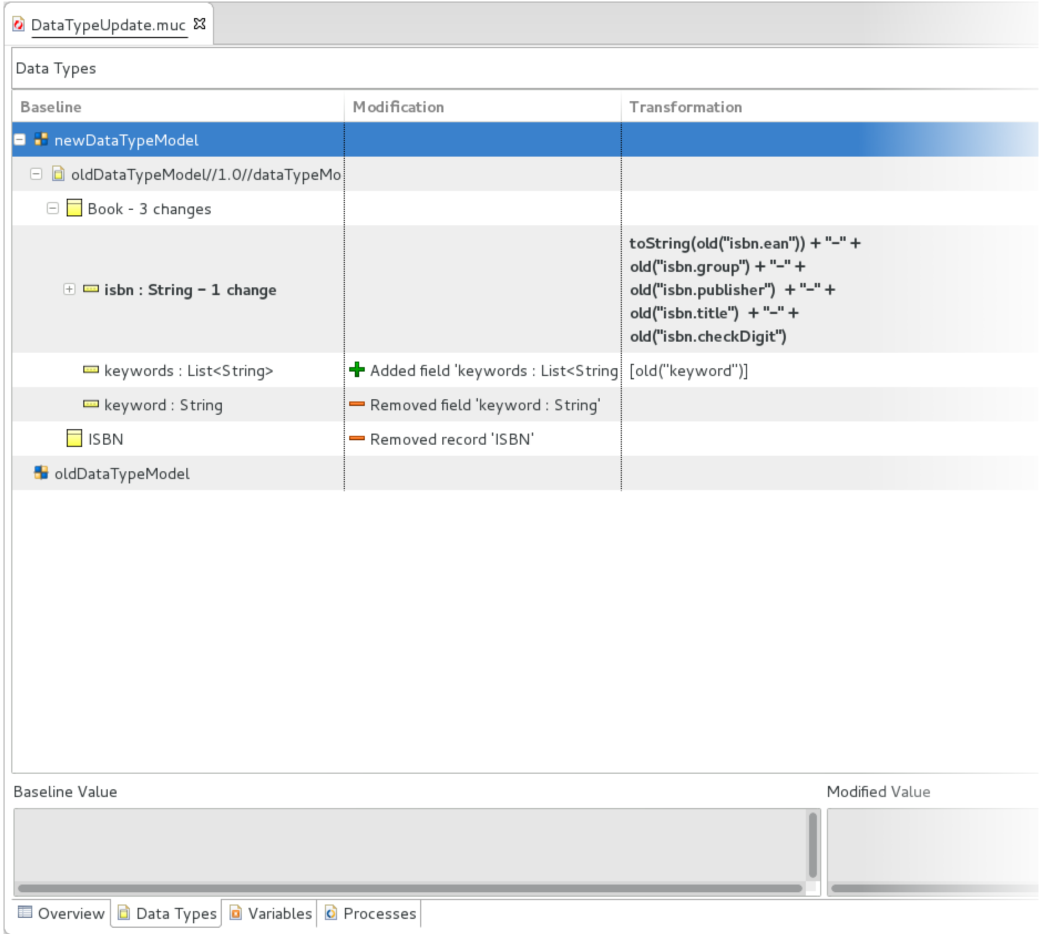


Figure 5.17 Model Update Configuration with the Data Type Changes

To perform the model update, do the following:

4. Make sure your server with the Execution Engine, possibly on the PDS or SDK Embedded Server, is running and your PDS is connected to it.

5. Upload the model to the server and create a model instance of the old model: In the GO-BPMN Explorer, right-click the old module and go to Run As > Model.

6. Upload the new model to the server: In the GO-BPMN Explorer right-click the new module and go to Upload As > Model.

7. Switch to the Management perspective.

8. Refresh the Module Management and Model Instances view and check that both models are uploaded and the source model is instantiated.

