WHITESTEIN Technologies

Living Systems® Process Suite

GO-BPMN Modeling Language

Living Systems Process Suite Documentation

3.1
Tue Jan 12 2021

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

Contents

Chapter 1

GO-BPMN Modeling Language

The *Goal-Oriented Business Process Modeling Notation* is a visual modeling language used to design business models. It extends the BPMN specified by OMG and enables you to apply either a goal-driven approach or the classical BPMN approach in model design.

GO-BPMN Modeling Language as a conservative GO-BPMN extension and provides elements and mechanism for goal-oriented business modeling. The goal-extension of the language enables you to create models that separate the goal (WHAT you wish to achieve) from the way it is achieved (HOW to achieve it). The goals are defined by Goal elements and the ways they can be achieve by Plans.

Additionally, GO-BPMN defines elements for organizational models and data structures, which are not covered by the BPMN specification.

Chapter 2

Encapsulation

Encapsulation is a mechanisms for hiding content so as to present them as a single relatively self-contained container for other elements: In a model, all data is encapsulated in a Module–the highest level encapsulation construct. A module contains a plethora of elements, including Processes, which encapsulate BPMN or GO-BPMN flows; these can include Sub-Processes, which encapsulate BPMN flows, etc.

2.1 Context

A *context* is a set of runtime data based on a namespace. It is created when the element that represents a namespace is instantiated and holds values of variables, execution statuses of elements, etc.

Every namespace has its own unique context in runtime or possibly multiple contexts, that is, a model instance has its context, a process instance has its own context, a plan has its own context, and every instance of a multi-instance sub-process takes place in its own context: Hierarchy of the contexts reflects the hierarchy of namespaces.

Lower contexts have access to higher contexts.

Since a context holds all runtime data, it secures the persistence of execution data and status: In case of execution interruption, the stored context can be used to restore the execution status.

2.1.1 Visibility

Generally, a context can access and see elements of its parent context, but not vice versa; for example, a process cannot see the context of its sub-process.

Visibility of an element determines the rules of its accessibility from within or out of their context. It may be `public` or `private`. A private element can be referred to only by the elements of its own context. In addition, Record Fields can be also `protected`: they are accessible only from within its data type hierarchy.

2.1.2 Namespaces

Some elements that represent encapsulation constructs, such as, modules, process, etc. are also namespaces. Namespace construct must have a name. Within a namespace container, the elements with a semantic value must have unique names.

Though element names are often optional, in a single namespace, modeling elements must have unique names, if these are specified.

The namespaces constitute a hierarchy: top is the Module namespace, then the Process namespace, for goal processes Plan namespace and then Sub-Process namespaces. On runtime each namespace typically results in one context or multiple contexts. For example, one module context, multiple contexts for looping Sub-Process.

In GO-BPMN, the following elements represent namespaces:

- **Modules**

  An element in another module is referenced using the name path in the form `::`. This happens if the target element is in an imported module.

- **Processes**

  A Process instance cannot access elements in another process instance.

- **Plans**

  A Plan instance cannot access elements in another Plan instance.

- **Sub-Processes**

  A Sub-Process instance cannot access elements in another Sub-Process instance.

2.1.3 Metadata

Metadata are data pairs comprising a data key (name) and a data value, which provide additional data about a modeling element.

Metadata can be defined for any modeling element (element with execution semantics) with the exception of Modules. As providing background information, metadata are local to their owners and may contain *exclusively* constants.

2.1.4 Variables

Encapsulation elements with context can define variables, that will hold a value of a type on runtime, for these contexts; on runtime, such a variable is instantiated as part of the context: On context initialization, variables are assigned their initial value. Variables can use other variables of their own context or their parent contexts for their initialization.

Depending on the immediate parent context of a variable, we distinguish global variables defined in modules, and variables defined in processes, plans, and sub-processes.

A variable has the following properties:

- **Type**: data type of the value the variable can hold on runtime
- **Initial value**: value assigned to the variable when its context is created
- **Visibility**: access rules to the variable

If true, the Module can become a Model instance.

- **Monitoring**: property defining if the variable is used for monitoring purposes

Every variable must define its name and data type of the value it can hold.

> **Note:** In expressions, you can define `local variables`. The scope of these variables is the expression or the expression block.

2.1.5 Module

A *Module* serves as a container for model resources, such as, BPMN processes, organization hierarchies, etc. It can include other Modules, called Module imports. As such it resembles a jar file with dependencies.

On runtime, a module can act as a model and be executed.

It represents a namespace; hence to reference its elements from another module, you need to include the module name in the element call, for example, `myModule::myVariable`.

A module defines the following properties:

- **Version**: version of the Module
- **Executable**: Boolean Attributes

  If true, the Module can become a Model instance. Note that if you import an executable module to another executable module, and you instantiate the importing module (the parent), the imported executable module will be instantiated as part of the model instance.
- **Module Imports**: imported Modules
- **Terminate condition**: Boolean condition defining if a module can be instantiated

  The condition is checked for the first time after the first transaction (module instance is created, process instances are created, process' start events fire). After that it is checked constantly during the entire life of a module instance. When evaluated to true, the Module instance is terminated.

  Note: Terminate conditions of imported Modules are not evaluated.

2.1.5.1 Module Import

A *Module import* allows you to reuse existing modules: it is include of a Module similar to dependencies of jar files: it allows a parent Module to use the resource of its Module imports, which are read-only includes of their Modules.

A Module can import one or several Modules, however, you can import a particular Module only once. Modules cannot be imported recursively (if moduleA imports moduleB, then moduleB cannot import moduleA).

> **Note:** To refer to the elements in Module imports from the importing module, you need to explicitly define the namespace of the referred Module; for example, `myModule::myVariable`.

Chapter 3

Model

A *model* is an inclusion of an executable Module and its Module Imports. It is not explicitly represented by any component: any executable module with all its resources and module imports represents a model.

While you can upload any Module to the server, you create Models only over Modules that are executable. An executable module serves as a static basis for creation of an arbitrary number of model instances: An model instance uses the model with its static data as a blueprint, while the instance holds its runtime data, such as, what element is currently being executed, what are statuses of the elements, values of variables, etc.

If a module imports other modules, whether the imported module is executable or not has no impact on execution.

During its lifetime a Model instance goes through a set of execution statuses:

- **Created**: Model context is created and contexts of individual module instances are created.
- **Running**: Context data are initialized, initial values are assigned, and all BPMN-based and Goal-based Processes in all executable modules are instantiated.

  Note that this happens in a bottom-up manner: First the modules that are "lowest" in the hierarchy are initialized: If module A imports module B and module B imports module C, then C is initialized first, then B and only then A: this allows you to use data from C and B in A, but not vice versa.

  A Running Model instance can be suspended: all its Process instances and their elements are suspended and no execution is taking place.

  > **Note:** If a Model instance attempts to perform an invalid action immediately when it becomes Running and an error occurs, the initialization is rolled back and the Model instance goes back to the Created status.

- **Finished**: Model instance becomes Finished, when all its Process instances are Finished.

  You can finish a model manually if required.

3.1 Suspend

A *Running* model instance can be suspended: on suspend, its execution is paused immediately so that no changes on runtime data can take place. Execution of all running elements is interrupted and all elements become *Suspended*. A suspended model instance is read-only.

A Suspended Model instance may be resumed. When resumed the execution of the Model instance continues from the point when it was suspended. It becomes Running and all asynchronous inputs received by the Model instance while suspended (Signals, elapsing of time periods of Timer Events) are received and processed.

Note: If a Timer Event, either a Timer Start Event or a Timer Intermediate Event with a duration is suspended while Running, the duration is checked with regards to the time, when the Model instance was suspended. For example, a Timer Event with a duration of 60 minutes was triggered at 1 p.m.: if the Model instance is resumed at 1.30 p.m., the Timer Event continues running until 2 p.m.; if the Model instance is resumed at 3.00 p.m., the Timer Event is finished and the outgoing Flow is taken immediately. For cyclic events, only the last occurrence of the event is processed: For example, if a BPMN-based Process is to be instantiated every day at 12 p.m. and the model instance is resumed after three days at 1 p.m., only one process instance is triggered.

3.2 Finish

When a model instance receives a request to finish, the following happens:

- Active activities (Tasks and Sub-Processes) fail and become *terminated* immediately.
- Processes become *finished*:
  - In Goal Processes, all Achieve Goals become *deactivated* immediately while Maintain Goals finish their current cycle and then become *deactivated*.
  - In BPMN Processes, all active Activities its BPMN-based Process instances are terminated (fail).
- All *alive* to-dos become *interrupted* so they cannot be submitted.
- As a result, the model becomes *finished* since all process instances are *finished*.

Chapter 4

Process Model

A Process Model is the set of *processes* within a model. Process is a container element for you Goal or BPMN process workflow. They are always part of a Module, with one Module containing an arbitrary number of Processes. Just like a Module, a Process represents a namespace which, on runtime, results in a local process context, a Process instance. A Process can contain other elements that represent further nested namespaces and result in further local contexts in the Process instance context, such as, Sub-Process contexts.

There are two types of Processes:

- Goal-based which makes use of the Goal extension of GO-BPMN
- BPMN-based defined as pure BPMN Processes.

When a process instance is to be created, the server checks if the Process is executable: if the Process is not executable, no instance is created. If it is executable, instantiation takes place:

1. Process instance based on the Process is created.
2. Process namespace instance is created.
3. The Process instance becomes **Running**.

   > **Note:** What happens when the process instance becomes running depends on whether it is a BPMN Processes and Goal Processes.

4. On execution finish, the Process instance becomes **Finished**.

When a Model is instantiated it attempts to create instances of all its processes including any processes in its Module imports: for each Process it checks first whether the process is executable as stated above and then whether it can be instantiated automatically.

Processes can be reflected as Records: this allows you to create instances of Processes by instantiating them as Record instances; for example, you can send the name of the Process as a parameter to the Execute Task and instantiate it in the Execute Task from another Process; for further information, refer to the `Standard Library documentation`.

Process Attributes

- **Executable** is a Boolean attribute set by default to true so that the Process can be instantiated

  If false, the process is cannot be instantiated: as a result, if a Model instance is triggering processes on start automatically or with another mechanism later (such as, signals), the process is not instantiated.
- **Visibility** defines the access rules to the Process.
- **Create activity reflection type** creates a Record that reflects the Process.

4.1 Goal Processes

A Goal-based Process is a Process with a Goal model.

On execution, after the parent model instance becomes Running, one process instance based on the process is created.

When instantiated, the following takes place:

1. Context data are created and initialized (initial values are assigned).
2. All top Goals are triggered (become Ready and, if their conditions are fulfilled, immediately Running) and the Process instance becomes *Running*.
3. When no Achieve Goal is Active (Ready or Running) and no Plan is running, the Goal-based Process instance becomes *Finished*.

4.2 BPMN Processes

A BPMN-based process encapsulates a BPMN-based process model.

It is instantiated when the parent model instance is instantiated, possibly multiple times by different start events.

> **Note:** A BPMN-based Process may be used as a Reusable Process and be instantiated as a sub-process instances.

When a BPMN-based process is being instantiated, the following happens:

1. Context data are created and initialized (initial values are assigned).
2. The process instance becomes *Running* and its Start Event, and Activities without incoming Flow are triggered and produce tokens. The token is passed to their outgoing flow: the token marks the currently executed step.
3. When no process element is active (there is no token in the workflow), the Process instance becomes Finished.

A BPMN-based Process defines the following specific properties:

- **Instantiate automatically**: Boolean attribute defining if the process creates its instance when requested (if false, the Process can be triggered only as a Reusable Sub-Process)
- **Parameters**: an arbitrary number of parameters with a parameter name and type (when called as a Reusable Sub-Process, the parameter value is added)

4.3 Reusable Processes

A Reusable Process is an application of a BPMN-based Process when the Process is called by a Reusable Sub-↩ Process activity (Reusable Sub-Processes). One Reusable Process can be called by an arbitrary number of Sub-Process activities, thus providing a convenient mechanism for reuse of workflows.

Any BPMN-based Process can be used as a Reusable Process provided it meets the following criteria:

- It starts with a None Start Event, that is, no other Start Events are triggered.
- It is public.

If a BPMN-based Process is used only as a Reusable Sub-Process it can define an arbitrary number parameters as the process properties. Note that the properties can be defined as required.

> **Note:** In Living Systems® Process Design Suite, a Reusable Process must be marked as executable. Also, consider disabling the Instantiate Automatically on the Process so that it is not instantiated when the parent Module is instantiated.

Reusable Processes can have the *Parameter name* property.

Process Modeling

4.4 Process Modeling

A process contains workflows consisting of modeling elements with execution semantics, which define how it is executed; this includes processes themselves, goals and plans, process body elements with semantics, such as, gateways, intermediate events, start events, activities, etc.

Every modeling element has at least the following properties:

- **Name**: optional identifier of an element in its namespace
- **Description**: free-text description
- **Assignments and Monitoring**: set of expressions executed at a certain point of the element's life

4.4.1 Assignments

Assignments serve to define expressions that are executed when a particular element enters a particular life cycle status. The mechanism is available on modeling elements with execution semantics within a Process and on Processes themselves. Typically, an assignment defines an expression that assigns a value to a slot, such as a variable.

You can define an assignment for the following:

- Flows, Start Events, End Events, Exclusive Gateways:

  Assignment is performed when events are executed (token passes through the element).
- Intermediate events, Activities, and Parallel Gateway:

  The elements can define the following assignments:
  - **Start assignment** is performed when the token enters the element.
  - **End assignment** is performed always when *a token leaves* the element or *is removed* from the element, for example, due to failure, termination, or restart.
  - **Accomplish assignment** is performed when a token when the flow element finishes "successfully" and the token leaves using the outgoing flow. The Accomplish assignment is performed always before the End assignment.
- Goals and Plans can define assignment for individual life-cycle statuses.

4.4.2 Monitoring an Element

To define monitoring related activities, modeling elements can define

- monitoring assignments: a special type of Assignment intended to implement the monitoring logic Monitoring assignments are available on modeling elements with execution semantics within a Process and on Processes themselves (it is identical to the Assignment mechanism).
- monitoring flags: available on variables and on Records If flagged as monitoring, the elements are displayed with a monitoring icon to indicate that the entity is involved in monitoring. The flag does not impose any semantics by itself.

4.4.3 Signal

A signal is a means of communication within a model instance or with another model instances.

It is produced by Throw Signal Events and processed by Signal Start Events and Catch Signal Events of the target model instances:

- All Signal Start Events in the target model instances
- All Catch Signal Events in the target model instances that are waiting for the signal, that is, they hold a token at that moment

Alternatively, you can create a Signal object explicitly.

Signal object cannot be or recursively contain a reference, closure, Goal, Plan, or Process instance.

4.4.4 Errors

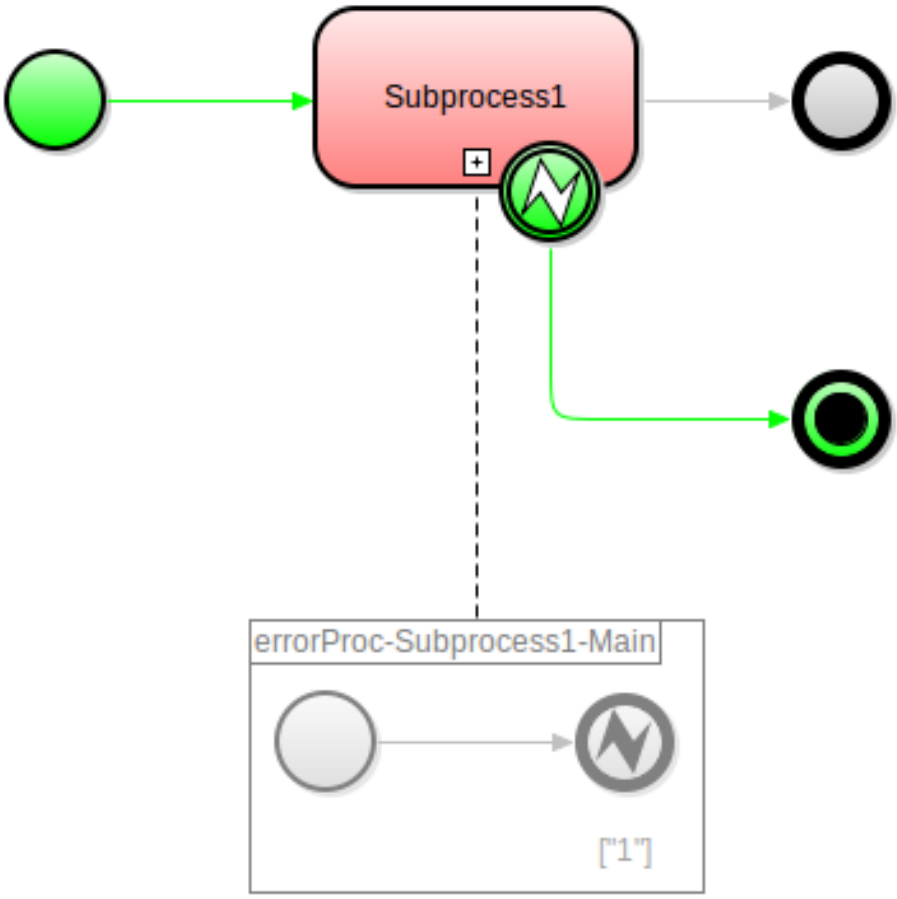
An *Error* represents a critical problem in execution of your process, which should cause the execution to stop immediately unless explicitly handled and corrected.

