

## 4.7 Business Process Model and Notation

This section introduces the *Business Process Model and Notation (BPMN)*, developed under the coordination of the Object Management Group. Version 2 of this international standard introduces a series of modifications, including a new extension of the acronym. BPMN used to stand for *Business Process Modeling Notation*. In Version 2, the standard also defines a meta-model, so that *Business Process Meta Model and Notation* would have been a valid choice. Unfortunately, the term *meta* was dropped, resulting in the rather imprecise official extension we now see in this section's heading. In the remainder of this book, we will mostly use the acronym.

The intent of the BPMN for business process modelling is very similar to the intent of the Unified Modeling Language for object-oriented design and analysis. To identify the best practices of existing approaches and to combine them into a new, widely accepted language. The set of ancestors of BPMN includes graph-based and Petri-net-based process modelling languages, such as UML activity diagrams and event-driven process chains.

While these modelling languages focus on different levels of abstraction, ranging from a business level to a more technical level, the BPMN aims at supporting the complete range of abstraction levels, from a business level to a technical implementation level. This goal is also laid out in the standards document, which states that “The primary goal of BPMN is to provide a notation that is readily understandable by all business users, from the business analysts that create the initial drafts of the processes, to the technical developers responsible for implementing the technology that will perform those processes, and finally, to the business people who will manage and monitor those processes. Thus, BPMN creates a standardized bridge for the gap between the business process design and process implementation.”

The BPMN defines several diagram types for specifying both process orchestrations and process choreographies. Since this chapter focuses on orchestrations, only business process diagrams and collaboration diagrams are discussed in this section. Diagram types regarding process choreographies, that is, conversation diagrams and choreography diagrams, will be discussed in the next chapter.

To classify the level of support that a particular BPMN software tool provides, the standard introduces so called conformance classes.

- *Process Modeling Conformance*: The process modeling conformance class includes the BPMN core elements, process diagrams, collaboration diagrams and conversation diagrams. Subclasses are defined that contain a limited set of visual modelling elements (*Descriptive* subclass), an extended set of modelling elements (*Analytical* subclass) and modelling elements that are required to represent executable processes (*Common Executable* subclass), respectively.
- *Process Execution Conformance*: The process execution conformance class requires a software tool to support the operational semantics of BPMN. If, in addition, the mapping from BPMN to WS-BPEL as defined in the standard is implemented, the tool satisfies *WS-BPEL Process Execution Conformance*. WS-BPEL and the mapping from BPMN to this XML language is addressed in Chapter 7.
- *Choreography Modeling Conformance*: The choreography modeling conformance class includes the BPMN core elements, collaboration and choreography diagrams. Choreography modelling will be discussed in Chapter 5.

#### 4.7.1 Principles

The BPMN standard defines a notation and a meta model that organizes the concepts used in the notation. While much more complex, the BPMN meta model is similar to the meta model discussed in Section 3.5. To avoid redundancy and to provide a solid basis, the standard is organized in layers.

The BPMN Core Structure is the foundation of the standard, which defines generic concepts like *BaseElement*, which is the abstract super class for most BPMN elements. These concepts are refined subsequently in packages related to processes, choreographies, collaborations, and conversations. The reader interested in the BPMN meta model is referred to the BPMN standard, referenced in the bibliographical notes at the end of this chapter. This text concentrates on the language constructs and their execution semantics rather than on the organization of the standard.

The basic BPMN modelling elements allow expressing simple structures in business processes, while expressive power is added by the complete element set. The basic elements are easy to comprehend, so that process designers and practitioners can use the language without extensive training. When process designers become familiar with the language, more elaborate language elements can be added.

The graphical notation of a business process is complemented with a set of attributes. These attributes can be associated with the complete process diagram and with particular elements. Some attribute values have implications on the visual appearance of the symbols used in process diagrams. For instance, whenever a gateway activates a single outgoing edge from a set of outgoing edges, the gateway is marked with the X symbol to indicate its *exclusive or split* semantics.

The BPMN has the flavour of a framework rather than of a concrete language, because some aspects, for instance, expressions, are not covered by the standard, and left to the process designer. Expressions are used, for example, to decide which branch to follow in the case of an *exclusive or split*. During business process modelling projects, the persons responsible can use a language of their choice. However, within one business process diagram, only one expression language can be used.

In case a high-level business process is modelled, informal textual expressions might be useful. An example would be “if the credit amount exceeds 5000 Euros, then the monthly income of the client needs to be checked”. In this case, the language to formulate expressions would be plain English text. If business processes need to be represented at a technical implementation level, formal languages with an operational semantics, such as programming languages, are required.

Organizational aspects are represented in the BPMN by pools and swimlanes, similar to those in UML activity diagrams. There is a hierarchy of swimlanes within a given pool: lanes, and arbitrarily nested sub-lanes. Lanes represent organizational entities such as departments in organizations. Sub-lanes can be used to define organizational entities within departments. Nesting of arbitrary depth is permitted, but process diagrams might get cluttered in case of extensive nesting. By drawing flow objects in swimlanes, the organizational entity responsible for performing the specific objects can be represented graphically.

Each pool may specify a concrete organization, but it may also represent a placeholder for a specific organization, that is, a role. Examples of roles in a supply chain scenario are “supplier”, “manufacturer”, and “customer”. When it comes to enacting business processes, concrete organizations are bound to these roles, so that one concrete supplier interacts with a specific manufacturer, which interacts with a specific set of customers.

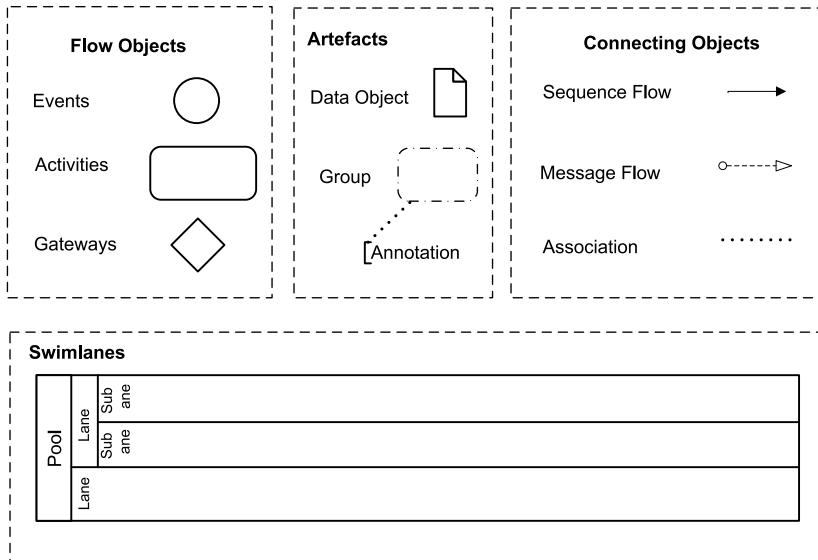
Each process resides in a single pool. As a consequence, each process is performed by a single organization. Business processes can interact with business processes enacted by other organizations in order to realize business-to-business scenarios. These assumptions of the BPMN regarding pools and the scope of business processes are in line with Definition 1.1.

#### 4.7.2 Business Process Diagrams

The notational elements in business process diagrams are divided into four basic categories, each of which consists of a set of elements, shown in Figure 4.76.

Flow objects are the building blocks of business processes; they include events, activities, and gateways. The occurrence of states in the real world that are relevant for business processes and, more generally, anything relevant that happens, can be represented by events. Activities represent units of work performed during business processes. Gateways are used to represent

the split and join behaviour of the flow of control between activities, events, and gateways.



**Fig. 4.76.** BPMN: categories of elements

Artefacts are used to show additional information about a business process that is “not directly relevant for sequence flow or message flow of the process”, as the standard mentions. Data objects, groups, and annotations are supported artefacts. Each artefact can be associated with flow elements. Artefacts serve only information purposes, so that the execution semantics of a process is not influenced by them.

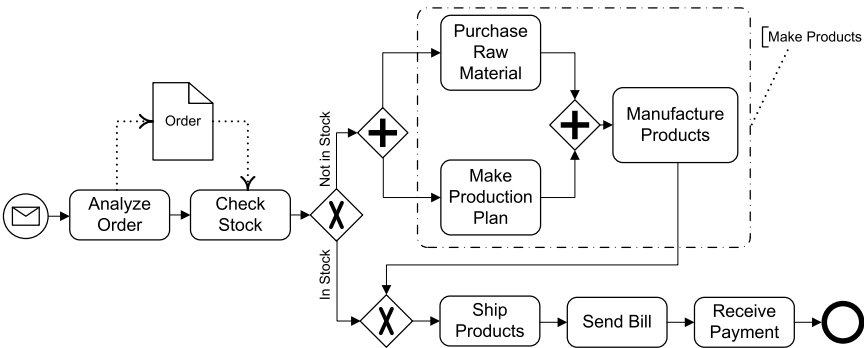
Data objects are represented simply by a name; the internal structure of data objects cannot be defined in BPMN. The main purpose of data object artefacts is documentation of the data used in the process. By directed association edges, the modeller can represent the fact that a data object is read or written by a process activity. Paper documents, electronic information, as well as physical artefacts, like shipped products, can be represented by data objects.

Text annotations document specific aspects of the business process in textual form. The text is graphically associated with the object in the business process diagram that the text explains. Group objects are artefacts that are used to group elements of a process. Groups do not have a formal meaning; they just serve documentation purposes. Groups may span lanes and even pools.

