Living Systems® Process Suite

Modeling

Living Systems Process Suite Documentation

3.6
Mon Nov 1 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 Main Page **1**

1.1 Running Designer . . . 2

2 Model Design **5**

2.1 Model Structure . . . 6

2.1.1 GO-BPMN Projects . . . 6

2.1.1.1 Creating GO-BPMN Projects . . . 7

2.1.1.2 Closing GO-BPMN Projects . . . 7

2.1.1.3 Referencing Projects . . . 8

2.1.2 GO-BPMN Modules . . . 8

2.1.2.1 Creating GO-BPMN Modules . . . 8

2.1.2.2 Importing Modules . . . 11

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 . . . 17

2.2.2 Defining Properties . . . 18

2.2.3 Disabling Elements . . . 20

2.2.3.1 Defining Formatting of Disabled Elements . . . 23

2.2.4 Working with Diagrams . . . 23

2.2.4.1 Creating a Diagram . . . 25

2.2.4.2 Inserting Element Views . . . 25

2.2.4.3 Aligning Element Views 26
2.2.4.4 Matching Element Views Size 27
2.2.4.5 Spreading Diagram Element Views 27
2.2.4.6 Changing the Element Type 27
2.2.4.7 Deleting Elements from Diagrams 28
2.2.4.8 Snapping to Grid 28
2.2.4.9 Resetting the Anchor Position of a Line Element 28
2.2.4.10 Customizing the Palette 29
2.2.4.11 Displaying and Hiding Page Borders in Diagrams 30
2.2.4.12 Limiting Diagram Frame Nesting Level 31
2.2.4.13 Inserting Hyperlinks 31
2.2.4.14 Switching Iconic and Decorative Notation 32
2.2.4.15 Hiding and Displaying Compartments of Diagram Elements 33
2.2.4.16 Changing Line Style 33
2.2.4.17 Formatting Settings 35
2.2.4.18 Diagram Printing 39
2.2.4.19 Applying Automatic Layout in Diagrams 43
2.2.5 Comparing Resources 44
2.2.5.1 Comparing Two Resources 44
2.2.5.2 Comparing Three Resources 46
2.2.5.3 Comparing Version-Controlled Resources 48
2.2.5.4 Merging 48
2.2.6 Modeling Status 48
2.2.6.1 Setting a Modeling Status of an Element 49
2.2.6.2 Defining a Modeling Status 49
2.2.6.3 Defining Modeling Status for a Project 50
2.2.6.4 Exporting and Importing a Modeling Status 50
2.2.6.5 Disabling Presentation of a Modeling Status 50
2.2.7 Todo and Task Markers 51
2.2.8 Validation 53

2.2.8.1 Configuring Validation 54
2.2.8.2 Validating Old Modules 55
2.2.8.3 Hiding Validation Markers 56
2.2.9 Copying Fully Qualified Names of Elements 56
2.2.10 Search 56
2.2.10.1 Searching for GO-BPMN Entities and their Usage 57
2.2.10.2 Searching for Elements 58
2.2.10.3 Searching for Element Usages 58
2.2.10.4 Searching for Dependent Tasks 59
2.2.10.5 Searching for Call Hierarchies 59
2.2.10.6 Searching for Unused Elements 59
2.2.10.7 Searching for Modeling Elements by ID 60
2.2.11 Solving Errors 61
2.3 Processes 61
2.3.1 Modeling a Process 61
2.3.2 Defining Element Labels 62
2.3.3 Defining Process Parameters 63
2.3.4 Defining Process Variables 64
2.3.5 Modeling a Process 64
2.3.5.1 Changing Task Type 65
2.3.5.2 Removing Invalid Task Parameters 65
2.3.6 Reusing a Process 65
2.3.6.1 Extracting Process Elements into a Reusable Sub-Process 65
2.4 Variables 65
2.4.1 Global Variable 66
2.4.2 Process, Forms, and Sub-Process Variable 67
2.4.3 Local Variable 67
2.5 Organization Models 67
2.5.1 Creating an Organization Model 67
2.5.2 Assigning a Role or Organization Unit to a Person 69

2.6 Documents . . . 69
2.6.1 Defining a Document . . . 70
2.6.2 Navigate Away from Document . . . 71
2.6.2.1 Creating a Model Instance and Navigating to its To-Do on Document Submit . . 72
2.7 Forms . . . 73
2.8 Localization . . . 73
2.8.1 Creating Localization Identifiers in the Localization Editor . . . 73
2.8.2 Creating and Calling Localization Identifier in the Expression Editor . . . 74
2.8.3 Calling Localization Identifiers . . . 75
2.8.4 Searching for Usages of Localization Identifiers . . . 75
2.8.5 Identifying Unlocalized Strings . . . 76
2.9 Function . . . 76
2.9.1 Calling a Function . . . 76
2.10 Queries . . . 76
2.10.1 Defining Queries . . . 76
2.10.2 Calling Queries . . . 77
2.10.3 Standard Queries . . . 77
2.10.3.1 Filtering Results in Standard Queries . . . 78
2.10.3.2 Ordering in Standard Queries . . . 79
2.10.3.3 Generating Queries for Shared Records . . . 82
2.10.4 HQL Queries . . . 82
2.10.5 Native Queries . . . 84
2.11 Data Type Model . . . 85
2.11.1 Creating a Record . . . 85
2.11.2 Creating a Record Field . . . 86
2.11.3 Defining a Method of a Record . . . 88
2.11.4 Defining a Subtype of a Record . . . 89
2.11.5 Creating a Relationship Between Records . . . 89
2.11.5.1 Setting Fetching . . . 91
2.11.6 Using Records from Other Definitions . . . 91

2.11.7 XML Mapping in a Data Type Model 92
2.11.8 Creating a Record Composition 92
2.11.9 Creating an Interface 93
2.11.10 Comparing Records 94
2.11.10.1 Defining Fields for Record Comparisons 94
2.11.10.2 Comparing Records with Fields on Related Records 95
2.11.11 Presentation of Record Diagrams 95
2.11.11.1 Displaying and Hiding Record Compartments 95
2.11.11.2 Viewing Record Hierarchy 96
2.11.12 Importing Data Types from an XSD File 97
2.11.13 Record Validation 97
2.11.13.1 Validating a Record 97
2.11.13.2 Defining a Constraint 98
2.11.13.3 Defining a Constraint Type 100
2.11.13.4 Filtering Constraints on Validation using Tags 100
2.12 Persistent Data 101
2.12.1 Defining Data Model Properties 101
2.12.1.1 Extracting Affixes from Data Model 102
2.12.2 Generating a Data Model from a Database Schema 103
2.12.3 Creating a Shared Record 104
2.12.4 Creating a Shared Record Field 105
2.12.5 Locking Shared Records for Changes (Optimistic Locking) 106
2.12.6 Setting up Optimistic Locking on a Shared Record 106
2.12.7 Setting up Relationships Between Shared Records 107
2.12.7.1 Enforcing Not Null Foreign Key to Related Records 108
2.12.7.2 Defining Indexes 109
2.12.7.3 Defining a Shared Field with a Foreign Key of a Related Record 110
2.12.8 Auditing: Shared Record Versioning 111
2.12.8.1 Setting up Auditing 112
2.12.8.2 Auditing a Shared Record 113

2.12.8.3 Excluding a Shared Field or Record from Auditing 114

2.12.8.4 Including and Excluding a Relationship End from Auditing 115

2.12.9 Caching a Shared Record 115

2.12.9.1 Defining Cache Regions 116

2.12.9.2 Disabling Cache Regions 116

2.12.10 Change Proxy: Transient Data of Shared Records 116

2.12.10.1 Creating a Proxy 117

2.12.10.2 Creating a Proxy of a Related Record 118

2.12.10.3 Merging a Proxy Set 119

2.12.10.4 Deleting a Record Using a Proxy 120

2.12.10.5 Checking if an Object is a Change Proxy 120

2.12.10.6 Checking the Proxy Level of a Change Proxy 120

2.13 Constants 121

2.14 Decision Tables 121

2.14.1 Designing a Decision Table 122

2.14.1.1 SFEEL 125

2.14.1.2 Merging and Splitting Cells 127

2.14.1.3 Copying Rules 127

2.14.2 Creating, Saving and Loading a Decision Table 128

2.14.3 Evaluating a Rule with a Decision Table 129

2.15 Webservice Processes 129

2.15.1 REST Webservice Processes 129

2.15.1.1 Creating a REST Server Process 130

2.15.1.2 Creating REST Client Process 132

2.15.2 SOAP Webservice Processes 132

2.15.2.1 SOAP Server Process 133

2.15.2.2 SOAP Client Process 137

2.16 Jasper Reports Integration 141

2.16.1 Integrating Application User Interface with a JasperServer 142

2.16.2 Enabling Expression Language in Jasper Reports 142

2.16.3 Displaying a Jasper Report . . . 143
2.16.3.1 Mapping of Parameter Data Types . . . 143
2.16.3.2 Exporting a Jasper Report . . . 143
2.17 Sharing Resources . . . 144
2.17.1 Module Export . . . 144
2.17.1.1 Export Configurations . . . 145
2.17.1.2 Exporting a Module with General Export . . . 145
2.17.1.3 Exporting a Module with GO-BPMN Export . . . 147
2.17.1.4 Exporting a Module to XPDL . . . 149
2.17.2 Resource Import . . . 150
2.17.2.1 Importing Model Packages to Workspace . . . 151
2.17.2.2 Importing XPDL . . . 152
2.17.2.3 Importing File System Structure . . . 152
2.17.3 Libraries . . . 152
2.17.3.1 SharePoint and Exchange Client Libraries . . . 153
2.17.3.2 Importing a Library to GO-BPMN Projects . . . 153
2.17.3.3 Creating Libraries . . . 153
2.17.3.4 Removing a Library from a Project . . . 154
2.18 Presenting a Model . . . 154

3 Model Execution 155

3.1 Server Connection from Designer . . . 155
3.1.1 Connecting Designer to an LSPS Server . . . 156
3.1.2 Connecting Designer to an LSPS Server using a Proxy Server . . . 157
3.1.3 Connecting Designer to Designer Embedded Server . . . 157
3.1.4 Loading and Resetting Database of Designer Embedded Server . . . 158
3.1.5 Configuring Mail Server on Designer Embedded Server . . . 159
3.2 Model Upload . . . 159
3.2.1 Uploading a Module from the Modeling Perspective . . . 159
3.2.2 Uploading a Module from the Management Perspective . . . 160
3.3 Model Instantiation . . . 161

3.3.1 Creating a Model Instance from the Management Perspective 161
3.3.2 Creating a Model Instance from the Modeling Perspective 162
3.4 Debugger 163
3.4.1 Debugger Implementation 164
3.4.2 Setting up Debugger 165
3.4.2.1 Debugger on Designer Embedded Server 165
3.4.2.2 Debugger on SDK Embedded Server 167
3.4.2.3 Debugger on a Remote Server 169
3.4.3 Debugging a Model 171
3.4.3.1 Adding and Removing Breakpoints 173
3.4.3.2 Defining a Breakpoint Condition 174
3.4.3.3 Resuming Breakpoints 174
3.4.3.4 Enabling and Disabling Breakpoints 174
3.5 Profiling Tools 175
3.5.1 Profiling with LSPS Profiler 175
3.5.2 Profiling with Trace Logger 177

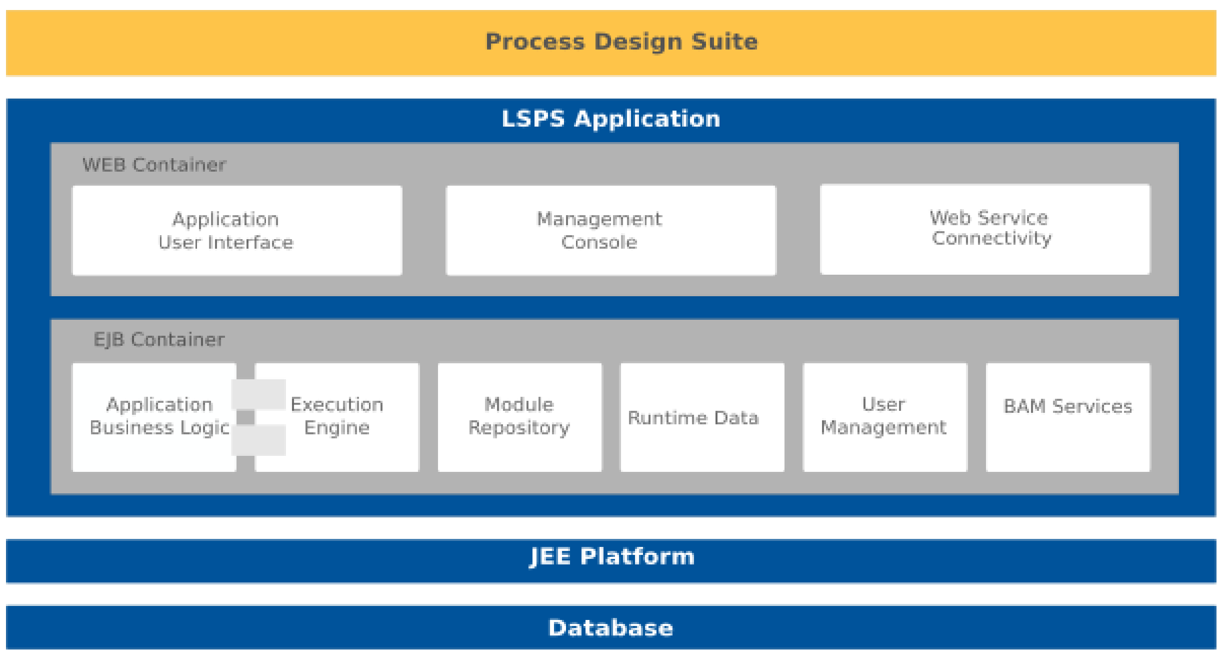
4 Model Update 179

4.1 Model-Update Processes 180
4.2 Transformation 181
4.2.1 Record Transformation 182
4.2.2 Variable Transformation 183
4.2.3 Asynchronous Modeling Elements Transformation 183
4.2.3.1 Transformation Strategies 183
4.3 Model Update Configuration 184
4.3.1 Creating a Model Update Configuration 185
4.3.1.1 Copying Model Update Data 186
4.3.2 Editing Model Update Configuration 186
4.3.2.1 Displaying Matching Data in Model Update Configuration 187
4.3.2.2 Changing Element Mapping in Model Update Configuration 188
4.3.2.3 Changing Model Update Settings 188
4.3.2.4 Defining Transformation 189
4.3.2.5 Creating a Model Update Process 192

Chapter 1

Main Page

Models and related resources for the Living Systems® Server, are created in an integrated development environment called Living Systems® **Designer** . The environment gives you access to tools for modeling of business processes, communication with LSPS Servers, development of your custom application, etc.



Note: If you install the LSPS Repository, you can generate the LSPS Server EAR with the 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 Designer 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 Designer

After installation, run Designer:

1. Go to <LSPS_HOME>.
2. Run the `lsps-design` binary for your platform.
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.

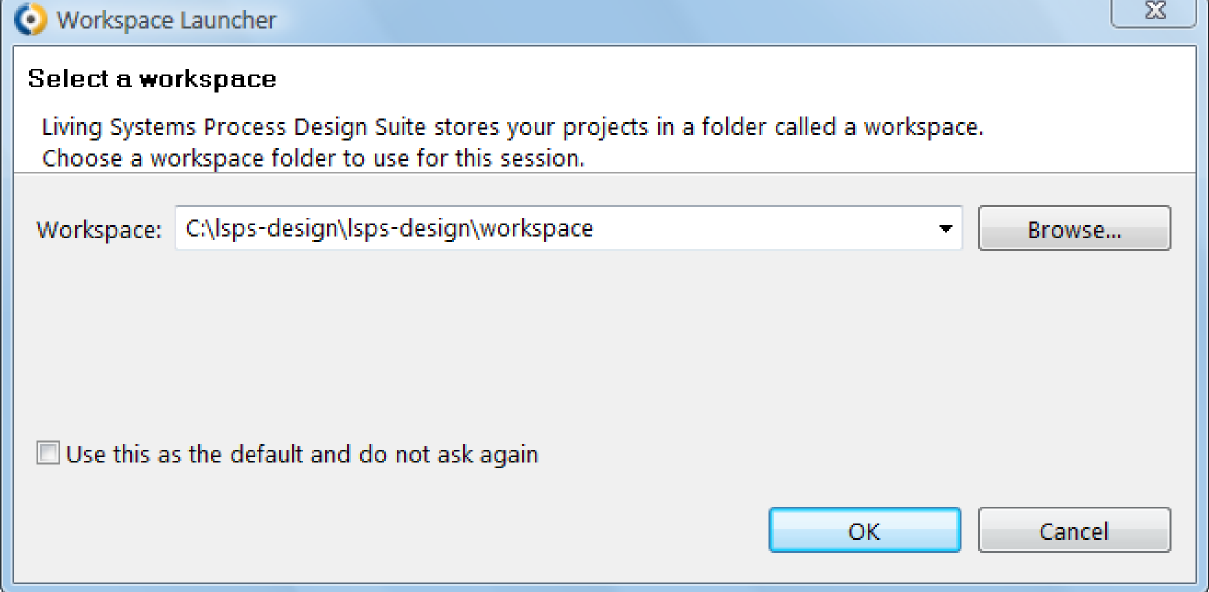


Figure 1.1 Workspace Launcher dialog

If you are starting Designer in a new workspace, Designer displays the Welcome page with links to perspectives, documents, and other Designer 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 Designer perspective: go to Window > Open Perspective > Other and in the Open Perspective dialog box, click a Designer perspective.

Chapter 2

Model Design

When designing your models and related resources for the Living Systems® Process Suite, you will do so in the integrated development environment Living Systems® Designer , 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 with 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,
- persistent data types,
- constants,
- decision tables.

To integrate you models with external systems, you can use

- support for web services in your processes.
- support for integration with Jasper reports.

It is recommended to use a version control system, such as *git* to store your models: Make sure to track `.project` directories of your models.

Note that you can 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. Generally, one project should be sufficient.
- Modules hold the definition files with model resources used for execution, such as, processes, data types, variables, etc. and are deployed to the server.

Consider keeping your model apart from other resources of your project. When planning your model structure, consider the `recommended best practise`

Note: We often use the term *model* and *model instance*: a model is the object that you can run on the server, run meaning you can create its model instances: It holds the static data serves as a blueprint for the model instances, while model instances hold the runtime data, for example, a model holds a process with an Activity and definitions of global variables, while its model instance will hold the data on whether the Activity is running and the current value of the variable. On design time, the model is an executable module with all its resources including any imported modules.

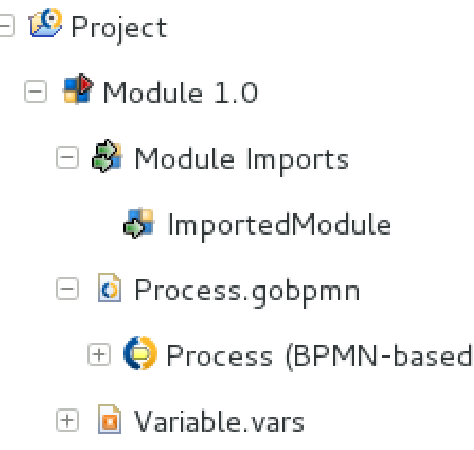


Figure 2.1 GO-BPMN Project Structure

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 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.

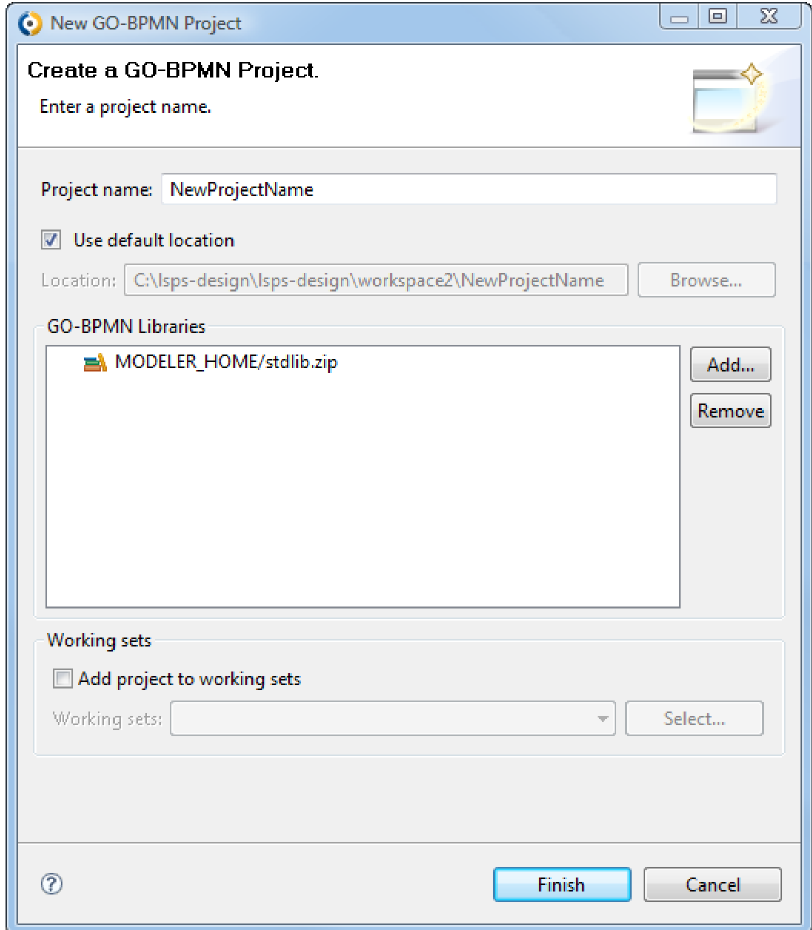


Figure 2.2 Creating a new GO-BPMN project

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.

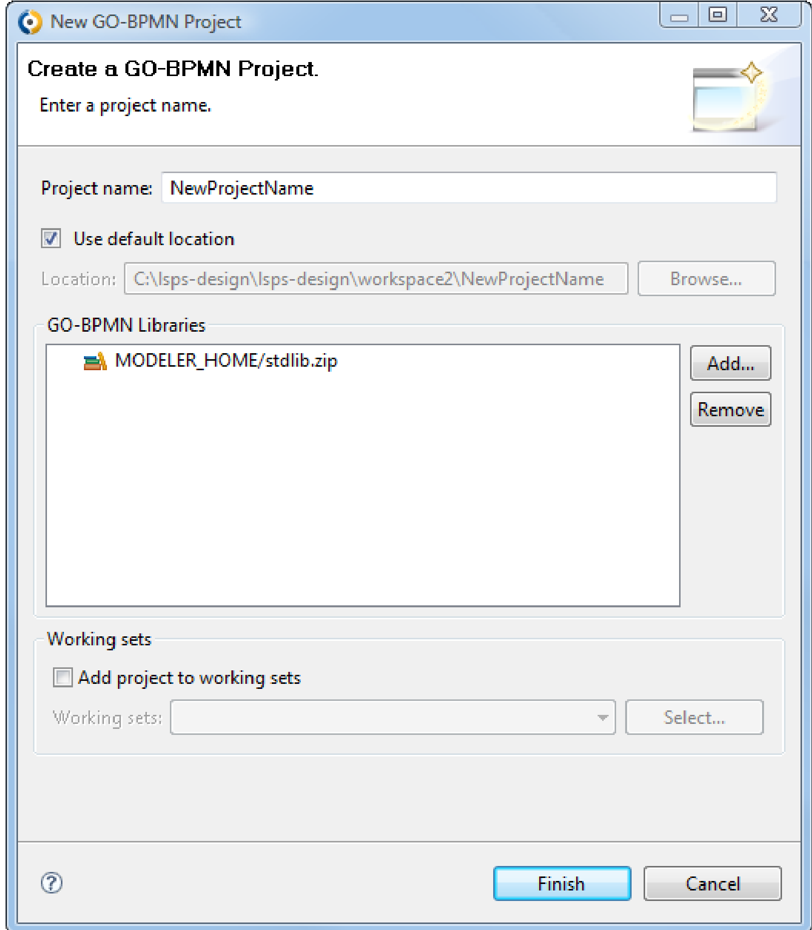


Figure 2.3 New GO-BPMN Module dialog box

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: designates the version of the module
     The user bumps the version to indicate that they've updated the module. The module preserves the same name, only the version number changes. Such module versions can coexist in the server Module repository: their organization and data type models are unified so as to preserve the data and role assignments. If you assigned a role from version 1.0 to a person, the person has the same role also in version 2.0 of the module: the role models are not distinguished by the version and are not removed, only added, when a newer version of the module is created. This is true for persisted data, that is, data based on shared records and their relationships, as well.

```
Other resources, such as documents, variables, etc., which are part of module instance contexts
```

are specific to the module in the given version.

As a result, different versions of a module are not expected in the same workspace and, if required, they must be located in different GO-BPMN projects to prevent name clashes.

- 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)

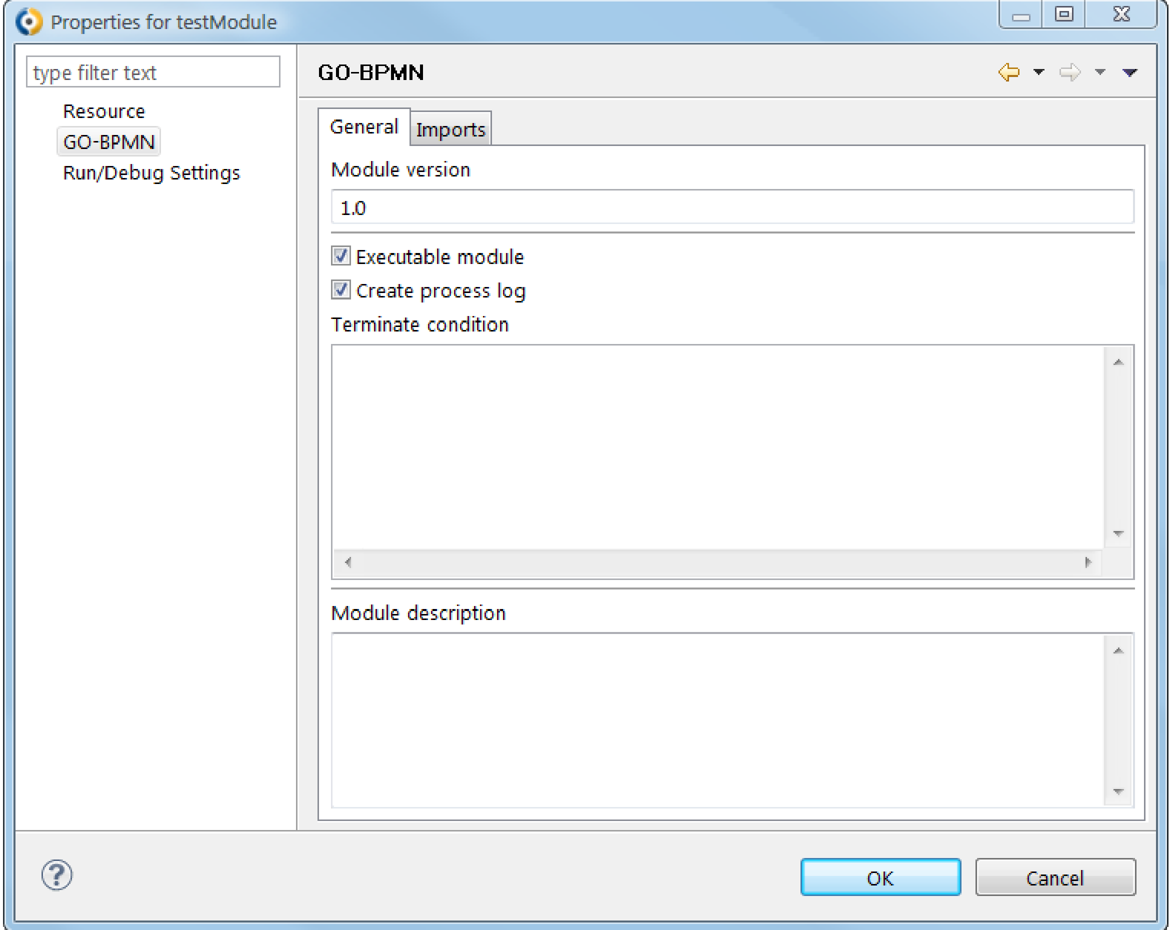


Figure 2.4 General tab of the Properties dialog box

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.

Note that imported executable Modules are instantiated along with their importing executable Module: The system creates a model instance with 2 or more Module contexts.

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.

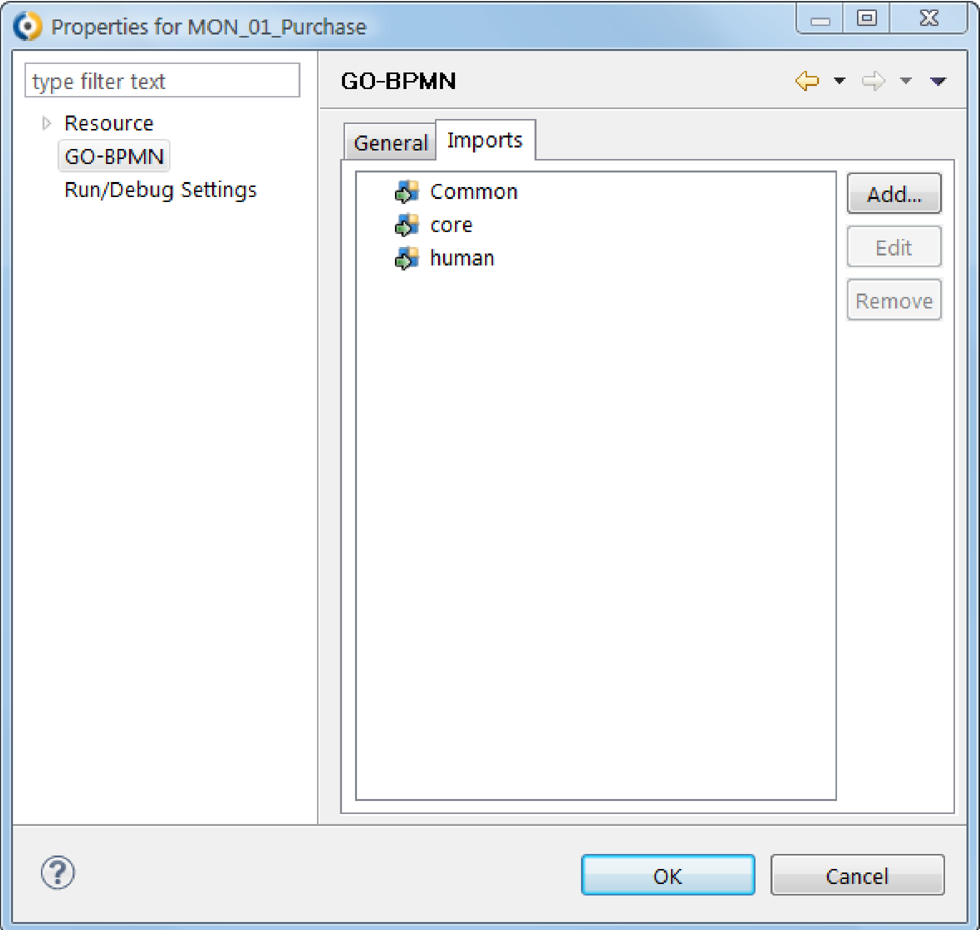


Figure 2.5 Imports tab with a list of imported modules

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 complex and difficult to follow in the GO-BPMN Explorer. To view such relationships in a comprehensive way, use the *Module Dependency View* with a diagram of projects and modules and their relationships.

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

To show transitive imports in the graph, unselect **Show Transitive Reduction** .

You can display dependencies of a particular module by typing its name or its part in the *Display dependencies of module* field.

To include projects in the graph, select the **Show Projects** button in the view menu.

To select the module in the GO-BPMN Explorer, double-click it.

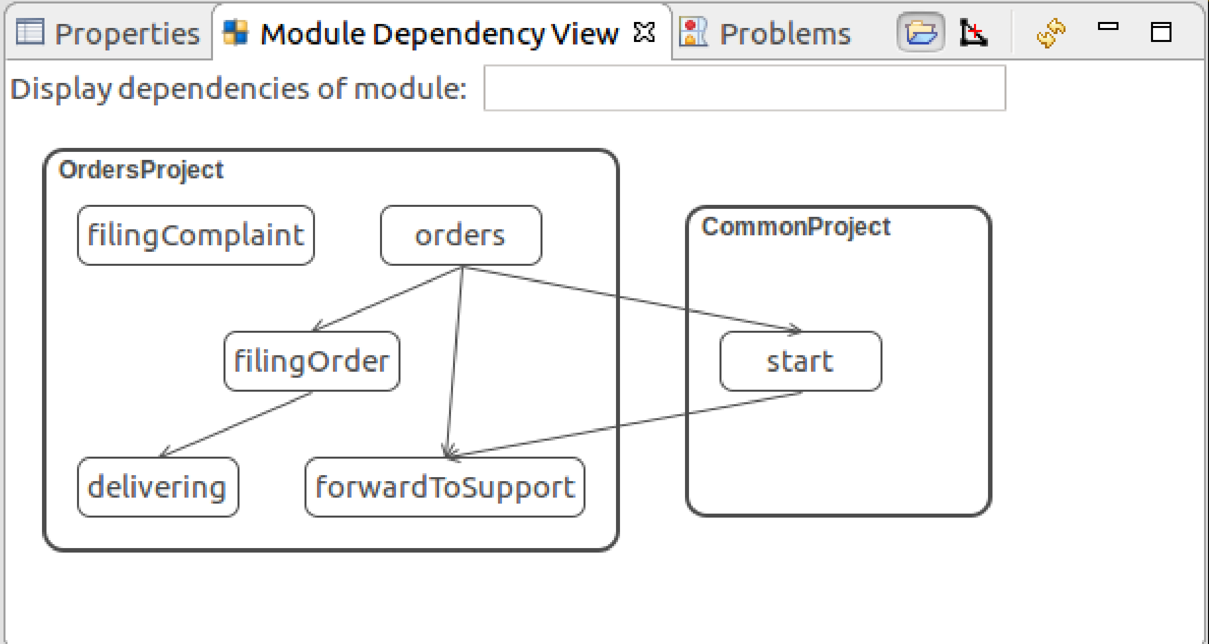


Figure 2.6 Module Dependency View with transitive reduction activated

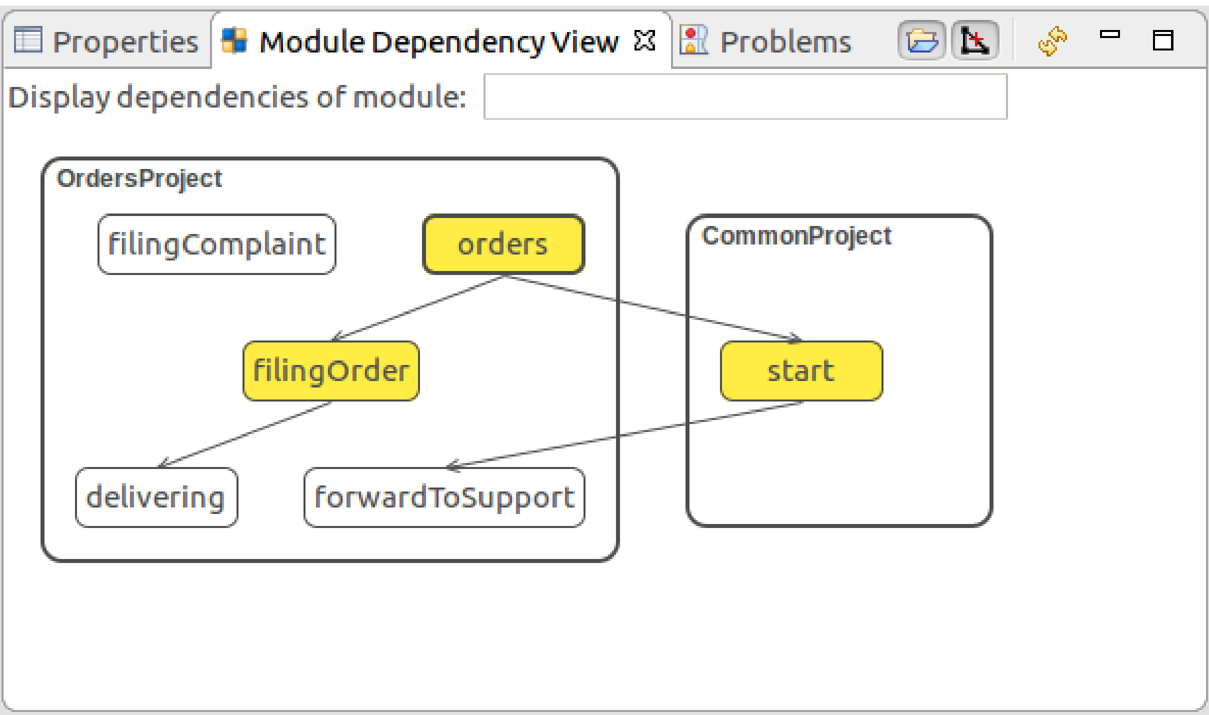


Figure 2.7 Module Dependency View with transitive reduction deactivated

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.

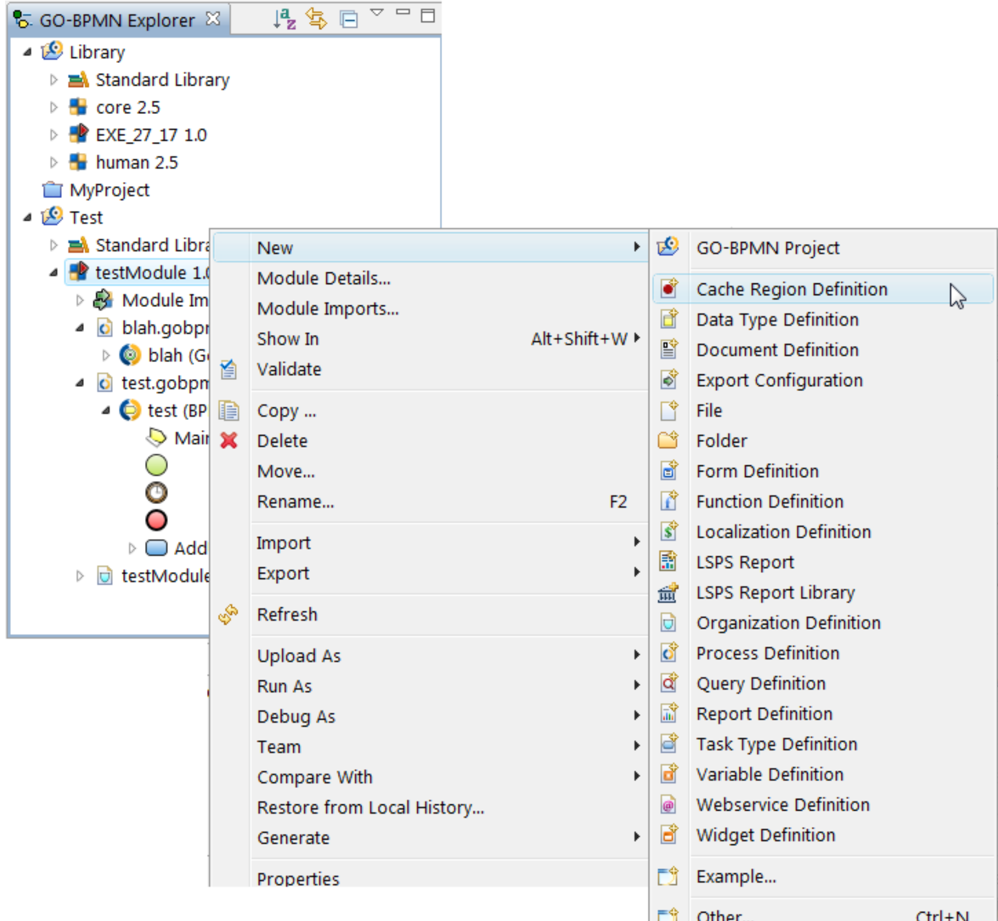


Figure 2.8 Creating a definition file

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 Designer 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.

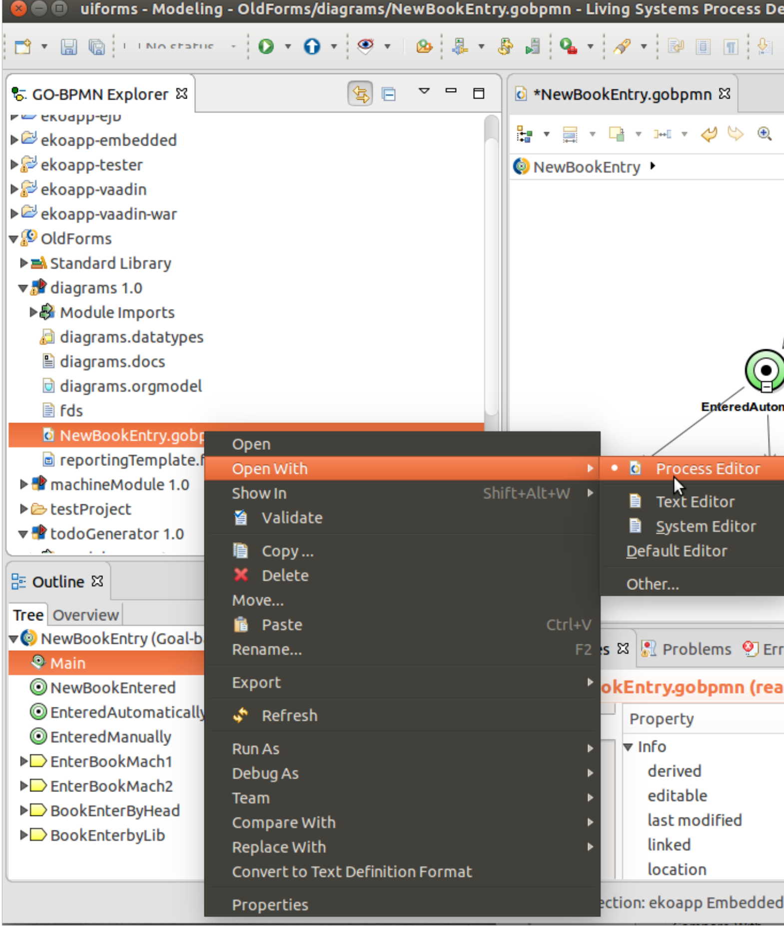


Figure 2.9 Selecting an editor

2.2 Generic Modeling Mechanisms

These are features and mechanisms that apply to the entire environment or certain types of 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
- Copying a Fully Qualified Name
- Searching for elements
- Identifying environment errors

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**.

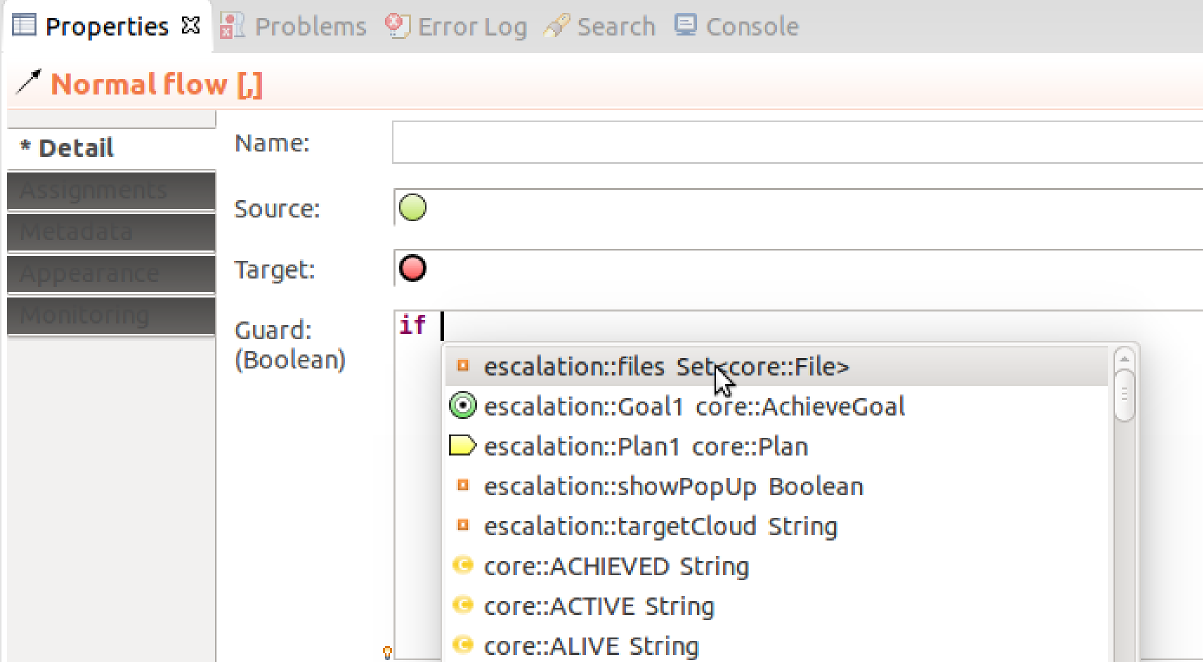


Figure 2.10 Content Assist menu

- 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**.

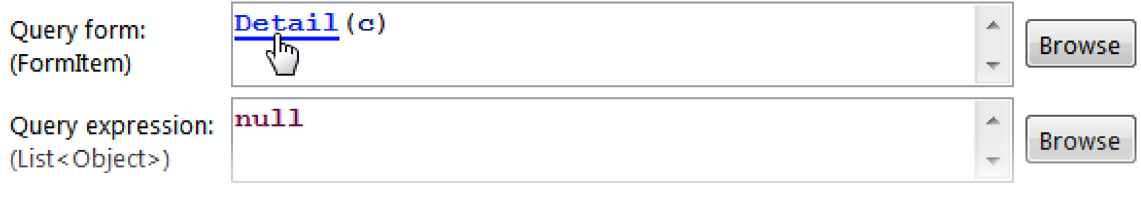


Figure 2.11 Link to definition

Hyperlinking is available also for the data type of the properties.

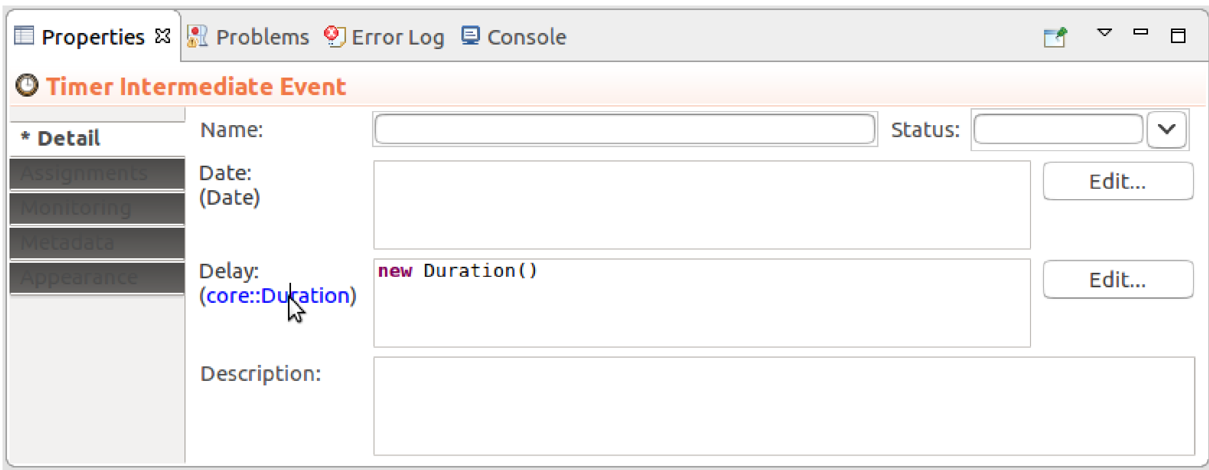


Figure 2.12 Linking to the type Duration

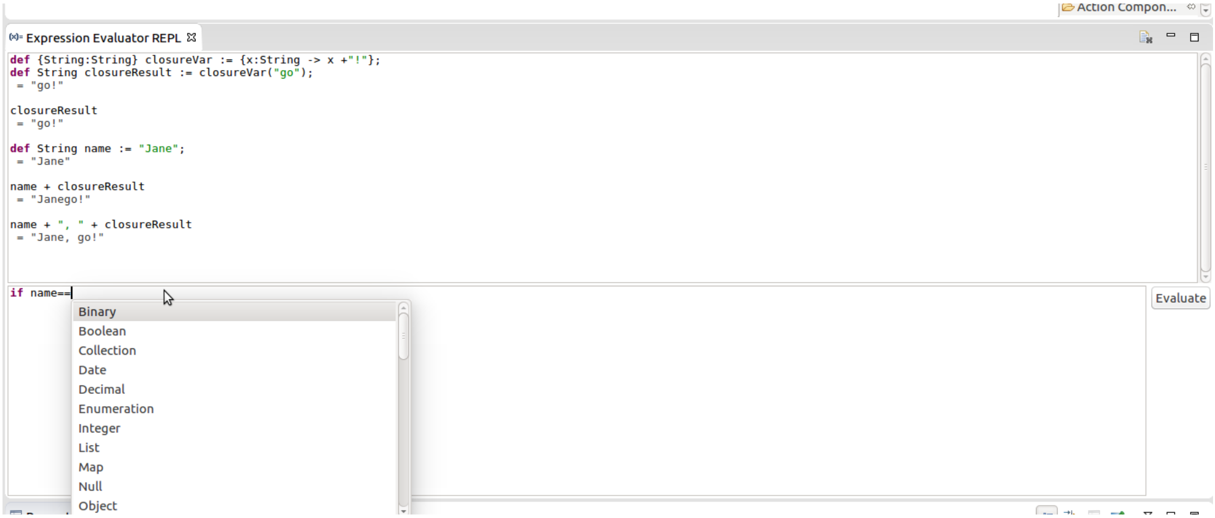
2.2.1.1 Testing Expressions

You can test expressions in the *Expression Evaluator REPL* view. Note that REPL uses UTC as its timezone.

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).

You can list the history of expression by pressing the arrow-up key and display auto-completion options by pressing *CTRL + Space*.



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 in the Management perspective of Management Console.

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 properties:

- *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.

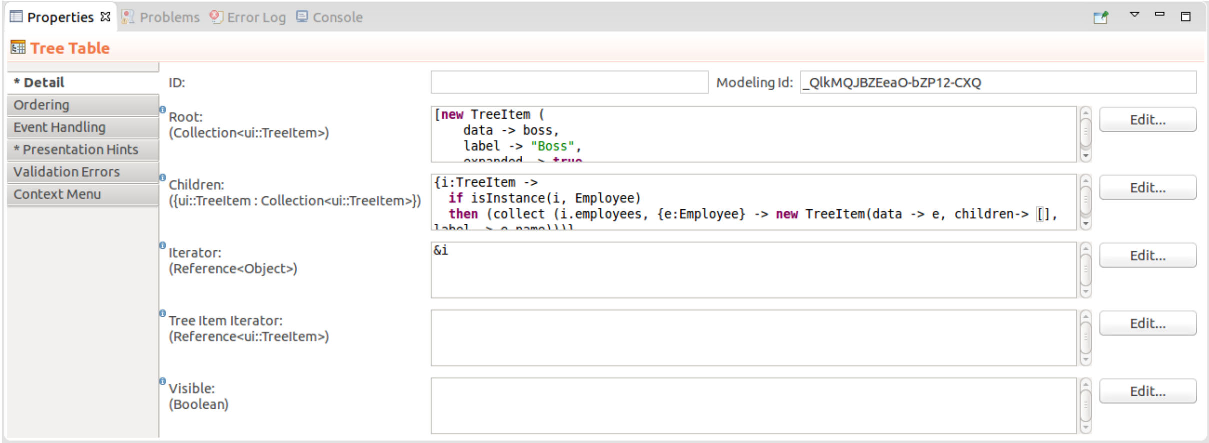
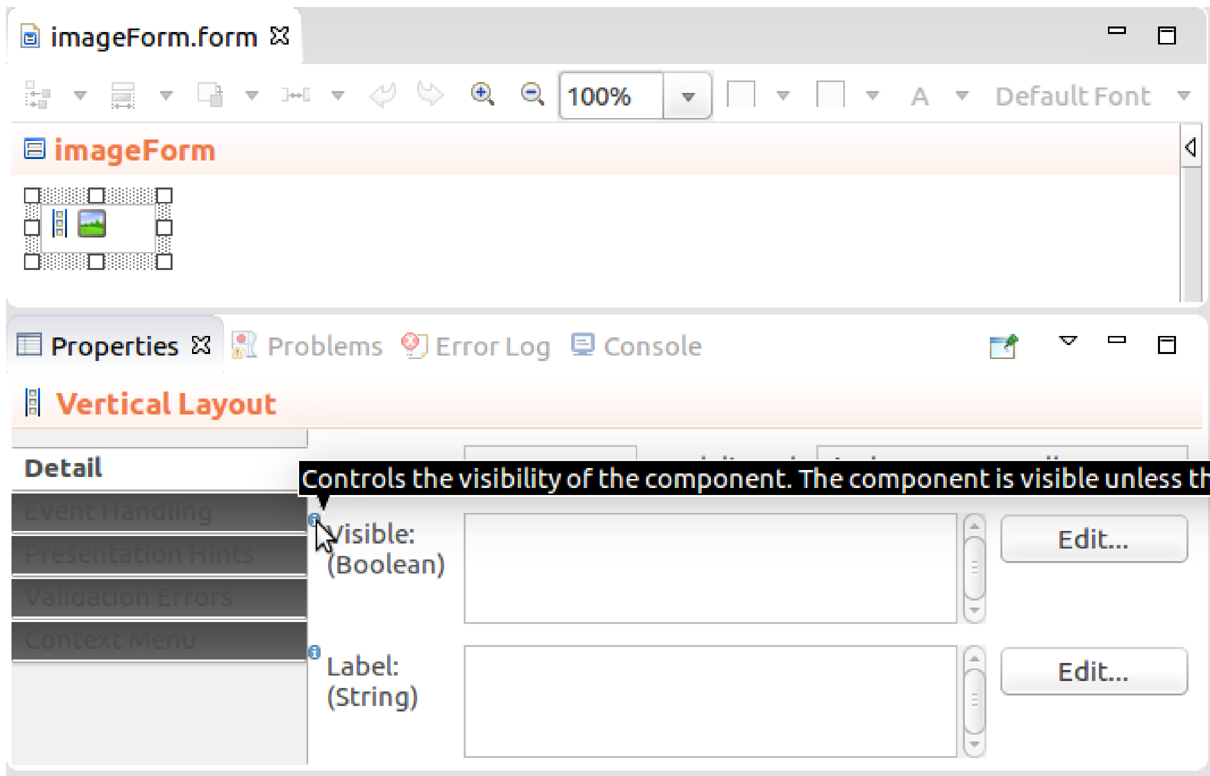


Figure 2.13 Properties view of a form component

Some properties have a tooltip with useful information.



If you press **Ctrl+left-click** over a data types in properties, the definition of the type will be displayed.

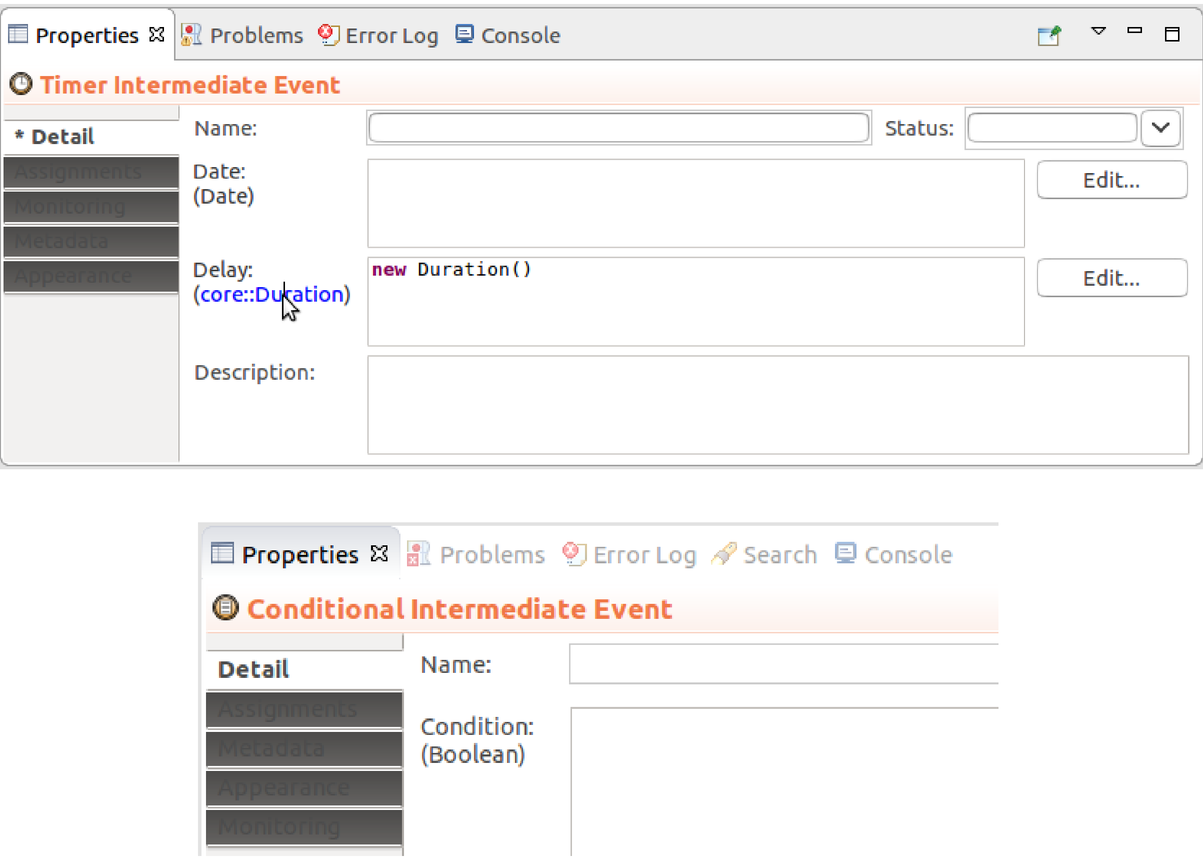


Figure 2.14 Condition on a Conditional Intermediate Event element

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

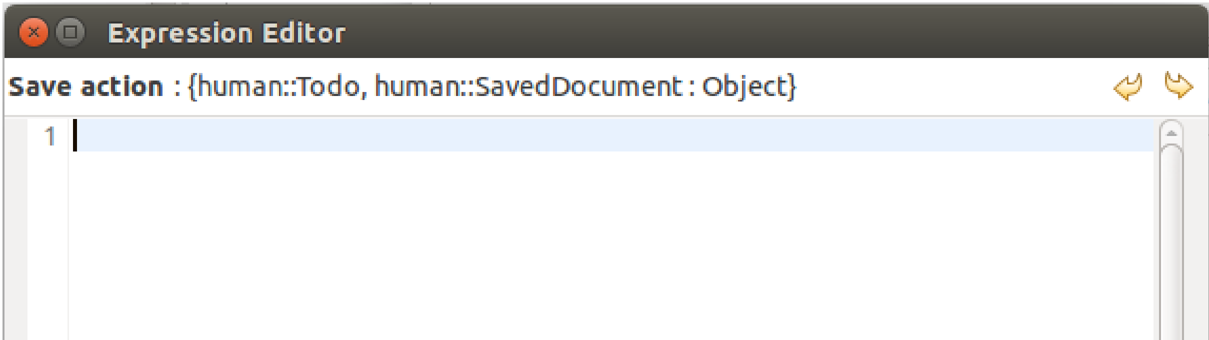


Figure 2.15 Editing Save action property

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, function, method, 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, record types and functions, must be disabled manually.