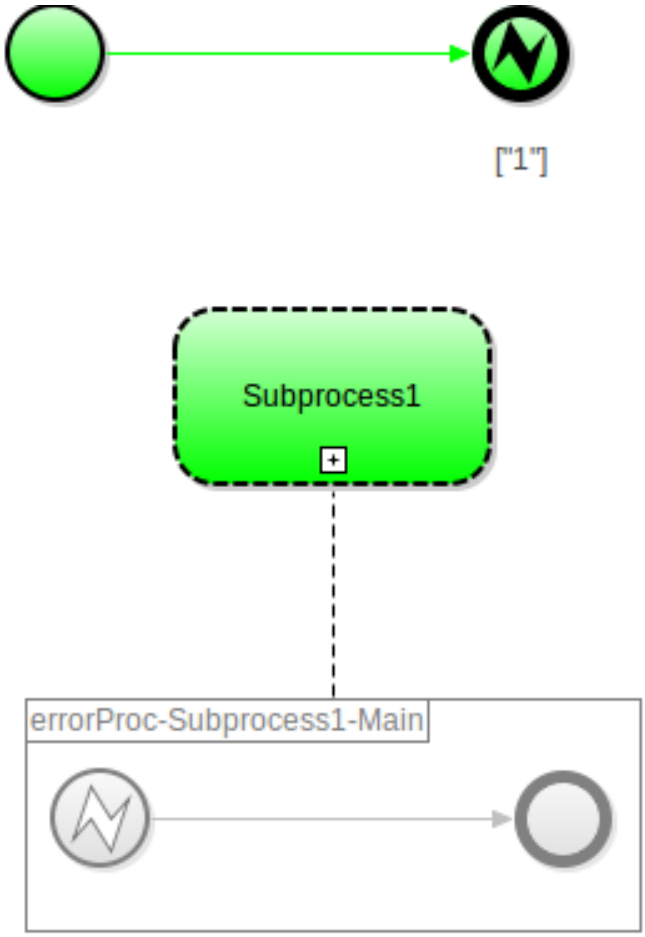
An error can be thrown by an Activity or an Error End Event.

> **Note:** Additionally, you can throw an error also with the `error()` function of the Standard Library or from your custom implementations as `com.whitestein.lsps.common.ErrorException`.

Once generated, the error is gradually propagated through its the immediate context and looking for an element, which could consume it. If the error is not consumed in the immediate context, it is gradually propagated throughout higher contexts. If not consumed within its process instance, an exception is thrown and the last transaction which caused the error is rolled back. Note that the last transaction, refers to the last `EJB transaction of the model instance`.

An error can be caught and consumed by a Error Intermediate Event or by Plans.


Subprocess1
errorProc-Subprocess1-Main
["1"]



4.4.5 Escalation

The Escalation mechanism resembles the signal mechanism; however, while signals can be consumed by multiple elements in the model instance, an escalation signal is consumed by a single element within its model instance.

Similarly to signals, to perform escalation, you need to create an escalation object with the Throw Escalation Event or Escalation End Event positioned at the appropriate location in your workflow. When the workflow enters a Throw Escalation Event or Escalation End Event event, the event produces an Escalation object, which is then propagated through its own context and then through parent contexts and "up" to higher contexts until caught by an Escalation Start Event or Catch Escalation Intermediate Event or until no higher context exists. Once caught by an Escalation Start Event or Catch Escalation Intermediate Event, the Escalation object is not propagated further. If there are multiple Escalation Catch elements in the same context, only one of the elements consumes the Escalation signal and only the catch event that consumes the escalation object produces a token.

The object must define its escalation code, a string that serves as its identifier: the code can be then used by filters of the Escalation Catch Events to filter the objects.

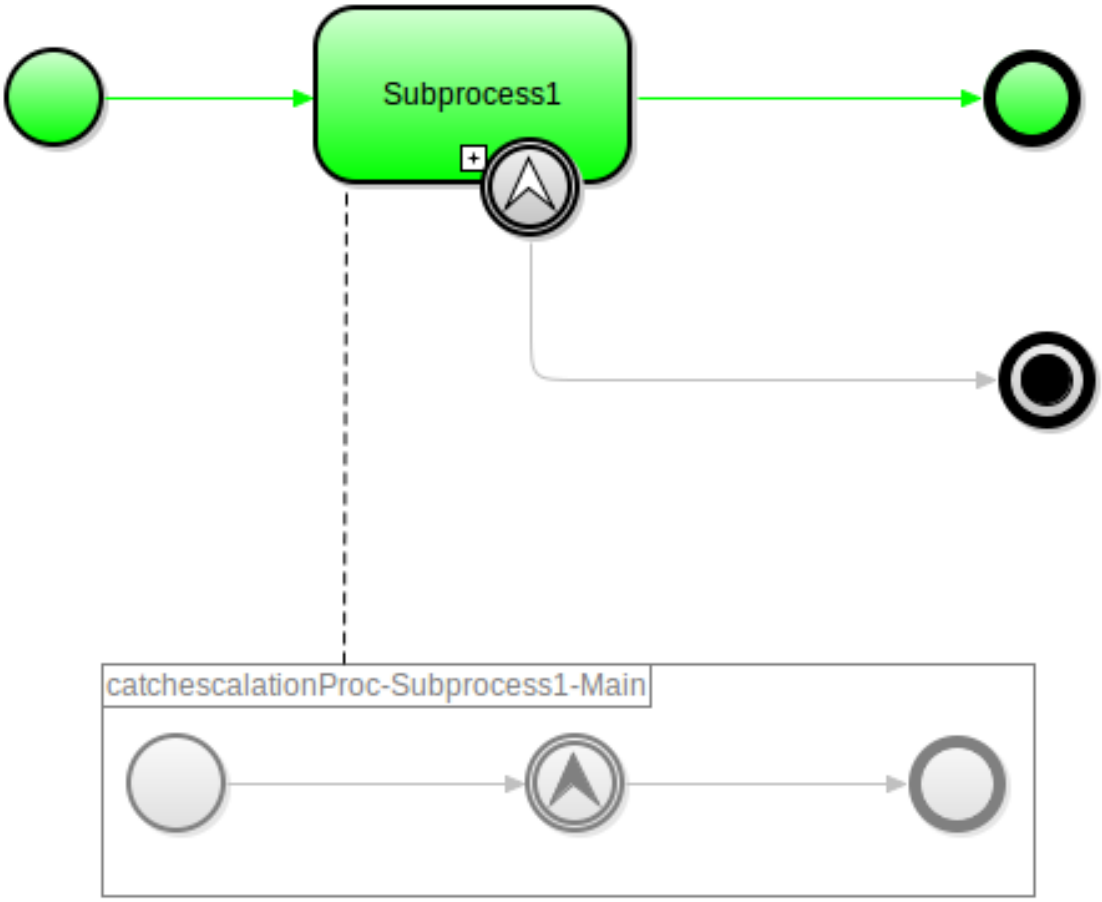


Figure 4.1 Demonstration of escalation with an Escalation produced in a subprocess and caught be a Catch Escalation Event

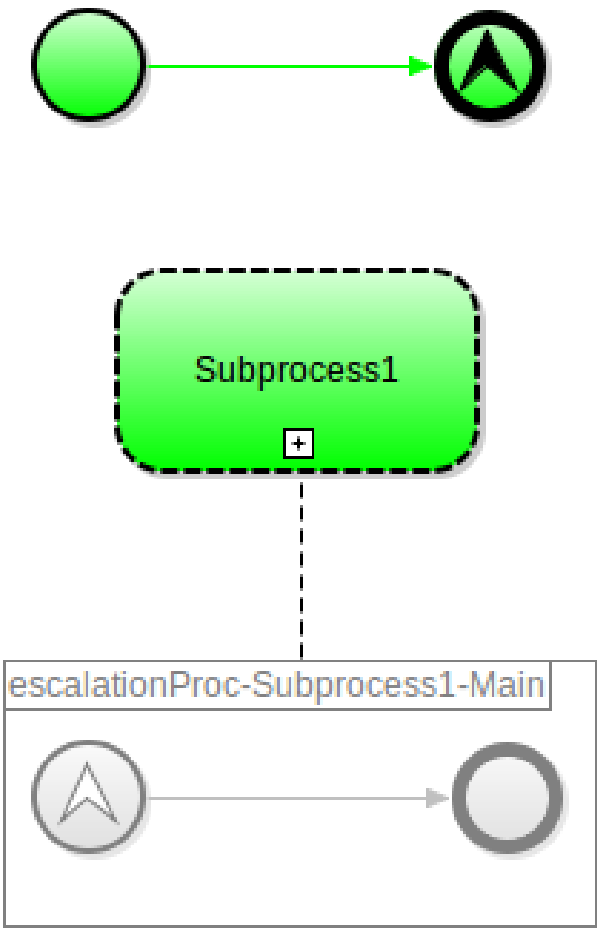


Figure 4.2 Process that terminates in an Escalation End Event which triggers the Escalation Start Event of its Inline Event Subprocess

Plan and BPMN Modeling Elements Goal Model

4.4.6 Plan and BPMN Modeling Elements

The Plan and BPMN-based processes use a similar set of modeling element in their workflows with some additional modeling element available for the BPMN-based process workflows so that you can model logic that is accommodated by Goal hierarchies:

- In a Plan, the workflow is triggered by the Plan.
- In a BPMN-Process, the workflow is triggered by the Process.

4.4.6.1 Plan Model

A *Plan Model* is a sum of all elements with execution semantics encapsulated in a Plan.

Every Plan Model:

- must contain one None Start Event, which is triggered, when the parent Plan becomes Running;

- must contain at least one End Event or an Activity with no outgoing flow;

Apart form that, it can contain an arbitrary number of activities, events, flows, and gateways connected with Sequence Flows (Connectors). The elements in the flows must meet their modeling rules and must create an uninterrupted workflow.

A Plan Model is triggered when the parent Plan becomes *Running* and its None Start Event produces a token and the respective namespace context is initiated.

4.4.6.2 BPMN Model

A *BPMN Process Model* is a sum of all elements with execution semantics encapsulated within a BPMN-based Process.

A BPMN-based Process:

- must contain at least one Start Event (it may contain several Start Events);
- must contain at least one End Events or an Activity with no outgoing flow;

Apart form that, it can contain an arbitrary number of activities, events, and gateways connected with sequence flows. The elements in the flows must create an uninterrupted workflow and meet any other modeling rules that apply.

4.4.6.2.1 Instantiation of Plans and BPMN Processes

When a Process with a None Start Event is instantiated:

1. Process namespace is created.
2. Local process context is initialized.
3. Process instance becomes *Running* and one Start Event produces token).
4. If there are no activity in the workflow (no more tokens in workflows), the Process instance is terminated and becomes *Finished*.

When a Process with another type of Start Event is instantiated:

1. Process namespace is created.
2. The respective Start Event condition is checked.
3. If the Start Event conditions are fulfilled, the local process contexts are initialized.
4. The Start Event triggers execution (releases the token) and the process instances becomes *Running*.
5. If there is no more activity in the workflow (no more tokens in workflows), the Process instance is terminated and becomes *Finished*.

   Note: If a BPMN-based Process instance ends with an uncaught Error End Event, the last transaction is rolled back.

Events Flows Activities Gateway Swimlanes

4.4.6.3 Events

An *Event* is any element in a BPMN-based Process or Plan model that triggers or terminates the workflow, thus modifying the execution pace or causing a flow change.

Based on the position and function in the workflow, the following types events are available:

- Start Events trigger workflow execution by creating contexts and producing tokens.
- Intermediate Events delay processes based on events or handle an event produced in an activity if used as a boundary element.
- End Events consume the incoming token.

4.4.6.3.1 Start Events

A Start Event indicates where a particular workflow starts. When triggered, it creates the context for the element it is in, that is a process, plan, or subprocess, and, in this context, produces a token, which leaves through its outgoing flow.

A Start Event has no incoming flow and only one outgoing Normal Flow.

In *Plans of Goal-based Processes and Sub-Processes*, its None Start Event is triggered when the Plan becomes *running*. A Plan can contain only a None Start Event.

While a model instance is running, any Start Events in its *BPMN-based Process* create a Process instance when their trigger occurs: if there is a Condition Start Event and its condition is true, the event is triggered when the model instance starts and whenever the condition becomes *false* and *true* again as long as the model instance is *running*.

Start Events with triggers are allowed in BPMN-based processes and inline event sub-processes:

- In a BPMN-based process: Start Events whose triggers occurred, are triggered when the model instance is created. Each Start Event creates its Process instance in the Model instance. If the trigger occurs again while the process instance is *running*, the respective Start Event creates a new Process instance.
- In an [inline event sub-process], if the trigger occurs at any time while the parent process instance or sub-process instance is running, the events are triggered: they create an instance of their sub-process and produce a token. Note that the inline event subprocess can be also a reusable sub-process so the Start Events can be in the referenced Process.

4.4.6.3.1.1 None Start Events

A *None Start Event* is triggered when its parent context is created. It does the following depending on its location:

- When in a BPMN-based Process, it creates the context of the process instance and produces a token when the model instance becomes running;
- When in a Sub-Process: it creates the context of the subprocess and produces a token when the sub-Process is triggered, that is, it receives a token via its incoming flow or, if the sub-process does not have an incoming flow, at the moment it is triggered.
- Plan, once the Plan becomes *running*, the None Start Event of the Plan Model is triggered.

Figure 4.3 None Start Event

4.4.6.3.1.2 Conditional Start Events

A *Conditional Start Event* defines a boolean condition expression as its trigger and starts a BPMN-based Process or an inline-event subprocess:

- When in a BPMN-based Process, the event creates and triggers an instance of the process:
  - when its condition is *true* at model instantiation
  - when the condition, previously evaluated to *false*, becomes *true* while process instance is running.
- When in an inline-event subprocess, the event creates and triggers an instance of the sub-process
  - when its condition is *true* at process instantiation
  - when the condition, previously evaluated to *false*, becomes *true* while process instance is running.

Note that the condition cannot reference local context since the context does not exist at the moment when it is evaluated.

Figure 4.4 Conditional Start Event

Conditional Start Event Attributes

- **Condition** evaluated when the event is triggered. If true, the event produces a token.

4.4.6.3.1.3 Signal Start Events

A *Signal Start Event* creates the context of a BPMN-based Process or an inline-event subprocess and produces a token when it receives a Signal that passes through the *Filter*.

Figure 4.5 Signal Start Event

- **Filter**: filter for the expected Signal object
- **Signal**: reference where to store the received signal value

4.4.6.3.1.4 Timer Start Events

A *Timer Start Event* defines a timer trigger. It starts a BPMN-based Process or an inline event subprocess:

- When in a BPMN-based Process, the event creates and triggers an instance of the process when the point in time define by its Timer trigger occurs or periodically always when the specified time period elapses while the parent model instance is *running*.
- When in an inline-event subprocess, the event creates and triggers an instance of the sub-process when the point in time define by its Timer trigger occurs or periodically always when the specified time period elapses while the parent process instance is *running*.

If the specified point in time has already occurred, the event is triggered immediately when the parent context is instantiated. If both the Date and the Period property are defined, the start event is triggered periodically with the period countdown starting from the Date value. If the duration is null or negative, the start event is executed only once.

Timer Start Event Notation

Figure 4.6 Timer Start Event

Timer Start Event Attributes

- **Date**: date when the event triggers execution, that is, produces a token
- **Period**: periodicity of process instance triggering

4.4.6.3.1.5 Escalation Start Events

An *Escalation Start Event* creates the context of an inline event subprocess and produces a token when it receives an Escalation object while its parent instance is *running* (parent can be either a process instance or a sub-process instance if the sub-process is nested in another sub-process).

In other than an Inline Event Sub-Process, the event is ignored.

Figure 4.7 Escalation Start Event

Escalation Start Event Attributes

- **Filter** defines the escalation codes of accepted Escalation objects. If the object does not have any of the defined codes, it is ignored. If no codes are defined, any Escalation object triggers the event.
- **Escalation** is the reference to a local or global context (for example, a variable or a variable field) that will store the received Escalation object.
- **Escalation code reference**: Event consumes only escalations with the defined escalation code (if the escalation code is not specified then the Event accepts all escalations).