Connecting objects connect flow objects, swimlanes, or artefacts. Sequence flow is used to specify the ordering of flow objects, while message flow describes

the flow of messages between business partners represented by pools. Association is a specific type of connecting object that is used to link artefacts to elements in business process diagrams.

Figure 4.77 shows a BPMN business process diagram, representing an ordering process. The example introduces the main elements of the language: events, activities, gateways, and sequence flow. The process model starts with an event. A sequence of activities to analyze the order and to check the stock are performed, before an *exclusive or split* is done. The latter is represented by a gateway with the respective marker.



**Fig. 4.77.** Business process diagram expressed in BPMN

If the ordered products are in stock, then the lower branch is selected. Otherwise the product has to be manufactured first, so that the upper branch needs to be chosen. The expression language used in this process diagram is plain English text (*In stock*, *Not in Stock*), so that humans can easily understand the conditions. The manufacturing part of the process can be seen as a detour, since both branches converge in the exclusive join gateway before the products are shipped, the bill is sent, and the payment is received.

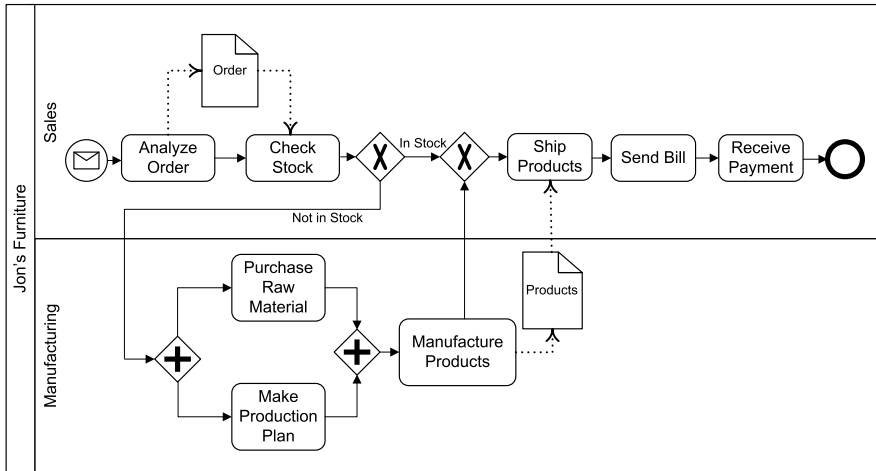
This process diagram also contains events that mark the start and end of the process. The start event is marked with an envelope symbol, indicating that the process starts on receiving a message. There are many different event types and markers for events, the most widely used of which will be discussed shortly.

Data in processes plays an increasingly important role. The example represents the order processed as a data object. Data objects can be associated with flow elements, indicating a relationship. In the example shown in Figure 4.77, there is a data object *Order*, which is associated to activities *Analyze Order* and *Check Stock*. The orientation of the association edge indicates the type of relationship. In our process diagram, *Analyze Order* writes the data object, while *Check Stock* reads it.

Grouping of activities using the group artefact can increase the understanding of the process model by humans. This is exemplified by a group an-

notated by *Make Products*. In this example it is rather obvious that the three activities are responsible for making the product in case the ordered products are not in stock. However, in larger process models grouping is a convenient way of expressing additional information for humans that can not conveniently be provided by the more formal modelling elements of the BPMN.

The roles involved in this process have not been represented in the process diagram. If the roles are important for the modelling purpose, for example, if responsibilities in the organization have to be defined or hand-overs between departments need to be investigated, roles must be represented in process diagrams. Figure 4.78 shows a process diagram of the ordering process introduced above, enriched with role information.



**Fig. 4.78.** Business process diagram with role information

There are two departments of the company modelled, *Manufacturing* and *Sales*. Receiving and analyzing the order as well as checking the stock and deciding about manufacturing the products is also decided by the sales department. Obviously, producing the ordered items is performed by the manufacturing department. Since hand-over between organizational entities is important, the model also contains a data object *Product*. This illustrates that also physical products can be represented by data objects in BPMN. In this case, the write edge from *Manufacture Products* to *Products* can be interpreted as the production of the physical goods. The read edge from *Products* to *Ship Products* refers to the use of the physical products during the shipment activity.

Activities

Activities are units of work. They are the major ingredients of business processes. The BPMN provides powerful means for expressing different types of activities. Figure 4.79 shows the activity types that the BPMN supports.

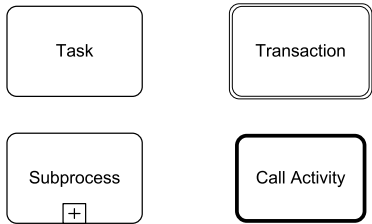


Fig. 4.79. Activity types in the BPMN

Activities characterize units of work. Activities which are not further refined are called atomic activities or tasks. Activities might also have an internal structure, in which case they are called subprocesses. Rather than showing the structure, the modeller can decide to hide the complexity of the subprocess, using the plus symbol. But subprocesses can also be expanded, exposing their internal structure.

An example of a subprocess is shown in Figure 4.80. In that figure, the collapsed subprocess is marked with the subprocess marker, and the expanded subprocess exhibits its internal process structure. The link between the representations is established by the unique identifier *Evaluate Credit Risk*.

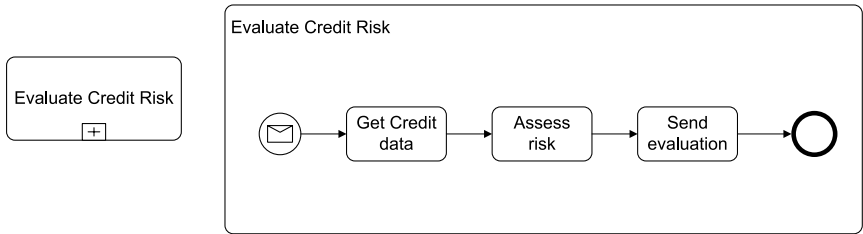
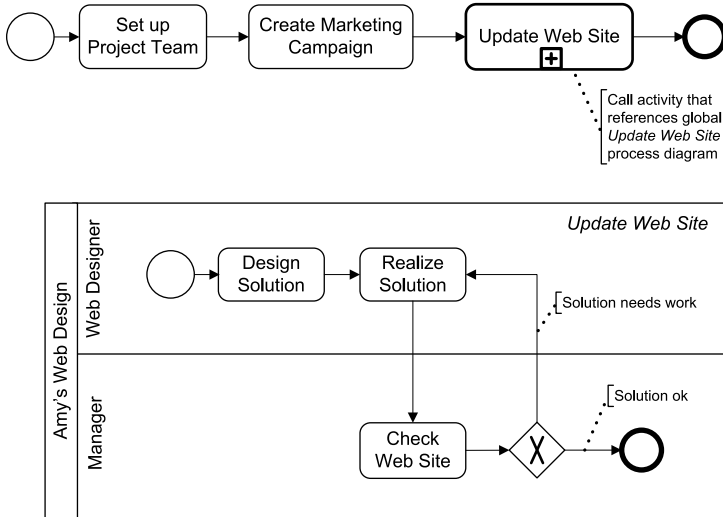


Fig. 4.80. Collapsed and expanded subprocess

Call activities can be used to refer to globally defined process diagrams, or tasks, facilitating reuse of activities. An example of a call activity involving a globally defined process diagram is shown in Figure 4.81. In the upper part of that figure, a simple process containing a sequence of activities is shown. The first activity is an embedded subprocess with activities to set up a project team and to create a marketing campaign. After these activities have been

completed, the embedded subprocess terminates. Then the *Update Web Site* activity is performed.

This call activity references the global process diagram shown in the lower part of that figure, reusing it. This design allows to define certain processes or tasks once to be used several times. In the example, each update of the web site could be realized by a call activity, reducing maintenance effort in large process repositories.



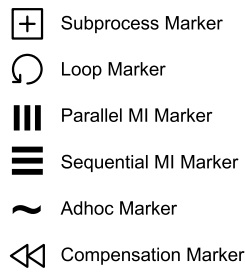
**Fig. 4.81.** Process diagram with a call activity that references a global process diagram; the reference is maintained in the respective attribute of the call activity

Activities can be marked with symbols that refine their execution semantics; activity markers are shown in Figure 4.82. We have already used the subprocess marker. Notice that transactions will be discussed later in this section after the required events have been introduced.

The loop marker is used to indicate that an activity is iterated during process execution. If the activity has the `LoopCharacteristics` attribute set, with attribute class `StandardLoopCharacteristics`, then the activity represents a *while* loop or a *repeat-until* loop. Whether the loop activity realizes a *while* loop or a *repeat-until* loop is guided by the `testTime` attribute. Setting it to `Before` realizes a *while* loop, while setting it to `After` realizes a *repeat-until* loop. The different types of loops can not be distinguished by the visual appearances of the respective loop activities in process models.