You can disable elements using their context menu or, in the case of textual element, such as, methods and textual function declarations, with the `@Disabled` annotation.

Disabling all methods in a method definition file

```
@Disabled
MyRecord {
~
 public String getTitle(){
  this.title;
 }
  public String getDate(){
  this.date;
 }
}
```

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.

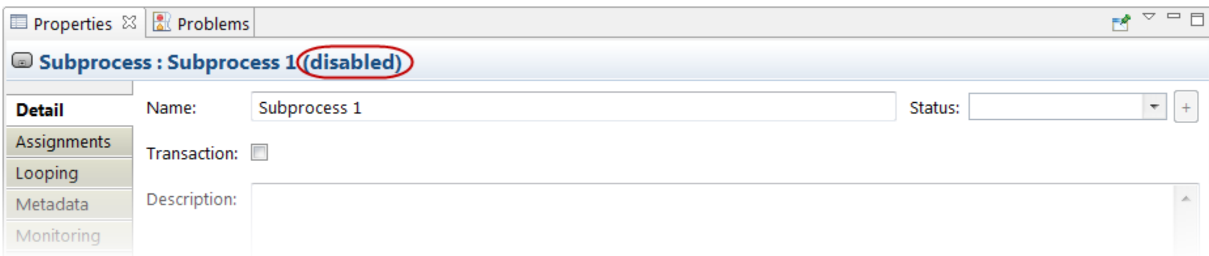


Figure 2.16 Disabled elements in the Properties view

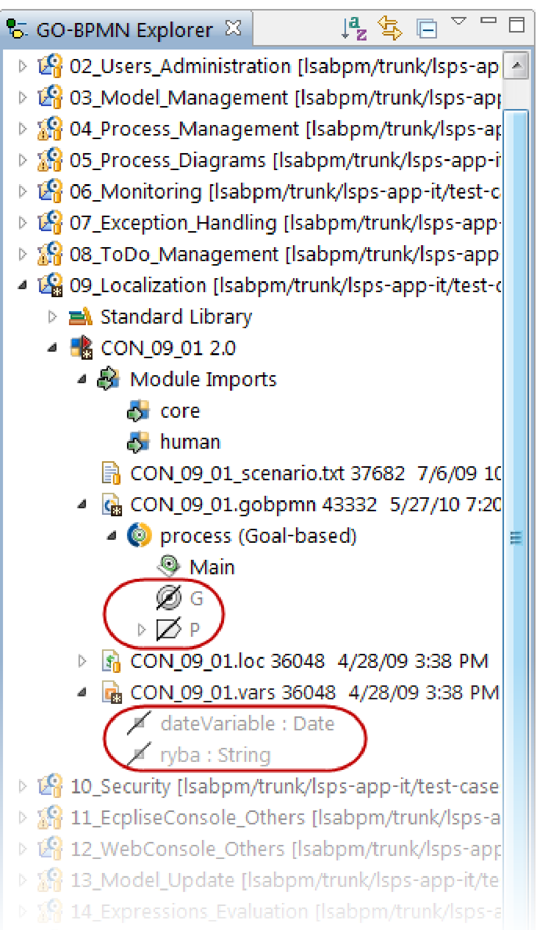


Figure 2.17 Disabled elements in GO-BPMN Explorer

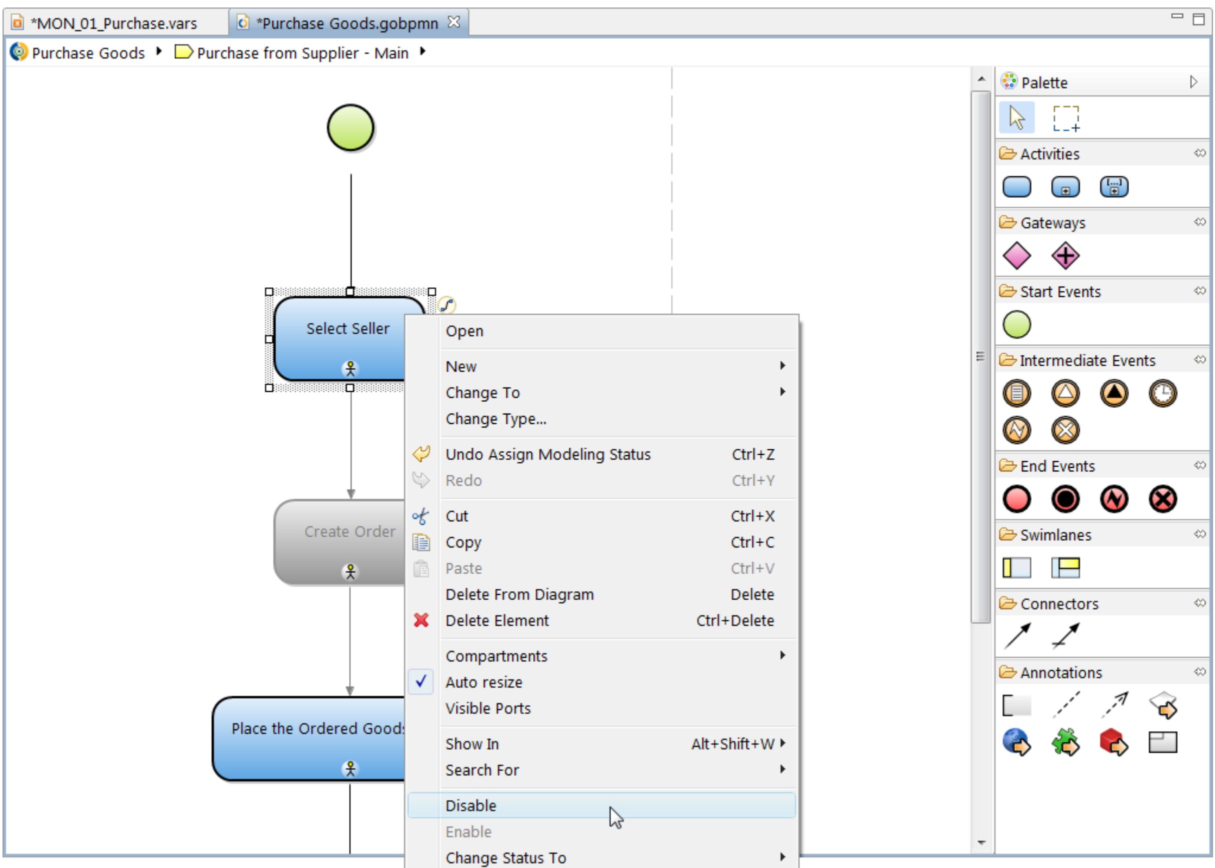


Figure 2.18 Disabling an element in a diagram editor

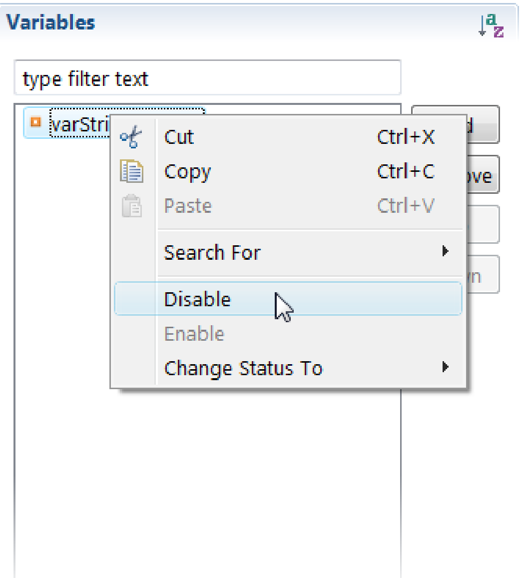


Figure 2.19 Disabling an element in a form-like editor

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 Designer > 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 in monitoring applications.
  - *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.

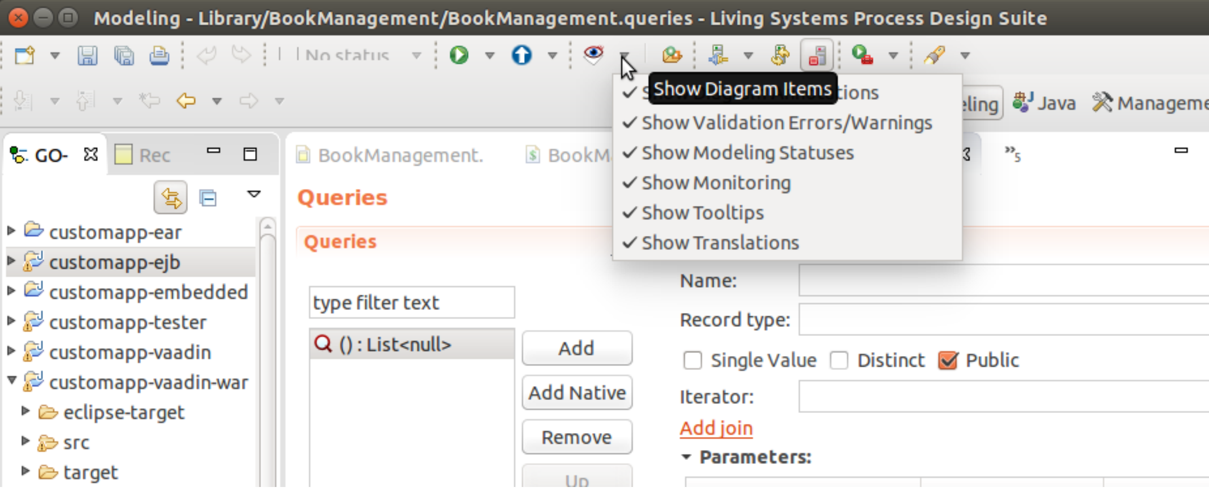


Figure 2.20 Menu with Global Diagram Editor Settings

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

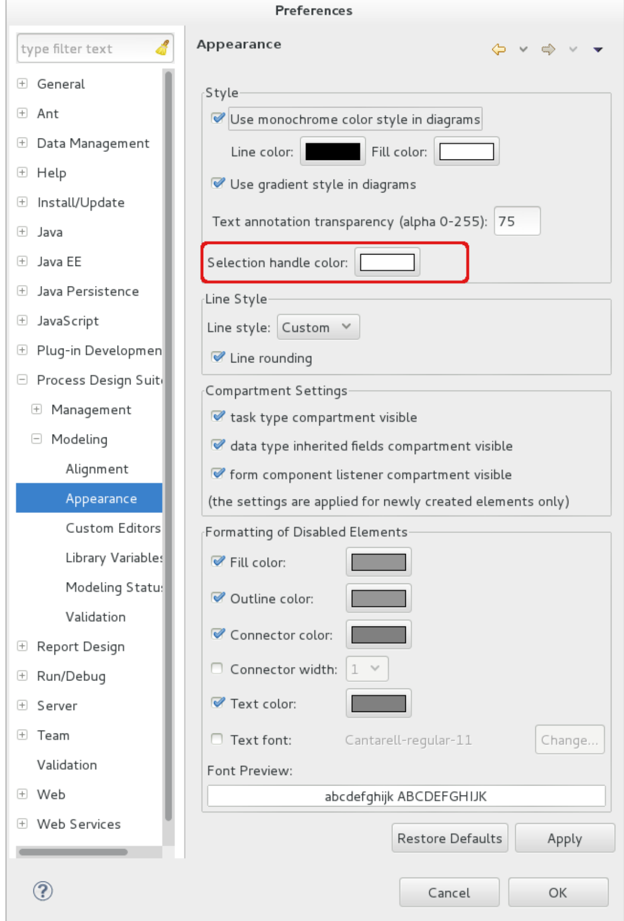


Figure 2.21 General settings

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.

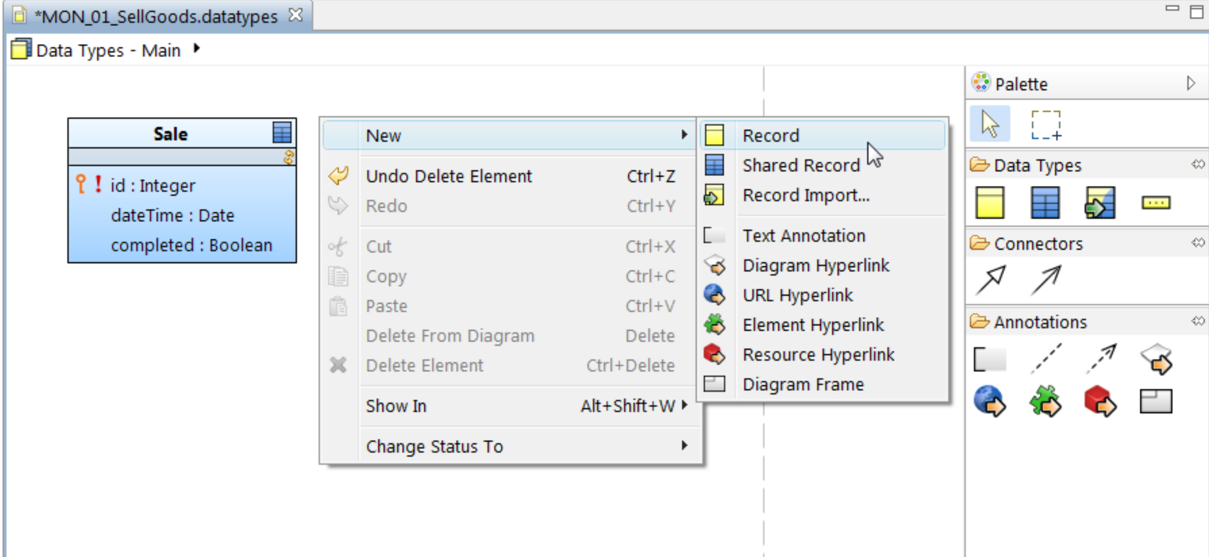


Figure 2.22 Inserting a new element through the context menu

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.

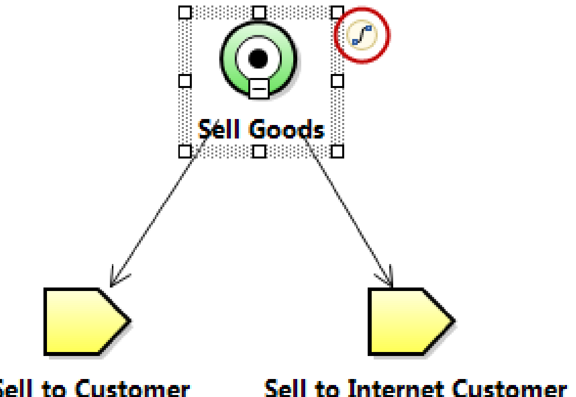


Figure 2.23 Quicklinker

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.

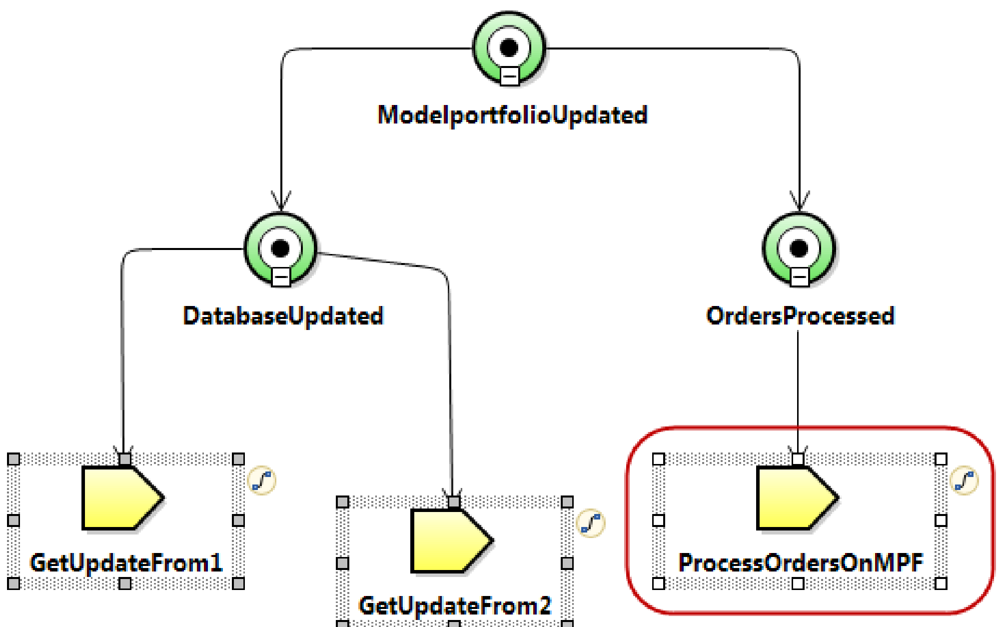


Figure 2.24 Primary element view in a selection

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 Element Type

To change the type of a 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.
   - If applicable, *Change Type* to change the type: 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 Designer > 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 Resetting the Anchor Position of a Line Element

When you connect two diagram elements with a line element, such as a flow, annotation, etc., you can adjust where they connect to the element views: drag the anchors, the black squares, pointing toward the center of the element view to the required position.

You can restore the original anchor position from the context menu of the line element.

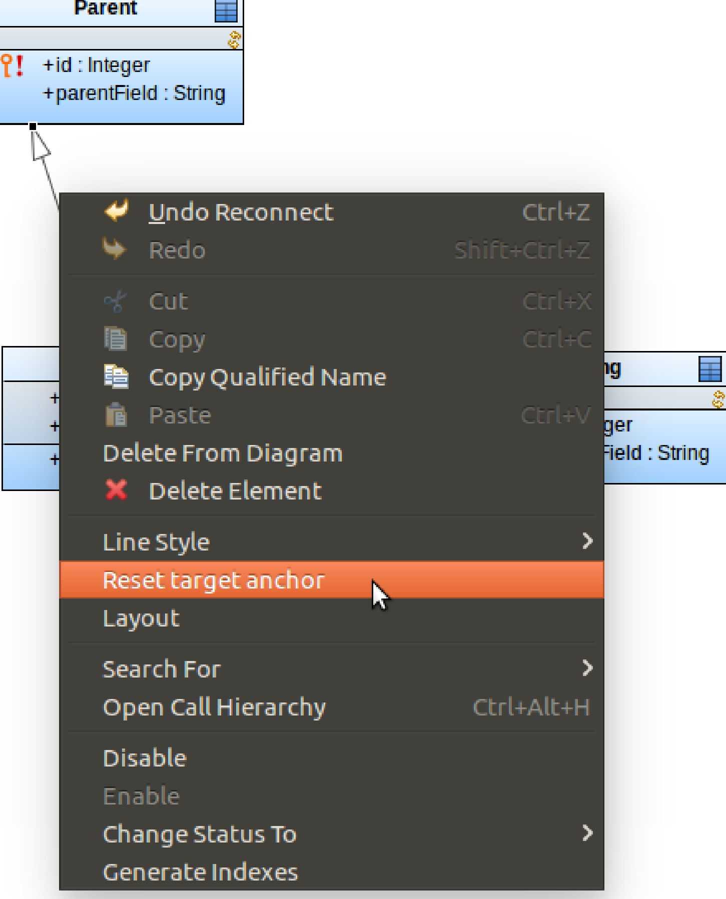


Figure 2.25 Resetting an anchor position

2.2.4.10 Customizing the Palette

You have activated a diagram editor.

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

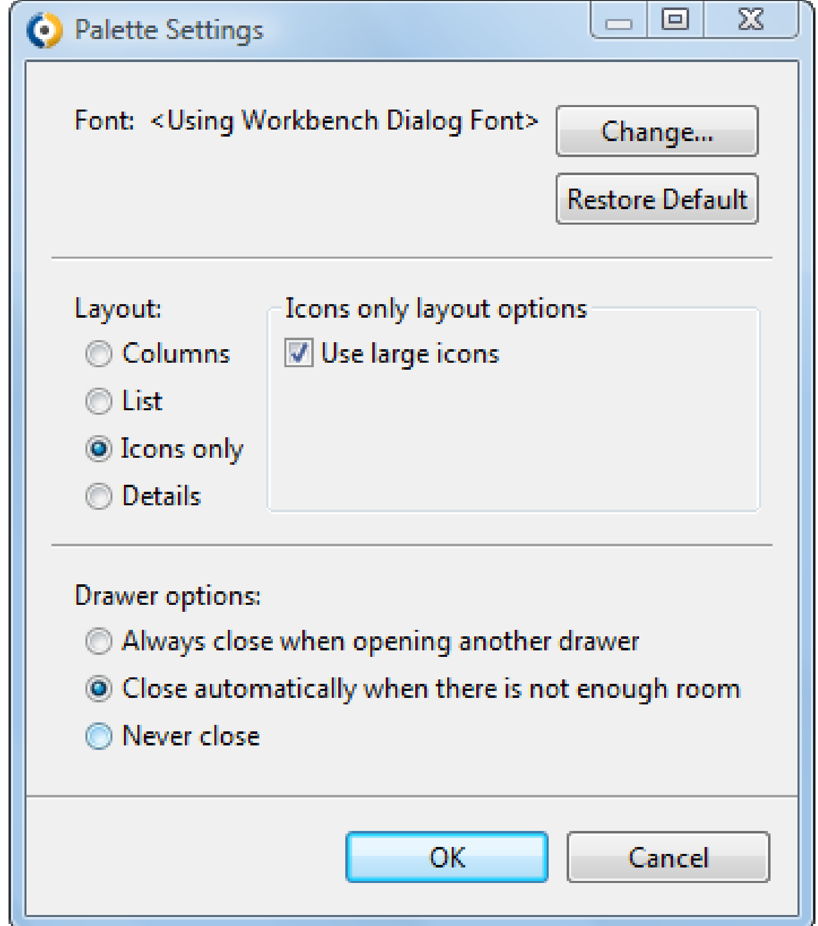


Figure 2.26 Customizing Palette

2. In the dialog box, define the palette properties.
3. Click OK to apply the settings.

2.2.4.11 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 Designer.
3. In the left pane, click Modeling.
4. Under Show Page Borders on Diagram, select the respective option.

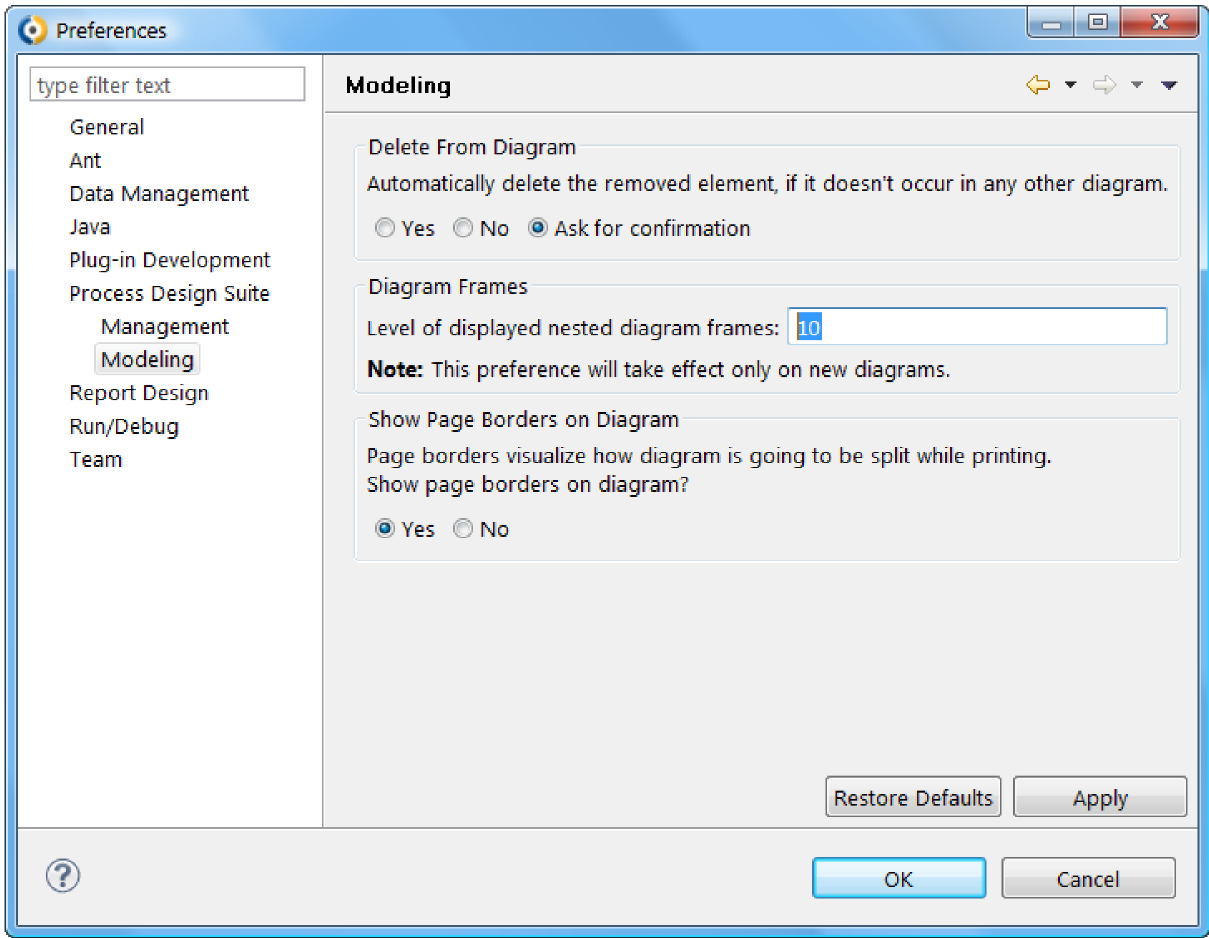


Figure 2.27 Setting the borders

2.2.4.12 Limiting Diagram Frame Nesting Level

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

2.2.4.13 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).

2.2.4.14 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.

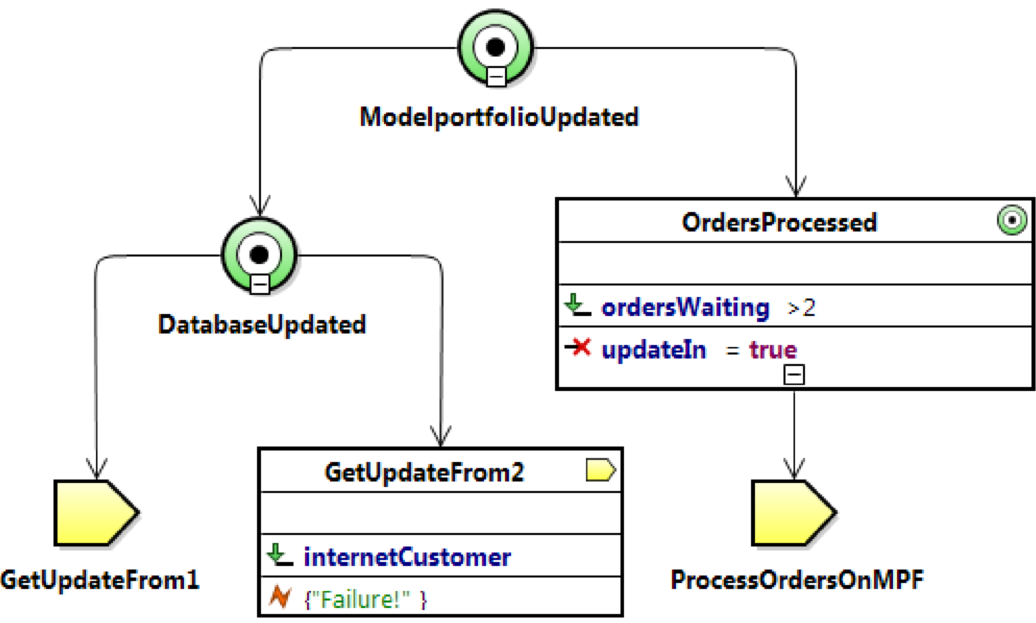


Figure 2.28 Goal hierarchy with views in decorative and iconic notation

2.2.4.15 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.

2.2.4.16 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).

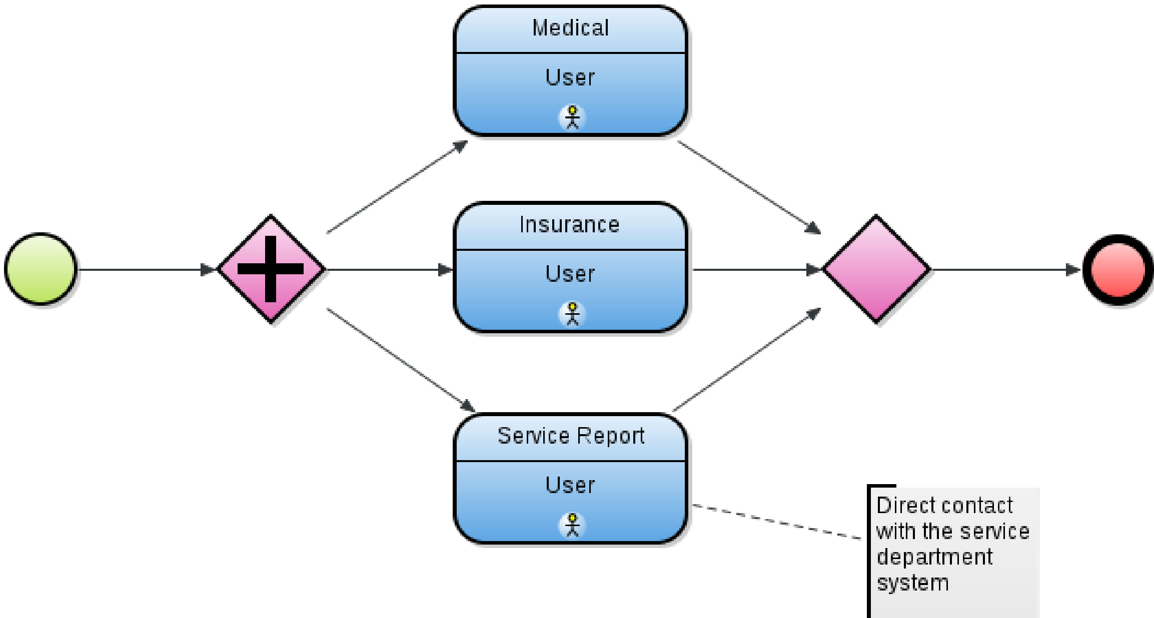


Figure 2.29 Diagram with direct line style

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

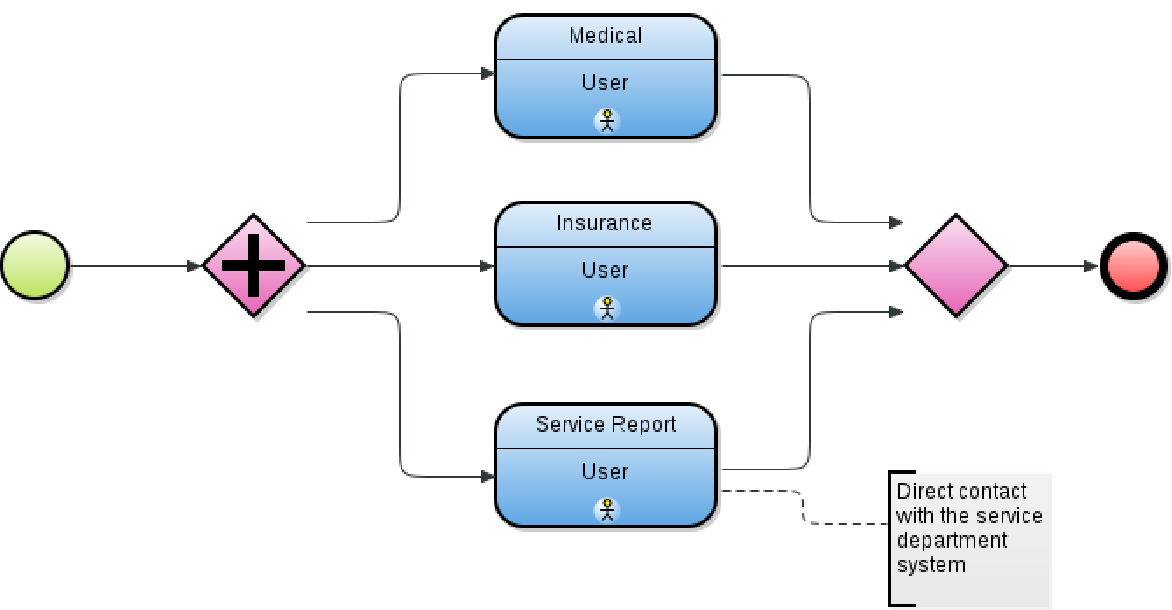


Figure 2.30 Diagram with tree line style

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

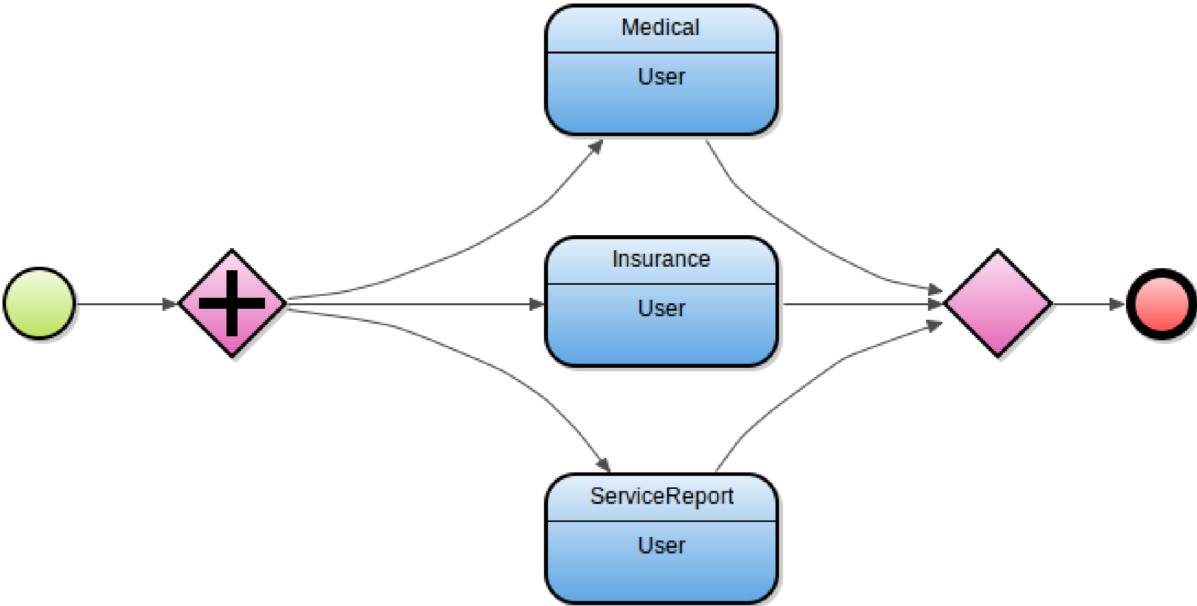


Figure 2.31 Diagram with custom line style

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.

2.2.4.17 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.

2.2.4.17.1 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.

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

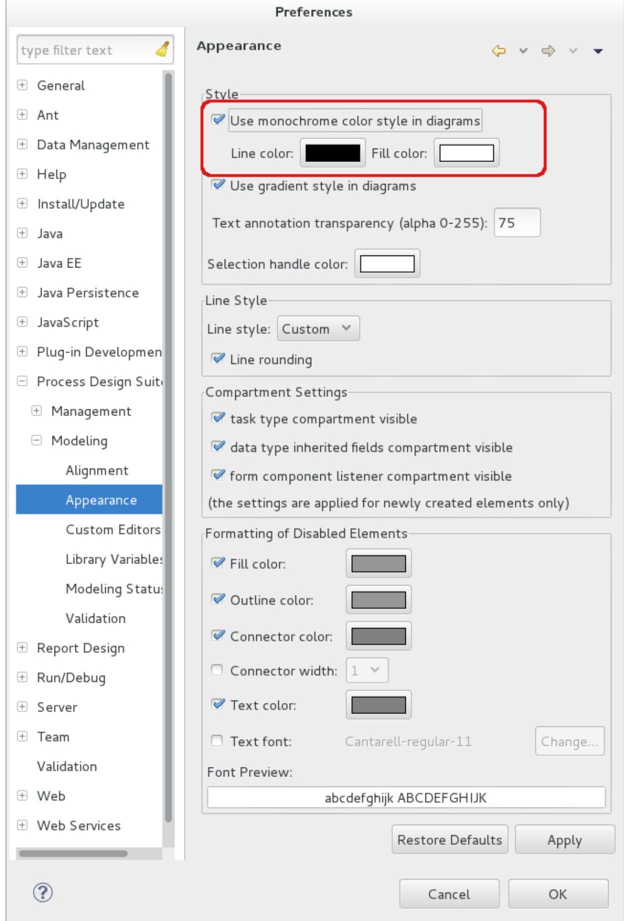


Figure 2.32 Appearance page of the Preferences dialog box

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).

2.2.4.17.3 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 Designer > Modeling > Appearance.
3. On the Appearance page on the right, select or unselect Use gradient style in diagrams.

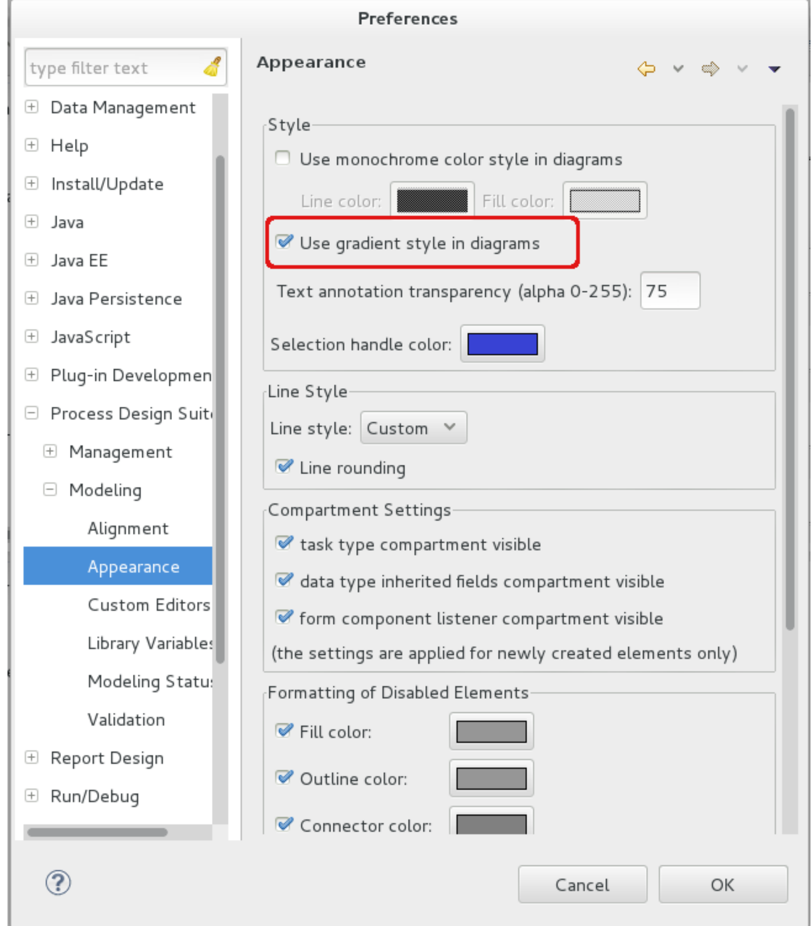


Figure 2.33 Appearance page of the Preferences dialog box

2.2.4.17.4 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 Designer > 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**.

2.2.4.17.5 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 Designer > Modeling > Appearance.
3. On the Appearance page on the right, click the Selection handle color box and select a color.

2.2.4.17.6 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 Designer > 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**.

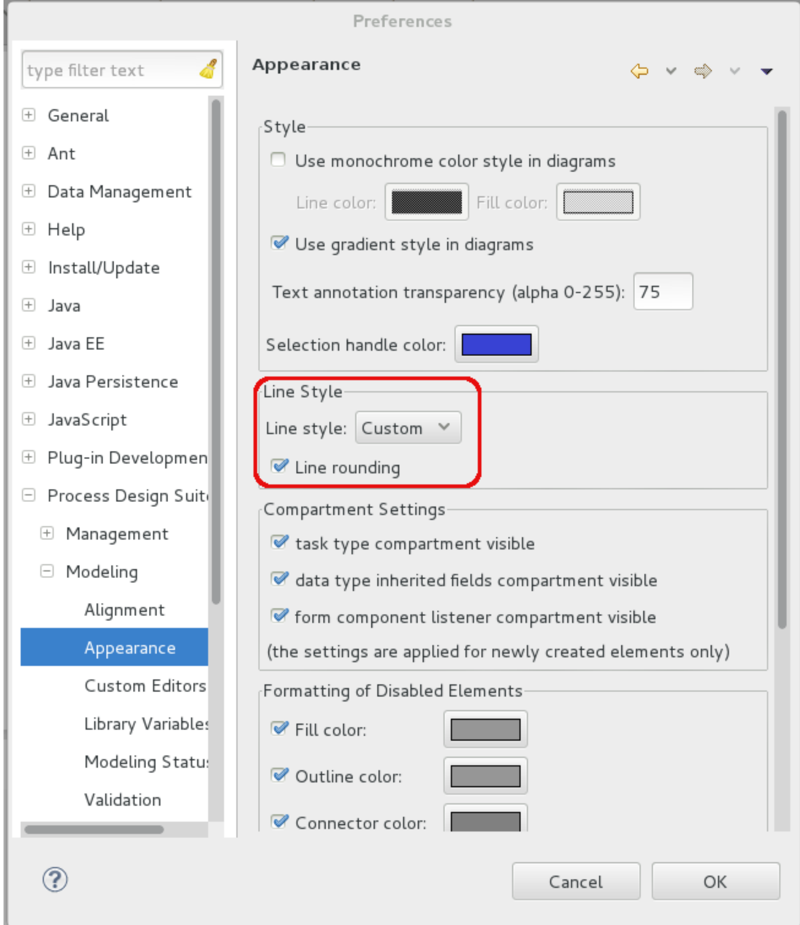


Figure 2.34 Select with the Primary Element in custom color

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

2.2.4.17.7 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.

2.2.4.18 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.

2.2.4.18.1 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.

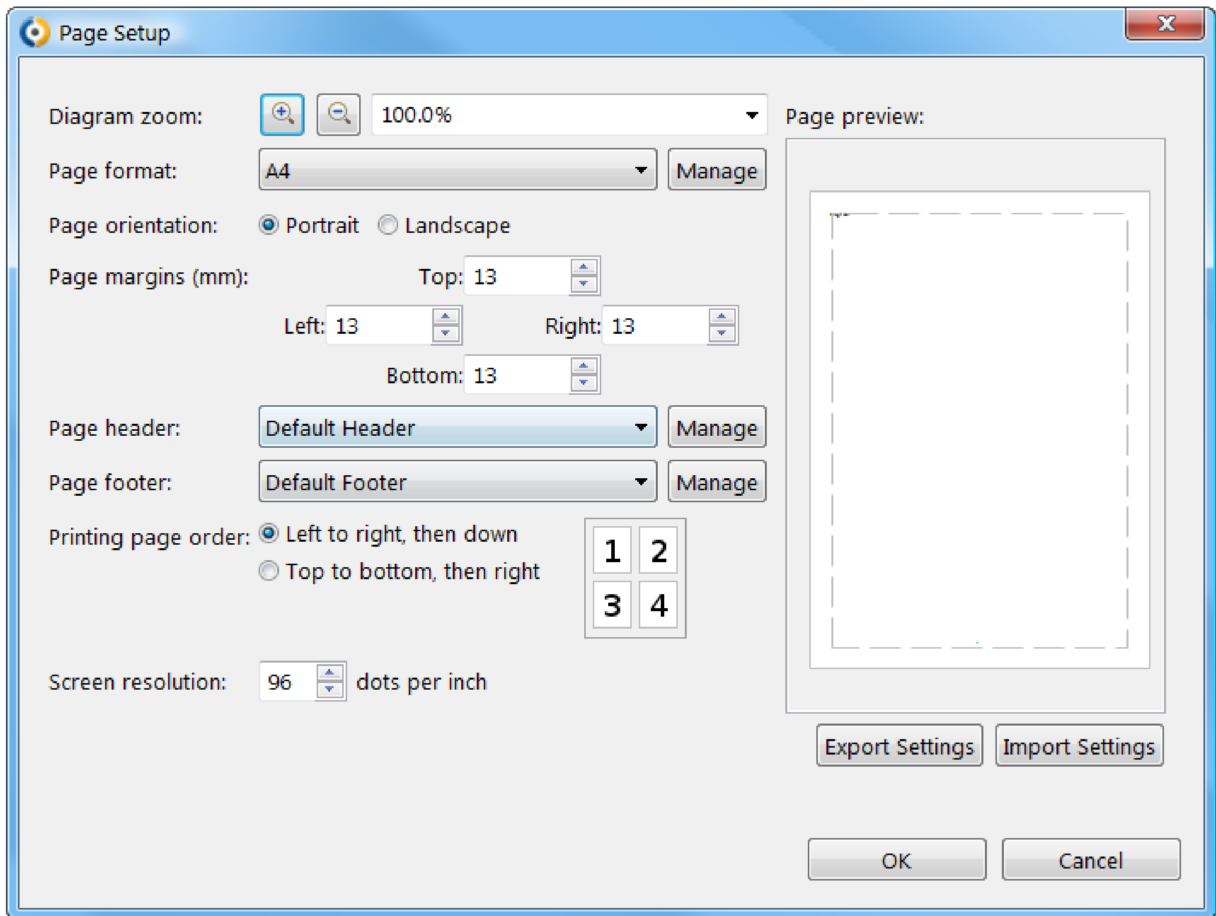


Figure 2.35 Page setup

2.2.4.18.2 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.

2.2.4.18.3 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.

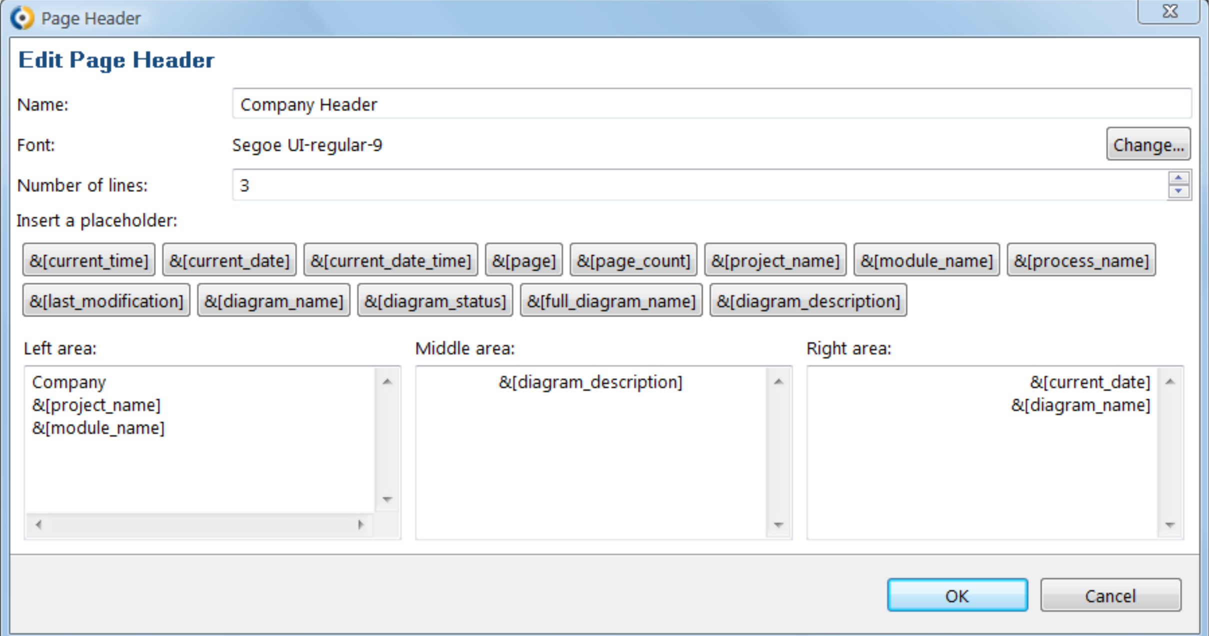


Figure 2.36 Page header setup

2.2.4.18.4 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.

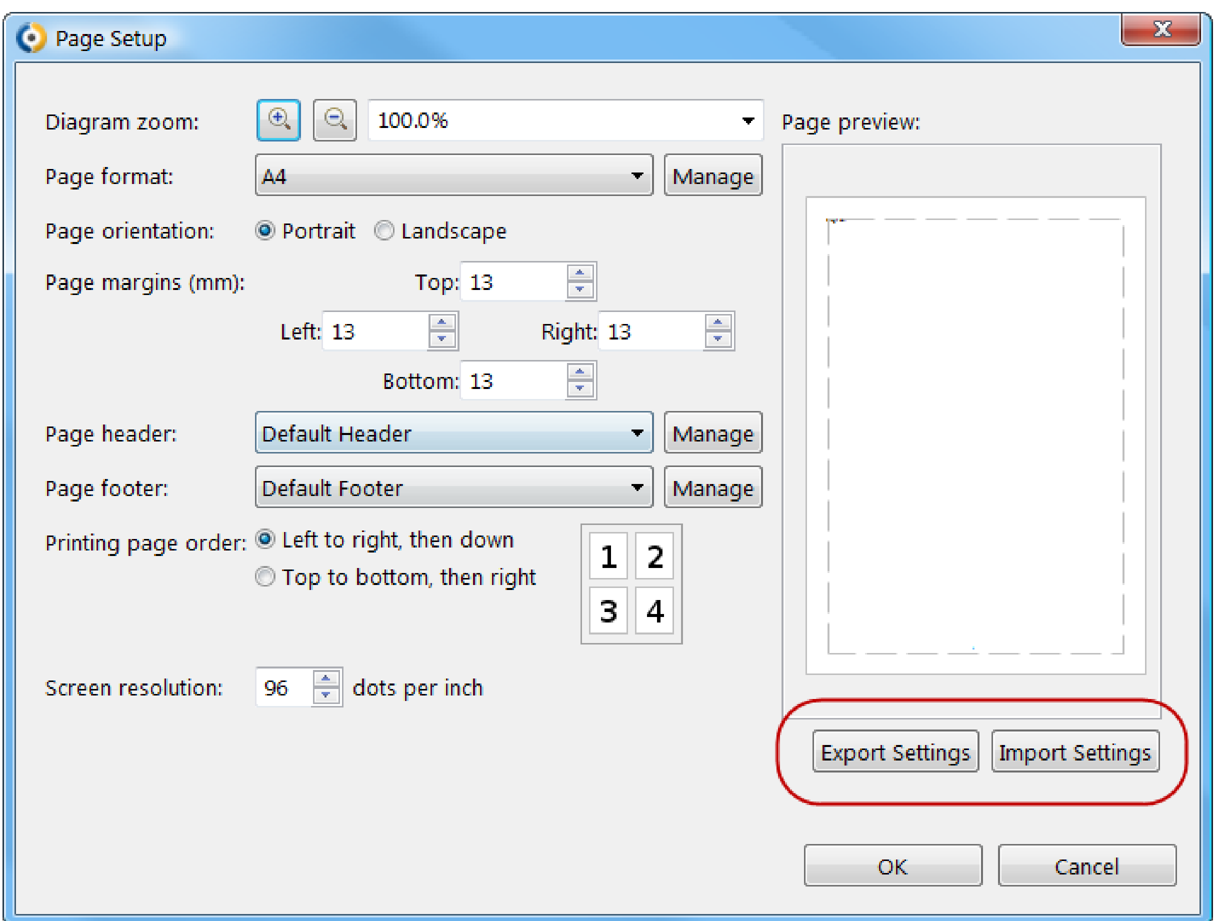


Figure 2.37 Exporting/Importing page setup

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

2.2.4.18.5 Exporting Diagrams as Images

The Designer 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 Designer: go to Window > Preferences and in the Preferences dialog, go to General > Appearance > Colors and Fonts, and under Designer, 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.

2.2.4.18.6 Printing Diagram Element Selection

To print a selection of diagram elements:

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

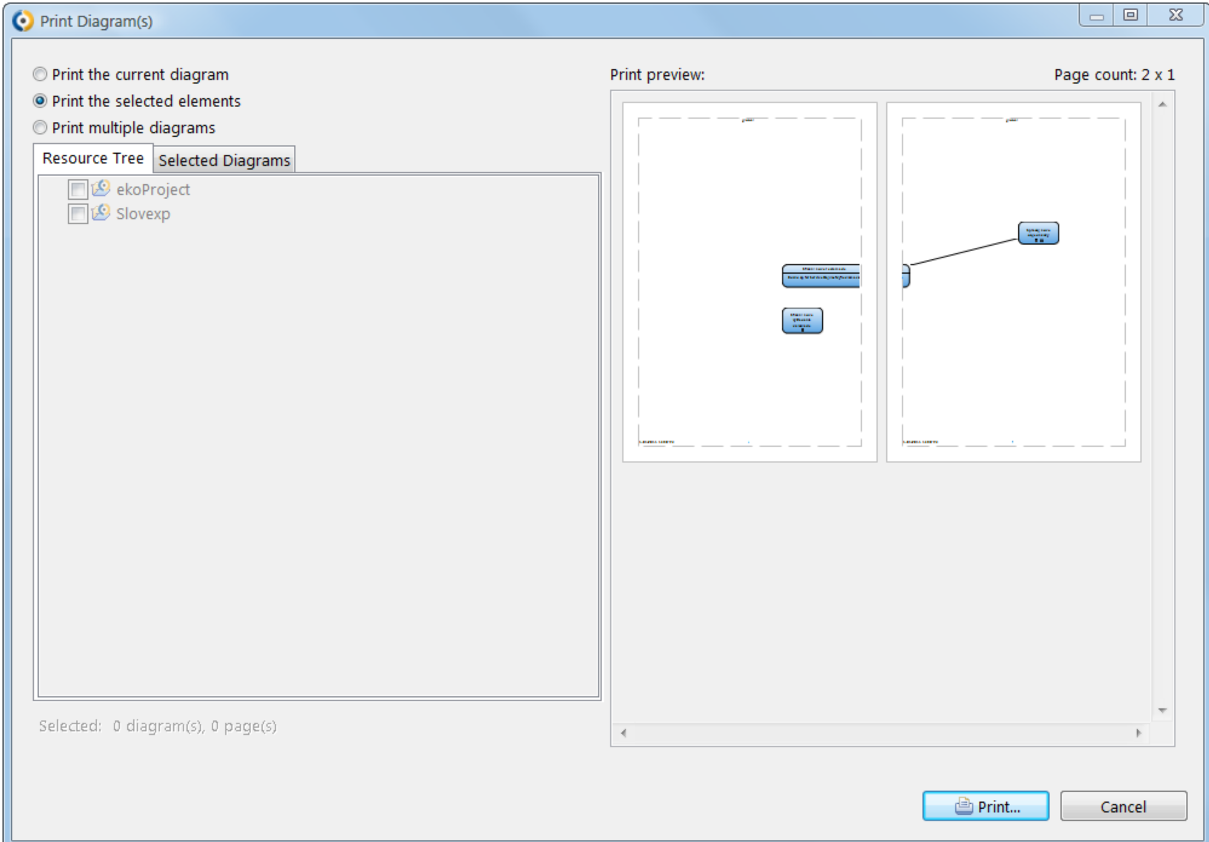


Figure 2.38 Print Diagrams dialog with the Print the selected elements selected

3. Check the print preview on the right.

2.2.4.18.6.1 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:

2.2.4.19 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**.

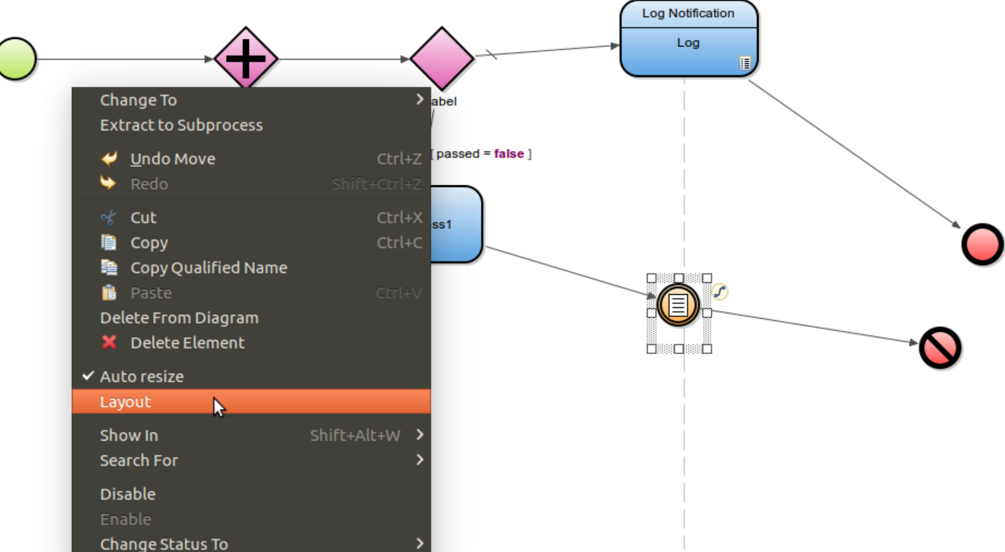


Figure 2.39 Performing autolayout

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).