9. In the Model Instances view, open the detail of your old model and check the execution diagram of the process.

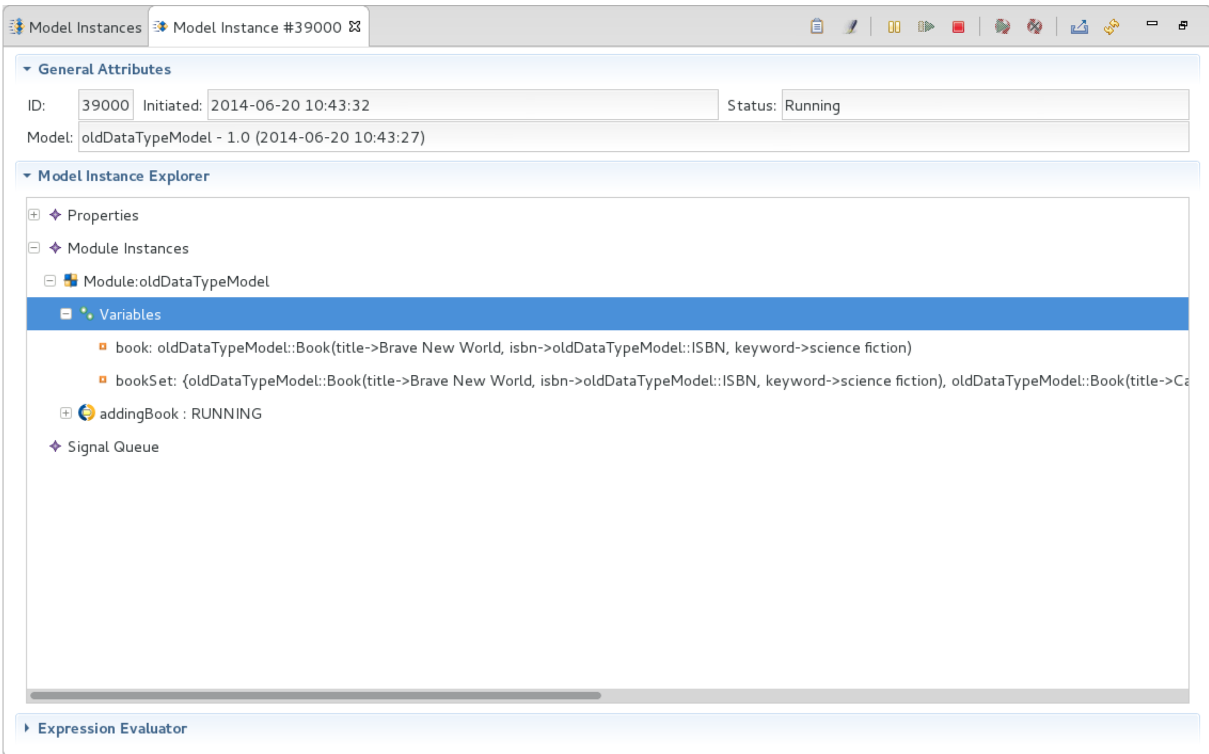


Figure 5.18 Detail of the Old Model with Old Global Variables

10. Perform the update:

    (a) In the Model Instances view, click the Model Update ( ) button.

    (b) In the Model Update dialog window, provide the path to your muc file in the Configuration file field and click Next.

    (c) In the refreshed dialog, do not apply any filtering, just click Next so that any available instances of the old model are updated (in this case, exactly one model instance is running).

    (d) Check the summary of the model update and click Finish.

    (e) Refresh the Model Instances view: The model instance should be in the Updated status.

    (f) Select the model instance and click Resume ( ). The model instance becomes Running.

11. Check the Log view for the log message of the post-process.

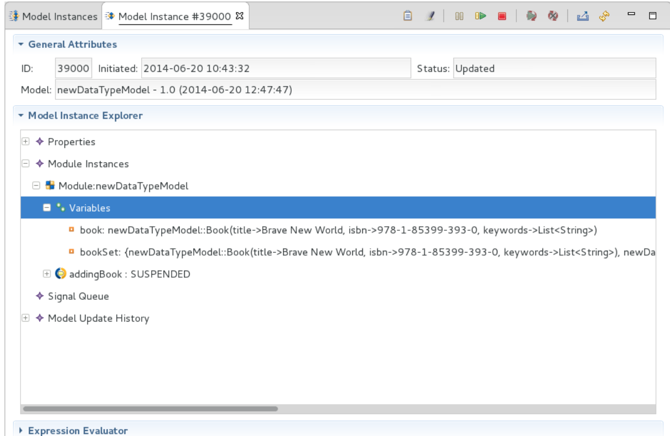


Figure 5.19 Detail View of the Updated Model Instance with New Global Variables

The variables hold values of the new data types: the record values were transformed according to the transformation expression defined for the data types.