Multiple instances tasks have the `LoopCharacteristics` attribute set, with attribute class `MultiInstanceLoopCharacteristics`. The multiple instances of an activity can be executed sequentially or in parallel. The number

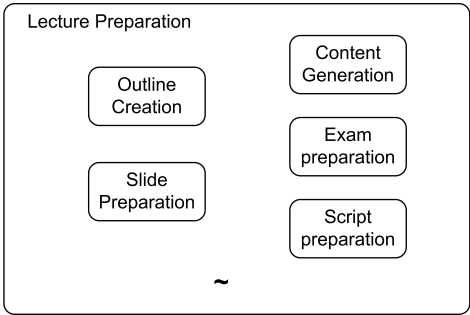




**Fig. 4.82.** Activity markers refine the behaviour of activities

of instances is either specified by an expression that returns an integer value or by the cardinality of a list data object, discussed below.

The markers for sequential and parallel multiple instances activities are shown Figure 4.82. A *for* loop with  $n$  iterations can be realized by a sequential multiple instances activity, whose expression evaluates to  $n$ .



**Fig. 4.83.** Sample adhoc process

A subprocess that is marked with an adhoc marker consists of a set of tasks that are not related to each other by sequence flow. The execution of tasks of the adhoc subprocess is not restricted. Each activity can be executed an arbitrary number of times. This means that adhoc activities are not embedded in sequence flow; they can be invoked without a specific trigger or event.

An adhoc subprocess is marked with a tilde symbol at the bottom of the rounded rectangle. Adhoc activities are very useful for unstructured parts of processes. Using `AdHocOrdering`, the modeller can define whether the activities in an adhoc subprocess can be executed in parallel or whether they are executed sequentially. An adhoc subprocess completes, if its `CompletionCondition` evaluates to true. An example of an adhoc subprocess that represents the preparation of a lecture is shown in Figure 4.83.



**Fig. 4.84.** Task types specify the kind of task that is represented

In BPMN, tasks can be decorated with task types which makes it easier for human readers to understand the specific type the task represents. Figure 4.84 lists the task types of the BPMN.

*User tasks* represent traditional workflow tasks that involve user interaction. When the process comes to a point where a specific task is to be performed by a user, the user is informed, for instance, by the appearance of a new work item in his or her inbox.

When selecting the work item, an application is started that the user works with in order to perform the task. To facilitate role resolution, role and skill information are typically associated with a user task. Integration with organizational modelling is required to facilitate role resolution, because the BPMN does not support the modelling of detailed organizational aspects.

*Manual tasks* are performed without the support of software systems. Sending a printed letter or transferring goods in a logistics environment are examples of manual activities. While the actual execution of these activities is outside the scope of an information system, the business process management system needs to be informed about the completion of a manual activity.

The completion information typically includes a return code, so that the system is aware of a successful or unsuccessful completion of the manual task. This information can be important for the remaining parts of the business process, so that in the case of unsuccessful completion, the business process can take compensating actions for the failed manual activity.

Business rules are logical rules to be interpreted by a rules engine. In BPMN we can model a task that triggers a business rule by marking it with a business rule marker and adding the appropriate information. When the *business rule task* task is executed, the business rule is invoked. The actual representation of business rules and their enactment using rules engines is not in the scope of the BPMN.

A *service task* is implemented by a piece of software, either using a Web services interface or an application programming interface to a software system. A *script task* is a task that uses some scripting language expression in order to be performed. Script tasks are used to represent simple functionality,

for which no dedicated software system is required. The particular scripting language used and the interaction platform for script expressions depend on the tool support available. When the script completes execution, the script task completes.

There are also task types related to sending and receiving messages. Since these task types rely on events, we will introduce events first and return to send and receive tasks only when we have done so.

A *compensation task* is invoked to compensate for activities that need to be undone. The compensation concept is strongly connected to transactions, which will be discussed in the context of compensation events in the next section.

## Events

Events play a central role in business process management, since they are the glue between situations in the real world and processes that will react to these events or trigger them. Events in a business process can be partitioned into three types, based on their position in the business process: start events are used to trigger processes, intermediate events can delay processes or they can be triggered during process executions. End events signal the termination of processes. There are obvious connection rules associated with these events. Start events have no incoming edges, end events have no outgoing edges, and intermediate events have both an incoming and an outgoing edge.















































































This book covers the most common event types, shown in Figure 4.85. The rows contain the event types, the columns the position (start, intermediate, end) and the nature of the event, discussed shortly. There are also intermediate events that are attached to boundaries of activities rather than having an incoming sequence flow.

The simplest type of event is the blanco event that has no marker. (The standard calls this event *none event*. Since blanco events are in fact events, we stick to the former terminology and use the term *blanco event*.) This event type is used whenever the cause of the event is either not known or is irrelevant for the current modelling purpose. Blanco events can be used as start events or as end events.

Events play two major roles, and each event in a process model plays exactly one of those. These roles are referred to by *catching* and *throwing*. An event is of catching nature, if the process listens and waits for the event to happen. Whenever the respective event happens, the process catches it and reacts accordingly. All start events are catching events.

An event is of throwing nature, if it is actively triggered by the process during process execution. Sending a message to a business partner is an example of a throwing event. All end events are throwing events, because the end event is actively triggered by the process.

Intermediate events can be either catching or throwing. A good example is the intermediate message event, which comes in two flavors. As throwing

	Start Events	Intermediate Events				End Events
	Catching	Catching	Boundary Interrupting, Catching	Boundary Non-Interrupting, Catching	Throwing	Throwing
<b>None or blanco:</b> Untyped events, indicate start point, state changes or final states.						
<b>Message:</b> Receiving and sending messages.						
<b>Timer:</b> Cyclic timer events, points in time, time spans or timeouts.						
<b>Escalation:</b> Escalating to an higher level of responsibility.						
<b>Conditional:</b> Reacting to changed business conditions or integrating business rules.						
<b>Link:</b> Off-page connectors. Two corresponding link events equal a sequence flow.						
<b>Error:</b> Catching or throwing named errors.						
<b>Cancel:</b> Reacting to cancelled transactions or triggering cancellation.						
<b>Compensation:</b> Handling or triggering compensation.						
<b>Signal:</b> Signalling across different processes. A signal thrown can be caught multiple times.						
<b>Multiple:</b> Catching one out of a set of events. Throwing all events defined.						
<b>Parallel Multiple:</b> Catching all out of a set of parallel events.						
<b>Terminate:</b> Triggering the immediate termination of a process.						

**Fig. 4.85.** Common event types in the BPMN, adapted from the BPMN Poster, BPM Offensive Berlin (2011)

event, the intermediate message event sends a message to a business partner. As catching event, the process waits for a message to come in, that is, it waits for the event to happen.

This example shows quite clearly the difference. Catching events wait for things to happen, while throwing events actively trigger events.

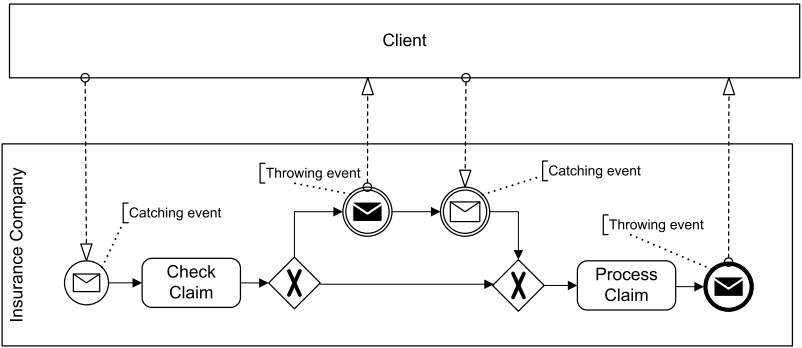


Fig. 4.86. Throwing and catching events

Throwing and catching events are further illustrated in Figure 4.86, which shows a claim handling process of an insurance company involving interactions with a client. The process starts with the start message event “catching” the claim message. (More precisely, with the start event catching the event that represents the incoming claim message.)

In case the claim is incomplete, the insurance company sends a request for clarification to the client. This sending of a message is represented by a throwing intermediate message event. The event symbol is marked with a black envelope to show this throwing behaviour, that is, the sending of the message. At this point, the process waits for the event that represents the receipt of the answer message from the client. The intermediate message event catches this event and continues with processing the claim and sending the response letter to the client in the (throwing) message end event.

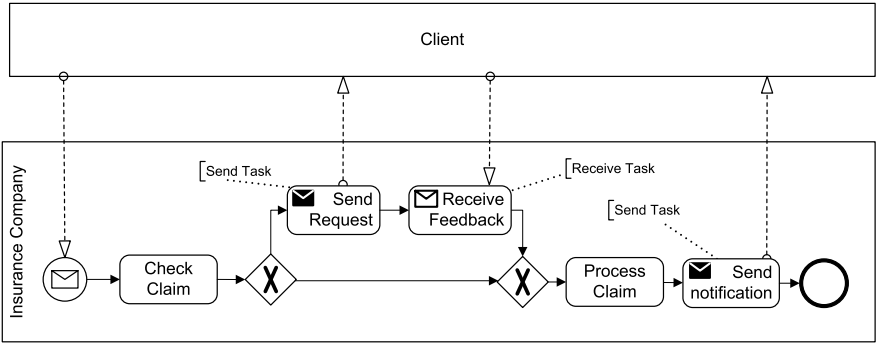


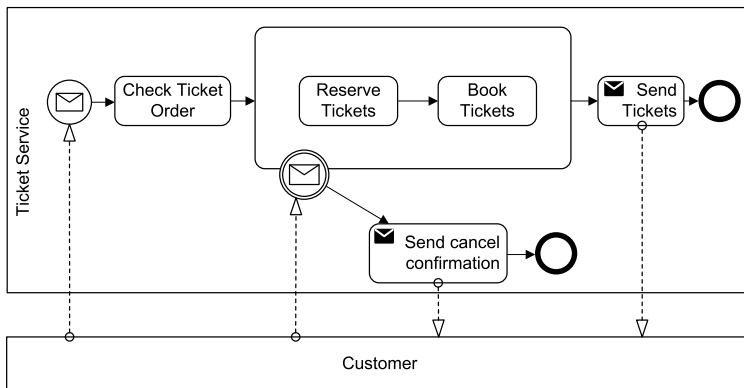
Fig. 4.87. Using markers to identify send tasks and receive tasks

Task types, introduced above, also provide options of expressing certain real-world situations, for instance, related to sending and receiving messages. Instead of send events, we can also use send tasks, that is, tasks that are marked with task type send. Like with sending and receiving events, a dark envelope represents send tasks, while the light envelope shows that a task receives a message. Figure 4.87 shows a process diagram using send and receive tasks. BPMN also supports a specific type of receive task that can be used to instantiate a process, but a receive event is much more appropriate in most cases.