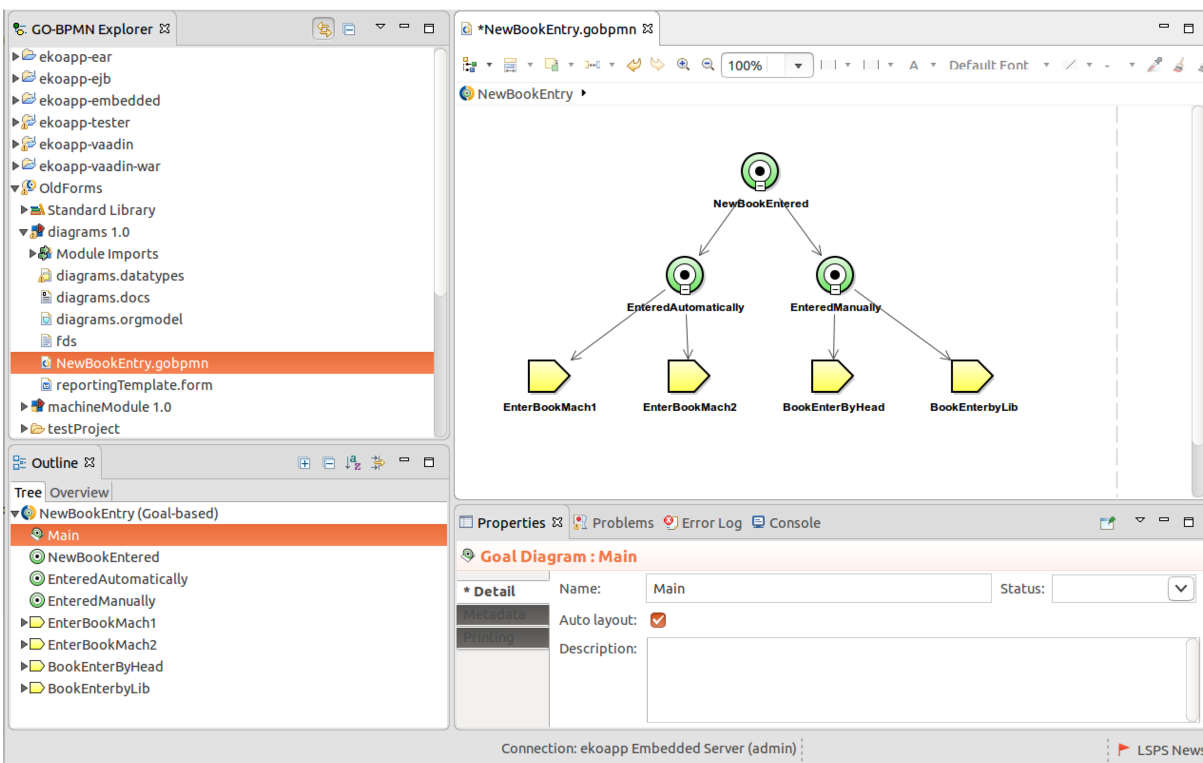


Figure 2.40 Goal diagram opened after Auto-layout was selected

2.2.5 Comparing Resources

You can compare the resources you create in Designer 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.

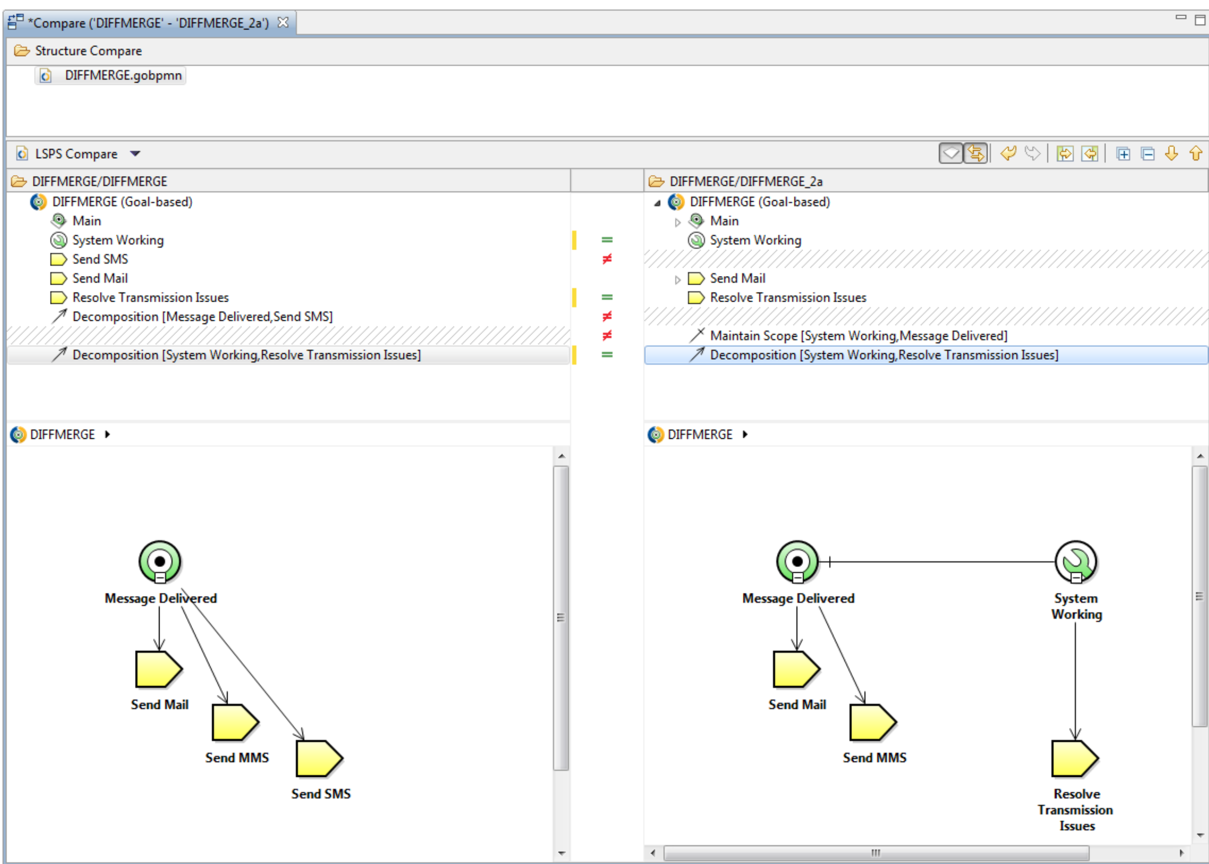


Figure 2.41 Comparing goal-based Processes

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

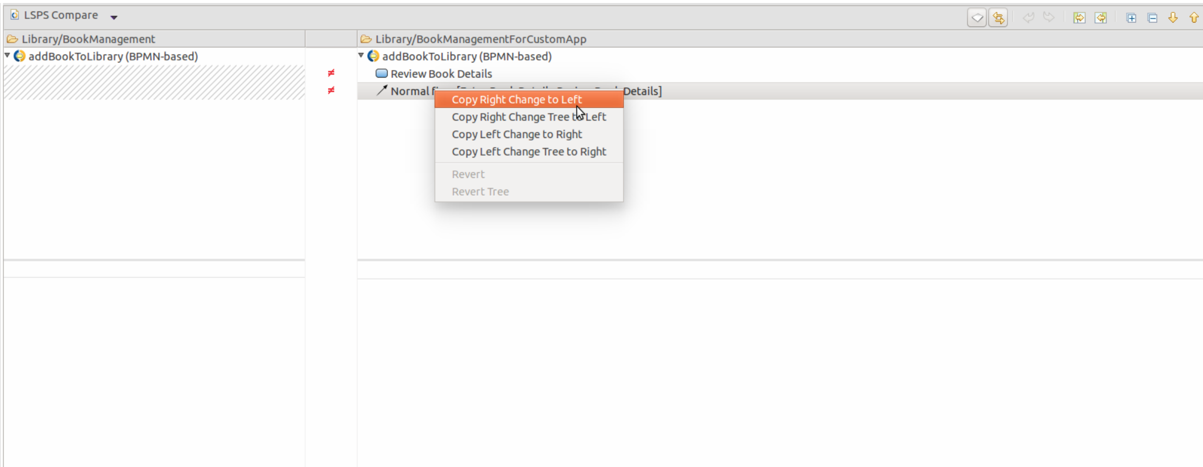
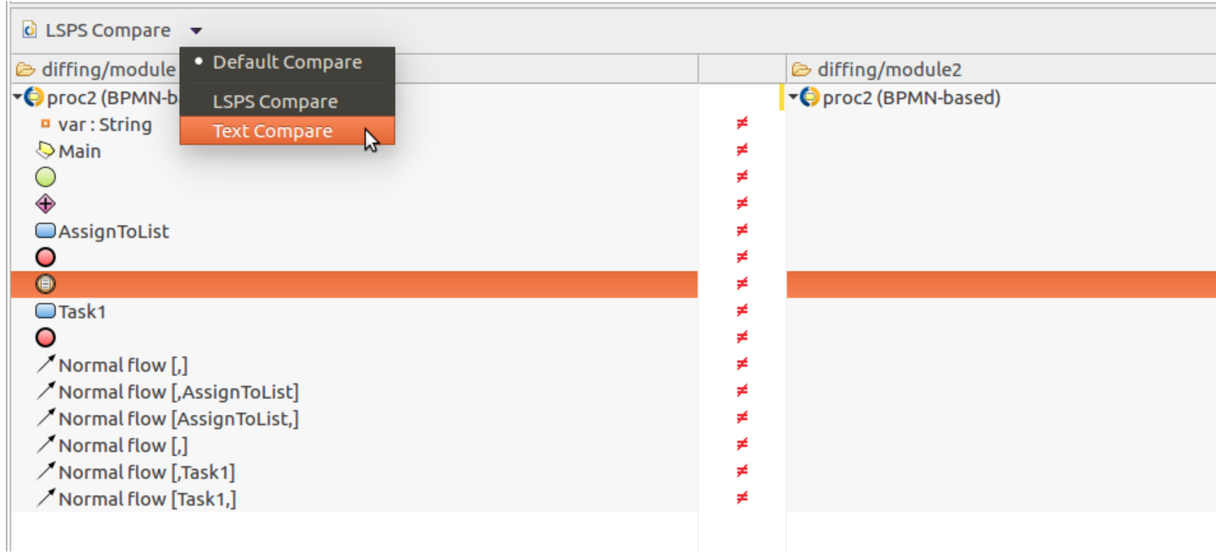


Figure 2.42 Resolving difference

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.

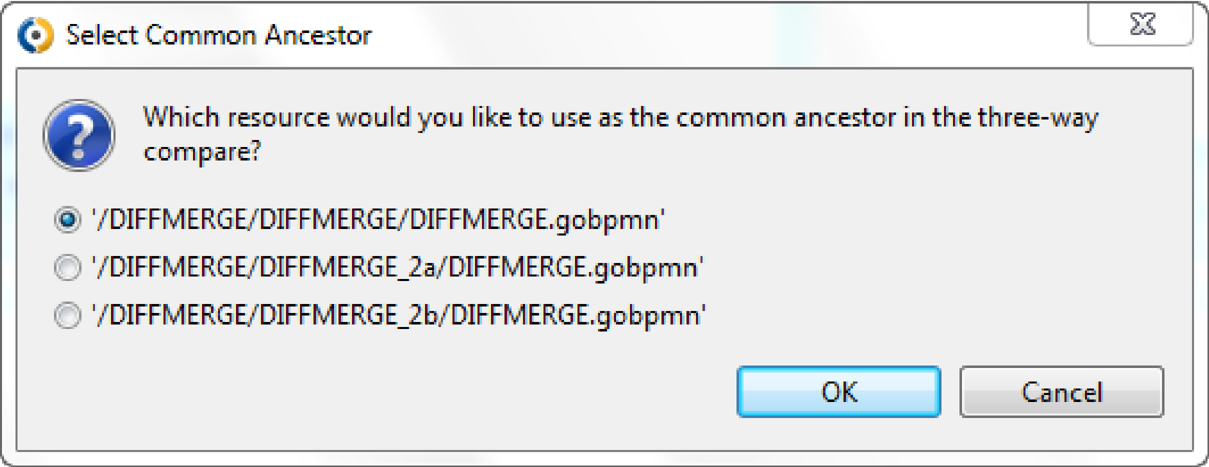


Figure 2.43 Defining the ancestor resource

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).

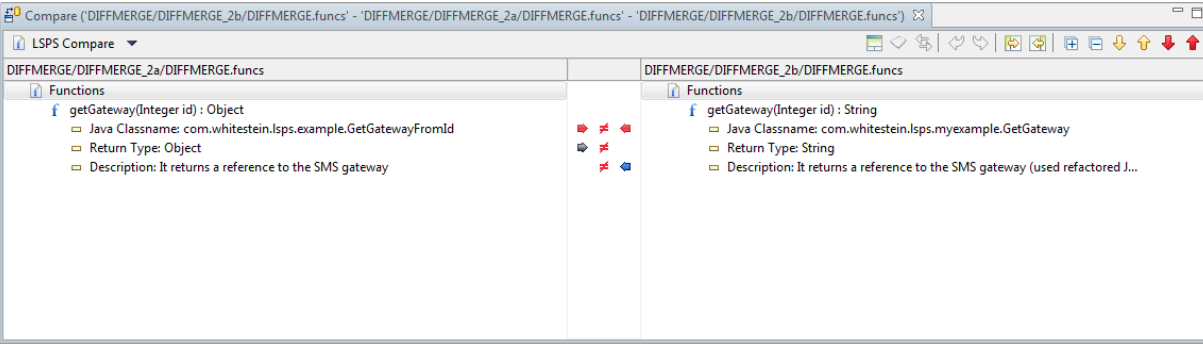
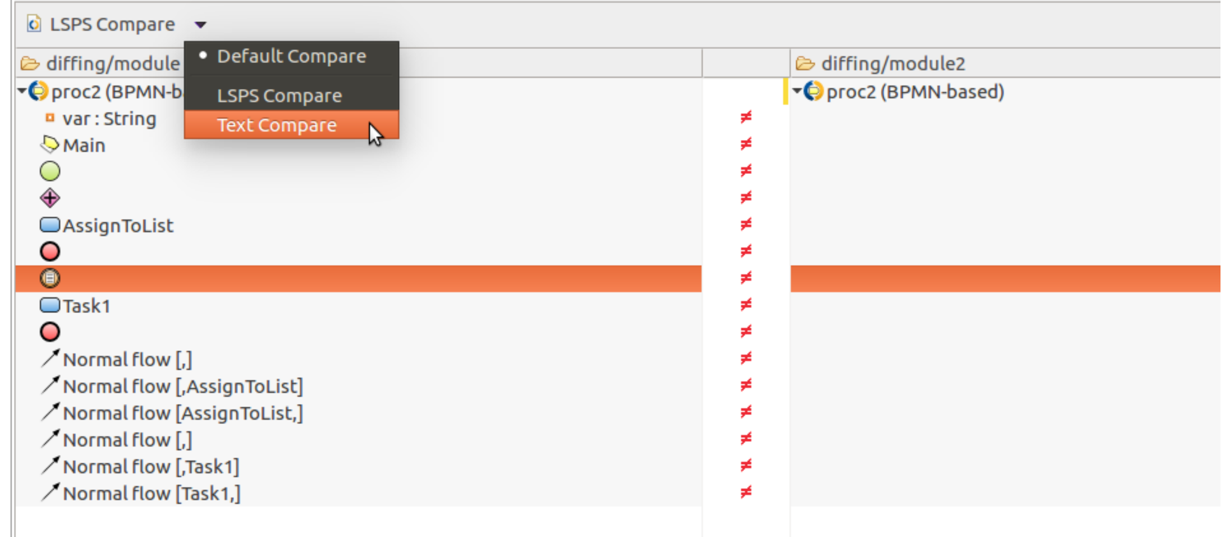


Figure 2.44 Comparing three resources

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 a visual element, do the following:

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.

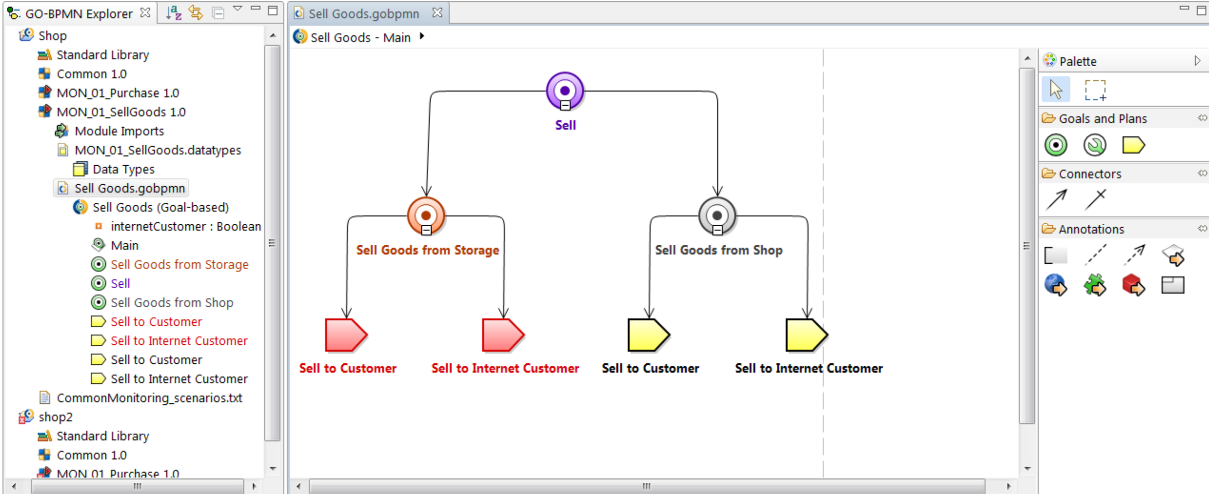


Figure 2.45 Elements with various modeling statuses

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

To specify modeling status on a text element, such as, a text function definition and methods, add the `@Status (<STATUS_NAME>)`, for example, `@Status (Agreed)`.

2.2.6.2 Defining a Modeling Status

To define, edit, or remove a modeling status in your workspace, do the following:

1. Go to **Window** -> **Preferences**.
2. In the Preference dialog, go to **Designer** -> **Modeling** -> **Modeling Status**.
3. Perform the required actions.

Note that modeling statuses can be redefined on individual projects.

2.2.6.3 Defining Modeling Status for a Project

To define, edit, or remove a modeling status in a GO-BPMN project, do the following:

1. In the GO-BPMN Explorer, right-click your project and select Properties.
2. In the left pane of the Properties dialog, click **Designer Modeling Status**.
3. Select the *Enable project specific settings* option.
4. Perform the required actions on modeling statuses.

2.2.6.4 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 Designer 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.

2.2.6.5 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.

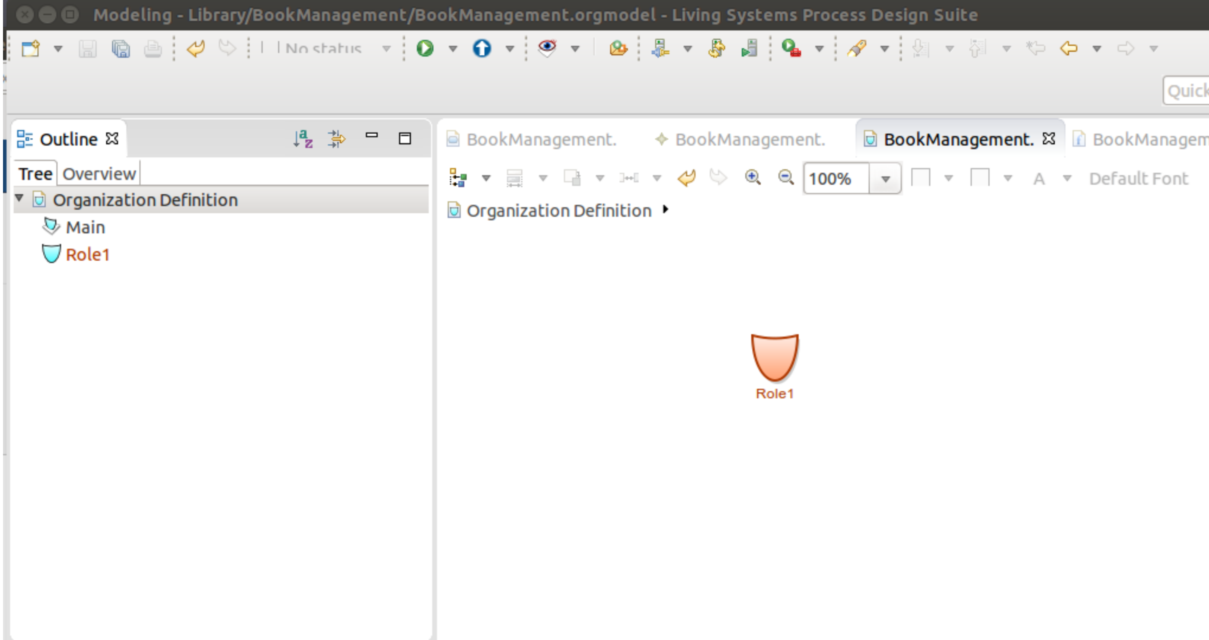


Figure 2.46 Modeling status applied on the element visible in diagrams and in views

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 **Designer** -> **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.

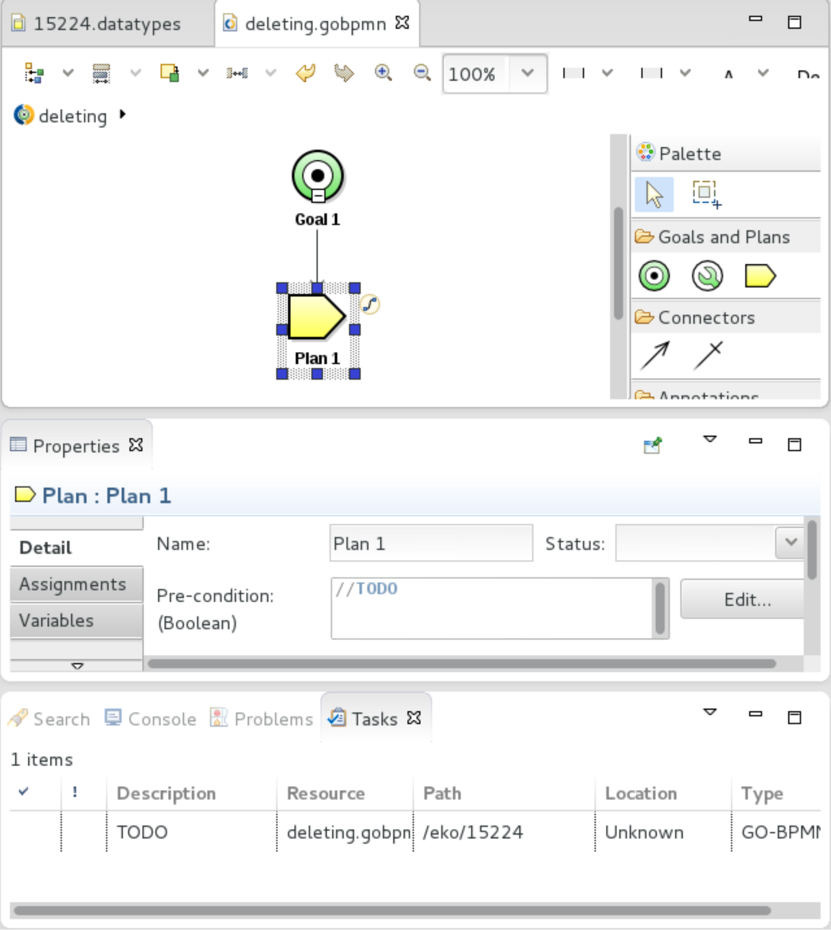


Figure 2.47 The TODO Marker

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.

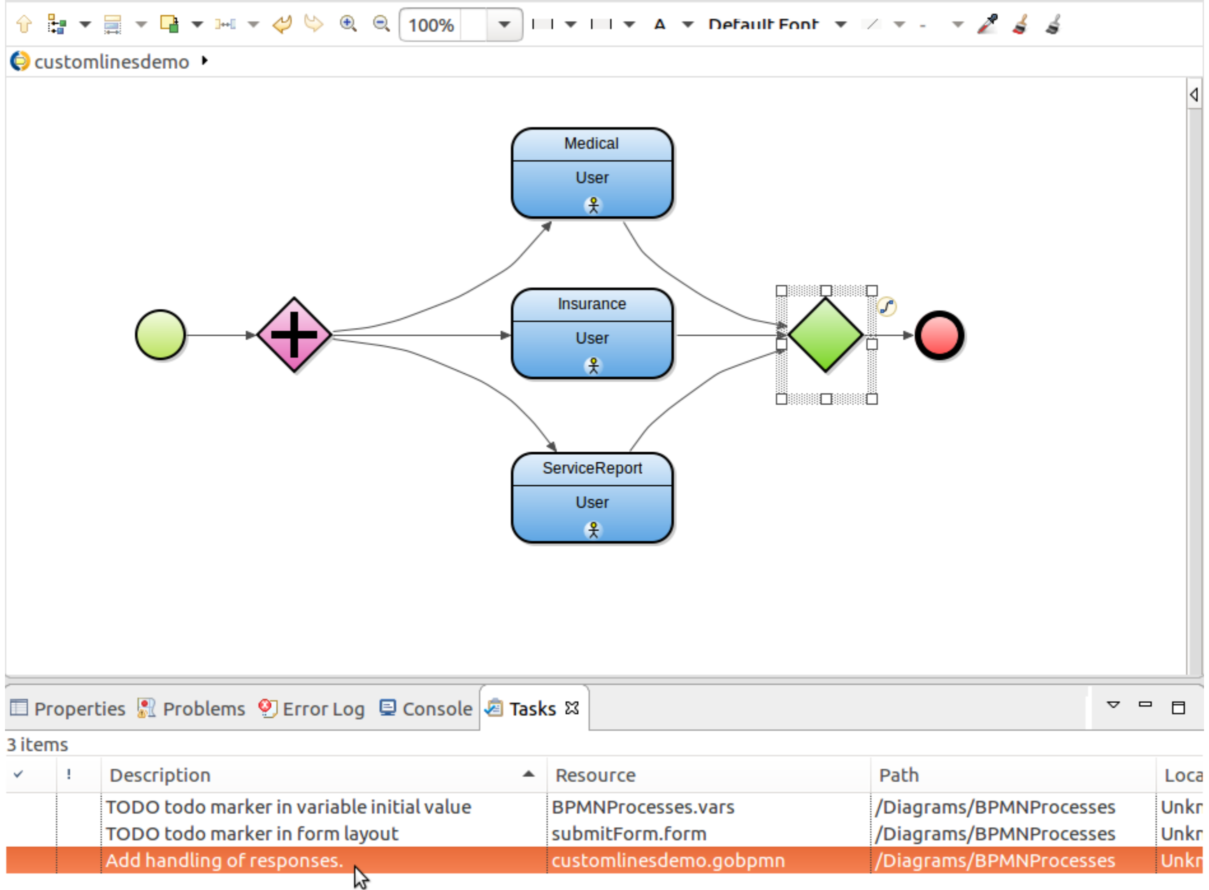


Figure 2.48 Modeling status with the normal priority on an element with its entry in the Tasks view

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.

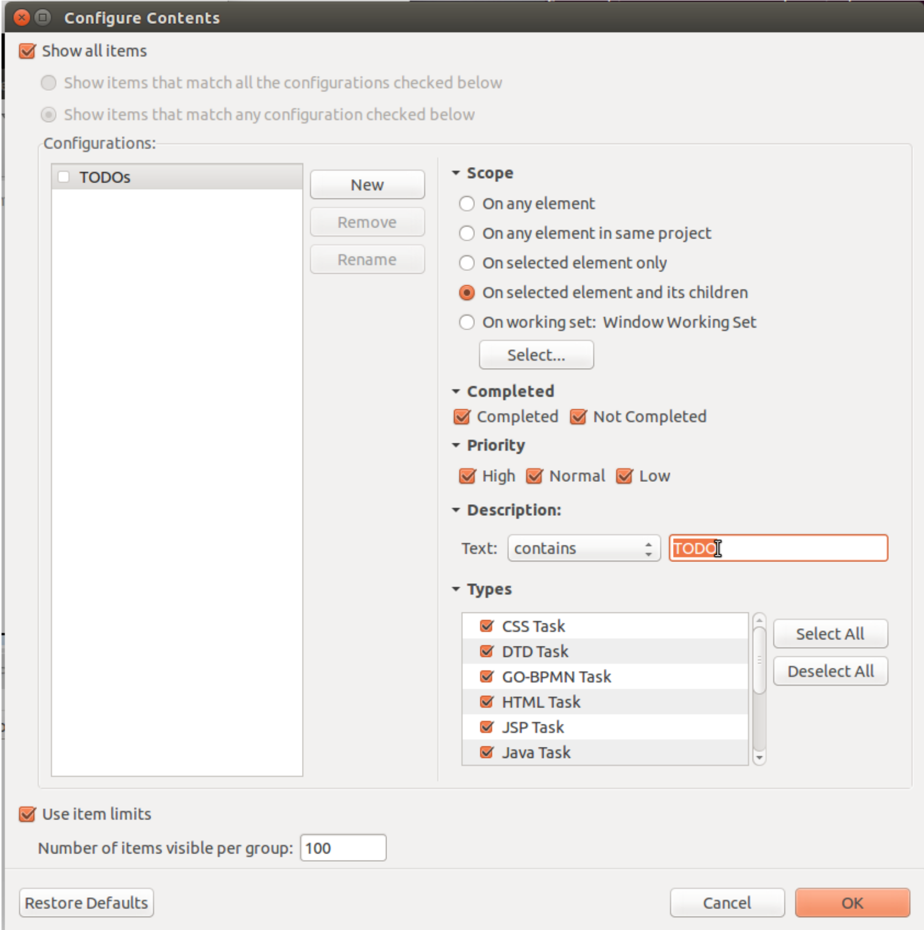


Figure 2.49 Setting contents properties for todo tasks

2.2.8 Validation

Disambiguation note: This section describes the validation of all model resources in Designer, such as, syntax of expressions, missing module imports, return values for element properties, etc. If this is not the topic you looking for, consider the following:

- Validation of records
- `Validation of forms data`
- `Validation of ui forms data`

Validation in Designer checks the language and modeling correctness of the saved workspace data. Note that closed projects, non-executable Processes and contents of non-main Pools are not validated. When it 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 : severe mistakes that prevent the compilation and deployment
- Warnings : 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 in some cases you can define the severity yourself.

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

The detected problems are listed in the Problems view. 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 Designer.
3. Expand Modeling and click Validation.

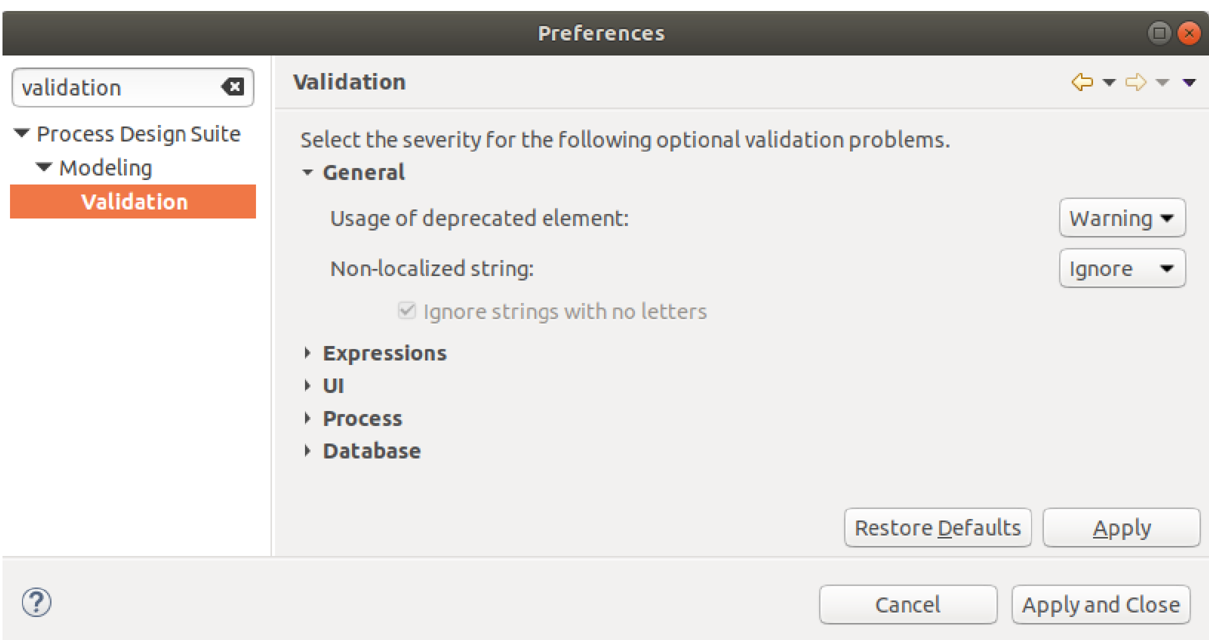


Figure 2.50 Validation page with predefined settings

4. Set the severity of individual problems:

- *General*:
  - Usage of deprecated element: an element with the flag *Deprecated* is used
  - Non-localized string: a String that is not marked as `not-localizable`
- *Expressions*:
  - Usage of deprecated equals: = is used as an equal sign (== is preferred)
  - Duplicate switch case: the case value is used in another case already
  - Potential NullParameterError: the call might receive a null argument
  - Unnecessary cast or instanceof: redundant casting or type check
  - Unused local variable: variable that is never used
  - Unused function parameter: function parameter in the function definition is never used
  - Ignored return value of function or expression without side effects: return value is not used
  - Usage of Null return type: function has Null as its return type (`void` is preferred)
  - Uninitialized record field: a record field is never initialized
  - Uninitialized variable: the variable is never initialized
  - Side-effect expression: an expression that 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.
- *UI* forms:
  - Unused form variable in UI forms: the form variable is never used
  - Missing max-text-size hint: the max-text-size hint is not defined for the input component
  - Mismatch of max-text-size and DB mapping text length: the max-text-size setting on the form component violates the text length set on the shared record field.
  - Redundant custom property: property of the custom component is not present in the underlying Record
- *Process*:
  - Unused process variable: process variable is never used
  - Unspecified non-required task parameter: optional parameter of the task has no values
  - Missing exclusive gateway default flow: exclusive gateway does not a default outgoing flow (none of the flows might be taken.)
- *Database*:
  - Database object name length: maximum database name of a column, table, foreign keys, etc. This setting is primarily intended for Oracle databases, which do not allow names longer than 30 characters.
  - Missing DB index for data relationship: the relationship is not indexed in the database

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

2.2.8.2 Validating Old Modules

Imported GO-BPMN Modules created in older versions of Designer 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:

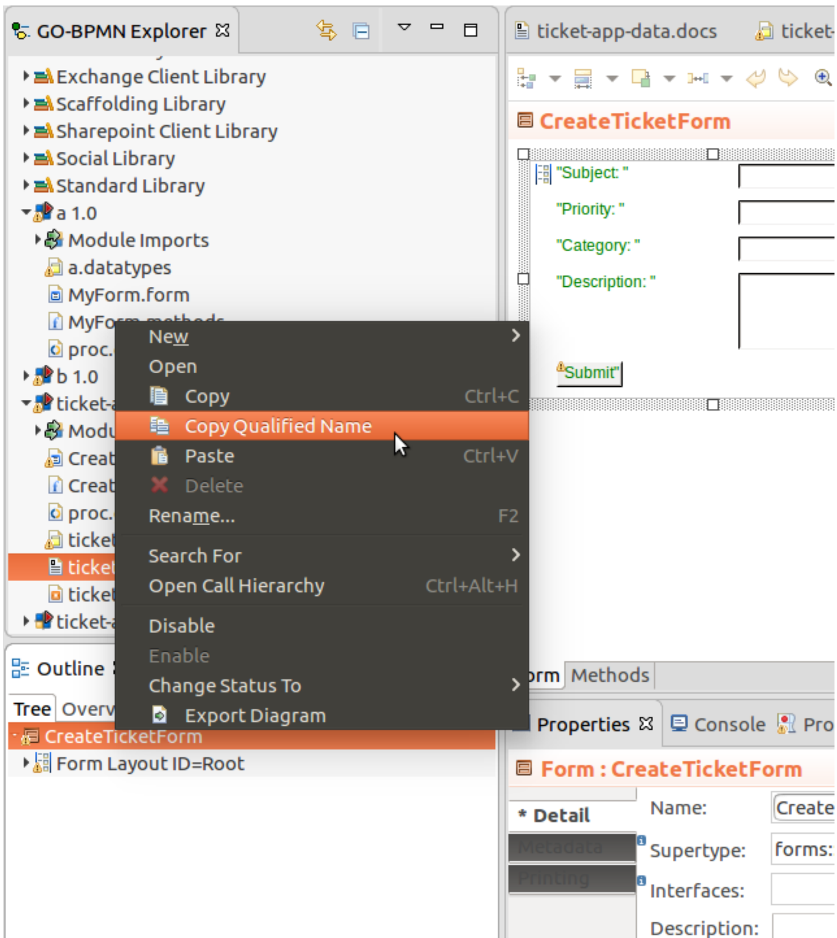
1. Click Show Diagram Items ( ) button on the main toolbar.

   The button is available only when a diagram editor is focused.

2. Unselect Show Validation Errors/Warnings.

2.2.9 Copying Fully Qualified Names of Elements

Note that for elements that can be referenced in the application, such as, Documents, Processes, global variables, etc. you can copy their fully qualified name into your clipboard with **Copy Qualified Name** in their context menu, typically in their Outline view, and paste it into your code.



2.2.10 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.

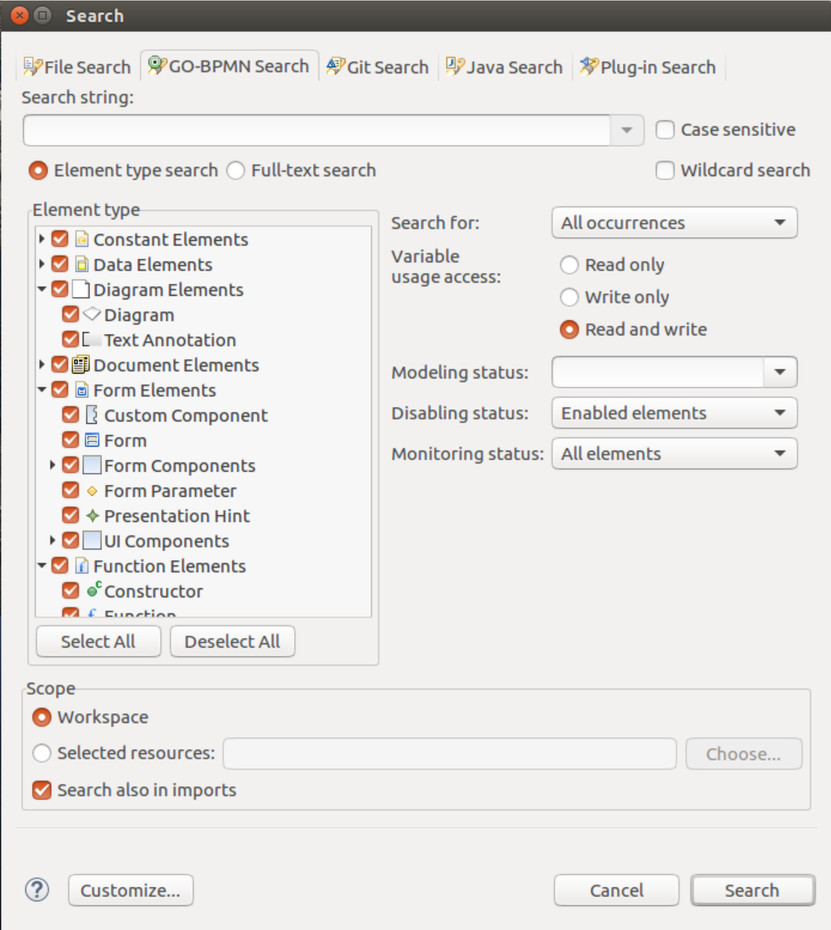
2.2.10.1 Searching for GO-BPMN Entities and their Usage

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.



2.2.10.2 Searching for Elements

The *Open Element* feature is a search that returns any element that contain 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.

2.2.10.3 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.

2.2.10.4 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.

2.2.10.5 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**.

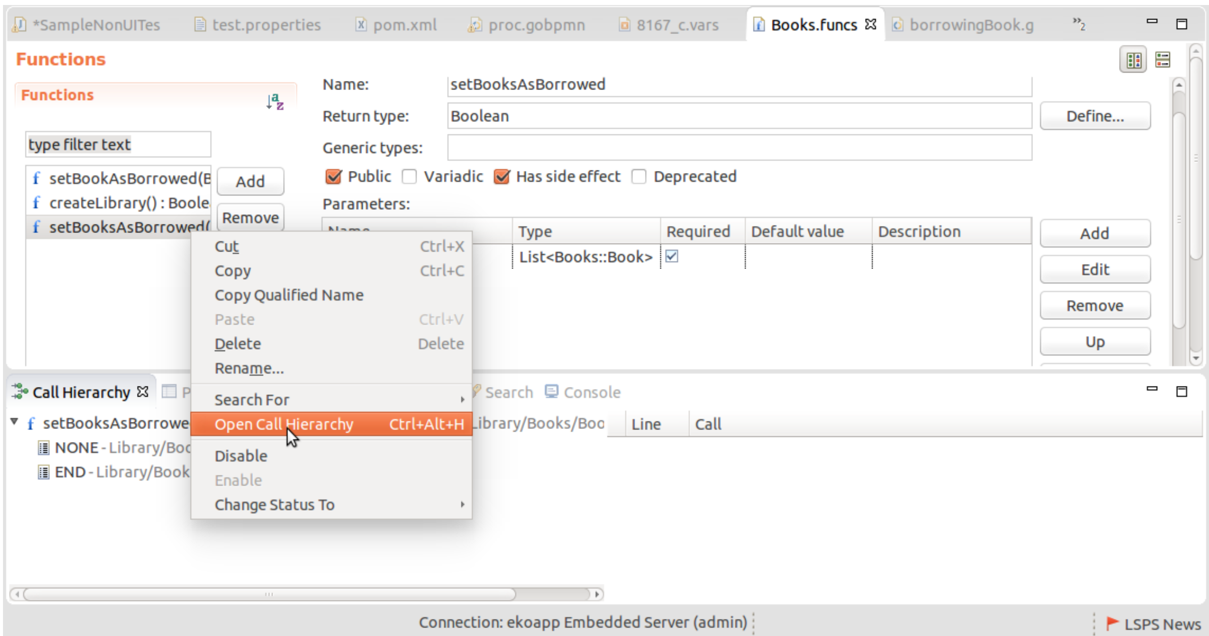


Figure 2.51 Opening call hierarchy of a function definition

2.2.10.6 Searching for Unused Elements

To search for unused elements, go to **Search** > **Find Unused Elements**.

> **Note:** The search does not return unused methods.

2.2.10.7 Searching for Modeling Elements by ID

When looking for the modeling element that caused an exception on runtime, use the modeling ID: The ID is included in the error log.

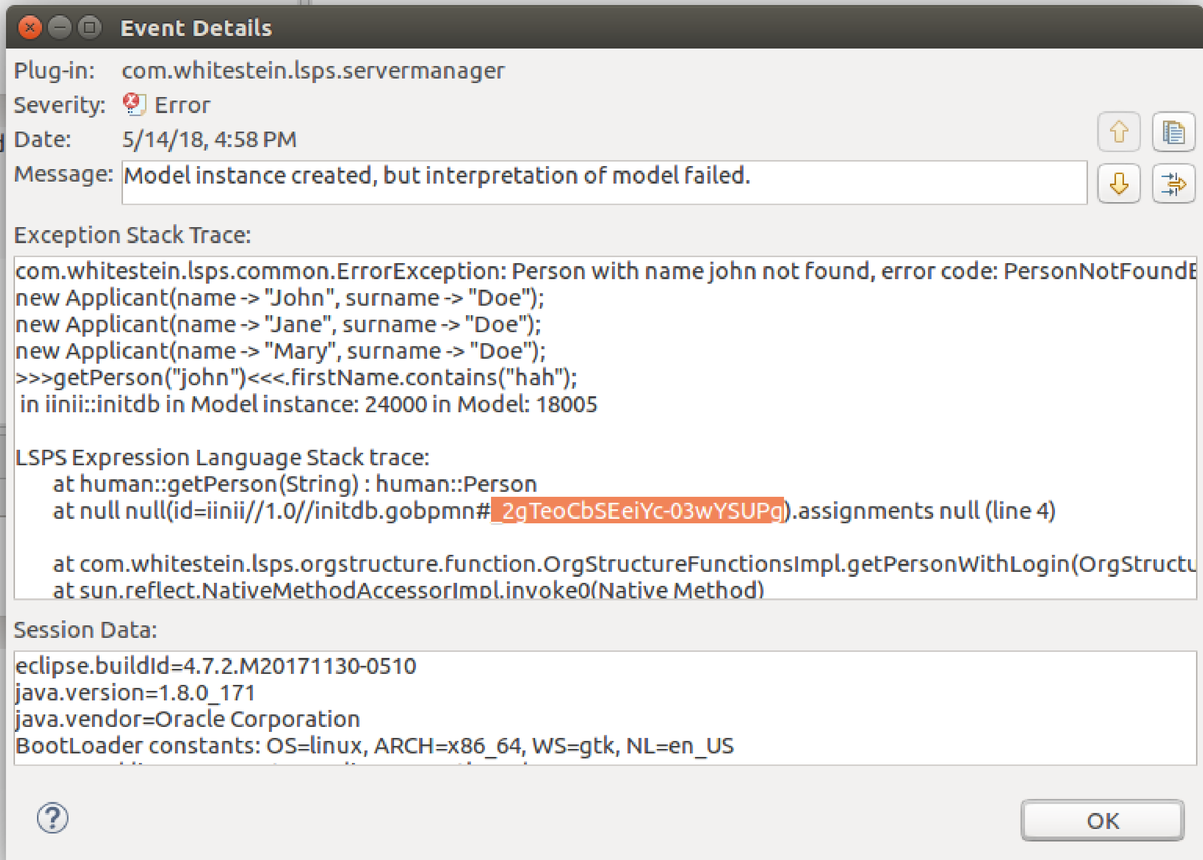


Figure 2.52 Error log with the modeling Id of the element that caused it highlighted

To search for modeling elements by their exact ID, go to **Search** > **Find Model Element**.

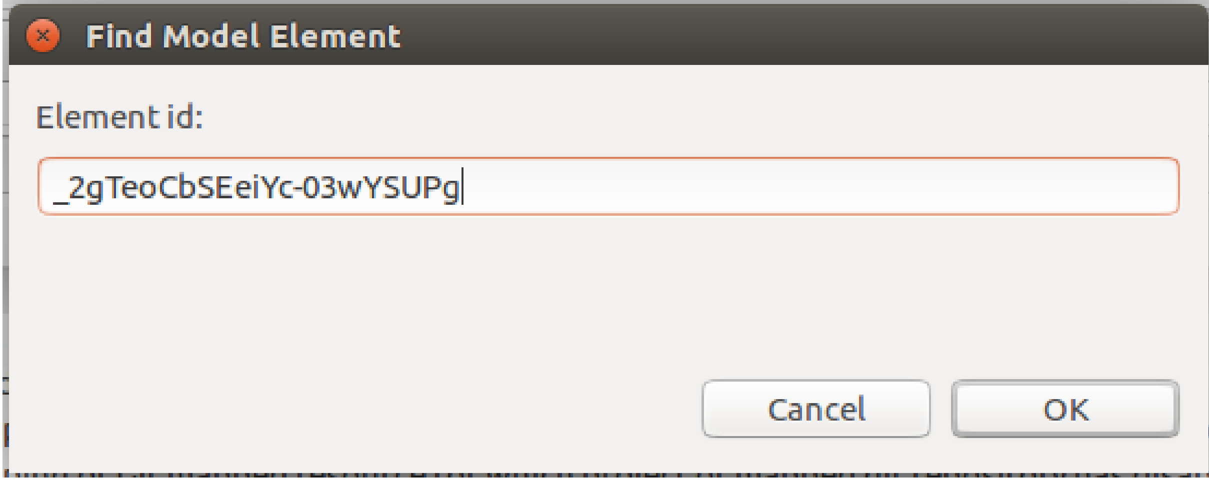


Figure 2.53 Search for the modeling element by its id

2.2.11 Solving Errors

Errors thrown by Designer 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 Designer feature, which appear primarily in the Console view.

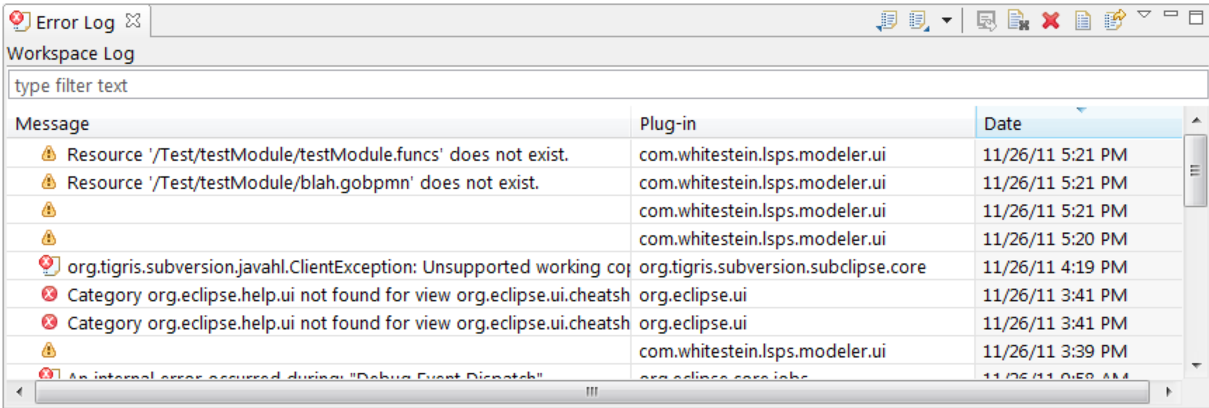


Figure 2.54 Error Log view

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 Modeling 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.

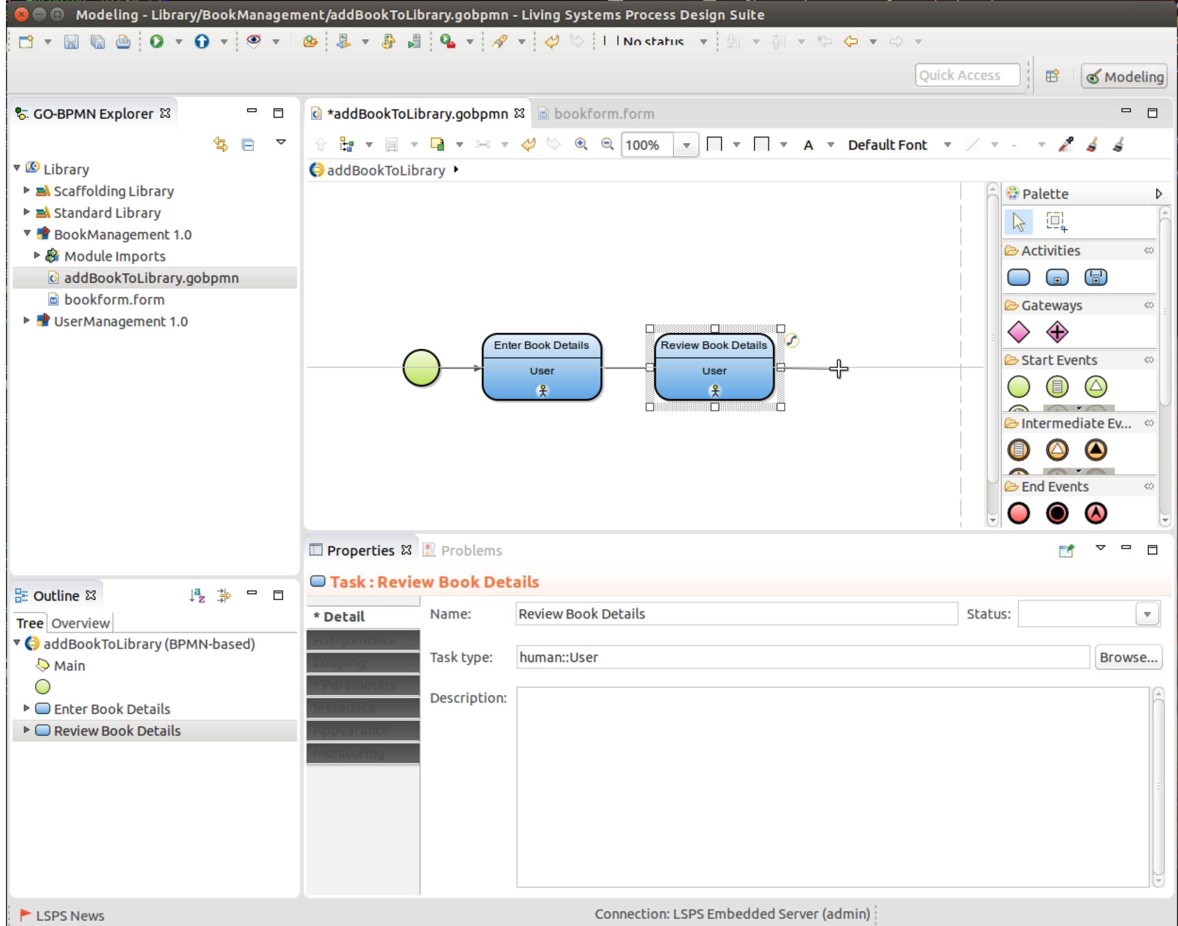


Figure 2.55 Creating Process workflow

2.3.2 Defining Element Labels

To allow the same name on multiple elements in a process diagram, all process elements with semantic value, which excludes Annotation, Diagram Frame, Hyperlinks, can define the Label property: if the label contains a value, it is displayed on the diagram instead of the name of the element. It is intended for purely presentation purposes and does not have to be unique in the namespace unlike the element name.

To define an element label, do the following:

1. Open the process diagram with the element.
2. Select the element.
3. In the Properties view, go to the *Appearance* tab and define the label in the *Label* field.