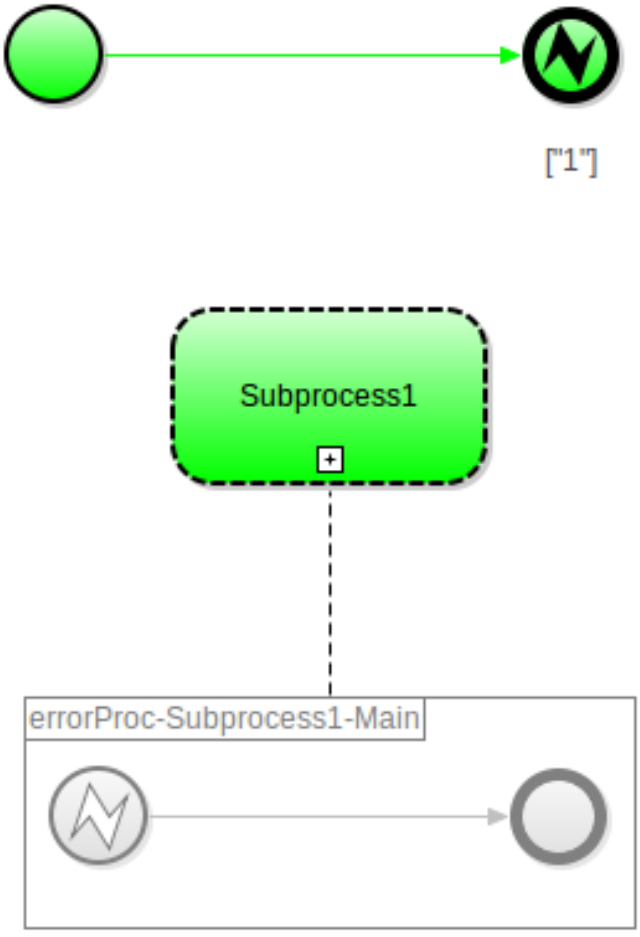
4.4.6.3.1.6 Error Start Events

An Error Start Event triggers its flow when it receives an Error object. You can use it only as a start event of an Inline Event Sub-Process: When the parent of the sub-process throws an error and the error code matches a code in the Error Code Filter of the Error Start Event, the event creates the context of the inline event sub-process and produces a token.

The parent can be either a process instance or a sub-process instance if that the sub-process is nested in another sub-process.

In other than an Inline Event Sub-Process, the event is ignored.

Figure 4.8 Error Start Event



Error Start Event Attributes

- **Error Code Filter** defines the filter that returns the expected errors (only error events from the Error reference that meet the filter criterion trigger the start event).
- **Error Code** is the reference to a local or global context (variable or a variable field) that will store the received error.

4.4.6.3.2 Intermediate Events

An *intermediate event* process element handles a predictable event that can occur during workflow execution, such as, an error, timeout, breached condition, etc. When such event occurs, the event produces a token, which takes the outgoing flow of the event.

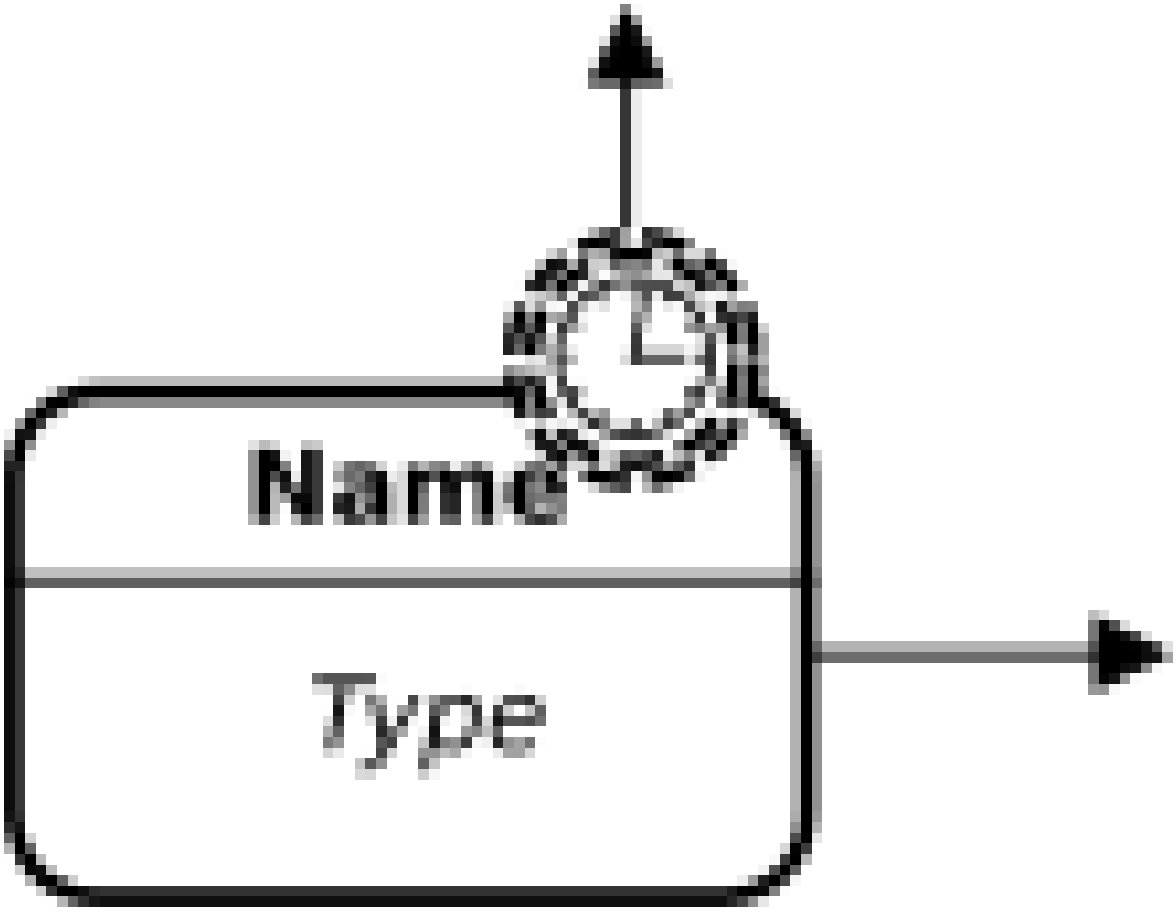
The scope the events listen to is defined by where they are placed:

- **on the boundary of an Activity**: If the activity is *Alive* and the event occurs, the outgoing flow of the Intermediate Event is taken.

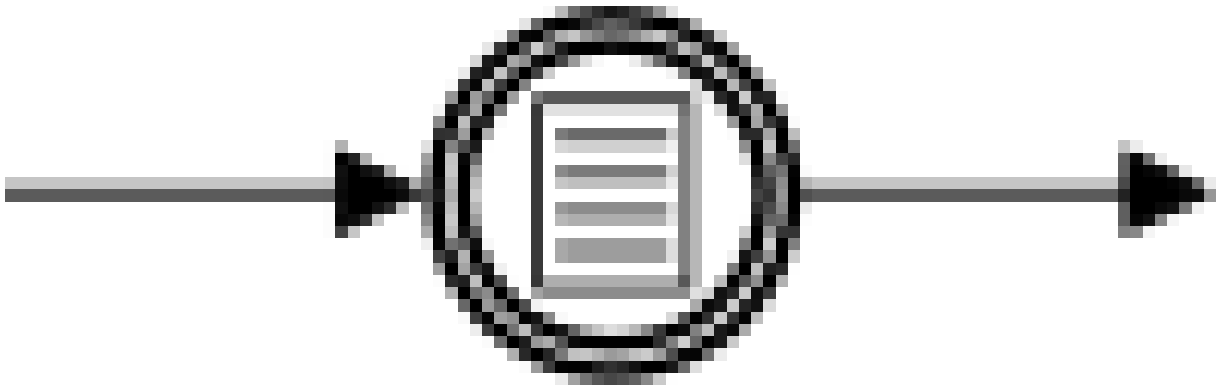
  > **Note:** If a boundary event is attached to a multi-instance Activity, all the instances of the Activity are interrupted when the Event is triggered.

  When the Intermediate Event on an activity is triggered, the activity can either continue or terminate. This depends on the interruptibility property of the Intermediate Event:

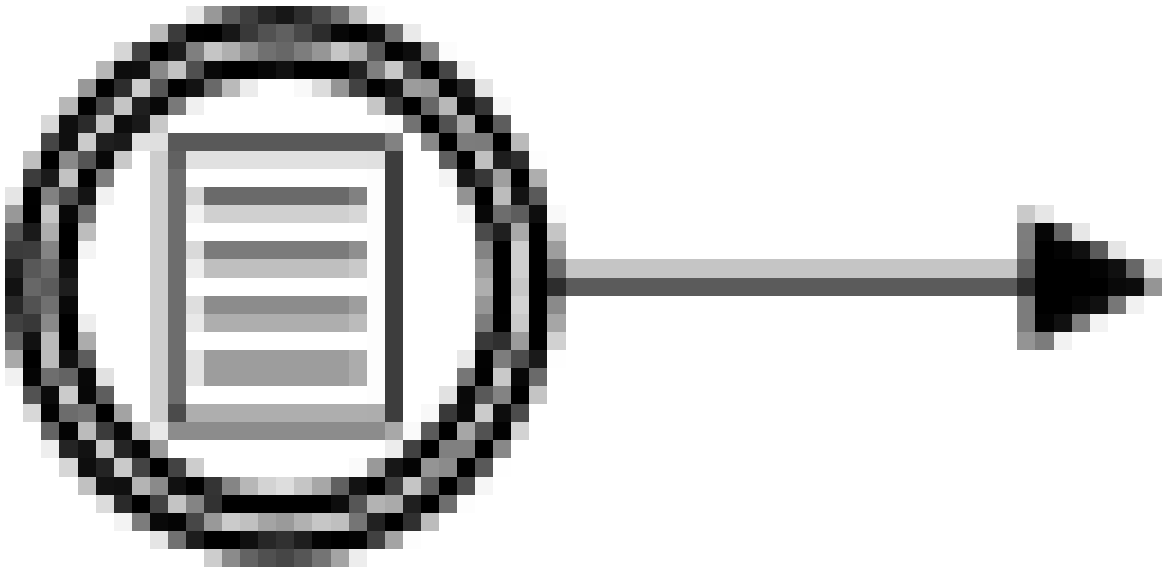
  - If the event is *interrupting*, the activity is deactivated when the event is triggered.
  - If the event is *non-interrupting*, the activity continues its execution: note that if the event occurs again while the Activity is alive, the boundary event is triggered again.



- **flow object with one incoming and one outgoing flow**: the event is triggered when the flow reaches the event. Once the event occurs, the execution continues via the outgoing flow of the Intermediate Event.



- **flow object with no incoming and one outgoing flow**: the event is triggered whenever the event condition occurs while its context is *running*.



4.4.6.3.2.1 Timer Intermediate Events

A *Timer Intermediate Event* either temporarily delays a workflow or triggers a workflow at a moment in time:

- If placed in a workflow as a flow object with one incoming and one outgoing Flow, the event pauses the workflow execution for the defined time period or until the defined point in time has occurred.

Note: The target element of a Sequence Flow leaving a Gateway must not be a Timer Intermediate Event.

- If placed on an Activity border, as soon as the Activity becomes active, the countdown of the time period is triggered (or the event is waiting for the point in time to occur).

  When the time elapses and the Activity is still active, the timer event triggers its outgoing flow. If the Activity finishes before the time elapses, the event is not used (Flow leaving the Activity is taken).

  There can be several Timer Intermediate Events on one Activity: whichever happens first is used. Similarly, also if an event defines both the duration and the date whichever happens first is used.

The event must define its time event either as a time duration—for example, 24 hours—or as a point in time, for example, a date: 2022-2-2) in one of its properties:

- **Date**: date when the event triggers execution, that is, produces a token
- **Duration**: periodicity with which the event triggers execution, that is, produces a token

Figure 4.9 Non-Interrupting Timer Intermediate Event

Figure 4.10 Interrupting Timer Intermediate Event

4.4.6.3.2.2 Error Intermediate Events

An *Error Intermediate Event* serves to define a workflow that is taken after an Activity produces an error: it explicitly produces an error or it ends with an Error End Event.

The element is placed on boundaries of Activities, that is, Tasks or Sub-Process. An Activity can have multiple Error Intermediate Events attached to its boundary.

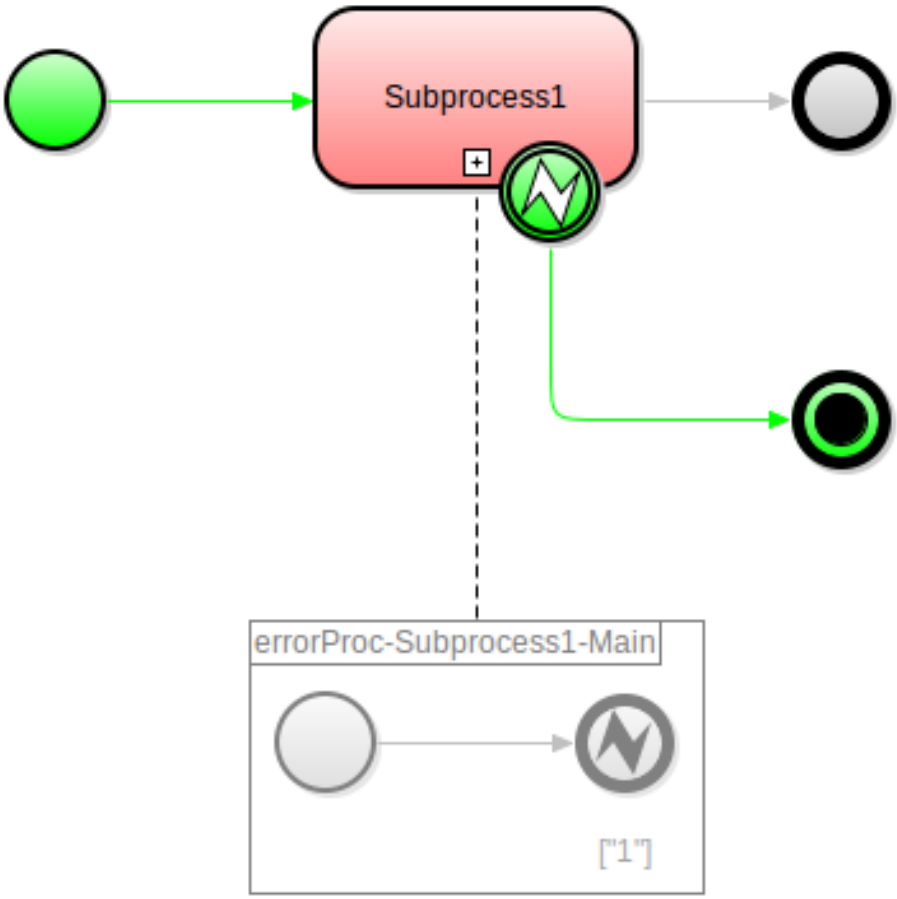
An Error Intermediate Event can define a set of errors it may consume. The errors are identified by their error code. Once the event catches one of the error codes, its outgoing Flow is taken. If no error code is specified (`null`), the Error Intermediate Event catches any Error produced in its child contexts unless it is processed by other Error Intermediate Event, which explicitly specifies the error code.

If there are several Error Intermediate Events attached to an Activity, the Error is caught and processed by the Error Intermediate Event with the priority depending on the way its error codes are defined:

1. only the error code of the particular error;
2. set of error codes including the particular error code;
3. no particular error code.

Figure 4.11 Non-Interrupting Error Intermediate Event

Figure 4.12 Interrupting Error Intermediate Event



Error Intermediate Event Attributes

- **Error Code Filter** defines the filter that returns the expected errors (only error events from the Error reference that meet the filter criterion trigger the event).

  If no filter is defined of returns `null`, the event catches all errors.

- **Error Code** is the reference to a local or global context (variable or a variable field) that will store the received error.

4.4.6.3.2.3 Cancel Intermediate Events

A *Cancel Intermediate Event* is an event that handles a Cancel End Event with the aim to cancel the actions of a Transaction Sub-Process.

Hence, it can be attached only to a boundary of a Transaction Sub-Process that contains a Cancel End Event. When the Sub-Process finishes with the Cancel End Event, the outgoing flow of the Cancel Intermediate Event attached to its boundary is taken.

Figure 4.13 Non-Interrupting Cancel Intermediate Event

Figure 4.14 Interrupting Cancel Intermediate Event

4.4.6.3.2.4 Conditional Intermediate Event

A *Conditional Intermediate Event* is an intermediate event which checks a particular condition and is triggered when the condition becomes *true*:

- If placed on an Activity boundary, the condition is checked continuously while the Activity is *active* or *running*. At the moment the condition becomes true, the execution of the Activity becomes *interrupted* and the Flow leaving the Conditional Intermediate Event is taken.

  Note: There can be one or multiple Conditional Intermediate Events attached to one Activity.

- If used in a workflow as a flow element with an incoming and outgoing flow, when the token reaches the Event, it is held until the condition becomes `true`.

Figure 4.15 Non-Interrupting Conditional Intermediate Event

Figure 4.16 Interrupting Conditional Intermediate Event

4.4.6.3.2.5 Throw Signal Intermediate Event

A *Throw Signal Intermediate Event* produces a Signal that can be caught by Start Signal Events and Signal Intermediate Events of particular model instances.

It can be used only as a workflow element with one incoming and one outgoing Flow.

When reached during execution, it sends the defined Signal to the defined model instances. If no model instance is defined, the Signal is sent to its parent model instance. In the target model instances, it is consumed by all running Catch Signal Intermediate Events waiting for the Signal at the given moment and the Signal Start Events.

The Throw Signal Intermediate Event has to define the signal and the target model instances:

- **Model instances** defines the IDs of model instances to which you want to send the Signal.
- **Signal value** defines the Signal value.