Returning to blanco events, blanco start events are of catching nature, while blanco end events are of throwing nature. Since there is no way of marking blanco intermediate events as either catching or throwing events, by convention, all blanco intermediate events are—by definition—of throwing nature.

Message events are among the most often used events in BPMN. We have already seen in Figure 4.86 message start events, message end events, and intermediate message events of catching and throwing nature. We now look at intermediate events on the boundary of activities, called boundary intermediate events, or attached intermediate events.

Each intermediate event of this kind is associated with an event context, used to determine whether the event has occurred. In particular, the intermediate event will be triggered only, if the activity that the intermediate event is attached to is still active when the event occurs.



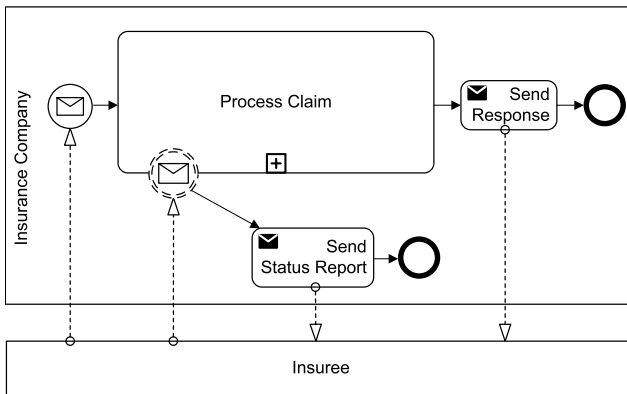
**Fig. 4.88.** Process diagram with interrupting boundary event

An example of a process model containing a boundary event is given in Figure 4.88. The process starts by a customer sending a ticket order to a ticket service. After receiving the message and checking the order, a subprocess is entered. In the subprocess, the tickets are reserved and finally booked. Notice that processes and subprocesses do not require start and end events. While it

is good practice to use start and end events on the process level, they might be dropped for simple subprocesses. The boundary event represents the option of the customer to cancel the order.

If the cancellation message is received while the subprocess is still active, the subprocess is cancelled, and the confirmation of the cancellation is sent. If no cancellation message is received while the subprocess runs, the subprocess completes and the tickets are sent.

In this example we have discussed a boundary event that interrupts the subprocess it is attached to. But boundary events might also be of non-interrupting nature.



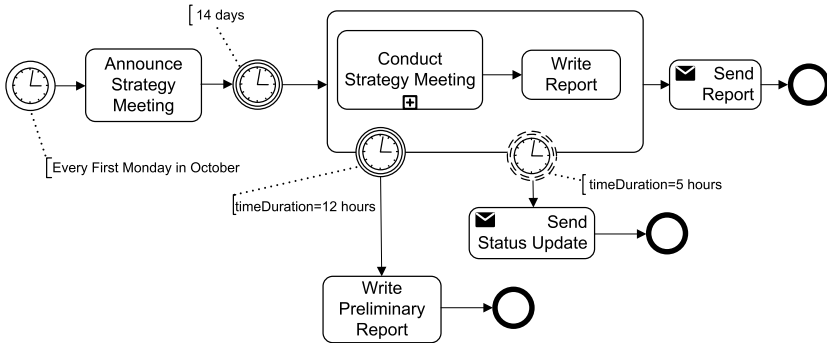
**Fig. 4.89.** Process diagram with non-interrupting boundary event

An example of a non-interrupting boundary event is shown in Figure 4.89. In this process, an insuree sends a claim report to an insurance company. The complex processing of the claim is hidden in the subprocess *Process Claim*. While this subprocess is active, the insuree can ask, even multiple times, for the current status of the claim handling. The respective incoming message sent is caught by the boundary event, and a status report is sent. Since the processing of the claim should not be interrupted by this request, the boundary event is non-interrupting, indicated by its dashed outline.

Timer events are used frequently in process diagrams. They are quite versatile, since they can represent time intervals, points in time, and timers, similar to count down watches.

Figure 4.90 shows a process diagram involving several types of timer events. The process starts with a start timer event. By the annotation we learn that the process is instantiated every first Monday in October. When this event is caught, a strategy meeting is announced. Then the process pauses for 14 days, represented by the intermediate timer event.

The execution semantics of intermediate timer events is as follows. When the previous activity is completed, the timer is started. This is like starting a



**Fig. 4.90.** Process diagram with interrupting and non-interrupting boundary timer events

count down watch with the value set to 14 days in this case. In Figure 4.90, we used an annotation to show the time period.

In BPMN, timer events have attributes that are used to represent timer values in a structured fashion. In particular, the attribute `timeDuration` holds an expression that defines the time duration the timer waits for. Attributes `timeDate` and `timeCycle` are used to specify points in time and recurring timers, respectively.

After the duration has elapsed, the subprocess and thereby the strategy meeting can be started. When this happens, two additional timers are started for the boundary events. Timer 1 with duration 12 hours for the interrupting timer event and Timer 2 with duration 5 hours for the non-interrupting timer event.

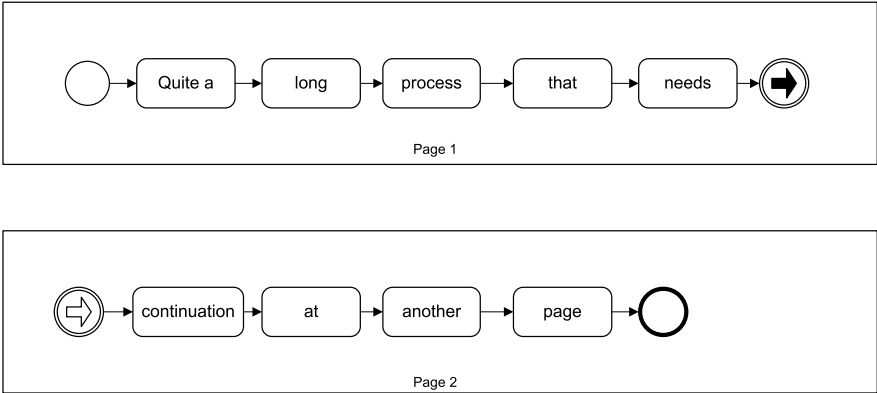
Assuming the subprocess is still active after 5 hours, Timer 2 is triggered, and a status update is sent. This event does not interrupt the subprocess. After being triggered, this timer is immediately reset, so that it can trigger the sending of the next status update after another 5 hours, if the meeting is still ongoing at that time. This example shows that non-interrupting boundary events can occur multiple times while the subprocess is active.

If the subprocess is not completed after 12 hours, Timer 1 elapses, and the subprocess is interrupted. A preliminary report is written, and the process terminates.

Link events are quite specific, since they—unlike all other events—do not represent something that happens in the real world. Rather, they are a means to layout large process diagrams that span multiple pages or screens. A part of the process ends with a link event of throwing nature, while the next part of the process starts with a link event of catching nature. Consequently, link events are intermediate events, even though they have no outgoing (throwing link event) or no incoming edge (catching link event).

Regarding the execution semantics, two matching link events are equivalent to a sequence flow. It is important to stress that link events do not connect





**Fig. 4.91.** Link events connect different parts of one process

multiple processes, but just parts of one process. Link events are illustrated in Figure 4.91.

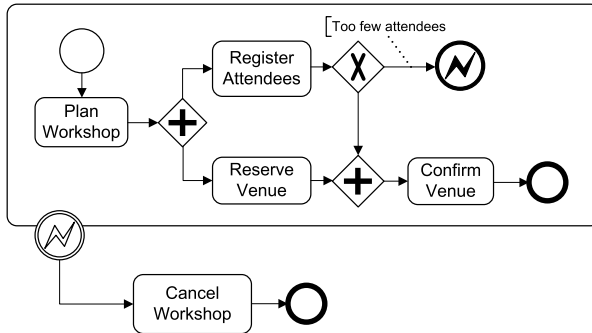
Error events also come in two flavours. A throwing error event indicates the occurrence of an error in a certain scope, for instance, in a subprocess. Catching error events are always on the boundaries of subprocesses. They catch the error and interrupt the subprocess, in case certain parts of the subprocess are still active. After catching an error, typically error handling activities are performed.