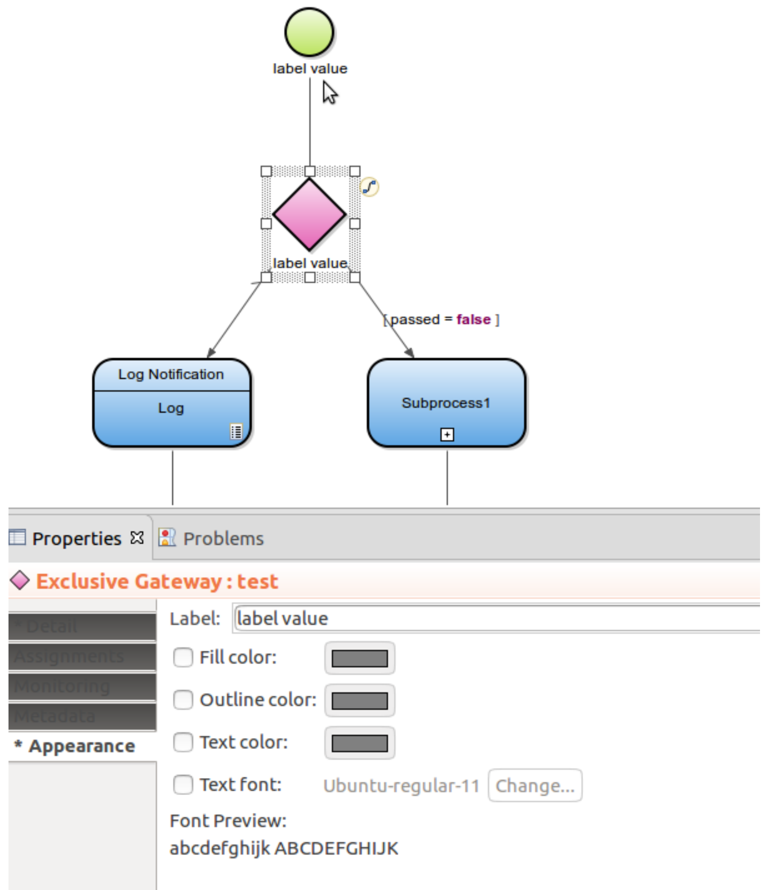


Figure 2.56 Identical labels on multiple process elements

2.3.3 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)
```

2.3.4 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.

2.3.5 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.

To insert a Task as part of your process, do the following:

1. Open the process definition. If using GO-BPMN processes, make sure to open the respective Plan.
2. In the palette, select the Task and click into the canvas where you want to place the Task.
3. Select the type of the Task in the dialog box.
4. Define the task parameters: you can do so either in the Parameters tab of its Properties view or in the built-in task editor after double-clicking the Task element.

Figure 2.57 Defining task parameters

The syntax of the parameter expression is a comma-separated list of key-value pairs:

```
<parameter1_name> -> <parameter1_value>,
<parameter2_name> -> <parameter2_value>
```

User Task parameters

```
title ->  "My Pending Item",
performers ->  {anyPerformer()},
uiDefinition ->   myTodoForm()
```

2.3.5.1 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 the expression with parameter values remains unchanged and you need to adapt the parameters to the new Task Type.

2.3.5.2 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.

2.3.6 Reusing a Process

A process can be reused in another process: a reused process is instantiated as a subprocess of the parent process within its own context. You can reuse a process either with the *Reusable Sub-Process*, which is a standard BP↩ MN way or *Execute* task type, which allows you to reuse a process or task without actually knowing what you are reusing:

- To reuse a process in a `Reusable Sub-Process` task, insert the Sub-Process task to your process flow and, in its Detail properties, set its *Reference Process Name* to the process.
- To reuse a process in an Execute task, in the process properties, set the *Create activity reflection type* and set it as the *activity* parameter of the Execute task.

In both cases, make sure your Process has the *executable* option selected. Also, consider disabling the *Instantiate Automatically* on the Process so that it is not automatically instantiated when the parent Module is instantiated.

2.3.6.1 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**.

2.4 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.

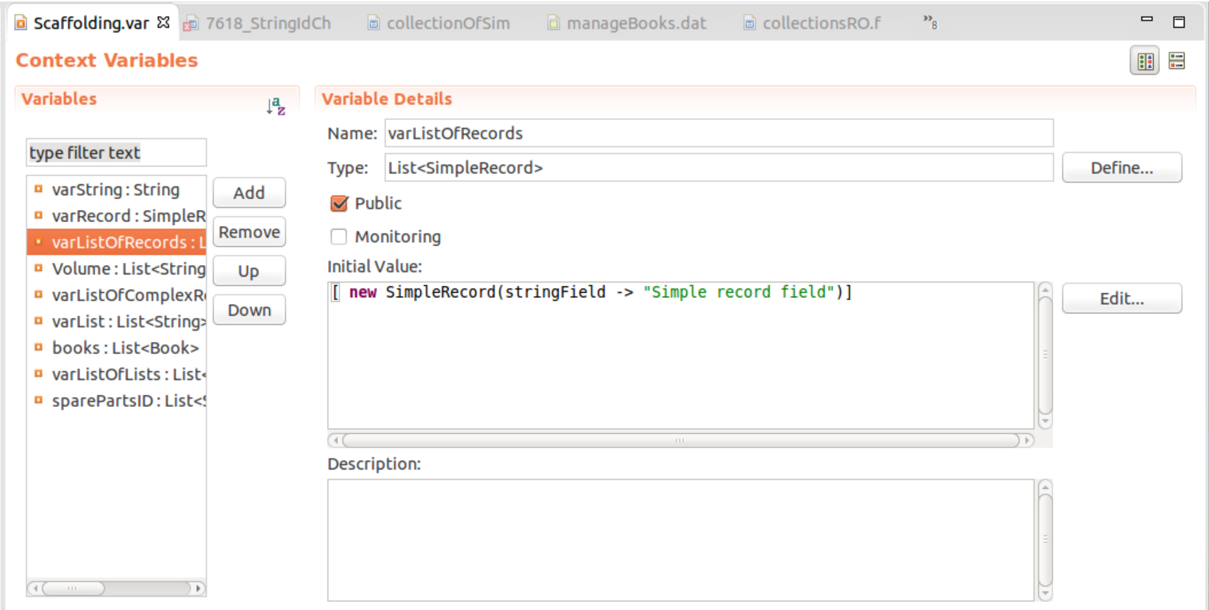


Figure 2.58 Editing variable definitions

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.

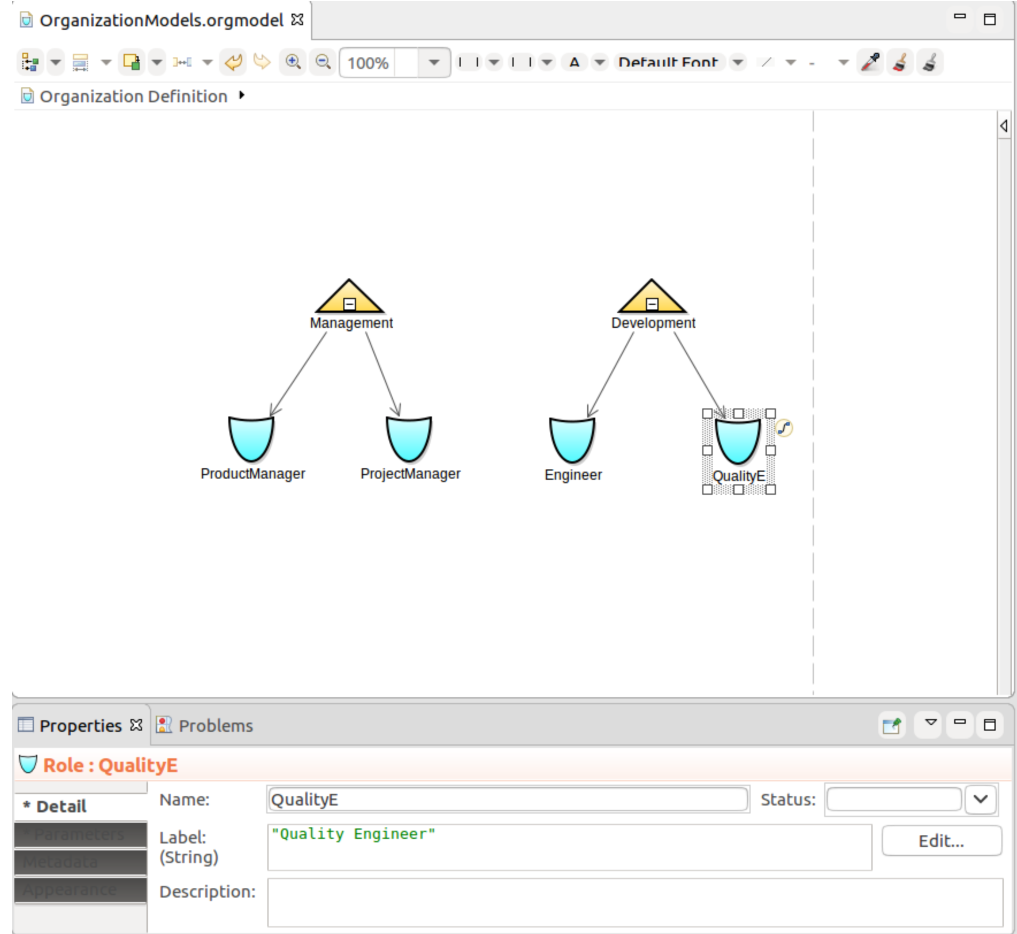


Figure 2.59 Editing the Main diagram of an organization definition file

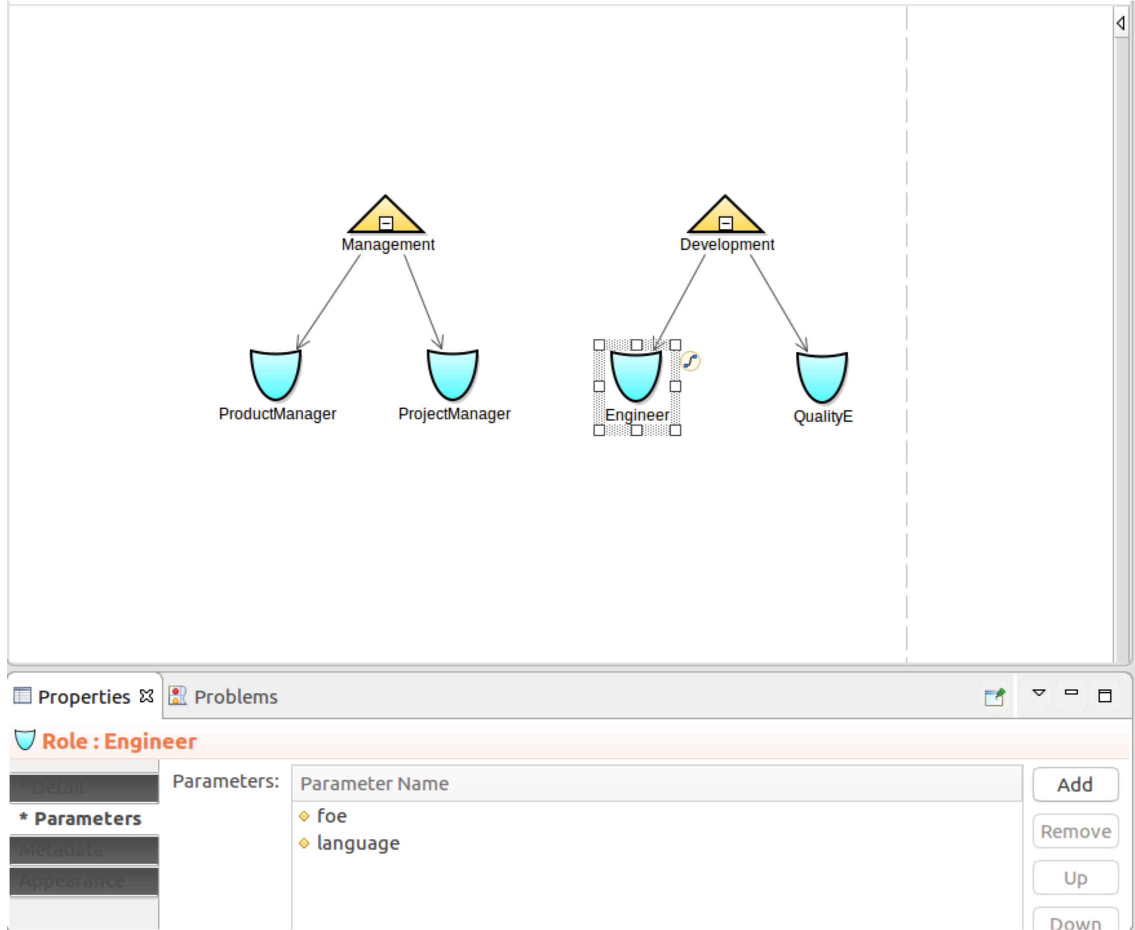
3. For units and roles, optionally define parameters in their properties.

- On the Detail tab consider defining the label, a String that is used to represents the Unit or Role

  Labels typically hold a human-readable name of a Unit or Role, for example, for a role `QE`, you could define the label `Quality Engineer`. To display the name of the role in a form, you could use the getLabel() function, for example, `QE.getLabel()`.

- Define `parameters of your units or roles` if required in the Parameters tab.

  Important: Any parameters of a unit or role must be explicitly defined also on all its child units. Otherwise, the parameters will not be available on runtime for the child elements and management of roles might be more cumbersome.



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.

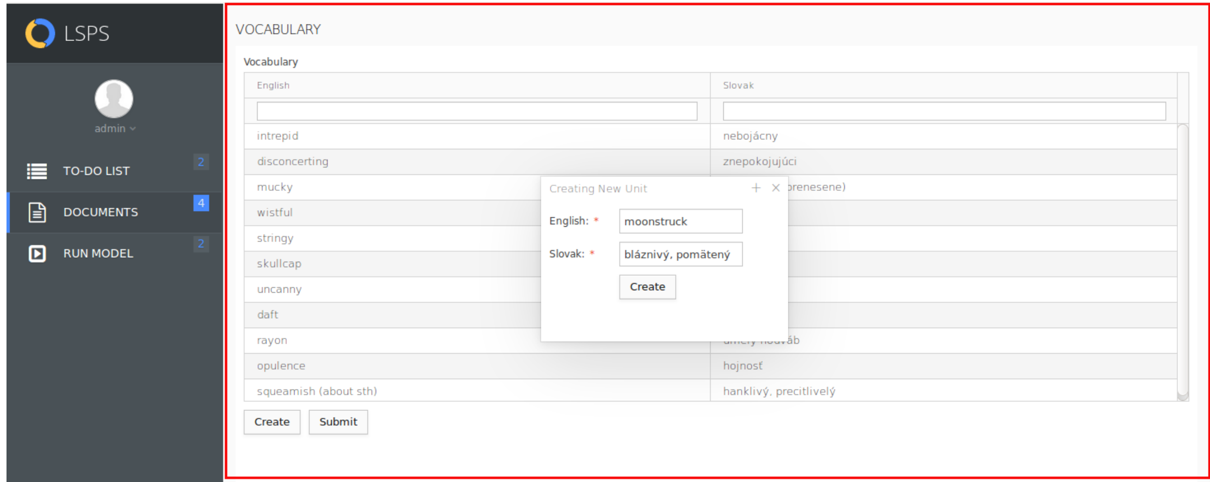


Figure 2.60 Form with a table in a document

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 the 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 an instance of
//shared record 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.

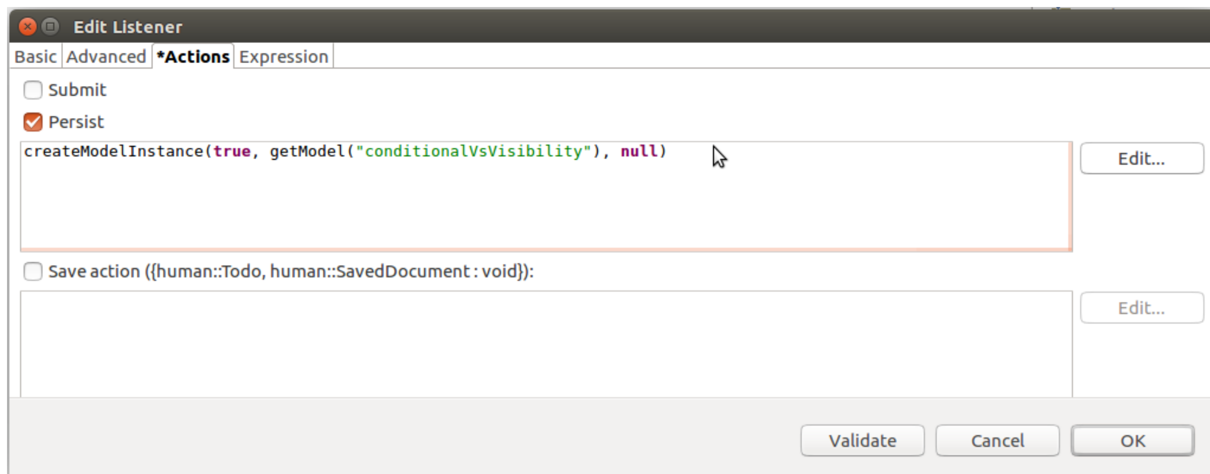
Example Navigation Factory

```
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

A *form definition* contains one form, which defines the content displayed in the front-end application (*Application User Interface*). It can be displayed as the content of a document or a to-do.

You can learn more about forms in the `Forms Guide`.

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 the `Developing a Custom Application` 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.

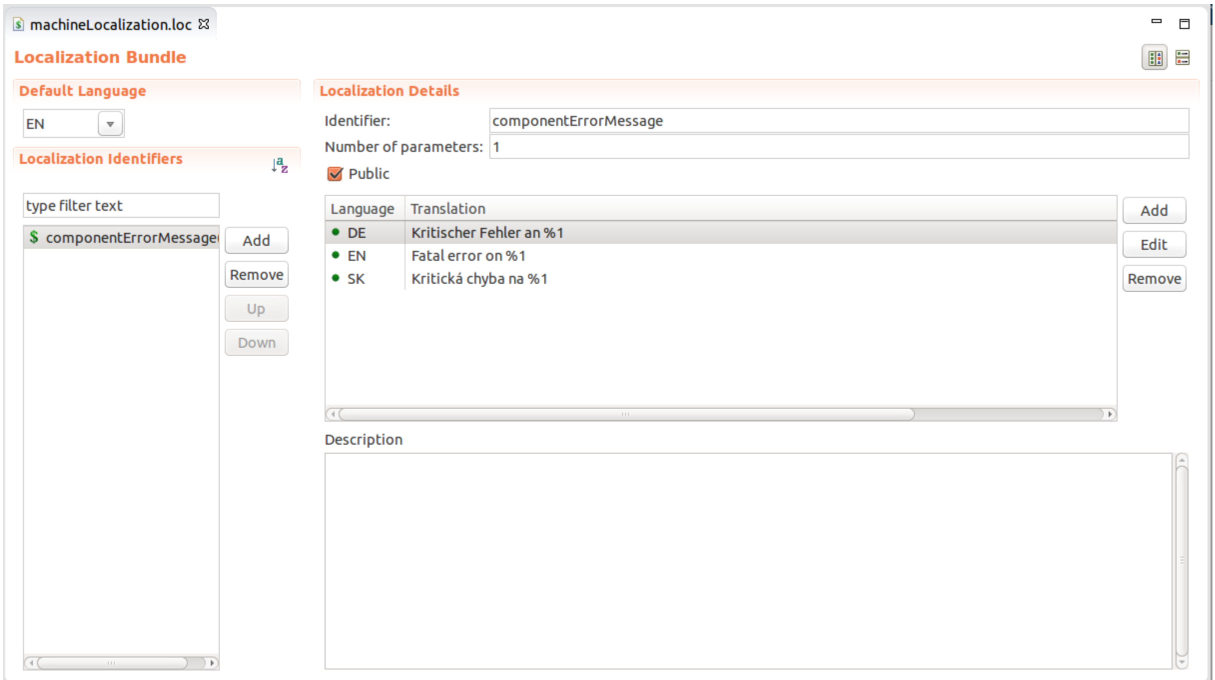


Figure 2.61 Identifier with parameter

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.

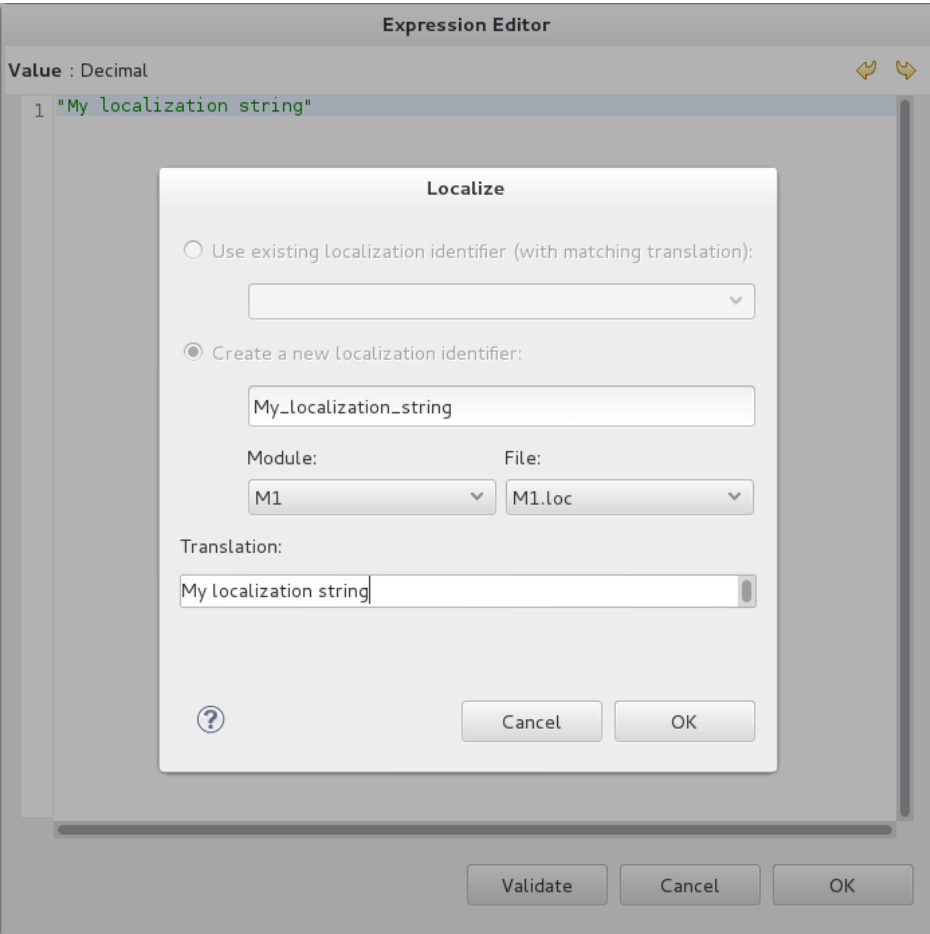


Figure 2.62 Localization dialog box with an identifier definition

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", sectorA)`.

To force a localization in a locale, call the `localize()` function, for example:

```
localize($save_button())
```

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 *Designer* > *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 Function

LSPS comes with many useful functions in its `Standard Library` as well as other libraries. However, if no appropriate function is available, you can implement your own functions either in the Expression Language or in Java:

- `Functions in the Expression Language` are created directly in the function definition file.
- `Functions with Java implementation` require that the implementation is integrated in the Application User Interface. Therefore, you will need Designer and LSPS Repository installed and set up so you can generate and modify the LSPS application. Detailed instructions are available `here`.

2.9.1 Calling a Function

Functions are called by their name followed by their parameters:

```
<FUNCTION_NAME>(<FUNCTION_PARAM1>, <FUNCTION_PARAM2>)
```

2.10 Queries

Queries serve to request data from the underlying LSPS database or an external database.

> **Important:** It is not possible to query system shared Records, such as, human::Person, human::↩
> SavedDocuments, etc. Use functions of the Standard Library to acquire their values.

2.10.1 Defining Queries

To define a query, do the following:

1. Create a query definition file in your Module: right-click the Module, go to **New** > **Query Definition**.
2. In the query editor, click the add button for the type of query you want to create:
   - standard queries defined in the Expression Language
   - HQL queries defined in HQL
   - native queries defined as database statements

2.10.2 Calling Queries

A query is called in the same way as a function: `<query_name>(<parameters>)`.

2.10.3 Standard Queries

A *standard query* returns shared Records from the database. Its properties are defined in the Expression Language, which makes it simple to create a query; however, for more complex queries, it might not be a suitable option (consider using either HQL or native queries).

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. The result is not influenced by the start index and result size properties.

  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.

2.10.3.1 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.

2.10.3.1.1 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.

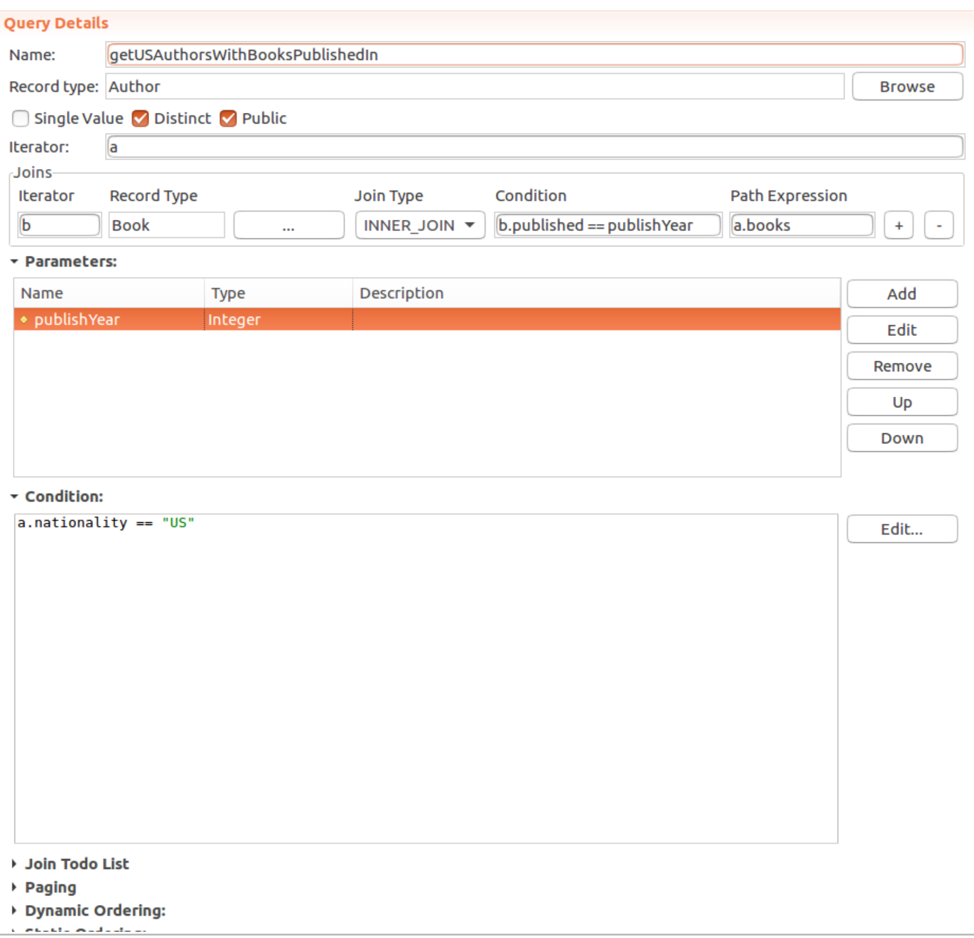
You can use joins on Records that are connected to the returned Record with a Data Relationship. Make sure, the end pointing to the related Record is named. Note that joining on a query does not fetch the joined table: the query still returns only the Records of the query return type.

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.



2.10.3.2 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.

> **Note:** Mind that the ordering is governed primarily by your database collation setting. Hence if you require change of ordering, such as, change of ordering characters with diacritics, change your database settings. The database of the embedded server uses utf8 character set and collation.

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.

2.10.3.2.1 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 2.63 Dynamic ordering in a query

2.10.3.2.2 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)

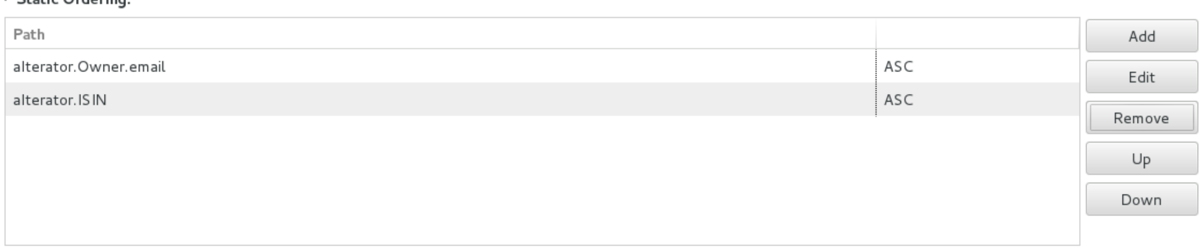


Figure 2.64 Static ordering query

2.10.3.3 Generating Queries for Shared Records

The mechanism for generating queries creates *standard queries* for shared Records of the Module. Note that to get all shared records, you can use the `findAll(<RECORD_TYPE>)` function of the Standard Library.

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.

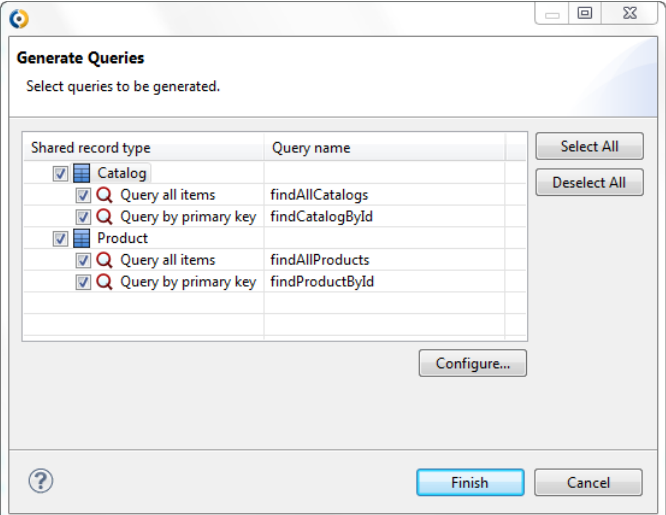


Figure 2.65 Generating queries

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**.

2.10.4 HQL Queries

HQL queries are defined in the Hibernate's HQL language extension for LSPS-related features and are intended for cases where standard queries do not suffice, for example, when you want to group or subselect in a query.

> **Note:** The underlying Hibernate is in version 4.2.19

HQL queries have the following restrictions:

- A shared record or field is accessed as in the Expression Language, that is, with its name and the dot operator to navigate.
- The query must return one or multiple shared records.
- To access properties, the query must always use an HQL alias.
- The query must contain the `select` clause;

An HQL query defines the following properties:

- **Name**: name used to call the query
- **Result type**: primitive data type, such as a String, or a non-shared record of the return value
- **Single value**: whether the query is required to return a single value (If selected and the call returns multiple values, it fails.)
- **Public**: availability of the query in importing Modules
- **Parameter**: query parameters
- **Query**: expression resolved to a String with the HQL query
- **Start index**: index of the first entity returned in a result
- **Result size**: size of the batch request
- **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. The count value is not influenced by the start index and result size properties. Note that HQL count queries do not support queries with `group by`; for example, an HQL count query for the query `select author from Author as author left join author.books group by author` will fail: To create such count queries, create them as separate HQL queries.

  Important: Single-to-single relationships (1:1) translate to Hibernate as 1:N relationships. If your HQL query is using such a relationship make sure you use the correct syntax. For example if you have a named 1:1 relationship *toB* from record A to B, using `A.toB` in such queries as `select a from A a where a.toB.id = 2` will fail. Use an appropriate join to resolve the problem, for example, `select a from A a inner join a.toB owner where owner.id = 2`.

Here are a few HQL query string examples:

- get all records of type `Author`:

  ```
  "select author from Author author"
  ```

- get all records with a field matching a parameter:

  ```
  "select
     author from Author as author
     where author.id = :id"
  ```

- get all records with a value in a related record:

  ```
  "select b
     from Book b
     join b.authors as a
     where a.surname = :authorSurname"
  ```

- get records with a field that is empty:

```
"select book from Book book where book.quote != null"
```

- get records with a field that contains a string:

```
"select author from Author as author
 where authorBook.title like concat('%', :filter, '%')"
```

- get records with a value in a related record:

```
"select author from Author as author
 left join author.books as authorBook
 where authorBook.title like :bookParam"
```

- get addresses of authors:

```
//note the property path in select:
"select author.address from Author author"
```

- get subrecords:

```
 //The record Communication has the subrecord Comment.
//select all Communications, which are of the soc::Comment type:
select c from Communication c
where c.class = 'soc::Comment'
```

2.10.5 Native Queries

A *native query* is defined as a 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; 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
  - 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.

2.11 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.

Records resemble OOP classes:

- The structure of a Record is defined by Record Fields with every Field defining its data type, be it a built-in data type or a Record.
- Records can inherit from each other.
- Behavior of Records is defined in methods in methods definition files.
- Records can be defined as and implement Interfaces.

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
   - **Public**: record is accessible from importing modules
   - **Interfaces**: comma-separate list of realized `interfaces`
   - **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
- **Monitoring**: displays an icon to indicate that the Record is involved in `Monitoring`
- **Deprecated**: flag to denote a deprecated Record

  If the attribute is true, the validation displays an information message that the Record is deprecated.
- **Label**: a String that is assigned to the record

  Labels typically hold a human-readable name of the record, relationship end or enumeration, for example, for a Record `NaturalPerson`, you could define the label `natural person`. To display the type of a NaturalPerson record in a form, you could use the getLabel() function, for example, `myPerson.getType().getLabel()`. You can use the Label also for API calls.

4. Optionally, define XML Mapping.

Now you can insert fields into the record and define relationships and their properties if applicable.

2.11.2 Creating a Record Field

To create a record field, do the following:

1. On the canvas, select the record and press the Insert key to add a new field.
2. Select the field and define its properties in the Properties view.
   - **Name**: name unique within the Record
   - **Type**: data type of the record field

     Either enter the type directly, for example, Set<Map<Integer, String>> or click the **Define** button to use the wizard.

     > **Important:** It is not recommended to use fields of complex data types. Also note that fields of collections with shared records in shared records are **not supported**: serialization of such fields will result in a *DatabaseError* exception. Use Relationships to associate a record with another record or a collection of records.
   - **Visibility**:
     - public: marked with +; field is accessible from anywhere
     - protected: marked with #; accessible only from within the given data type hierarchy (can be used by any subtypes)
     - private: marked with -; accessible only from the record and its methods (define methods to access from outside)

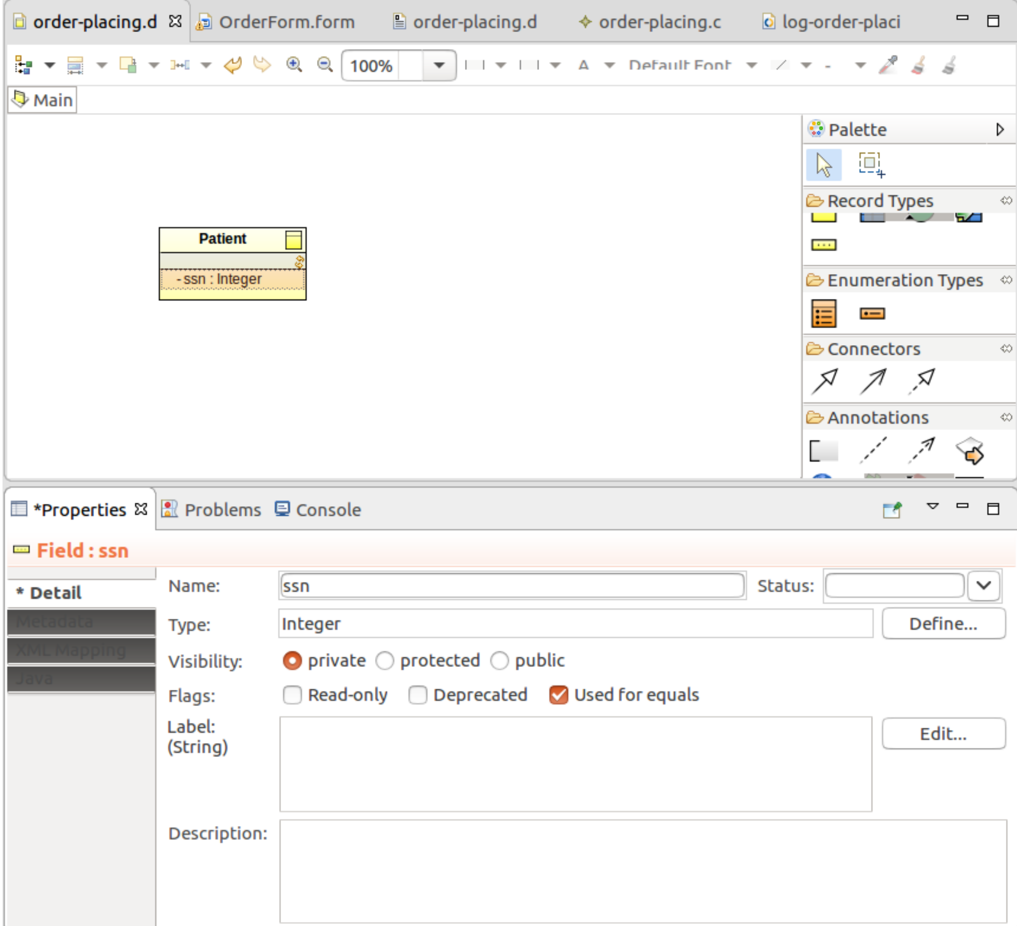


Figure 2.66 Field with its properties

- **Label**: a String that is used to represents the Record Field

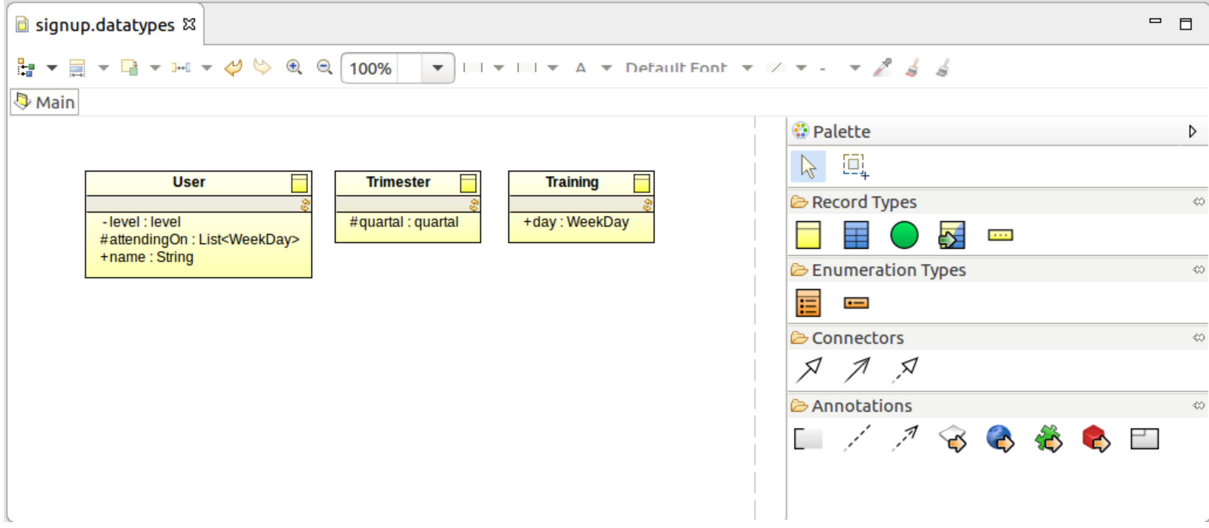
  Labels typically hold a human-readable name of a Record Field, for example, for an Enumeration literal `Weekday.MONDAY`, you could define label `Monday`. To get the label value for the enumeration literal, you will use the `Weekday.MONDAY.getLabel()`.

- **Read-only**: if true, the field is read-only (instantiated on model instance instantiation; not modifiable after that)

- **Deprecated**: flag that denotes 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



2.11.3 Defining a Method of a Record

To define methods of a Record, do the following:

1. Open or create a method definition file for the Record:
   - To create a new method definition, right-click the Record and select *New Methods Definition*.
   - To open an existing definition, double-click the methods files in the GO-BPMN Explorer.
2. Add or edit the methods following the `method syntax`.

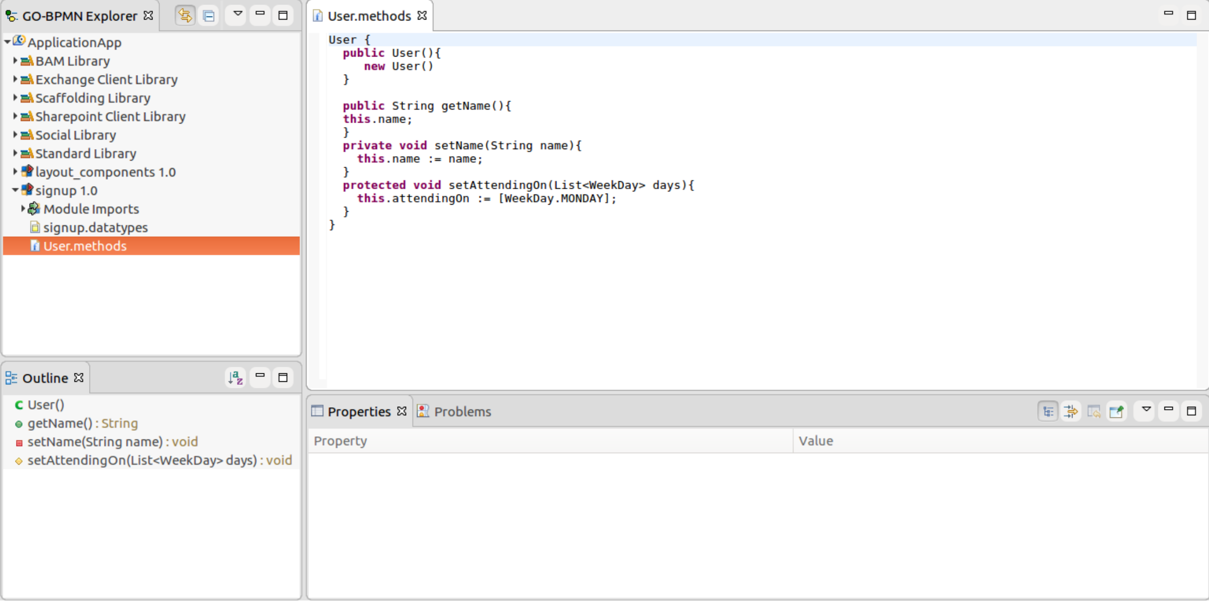


Figure 2.67 Method definitions: constructor and methods with different visibility

2.11.4 Defining a Subtype of a Record

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`.

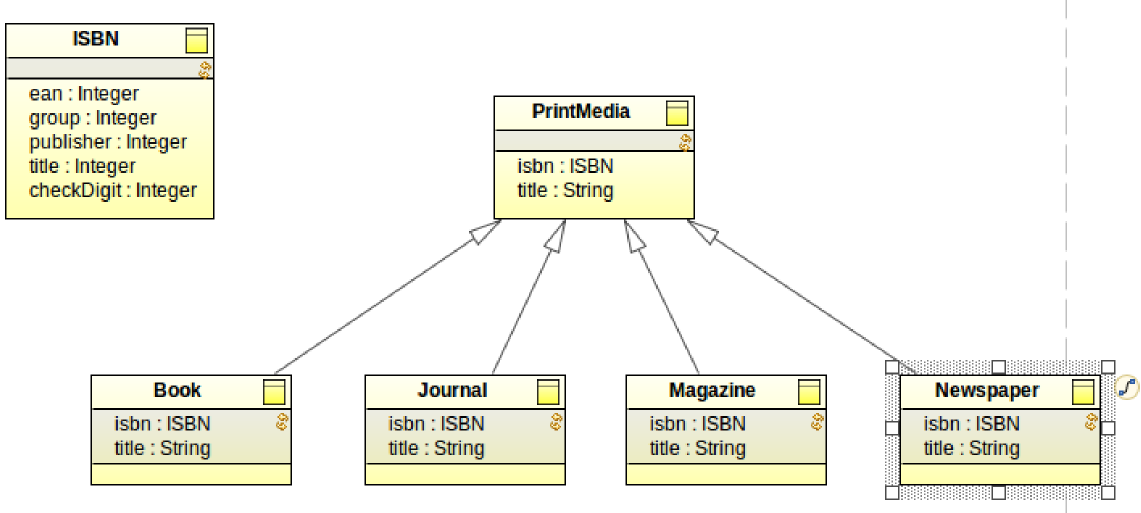


Figure 2.68 Hierarchy of Records with multiple subtypes

2.11.5 Creating a Relationship Between Records

A relationship establishes a logical connection between records.

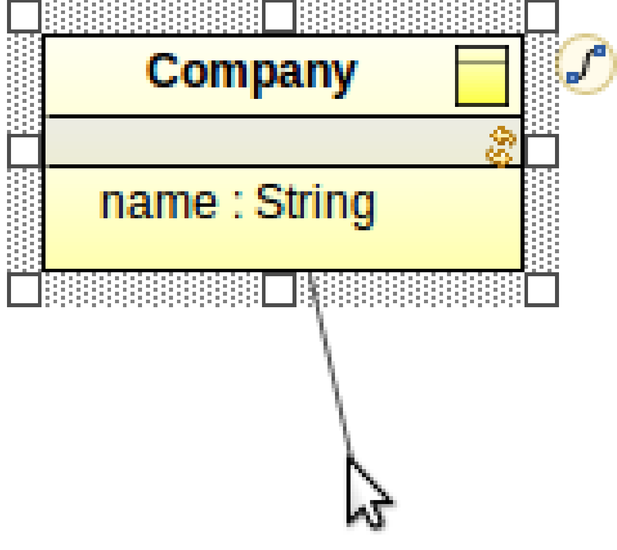
Record relationships are symmetrical: they are set on both navigation ends of the relationship. Therefore it is not necessary to make one of the record the owner of the relationship.

Information on the mechanisms, related elements, comparing of records and record properties is available in the `GO-BPMN Modeling Language Guide`.

> **Important:** It is *not recommended to create a relationships from a shared record to a common record* since the target non-shared record is stored as BLOB. This can cause problems when refactoring and database migration; for example, when moving the record to another module since this changes the fully qualified name.

To create a relationship between two records, do the following:

1. In the Data Type Editor, click the quicklinker icon and pull the link toward the target Record.



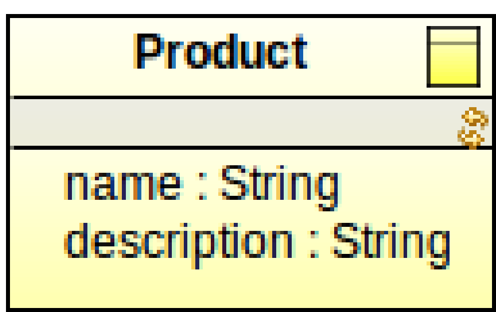


Figure 2.69 Creating Record Relationship

The navigation ends represent the Target and Source ends. Which is which depends from 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 its Properties view.

2. Select the relationship and open its Properties view:

   (a) Define its general properties on the Detail tabs.

   (b) Define the properties of the navigation ends on the Target and Source tabs:

      - **Name**: name of the Relationship end used to navigate through to the target record
      - **Visibility**: visibility of the navigation
        - public: accessible from anywhere including importing modules
        - protected: accessible only from within the data type hierarchy (can be used by any subtypes)
        - private: accessible only from the record and its methods (define methods to access from outside)
      - **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
      - **Label**: a String that is assigned to the record
        Labels typically hold a human-readable name of the record, relationship end or enumeration; for example, for a Record `NaturalPerson`, you could define the label `natural person`. To display the type of a NaturalPerson record in a form, you could use the getLabel() function, for example, `myPerson.getType().getLabel()`. You can use the Label also for API calls.

   (c) Define database-related properties of the relationship ends on the *Target DB Mapping* and *Source DB Mapping* tabs.

- **Lazy**: whether loading of the related 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.

(d) On relationships with one or two shared records, define the additional properties.

2.11.5.1 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.

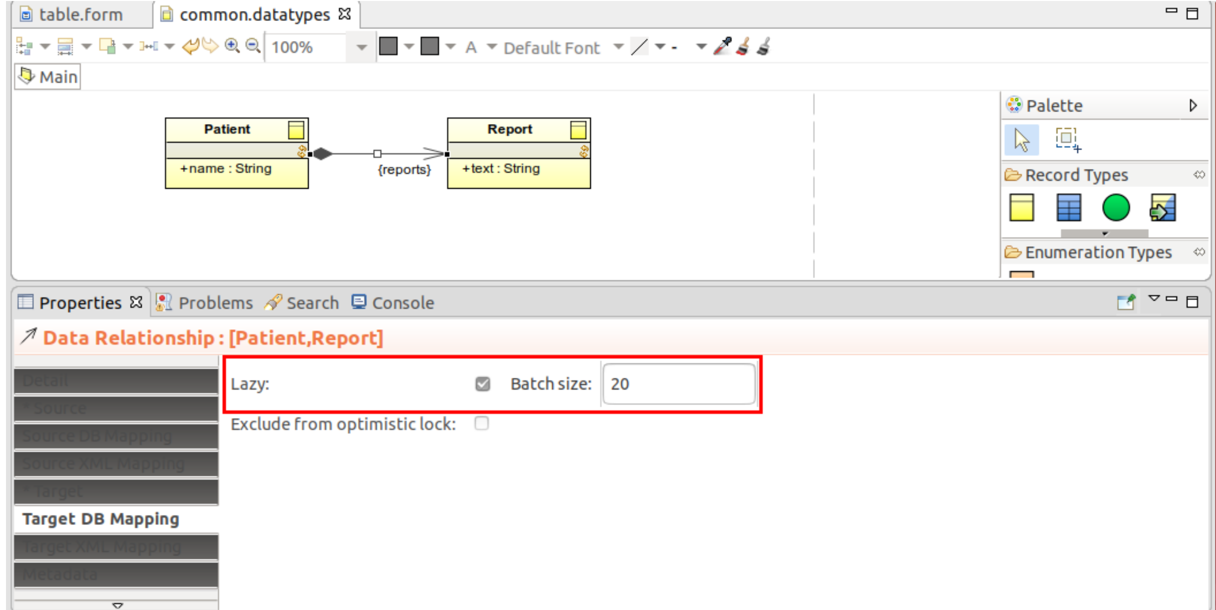


Figure 2.70 Lazy fetching configuration

2.11.6 Using Records from Other Definitions

To display an element from another data type definition file, possibly from a definition file in an imported Module, use the Record Import element, which is available in the palette of the editor.

2.11.7 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 end*, 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.

2.11.8 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`

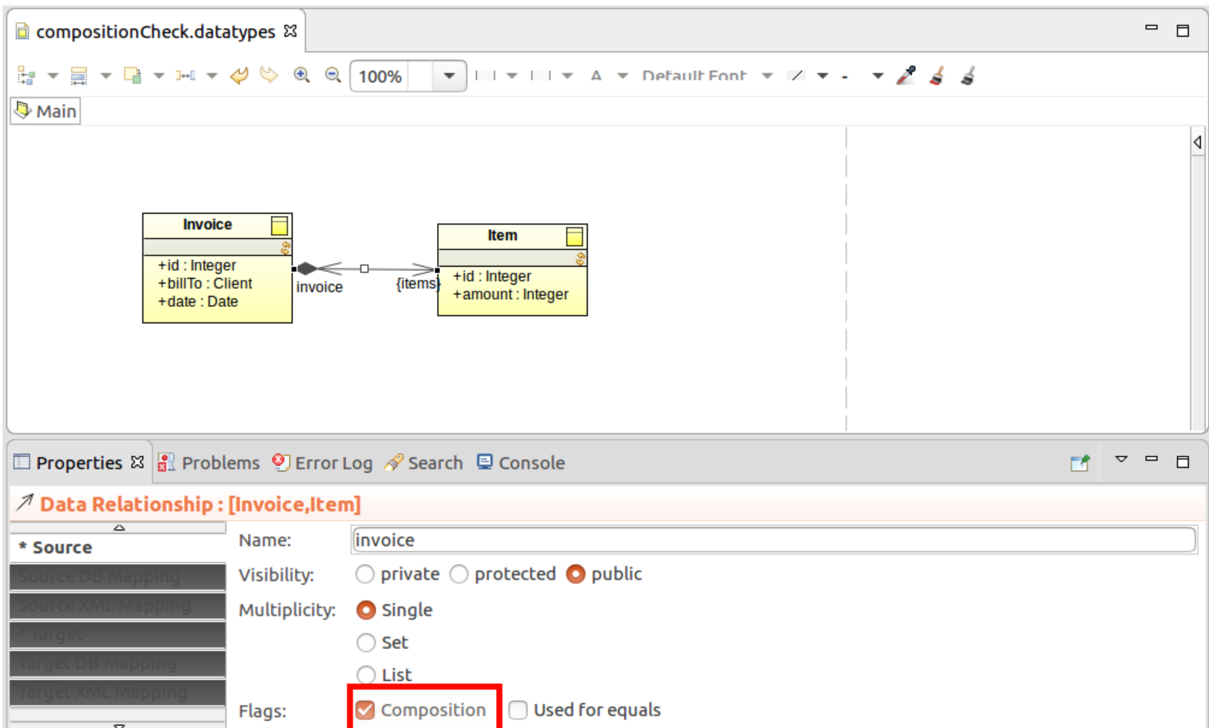


Figure 2.71 Composition setting

2.11.9 Creating an Interface

An *Interface* is defined as a special type of Record with no Fields or Associations. Records that implement the Interface are connected to it with a Realization connector.

To create an Interface and its implementations, do the following:

1. Create the Interface Record:
   (a) Create or open a Record that represents the Interface.
   (b) In the Properties view of the Record, select the *Interface* checkbox.
   (c) Create the methods file with the Interface methods: Right-click the Interface and click *New Methods File*.

```
    Nameable {

  public String getName();
  public void setName(String name);
}
```

2. To create a Record that implements the Interface, do the following:
   (a) Create the Record and connect it with its Interface with the Realization connector.
   (b) Right-click the Record and select *New Methods Definition*.
   (c) Implement the Interface methods for the Record.

Note: You can get to the methods file from the Interface Record by pressing Ctrl and double-clicking the method in the Record diagram elements (to display the methods in your Record, right-click the Record, go to *Compartments* and select *Methods*).

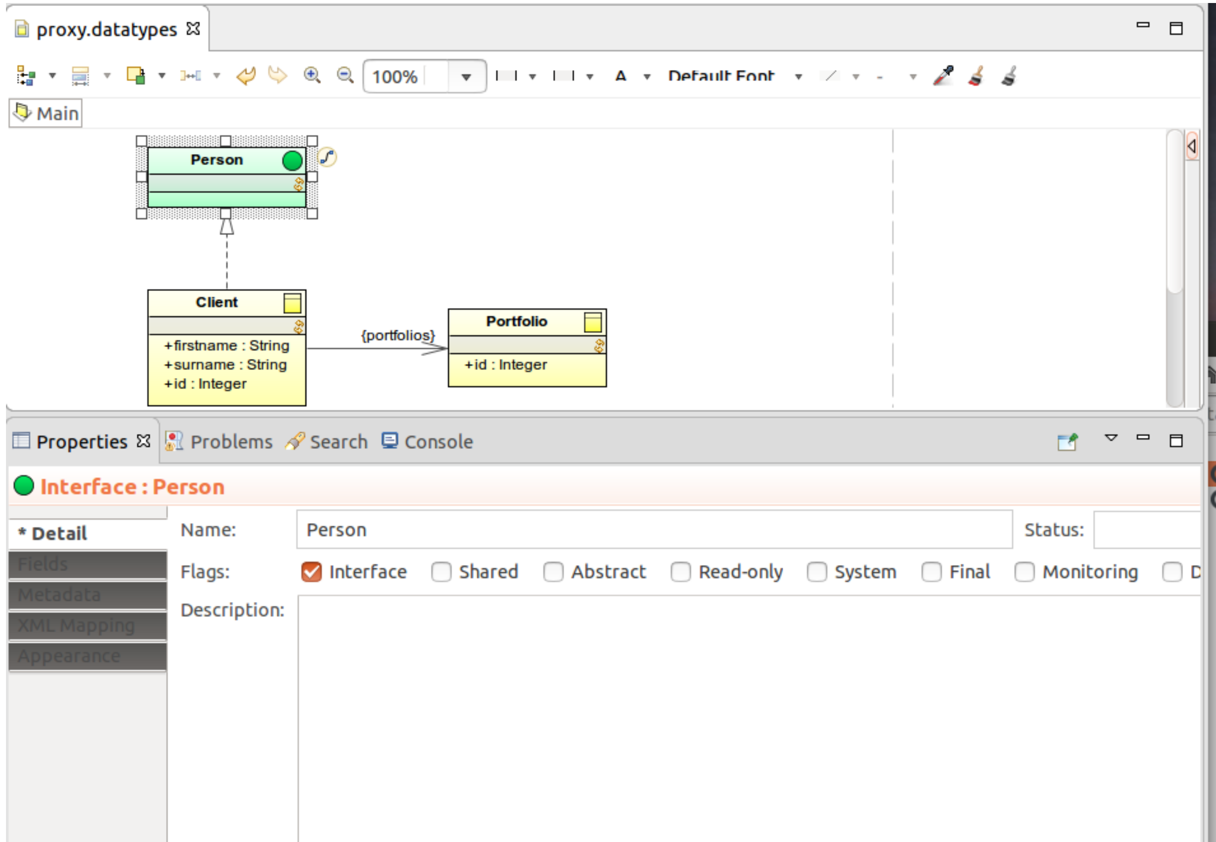


Figure 2.72 Interface with an implementation

2.11.10 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.

2.11.10.1 Defining Fields for 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.

2.11.10.2 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.

2.11.11 Presentation of Record Diagrams

2.11.11.1 Displaying and Hiding Record Compartments

To display or hide additional compartments in Record views, such as, their methods,inherited methods, Fields and inherited Fields, right-click the Record view in the Diagram, go to **Compartments** and select or unselect the compartments.

You can then click the method compartment to open its declaration.

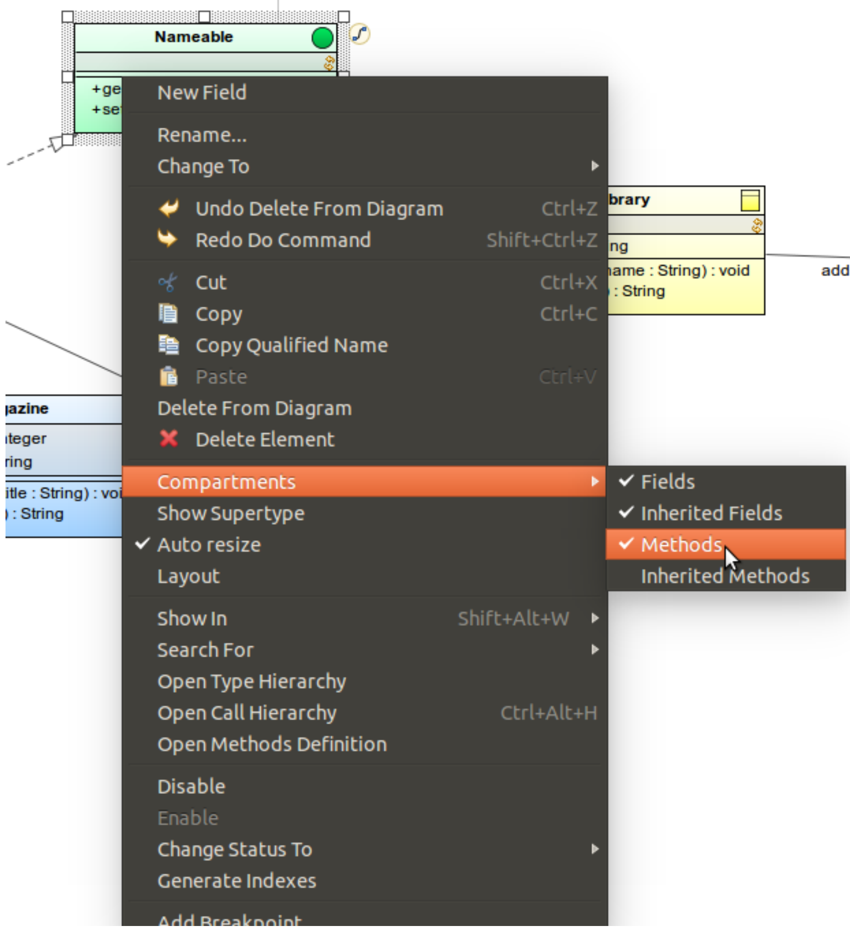


Figure 2.73 Available Record compartments

2.11.11.2 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**.

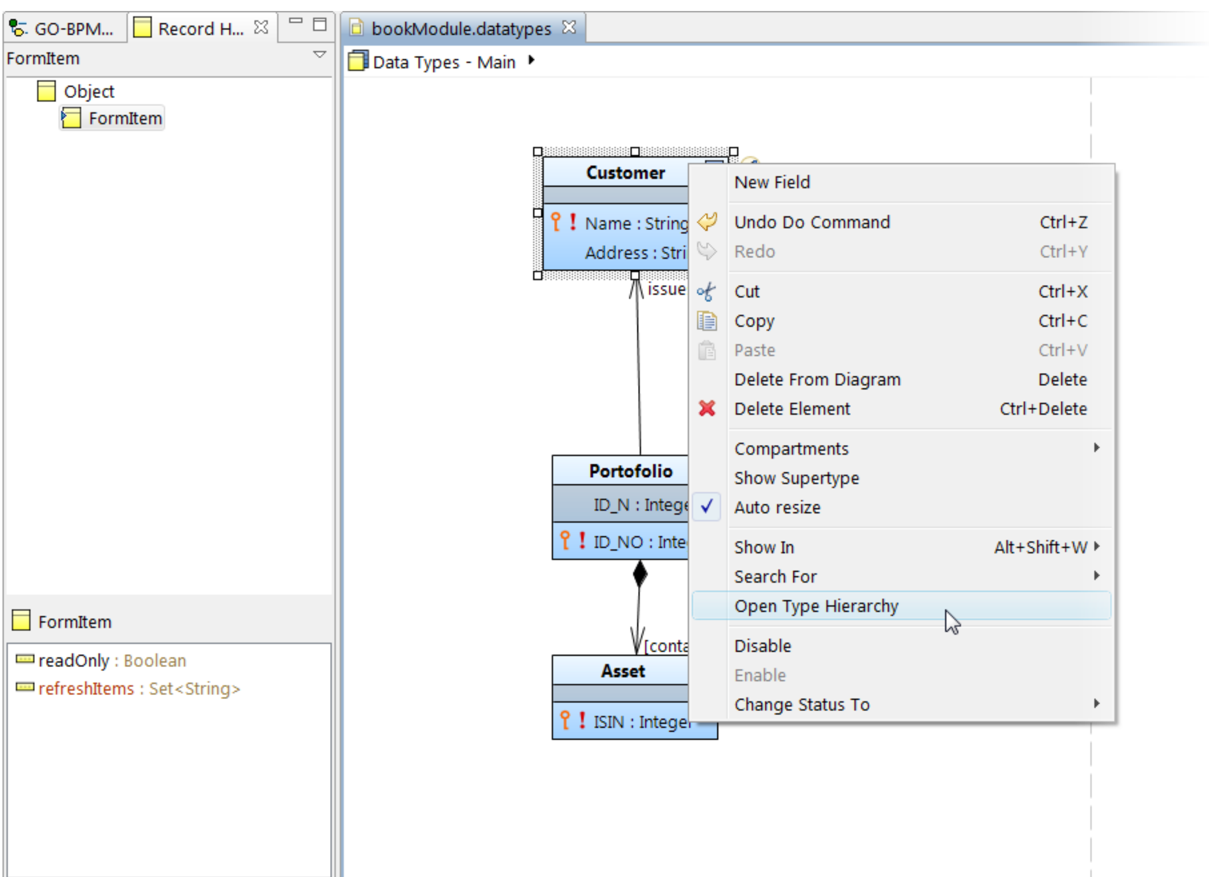


Figure 2.74 Displaying a data type in the Record Hierarchy view (using the context menu) with the Record Hierarchy view on the left

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

2.11.12 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.