Figure 4.17 Non-Interrupting Throw Signal Intermediate Event

Figure 4.18 Interrupting Throw Signal Intermediate Event

4.4.6.3.2.6 Catch Signal Intermediate Event

A *Catch Signal Intermediate Event* is triggered when it catches a Signal. Subsequently it produces a token which takes its outgoing Flow.

It can be either attached to a boundary of an Activity or used as a workflow element with one incoming and one outgoing Normal Flow. When active it waits until it has receives a Signal.

It can define a filter for the Signals it acceptst; if the received signal does not meet the defined filter criteria, it does not send a token.

The Catch Signal Intermediate Event cannot handle Signals of the type Reference, goal, Plan, or ModelInstance. Catching such signals causes a runtime exception.

Catch Signal Intermediate Event defines:

- **Filter**: filter of the accepted Signal
- **Signal**: reference to a storage, where the caught signal is stored

Figure 4.19 Non-Interrupting Catch Signal Intermediate Event

Figure 4.20 Interrupting Catch Signal Intermediate Event

4.4.6.3.2.7 Throw Escalation Intermediate Event

A *Throw Escalation Intermediate Event* is an Event which sends an Escalation object when triggered.

It can be used only as a workflow object with one incoming and one outgoing flow. When reached during execution, it sends the defined Escalation object, which is propagated throughout its context and up to higher contexts either until consumed by a Start Escalation Event or a Catch Escalation Intermediate Event or until no higher context is available.

Throw Escalation Intermediate Event Attributes

- **Escalation code** defines the escalation code sent with the escalation object.
- **Escalation** defines the payload of the Escalation object.

  Important: It is not possible to filter according to information in payload. You can use only the escalation code.

4.4.6.3.2.8 Catch Escalation Intermediate Event

A *Catch Escalation Intermediate Event* is a boundary intermediate event that catches and consumes an Escalation object that was thrown in its Activity and meets the filter criterion, and produces a token. It can be used only as a boundary element on an Activity and with an outgoing Normal Flow.

The event is active while the Activity is active. When the Escalation object is thrown in the Activity or its child contexts, the boundary Catch Escalation Intermediate Event consumes the Escalation object and produces a token which takes its outgoing Flow. Note that just like other boundary Intermediate Events, it can be non-interrupting or interrupting.

Catch Escalation Intermediate Event must define a reference object, where the object is stored. It can optionally define a filter definition: if the received Escalation object does not meet the filter criteria, it is ignored.

Catch Escalation Intermediate Event Notation

Figure 4.21 Non-Interrupting Catch Escalation Intermediate Event

Figure 4.22 Interrupting Catch Escalation Intermediate Event

Catch Escalation Intermediate Event Attributes

- **Filter** contains the escalation codes of accepted Escalation objects.
- **Escalation** defines a reference to a storage, where the caught escalation signal is to be stored.
- **Escalation code reference**: Event consumes just escalations with the defined escalation code (if the escalation code is not specified then the Event accepts all escalations).

4.4.6.3.3 End Events

An *End Event* ends a workflow (consumes a token).

A workflow has one or multiple End Events and one or multiple Sequence Flows can enter an End Event. No outgoing Flow is allowed.

There are multiple types of End Events, which differ by the actions they perform when an execution flow enters the end events, that is, how they behave when they consume a token.

4.4.6.3.3.1 Simple End Events

A *Simple End Event* is the basic end Event type which consumes the incoming token; other tokens in the workflow remain uninfluenced. The particular workflow is finished successfully if it does not contain any other tokens.

Simple End Event Notation

Figure 4.23 Simple End Event

4.4.6.3.3.2 Terminate End Event

When a workflow reaches a *Terminate End Event*, all tokens in the namespace are consumed and the execution ends successfully.

If there are other activities in the workflow, they are instantly terminated (as soon as a Terminate End Event consumes one token, any other tokens in the workflow are discarded).

Terminate End Event Notation

Figure 4.24 Terminate End Event

4.4.6.3.3.3 Error End Events

A *Error End Event* is an end event that implement error handling on a workflow end. The actions when triggered depend on the type of the parent namespace of the End Event. When a token enter an Error End Event, enters an Error End Event the following happens:

- if in a Sub-Process:

The End Event generates an error with the respective Error Code (see Error) and the workflow is finished. The error is distributed gradually to higher namespaces and can be caught by an Error Intermediate Event contained in any parent namespace as to trigger a compensation process (see Error Intermediate Event).

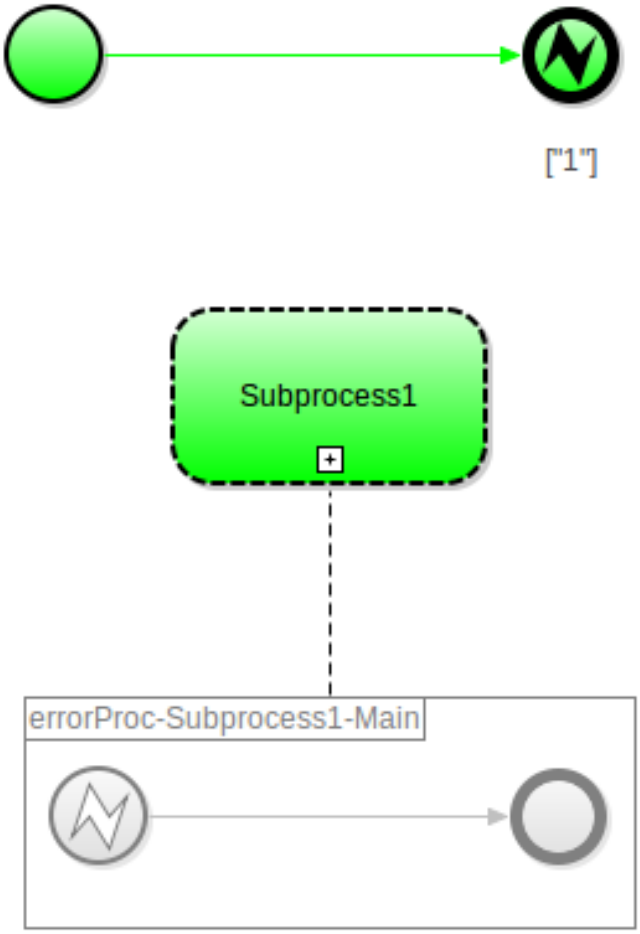
- Plan Models

  If a Plan Model ends with an Error End Event, the Plan fails (becomes `Failed`).

- BPMN-based Processes

  If a BPMN-based Process instance finishes with an Error End Event, the last transaction is rolled back and an exception is created.

Figure 4.25 Error End Event



Error End Event Attributes

- **Error code** defines the error code to be sent.

4.4.6.3.3.4 Cancel End Events

A *Cancel End Event* is a special End Event which ends a Transaction Sub-Processes.

When a token enters a Cancel End Event and the parent Transaction Sub-Process has no Cancel Intermediate Event attached, the last transaction is rolled back.

If there is a Cancel Intermediate Event attached to the parent Sub-Process, the Sub-Process execution fails (the Sub-Process becomes Interrupted) and the outgoing Flow of the attached Event is taken.

Cancel End Event Notation

Figure 4.26 Cancel End Event

4.4.6.3.3.5 Throw Escalation End Events

When a workflow ends with a Throw Escalation End Event, the event generates an Escalation object with the respective code and the workflow finishes. The escalation is distributed gradually to higher context and can be caught by a Catch Escalation Intermediate Event or Escalation Start Event in any parent context.

Escalation End Event Notation

Figure 4.27 Escalation End Event

Escalation End Event Attributes

- **Escalation code** defines a code sent with the escalation object that serves as the escalation identifier.
- **Escalation** defines the payload of the Escalation object.

  Note: It is not possible to filter according to information in payload. You can use only the escalation code.

4.4.6.3.3.6 No Exit End Events

A *No Exit End Event* is an end event that consumes a token in the workflow of an Inline Event Sub-Process without triggering the Sub-Process' outgoing Flow: no token for any outgoing Flow if present is produced but the Sub-↩ Process execution finishes with success.

If a Process with such an event is used by a Reusable Inline Event Sub-Process which is not used as part of the workflow, that is, it does not have an outgoing Flow, the No Exit End Event behaves as a Simple End Event.

Figure 4.28 No Exit End Event

4.4.6.4 Flows

A *Sequence Flow* is a connector which establishes an oriented relationship between two elements of a workflow (Activities, Events, and Gateways) and defines their execution order. The workflow is taken following the indicated direction and execution semantics of other workflow objects. In GO-BPMN, only the Normal Flow is supported.

> **Note:** Default Flow is considered a special case of Normal Flow.

4.4.6.4.1 Normal Flow

A *Normal Flow* is a Flow type showing the order of the process Activities it is connecting.

It has a source and a target and indicates the execution behavior.

If the source of a Normal Flow is an Exclusive Gateway, a Normal Flow can be provided a guard. A guard is a Boolean condition (defined using Expression Language), which has to be true, before the Flow is taken. If the guard condition is not true, the respective Normal Flow cannot be used (the token cannot pass the flow).

A source or target of a Normal Flow may be:

- Event
- Activity
- Gateway

On execution, the flow transfers the token from the source to the target. If a guard of the flow is defined (the source has to be an Exclusive Gateway), on token receiving, the Flow guard is evaluated:

- if true, the token is send to the target (the target is triggered);
- if false, the token is held by the Flow and the Flow guard is evaluated continuously.

A Normal Flow is depicted as a solid single line with an arrowhead directed toward the target element. If a guard is provided, it is shown in square brackets near the arrow.

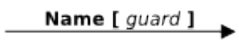


Figure 4.29 Normal Flow with a guard

Properties:

- **Guard** defines a condition, which has to be to true if the token is to pass through the Flow.

4.4.6.4.2 Default Flows

A *Default Flow* is a special Normal Flow, which is taken if no other Flow can be used.

The Default Flow represents the last option among the available Flows leaving an Exclusive Gateway. If guards of other Flows prevent them from being taken (they are evaluated to false), the Default Flow is used.

A source element of a Default Flow is an Exclusive Gateway.

The Default Flow has a default marker (slash) show at the beginning of its arrow line.

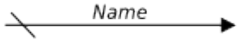


Figure 4.30 Default Flow

4.4.6.5 Activities

An *Activity* represents a piece of work that need to be done and that either by a human or a system (machine). The term covers Tasks, which are considered atomic, and the Sub-Process, which encapsulate other elements.

An Activity is executed whenever it receives a token: the token is typically passed by the incoming flow of the Activity: each token results in a single Activity execution.

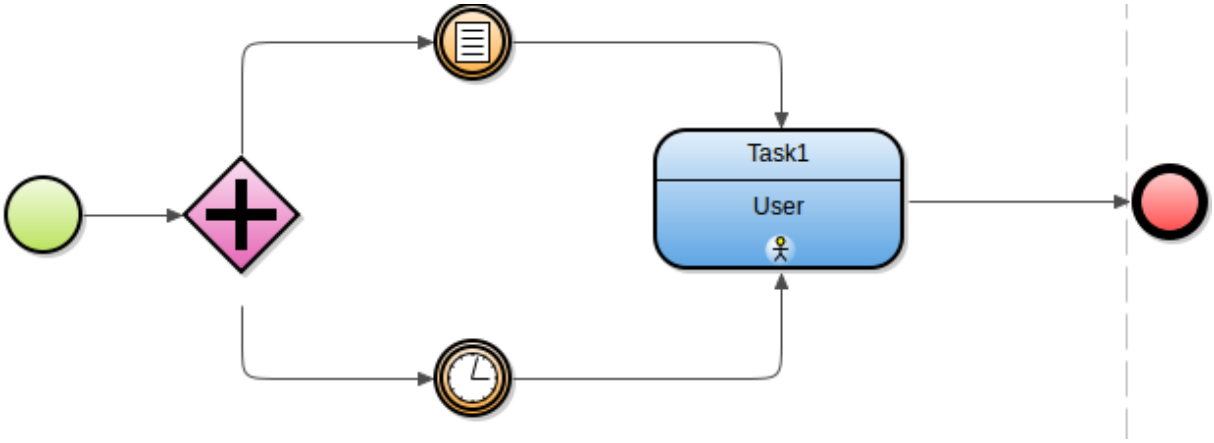


Figure 4.31 Caption text

In the example above, the parallel gateway "splits" the incoming tokens into two tokens: further down your execution flow, the tokens enter an Activity: each token will result in its own Activity execution, in this case, the activity is executed twice: once for the token from the Condition Intermediate Event and once for the token from the Timer End Event.

The logic of the execution depends on the activity type and additional mechanisms, such as, looping. Mind that looping does not influence the number of Activity executions.

An activity has an arbitrary number of incoming and one or none outgoing sequence flow. The flows influence the execution in the following ways:

- Incoming flows:
  - If an activity does not have an incoming Flow, it is instantiated when the process is instantiated. If there are multiple Activities with no incoming flows in one Process, all such Activities are instantiated (multiple tokens are produced).
  - If it has one or multiple incoming flow, it is instantiated always when the any of the flows send a token to the activity.
- Outgoing flows:
  - If an activity has no outgoing flow, the execution finishes along with the Activity execution (its token ceases to exist after the activity is accomplished).

Activities can have intermediate events attached to their boundary. These events react to specific events or conditions that can occur during the activity execution; for example, they can be used if the execution of an activity should time out after a certain amount of time.

4.4.6.5.1 Tasks

A *Task* is an atomic Activity in a workflow: it represents the smallest logical piece of work, which cannot be broken down any further (for example, sending a file, filling in a questionnaire, displaying a text, etc.).

A Task can have none or multiple incoming, and none or one outgoing Normal Flow.

When the workflow hits a task, the task becomes alive and its logic is executed: if the execution is successful, the task becomes accomplished. If the parent model instance is suspended, an alive task also becomes Suspended.

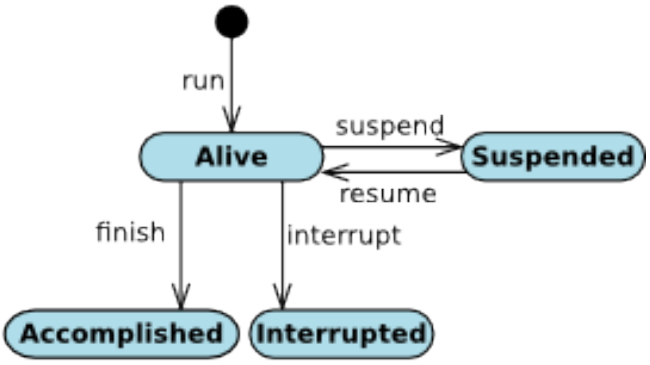


Figure 4.32 Task Lifecycle

Task types can be reflected as Records: this allows you to create instances of tasks by instantiating them as Record instances. Typically you will then send the Record as a parameter to the `Execute Task` which makes sure its gets executed as a task in your workflow. In the LSPS implementation, a task type is reflected after you set the `Create activity reflection type` flag.

> **Note:** To-dos generated by human tasks can be saved. The human task remains alive while the to-do is saved.

Every task is of a particular task type: the task type determines what kind of action the task performs. Depending on the type, every task has a set of parameters, which define the input and output data for the task execution.

You can execute a task multiple times in one execution using the looping mechanism.

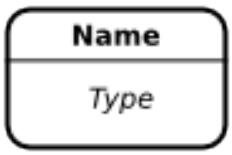


Figure 4.33 Task notation

In addition to the common modeling element attributes and apart from attributes specific for a task type, tasks define the following attributes:

- **Public**: task type visibility

  If not Public, the task type cannot be used by any importing modules.

- **Deprecated**: flag which signalizes that the task will be removed in the next version of the model or application If the flag is selected, on validation, a warning notification is logged about that the user is using a deprecated task.
- **Parameters**: task specific parameters Parameters define the following properties:
  - **Name**: parameter name
  - **Type**: data type of the parameter
  - **Required**: if true the parameter is obligatory
  - **Dynamic**: if true the parameter value is wrapped in a non-parameteric closure automatically (the parameter is then processed as `{ -> <parameter_value> }`.) This is the case of the `uiDefinition` parameter in the `User` task. Hence the user does not need to define the uiDefinition parameter as `{ -> <form()> }` but only provides directly the call to the form (`form()`).
- **Class name**: name of the task class

4.4.6.5.2 Sub-Process

A Sub-Process is a compound Activity encapsulating a workflow: it is a “process within a process” that serves to organize the workflow content.

It exists in its own context with its variables and parameters and its workflow is triggered as part of its process. It can be triggered by token passed by its incoming flow or by an event: if you want to trigger a Sub-Process with events, use an Inline Event Sub-Process.

A Sub-Process can be of the following types:

- Embedded Sub-Process defines its workflow in-place inside the parent Process or Sub-Process
- Reusable Sub-Process defines a reference to a process and uses this process as its content

4.4.6.5.2.1 Embedded Sub-Process

An *Embedded Sub-Process* is a Sub-Process that is defined as part of its parent Process or Sub-Process. Its workflow must contain one None Start Event or one or multiple Activities with no incoming Flows.

On runtime, a sub-process instance is created when the token passes to the Sub-Process through its incoming flow: then the None Start Event produces a token and Activities with no incoming Flow are triggered. The execution finishes when there are no tokens in the sub-process instance.