An example involving error events is shown in Figure 4.92. A subprocess for planning a workshop consists of planning the workshop followed by concurrent activities involving the registration of attendees and reserving a venue. If too few attendees register for the workshop, an error is thrown, and the workshop has to be cancelled. The cancellation is facilitated by a boundary error event that catches the occurrence of the error within the subprocess. In this example, error handling is done by the *Cancel Workshop* activity.

Notice that once the error event is thrown, any running activity in the subprocess will be interrupted. In the example, *Reserve Venue* might be still running at that point in time. If this is the case, it is interrupted immediately, since no venue needs to be reserved when the workshop is cancelled.

Compensation events are strongly connected to transactions. Transactions are specific subprocesses, whose activities have transaction semantics, specified by a transaction protocol. Probably the most widely used transaction protocol is the ACID model, which states that transactions have the following ACID properties:

- *Atomicity*: Either all activities in a transaction are executed successfully or none is.
- *Consistency*: The correct execution of a transaction brings the system from one consistent state in another consistent state.



**Fig. 4.92.** An error is thrown in a subprocess; it is caught by an error boundary event attached to that subprocess

- *Isolation*: The activities of a transaction are executed in isolation from other transactions, that is, transactions do not interfere with each other.
- *Durability*: Effects of transactions survive any system failure that might occur at a later point in time.

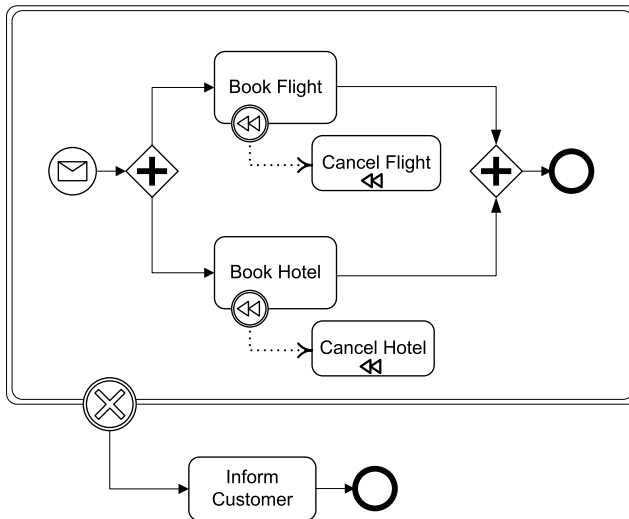
Assuming the ACID transaction model, all transactions need to obey the atomicity property: Either all activities of the transaction need to be successfully completed, or none at all. In database systems, this all-or-nothing property of transactions is typically implemented by locking protocols or multiversion concurrency control schemes.

In business processes, the situation is a bit more complex, since we cannot lock large parts of business processes for an extended period of time or create multiple versions of the same data object. In business process management, the typical assumption is that each activity is executed in an atomic fashion. This means, however, that one activity of a transaction can have completed already, when another activity decides to fail. In this case, the first activity needs to be undone, using compensation. An example is used to illustrate this concept.

Figure 4.93 shows a business process diagram that contains a transaction. The transaction involves activities to book a flight and to book a hotel. The all-or-nothing property of this transaction states that either both activities are performed successfully or none will. This property makes sure that the traveller will not end up with a flight booked and no hotel room booked, or vice versa.

Thus, the process needs to rule out that one activity of the transaction succeeds, while the other activity fails. This is done by compensation. Assume that the hotel booking activity is performed successfully, but the booking of the flight fails. In this case, the booking of the hotel room needs to be undone. This is done by cancelling the booking of the hotel room.

When one activity in a transaction subprocess fails, for all activities of the transaction that have been successfully executed already, the compensating



**Fig. 4.93.** Business process diagram with transaction and compensation elements, adapted from Object Management Group (2011)

activities are started. In this example, after the booking of the flight fails, the *Cancel Hotel* compensating activity of the *Book Hotel* activity is executed. Thereby, the booking of the hotel room is undone, so that the effects of neither of the activities in the transaction are persisted.

The cancellation boundary event catches the unsuccessful completion of the transaction. It can be used to execute activities after the transaction has unsuccessfully completed, like informing the customer in the example.

Signal events communicate certain situations to a wide audience. For each signal throw event, there can be several events that catch the signal. Similar to a flare that can be seen in a wide perimeter, a signal can be caught by different parts of the same process, by other processes within the same process diagram, and even by other process diagrams. Signals are similar to the event publication / event subscription mechanism in distributed computing, where a given signal event can have many subscribers throughout the process landscape.

There are additional types of events that the BPMN supports. For more information about these—less frequently used—events, the reader is referred to the standards document.

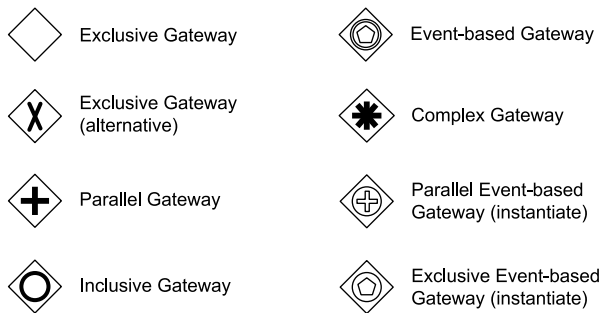
## Sequence Flow and Gateways

In the BPMN, control flow is called sequence flow. Sequence flow is represented by solid arrows between flow objects, that is, activities, events, and gateways. BPMN supports several types of sequence flow, including normal flow, conditional flow, default flow, and exception flow.

The normal flow of a business process represents expected and desired behaviour of the process. It begins in the start event of a process diagram and continues via a set of flow objects until it reaches an end event.

Exceptional situations are represented by exception flow. With respect to process execution semantics, there is no difference between normal flow and exception flow. The only difference is that exception flow does not define the desired flow of the process, but exceptional situations. Exception flow is created by intermediate events attached to the boundary of an activity, as discussed above in the context of boundary events.

There are two additional types of sequence flow, namely conditional flow and default flow. Since these play important roles in the context of gateways, we introduce gateways first.



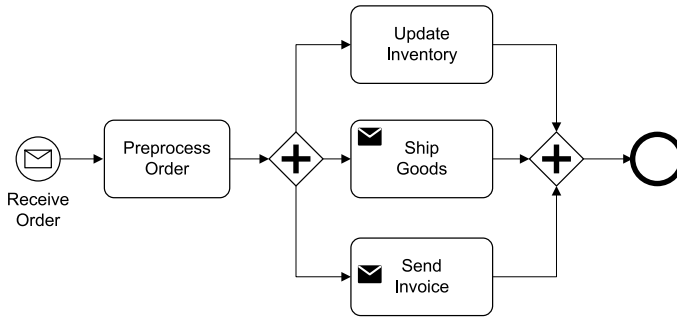
**Fig. 4.94.** Gateway types in the BPMN, Object Management Group (2011)

In BPMN, each gateway acts as a join node or as a split node. Join nodes have at least two incoming edges and exactly one outgoing edge. Split nodes have exactly one incoming edge and at least two outgoing edges. We can also express gateways with multiple incoming and multiple outgoing edges in BPMN. These gateways are called mixed gateways. Since two behaviours—split and join—are expressed by a single concept (for example, exclusive or), best practice is not to use mixed gateways but to use a sequence of two gateways with the respective split and join behaviour instead.

The gateway types of the BPMN are shown in Figure 4.94, that is, the *exclusive*, the *parallel*, the *inclusive*, the *event-based*, the *complex* and two instantiation gateway types, which are discussed later in this section.

As shown in that figure, there are two representations of the exclusive gateway, one without a marker and one with a marker. This feature of an unmarked gateway in the BPMN can be considered inconsistent with the definition of an unmarked, that is, blanco event. The *blanco gateway* actually represents a particular kind of gateway, while the *blanco event* does not. To avoid misunderstandings, we recommend to always use gateway markers.

Parallel split and join is supported by virtually any process modelling language. In the BPMN, there is a parallel gateway that can be used to represent *and split* and *and join* behaviour.



**Fig. 4.95.** Example involving the parallel gateway

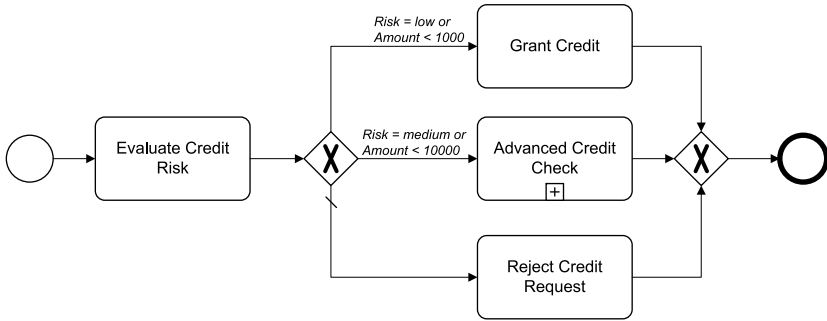
An example is shown in Figure 4.95. This process starts with receiving and preprocessing an order. Then the parallel gateway triggers the execution of three activities. The inventory is updated, the goods are shipped, and the invoice is sent. There are no execution constraints defined between these activities, they can be executed concurrently. When the activities have completed, the *and join* synchronizes the parallel flows, and the process terminates.