2.11.13 Record Validation

To check if the values of record properties meet some criteria, use validation with constraints. Constraint validation checks a value of a record property against any constraints defined for the property. Constraints define the conditions the value must meet.

The conditions are defined by the constraint type of the given constraint: The constraint type defines the semantics of the validation—for the constraint to be met, the value of the record or property must meet the condition defined by the constraint type.

2.11.13.1 Validating a Record

You can check if a value of a record property meets the underlying constraints by calling the `validate()` function. The call returns a list of violations with information on which field violates the given constraint and a violation message.

You can check the constraints defined either for a property or for the entire record:

- When checking constraints of a property, all constraints defined for the record field are checked.
- When checking constraints of a record, all constraints on the record's fields, the record itself, and its super types are checked and that in this order.

To provide additional granularity to validation on runtime, validation can pass tags to refine the validation.

```
def Book b := new Book(isbn -> "123");
validate(record -> b, property -> Book.isbn, tags -> {onCreate(),stock()}, context -> null)
//returns a list with the constraint violations of the isbn value
//that have the tag onCreate or stock
```

Instead of tags, the *validate()* call can pass the *context* parameter, which is an object of type Map<String, Object> (either context or tags can be passed).

> **Note:** When validating record data that is displayed in a form component (it is bound to the form component), you can display violation messages in the form with the `showDataErrorMessages()` function and from a ui::form with the `showConstraintViolations()` function.

2.11.13.2 Defining a Constraint

A *constraint* defines what value of a record field or property is allowed.

It is defined for a record field and is of a particular constraint type. The constraint type 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.

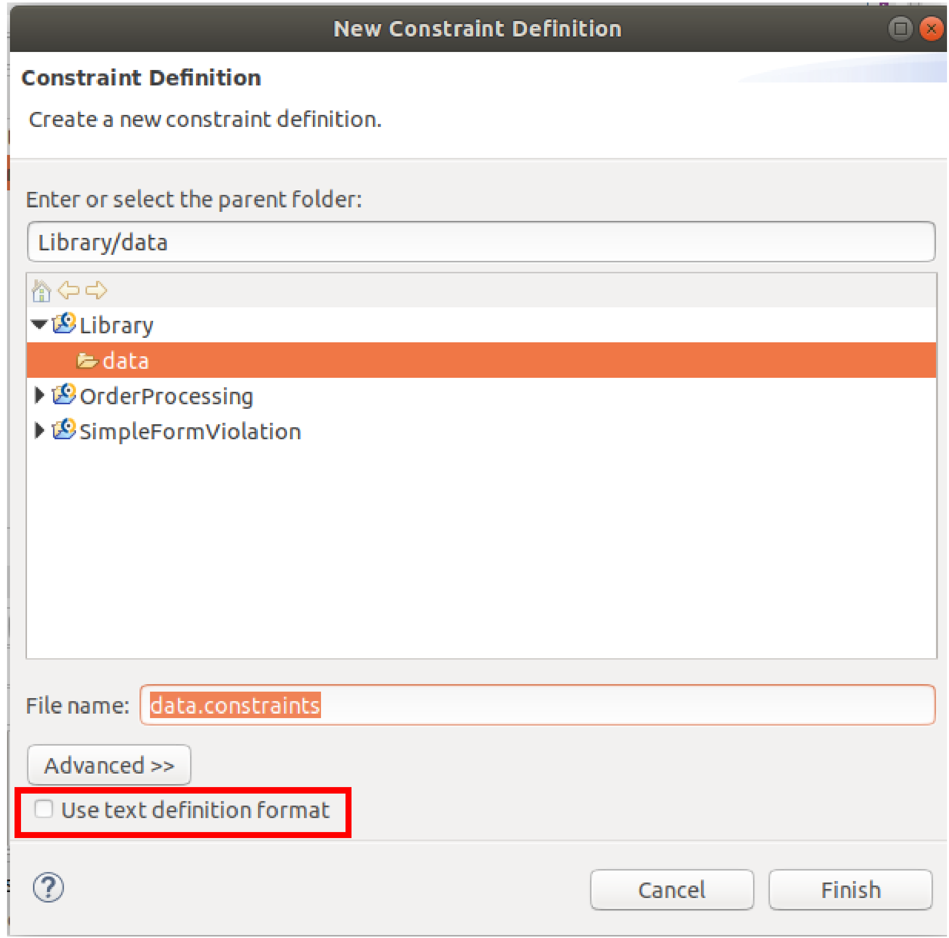
For details on constraints and their properties, refer to the `Modeling Language Guide`

You can define constraints in the gui or text editor.

2.11.13.2.1 Defining a Constraint in GUI Editor

To define a constraint for a Record or a Property in the GUI editor, do the following:

1. Create a constraint definition file: make sure that the *Use text definition format* is not selected.



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`.

constraints.constraints | constraints.tags

Constraints

Constraints

type filter text

ID	Record (property)	Constraint type	Tags
Book.isbn.Format	Book.isbn	RecordValidity()	ONAPPROVALTAG and INPU
Book.isbn.NotNull	Book.isbn	NotNull()	ONAPPROVALTAG
Book.authors	Book.authors	RecordCollectionValidity()	ONENTRYTAG
Author.name.Format	Author.name	AuthorName()	

Add | Remove | Up | Down

Constraint Details

ID: Book.isbn.Format
Record (property): Book.isbn
Tags: ONAPPROVALTAG and INPUTPROCESSTAG
Constraint type: RecordValidity() | Edit...

Figure 2.75 Defining constraints

3. Use `validate()` on the record or property instances.

2.11.13.2.2 Defining a Constraint in Text Editor

To define a constraint for a Record or a Property in the GUI editor, do the following:

1. Create a constraint definition file. Make sure that the *Use text definition format* is selected.
2. In the constraint editor, enter the constraints following the syntax

```
<Annotations>
<ConstraintID> {
  <RecordOrPropertyToValidate>,
  <TagExpression>,
  <ConstraintType>
}
```

Example text definition of a constraint

```
@Meta("notification" -> "john") //metadata
@Disabled //disabled
Book.isbn.Format {
  Book.isbn, //record property
  onCreateTag, //tag
  ISBN13() //constraint type
}
```

Now you can use the `validate()` call on the record or its property to check its values.

2.11.13.3 Defining a Constraint Type

> **Note:** The Standard Library contains an extensive set of constraint types. Hence check the `available constraint types` before defining 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`.

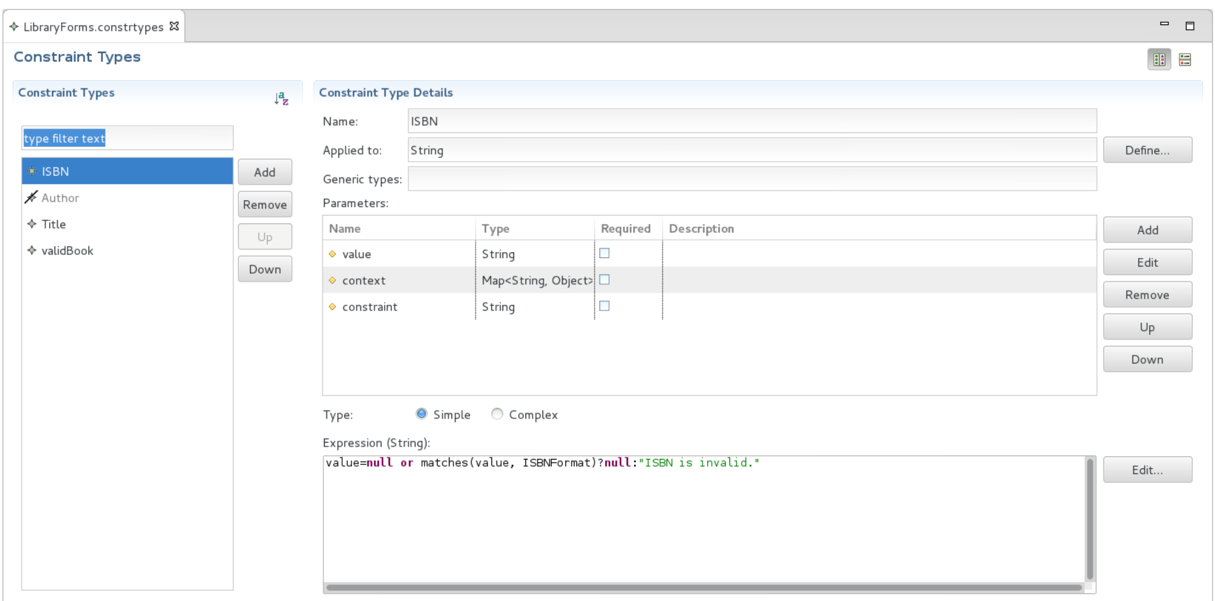


Figure 2.76 Defining constraint type

2.11.13.4 Filtering Constraints on Validation using Tags

Typically tags define the stages at which the constraint should apply, such as, `create` or `update` so that a constraint is applied only if the has one of the tags required by validation.

Note that constraints that do not define any tags are checked always when their record or property is validated.

```
def Book b := new Book(title -> null, isbn -> "123");
b.title := "1984";
//will return constraint violation defined for any of the book fields
//that have a tag list that includes the update or create tag:
validate(b, null, {update(), create()}, null)
```

To define a tag, 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.
5. Add your tags to the constraints and *validate()* calls.

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 your data, use shared records.

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. Any readings, modifications, and deletions of shared record instances are reflected in their database entry. If it is necessary to postpone persisting of changes on a shared record, you can use a change proxy object.

The data type model is applied on the database when module is uploaded to the server. The way it is applied is defined by the database schema update strategy(Connecting Designer to an LSPS Server) (similar to the `hbm2ddl` Hibernate configuration):

- When uploading from the Modeling perspective, the strategy is defined by the server connection configuration.
- When uploading from the Management perspective, the strategy is defined per `model upload`.
- You can also enable schema update per Record.

Mind that if the data type model is already used in production and cannot be dropped and recreated since it contains business data, you will need to migrate the database and potentially update running model instances before using the new data type model.

> **Important:** When you start using shared records, consider using a database-versioning tool, such as, Flyway: This will allow you to track how you change the shared records and their relationship. Failing to use a database-versioning tool, might case a non-trivial effort when updating the data model that is in production later (for details refer to the `migration instructions`).

The persistence mechanism of shared Records relies on 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 applied.

> **Note:** Variables of a shared record type must be fetched anew in each new transaction: This might cause performance issues. For further details on model transactions, refer to the `Modeling Language Guide`.

2.12.1 Defining Data Model 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

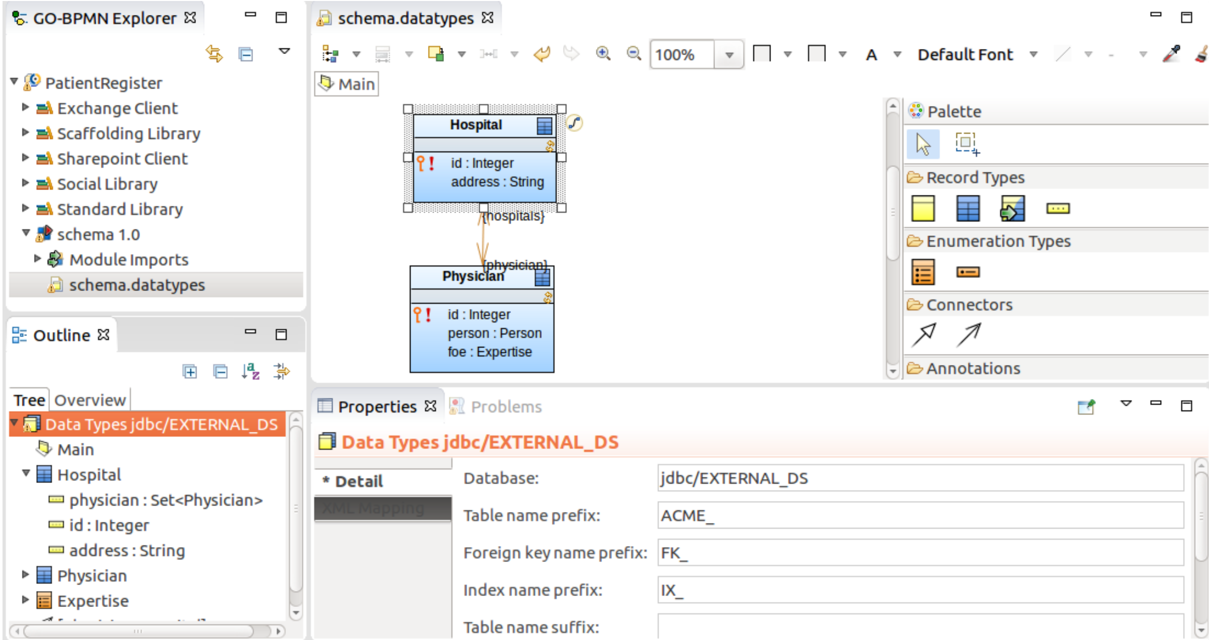


Figure 2.77 Defining database properties for a data type hierarchy

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.

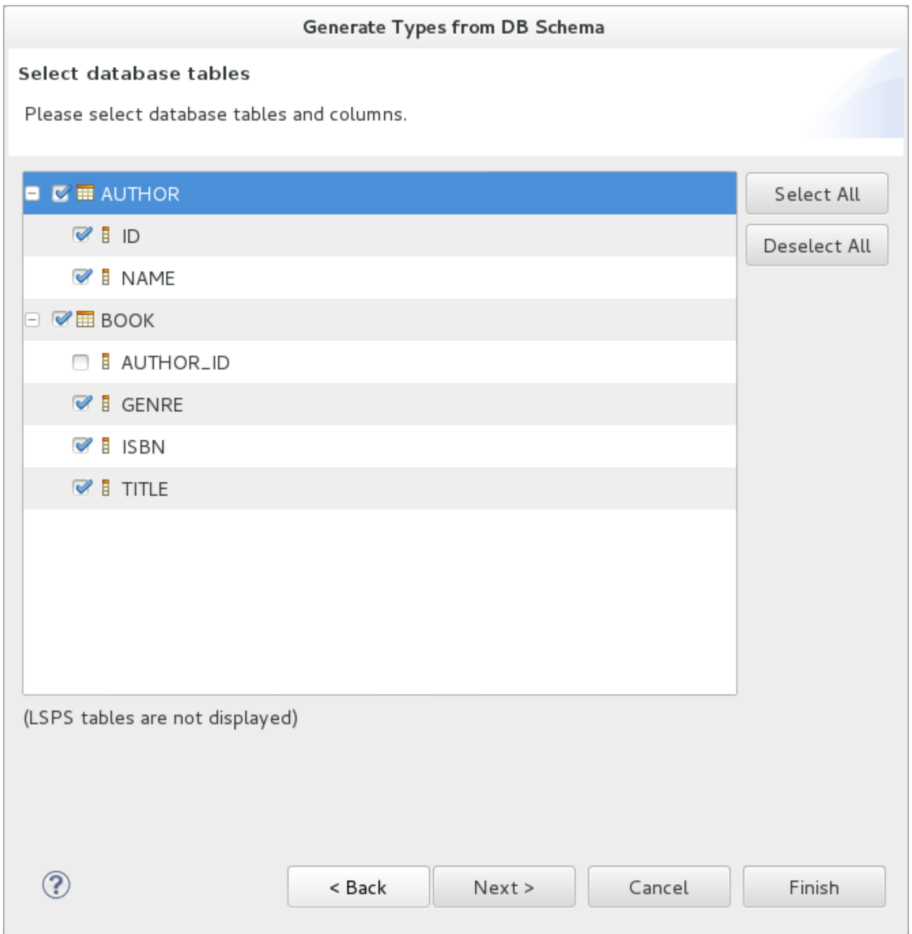


Figure 2.78 Selection of Database Entities

7. Select the target location and enter the resource name, and click **Finish**.

Note that the generated data type definition does **not** contain any Diagram views of the shared Records: Drag the shared Records from the GO-BPMN Explorer onto the data type canvas if required.

> **Important:** If the underlying database table is already mapped to an existing type but contains additional columns when you perform the type generation (the existing record type, does not contain the fields of the table columns), *no additional* fields are created on the existing type.

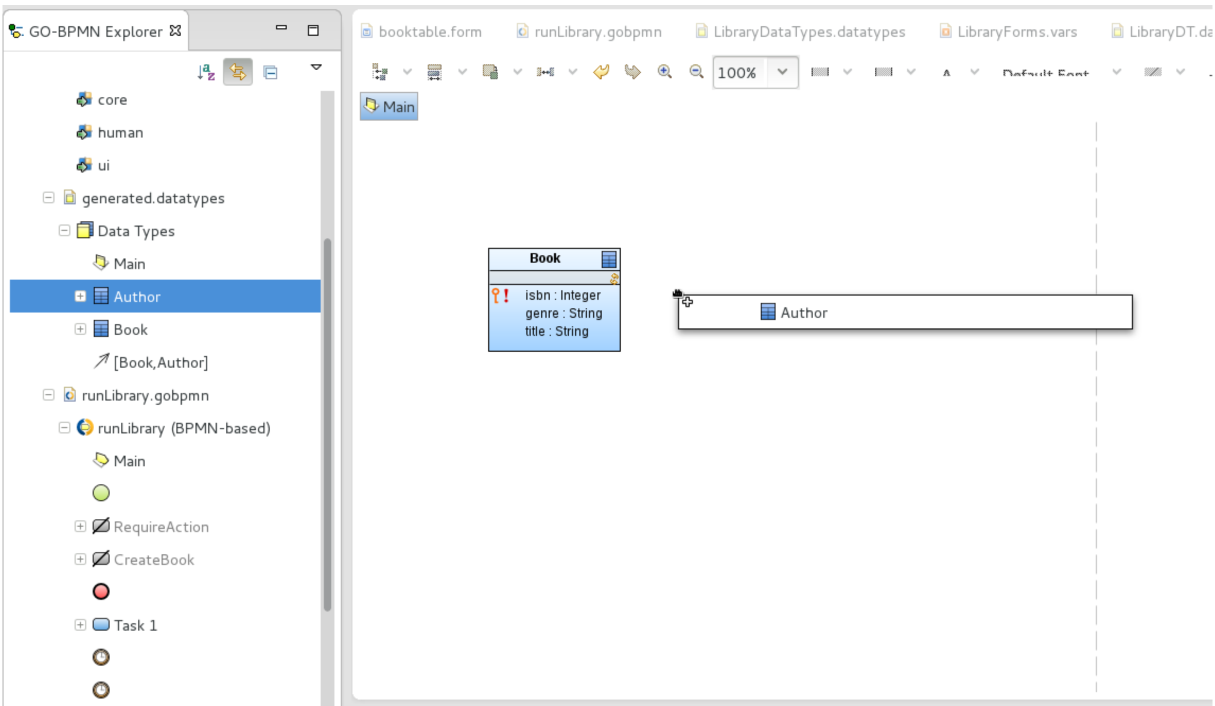


Figure 2.79 Adding a depiction of a generated shared record onto a data type diagram

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
      If the record is a child of another record, the inherited fields are stored in the table of the parent.
    - *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.
   - **BLOB length**: space reserved for the field value in bytes defined as an expression that returns an Integer, for example `1024*1024`. The expression cannot access constructs and variables from parent contexts.

     The property applies to fields of other than simple types that need to be serialized when stored in the database, such as, Records, Collections, etc.
   - **Not null**: if true, the field must not be `null`. If you try to create a record
   - **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.
   - **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.

     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.
   - **Sequence name**: the database sequence used for the auto generated field (If it does not exist, it is be created.)

2.12.5 Locking Shared Records for Changes (Optimistic Locking)

To prevent overwrite of shared record instances if the record was changed since it was loaded, use optimistic locking: The optimistic locking mechanism prevents changes to shared record instances if the record changed since it was loaded.

For example, if you save a to-do that works with a shared record and another user changes the record from another to-do and you attempt to submit the to-do with your changes, the action fails with an `Conflict on entity` exception and your transaction is rolled back unless the exception is handled.

With optimistic locking, the record instances store their version in a dedicated field: the version field is updated always when the given record instance changes. If you are changing a record instance and the stored version changed in the meantime, the server returns an exception when you try to apply your changes.

To allow some of the record fields to be changed freely without ever causing a conflict, exclude them explicitly from optimistic locking: a change on such fields does not cause an update of the record version nor does it trigger the version check.

2.12.6 Setting up Optimistic Locking on a Shared Record

To set up locking on a shared Record, create the version field on the record:

1. Add an Integer or Date field to the record; it is recommended to use the Integer type.
2. On the *DB Mapping* tab of its Properties view, select *Version*.

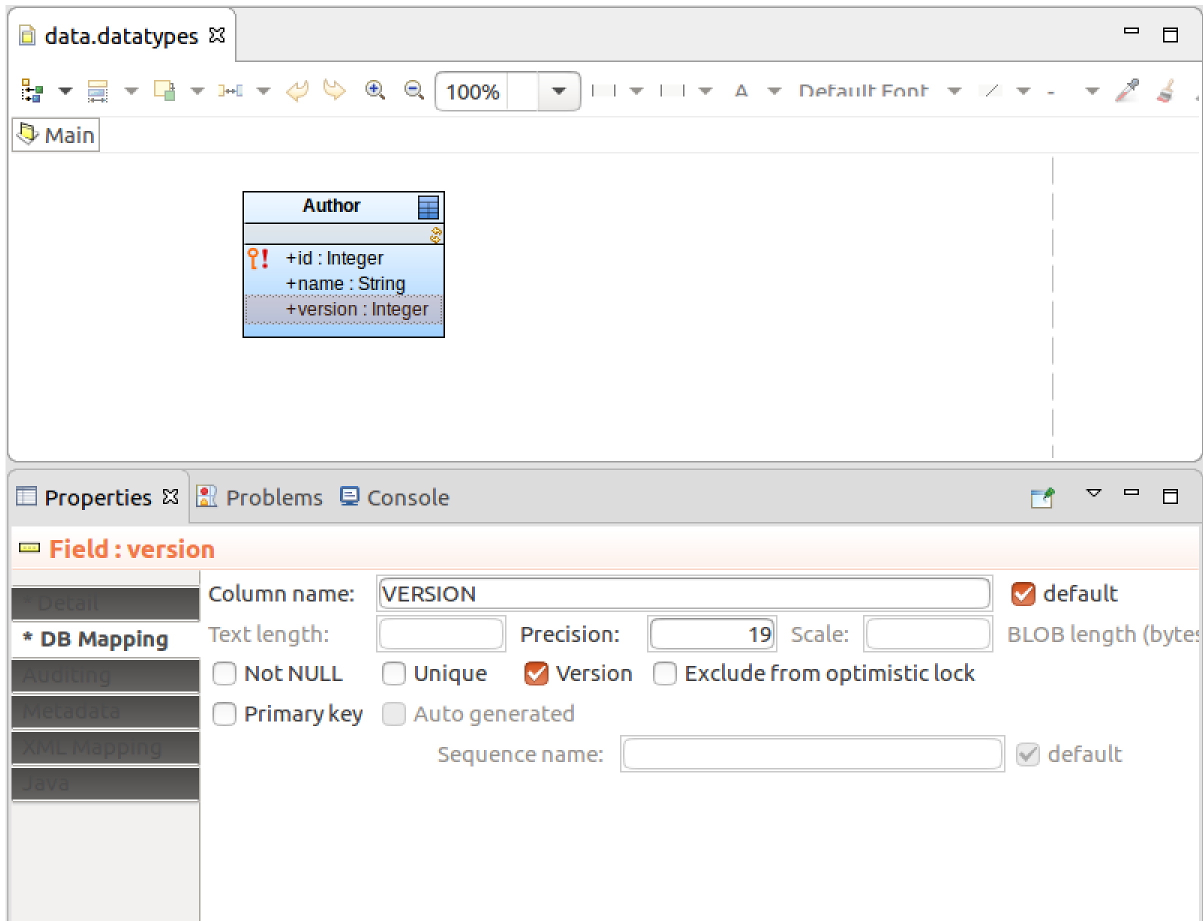


Figure 2.80 The Version field of a Shared Record

3. Optionally, exclude record 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**.

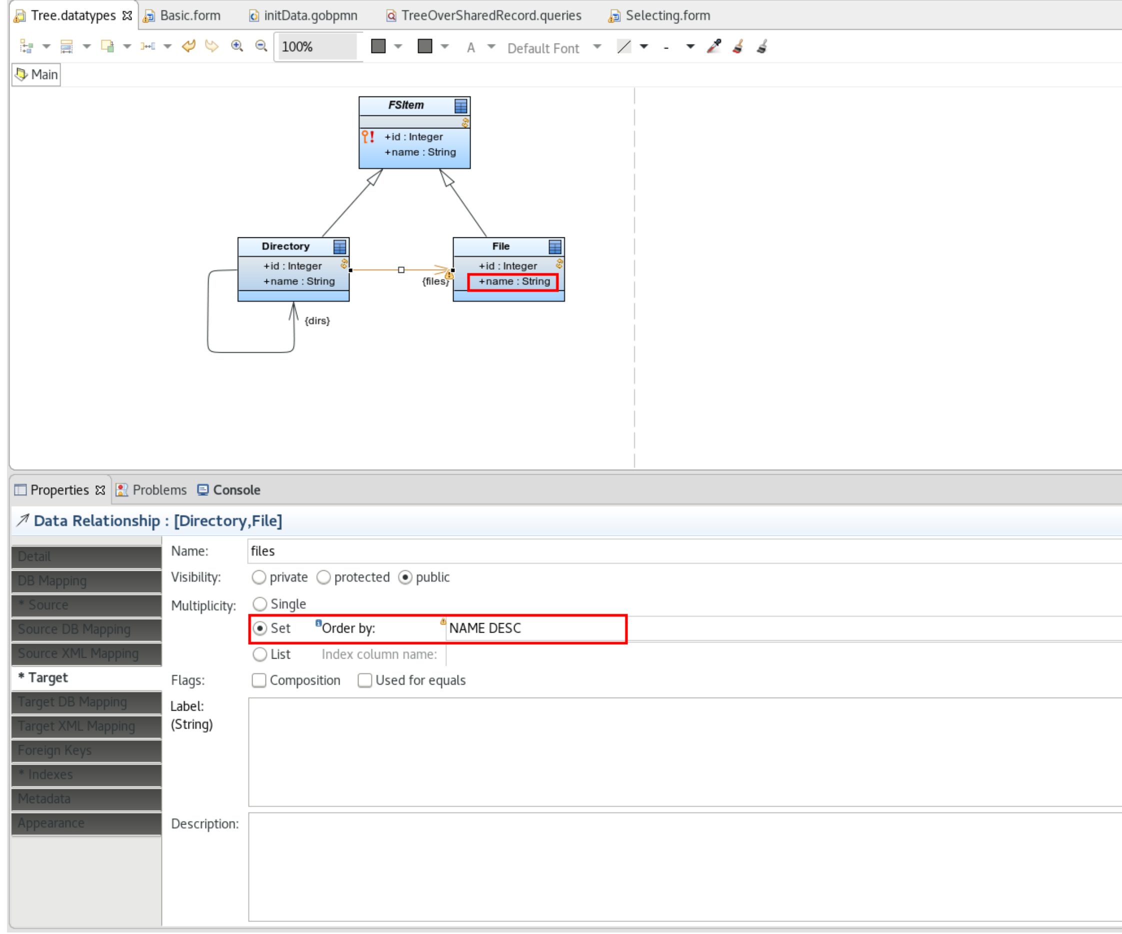
   Important: For record relationships with optimistic locking, make sure to **name both ends** of the relationship so that any of the ends can be used by Hibernate as the owner of the relationships.

2.12.7 Setting up Relationships Between Shared Records

Relationships between shared 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.

To define a relationship between one or two shared records, do the following:

1. Create and define the generic relationship as for non-shared records.
2. On relationship end pointing to a shared record, define the properties related to the database mapping:
   - on the **Source**/**Target** tab of a end:
     - for Set multiplicity define the database column that should be used to ordered the records



     - for List multiplicity define the index column
   - on the **Source**/**Target DB Mapping** tab:
     - **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
- **BLOB length** on ends pointing to a non-shared record from a shared record: the space reserved for the non-shared record value since the entire tree of the non-shared record is serialized and stored.
  BLOB is defined as an expression block that returns an Integer, for example `1024*1024`. Note that you cannot access constructs and variables from parent contexts from the expression block.

3. On relationships between two shared records:

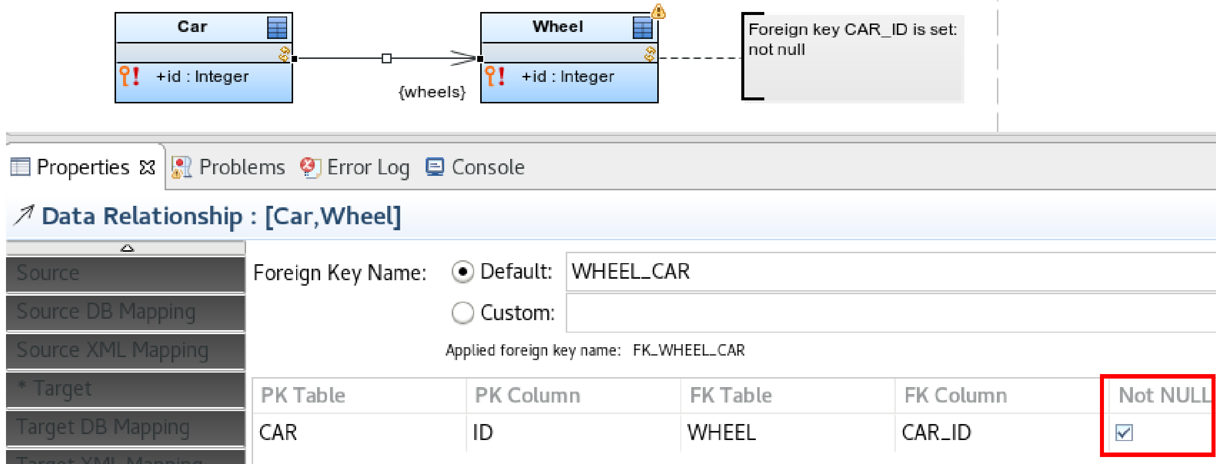
   - On the *DB Mapping* tab, define whether the underlying database structure should be updated, when the relationship structure is modified on the *DB Mapping* tab.
   - On the *Foreign keys* tab: adapt the foreign mapping if required. Make sure the types of the primary key and of the foreign key match.

2.12.7.1 Enforcing Not Null Foreign Key to Related Records

To enforce a non-null value of a foreign key on a related record, do the following:

1. Select the relationship.
2. In the *Properties* view on the *Foreign Keys* tab, select the *Not NULL* flag for the given private-key table.
3. Consider naming the end pointing from the end with the foreign key to the end without it. This will make sure that you can actually create the records.

Important: Consider the following: you create the record Car pointing to a record Wheel connected with a relationship. The end pointing to Wheel is a Set and is named `wheels`. The foreign key WH↩ EEL_CAR contains has the ID of the car, which must not be null (you can check the foreign key on the *Foreign Keys* tab in the *Properties*view).



If you now execute `new Car(wheels -> {new Wheel()})`, the execution fails: The server first attempts to create the Wheel, but the Wheel has no foreign key to a car. If you create the car first (`new Car()`), you will not be able to assign it to a wheel when creating it.

2.12.7.2 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.

> **Important:** The absence of indexes on your shared records can cause performance issues. It is recommended to define indexes to prevent slow search on your database data.

To define indexes for a table of a shared record, 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.

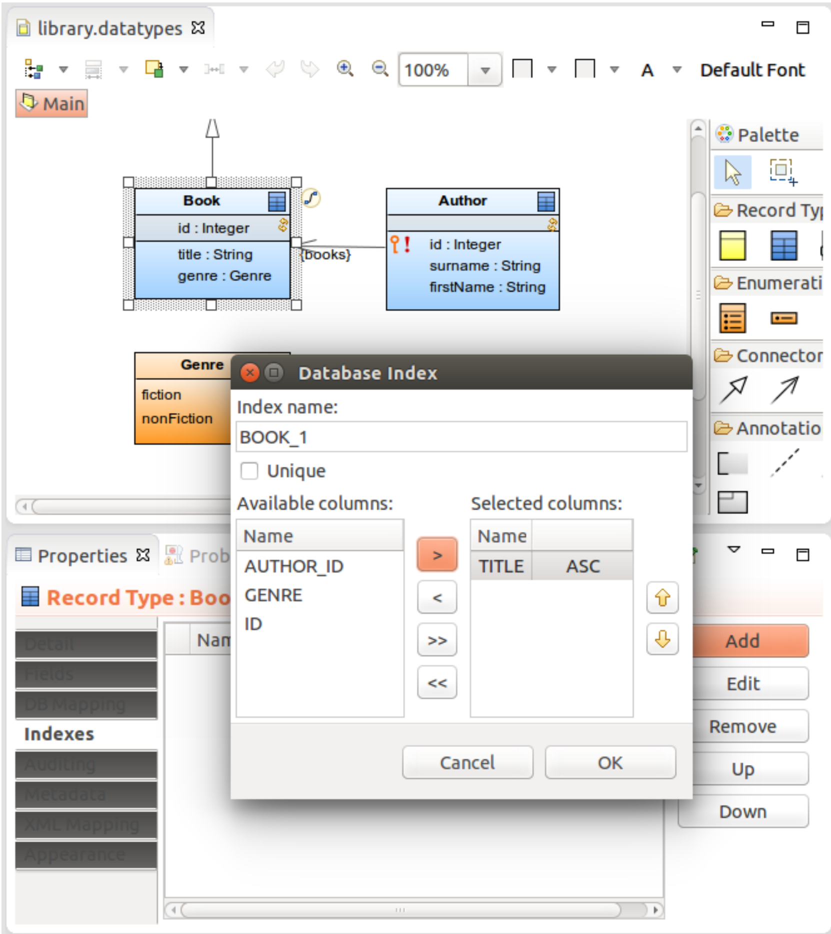


Figure 2.81 Defining index

5. Upload the Module.

2.12.7.2.1 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**.

2.12.7.3 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.

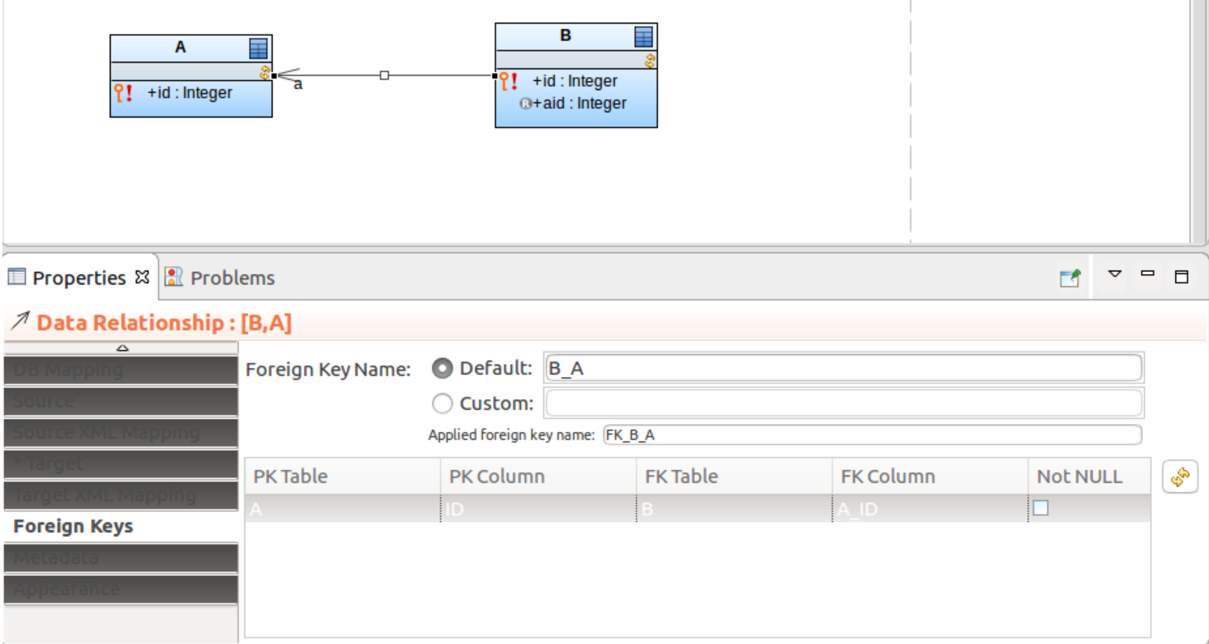


Figure 2.82 Foreign key on the relationship end

4. On the source record, create a read-only field that will be mapped to the foreign key.

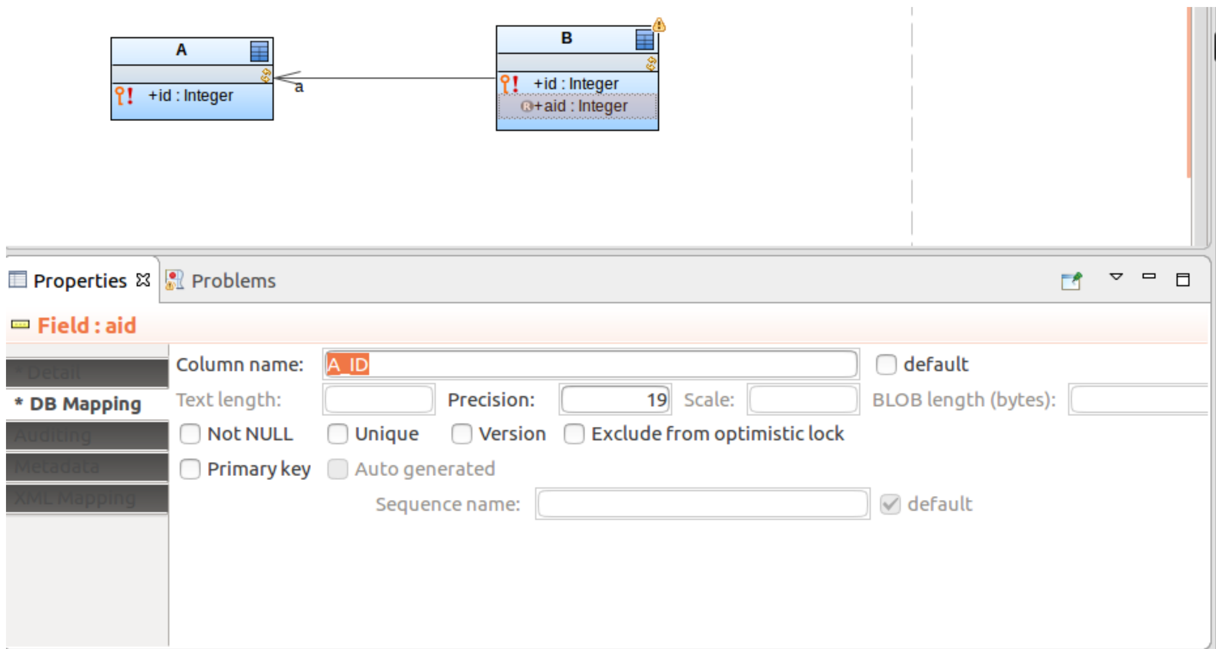


Figure 2.83 DB mapping of the field

5. On the DB Mapping tab of the field Properties view, 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);`

2.12.8 Auditing: Shared Record Versioning

The auditing of shared records refers to storing of all versions as the record instances as they change, thus providing a version-control mechanism.

When you change instances of an audited Record in a transaction, the auditing mechanism creates a revision entity with a "snapshots" of the changed records in the auditing table of the record. Optionally, it enters the Record name and revision ID for each changed record into the entity name table so you can look up any other audited records that were changed in that transactions as well.

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 entity of the Revision Entity type, for example, with the ID `1`.
2. Record the changes on the Book instances: auditing adds 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 the record type: Hence two entries with record BOOK and the revision ID `1` are created.

2.12.8.1 Setting up Auditing

To set up auditing, do the following:

1. Create the Revision Entity record that reflects the table with revisions:
   (a) Create a *RevisionEntity* shared record.
   (b) On its *Auditing* tab in the Properties view, select the *Revision entity* option.
   (c) Typically you will add also the *timestamp* field to the record to be able to request the timestamps of revisions: add a field of type Integer or Date and set it as *Revision timestamp* on the Auditing tab in its Properties view.

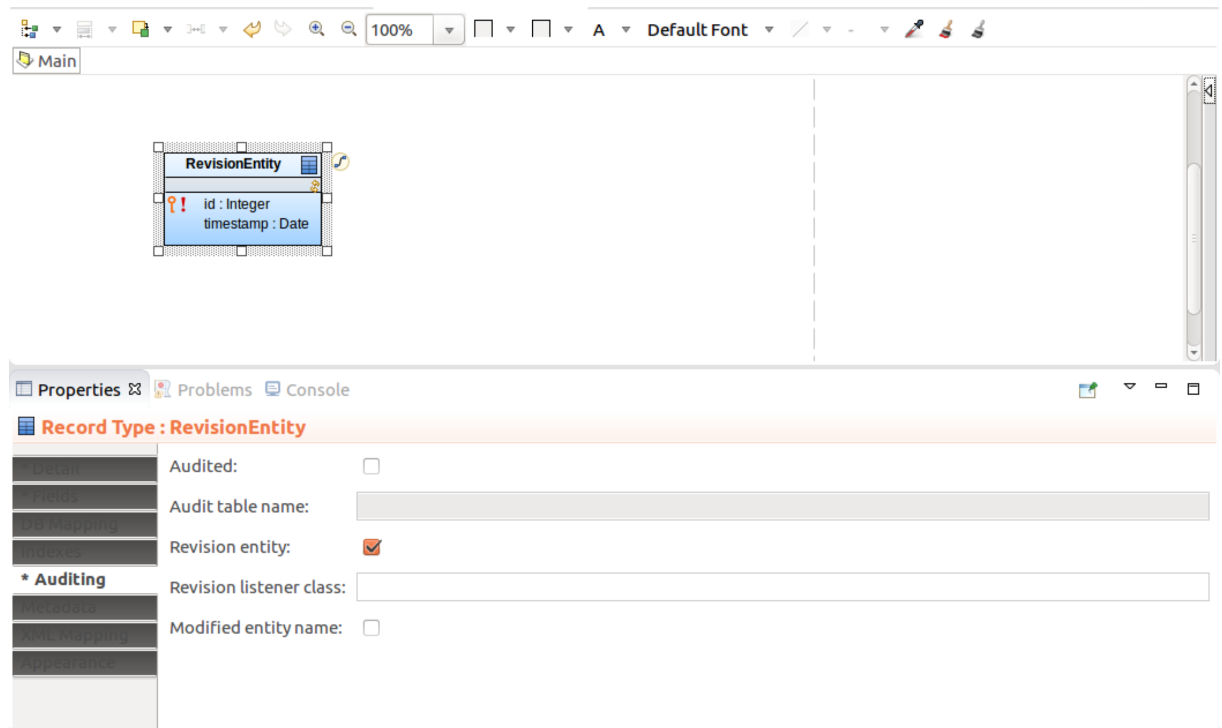


Figure 2.84 Revision entity shared record

> **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 Customizing Entity Auditing).

2. Optionally, you can store the list of revision IDs with the entities that were changed so you can easily look up revisions based on the entities they changed efficiently:

   (a) Create the *EntityName* shared record related to your *RevisionEntity* record.

   (b) In its Properties view on the Auditing tab, select the *Modified entity name* property.

   (c) On the relationship end pointing to the *EntityName* record, set the multiplicity to *Set*.

   (d) On the *EntityName* record, add a String field that will hold the name of the changed entity: On the *Auditing* tab of its Properties view, select its *Entity name* property.

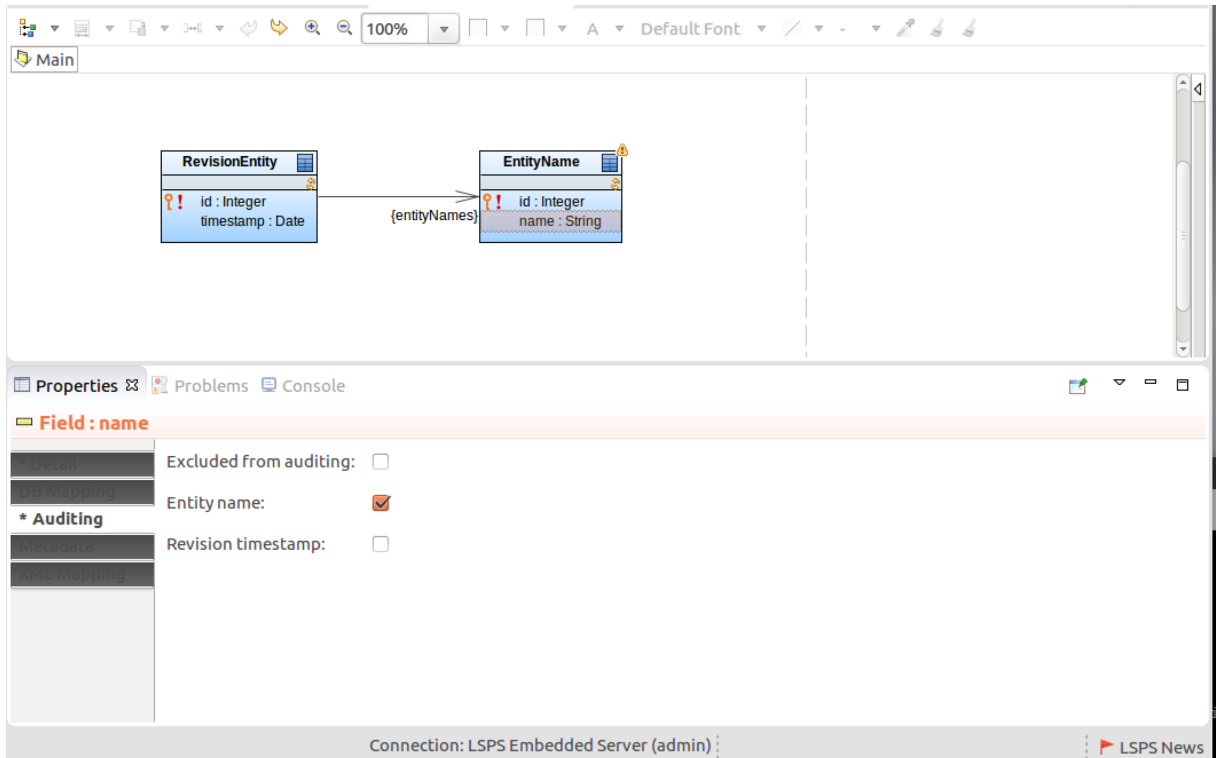


Figure 2.85 Shared record of the modified entities of a revision

3. Upload the module.

   Important: Only *one* revision entity record and modified entity name record can exist on an LSPS Server.

4. Now you can set your shared records as audited: open the record's properties and on the *Auditing* tab select the *Audited* option and upload the Module.

2.12.8.2 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.

3. If the record has a relationship to another audited record with one-to-many multiplicity, name both ends of the relationship unless you excluded the relationship from auditing.

4. 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.

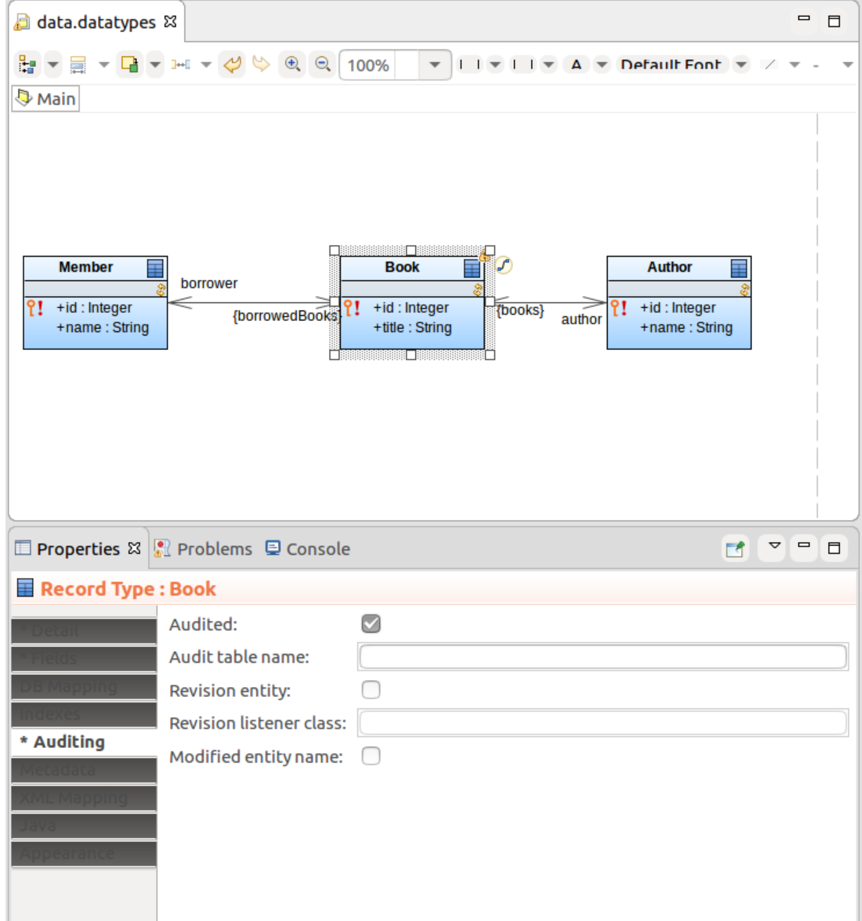


Figure 2.86 Audited shared Record

2.12.8.3 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.

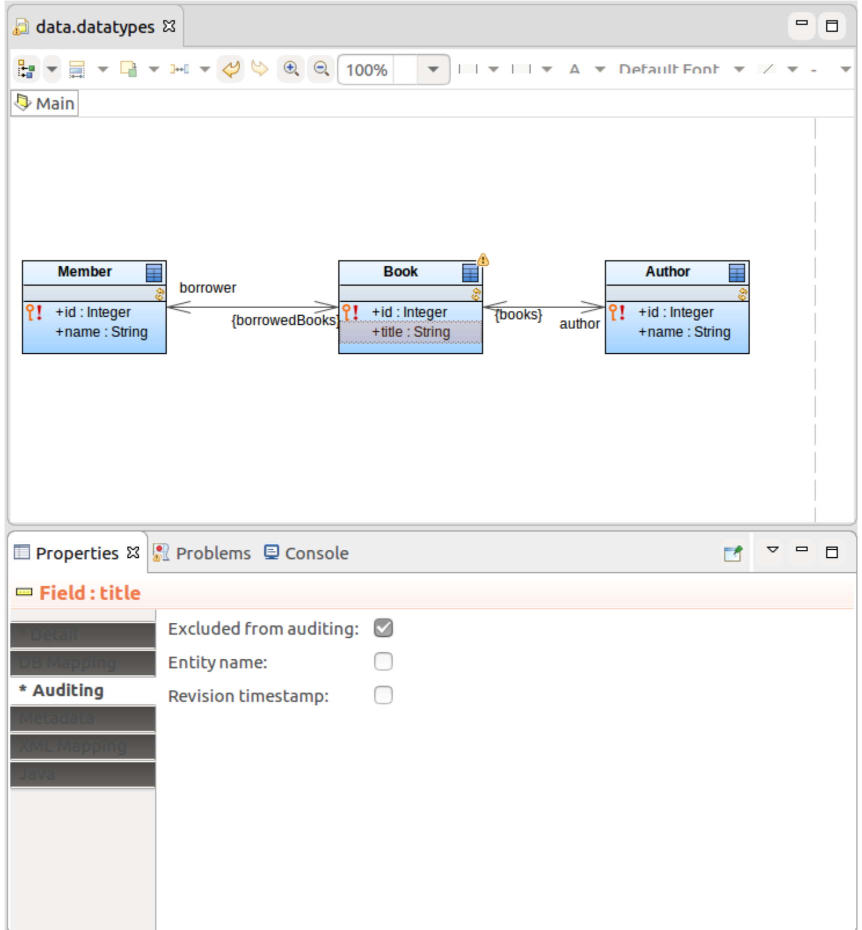


Figure 2.87 Field excluded from auditing

2.12.8.4 Including and Excluding a Relationship End from Auditing

By default relationships between audited records are audited as well.

To exclude a 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.

> **Important:** When auditing a record relationship with one-to-many multiplicity, both ends of the relationship must be named.

2.12.9 Caching a Shared Record

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 transaction or model instance life.

> **Note:** 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.

The caching is defined by cache regions, which are added to the LSPS Server cache on module upload.

2.12.9.1 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.

2.12.9.2 Disabling Cache Regions

To disable the LSPS system cache regions, define the disabled cache regions in the `<YOUR_CUSTOM_APP>-ejb/src/main/resources/cache-regions.properties` file of your custom LSPS application.

2.12.10 Change Proxy: Transient Data of Shared Records

To work with preliminary business data in one or multiple shared record instances, create a change proxy set with proxies of your shared records and introduce your changes to the proxies.

A proxy exists on a proxy level greater than 0: the "real" records exist on proxy level 0 and are not considered proxy objects. When you create a proxy from a record, the proxy exists on level 1; when you create a proxy from this proxy, it exists on proxy level 2, etc.

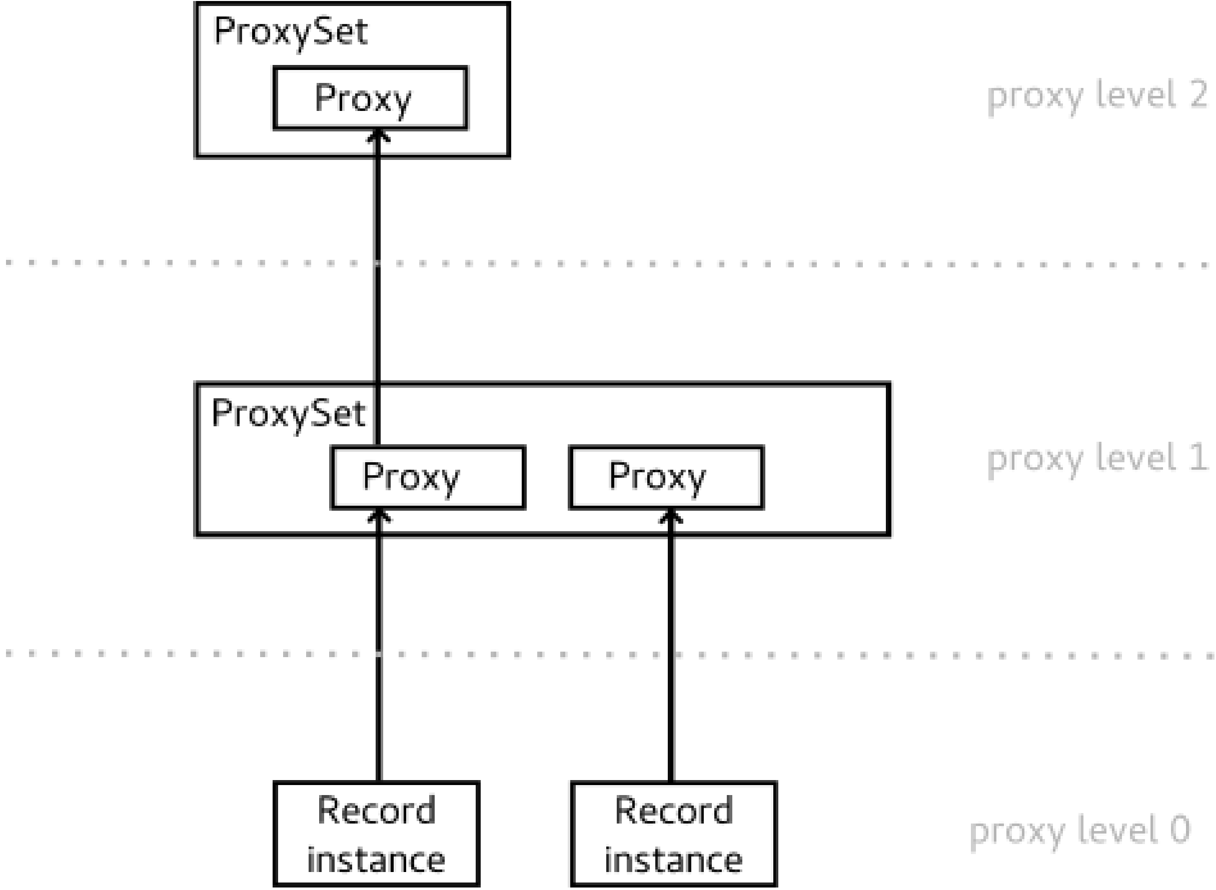


Figure 2.88 Record instances with their proxy trees

Proxies of a record have to constitute a continuous chain of proxies across all their proxy levels: when you create a proxy of a record on proxy level 2, its proxy is automatically created also on level 1.

A proxy is created by a proxy set: a proxy set holds its proxies with their changes and holds the information on its proxy level. To apply the changes on the proxies held in a proxy set, merge the proxy set.

Proxies are useful especially in forms to allow to collect and validate all data before persisting it; refer to `Using Proxies in Forms`, and for usage examples to `Creating a Model Popup` and `Creating a Public Popup`.

2.12.10.1 Creating a Proxy

A proxy is created by a proxy set and it is the proxy set that defines the proxy level of its proxies. Hence to create a proxy, you need to first create a proxy set:

1. To create a proxy set, call the `createProxySet()` function:
    - To create a proxy set on proxy level 1, call the function with the `null` parameter.
    - To create a proxy set on another proxy level, call with the parameter defining the underlying proxy set.
2. To create proxies in a RecordProxySet, use its `proxy()` method:
    - To create a proxy of existing record instances:

  - `proxy(T record*, Property... property)` or `proxy(T record*, List<Property> properties)`
  - `proxies(Collection<T> records*, Property... properties)` or `proxies(Collection<` `records*, List<Property> properties)`

  If the proxy of the record already exists in the proxy set, the method returns the proxy.