Also, an embedded Sub-Process can be marked as a *transaction* embedded Sub-Process to mark it as comprising a business transaction that is "atomic". Its workflow can finish with a Cancel End Event: the subprocess must have a Cancel Intermediate Event on its border with an outgoing flow: When the flow finishes with the Cancel End Event, the outgoing flow of the border Cancel Intermediate Event is taken.

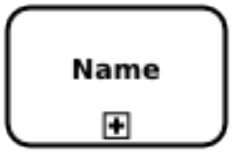


Figure 4.34 Embedded Sub-Process notation

4.4.6.5.2.2 Reusable Sub-Process

A *Reusable Sub-Process* references a Process: when triggered, it instantiates the Process as a subprocess.

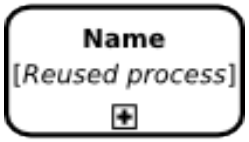


Figure 4.35 Reusable Sub-Process notation

Reusable Sub-Process Attributes

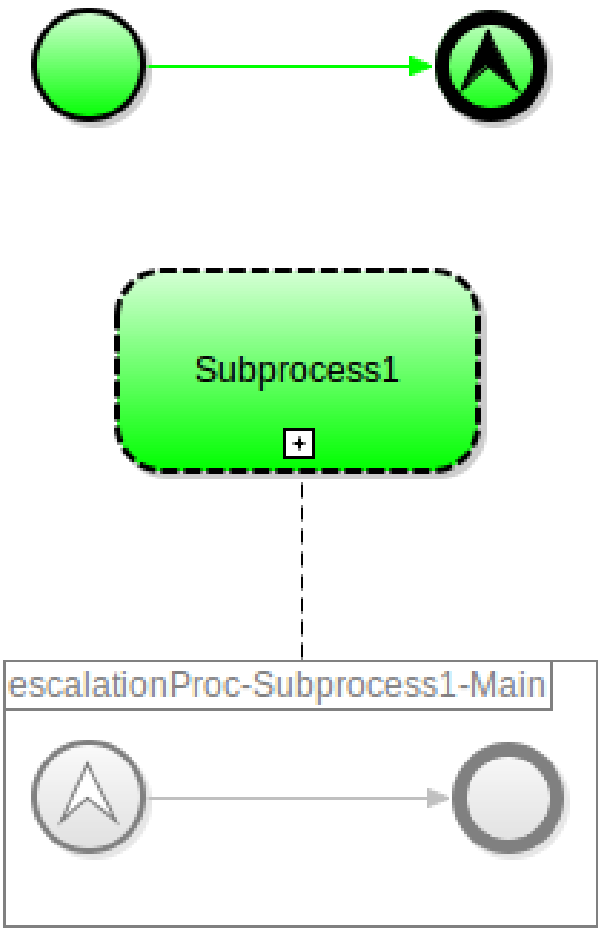
- **Referenced Process Name** defines the assigned Reusable Process.
- **Parameters** are key-value pairs used as parameters when the Sub-Process is instantiated (similar to process parameters)

4.4.6.5.2.3 Inline Event Sub-Process

When you mark a Sub-Process as an inline sub-process, its Start Events are considered part of the parent context, which is either a process or another sub-process. Such start events can be of different types so they are triggered when a particular event occurs in their parent Process or sub-process: When they start with a None Start Event they are triggered when the parent is triggered and in the case of other Start Events whenever the trigger event occurs in the parent.

> **Note:** Also a BPMN-based process can contain any type of Start Events: this allows you to trigger it when it is used in a Reusable Subprocesses.

For example, if a process instance finishes with an Escalation End Event, all Escalation Start Events in its inline event subprocesses receive the escalation and one creates its sub-process instance. This allows you to consume a token and at the same time, produce a token somewhere else while keeping the process instance running.



An Inline Event Sub-Process can be placed into the workflow *with an incoming and outgoing Flow* or *with no incoming or outgoing Flow*.

- When used as part of a Process or Sub-Process workflow with at least one incoming Flows and one outgoing Flow, the workflow inside the Sub-Process must contain a None Start Event or one or multiple Activities without an incoming Flow and it can contain an arbitrary number of Start Events other than the None Start Event. The subprocess is then triggered as follows:
  - None Start Event is triggered when a token passes through the incoming Flow into the Sub-Process: a new Sub-Process instance is created and the None Start Event produces a token.
  - Start Events of other types are triggered whenever their event occurs during the life of the parent instance, that is, either a Process or Sub-Process instance.
  - Activities with no incoming Flow are triggered whenever a Sub-Process instance is created and that as part of the Sub-Process instance.
- When part of a Process or Sub-Process with no incoming or outgoing Flows:
  - None Start Event is triggered when the parent Process or Sub-Process is triggered.
  - Start Events of other types are triggered whenever their event occurs during the entire life of the parent Process or Sub-Process.
  - Activities with no incoming flow are triggered whenever a Sub-Process instance is created and that as part of the Sub-Process instance.

Note: Inline Sub-Processes cannot use the looping mechanism.

4.4.6.5.3 Looping

Looping Activities are executed multiple times: however, the individual loop runs are not considered different Activity instances, that is, looping *does not* change the amount of tokens.

Loops can be executed in the following ways:

- standard (for): Activity is repeated successively (serially) the defined number of times unless the loop condition becomes false.
- multi-instance (foreach): Activity is repeated in parallel or in sequential manner over a list of items.

  Note: Alternatively, loops can be also created by cycling the workflow using flow elements: In such loops, a Gateway has an outgoing flow "returning" to a preceding Gateway.

Looping defines its iterator, which stores the number of the loop starting from 0 and is incremented by 1 after each loop.

4.4.6.5.3.1 Multi-Instance Looping

Multi-instance looping enables you to execute a loop of one Activity on a defined list of objects, and that, in a parallel or sequential way: when a token enters a multi-instance looping Activity, the system creates multiple instances of the Activity, for example, multiple To-dos generated by the same Task. However, all the instances represent the execution of the single Activity: only one token exists even though there are multiple instances of the Activity. The activity becomes accomplished only after all its instances are finished; and similarly, if one of the Activities fail with an Error for example, all the instances finish.

Looping iterator is initialized to 0 and incremented by 1 after each loop.

Multi-instance looping defines the following:

- list of elements the looping is performed for (For each attribute);
- how the multi-instance loop is to be executed:
  - sequential: when one instance of the Activity finishes, a new instance is created
  - parallel: activity instances for all list objects are started at once; the Activity finishes after all its instances finished.

Multi-Instance Looping Notation

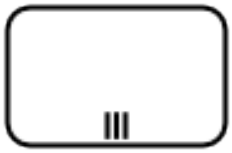


Figure 4.36 Activity with multi-instance loop

Multi-Instance Looping Attributes

- **Loop activity** defines the activity to be looped.
- **For each** contains a list of objects to be used in the loops.
- **Ordering** defines loop ordering strategy (sequential or parallel).

4.4.6.5.3.2 Standard Looping

Activity with a Standard looping definition is repeated serially.

On each loop of standard looping, the system checks before each loop if the loop condition is true. If it is false, the looping finishes. This check can be performed also after each loop. The maximum number of loops is determined explicitly by the loop maximum. Once this maximum is reached, the looping finishes regardless of loop condition. When the looping is finished the Activity release a single token to its outgoing flow.

> **Note:** If the loop condition is evaluated always before the execution of the loop and the condition is at the first evaluation evaluated false the Activity will be never performed.

Looping iterator is initialized to 0 and incremented by 1 after each loop.

Standard Looping Notation

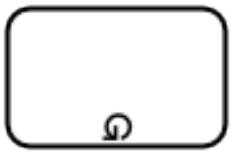


Figure 4.37 Activity with standard loop

Standard Looping Attributes

- **Loop**: name of the loop iterator
- **Loop condition**: condition that finishes the loop; it defines also its time time, the moment when the loop condition is checked (before or after a loop)
- **Loop maximum**: maximum number of loops

4.4.6.6 Gateway

A Gateway is a workflow modeling element used to direct, or fork or merge workflows.

GO-BPMN defines the following Gateway types:

- Parallel Gateway creates or merges multiple flows
- Exclusive Gateway selects one flow out of multiple flows

> **Important:** The target element of a Sequence Flow leaving a Gateway must not be a Timer Intermediate Event.

4.4.6.6.1 Parallel Gateways

Parallel Gateway changes the number of parallel flows in the process.

It can have multiple incoming and outgoing flows:

- The gateway waits until it has received workflows from all incoming flows (tokens from all incoming flows must enter the gateway).
- Once all incoming flows have reached the gateway, all outgoing flows are taken (possibly multiple tokens are produced).

Figure 4.38 Parallel Gateway notation

4.4.6.6.2 Exclusive Gateways

An *Exclusive Gateway* directs the flow so that exactly one outgoing flow is taken depending on the circumstances, rendering it a decision-making mechanism similar to the *switch* language construct.

An exclusive gateway has one or several incoming normal flows and one or several outgoing flows. If there are multiple outgoing flows, each flow, apart from the default flow, must define a guard with a Boolean condition.

When a workflow enters an Exclusive Gateway, one of the following happens:

1. The first outgoing flow with the guard condition which is `true` is taken.
2. If no such flow is available, the default flow is taken.
3. If no default flow is available, the execution fails with a *NoValidBranchError*.

Figure 4.39 Exclusive Gateway notation

4.4.6.7 Swimlanes

Swimlanes serve to denote and set common performers for multiple elements of a process workflow: if a human task has no performer, the performer set the closest ancestor swimlane is used. This applies to elements in Reusable Sub-Processes.

Swimlanes can contain any element allowed by their parent entity, be it a Plan, BPMN-based Process, or Sub-↩ Process: if you are using Swimlanes in a Plan, you can use the same elements, which you would use in a Plan normally.

Pool

A Pool groups a workflow executed fully or partially by a common set of performers, typically one organization or department. It sets the common performer set: The performers property set on the Pool is adopted by process elements that require the performers parameter but do not define it themselves.

Each BPMN-based process, plan body, or sub-process may have an arbitrary number of Pools with one Pool marked as the Main Pool: Only the flow in the Main pool is executed. Non-main Pools serve for informational purposes only and are excluded from validation.

Pools may contain an arbitrary number of Lanes. Once a Pool contains at least one Lane, the Pool cannot contain any other BPMN elements: all flow elements must be in a Lane.

Note Sequence Flows cannot cross Pool boundaries.

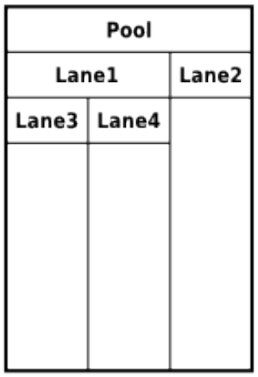


Figure 4.40 Vertical expanded Pool with Lanes

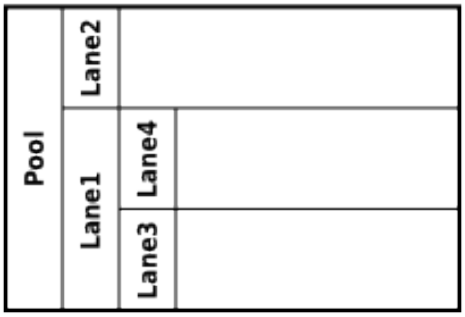


Figure 4.41 Horizontal expanded Pool with Lanes

Figure 4.42 Vertical collapsed Pool with Lanes

Figure 4.43 Horizontal collapsed Pool with Lanes

Pool Attributes

- **Main** is a Boolean attribute; if true the workflow in the Pool is executed.
- **Performers** defines the performers of the Tasks in the Pool if the Tasks do not define their performers or the setting is overriden by a child Lane.

Lane

A Lane holds a part of a workflow of its parent, either a Pool or another Lane, and defines the performers of the activities in the Lane.

A Lane can contain other Lanes: if a Lane contains a Lane, it cannot contain any other BPMN elements. All workflow elements in a Lane take over the Performers parameter value of the Lane. If a parent Lane does not define the parameter, the parameter of the "closest" parent Lane is used. If no such Lane exists, the performers set on the Pool are used.

Sequence Flows of Lane workflows can cross Lane boundaries.

4.4.7 Goal Model

A Goal Model is a hierarchy of Achieve and Maintain Goals, and Plans and their connections in a Goal-based Process. It specifies what needs to be achieved by the process while the Plan bodies specify how to achieve it.

After all executable Modules in a Model are instantiated, processes of executable Module instances are instantiated and Goal processes trigger all their top Goals: The top Goals become *ready* and if their pre-condition is met then *running*. When a Goal becomes *running*, all its sub-Goals become *ready*: the Goal triggers all its sub-Goals; a *ready* sub-Goal can become *running* and commit further to their child modeling elements, that is, either to Achieve Goals or to Plans. If a Goal has several Plans as its child elements, the Goal triggers one Plan, and that either the first Plan with its pre-condition evaluated to true or if no Plan pre-conditions are provided, a randomly chosen Plan.

> **Important:** Since pre-conditions are evaluated continuously, make sure they do not perform any assignments to prevent performance issues.

If a Plan fails, another Plan is triggered. If all Plans fail, the parent Goal fails.

4.4.7.1 Achieve Goal

An Achieve Goal is a modeling element that represents a condition or a state that is to be achieved. Achieve Goals typically represent explicit objectives of the process, such as, placing an order, producing a car, sending a message, etc.

An Achieve Goal exist in a goal hierarchy: it may have no parent modeling element or it can have another Achieve Goal as its super-Goal. It can have one or multiple child elements connected with the Decomposition flow. Its child elements can be either Achieve Goals or Plans: It is not possible to decompose a single Achieve Goal into both Goals and Plans.

On runtime, Achieve Goals go through a set of statuses defined by their life cycle).

An Achieve Goal can define the following:

- **Pre-condition** is a Boolean expression checked continuously while an Achieve Goal is Ready. If a pre-condition becomes `true`, the Goal becomes Running.

  Important: Since pre-conditions are evaluated continuously, make sure they do not perform any assignments to prevent performance issues.

- **Deactivate condition** is a Boolean expression checked continuously while an Achieve Goal is Not finished (Inactive/Ready/Running). If evaluated to `true`, the Achieve Goal, its sub-Goals and Plans become Deactivated immediately.
- **Visibility** defines Goal access rules: if set to *Public*, the Goal is accessible from the entire model; if you unselect the option, it is private and accessible only from within its module.

Figure 4.44 Achieve Goal in the iconic notation

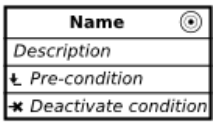


Figure 4.45 Achieve Goal in the decorative notation

4.4.7.1.1 Achieve Goal Life Cycle

When a GO-BPMN process is instantiated, its Goals are instantiated as well: all Goals become *inactive*. An **inactive** Achieve Goal is not prepared for running and waits for its activation. All top Achieve Goals of the process instance become *ready*: When a goal becomes **ready**, it has its pre-condition checked. If the pre-condition evaluates to `true`, the goal becomes *running*. The pre-condition is checked continuously while the Goal is ready.

When a goal becomes **running**, it activates either all its sub-goals or one Plan, that becomes *ready*.

A Running Achieve Goal can become *achieved* or *failed*:

- It becomes *achieved* when one of its Plans is achieved or all of its sub-Goals are *achieved* or *deactivated*;

- It becomes *failed* when at least one of its sub-goals failed or none of its Plans was achieved.

Important Achieve Goal status can be influence by the deactivation and activation mechanism.

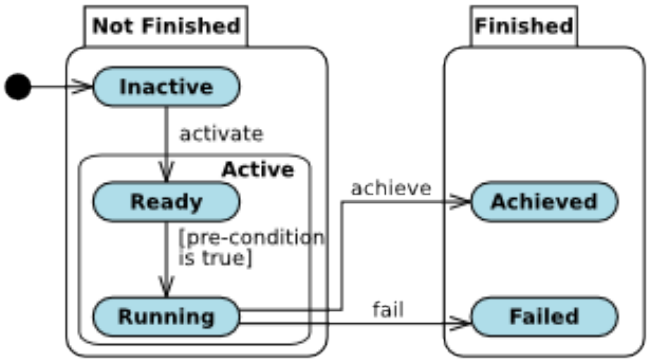


Figure 4.46 Lifecycle of Achieve Goals

4.4.7.2 Decomposition

In a Goal model, Decomposition represents a relationship used either for detailing a Goal to its sub-Goals, or specifying how a Goal can be achieved by corresponding Plans.

A Decomposition is presented as a connector used in Goal hierarchies and enables you to establish acyclic oriented relationships between:

- two Achieve Goals,
- Maintain Goal and Achieve Goal, and
- Goal and Plan.

Maintain Goals cannot be targets of a Decomposition (every Maintain Goal is a top Goal).

During execution the following rules apply:

- If a Goal is decomposed in sub-Goals, all the sub-Goals are committed to and executed. The super Goal is accomplished only if all its sub-Goals are Achieved or Deactivated.
  - If one of the sub-Goal fails, the super Goal fails and any other sub-Goals become deactivated. The failure is distributed also to other parent Goals of the failed Goal, that is, if a Goal, which fails, has any parent Goals, these parent Goals fail as well.
  - Super Goal can succeed even if one or multiple Goals were deactivated. If a Goal is decomposed in several Plans, only one of the Plans is triggered (either the one with the pre-condition evaluated to `true` or the first randomly-chosen Plan). When a Plan is Achieved, its parent Achieve Goal becomes Achieved or its parent Maintain Goal becomes Ready.
  - If a Plan fails, an alternative Plan is selected.
  - If no other Plan can be triggered, that is, if all available Plans fail, the parent Goal fails.