Exclusive gateways are also available in any process modelling language. The gateway realizes an “exclusive” behaviour, because exactly one option is chosen from a set of alternatives. An example is shown in Figure 4.96. After the credit risk is evaluated, an exclusive gateway is reached.

This gateway decides which checking activity shall be executed. The credit is granted if the credit risk is low or a certain threshold value is not exceeded. In case of medium credit risk, a subprocess for an advanced credit check is started. If neither of these conditions holds, the credit request is rejected. This behaviour is operationalized by formal conditions, attached to the sequence flows. A sequence flow stores its condition in its `conditionExpression` attribute.

To decide on the branch to select, the exclusive gateway uses these conditions. These sequence flow edges are then specialized to condition flow edges. The standard defines that the conditions are “evaluated in order”. The first condition that is evaluated to true is chosen. An exclusive gateways might have a default edge, which does not have a condition attached. It is always evaluated last. This execution semantics makes sure that exactly one outgoing edge will be triggered.

Notice that this property holds even if there are overlapping conditions on the outgoing edges of the gateway. In Figure 4.96, the conditions are overlapping, since both conditions might evaluate to true, for example, if *Risk=medium and Amount=800*. Assuming that the conditions are evaluated



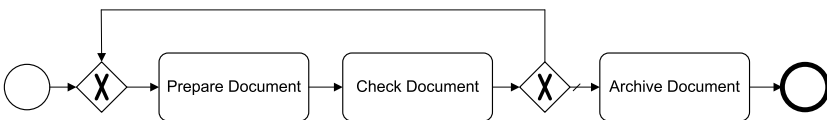
**Fig. 4.96.** Exclusive gateway with conditions and default flow

from top to bottom, then the first condition evaluates to true, and *Grant Credit* is selected. However, if the other condition appears first in the condition evaluation ordering, then *Advanced Credit Check* is chosen.

If the requested amount exceeds 10000, none of the conditions evaluates to true, so that the default flow is taken and the request is rejected. In any case, the exclusive or split semantics of the gateway is realized.

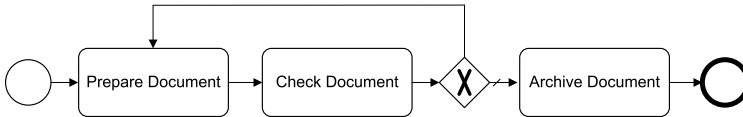
Since the decision is taken based on data, for instance, the value of the *Risk* and *Amount* data objects, the exclusive gateway is also called data-based exclusive gateway.

Figure 4.97 shows a process diagram with a loop. The loop is represented by two exclusive gateways, a join gateway and a split gateway. It is a characteristic of a loop that the join appears before the split in the process flow. After a document is prepared, it is checked. Depending on the outcome of the checking activity, either the process is continued with archiving the document or the loop is iterated.



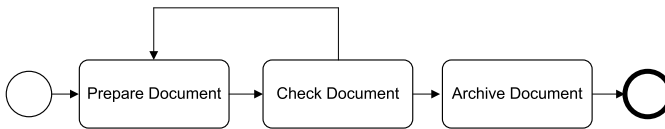
**Fig. 4.97.** Exclusive gateways realizing a loop

In BPMN, the process could also be represented by using a single gateway instead of two gateways. The split gateway that decides whether to iterate the loop needs to be kept. Instead of the join gateway in the beginning of the process, the edge from the split gateway can directly lead to the first activity in the loop. According to the BPMN, activities with multiple incoming edges act as merge nodes. Such an activity gets enabled and can be executed, if one of its incoming edges is triggered. This process is shown in Figure 4.98. It has exactly the same execution semantics as the process shown in Figure 4.97.



**Fig. 4.98.** Process diagram with uncontrolled flow

Process activities with multiple outgoing edges are also possible in BPMN. In this case, each of the outgoing edges will be followed. These activities might lead to modelling errors. The reason being that the split behaviour of activities with multiple edges is different from their join behaviour. Activities with multiple outgoing edges represent a parallel split gateway, while activities with multiple incoming edges realize a merge gateway.



**Fig. 4.99.** Process diagram with split and join activities, representing a livelock

This fact is illustrated by Figure 4.99, which shows a variant of the previously discussed process with activities acting as split nodes (*Check Document*) and activities acting as join nodes (*Prepare Document*). As a result, for each iteration of the loop, both outgoing edges of the checking activity are triggered. For each iteration of the loop, the document is archived. In addition, the loop will never terminate, resulting in a livelock.

This modelling error could be fixed by attaching conditions to the outgoing edges of the *Check Document* activity, that is, by using conditional flow. However, it is good practice for process activities to have exactly one incoming edge and exactly one outgoing edge. The split and join behaviour of the process should be represented explicitly by gateway nodes rather than implicitly by activities with multiple incoming or multiple outgoing edges.

Just like an exclusive gateway, an *event-based* gateway realizes an exclusive choice. However, rather than deciding itself using process data, the gateway uses the environment to let others decide on how to continue the process. It allows several events to happen, and the environment decides on what actually will happen.

A typical usage of this pattern is shown in Figure 4.100. The process starts with the sending of an invoice by a reseller to one of its customers, followed by an event-based gateway. When the gateway is reached, two things can happen. Either the funds are received or the timer event occurs. Whichever occurs first, decides, that is, the environment decides on how the process continues.

On the completion of the gateway, the *Receive Funds* task is enabled. At the same time, a count down timer for the intermediate timer event is

started with a duration of 14 days. If the amount is received within 14 days, the intermediate timer is deactivated and the process completes. If, however, the customer does not pay within 14 days, the timer event occurs, and a reminder is sent. Afterwards, the gateway is reached again, and the customer has another 14 days for paying his invoice.

Notice that only catching intermediate events and receive tasks can occur after event-based gateways. The standard defines that either receive tasks or message intermediate events can occur after a given event-based gateway, not both. In addition to timer intermediate events, like shown in the example, signal events and a few other events are allowed at this position, but message events and timer events occur most frequently.

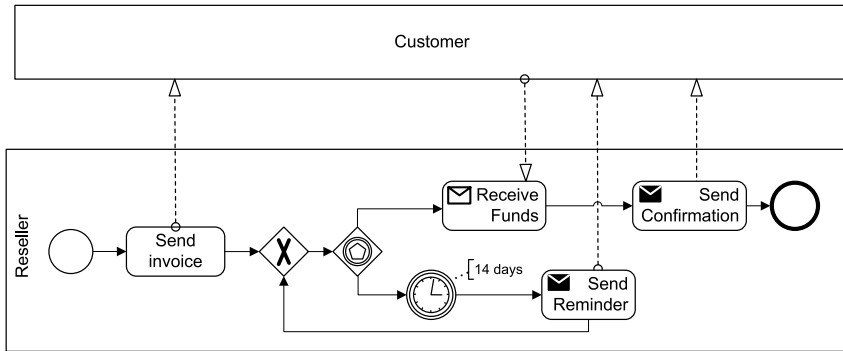


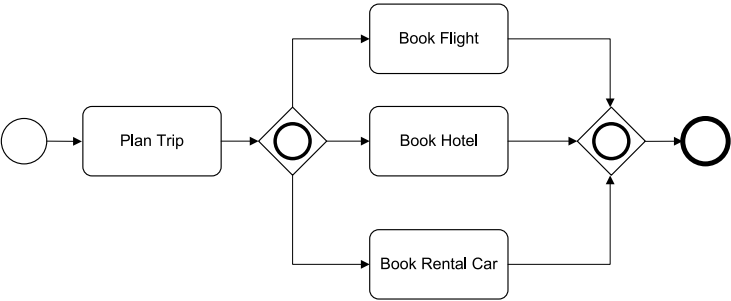
Fig. 4.100. Example of an *event-based gateway*

The semantics of an event-based gateway is fundamentally different from the semantics of a data-based exclusive gateway. In an event-based gateway, multiple activities are enabled and ready for receiving messages at the same time, realizing the *deferred choice* pattern. In the data-based exclusive gateway, the decision is made by the gateway itself—more precisely, by the conditions associated with the condition flow edges leaving the gateway. However, both gateways exhibit an *exclusive or* semantics.

The *inclusive* gateway exposes the most flexible behaviour, since it subsumes and extends both exclusive gateways and parallel gateways. Inclusive gateways can be used in situations where an arbitrary non-empty set of outgoing branches need to be selected. As with the *data-based exclusive or split*, it is the responsibility of the modeller that at least one branch be chosen. An example of an inclusive is shown in Figure 4.101, where a trip is planned and then—depending on the concrete planning of the trip—any subset of flight, hotel, and rental car is booked.

A *complex gateway* allows the definition of combined split and join behaviour. Consider a *complex split gateway* with outgoing sequence flows to *A*, *B*, and *C*. The gateway may define that either *A* or, jointly, *B* and *C* need



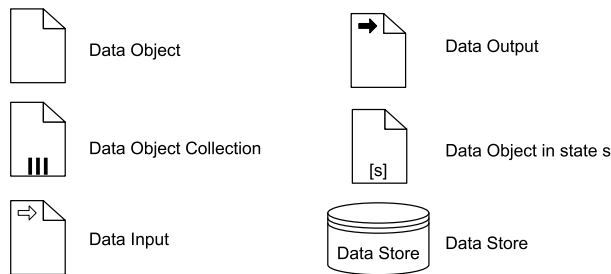


**Fig. 4.101.** Example of an *inclusive or* gateway

to be executed. It may also define that any pair of sequence flows is valid. The behaviour is specified in the activation condition and an expression of the gateway. The behaviour of the complex gateway is not known from its visual appearance, so that modellers should use this construct with caution.