- To create proxies of record types, which is useful for cases when the record instance is to be created only once all the record data has been collected:
  - `proxy(Type<T> recordType*)`
  - `proxyOfProperties(Record record*, Property... properties)`

Example: Creating a proxy set on level 1 and creating a proxy:

```
//creating a proxy set on proxy level 1:
def RecordProxySet rps := createProxySet(null);
//creating a proxy of a record type in the proxy set:
rps.proxy(Persona);
//creating a proxy of a record in the proxy set:
rps.proxy(john);
```

Each proxy exists in a continuous chain of proxies across all previous proxy levels. If the proxy on a previous level does not exist, it is automatically added to the underlying proxy set on the lower level.

Consider the following case:

```
def SimpleRecord rec := new SimpleRecord(number -> 1);
//empty proxy set on level 1:
def RecordProxySet set1 := createProxySet(null);
//proxy set on level 2:
def RecordProxySet set2 := createProxySet(set1);
//proxy of rec added to proxy set on level 2:
def SimpleRecord proxy2 := set2.proxy(rec);
```

On runtime proxy of *rec* is created in set1 when it is created in set2. This allow successful merge of the proxy from set1 to set2 and to its rec when requested.

2.12.10.2 Creating a Proxy of a Related Record

- To create a proxy of a related record instance, use the `proxy(<RecordInstance>, <Properties↩ OfRelatedRecords>, ..)` call when creating the proxy.

```
def Persona john := new Persona(name -> "John", address -> new Address());
def RecordProxySet rps := createProxySet(null);
//creates a proxy of john and its related address in rps:
def Persona johnProxy := rps.proxy(john, Persona.address);
```

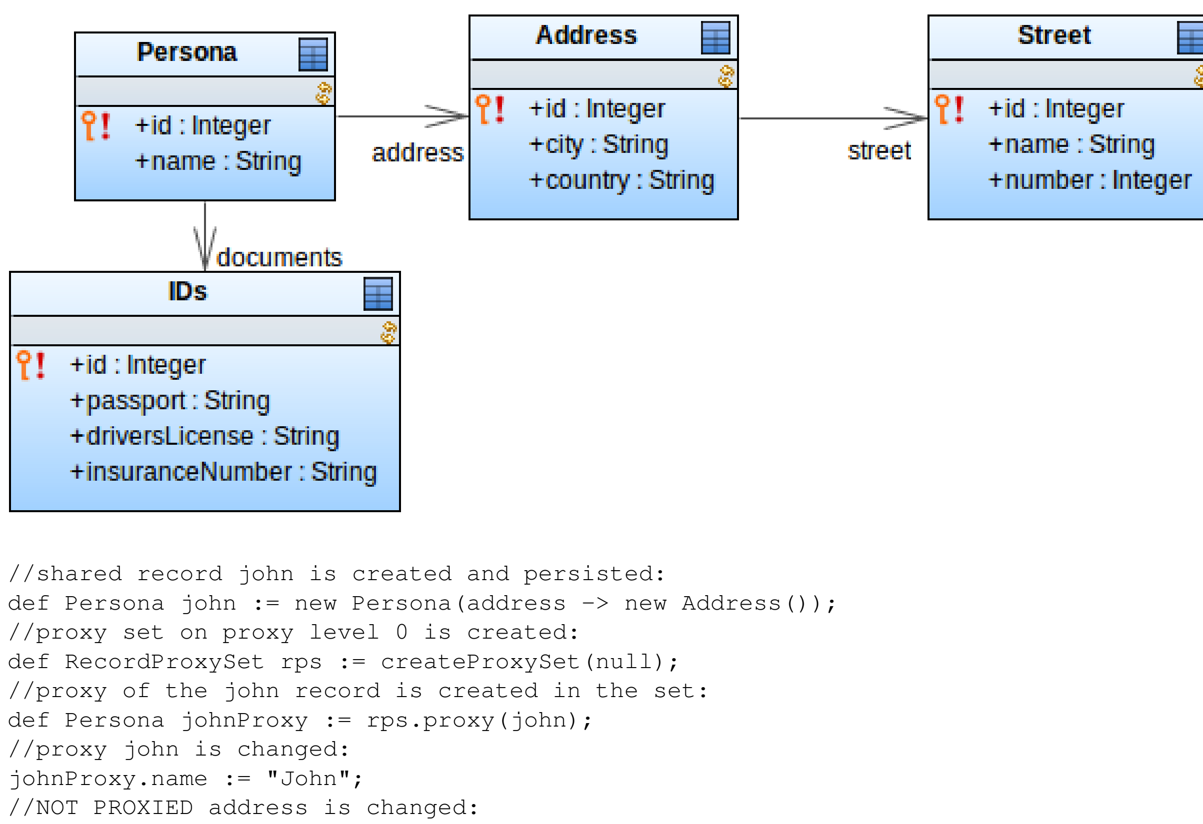
- To create a proxy of a record type and its related record type, create the proxies and assign the related proxy to the proxy:

```
def Persona personaProxy := rps.proxy(Persona);
personaProxy.address := rps.proxy(Address);
```

- To create a proxy of a related record type at the end of a set or list relationship and merge it, create the proxies of record on both end and use add to establish the relationship on the proxy level:

```
def Persona personaProxy := rps.proxy(Persona);
def Address a1 := rps.proxy(Address);
personaProxy.addresses.add(a1);
```

Important: If you don't create a change proxy of a related record, the system will access the record instance on the underlying level when you navigate from the proxy to the related record. Let's take the following data model and code:




```
//shared record john is created and persisted:
def Persona john := new Persona(address -> new Address());
//proxy set on proxy level 0 is created:
def RecordProxySet rps := createProxySet(null);
//proxy of the john record is created in the set:
def Persona johnProxy := rps.proxy(john);
//proxy john is changed:
johnProxy.name := "John";
//NOT PROXIED address is changed:
johnProxy.address.country := "USA";
```

In this case, a record with an empty person and the address country "USA" is persisted.

Note that navigation across multiple relationships is not supported in the `proxy()` call; hence for the example data model, this call is not valid:

```
def Persona johnProxy := rps.proxy(john, Persona.address.street);
```

2.12.10.3 Merging a Proxy Set

To merge a proxy set into the underlying records or proxies, call `merge(Boolean checkConflicts*)`, which returns the list of merged records.

Note: It is possible to merge a proxy directly to the underlying context without a RecordProxySets object; however, when you use a RecordProxySet and attempt to add a proxy object, such as, a proxy of a related record, the RecordProxySet checks if such a proxy already exists among its proxies thus keeping a clean tree of its proxied objects. It is strongly recommended to prefer RecordProxySets over direct merge.

If using optimistic locking on your records, consider using the *merge* call of with the Boolean parameter *check*↩ *Conflicts* set to true so the merge checks for conflicts on the underlying records or proxies and handle the possible conflicts.

2.12.10.3.1 Handling Optimistic Lock on Proxy Set Merge

The `merge(true)` call on a RecordProxySet fails if one of its proxies tries to merge its changes into a record with optimistic locking that has been changed by another `transaction` since the proxy was created.

For proxies of records with optimistic locking, consider either handling the possible optimistic-lock exception or adjust your data model so that the collisions cannot occur. For example, aggregate the data that might collide in a collection and create the resulting entity later.

Note that the `merge(true)` call might succeed in this scenario

- when the underlying record was changed in the same transaction;
- if the underlying record as well as its proxies are created in the same transaction.

Catching OptimisticLockException

```
def RecordProxySet rps := createProxySet(null);
rps.proxy(myRec);
try rps.merge(true)
    catch "com.whitestein.lsps.common.OptimisticLockException" ->
    //log a message if the record has been changed:
    log("failed to merge changes", 100)
end
```

> **Important:** If you assign to a proxied property an object from a higher proxy level, on merge, the association is applied on the target object: the original relationship is removed.

2.12.10.4 Deleting a Record Using a Proxy

To delete a proxy, call the *deleteRecord()* function on the proxy: this will delete the underlying record or proxy on merge.

```
//persona john is persisted:
def Persona john := new Persona(name -> "John");
def RecordProxySet rps := createProxySet(null);
//proxy of john created on proxy level 1:
def Persona johnProxy := rps.proxy(john);
//proxy object is deleted (persisted shared record still exists):
deleteRecords(johnProxy);
//on proxyset merge, the persisted john record is deleted:
rps.merge(false)
```

2.12.10.5 Checking if an Object is a Change Proxy

You can check if a record is a proxy with the `isProxy()` call.

2.12.10.6 Checking the Proxy Level of a Change Proxy

You can check the proxy level of a proxy with a `getProxyLevel(<Proxy>)` call, which returns an Integer. 0 designates the execution level with "real" context data.

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 Decision Tables

Decision tables serve to define business rules, which are then used to adapt the behavior of models. The tables can be modified on runtime.

> **Note:** The LSPS implementation of Decision Model and Notation (DMN) is based on the industry standard for decisions established by OMG, DMN version 1.1.

Decision tables are defined in a dedicated editor in Designer as part of a module. The module can then evaluate a rule on runtime and be designed to behave differently depending on the rule outcome.

A decision table is considered a shared record. To evaluate a rule use the `evaluation()` call with the input expressions on the record: The call checks which input entries fit the table input values and returns their output values.

A decision table can be modified by users from their process application in their browser: They are made available by the Decision Table components in form definitions.

ApplicationTrainingDays			
F	trainer	level	trainingDays
LSPS	Trainer	Level	Set<Day>
	== Trainer.JANE, == Trainer.JOHN	== Level.BEGINNER, == Level.INTERMEDIATE, == Level.ADVANCED	Day.MONDAY, Day.TUESDAY, Day.WEDNESDAY, Day.THURSDAY, Day.FRIDAY
	This is the name of the trainer.	*This is the level of the trainee.*	*Resulting training days.*
1	== Trainer.JANE	== Level.BEGINNER	{Day.MONDAY, Day.TUESDAY}
2		== Level.INTERMEDIATE	{Day.TUESDAY, Day.WEDNESDAY}
3		== Level.ADVANCED	{Day.WEDNESDAY, Day.THURSDAY}
4	== Trainer.JOHN	== Level.BEGINNER	{Day.MONDAY}
5		== Level.INTERMEDIATE	{Day.TUESDAY}
6	Input	== Level.ADVANCED	{Day.WEDNESDAY} Output

Figure 2.89 The evaluation call to the decision table takes the trainer and level arguments and returns the trainingDays value.

2.14.1 Designing a Decision Table

To design a decision table, do the following:

1. Import the `dmn` module with the decision-tables-related resources.
2. Create a decision-table definition file: right-click the module and go to **New** -> **Decision Table Definition**, enter the decision table name, for example, `TrainingDays`.
3. Select the language of the rules:
   (a) Click the cell with the value `LSPS`.
   (b) In the context menu, select the language:
      - LSPS: you will specify the input values as full Boolean expressions or as the right side of Boolean expressions with an operator, for example, `< 1000`, `in 1..1000`
      - SFEEL: though simple, the language is very restrictive; refer to the SFEEL section for further information.
4. Define the input parameters as blue columns:
   (a) Add or remove the columns to match the number of input parameters: click the `:::` icon in the title of a blue column and in the context menu, select the required action.

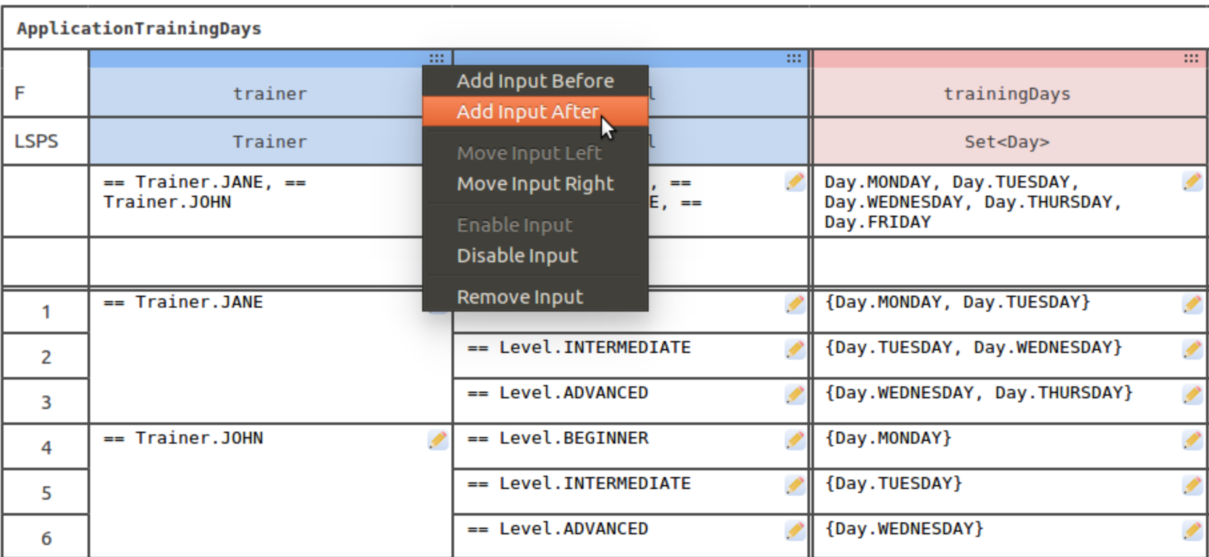
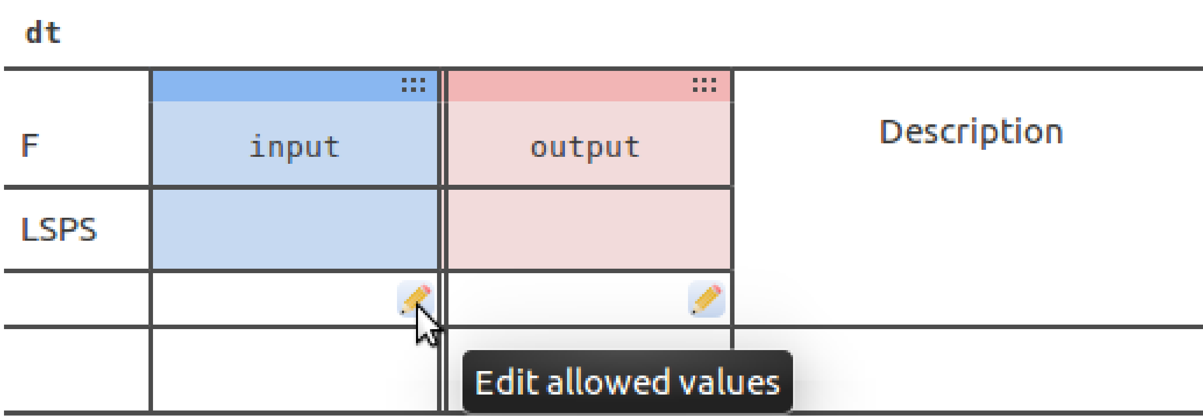


Figure 2.90 Adding input column

(b) In each column header, define the properties of the input parameter:

- Define the **label of the input in the top cell** of the column header
- Define **the data type of the input in the cell below** (If undefined, it is considered an *Object*).
- Optionally in the cell bellow the data type, define the allowed input values: click the **Edit Allowed Values** button and use the displayed dialog.



You can use the input expressions, that is, the *column captions*, to reference the values passed to the decision table, for example, if an input column has the name `income` you can use it in an expression, such as, `getIncomeInEur(income) > 1000`.

Note that while output columns can make use of input columns, input columns cannot see the output columns.

5. If you require special handling on a rule, or you need to prepare additional data for the output, define an input with no name. Note that such inputs are not considered inputs and are not set by an input argument.

ApplicationTrainingDays				
F	trainer	level		trainingDays
LSPS	Trainer	Level		Set<Day>
	== Trainer.JANE, == Trainer.JOHN	== Level.BEGINNER, == Level.INTERMEDIATE, == Level.ADVANCED		Day.MONDAY, Day.TUESDAY, Day.WEDNESDAY, Day.THURSDAY, Day.FRIDAY
	This is the name of the trainer.	*This is the level of the trainee.*		*Resulting training days.*
1	== Trainer.JANE	== Level.ADVANCED	application.applicant.name == "Special Trainee"	{Day.MONDAY, Day.TUESDAY, Day.WEDNESDAY, Day.THURSDAY, Day.FRIDAY}
2	== Trainer.JANE	== Level.BEGINNER		{Day.MONDAY, Day.TUESDAY}
3		== Level.INTERMEDIATE		{Day.TUESDAY, Day.WEDNESDAY}
4		== Level.ADVANCED		{Day.WEDNESDAY, Day.THURSDAY}
5	== Trainer.JOHN	== Level.BEGINNER		{Day.MONDAY}
6		== Level.INTERMEDIATE		{Day.TUESDAY}
7	== Trainer.JOHN	== Level.ADVANCED		{Day.WEDNESDAY}

Figure 2.91 Handling a special case in first rule using anonymous input

6. Define the result outputs as red columns:

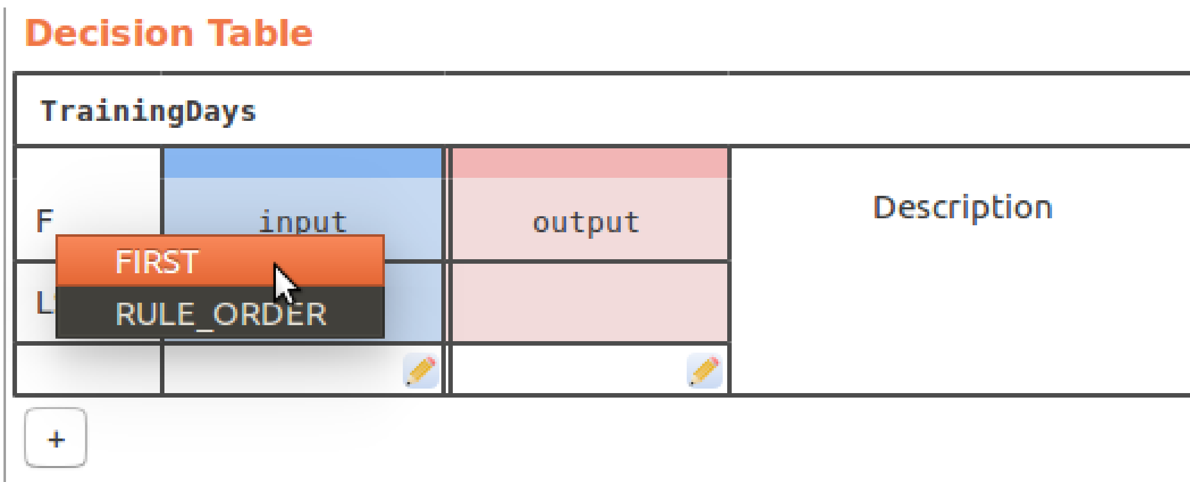
   (a) Add or remove the columns to match the number of output values: click the **: : :** icon in the label cell of a red column and in the context menu, select the required action.

   (b) In each column header, define the properties of the output parameter:

   - Define the **label of the output in the top cell** of the column header
   - Define **the data type of the output in the cell below** (If undefined, it is considered *Object*).
   - Optionally in the cell bellow the data type, define the allowed output values: click the **Edit Allowed Values** button and use the displayed dialog.

7. Click the policy cell to set the hit policy:

   - **FIRST**: the output of the first satisfied condition set is returned as the result.
     The return value is one instance of the outputs.
   - **RULE_ORDER**: the outputs of all satisfied conditions are returned as a list; the order of the outputs follows the order, in which the rules are defined.



8. Define the rules: click the + sign to add a row for a new rule to the table and define the input conditions that must be true for the rule to apply and their output values.

9. To manage the rules, select and right-click the number in the rule row and select the required action; you can copy and paste a rule, remove a rule, change the order of rules, etc.

ApplicationTrainingDays

F	trainer	level
LSPS	Trainer	Level
	== Trainer.JANE, == Trainer.JOHN	== Level.BEGINNER, == Level.INTERMEDIATE, == Level.ADVANCED
	This is the name of the trainer.	*This is the level of the trainee.*
1	== Trainer.JANE	== Level.ADVANCED
		== Level.BEGINNER
		== Level.INTERMEDIATE
		== Level.ADVANCED
		== Level.BEGINNER
		== Level.INTERMEDIATE
		== Level.ADVANCED

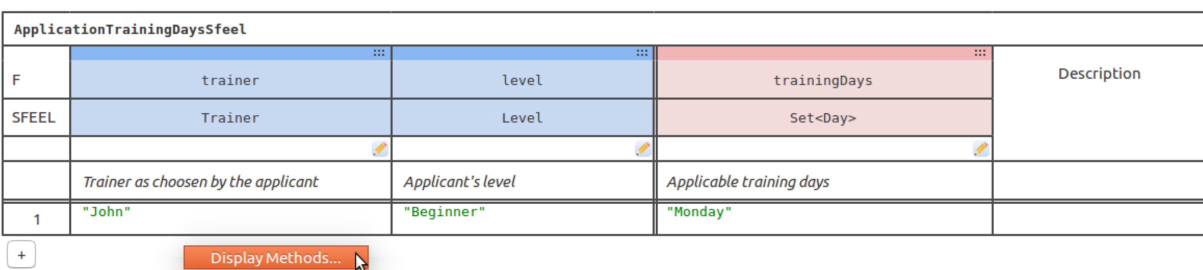
Add Rule Above
Add Rule Below
Move Rule Up
Move Rule Down
Copy Rule
Paste Rule
Enable Rule
Disable Rule
Remove Rule

+

You can design the decision tables also programmatically with the API methods, such as, `setHitPolicy()`, `addInput()`, `addOutput()`, `addRules()`, and others. Refer to the `dmn documentation`.

Now you can use the decision table in your model to evaluate a rule and `add it to a form` so it can be edited from the Process Application.

To check the constructor and methods of the decision table, including the `evaluate()` method which you call on a decision table with input arguments to get the output value, right-click below the table and select **Display Methods**.



ApplicationTrainingDaysSfeel

F	trainer	level	trainingDays	Description
SFEEL	Trainer	Level	Set<Day>	
	Trainer as choosen by the applicant	*Applicant's level*	*Applicable training days*	
1	"John"	"Beginner"	"Monday"	

2.14.1.1 SFEEL

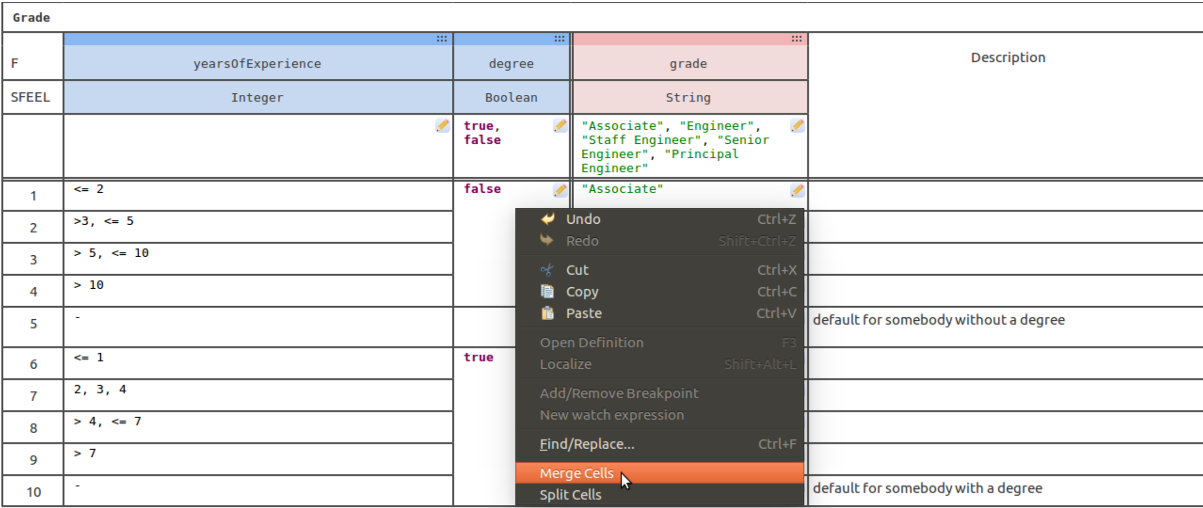
SFEEL is a simple expression language in which you can define the business rules in a decision table. However, mind that the rules must not have any side effects; they can use only basic data types and simple unary conditions with no external resources. If you require more complex expressions in your decision table, use the LSPS Expression Language.

Even if you set your decision table to use SFEEL, the input and output data types on decision tables are defined as their LSPS equivalents:

Original SFEEL	LSPS SFEEL	Example
boolean	Boolean	true, false
number	Decimal or Integer	11.0001, 12
string	String	"hello"
days and time duration	duration(string_literal)	duration("P2DT3H4M30.1S")
year and months duration	duration(string_literal)	duration("P1Y10M")
time	time(string_literal)	time("10:00")
date	date(string_literal)	date("2012-12-24"), date("2012-12-24+09:00")

2.14.1.2 Merging and Splitting Cells

To merge a cell with the cell below in your decision table, right-click the cell and select **Merge Cells**. You can split a merged cell into the individual cell analogously: select **Split Cells** from the context menu.



2.14.1.3 Copying Rules

To copy a rule in your decision table, click its row number and select **Copy Rule**. To paste it, click the row number of the row under which you want to insert the rule and select **Paste Rule**.

ApplicationTrainingDays

F	trainer	level	trainingDays
LSPS	Trainer	Level	Set<Day>
	== Trainer.JANE, == Trainer.JOHN	== Level.BEGINNER, == Level.INTERMEDIATE, == Level.ADVANCED	Day.MONDAY, Day.TUESDAY, Day.WEDNESDAY, Day.THURSDAY, Day.FRIDAY
	This is the name of the trainer.	*This is the level of the trainee.*	*Resulting training days.*
1	== Trainer.JANE	== Level.BEGINNER	{Day.MONDAY, Day.TUESDAY}
2		== Level.INTERMEDIATE	{Day.TUESDAY, Day.WEDNESDAY}
3		== Level.ADVANCED	{Day.WEDNESDAY, Day.THURSDAY}
4	[illegible]	== Level.BEGINNER	{Day.MONDAY}
5		== Level.INTERMEDIATE	{Day.TUESDAY}
6	[illegible]	== Level.ADVANCED	{Day.WEDNESDAY}

Add Rule Above
Add Rule Below
Move Rule Up
Move Rule Down
Copy Rule
Paste Rule
Enable Rule
Disable Rule
Remove Rule

2.14.2 Creating, Saving and Loading a Decision Table

You can **create a new decision table instance** with the constructor with the argument `true`: `<Constructor>(true)` will construct the table based on the definition.

If you want to use the same decision table on multiple occasions, for example, from different process instances, you can persist it and load it later.

You can **save a decision table** with the `save(<StringID>)` method of the decision-table instance. To **load a decision table**, create its uninitialized instance and then call the `load(<StringID>)` method on it.

- `<Constructor>(false)` will only create the decision table without values so you can load a saved table into it using the `load()` call. **Example loading of a decision table if it exists**

```
def ApplicationTrainingDays atd := new ApplicationTrainingDays(false);
//create a table with the TrainingDaysTable id if it does not exist in the db:
if (atd.load("TrainingDaysTable") == false) then
   atd := new ApplicationTrainingDays(true);
   atd.save("TrainingDaysTable")
end;
```

These calls enable you to save and load a decision table from the Process Application via the `Decision Table` component. Note that you can change the decision table definition in the component as well.

To **find a saved table,** call `findDecisionTableIds(<IdPattern>)` or `findDecisionTables(<IdPattern>)`.

> **Note:** You will not be able to check the return type of an evaluate() call on design time for tables acquired with the `load()` method call since the tables could be edited on runtime and their input and output parameters might change.

You can check the logic in the methods of the decision table (right-click below the table and select **Display Methods**).

2.14.3 Evaluating a Rule with a Decision Table

When you want to make a decision based on the rules of a decision table, perform an `evaluate()` call on an instance of the table; to obtain a decision table, you can either create a new instance of the table or load a previously persisted table with the save() call. The `evaluate()` call evaluates the input conditions of the rules as defined in the table and returns one or multiple results in which all conditions are `true` or `null`; Mind that the number of results, that is, whether results only from one row or all matching rows is returned, depends on the hit policy setting.

To evaluate a rule in a new instance of a decision table, create the instance and call evaluate() on it. Note that in this scenario, the data types returned by the table are explicitly stated:

- Table with the FIRST hit policy returns an instance of the output(s) or null if there are no matches.

```
def TrainingDaysDT trainingDay := new TrainingDaysDT(true);
//the type of return value depends on how you define the input parameters;
//input parameters are defined as position parameters, return value is a list of the given type:
def List<TrainingDays> ltd;
ltd := trainingDay.evaluate(Trainer.MIKE, Level.INTERMEDIATE);
//input parameters are defined as a map, return value is a list of map of objects of the given type:
def List<Map<String,Object>> mtd;
mtd := trainingDay.evaluate(["trainer" -> Trainer.MIKE, "level" -> Level.INTERMEDIATE]);
```

- Table with the RULE_ORDER hit policy returns a List or Map of outputs (empty map if there are no matches):

```
def TrainingDaysDT trainingDay := new TrainingDaysDT(true);
def List<List<TrainingDays>> ltd;
ltd := trainingDay.evaluate(Trainer.MIKE, Level.INTERMEDIATE);
//or:
def List<Map<String,Object>> mtd;
mtd := trainingDay.evaluate(["trainer" -> Trainer.MIKE, "level" -> Level.INTERMEDIATE]);
```

Important: Designer and the LSPS Server do not check the type of the decision table: hence, if a variable is of a type of a particular DecisionTable and you assign it another type, it does not cause any exception. This allows editing of the DecisionTable structure on runtime from the Process Application.

2.15 Webservice Processes

As part of your models, you can design processes that act as web-service servers or clients: The procedures differ slightly depending on whether you are creating a SOAP client or server,or a REST client or server.

2.15.1 REST Webservice Processes

You can create a process that will act as a RESTful Web Service server or client using dedicated task types:

- server process with tasks generated based on the RESTful definition
- client process with the HttpCall task type

The RESTful Web Services are implemented as required by *JAX-RS: Java TM API for RESTful Web Services version 2.0 Final Release*.

2.15.1.1 Creating a REST Server Process

To create a Process that will handle REST requests, you need to create a definition and generate task types based on the definition. The task types will then serve to design the Process.

To create a REST web service server process, do the following:

1. Create the records for the server:
    - **input** record for data received from the web service client
    - **output** record for data sent as a response to the web service client
    - **error** record for data sent to the client as an error
      If you plan to use JSON as content type, make sure the properties of the records are only of the following types:
        - String, Integer, Decimal, Boolean, Date, or Enumeration.
        - Set, List, Map with String, Integer, Decimal, Boolean, Date, Enumeration members, or Set, List, Map members with members of these types.
        - References to records with properties of the types as defined above.
2. Create a web service definition file:
    (a) Right-click the Module, go to **New** > **RESTful Webservice Definition**
    (b) Open the file in the web service editor and define the web service properties:
    - *Path Template*: template of the URL path of the service
      Define the Path Template with any path variables you want to use: the variables will be added to the Path Variables table below automatically. The template supports path variables, such as, `{itemID}`, and regular expressions. The REST server URL is then resolved as `<LSPS_Server_URL>/lsps-ws/rest/<Path_Template>` and path variables are accessible as parameters of the waitRestRequest task.
    - *Method*: method type
    - *Input type*: input record
    - *Output type*: output record
    - *Error type*: error record
    - In the Path Variables table, adjust the data type of individual variables: the generated task types will have the variables as parameters.

    (c) If applicable, select *Generate optional parameters for access to HTTP headers*: if selected, the generated task type will contain the responseHeaders parameter so you can modify the response headers; otherwise, you will be able to use only the default headers.
    (d) If applicable, select *Generate optional parameters for access to HTTP request parameters*: if selected, the generated task type will contain the requestHeaders parameter so you can modify the request headers; otherwise, you will be able to use only the default headers.
    (e) Click **Generate** to generate task types required for the process and design the process with these task types:

The system generates the following task types:

waitRestRequest waits until it receives a web service request from client: it accepts a request if the path of the request matches its path template. If there are several tasks waiting for a request then the task with a more precise path template handles the request. Once a task accepts the request, it finishes, and the process instance execution proceeds.

The *waitRestRequest* task defines the following parameters:

- input: input received from the client

- requestId: ID of the call
  The ID is used by the response tasks to identify the call; the value is generated by the LSPS Server.
- parameters with path variables as defined in the web-service definition
- htttpHeaders (optional): parameter with received HTTP header
- requestParams (optional): parameter with received request parameters

sendRestResponse sends a response to the request with the request ID defined by its parameter.

The task defines the following parameters:

- output: response sent to the client
- requestId: ID of the call

sendRestErrorResponse sends a fault message to the client when the received request call is evaluated as incorrect or fails.

The task defines the following parameters:

- error: error sent in the error response to the client
- requestId: reference to the ID of the call

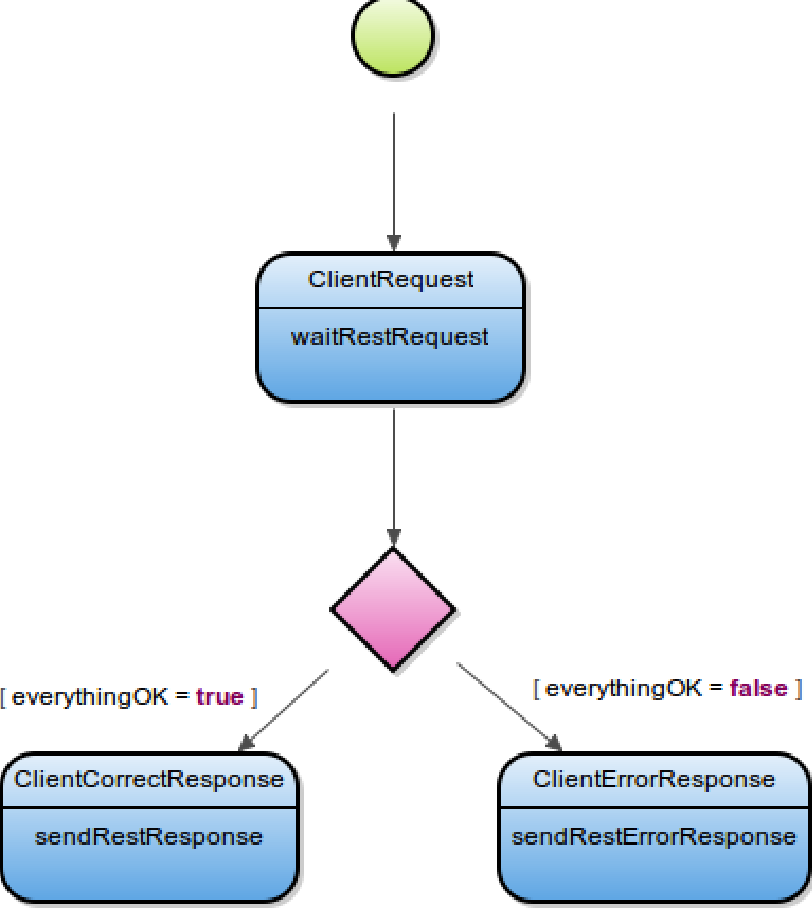


Figure 2.92 REST Server Process

Deploy the Model with the Process and run its instances as necessary.

Important: If the timeout period elapses and the server has not sent any response, the server sends a timeout response to the client. The default time out is set to 10 seconds.

2.15.1.2 Creating REST Client Process

To create a Process that will serve as a REST client, use in the workflow the `HttpCall` task type or use your `own task type`.

Make sure to define all required parameters of the task.

Example GET HTTPCall task parameters

```
httpMethod /* String */ -> "GET",
url /* String */ -> "http://localhost.com:9500/3_2/section/_search",
request /* Object */ -> "{""query"":{""term"":{""title"":""model""}}}",
requestContentType /* String */ -> "application/json",
isSynchronous /* Boolean */ -> true
//equivalent to:
//curl -H "Content-Type: application/json" -X GET
//http://localhost.com:9500/3_2/section/_search
//-d '{"query":{"term":{"title":"model"}}}
```

Example POST HTTPCall task parameters

```
httpMethod /* String */ -> "POST",
url /* String */ -> "http://localhost:8080/lsps-ws/rest/client/123",
request /* Object */ -> convertToJson(new Input(int -> 333)),
requestContentType /* String */ -> "application/json" //"application/xml",
response /* Reference<String> */ -> &response,
responseCode /* Reference<Integer> */ -> &responseCode,
login /* String */ -> "admin",
password /* String */ -> "admin",
readTimeout /* Integer: timeout in ms */ -> 9999,
requestHeaders /* Map<String,String> */ -> [ "Accept" -> "application/json"],
responseHeaders /* Reference<Map<String,String>> */ -> ,
isSynchronous /* Boolean */ -> false
```

2.15.2 SOAP Webservice Processes

To implement a process that will serve as a SOAP server or client, you need to create the resources that represent the web service operations and design the web-service process.

You will generate Task types that implement the web service communication and will act either as a Web Service Server or a Web Service Client):

- Web-service client tasks:
  - For every operation, 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.

2.15.2.1 SOAP Server Process

To design a process that will serve as a SOAP web-service server, you need to:

- generate task types that can handle the client web-service call:
  - *waitForInvoke* waits for a client call
  - *sendResponseToInvoke* sends a response to the client call
  - *sendErrorToInvoke* sends an error response to the client call
- design a process with the tasks of these types that will serve such client request.

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).

You can check and manage Model instances that are serving a web service call in the `Web Services view` or `Management Console`.

2.15.2.1.1 Creating a SOAP Server Process from Scratch

To generate task types and design a process acting as a SOAP web-service server without a WSDL, 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 the data structures must be defined as Records even if they contain only one field of a primitive type.
2. Check the XML mapping of the records, their fields, and relationships on the *XML Mapping* tab of *Properties* to define how to generate the XSD and WSDL files:
   - To allow empty tags, such as, `<ns1:someValue xmlns:ns1="urn:lsps:myws"/>`, select the "Use xsi:nil flag" on the XML Mapping of the field or record: this translates to the `nillable="true"` attribute in the generated *xsd* file. The empty tag is sent when the value of the field or record is `null`.
   - To allow a field or record to be missing, select the *Optional* flag: this translates to the `minOccurs="0"` attribute of the XSD tag so if the field or record is absent, no tag is inserted.

   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) In the web service editor, define the web service properties:

- Service name: name used as the service name and the task name in the generated task types The target service URL is resolved as `<LSPS_Server_URL>/lsps-ws/service/<MODULE_NAME>/<Service_Name>`.
- Service namespace: namespace of the Web service used in the generated WSDL
- Input type: input data type you defined
- Output type: output data type you defined
- Soap fault types: fault data types you defined

(c) Select the options for applicable settings of headers.

(d) Click **Generate** to generate 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:

- **waitForInvoke** waits until it receives a web service client call: Once the task has received a client call, the task stores the client input and finishes. The task has 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 accessible from 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** sends a response to the wait point of the request ID. The task has 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 accessible from the `management tools`.

    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** sends a fault message to the client in cases the received request call is considered incorrect or failed. The task has 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 accessible from the `management tools`.

    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

    Important: Make sure to define the id parameters on the **waitForInvoke**, **sendResponseToInvoke**, and **sendErrorToInvoke**. Failing to do so might result in transaction rollback.

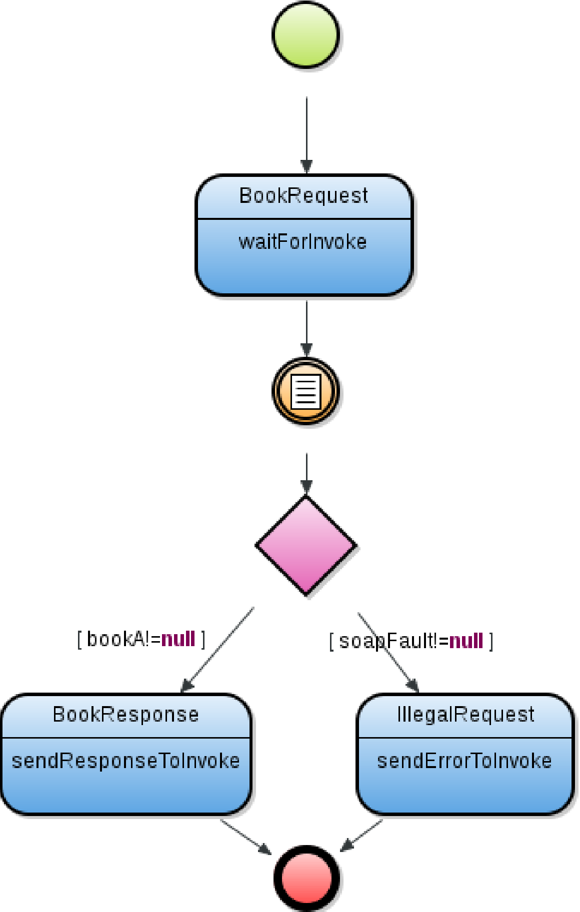


Figure 2.93 Workflow of a process serving as a web service server

5. 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.

2.15.2.1.2 Creating a SOAP Server Process from WSDL

To generate task types and the underlying data types for 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 > Web-service 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 storing input, output, and error data types that will hold the data during the web service invocation, in global variables.

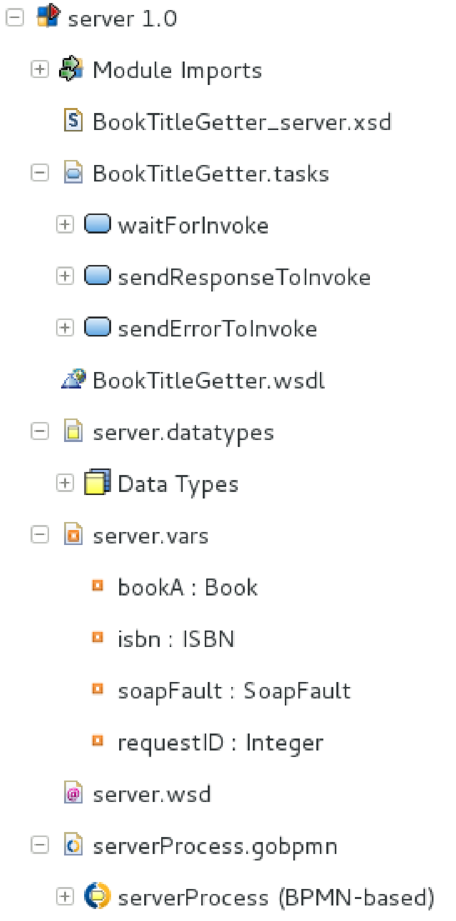


Figure 2.94 Web Server Service Module

3. Design the process workflow with the generated Webservice-Server Tasks.

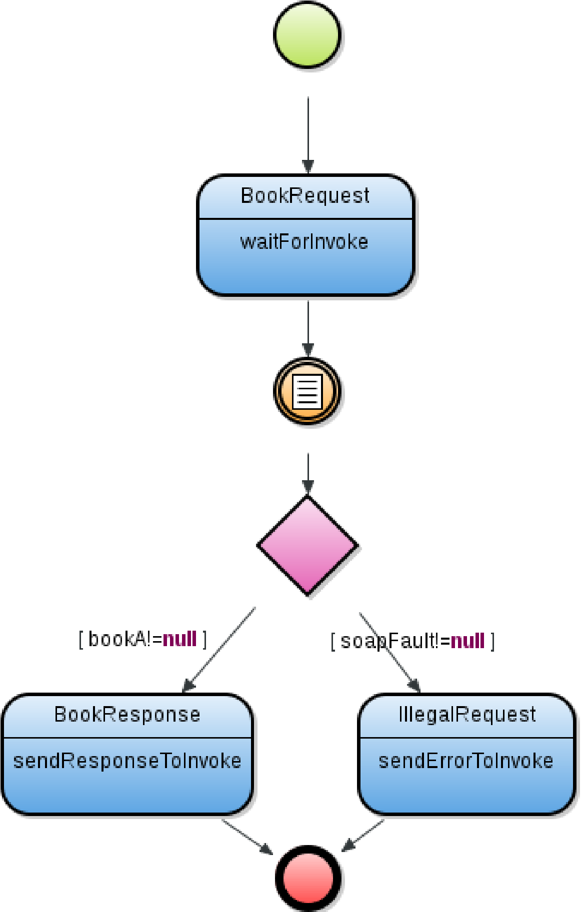


Figure 2.95 Workflow of a Process Serving as a Web Service Server

4. Expose the WSDL and XSD files as appropriate.

2.15.2.2 SOAP Client Process

To design a Process that acts as a web service client, you will generate artifacts based on the WSDL file of the target web service. These will include the following:

- invoke task types for each operation: The tasks of the invoke task type perform the web service operation calls.
- fault data types: Fault data types to store returned web service errors
- data types used by the generated task types

2.15.2.2.1 Creating a SOAP Client Process

Important: Resources for a web service client can be generated only for SOAP 1.1 web services.

To create data types and task types that will constitute you web-service client process, do the following:

1. Get the WSDL and any other related resources.
   Web service WSDL is usually available on a URL provided by the web service server.

2. When creating a web service client for a web service server *process*, include the XSD files.

3. In the GO-BPMN Explorer, right-click your Module and in the context menu, click Generate > Webservice Client.

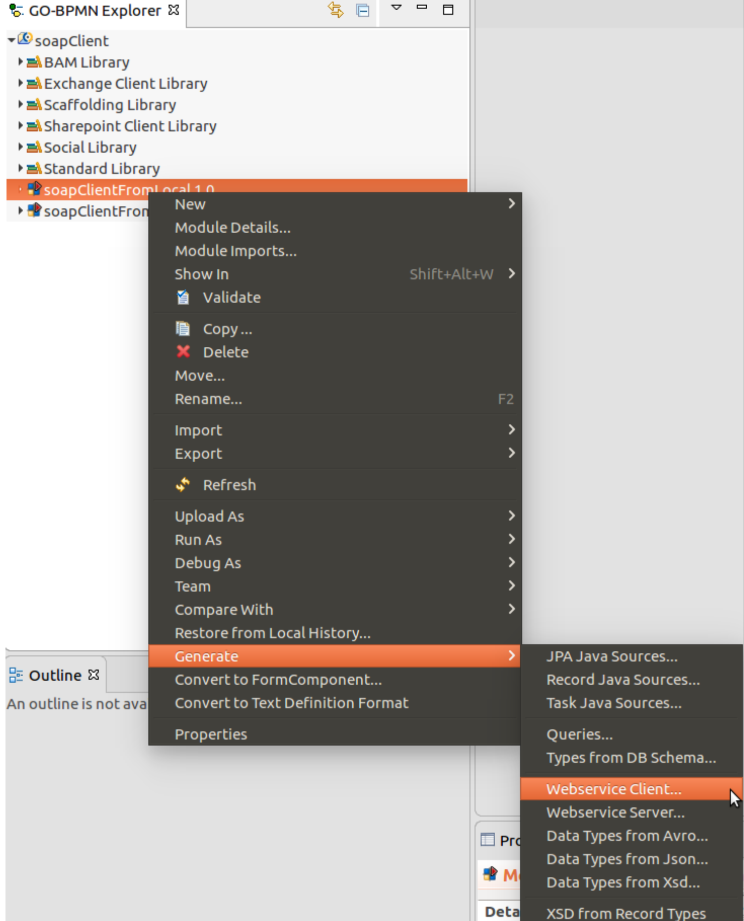


Figure 2.96 Generating artifacts for web service client

4. 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 options for the additional parameters: these will be generated as properties of the task types.

   (d) Click OK.

   (e) Select the required web-service operations: for every operation, one task type and the respective data types will be generated. Tasks of the task type call the web-service and request the operation.

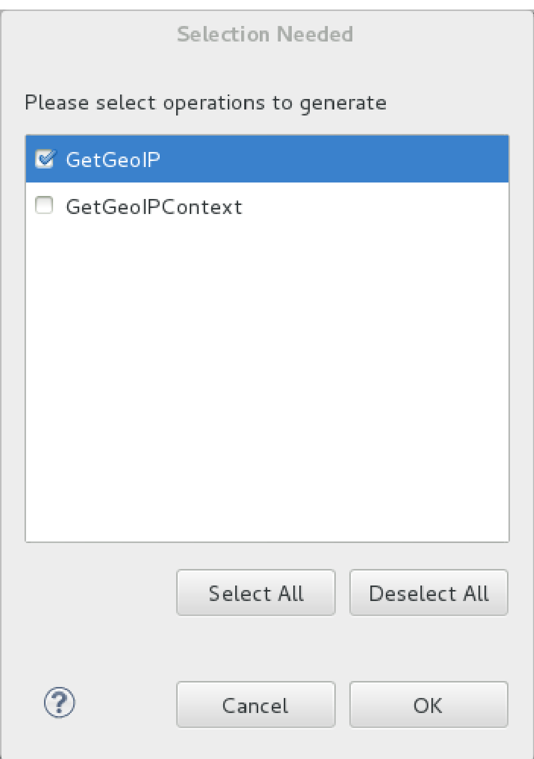


Figure 2.97 Selecting relevant operations

(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.

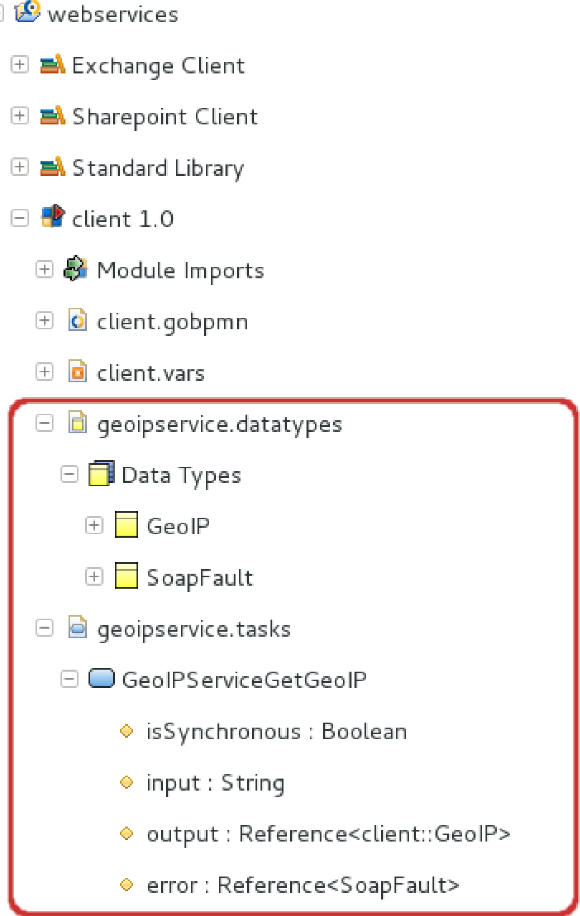


Figure 2.98 Generated Task Types and Data Types

5. Design a process with the generated task types. Each task type represents one WSDL operation with the operation's messages and data defined by its properties:

   - isSynchronous: if true the web service call is synchronous
     Synchronous task block the execution in the entire process until the server response is received, while asynchronous tasks allow other parts of the process instance to continue their execution (other tokens in the process continue to move); 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 returns 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, an Error 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

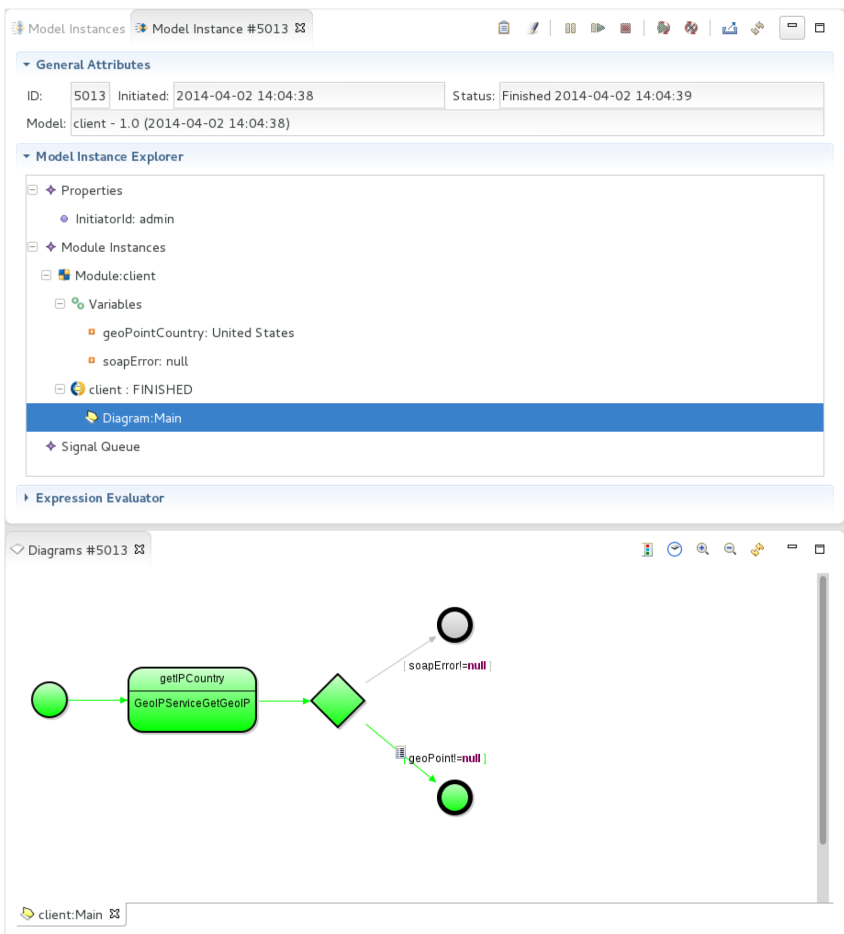


Figure 2.99 Model instance of a web service client Process

2.16 Jasper Reports Integration

The support for Jasper Reports is provided by the *reports* Module of the Standard Library: the Module contains the reports function definition with the following functions:

- *jasperReportUrl()* returns the URL of a Jasper Report on a Jasper server
- *embeddedJasperReportUrl()* returns the URL of the JasperReport in the LSPS Embedded Server

  Note: The LSPS Embedded Server integrates the Jasper Report library; hence no Jasper Server is called when working with the Embedded Server.
- *jasperReportPDF()* generates the PDF version of the report with the provided parameters

2.16.1 Integrating Application User Interface with a JasperServer

To work with Jasper Reports on a Jasper Server, you need to obtain access to reports on an external Jasper Server. You can do so with the *jasperReportURL()*.

Example URL of a Browser Frame component that points to a report on a remote Jasper server with no parameters

```
jasperReportUrl(
  "http://myremoteserverip:9090/jasperserver",
  "jasperadmin",
  "jasperadmin",
  "/reports/MyReports/myReport",
  false,
  ["topLimit"-> 120]
)
```

Important: Insert the full path to your report.

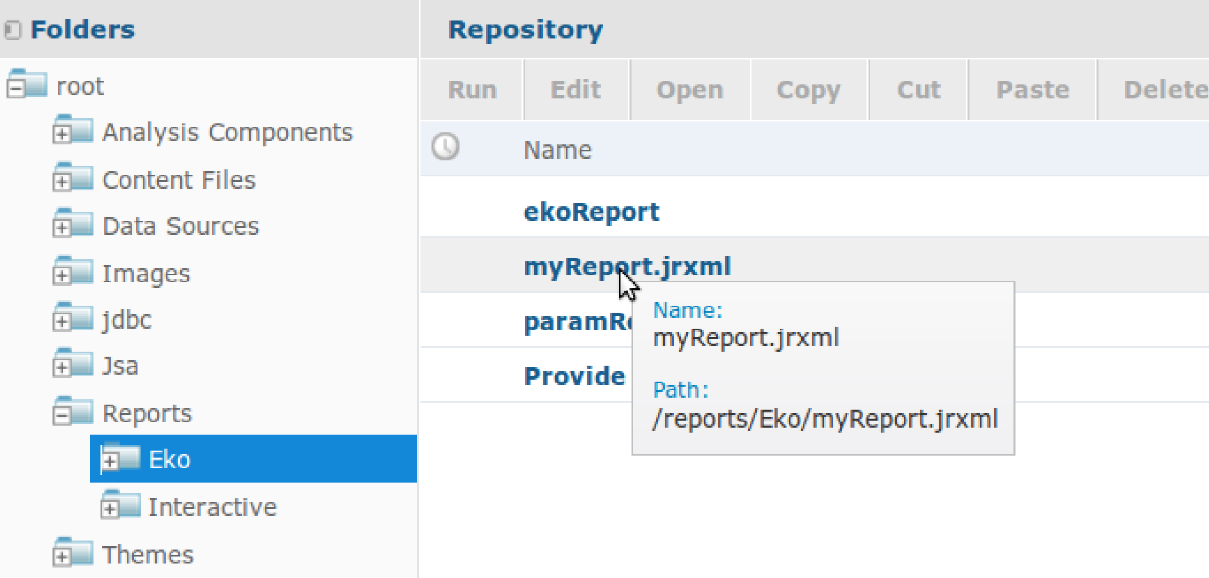


Figure 2.100 Path of the report in the web client

2.16.2 Enabling Expression Language in Jasper Reports

To enable data queries of Jasper Reports in Expression Language, do the following:

1. Add `lsps-tools/jasperreports-extension/lsps-tools-jasperreports-extension-<VERSION>.jar` to the classpath of your Jasper server (you can download its latest version with dependencies `here`).
2. Set the property *com.whitestein.lsps.datasource.url* to point to base URL to the RESTful API
   For the Embedded Server, this would be set to `http://localhost:8080/lsps-monitoring-ws/resources/rep`
3. Set the following parameters for your reports:
   - *query language* to `LspsExpression`
   - `LSPS_MODULE_ID` or `LSPS_MODEL_INSTANCE_ID` to define the context for the report data
   - authentication data as `LSPS_USERNAME` and `LSPS_PASSWORD`
4. Define the data query in the Expression Language. Make sure the Expression returns a string with results in a valid JSON format.

2.16.3 Displaying a Jasper Report

You will typically display a Jasper report in a form using a Browser Frame or you will navigate away from the Document or To-Do to the report URL: On the Embedded Server, you can use the `embeddedJasperReportUrl()` which returns the URL of the JasperReport: The LSPS Embedded Server integrates the Jasper Report library; hence no Jasper Server is called.