Figure 4.47 Decomposition notation

4.4.7.3 Maintain Goal

A Maintain Goal serves to make sure a condition is true and that either while an Achieve Goal or while the process is running. The Achieve Goal or the Process represent the *scope* of the Maintain Goal: if the Achieve Goal or Process is running and the condition on the Maintain Goals becomes false, the Maintain Goal is triggered: The Maintain Goal and its sub-tree serve to make the condition true again.

A Maintain Goal is always a top Goal and can be decomposed in Plans or Achieve Goals.

The scope of a Maintain Goal is the Process by default. To define an Achieve Goal as its scope, use the Maintain Scope connector.

> **Usage Example:** The maintain condition defines the minimum amount of material that must be on stock at all times. If the amount on stock is lower than the defined amount, the condition becomes false, and the Maintain Goal is activated: Plans or sub-Goals attached to the Maintain Goal provide for restocking.

4.4.7.3.1 Maintain Goal Attributes

- **Maintain condition** is a Boolean expression checked continuously while the Maintain Goal is Ready. When the condition becomes *false*, the Maintain Goal becomes *Running*.

  > **Important:** Since pre-conditions are evaluated continuously, make sure they do not perform any assignments to prevent performance issues.

- **Visibility** defines the access rules to the Maintain Goal.

A Maintain Goal can be shown either in the decorative or iconic notation.

Figure 4.48 Maintain Goal in the iconic notation

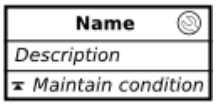


Figure 4.49 Maintain Goal shown in the decorative notation

4.4.7.3.2 Maintain Goal Life Cycle

During execution of a Goal-based Process instance, Maintain Goals go through a particular set of stages, referred to as a life cycle. At every stage, the goal is in a particular state.

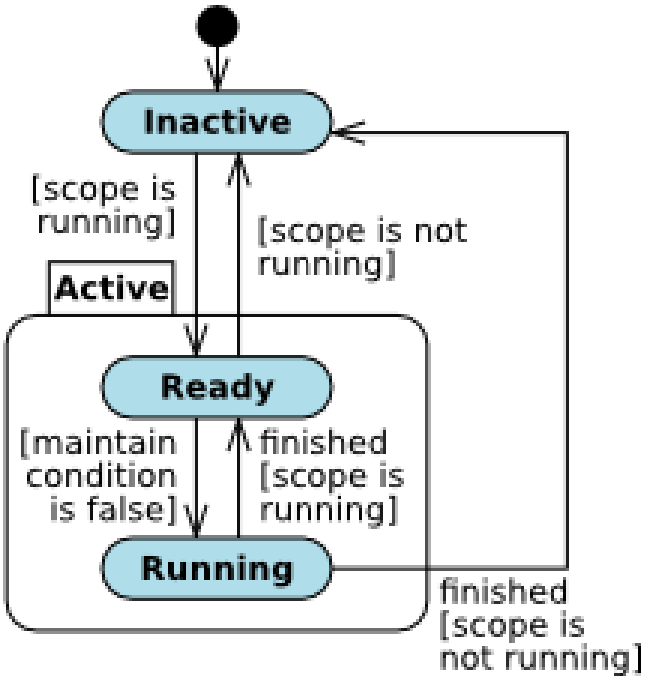


Figure 4.50 Maintain Goal life cycle without the Activation mechanism

The execution of a Maintain Goal depends primarily on its Maintain Scope. A Maintain Scope may be the entire parent Goal-based Process or a particular Achieve Goal.

A Maintain Goal is *Inactive* if its scope is not running:

- If the scope is the parent process, the maintain goal becomes immediately Ready
- If the scope is an Achieve Goal, the Maintain Goal remains Inactive while the Achieve Goal is Inactive or Ready.

When the scope becomes *Running*, the Maintain Goal becomes *Ready* and the maintain condition is checked continuously.

At the moment, when the maintain condition becomes `false`, the Maintain Goal becomes *Running*: its sub-goals become Ready or one of its Plans becomes Running. Note that the maintain condition is not checked while the Maintain Goal is Running.

When the sub-tree execution of a Maintain Goal is finished, the following can occur:

- If the scope is still Running, the Maintain Goal becomes Ready and the maintain condition is being checked again.
- If the scope is no longer Running, the Maintain Goal becomes Inactive.

If the scope is an instance of a Goal-based Process which is Finished, the Inactive state is only transient and the Maintain Goal becomes *Deactivated*.

4.4.7.3.3 Maintain Scope

A Maintain Scope flow defines the Achieve Goal scope of a Maintain Goal (the flow may connect only an Achieve Goal and a Maintain Goal and each Maintain Goal can have only one Maintain Scope. If you require a more diverse Maintain Scope of a Maintain Goal, consider defining a maintain condition.

Maintain scope is visualized as an oriented solid arrow with a short perpendicular line as an arrowhead pointing toward the Achieve Goal, which is the scope of the maintenance.

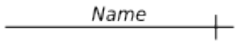


Figure 4.51 Maintain Scope

4.4.7.3.3.1 Maintain Goal with Maintain Scope

If the element of the maintain scope is:

- reactivated, the Maintain Goal becomes Ready.
- not active, the Maintain Goal is Inactive.

4.4.7.4 Plan

A Plan is an element of a Goal hierarchy which specifies what to do in order to achieve its parent Goal. It contains a Plan Model, which is an uninterrupted work flow of Events and Activities connected with Flows.

A Plan must have exactly one incoming decomposition originating from a Goal: one Plan can have only one parent Goal. It is the leaf element of a Goal hierarchy with no outgoing flow elements.

Figure 4.52 Plan in iconic notation

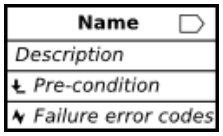


Figure 4.53 Plan in decorative notation

A Plan can define the following:

- **Pre-condition** is a Boolean condition, which is continuously checked while a Plan is Inactive.
- **Failure Error Codes** is a set of error codes (every error code being a string), which cause the Plan to fail after it has received any of the error codes: the error can be produced either by an error code end event in its body). If `null`, any error code causes the Plan to fail.

4.4.7.4.1 Plan Life Cycle

When a Process is instantiated, all the Plans are created and becomes *Inactive*. An Inactive Plan becomes *Running* when triggered by its parent Goal. At that moment, the Plan pre-condition is evaluated. If evaluated to `true`, the Plan triggers execution of its Plan Model, that is, its None Start Event produces a token and its namespace context is initiated.

> **Note:** If a Goal is decomposed in several Plans and you want the system to select a Plan based on some context data, use the pre-condition expression.

While Running, a Plan can be deactivated by its parent goal. A deactivated Plan becomes *Inactive*. A *Running* Plan becomes *Failed*, if its Plan Model ends with an Error End Event, or an error code is caught Errors.

A Plan becomes Achieved, if the execution of its Plan Model finishes with any other End Event.

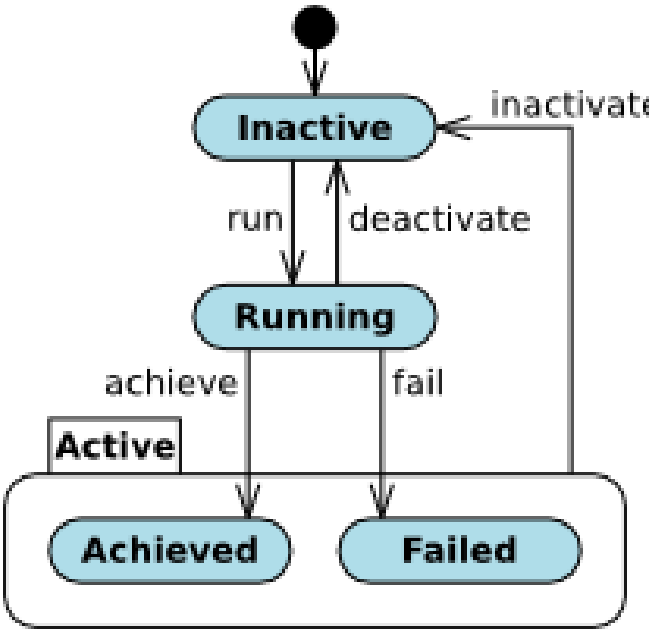


Figure 4.54 Plan life cycle

4.4.7.5 Goal Activation and Deactivation

Goal activation and deactivation is a mechanism that enables an instant change of a Goal status to either make the entire Goal sub-tree to come to a halt or to continue its execution.

You can activate and deactivate Goal either from the <../management/modelinstancemanagementpds.html↩ ::goalactivation>Management perspective of PDS or with the `activate()` and `deactivate()` functions of the Standard Library.

4.4.7.5.1 Deactivation

Goals, both Achieve and Maintain, can define a deactivate condition, which causes the Goal to become *Deactivated* when the condition becomes `true`.

The condition is checked continuously while the following is true:

- For Achieve Goals, while they are Not Finished (inactive, ready, or running).
- For Maintain Goals, while they are Alive (parent process instance is running).

When the condition becomes `true`, the Goal becomes *Deactivated* immediately. Goals can be deactivated manually as well.

When a goal is deactivated, the following happens:

1. All its sub-Goals and Plans in a top-down manner become *Deactivated*.
2. Any Running child Plans stop their execution immediately and becomes `Inactive`.

Note: An Achieve Goal or a Plan is deactivated also when their parent fails.

4.4.7.5.2 Activation

If an Achieve Goal is Achieved, Failed, or Deactivated, it is considered Finished. Only a finished Achieve Goal can be activated: on activation it becomes Ready. Note that the status of its parent Goals remains unchanged, only its sub-elements go through their life cycle.

A Maintain Goal can be deactivated at any time while it is Alive: on its deactivation, the Maintain Goal stops execution of all its sub-Goals and Plans instantly. A Maintain Goal also becomes Deactivated when the scoped Achieve Goal becomes Deactivated as well as when the parent Goal-based Process becomes Finished.

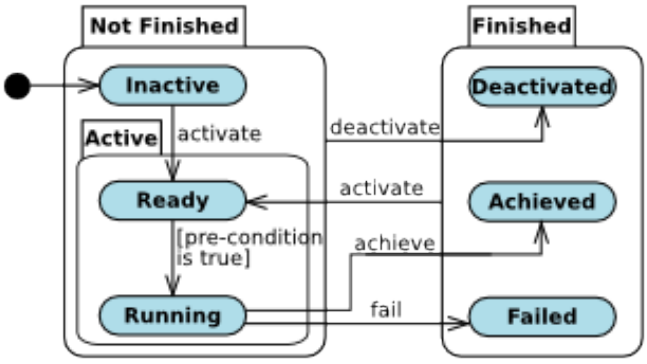


Figure 4.55 Life Cycle of Achieve Goals with Activation and Deactivation

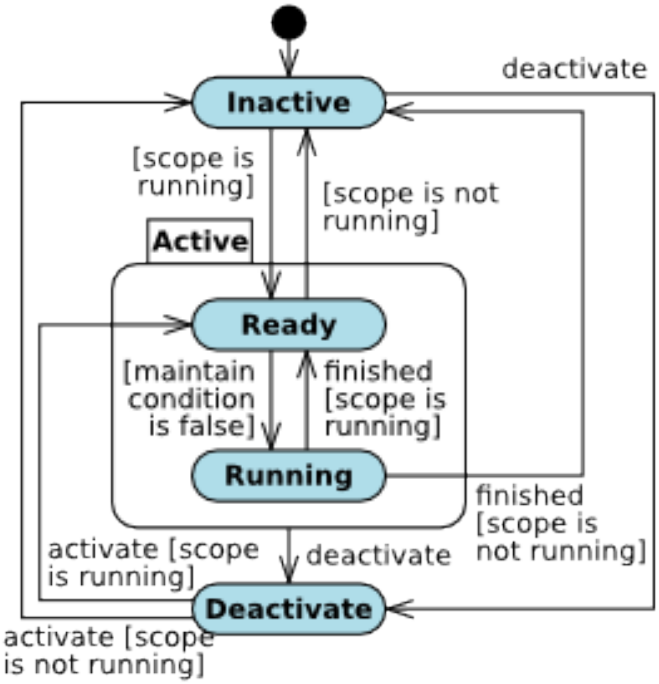


Figure 4.56 Life Cycle of Maintain Goals with Activation and Deactivation

Chapter 5

Data Type Model

A *data type model* comprises all user-defined data types called *records* along with their *relationships*. The purpose of records and their relationships is to define the data structure that accommodates the business data used in your model.

5.1 Records

A *record* represents a complex data type, such as, a person, product, service, etc.

On runtime, a record is used as a blueprint for its instances, which represent business objects: While an Invoice record defines the structure of an invoice, a particular invoice is represented by an instance of the Invoice record: the instances are created when the `new` operator is encountered; for example, you can create a person record instance of the Person record as `new Person("Doe", "John", date(1982, 1, 14))`. For further information on the behavior of operators when a record is involved, refer to the `Expression Language guide`.

The structure of a record is defined by a set of fields, for example, a record *Persona* could have the fields *surname*, *firstName*, and *dob*.

A record can inherit fields and properties from another record: it can become the subtype of a record. In such an inheritance relationship, it is frequently required that the supertype record be never instantiated: to apply this restriction, a record is defined as *abstract*. On the other hand, if a record cannot be used as a supertype, it is marked as *final*.

To prevent any changes to a record instance, a record is *read-only*. The record instance can be initialized and deleted during runtime; however, neither the record instance nor instances of its subtypes can be modified. Note that read-only records can only be targets of data relationships, not their sources.

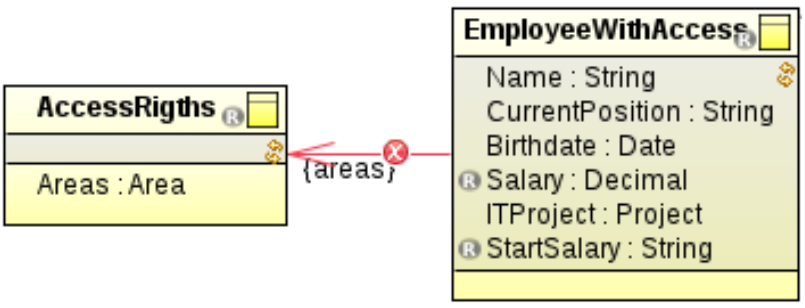


Figure 5.1 Invalid data relationship between read-only Records

If a record should not be accessible from model instances, it can be marked as a *system* record. System records cannot be instanced or modified from model instances. Such operations can be performed only by `custom objects` implemented in Java in your LSPS application code.

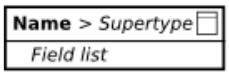


Figure 5.2 Record notation

5.2 Record Fields

A structure of a record is defined by its *record fields* with each field being of a particular data type. Simple data types should be used preferably.

5.3 Record Inheritance

Inheritance is an oriented relationship between two records in which the source Record is a more specific record than the target record, for example, the `Person` record as a supertype of the `NaturalPerson` and `Legal↩ Person` Records. The subtype record adopts all fields of the supertype record and the supertype's supertype records, etc. Hence a subtype is able to substitute its supertype in any operation valid for the supertype.

A *inheritance relationship cannot be cyclic*, for example, if type T has subtype V, the subtype V cannot have T as its subtype.

A subtype record inherits all fields of its supertypes. Hence a supertype can be used instead of its subtype. Let's assume a record 'Person' with fields `date_of_birth` and `mothertongue`. This record is the supertype of the record `Employee`. The `Employee` subtype contains the `date_of_birth` and `mothertongue` fields inherited from Person, and additional `salary` and `position` fields. Wherever the `Person` type is used, the `Employee` type can be used; however, not vice versa since the `salary` and `position` values would be missing.

If a record field is read-only, its value is set on model instantiation and cannot be changed during runtime. The setting is preserved when inherited, that is, if record A contains a read-only field A and record B is the supertype of record A, then the inherited Field remains read-only.