**Handling Data**

All business processes deal with information or physical artefacts. To represent information and physical artefacts, BPMN provides data objects. While the term data object seems to indicate digitalized information, it also covers physical objects, such as documents and products.



**Fig. 4.102.** Notational elements regarding data

The notational symbols regarding data in BPMN are shown in Figure 4.102. Often, data objects represent digitalized objects, such as orders in an information system. Since BPMN concentrates on process modelling, there are no data modelling capabilities available. This would also not be appropriate, since the UML provides excellent data and object modelling capabilities, for example, class diagrams.

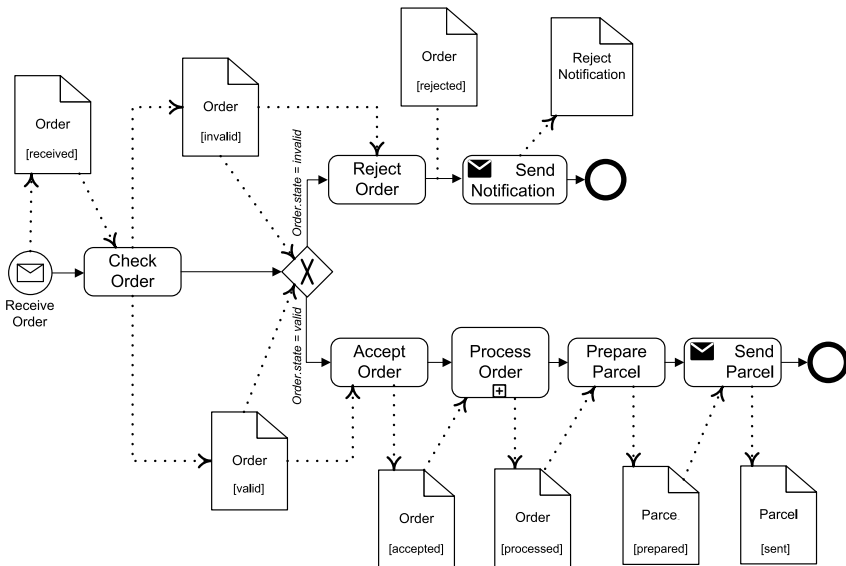
The relationships between data objects and activities or, more generally, flow objects are specified by data associations. A directed edge from an activity

to a data object means that the activity creates or writes the data object. Directed edges in opposite direction indicate read relationships.

Typically processes use data that has been created before the process has started. Examples of this type of data is customer information stored in a customer relationship management system or production information stored in a database. To represent that a process uses these types of data, input data objects can be used. Analogously, if data objects are created as output to be used by other processes, these are marked with a data output marker, as shown in Figure 4.102.

Information systems that store data can be represented in process models as data stores. Since BPMN covers a wide spectrum of application domains, also non-technical stores such as, for instance, warehouses, can be represented by data stores.

The life time of data items that are neither data input nor data output is restricted to the duration of the process instance, that is, data objects are volatile. An association of a data object with a data store, however, indicates that the data object is persistently stored in that data store. Therefore, data stores do not only serve the documentation purpose but also carry a semantics that is important for an implementation of the process.



**Fig. 4.103.** Process diagram involving data objects

A sample business process involving data objects is shown in Figure 4.103. In this order handling process, the start message event occurs when an order is received. This message contains a data object *Order* in state *received*, indicated by the association from the start event to that data object.

The *Check Order* activity might produce different results: either the order is valid or invalid. This behaviour is represented by two data object symbols in the diagram, which have different state markers, reflecting the outcome of the checking activity.

This illustrates that, in general, an association from an activity *A* to a data object *D* in state *s* means that *A* might change the state of *D* to *s*, but it might as well not. Modellers need to make sure that at least one of the data objects an activity is related with in a write association, is actually created as output.

In the example, the order checking activity changes the state of the data object to either *valid* or *invalid*. The current value or state of a data object can be used by expressions, for instance, by expressions that decide which path is taken following an exclusive gateway.

The respective attributes of the conditional flows leaving the exclusive gateway are visualized. If *Order.state=invalid*, the upper branch is chosen and the order is rejected. If *Order.state=valid*, the lower branch is chosen and the order is accepted. In the former case, a rejection message is sent. If, however, the order is accepted, the order is processed, a parcel is prepared and sent.

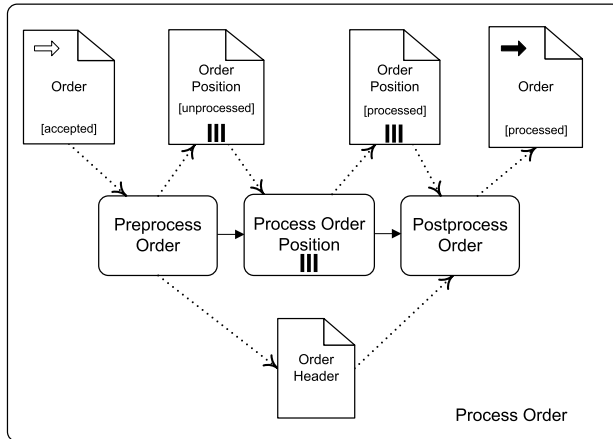
Notice that the *Prepare Parcel* activity reads an order in state *processed*. This is a typical use of data objects including states; it implements a business policy that a parcel can be prepared only if the respective order is processed. While this situation is obvious for the process shown, the language features provide additional expressiveness regarding data, which proves quite useful in real-world settings.

There is a shorthand notation for a data flow between activities that follow each other directly in sequence flow. Rather than providing two edges from and to, respectively, the activities, the data object can simply be associated with the sequence flow connecting these activities. This language construct is illustrated in Figure 4.103 to show the data flow between the *Reject Order* and *Send Notification* activities.

Data objects also come in collections. A process activity may process a collection of data, such as a list of data, instead of an individual data object. A sample business process involving a collection of data objects is shown in Figure 4.104.

As shown in Figure 4.103, the *Process Order* subprocess reads the order data object in state *accepted* and changes the state of that data object to *processed*. The data objects in these states are also shown in the subprocess refinement in Figure 4.104. Notice that the order in state *accepted* is a data input of the subprocess, while the order in state *processed* is data output of the subprocess. This example shows how data objects are communicated from a subprocess activity to its internal process and back.

When the subprocess starts, the order is preprocessed, resulting in a list of order positions and a data object representing the order header. The list of order positions serves as input to the *Process Order Position* activity. As



**Fig. 4.104.** Diagram of the *Process Order* subprocess from Figure 4.103, involving data object collections

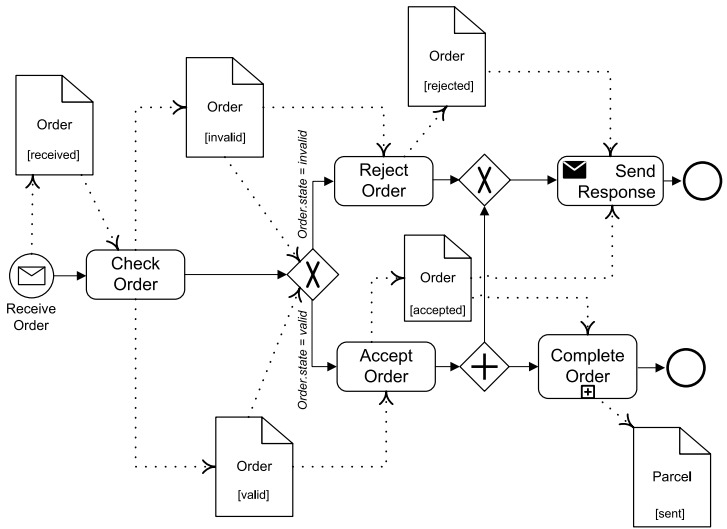
shown by the marker of this activity, it is a multiple instance activity. With a data object collection as input, the multiple instance marker indicates that an activity instance is created for each object in the collection. In our example, each order position is processed by an individual instance of the *Process Order Position* activity.

Once all order positions are processed, the respective data object collection is created, and the multiple instances activity terminates. Postprocessing of the order involves assembling the order positions and the order header to create the order, which is now in the processed state. That data object is provided to the follow-up activities on the process level as data output, as shown in Figure 4.103.

Finally, we sketch the execution semantics of data objects in BPMN. Each process activity is associated with input sets, which contain data objects which have to be available when the activity starts. Notice that this set can be empty in case an activity is not associated with any data object. In the example, the activity *Accept Order* in Figure 4.103 has one input set containing just one data object, namely the order data object in the valid state.

In general, however, there might be multiple input sets associated with a given activity. When sequence flow arrives at the activity, the input sets are visited. For each input set, the system checks if the data objects are available in the requested states. The activity can be started, once all data objects for an input set are available, making sure that an activity can deal with alternative input data objects.

This approach is illustrated in Figure 4.105, which shows a variant of the ordering process discussed above. In this variant, a response message is sent in any case. To realize this behaviour, the *Send Response* activity has two



**Fig. 4.105.** Process diagram involving multiple input sets of an activity

input sets, one of which consists of the order object in state *rejected*. The other consists of the same data object in state *accepted*.