To display your report in a Browser Frame, do the following:

1. Add the report jrxml file to your GO-BPMN project.
2. Open or create the Form definition.
3. Insert a *Browser Frame* component into the Form.
4. Define its URL property with the URL to the report using *embeddedJasperReportUrl*
   - on a Browser Frame as target Url `embeddedJasperReportUrl(thisModel().name, "MyBamReport/HelloModel.jrxml", [ "editable"-> true])`
   - on a Listener of a component, such as a Button: `Forms.navigateToUrl((embeddedJasperReportUrl(thisModel().name, "MyBamReport/HelloModel.jrxml", [ "editable" -> true ])))`

2.16.3.1 Mapping of Parameter Data Types

When you request a parametric Jasper report with one of the *reports* function of the Standard Library, on runtime, the data type of the parameter value is converted to its Java equivalent:

Expression Language Type	Java Type
String	java.lang.String
Integer	java.lang.Short
	java.lang.Integer
	java.lang.Long
	java.lang.Float
	java.lang.Double
	java.math.BigDecimal
Decimal	java.lang.Short
	java.lang.Integer
	java.lang.Long
	java.lang.Float
	java.lang.Double
	java.math.BigDecimal
Boolean	java.lang.Boolean
Date	java.util.Date
List	java.util.List
Set	java.util.Set

2.16.3.2 Exporting a Jasper Report

To allow the user to export a Jasper Report into PDF, Excel, or Word and download it, call *jasperReportExport()* in a form component. The function returns a File with the report.

Example FileResource of a Download component with a PDF report

```
new FileResource(jasperReportExport(thisModel().name, "MyBamReport/HelloModel.jrxml", ReportFormat.pdf ,
      [->]));
```

Example Click Listener of a *Generate PDF* button that generates the PDF report and sets the PDF Link component properties

```
{e ->
   def FileResource pdf :=
     new FileResource(
       jasperReportExport(
         thisModel().name,
         "MyBamReport/HelloModel.jrxml",
         ReportFormat.pdf,
         [->]
       )
     );
   PdfLinkComponent.setResource(pdf);
   PdfLink.Component.setEnabled(true);
}
```

2.17 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 Designer, or as XPDL files.
- Libraries

  Modules exported as libraries are intended to shared by users as as resources of reusable elements.

2.17.1 Module Export

You can export your Modules as follows:

- 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 Designer (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.

2.17.1.1 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.

2.17.1.1.1 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.

2.17.1.2 Exporting a Module 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.

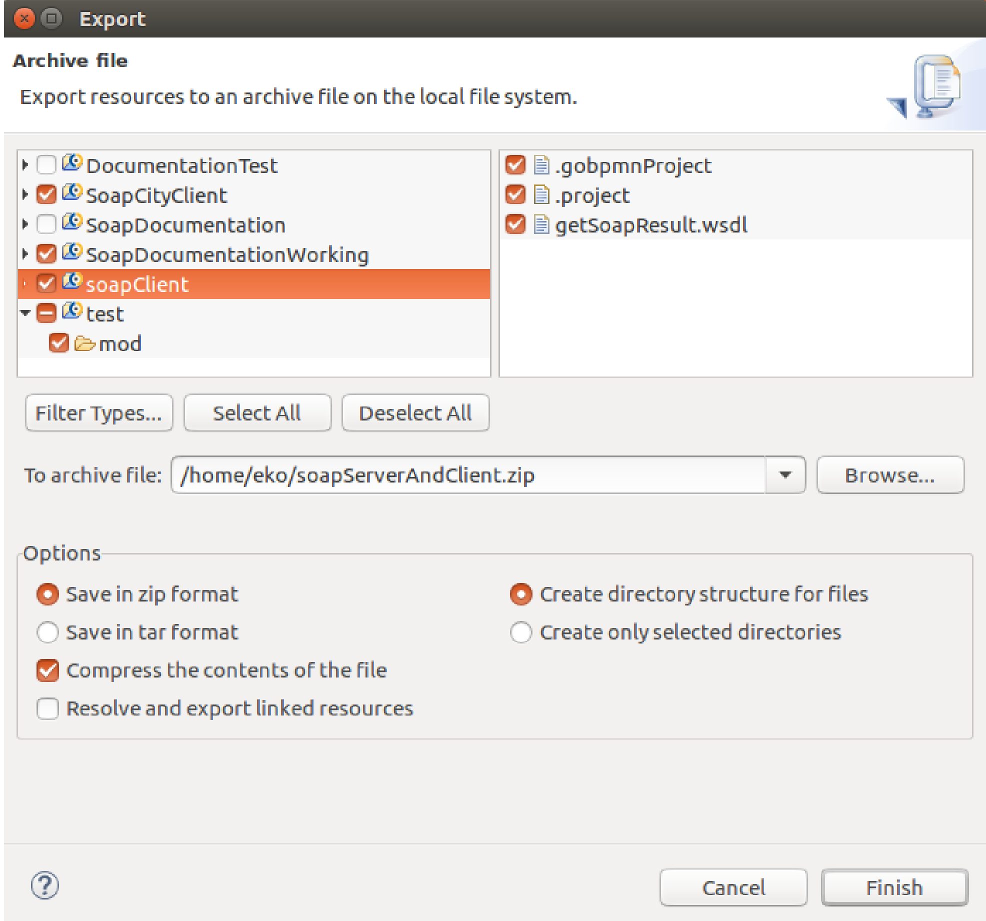


Figure 2.101 Selecting resources and files to be exported into an archive

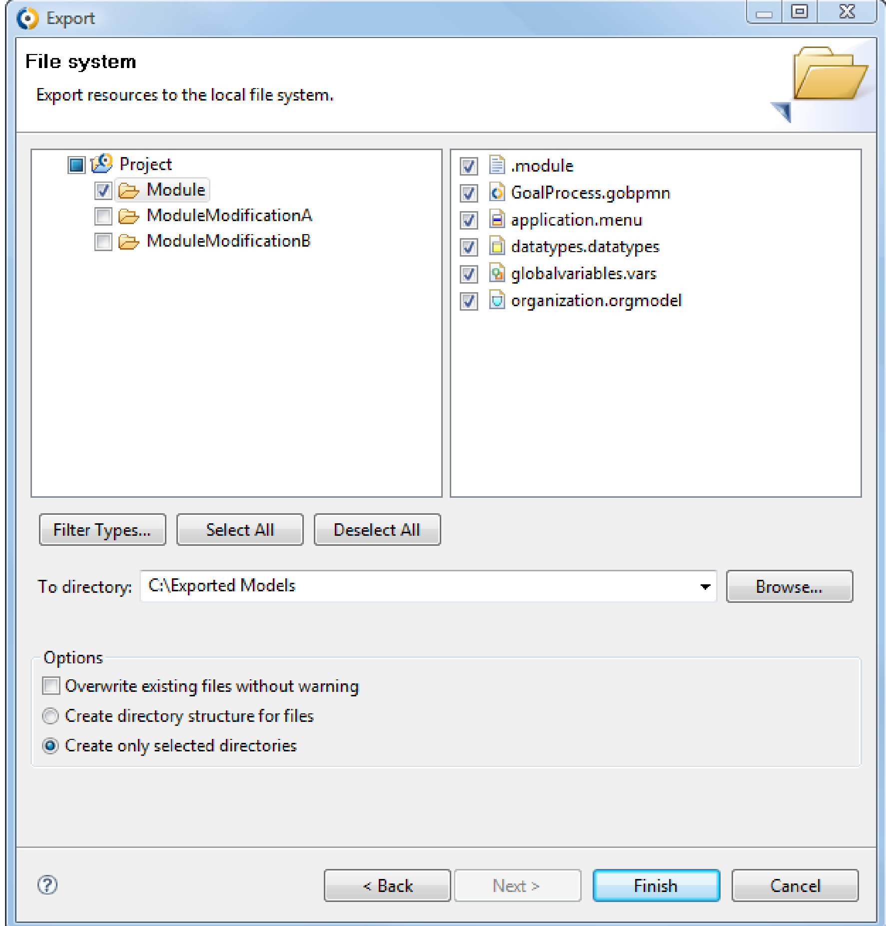


Figure 2.102 Selecting resources and files to be exported into a file system

4. In the Destination file text box, specify the path and the `archive name` of the file, for example, `archive.zip`.

2.17.1.3 Exporting a Module with GO-BPMN Export

The `GO-BPMN Export` feature allows you to create zip files with a Module, which can be deployed to the server: the output file contains the Module with all its dependencies.

> **Important:** If you are exporting Models intended for execution, make sure the respective Module is flagged as *executable* in its Properties (in GO-BPMN Explorer, an executable Module is indicated by a red arrow).

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 Designer and select GO-BPMN Export.

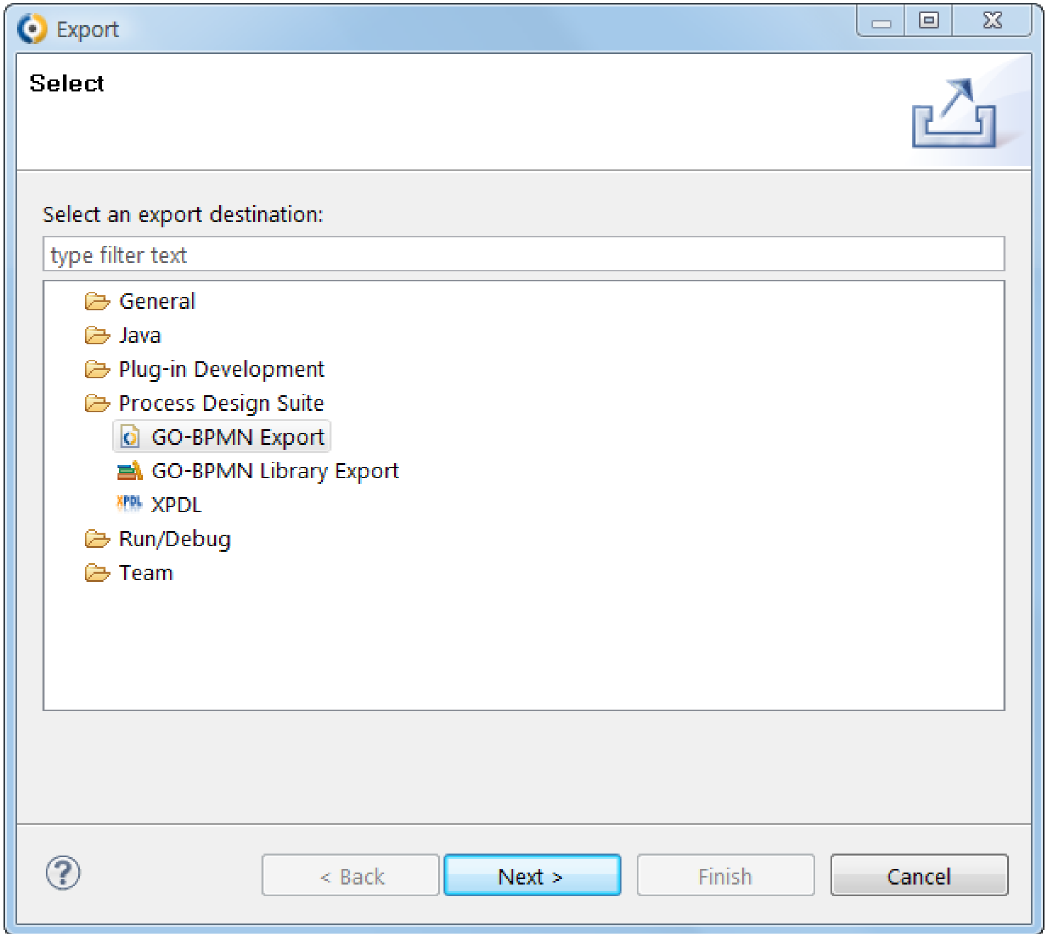


Figure 2.103 Selecting the GO-BPMN Export

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).

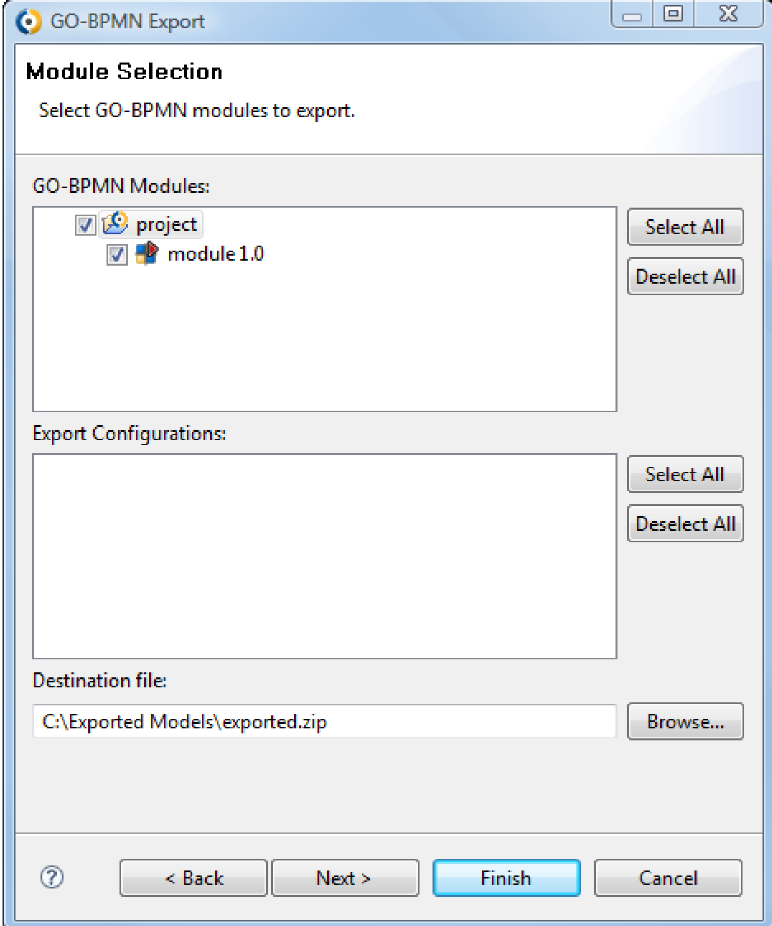


Figure 2.104 Selecting modules to be exported

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.

2.17.1.4 Exporting a Module 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 Designer 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;

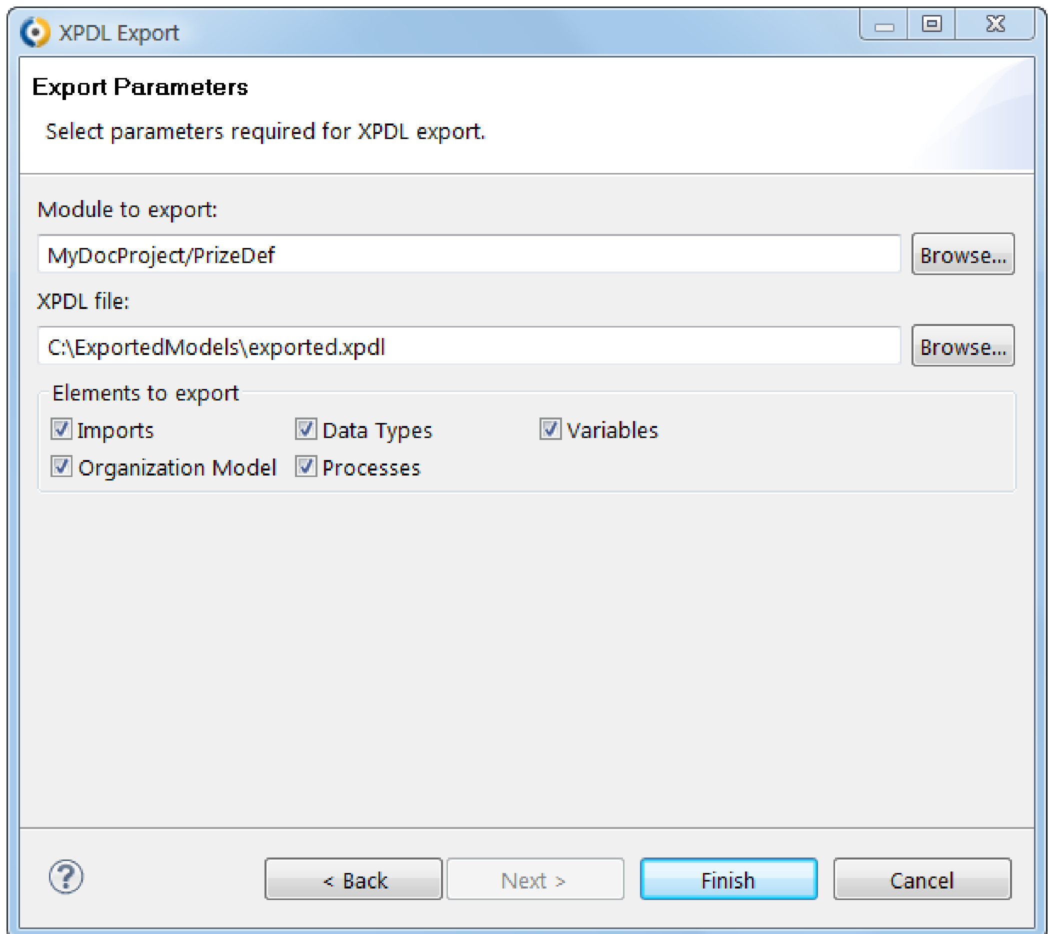


Figure 2.105 Defining XPDL Export settings

5. Click Finish.

2.17.2 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).

2.17.2.1 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.

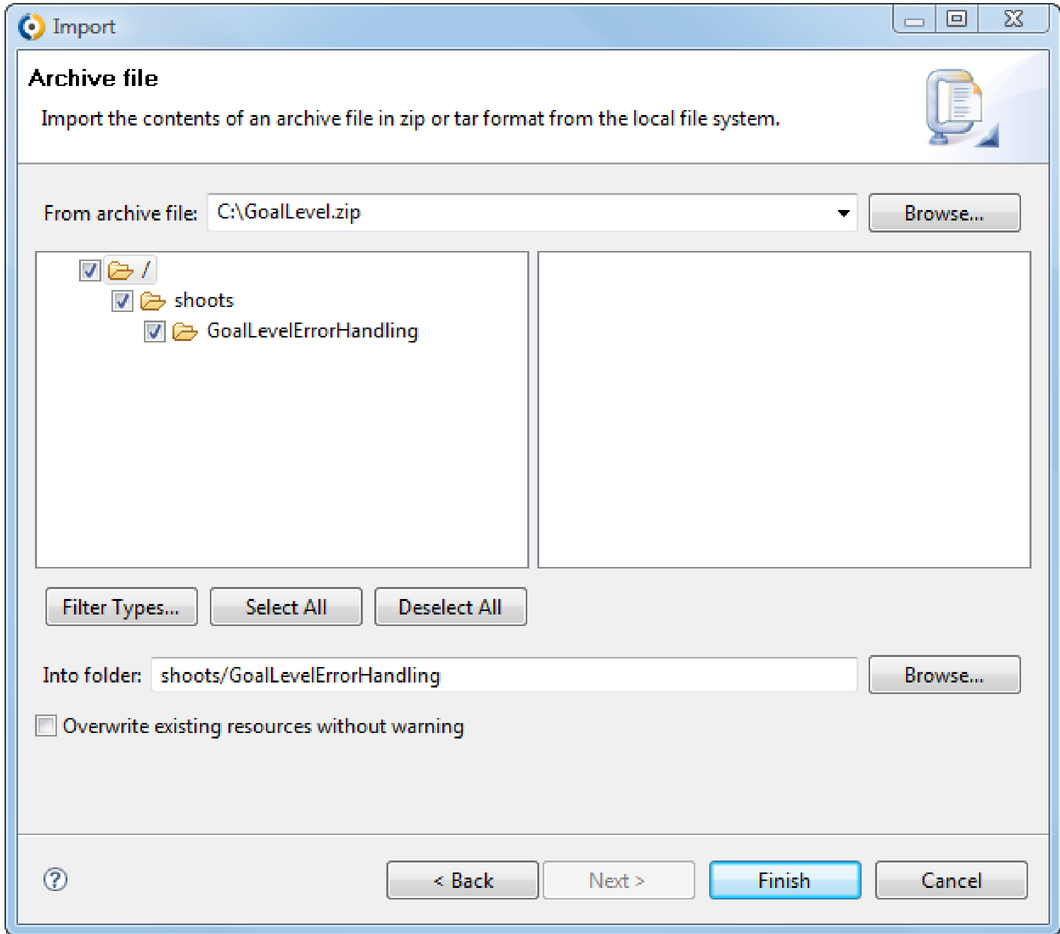


Figure 2.106 Importing an archive

5. Click Finish.

The model package content is imported to the workspace.

2.17.2.2 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.

2.17.2.3 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.

2.17.3 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

2.17.3.1 SharePoint and Exchange Client Libraries

Support for 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.

> **Important:** LSPS uses the mail server configured in your application server. These libraries serve to communicate with other parties from your models.

> **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).

2.17.3.2 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.

2.17.3.3 Creating 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.

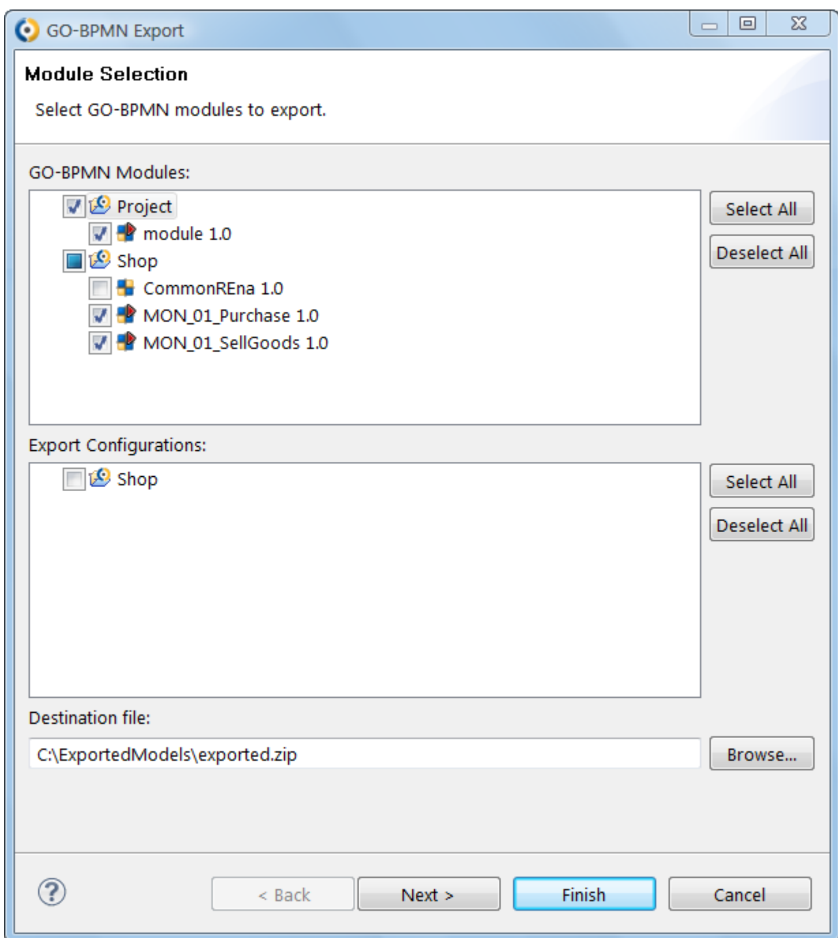


Figure 2.107 Selecting modules to be exported as a library

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.

2.17.3.4 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.

2.18 Presenting a Model

To present your Models to an audience, use the Business Modeling perspective: its layout is adapted to avoid unnecessary views and elements.

Chapter 3

Model 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 used to create a model instance.

To upload modules from Designer, first connect Designer to a server with LSPS Application Once you have established the connection, you can upload a model to the server and execute it.

From Designer, you can run your model in debug mode on the server you are connecting to presuming the underlying JVM runs in debug mode and check performance using profiler.

You can use also the command-line console as well as the Management Console to deploy and instantiate your model. However, you need to export your Model as a deployable bundle first.

When you upload a module with a data model that contains persisted data structures (shared Records), the 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 such situations are handled:

- **Do not change**: data-model tables remain unchanged.
- **Update by model**: New tables, columns and relationships are added; no data is deleted.
- **Validate**: data-model tables are validated against the new model. In case of inconsistencies, the model upload or launching fails.
- **Drop and create modeled DB tables**: 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 Designer

Designer connections to LSPS servers are set under **Server Connections** > **Server Connection Settings**. The connections to Designer and SDK embedded servers are added automatically when the servers are generated: For the Designer 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 Designer to an LSPS Server

To connect Designer to a server with LSPS Application other than Designer or SDK Embedded Server, do the following:

1. Set up the connection configuration:
   (a) On the menu bar, under **Server Connections**, click **Server Connection Settings**.
   (b) In the *Server 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 Designer Embedded server `http://localhost:8080`)
      - 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 configuration of model 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 For further details, refer to database schema strategy description.
   (c) Click **Test Connection** to make sure the settings are correct.
   (d) Select the connection to connect to Designer to the server.

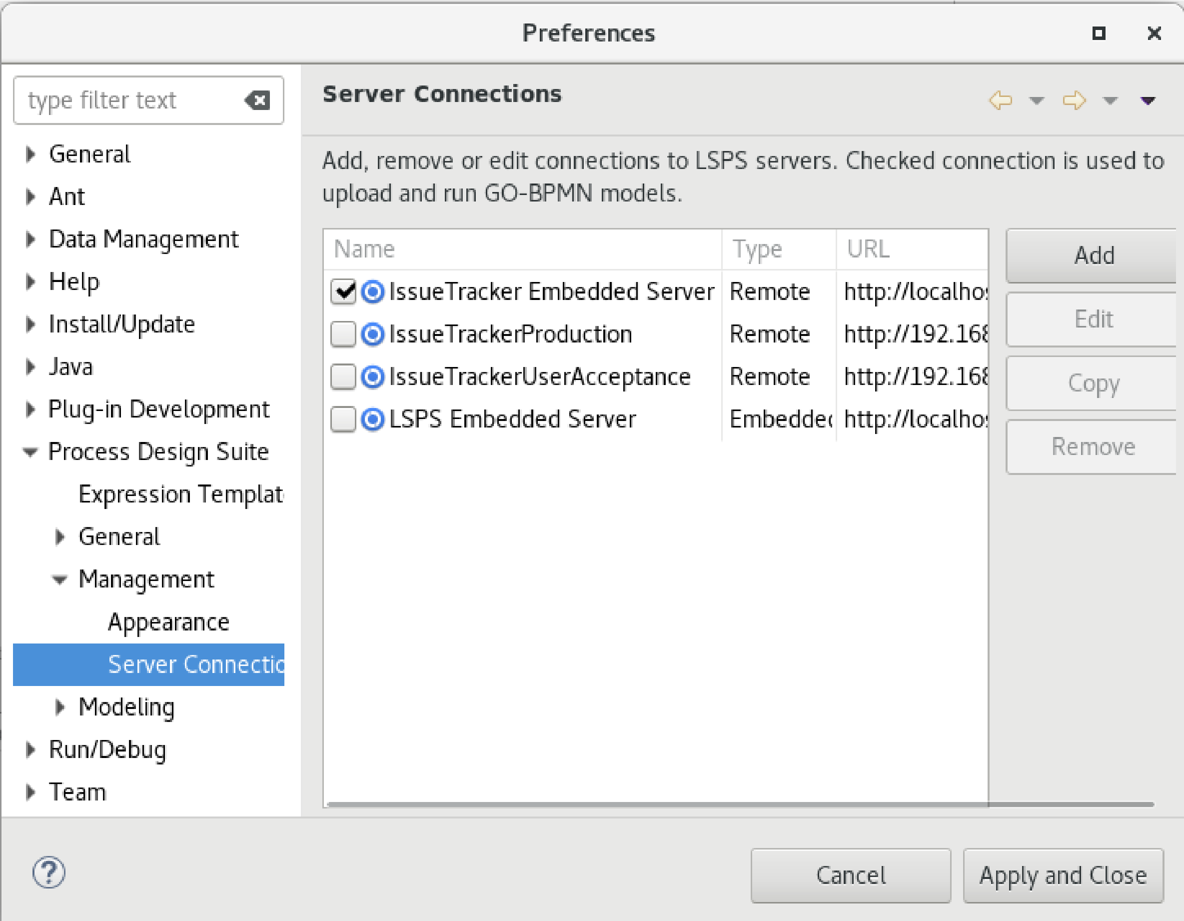


Figure 3.1 The remote connection set as preferred

2. If required, define a proxy server for your connection.
3. To change the connection, go to **Server Connections** > **Select**.

3.1.2 Connecting Designer to an LSPS Server using a Proxy Server

To use a proxy server to connect from Designer 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.
5. To connect to the server, go to **Runtime Connections** > **Select** and select the connection.

3.1.3 Connecting Designer to Designer Embedded Server

The Designer Embedded Server is a locally running server installed in the current workspace with the LSPS Application already deployed and an embedded database.

To install and launch a Designer Embedded Server, and to connect Designer to it, do the following:

1. Go to **Server Connections** -> **LSPS Embedded Server**.
2. Select **Start**: Designer will install the server, run it, and connect to it.

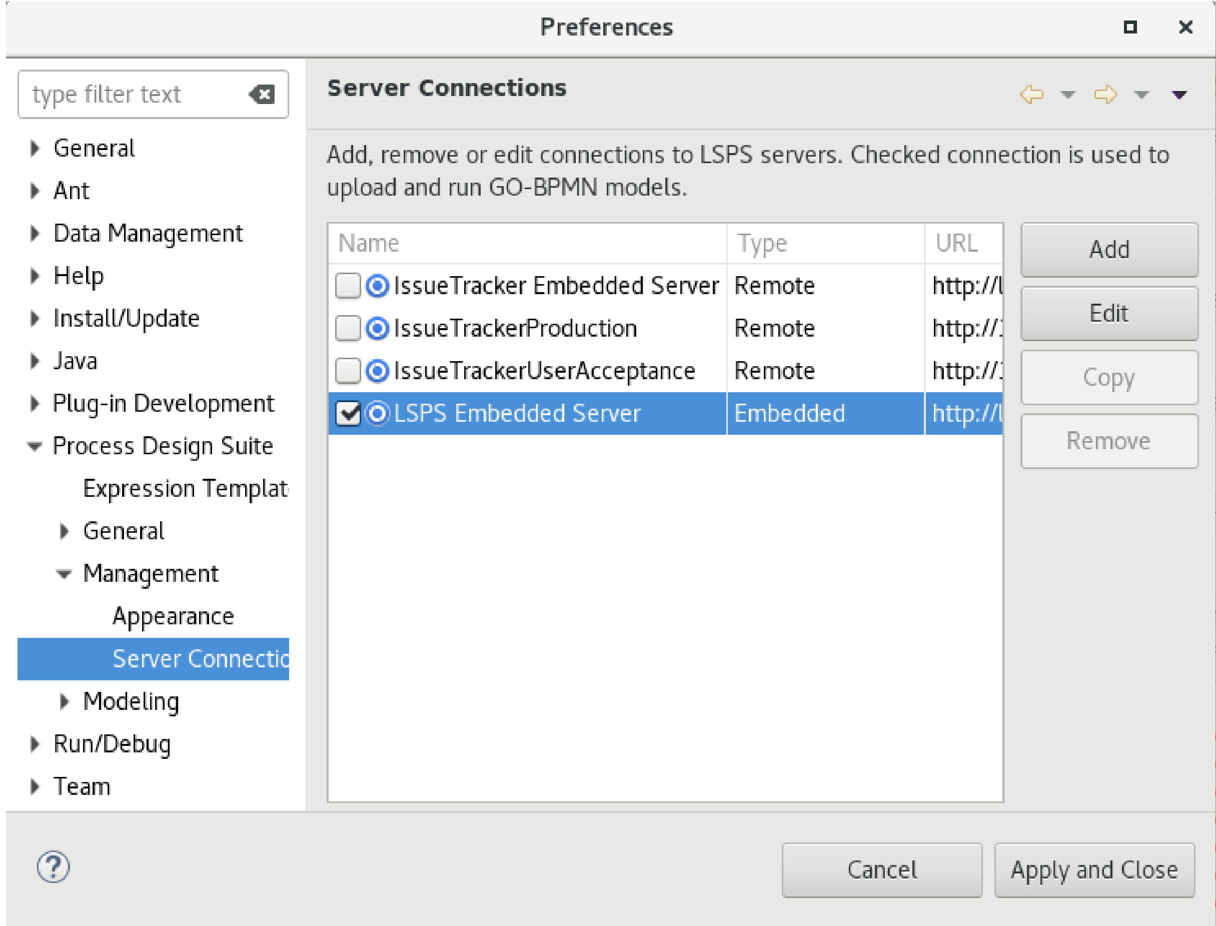


Figure 3.2 Server Connection page of the Preferences dialog box with connection to Designer Embedded Server as default

Alternatively, click the Start Embedded Server ( ) button on the main toolbar.

The server is installed 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 Designer 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 **Server 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.

3.1.5 Configuring Mail Server on Designer Embedded Server

For configuration of mail sessions for Designer Embedded Server, which is a WildFly server, use the server web console at `localhost:9990/` (if you have not done so yet, create a user using `add-user.sh` or `add-user.bat` in `<WORKSPACE>.LSPSEmbedded/wildfly-<VERSION>/bin`).

How to configure mail servers for your deployment environment is documented in the `Deployment and Configuration guide`: locate the information within the section for your server and for your SDK Embedded Server in the `LSPS Application Customization` guide.

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 Designer 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, Designer automatically creates an upload configuration, which is then used for upload.

When uploading a Module from the Modeling Perspective, the target server is by default the Embedded LSPS Server. To change the target server, go to **Server Connections** > **Server Connection Settings** and in the dialog select the default LSPS Server.

3.2.1 Uploading a Module from the Modeling Perspective

> **Note:** Ensure that the server with LSPS Server is running and your Designer is connected to it (Information on the current connection is in the line at the bottom of Designer).

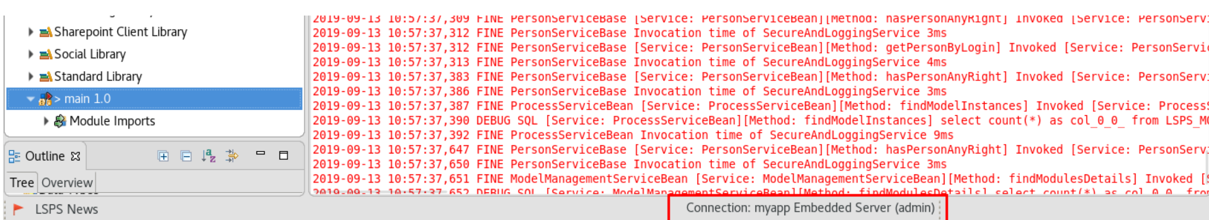


Figure 3.3 Address line with connection established

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.

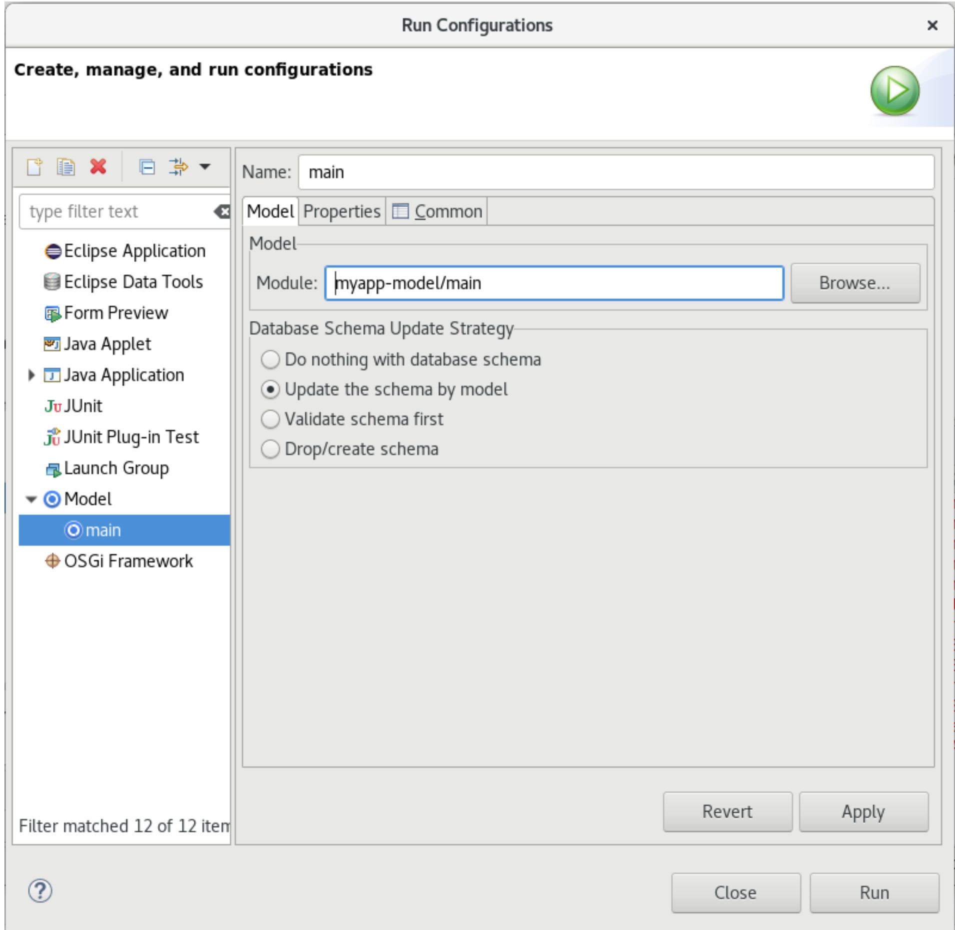


Figure 3.4 Run and upload configuration for the main module

3.2.2 Uploading a Module from the Management Perspective

Note: Ensure that the server with LSPS Server is running and your Designer is connected to it (Information on the current connection is in the line at the bottom of Designer).

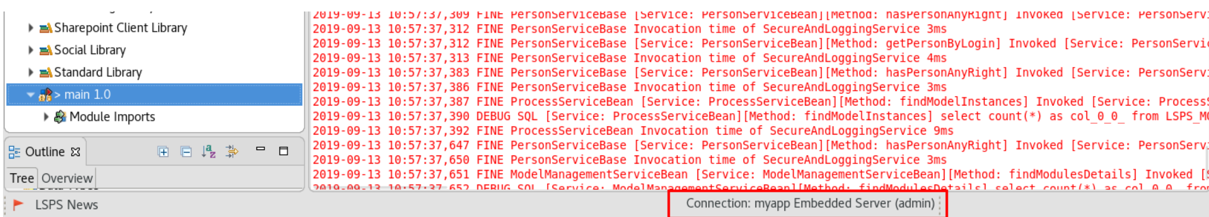


Figure 3.5 Address line with connection established

To upload a module to the LSPS Server to which your Designer 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 create model instances from Management Console or the Management perspective, but Designer allows you to upload and create model instances directly from your workspace.

When you create a model instance, the LSPS Server creates a runtime version of the model. Each model instance is created with a unique ID and optionally a set of initialization properties in the form of key-value pairs. These can include a special property, process entity, which is an id of a shared record and represents the data that the model instance works with.

For information on Model instance lifecycle, refer to the `GO-BPMN Modeling Language User Guide`.

3.3.1 Creating a Model Instance 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 Designer feature, which appear primarily in the Console view.

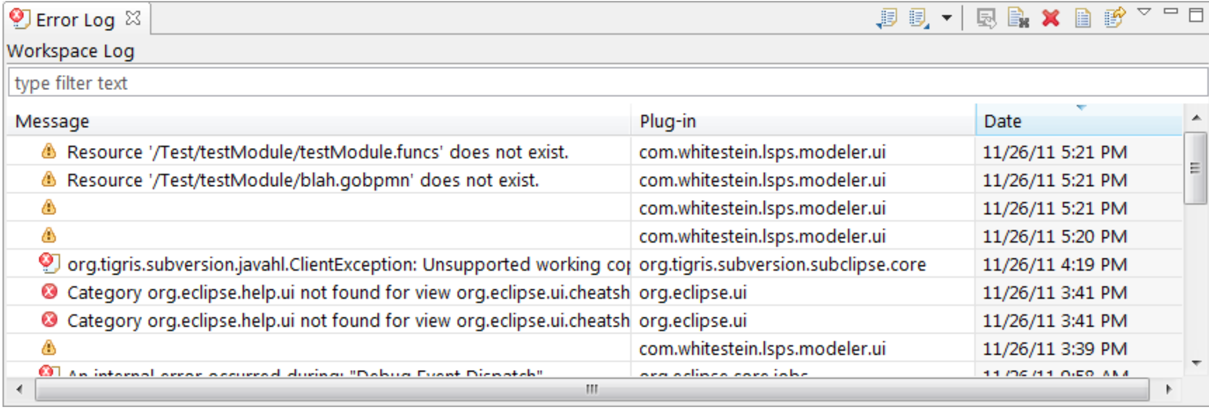


Figure 3.6 Error Log with error entries

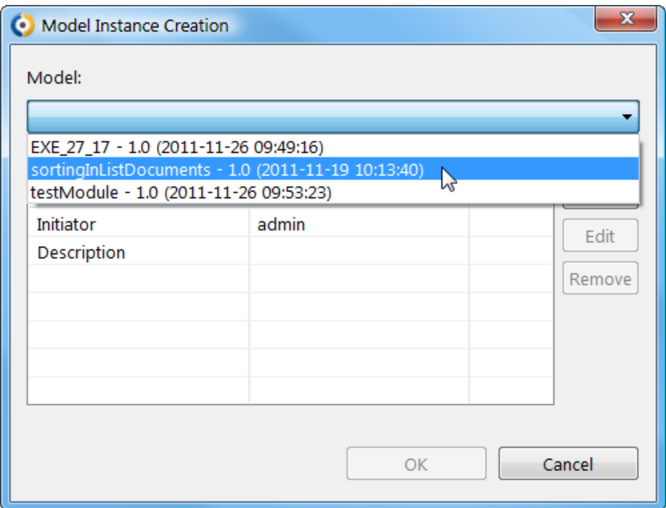


Figure 3.7 Creating a model instance based on an uploaded executable module

3.3.2 Creating a Model Instance from the Modeling Perspective

Note: Ensure that the server with LSPS Server is running and your Designer is connected to it (Information on the current connection is in the line at the bottom of Designer).

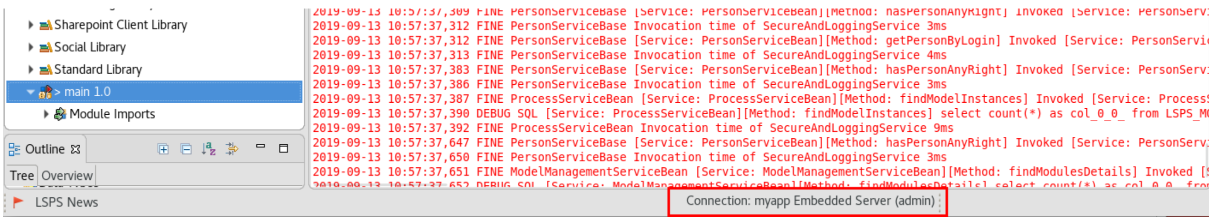


Figure 3.8 Address line with connection established

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.

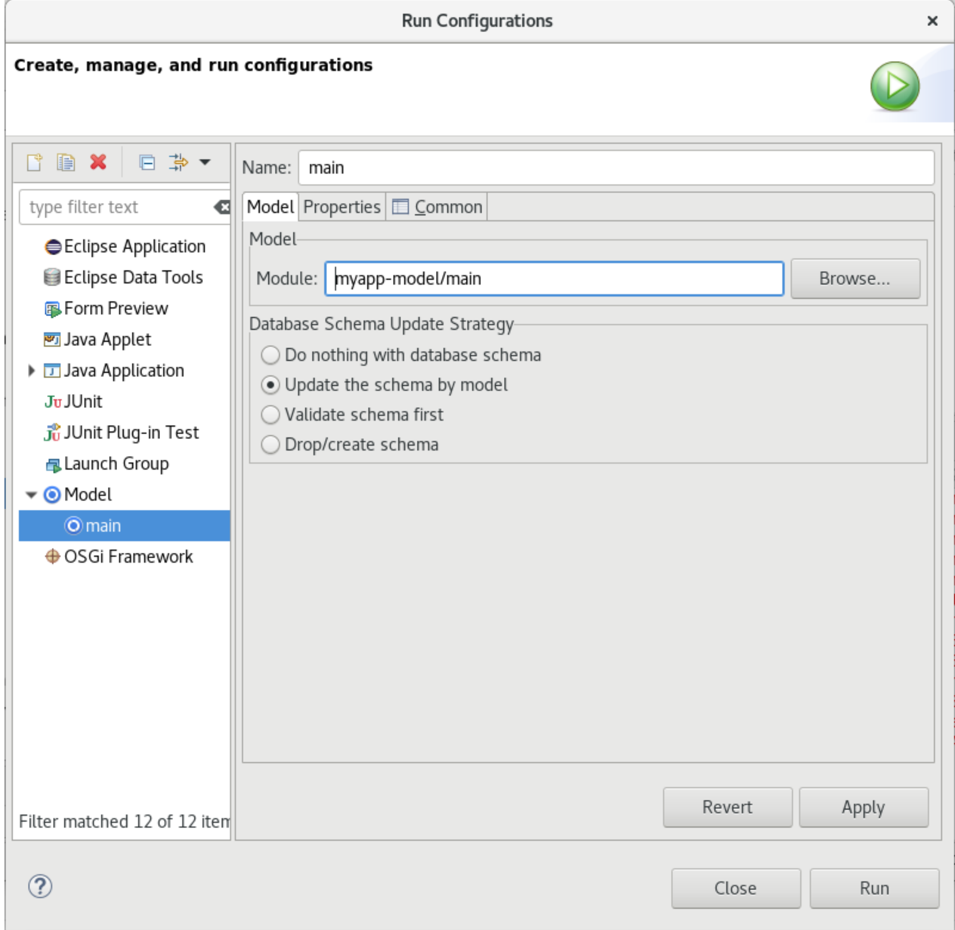


Figure 3.9 Run and upload configuration for the main module

3.4 Debugger

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.

3.4.1 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 Designer. 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 Designer 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

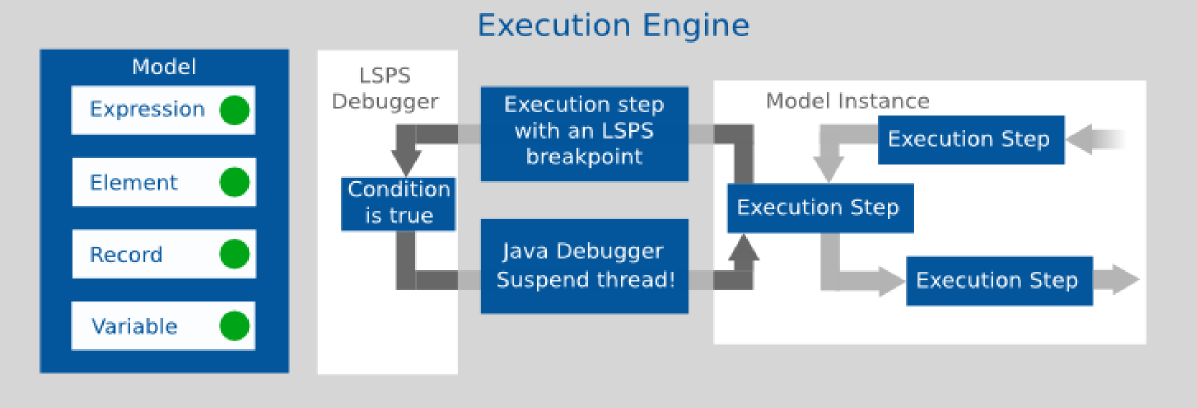


Figure 3.10 LSPS debugger implementation

3.4.2 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 Designer 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 Designer Embedded Server is integrated in Designer and it is sufficient to modify the server properties.

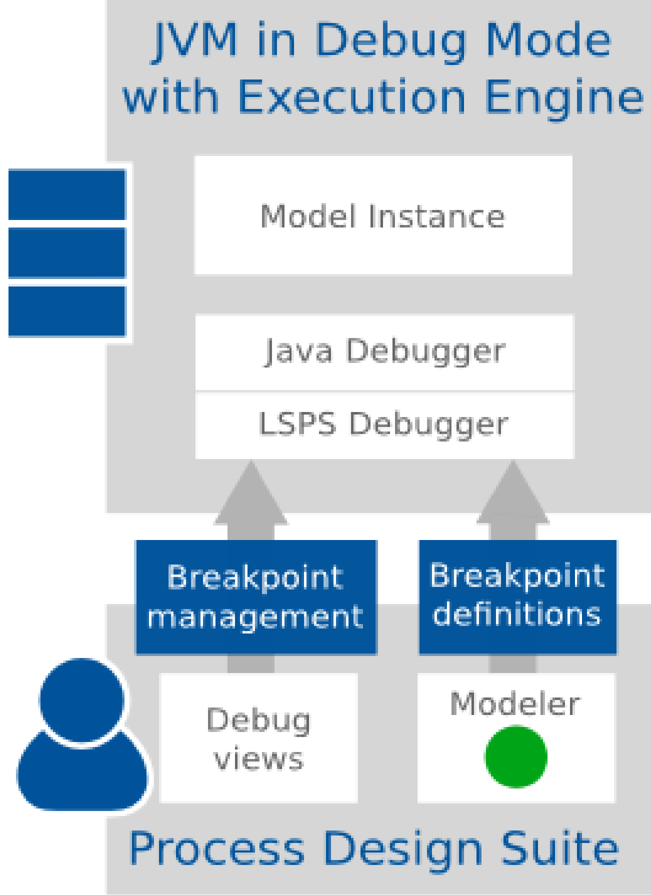


Figure 3.11 Debugger schema

3.4.2.1 Debugger on Designer Embedded Server

Note: The instructions are summarized also in a `screencast`.

When debugging on Designer Embedded Server, Designer connects to the server as to a remote server.

To run Designer Embedded Server in debug mode, do the following:

1. Go to **Server Connections** > **Server 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.

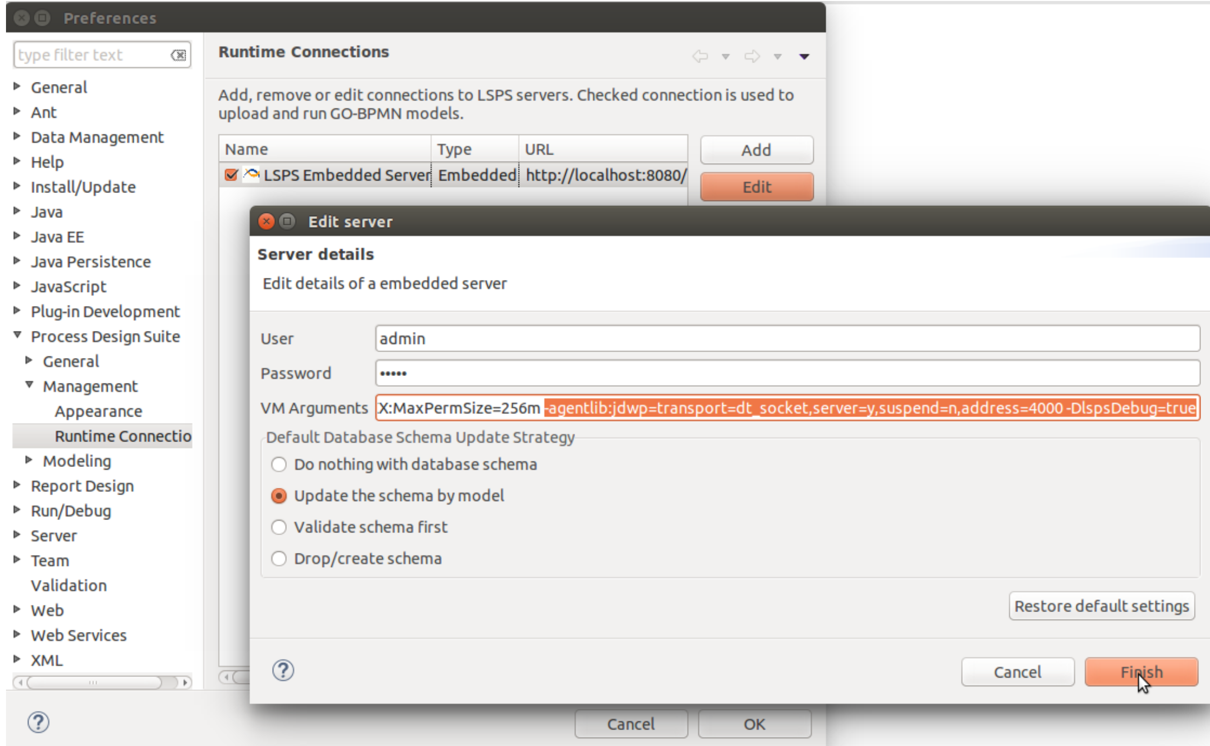


Figure 3.12 Debug parameters in Designer Embedded Server configuration

> **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 Designer 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).

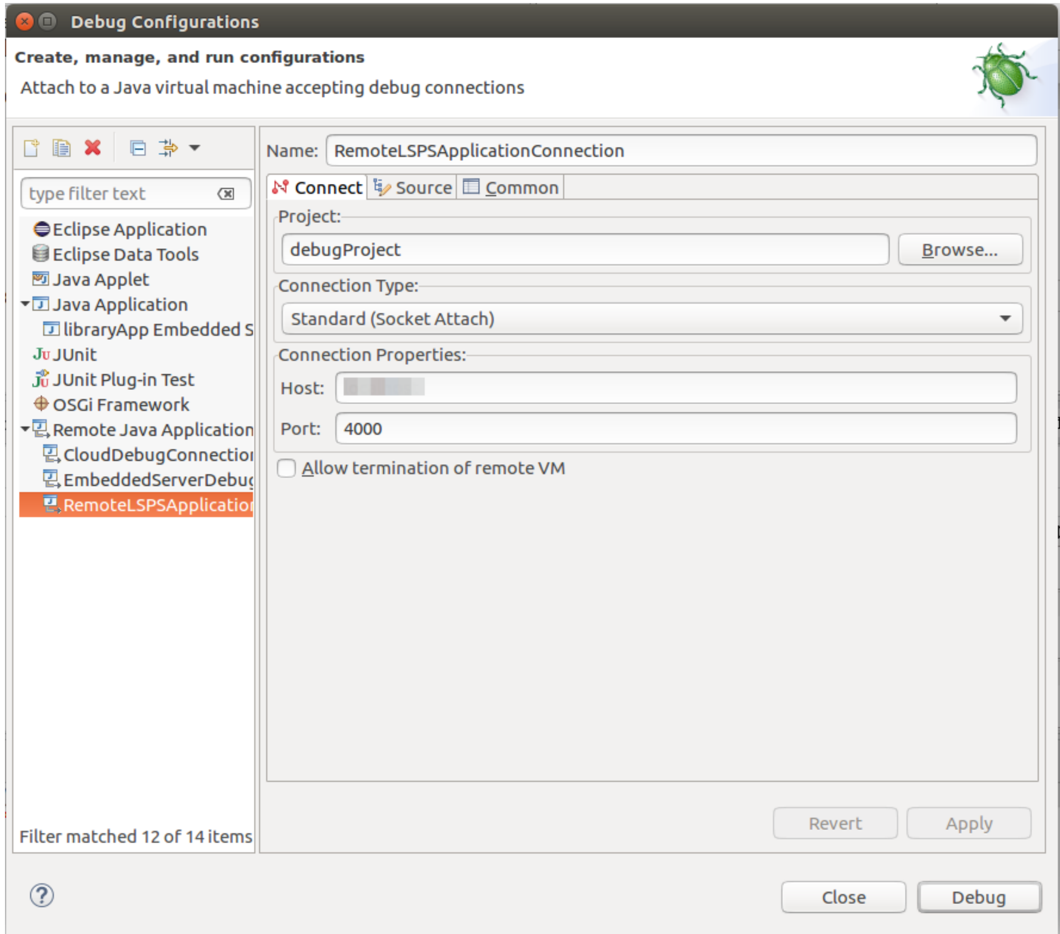


Figure 3.13 Setting debug configuration

(d) Run the debug configuration to attach the LSPS Debugger to the server.

6. Enable debugging: click the **Debug** button in the LSPS Breakpoints.

Now you can debug the model: add breakpoints to the model and run the model.

> **Important:** If your breakpoints are on expressions or code, not visual modelling elements, the debugger might be stuck before reaching the breakpoint: in such a case, double click the first method in the current thread to refresh the views.

3.4.2.2 Debugger on SDK Embedded Server

> **Note:** The instructions are summarized also in a `screencast`.

To debug your Custom Application User Interface locally on SDK Embedded Server, do the following:

1. Open the *Debug* perspective.
2. Go to Run > Debug Configurations > Java Application > <YOUR_APP_LAUNCHER> (for example, `MyCustomApp Embedded Server Launcher`).

3. Edit the configuration: add the `-DlspsDebug=true` VM argument.

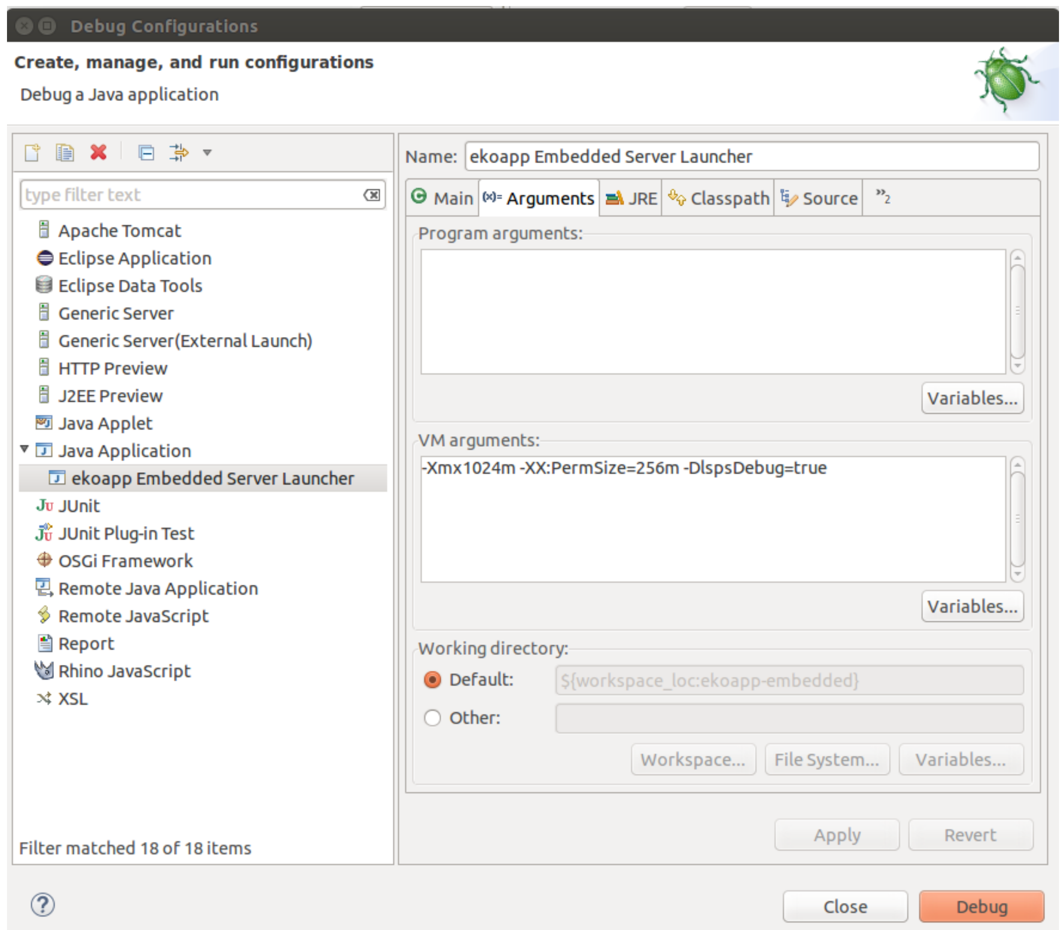


Figure 3.14 Setting application launcher configuration to debug mode

Under the hood, the connection is established on a free port automatically.