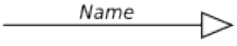


Figure 5.3 Inheritance notation

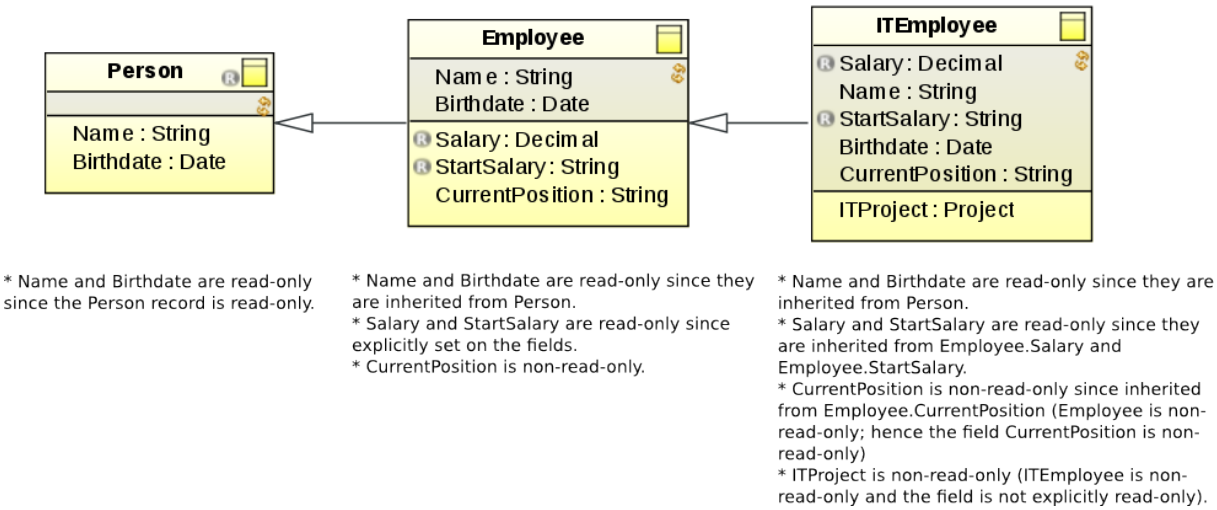


Figure 5.4 Person is the super type of the Employee and Employee is the super type of ITEmployee: All fields of Person are inherited by both subtypes and ITEmployee inherits all fields of Employee.

5.4 Record Import

The *record import* mechanism serves for importing records defined in an imported module or in another data type definition. Such imported records are referred to as record imports.

Record imports unlike records have the following limitations:

- If the record import is an import of a shared record, it can be related only to shared records:
- If the record import is an import of a shared record, any relationships of the record import can define a navigation directed toward the shared record import; However, navigation *out* of the shared record import is not supported.

  Note: These rules do not apply if the shared record of the record import is defined in another data type definition of the same module.
- No data relationship can be established between shared record imports if their parent data type definition uses a different target database.

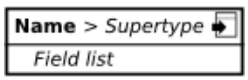


Figure 5.5 Record import notation

5.5 Data Relationships

A *data relationship* serves to establish logical connections between two records: the relationship defines the properties of navigation to either ends. Though one end is designated as the Target and the other as the Source end of the relationship, the relationship works both ways equally and properties of the navigations to either end are defined for both ends (they are symmetrical).

A relationship can be cyclic; that is, the source and target can be the same record. For example, an employee might need to be in a relationship with another employee where both are represented by instances of the same Employee record.

Note: Read-only records can only be targets of data relationships, but not their sources. This prevents a possible inconsistency of data.

Data relationships define the following:

Navigability *Navigability* of a data relationship end enables you to "move" to the related record. To establish navigability, the respective relationship end must define its name: every data relationship must define at least one of its ends' name.

Example: The records Author and Book are connected with a relationship. The end pointing to the Book is named "books" so you can navigate from the Author record to the Book record. Hence when you define a variable of the Author type, you can define as part of the definition also the Author's book. The relationship end pointing to the Author does not have a name. Therefore, you cannot create a Book *with its Author*.

Multiplicity Multiplicity of a relationship and defines how many record instances can be at the end of the relationship:

- **Single**: only one record instance of the record can be on this end of the relationship.

  For example, let's assume *Sport Shoes* and *Production Batch* records: a pair of shoes is produced as part of *only one* batch; hence the relationship from the Sport Shoes to the Production Batch has Single multiplicity.

- **Set**: multiple *different* record instances can be on this end of the relationship.

  In the example, this would be the multiplicity on the relationship end pointing to the Sport Shoes.

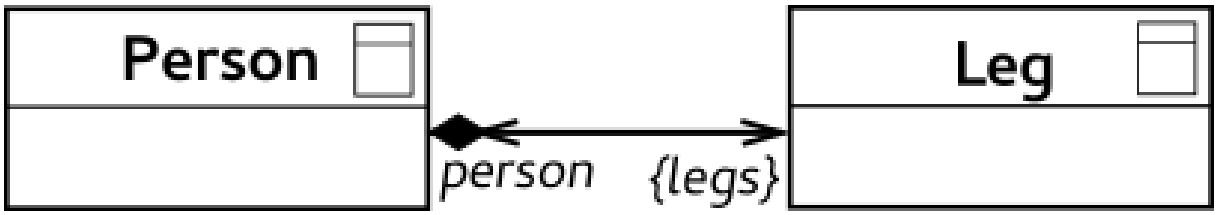
  If the relationship connects two *shared* records, the Set multiplicity can define the **Order By** property. The property defines the database column that is used to order the related record instances. If no value is specified, the instances are ordered according to the primary key. For example, if the shared record Person has a navigable Set relation to the shared record Citizenship and defines the Order by property as `countryCode`, then the *Citizenship* record instances fetched as related record instances by the expression `Person.citizenship` will the ordered according to the countryCode.

- **List**: multiple record instances can be on this end of the relationship

  Note: If one end defines the *List* multiplicity and the other end a Single multiplicity, then for every item of the List exactly one item in the other end is available. Such a relationship does not handle situations where one entity is available multiple times in the list. In this case, a join table is needed.

Composition A composition is a "target-is-part-of-source" relationship: the value at the target end cannot exist by itself, that is, without a source value. If the source value is removed, all its target values are removed.

The source end of a composition relationship must be of the *single* multiplicity, while the target end can be of any multiplicity: *when the value at the source end is deleted, the values at the target end are deleted as well*–cascade delete takes place.



Based on the example, when you delete a Person, all its Legs are deleted; when you create a Leg, it must have a relationship to a Person. If you delete a Person's leg, the Person is not deleted. Note that a leg can belong to only person only.

5.5.1 Deleting Record Instances in a Data Relationship

The record instances in a relationship are deleted depending on the relationship multiplicity as follows:

- On a relationship with **single multiplicity**:
  - if the source instance is deleted, the entire instance takes the value `null`.

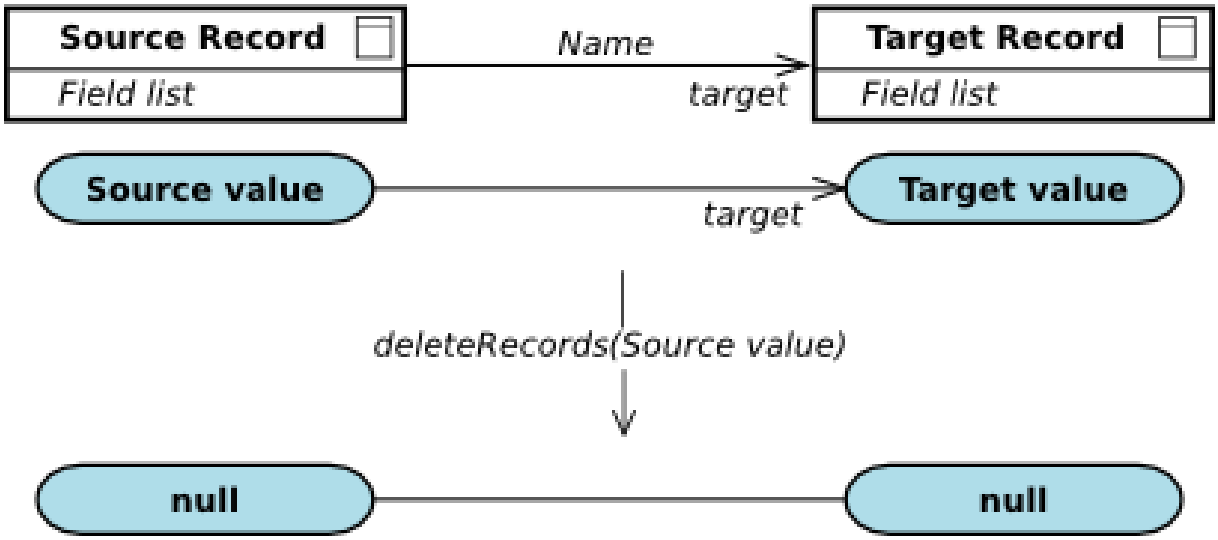


Figure 5.6 Deleting a Source Record instance

  - if the target instance is deleted, the target instance takes the value `null`.

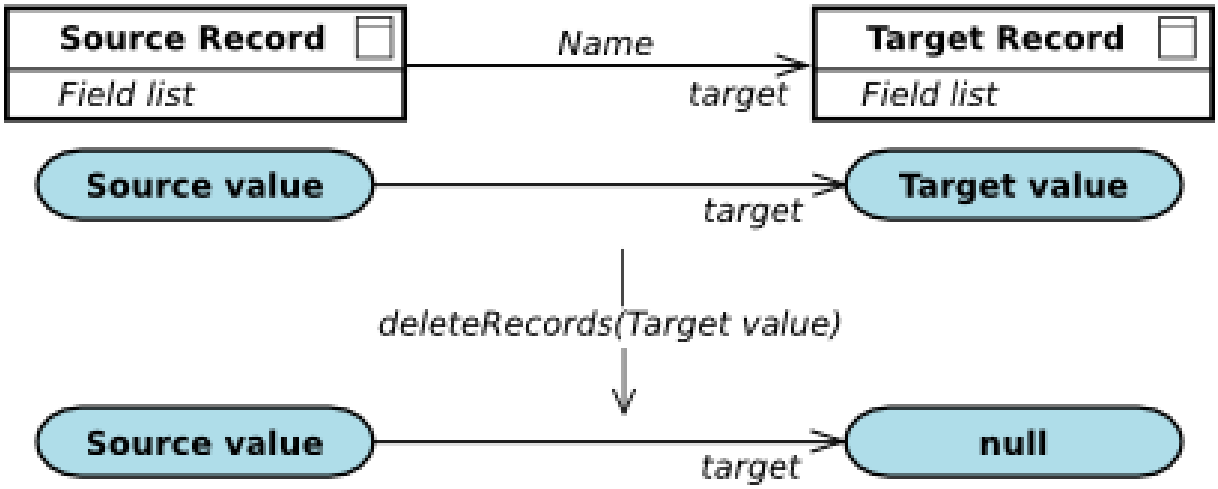


Figure 5.7 Deleting Target Record instance

If you want to delete the source instance as well, set the relationship end pointing to the source record as composition.

- On a relationship with **set or list multiplicity**, the deleted record instance of the set or list is removed from the relationship (the list or set record Instance with the null value is removed).

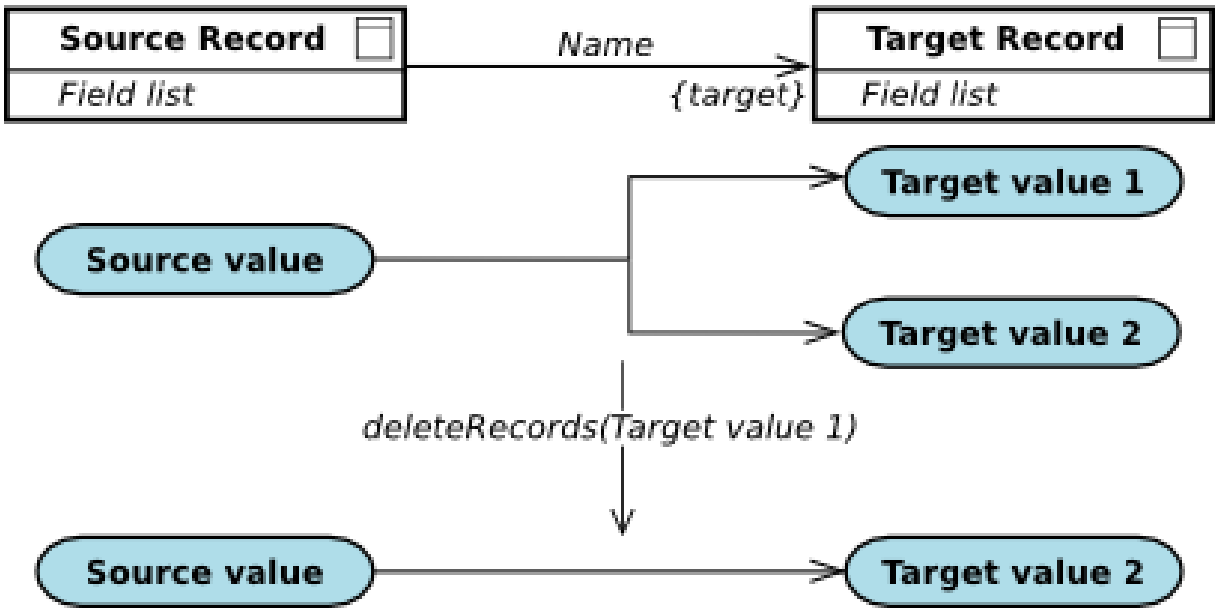


Figure 5.8 Deleting a Target Record instance in the Set Relationship

5.6 Shared Records

Shared records serve to persist data: Instances of shared records are persisted in a database, unlike instances of common records and hence survive their context. Any readings, modifications, and deletions of shared record instances are immediately reflected in the mapped database entry.

A shared record is mapped to a database table and its fields are mapped as the table columns. Note that if a fields of a shared record is of a different type than the Boolean, Integer, Decimal, String or Date type, you need to define the BLOB size so that a sufficient space is reserved in the underlying database column for the data (for information on LSPS implementation of shared records and the related mapping and fetching mechanisms refer to Process Design Suite User Guide.

Figure 5.9 Shared Record notation

5.7 Enumerations

Enumeration is a special data type that holds literals. The literals represent the possible literal values of the enumeration object and are called in the form `<enumeration_name>.<literal_name>`. The comparison operators =, !=, <, >, <=, >= can be used on the literals of the same enumeration type. Since the literals are arranged as a list of values, comparing enumeration literals is based on comparing their indexes: The order depends on the order of the literal as modeled in the enumeration.

Enumerations don't engage in relationships: they cannot be targets or sources of inheritance or data relations.

In addition to the common modeling element attributes, an enumeration defines the Deprecated flag, which signalizes that the enumeration will be removed in the next version of the model or application. In Living Systems® Process Design Suite, if the attribute is true, the validation displays an information message that the enumeration is deprecated.

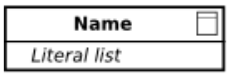


Figure 5.10 Enumeration notation

Chapter 6

Organization Model

An *organization model* serves to acquire a group of persons (users) that meet some requirements.

The model defines a hierarchy of organization elements, which group persons, the users of the application. The model can contain the following:

- A Role represents a group of persons with common expertise.
- A Unit is an umbrella element for one or multiple roles.
- Decompositions establish relationships between roles and units.

On runtime, a runtime version of the role can be assigned to a person: the person then belongs to the role and any ancestor role or organization unit. A runtime role can define parameters with values. Parameters of any ancestor can be used. If an ancestor has a parameter with the same name as its descendant, the parameter on the descendant is considered the same parameters as the parameter on the ancestor.

To acquire persons in a role or organization unit, call the unit or role as `<ROLE_UNIT>(<MAP_OF_PARAMETERS>)`. You can use the functions in the `human module of the Standard Library` for additional features.

6.1 Organization Roles

An *organization role* groups persons with common behavior, rights, expertise, etc.

One role is usually assigned to multiple persons and one person can have multiple roles. It can be decomposed to other roles and be a target of a decomposition, descendant of a role or organization unit: a person with a role belongs to all ancestor roles and organization units.

Roles can define parameters, which allow you to exclude persons who are in the role or its descendant role but do not have the specified parameter with the specified value.

Figure 6.1 Iconic role notation

Figure 6.2 Decoration Role notation

6.2 Organization Unit

An *organization unit* is an umbrella element for roles: A person is part of an organization unit if they belong to a role that is a descendant of the organization unit.

Note that users are added only to roles, not organization units.

It can be decomposed to roles or other organization units and be a target of a decomposition originating from an organization unit: a person with a role belongs to all ancestor roles and organization units.

Organization units can define parameters which allow you to exclude persons who are in a descendant role of the unit but do not have the specified parameter with the specified value.

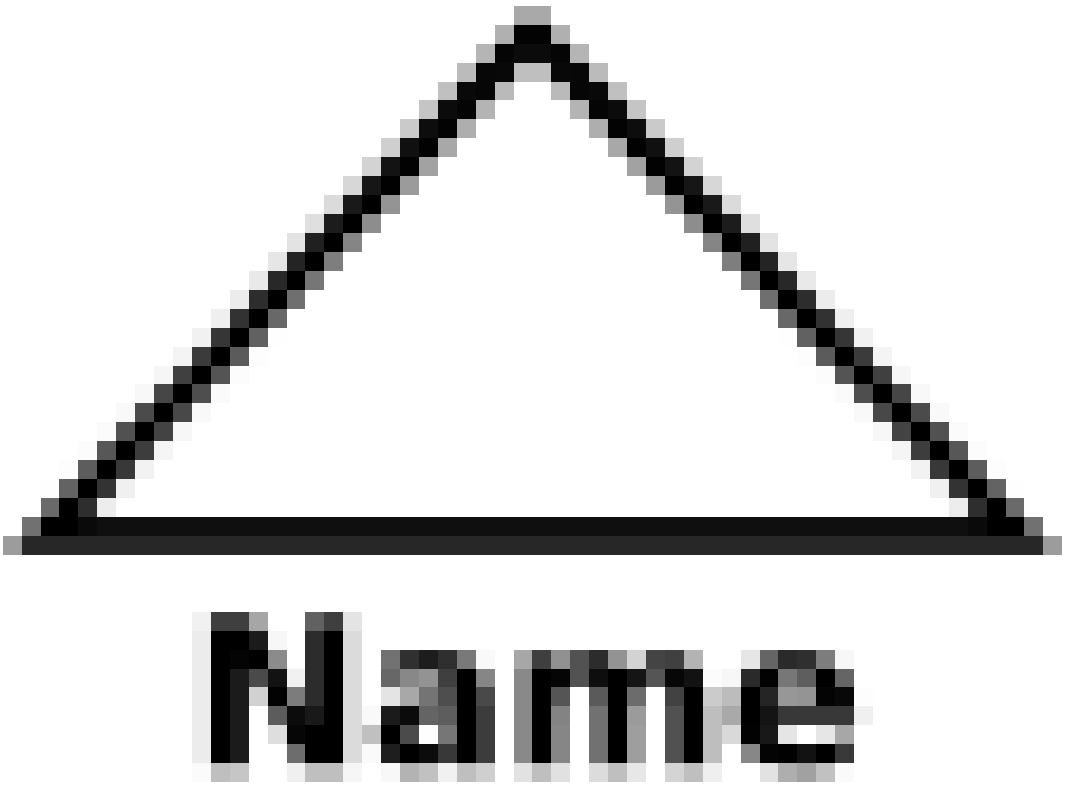


Figure 6.3 Iconic Unit notation

Figure 6.4 Decoration Unit notation

6.3 Decomposition in Organization Models

Note: GO-BPMN uses two decompositions: decompositions in goal structure and decomposition in organization models. The latter is documented below.

Organization units and roles can be decomposed with the decomposition relationship in one or several other organization elements: This enables you to create organization hierarchies and group organization elements; a person who belongs to a role belongs also to all its ancestor roles or role units.

One organization element can be the target or origin of multiple decompositions. However, decompositions cannot create cyclic relationships.

Decomposition can be used between the following organization elements:

- Unit-to-Unit decomposition: breakdown of an Organization Unit into a sub-Unit;
- Unit-to-Role decomposition: including a Role in the Unit;
- Role-to-Role decomposition: child Role representing a more specialized Role;

Note: The Role-to-Unit decomposition is not supported.

img{Decomposition.png, Decomposition notation, 90px}

6.4 Resolving Roles and Units to Persons

When you request persons with a role, it is resolved as follows:

- Without a parameter value or with a parameter value *null* or "∗", all persons with the role or its descendant roles are returned (role parameters are ignored).
- with a parameter value, it returns all persons with the role or its descendants which have the parameter with the specified parameter value or do not have the parameter (if the parameter has a different value, the person is excluded).

When you request persons of an organization unit:

- without a parameter value or with the parameter value *null* or "∗", it returns all persons with a descendants of the organization unit (parameters are ignored)
- with a parameter value, it returns all persons with a descendants of the organization unit which have the parameter with the specified value or do not have the parameter (if the parameter has a different value, the person is excluded).

Hence, parameters of the role units, organization units and roles provide a filtering mechanism over persons that belong to a role unit.

Example: Consider the following organization model:

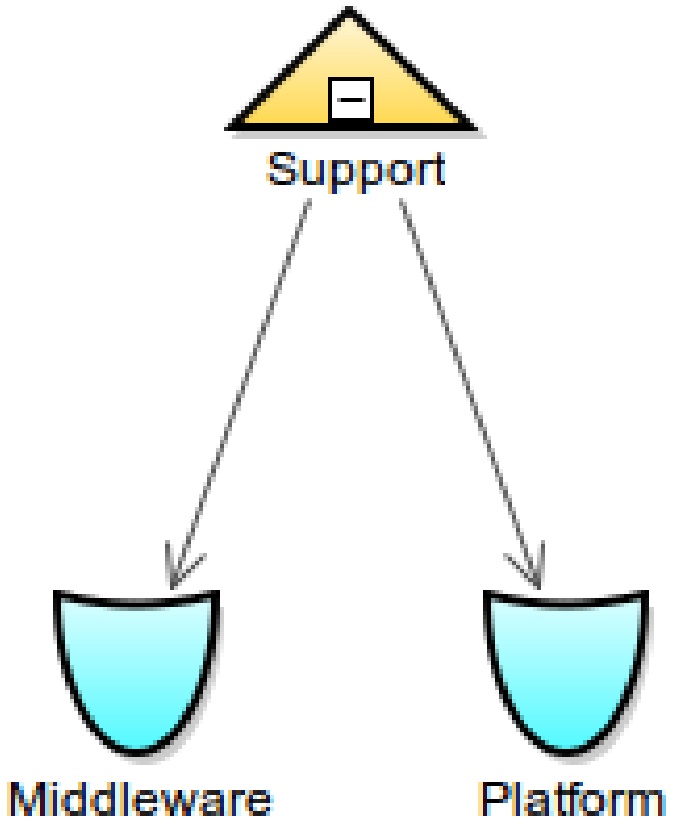


Figure 6.5 Unit decomposed in two Roles

- If a task is assigned to the Organization Unit Support (for example, `performers -> {Support([->])}`), it is assigned to all persons that have the *Middleware* or *Platform* Runtime Roles regardless of their parameter values.

- If a task is assigned to the Organization Unit Support with a parameter (for example, `performers -> {Support(["product"->"Wildfly"])}`), it is assigned to all persons that have the *Middleware* or *Platform* Runtime Roles with no parameter or with the parameter `product` set to `Wildfly`.

Example: The role `TechnicalExpert` defines the parameter `expertise`.

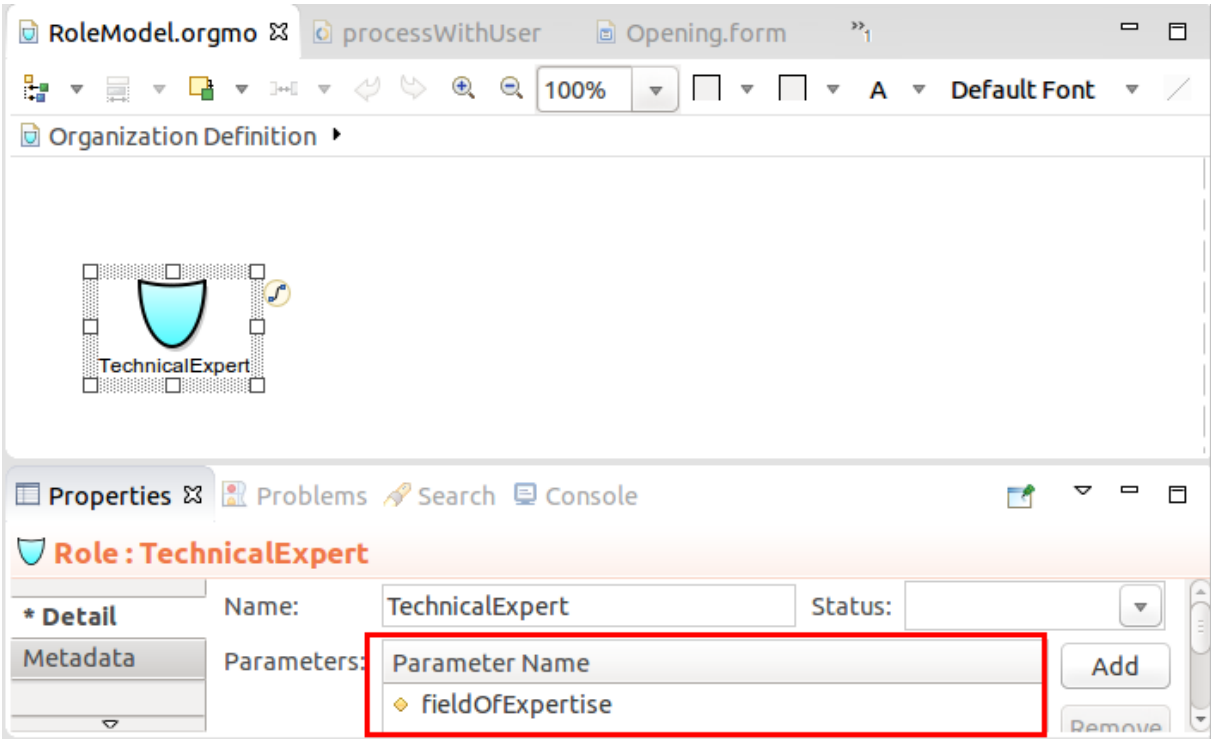


Figure 6.6 Parametric Role definition

The person *Eva* is a technical expert without a *field of expertise*: she is considered an expert in all fields and has the runtime role `TechnicalExpert` with no parameter.

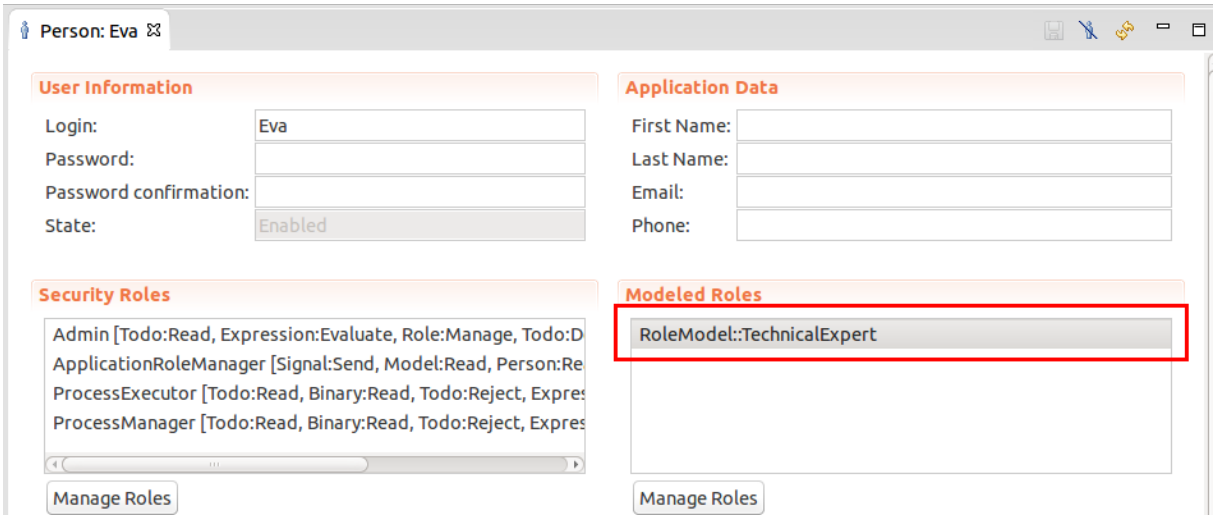


Figure 6.7 Person view of a person with the runtimee role TechnicalExpert with no parametric value

John and James are technical experts specialized in hydraulics and electrics: They have the runtime roles `TechnicalExpert` with the `expertise` parameter set to `hydraulics` and `electrics`.

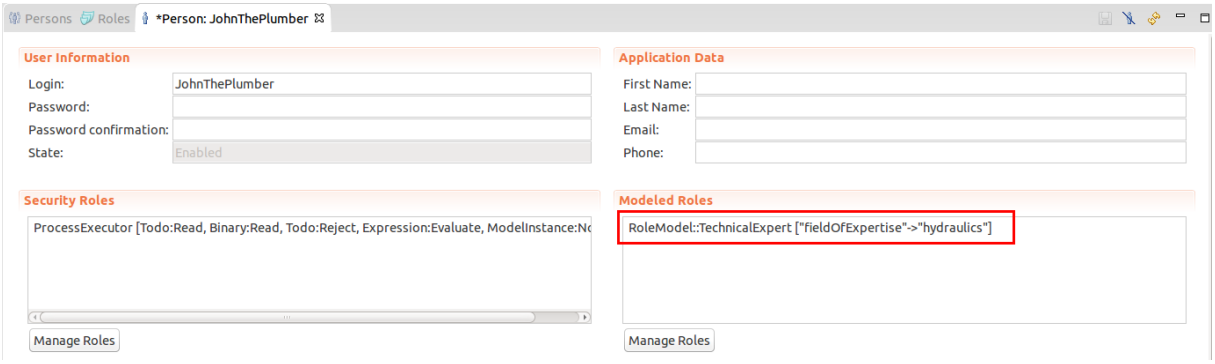


Figure 6.8 Person view of a person with the runtime role TechnicalExpert with the 'hydralics' expertise parameter

- If a task is assigned to *TechnicalExpert([->])*, the task is assigned to
  - all three persons regardless of the parameter values (any of 'TechnicalExpert' can perform the task);
- If a task is assigned to a `TechnicalExpert("expertise"->"hydraulics"`, the task is assigned to:
  - Eva, the technical expert with no parameter value,
  - John, the technical expert with the `hydraulics` parameter value James is left out.

6.5 Organization Element Import

For presentation purposes, you can add the views of the Roles and Unit from other organization definitions to the diagram in your definition as Role and Unit Imports. Note that such imported Role or Unit views cannot be decomposed, however, they can be a target of a Decomposition.

Note: Roles and Unit from other Modules become available for Diagram import only after their Module is imported.

Chapter 7

Diagrams

Diagrams provide space for a graphical representation of elements of a particular type in a Module: Visual representation (an element view) of one modeling element can appear several times in one or several Diagrams while still referencing the same single element. Note that a modeling element can be shown in one Diagram only once.

To add additional information for the reader of a Diagram, you can use diagram elements: these are the only elements that actually exist only within the Diagram.

A diagram can depict only elements from the same type of definition:

- Goal Diagrams depict Goals, Plans, and their Decompositions;
- Plan Diagrams depict modeling elements of a Plan (events, activities, flow objects, etc.);
- BPMN Diagrams depict modeling elements of a BPMN-based Process (events, activities, flow object, etc.);
- Data Type Diagrams depict Data Types and their subtype relations;
- Organization Diagrams depict views of Organization Units, Roles, and their Decompositions.

7.1 Goal Diagram

A Goal Diagram is a Diagram depicting an arbitrary number of modeling element of a goal model. It may contain Goals views, Plans views, and vies of their Decompositions, plus allowed diagram elements (Text Annotations, Associations).

Owned by Goal-based Process it depicts only views of Goals and Plans of the respective Process. A Goal-based Process may contain an arbitrary number of Goal Diagrams.

7.2 Plan Diagram

A Plan Diagram is a Diagram depicting element views of a plan body.

It may show views of one or several modeling elements of a plan body, and objects owned by the Diagram (Annotations and Associations).

Plan Diagrams are owned by Plans. One Plan may contain an arbitrary number of Plan Diagrams. Plan Diagrams can show view of any modeling element contained in the respective Plan Body, however, a modeling element can be shown in one Plan Diagram only once.

7.3 Process Diagram

A Process Diagram is a Diagram depicting element views of a body of a BPMN-based Process.

It may show views of one or several modeling elements of a body of a BPMN-based Process, and objects owned by the Diagram.

7.4 Organization Diagram

An Organization Diagram is a visual representation of a part or of an entire Organization Model.

Organization diagrams are owned by organization models. One model may contain one or several organization diagrams. Organization diagrams can show any organization element contained in an organization model, however, in one diagram, every element can be shown only once.

An organization diagram may contain diagram elements (Text Annotations, Associations).

7.5 Data Type Diagram

A *Data Type Diagram* is a diagram depicting element views of Record types.

Data Type Diagrams are owned by Modules and may contain views representing `Record types` and related entities: imported Record types, inheritance relationships of Records, and general diagram elements (Text Annotations, Associations).

A Module may own an arbitrary number of Data Type Diagrams. View of one data type element (record, record import, inheritance) may be shown in an arbitrary number of Data Diagrams.

7.6 Diagram Elements

Diagrams can *contain* diagram elements, which serve for documentation purposes, and have no execution semantics (they do not influence execution in any way). As such they are not considered elements of the definition but belong to the diagram only.

7.6.1 Diagram Frames

Diagram frames are diagram elements, which allow you to display the content of another diagram as read-only. The content of the diagram frame reflects the current state of the referenced diagram.

Diagram frame can display only a diagram of the same type as the diagram; for example, you cannot insert a goal diagram frame to an organization diagram. A diagram frame cannot reference itself.

> **Tip:** Use directed associations and annotations to establish logical links between an element shown in a diagram frame or the entire diagram frame and other elements in the diagram.

7.6.2 Hyperlinks

Hyperlinks are diagram elements that provide direct links to a location, resource, or element: When you click a hyperlink, the linked entity is displayed.

There are four types of hyperlinks:

- Diagram hyperlink: link to another diagram;
- URL hyperlink: link to a URL;
- Element hyperlink: link to an element of the project;
- Resource hyperlink: link to a module resource of the project.

7.6.3 Text Annotations

A *Text Annotation* is a diagram element containing free-text information.

It does not influence execution: it is en element with no execution semantics. A Text Annotation provides additional information and has only an informative character. It belongs to the particular diagram.

It may be connected to one of several diagram elements using Associations. If left unconnected, a Text Annotation is intended to provide information about the entire Diagram.

Figure 7.1 Text Annotation notation

7.6.4 Associations

An association is an element without a semantic value used for informative linking of element views and diagram elements.

It may be assigned a particular direction to indicate a relationship orientation (Directed Association).

Figure 7.2 Association notation

Association attributes:

- **Direction** defines the arrow direction.