4. Run the launcher configuration in debug.

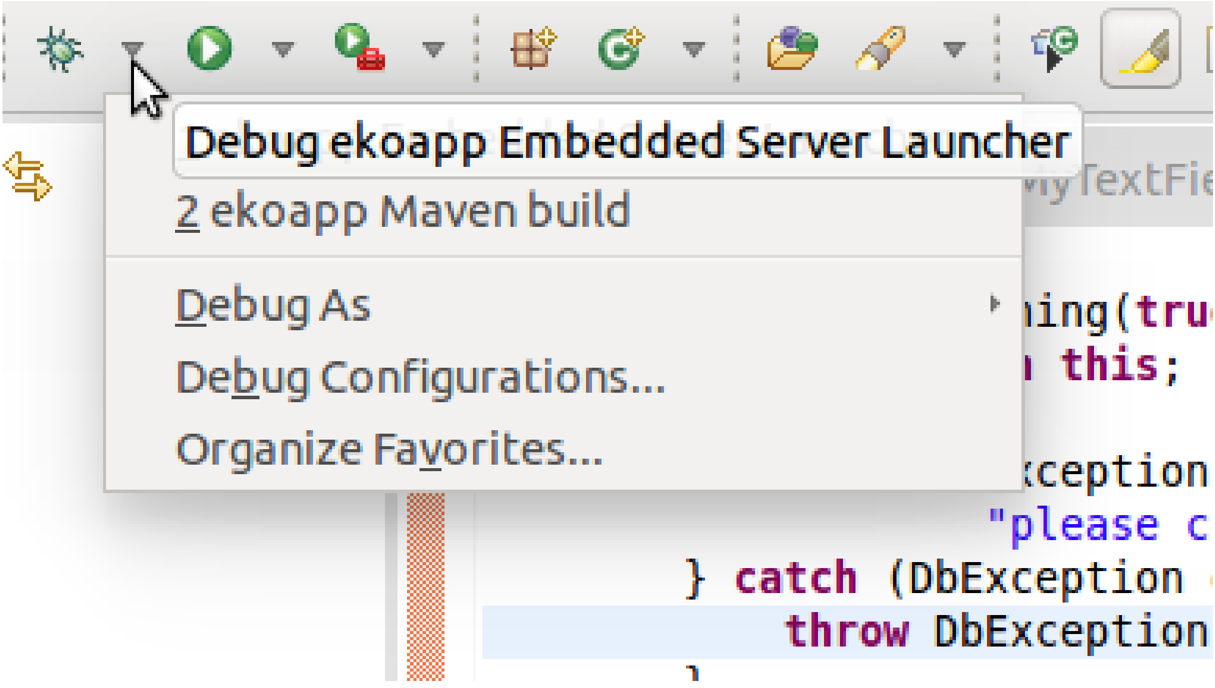


Figure 3.15 Running the launcher in debug

5. Enable debugging: go to the Debug perspective and click the Debug button in the LSPS Breakpoints.

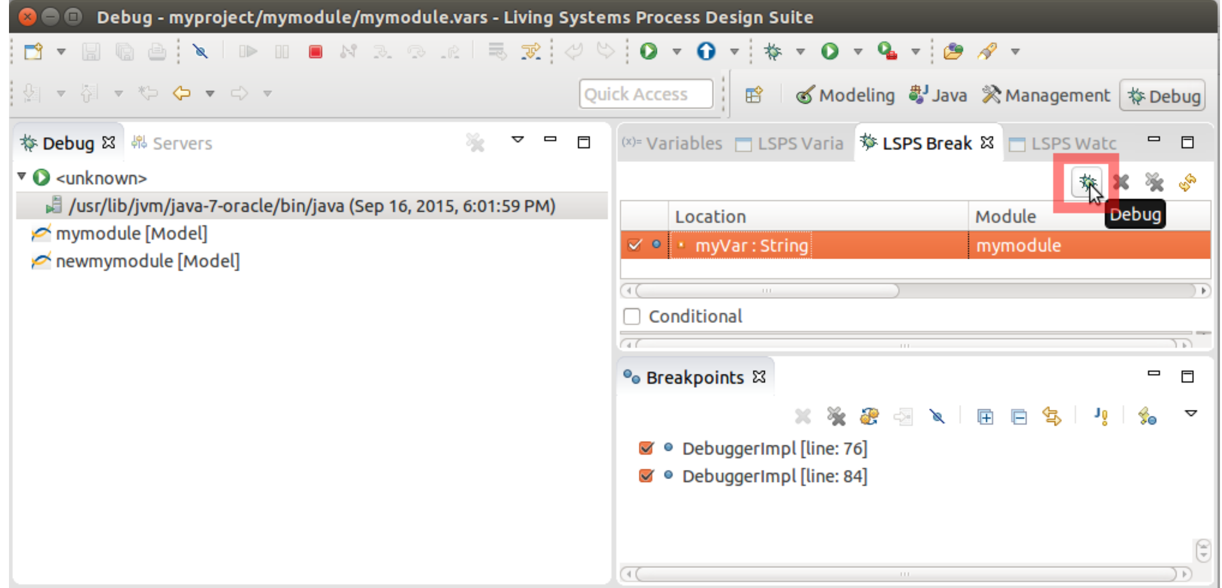


Figure 3.16 The LSPS debug is enabled: the Java breakpoints are in place.

6. Now you can debug the model.

3.4.2.3 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. Create an empty Java project in your workspace.
4. Go to Run > Debug Configurations > Remote Java Application.
5. Click the New button and on the right define the details of the remote server.

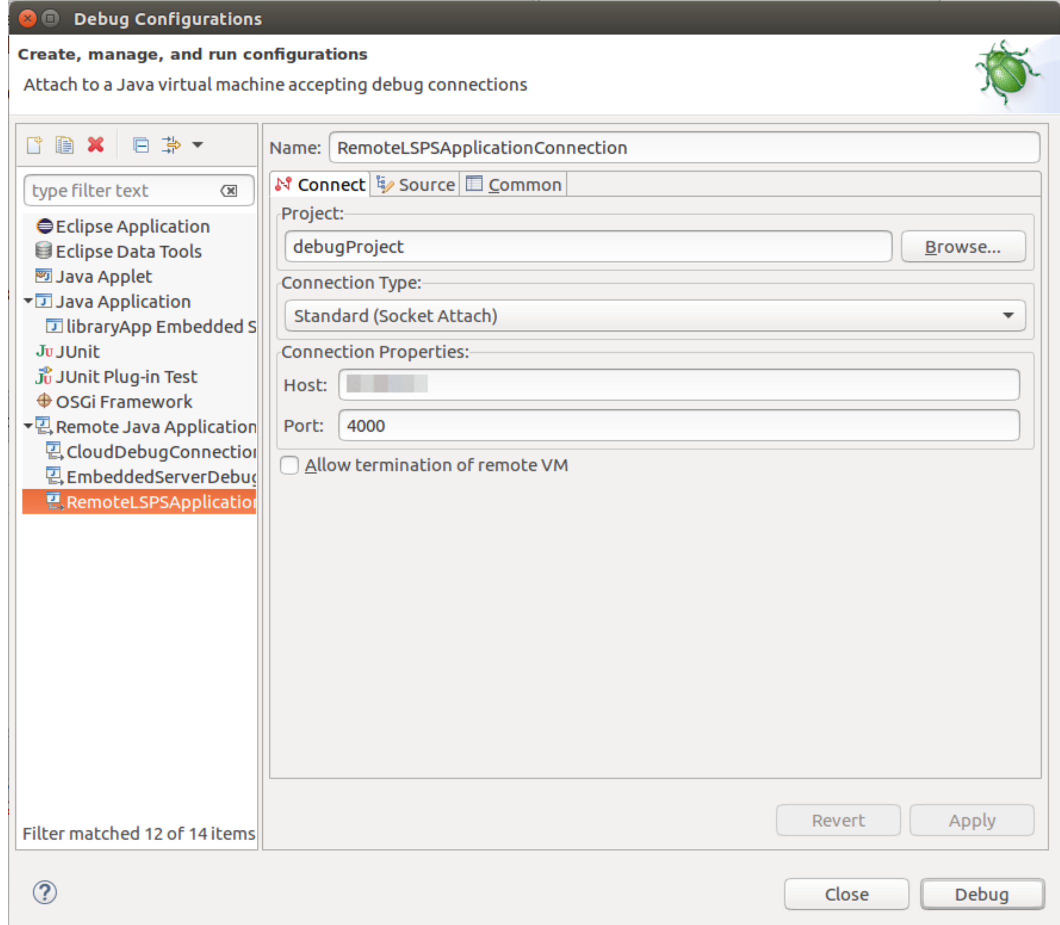


Figure 3.17 Setting connection to a remote server

6. Run the debug configuration to attach the LSPS Debugger to the server.
7. 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) Designer uploads all LSPS breakpoints to the server.

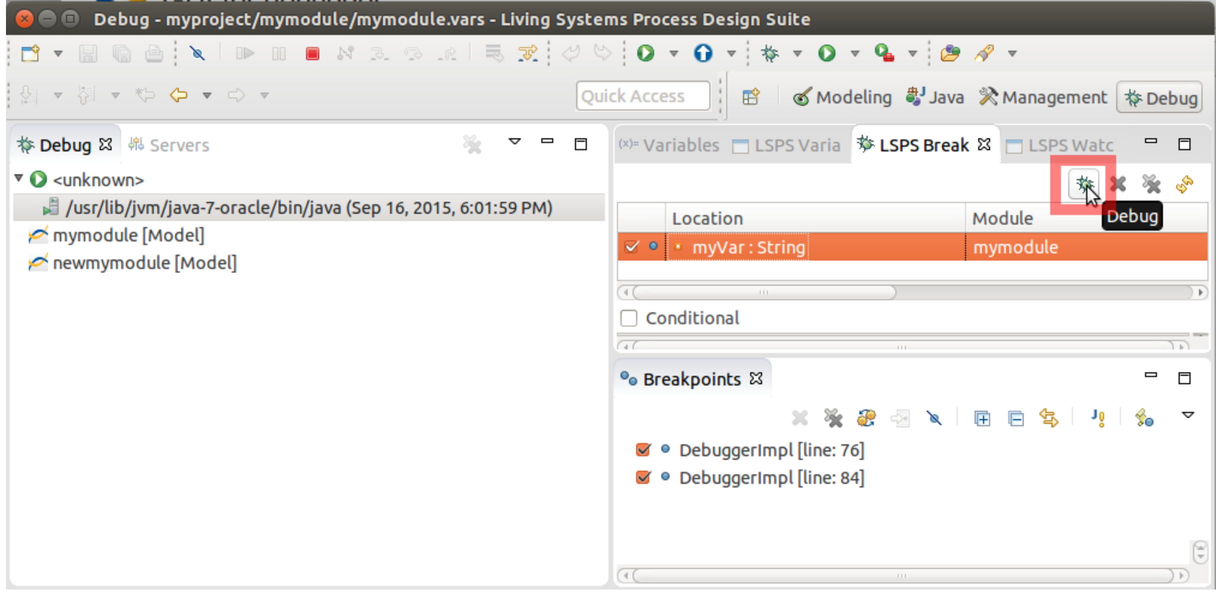


Figure 3.18 The LSPS debug is enabled: the Java breakpoints are in place.

8. Now you can debug the model: add breakpoints to the model and run the model.

3.4.3 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.

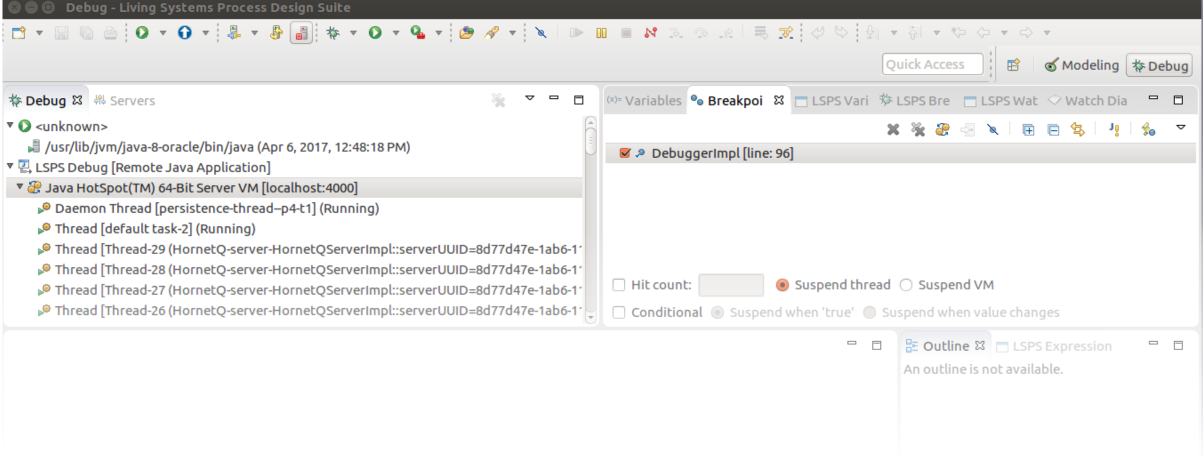
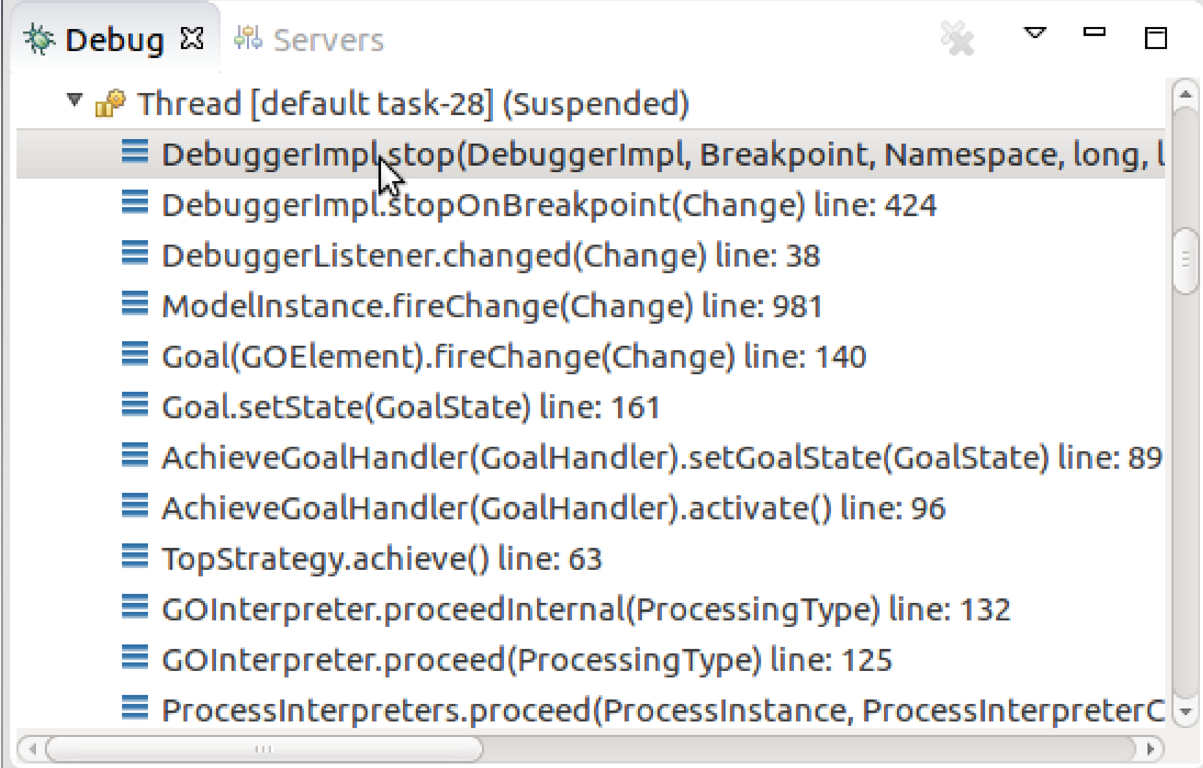


Figure 3.19 Local Embedded server in debug mode and debug enabled

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.



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

3.4.3.1 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.

Important: Breakpoints on method or function signatures are ignored (add a breakpoint to the first expression in their bodies).

A 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.

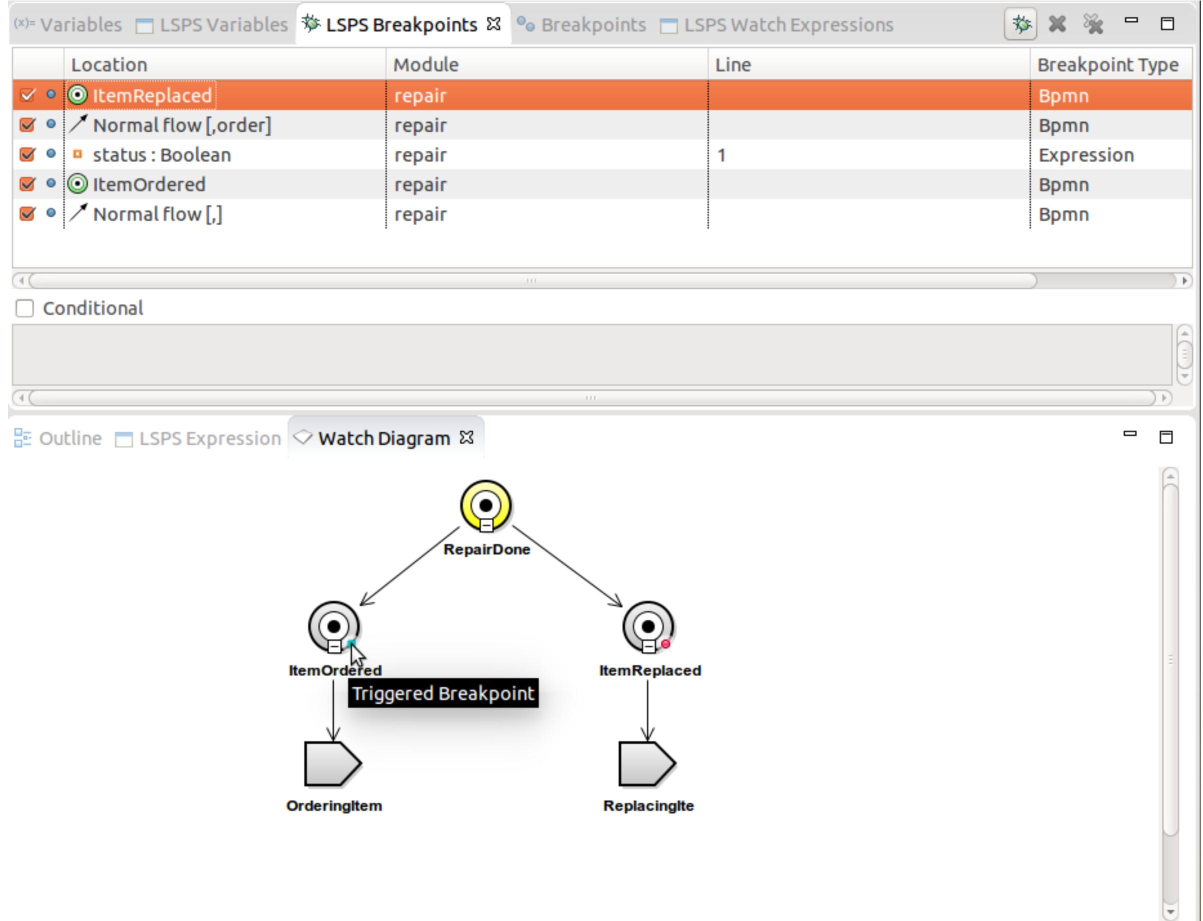


Figure 3.20 Active breakpoint in a Goal hierarchy

3.4.3.2 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.

3.4.3.3 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.

3.4.3.4 Enabling and Disabling Breakpoints

You can disable and enable LSPS breakpoints as well as define breakpoint conditions in the LSPS Breakpoints view.

3.5 Profiling Tools

The profiling tools 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, method, 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 when LSPS Profiler is not a feasible option. Also, it might be more convenient as input for analysis in other systems.

3.5.1 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 Designer Embedded server, do the following:
     (a) Click the *Stop Embedded Server* button to stop Designer Embedded server.
     (b) Go to **Server Connections** -> **Server Connection Settings**
     (c) In the Preferences dialog under **Designer** -> **Management** -> **Server Connections** edit the *LSPS Embedded Server* connection.

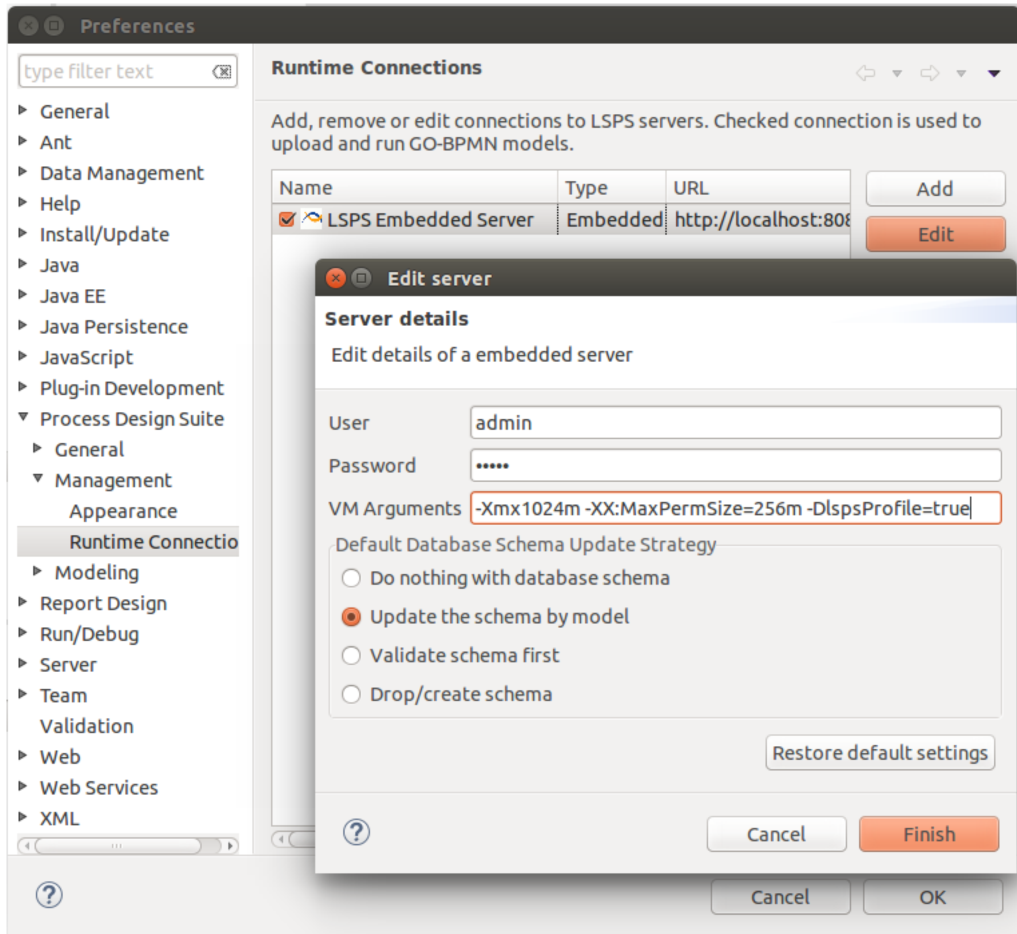


Figure 3.21 Profiler-enabling property on the Embedded Server connection setting

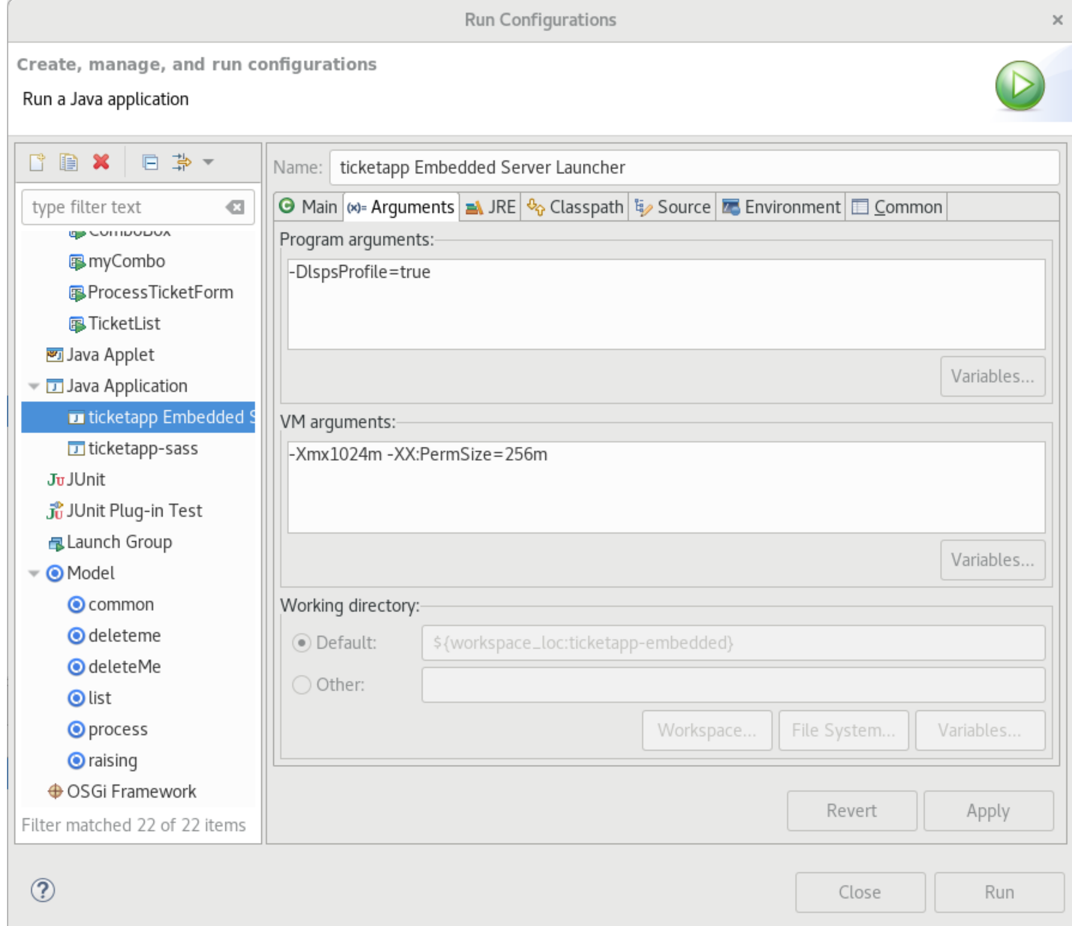
(d) Click the *Start Embedded Server* button to start Designer Embedded server.

- For the SDK Embedded server, do the following:

  (a) In the Console view of the server, stop the remote server.

  (b) Go to **Run** -> **Run Configurations**

  (c) In the *Run Configurations* dialog, find the launcher under *Java Application*.



- For remote connections, do the following:

  (a) Stop the remote server.

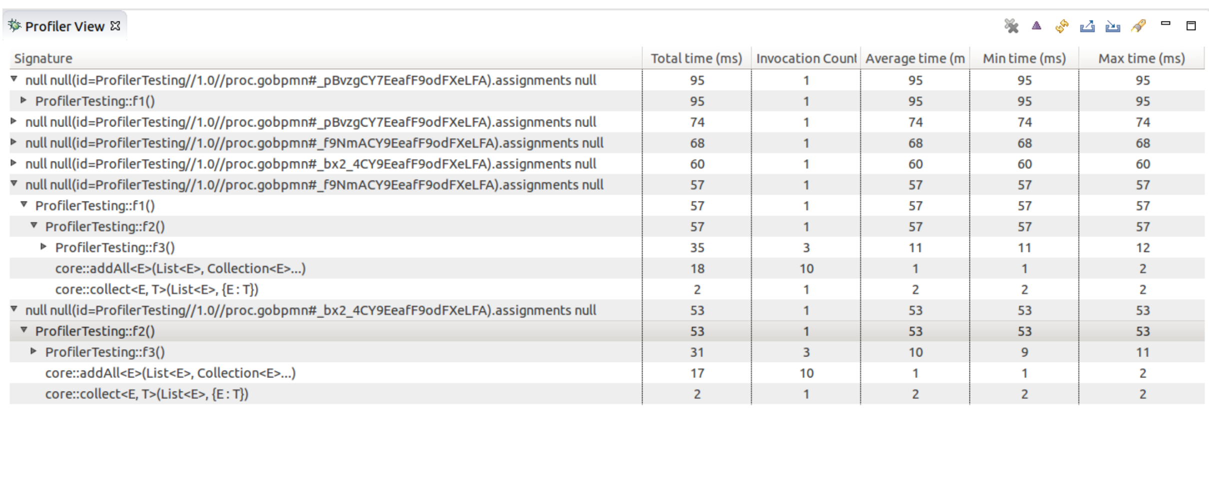
  (b) Run the server with the `-DlspsProfile=true` property.

2. In the Management perspective, open the *Profiler view*.

3. As applicable, upload and 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.



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 3.22 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.

3.5.2 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.TimerInterpr
   ```

   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 Designer Embedded Server is by default located in `<WORKSPACE>/.LSPSEmbedded/wildfly-<V↩ ERSION>/standalone/log/`

Note that if other users are running model instances on the target server, their model instances will be included in the results.

Chapter 4

Model Update

The LSPS *model update* mechanism allows you to update one or multiple 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 to get familiar with model comparing before using the Model Update Editor.

> **Important:** The model-update feature *does not support update of form definitions*; hence if you are updating semantics and data of a form that is used in a document or a to-do, make sure to `reset any running or saved to-dos` and documents. Saved documents and to-dos are NOT part of the update.

Model update is performed in the following 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.

> **Important:** On model update, changed closures are checked for compatibility. If an incompatible change is detected in a new version of a closure is detected, the system displays a dialog with details. You can then continue the model update; however, the updating might fail. In such a case, consider using the Restart strategy on the given element.

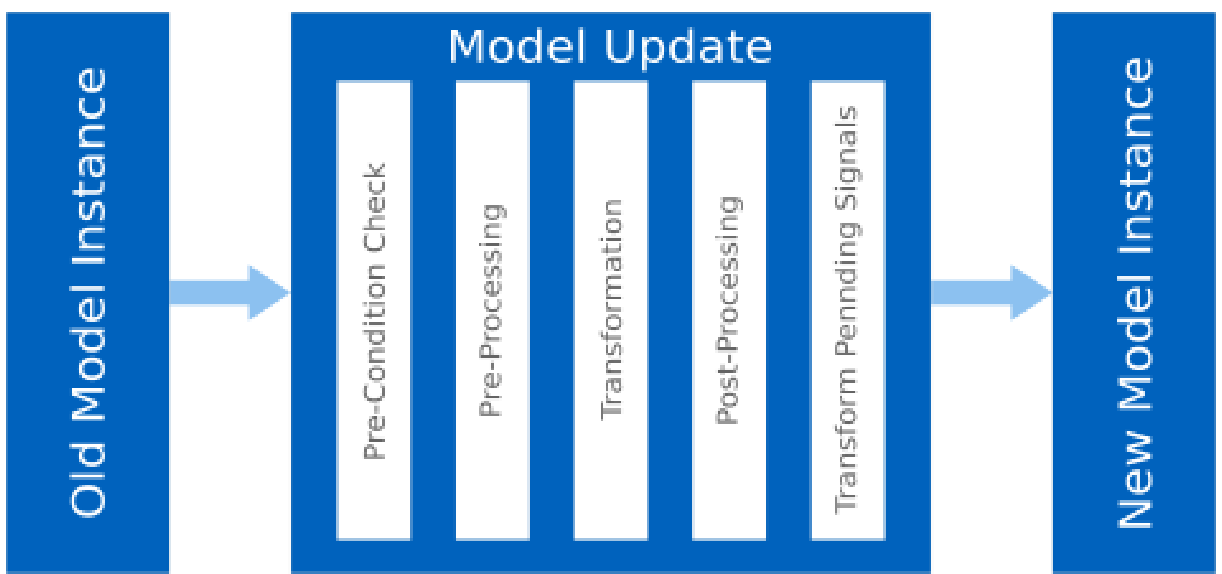


Figure 4.1 Schema of Model update process

The 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.

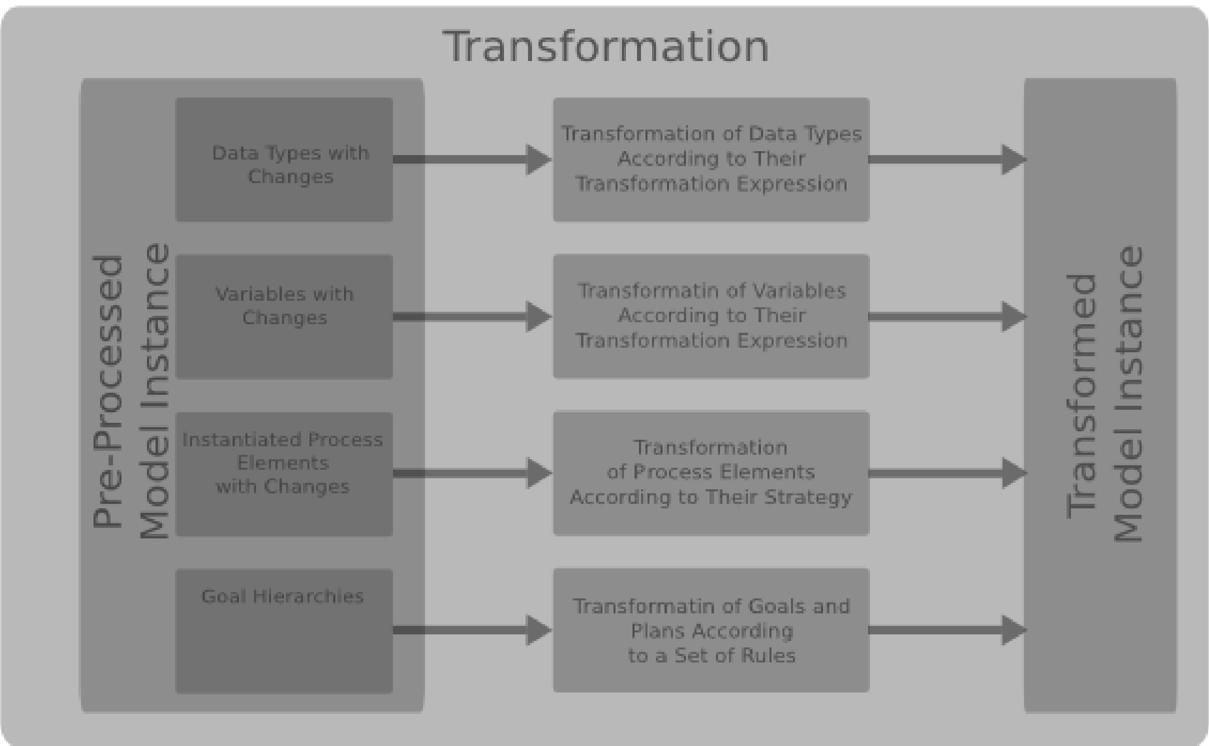


Figure 4.2 Schema of transformation

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.

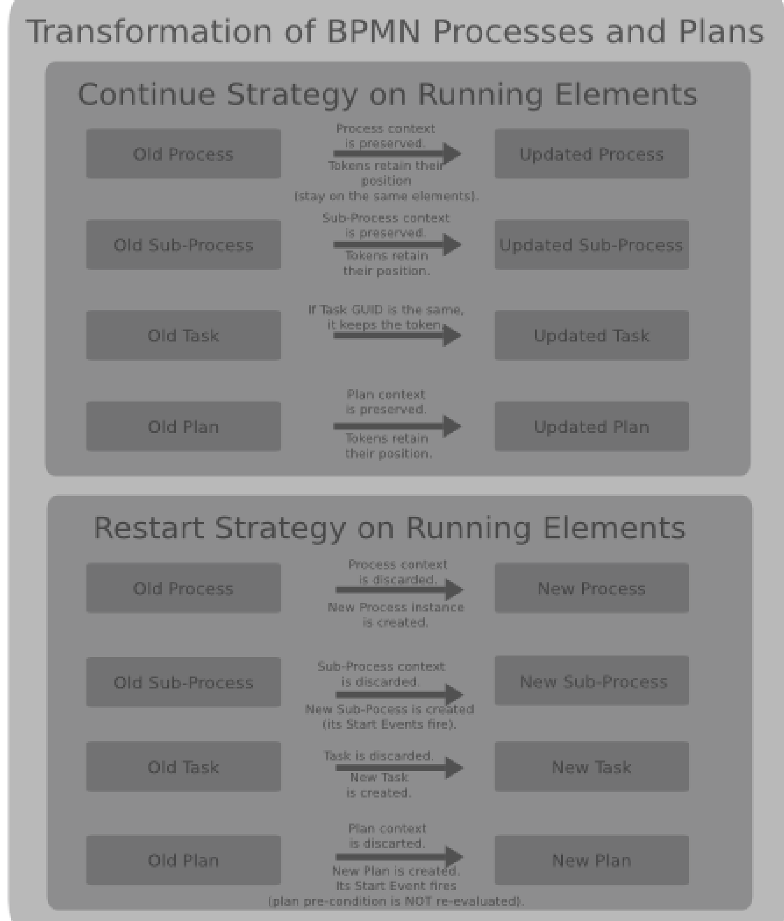


Figure 4.3 Transformation strategy semantics

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 Model Update Configuration

Model update configuration defines how the update of the given model instances is performed.

When preparing and testing the model update configuration, you will usually proceed as follows:

1. Import the model you want to update into your workspace.
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
5. Update the model instances.

Important: If your changes include incompatible changes to your data type model, suspend any running model instances and *migrate your business database* before update. Therefore, make sure to prepare any required migration scripts. Consider using generating the schema update scripts to acquire the first draft of the migration scripts.

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**.

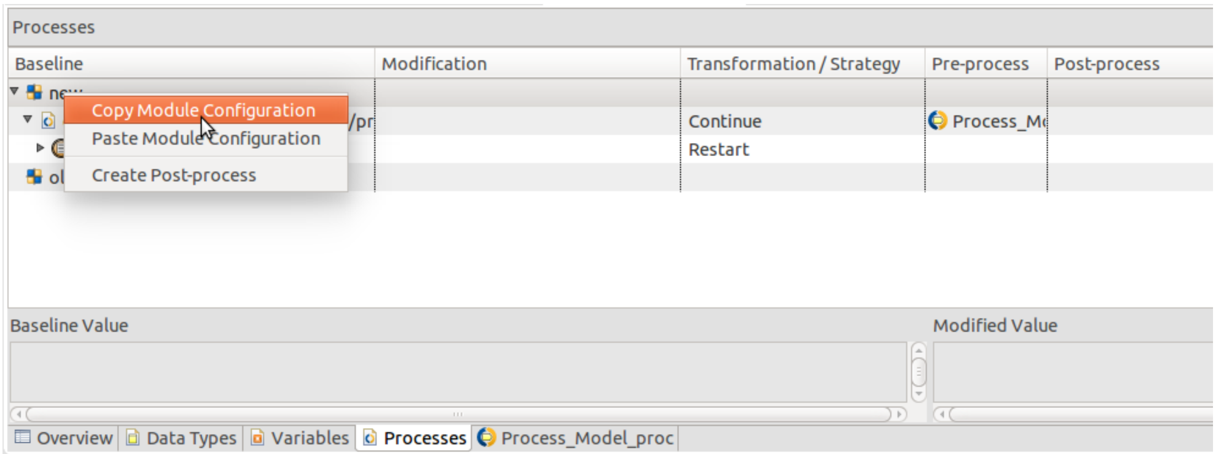


Figure 4.4 Copying model element changes on a module

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

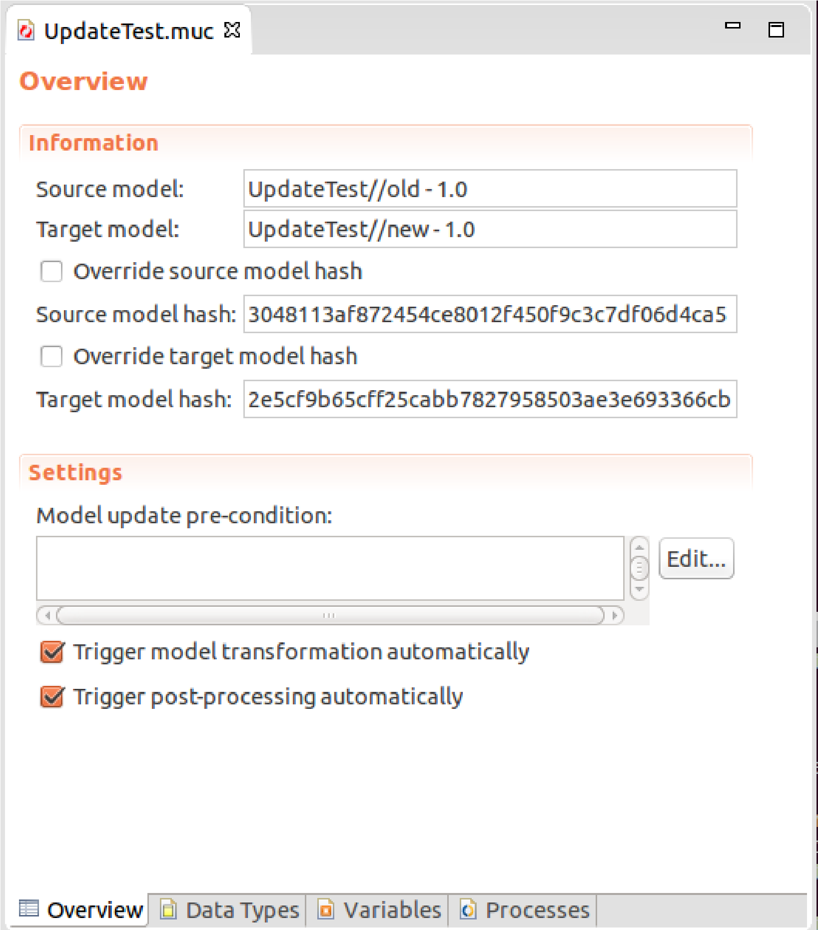


Figure 4.5 Model Update Editor

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 `management tools`.
    - 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.

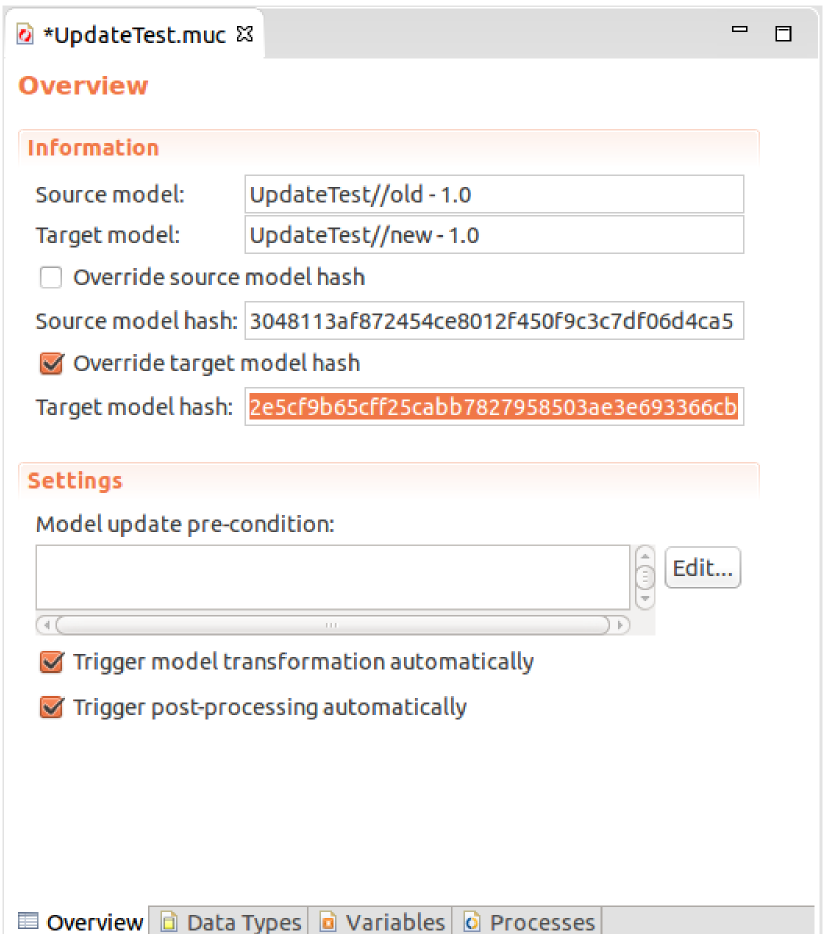


Figure 4.6 Changing hash code of the target Model

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.

> **Important:** On user tasks with forms consider setting the Restart strategy to prevent problems with saved to-dos and incompatible changes on data.

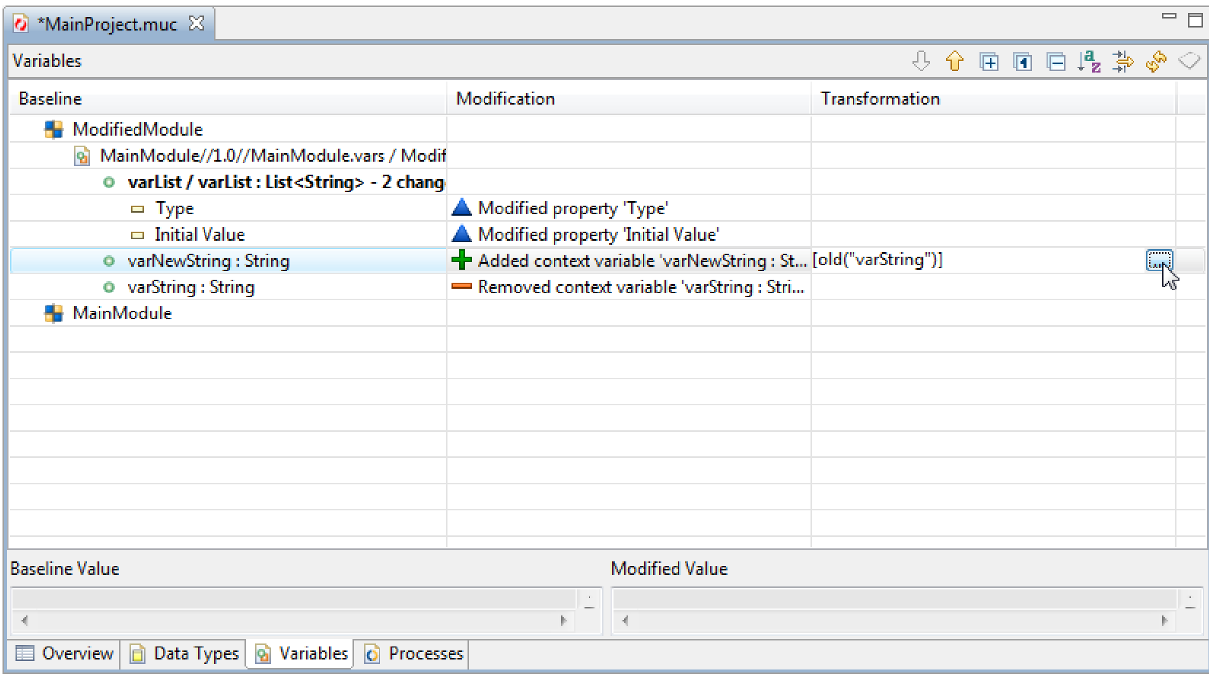


Figure 4.7 Defining variable transformation expression

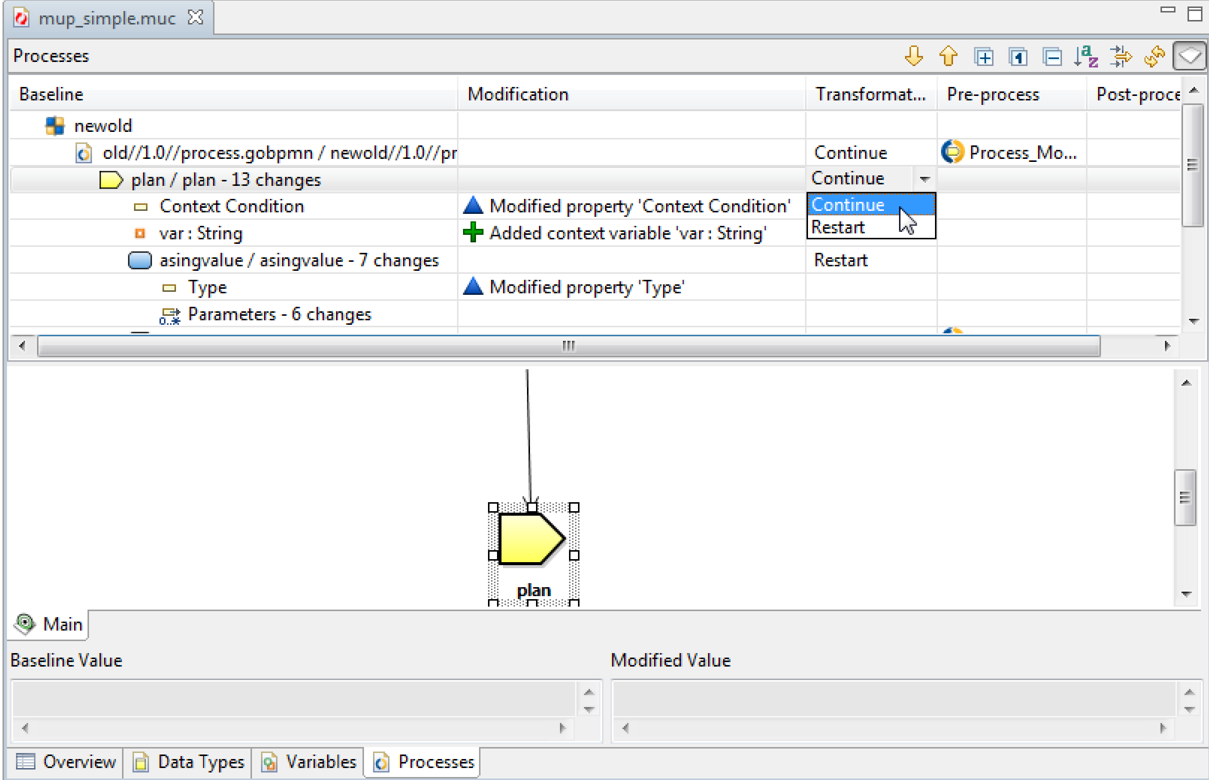


Figure 4.8 Selecting transformation strategy

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.

> **Important:** If your changes include incompatible changes to your data type model, you need to suspend any running model instances and migrate your business database before update.

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: 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.

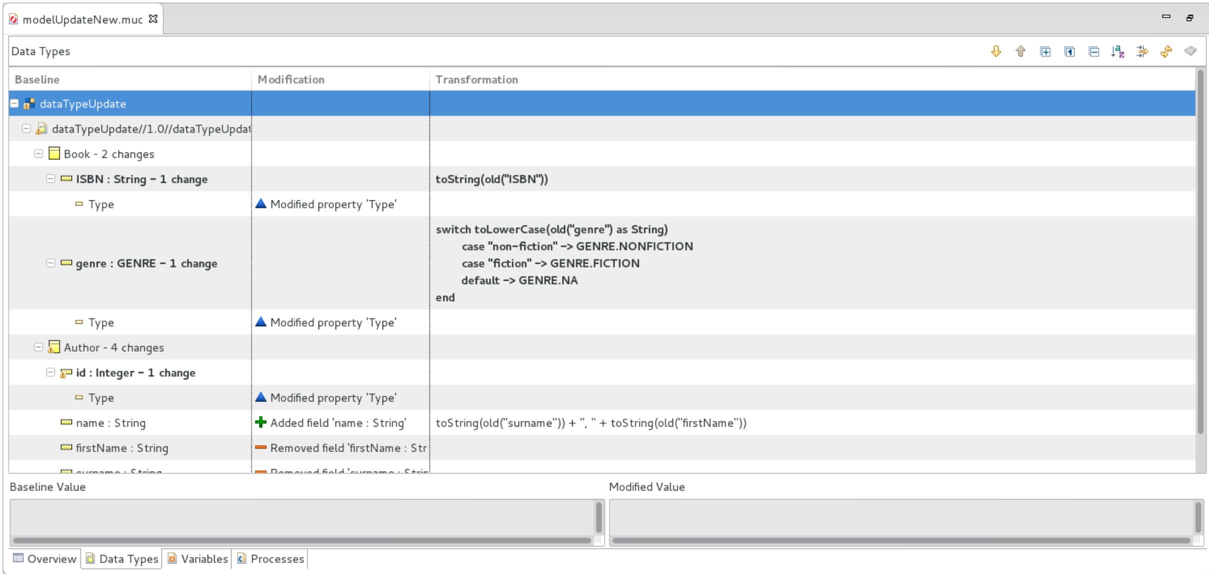


Figure 4.9 Defining data type transformation

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-process** 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.

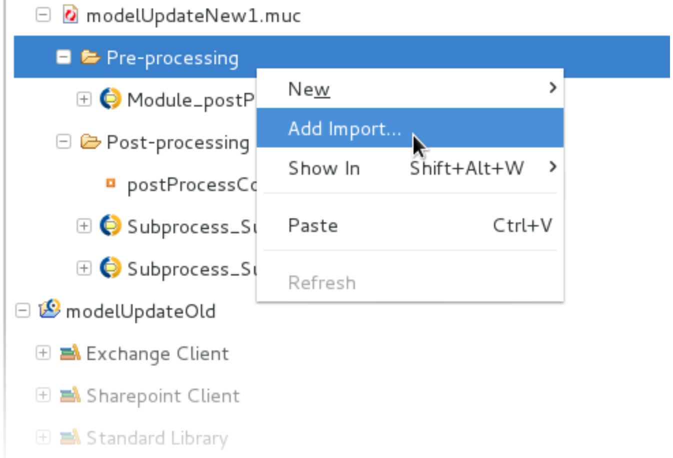


Figure 4.10 Importing Module

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.

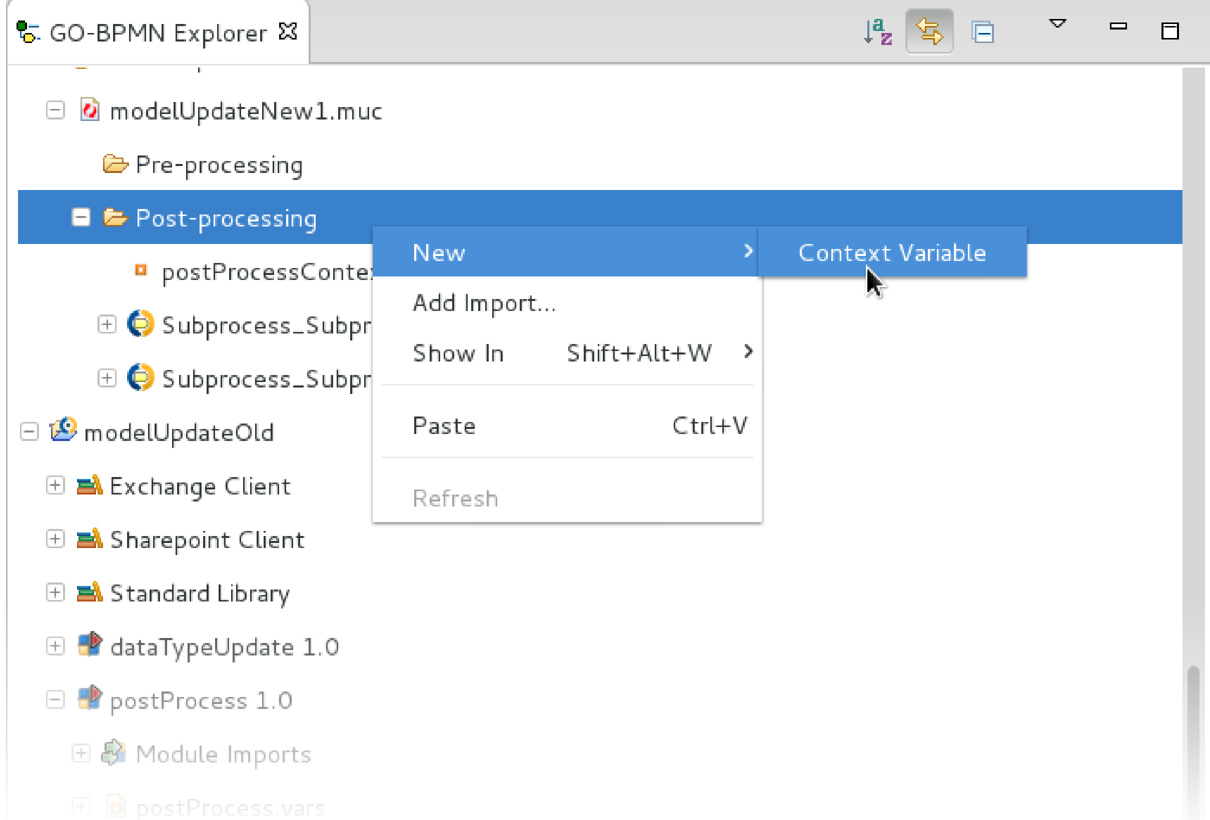


Figure 4.11 Creating variable in a model update process

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.