From the discussion of the process it is obvious that these input sets are alternative. Either the response message contains the information that the order is rejected or it sends a message that tells the client that the order is accepted. When control flow enters the *Send Response* activity, either of the input sets is available, realizing the intended process behaviour.

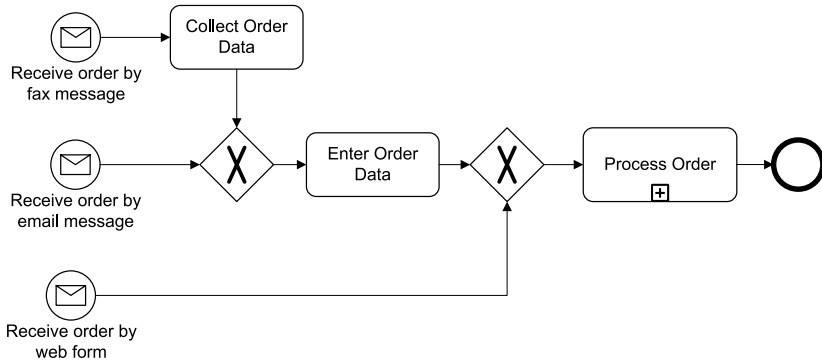
### Process Instantiation

So far, most aspects of BPMN process diagrams have been covered. Activities, events, gateways, and sequence flow were introduced and their execution semantics have been discussed. In formal language theory, the semantics of a language or grammar determines the meaning of the words, written in that language. In process languages like the BPMN, the meaning of the process diagrams—the words of that language—is defined by the behaviours that the diagram specifies.

For each language construct covered, its execution semantics was discussed. For example, a sequence flow between two activities restricts their execution ordering, after an exclusive gateway exactly one option will be chosen, etc.

However, so far we have disregarded the question when a process should actually be instantiated. This is an important aspect of the execution semantics of a process language. Luckily, the process diagrams discussed so far always had a single start event. In this case, the instantiation question can trivially be answered: A process should be instantiated if and when the start event occurs.

The case is more complex for process diagrams with multiple start events. The BPMN states that start events are alternative. This means that whenever one start event occurs, a process is instantiated.



**Fig. 4.106.** Process diagram with multiple alternative start events

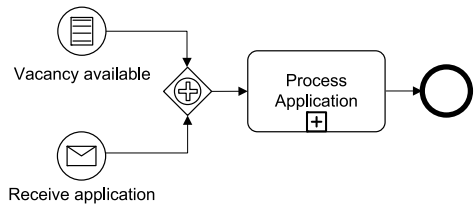
Figure 4.106 shows a process diagram with several start events. These events represent alternative ways of receiving an order. If the order is received by fax message, the data first needs to be digitalized in the *Collect Order Data* activity. Once the data is available in electronic form, it can be entered in the information system of that company, using the *Enter Order Data* activity. Notice that the order can be immediately entered in the information system, if it is received by email message. Finally, the order can be processed immediately after process start, if the order has been issued via a web form.

These alternative ways of receiving an order can be represented by multiple start events. These start events are alternative, and the process is in line with their alternative nature, because all start events are merged by exclusive gateways. Notice that in this example substituting an exclusive gateway with a parallel gateway would result in a deadlock situation.

However, there are situations which do require multiple start events. The BPMN reserves a specific element for these situations, shown in Figure 4.107. In that example, a process is instantiated only if an application is received and a corresponding vacancy is available. Notice that the availability of a vacancy is represented by a condition start event. (For illustration purposes, we use a condition start event here, even though BPMN allows only message start events). We use the parallel event-based gateway to capture this situation. A process is instantiated only if all incoming events have occurred.

This example covers another interesting aspect. When multiple start events are required to start a process, these start events need to be correlated with each other. Correlation is used to tie events to process instances.

In the concrete example, the *Receive application* event needs to reference a vacancy. When there are multiple vacancies and multiple applications re-



**Fig. 4.107.** Process diagram with two start events, both of which need to occur to instantiate the process

ceived, a process can only be started if an application is received and the vacancy referenced in the application is actually available.

To illustrate this concept, assume vacancies  $V1, V2$ , and  $V3$ . This means that condition start events occur only for these vacancies. If an application is received that references a vacancy that is not available, for example  $V4$ , then no process instance can be instantiated. If, however, an application is received which references vacancy  $V1$ , a process is instantiated. This makes sure that a process is instantiated only if there is an open position available for the application received.

4.7.3 Collaborating Processes

Business processes involving multiple organizational entities can interact with each other. The BPMN is not restricted to single-organization business processes, but is ready to express business processes of multiple organizations that collaborate.

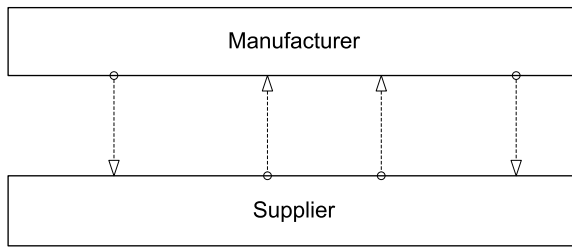
As already introduced, pools represent specific process participants or roles, such as role supplier or role customer. Lanes are used to represent organizational entities within participants. Typically, top level divisions within companies are represented by lanes, such as marketing and sales, operations, and logistics; but more fine-grained organizational entities can also be represented by sub-lanes, if required by the process.

Sequence flow is allowed within processes only, that is, between nodes that reside in a single pool. Therefore, sequence flow may cross lane boundaries, but it may never cross a pool boundary. Communication between processes can occur only through message flow.

The rationale behind this stipulation is as follows. Sequence flow defines an execution order of activities in a given process. Within an organization, we can set up procedures and rules, even a workflow engine, that make sure that the activities are executed as specified in the process model.

However, we can not ask for a certain execution ordering of activities in a process of one of our business partners. We can only send a message to our business partner, which will then influence its business processes. Therefore, business-to-business communication is handled exclusively through messages,

while intra-company communication can be handled directly through sequence flow.



**Fig. 4.108.** Business processes collaborating through message flow

Collaborating processes can be represented on different levels of abstraction. In the most abstract way, only the roles of the partners are represented and the message flows between them. There is no information about the internal processes available. Also the ordering of message flow edges from left to right does not have any meaning.

Figure 4.108 shows a collaboration diagram involving a supplier and a manufacturer. The diagram does not indicate that first the manufacturer sends a message to the supplier, even though the left most edge has that orientation. We can not even conclude from the diagram that the message flow actually happens. A message send event might be on an optional path, so that not all process instances actually send a message!

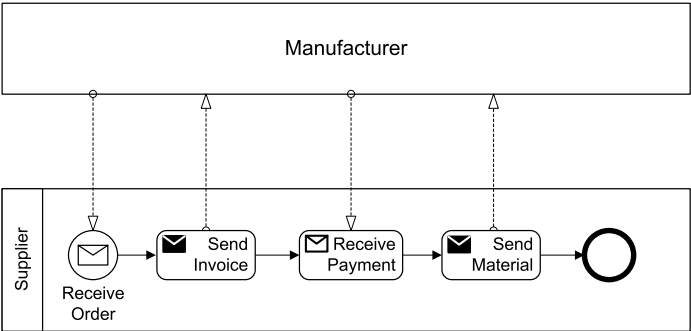
Since we cannot look inside these pools, they are called black box pools. Collaboration diagrams involving black box pools provide a high level view by providing roles of participants and message flow that might occur.

An example of a business process with one black box pool and one white box pool is shown in Figure 4.109. A manufacturer sends an order to its supplier, represented by a message flow from the manufacturer pool to the message start event of the supplier. Then an invoice is sent, payment is received, and the material is sent.

In a typical business-to-business collaboration, business partners communicate in a structured way by sending and receiving messages. While the externally visible behaviour of a process that runs in a given organization is essential for the overall communication, the internal process is not relevant. Pools can also be used to provide this form of abstraction. The internal structure of a business process can be abstracted from and, only the externally visible communication behaviour can be shown.

There are two advantages related to expressing only the externally visible behaviour. The first advantage is that the information hiding principle is followed, so that the complexity of internal business processes does not add to the complexity of the overall process.

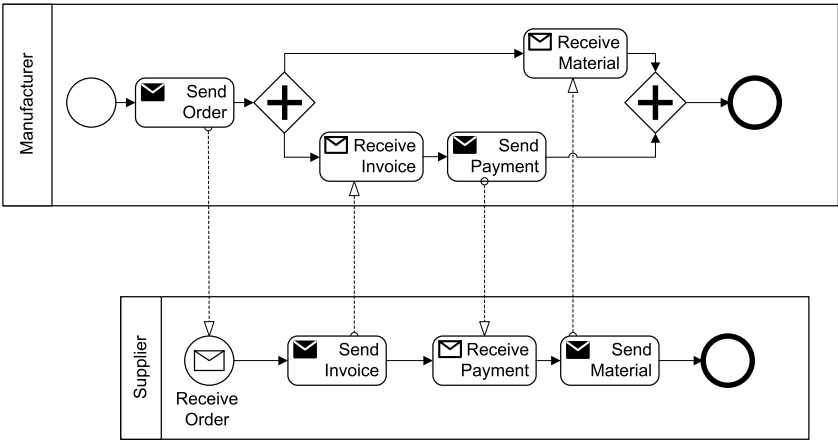




**Fig. 4.109.** Collaborating business processes with public process of the *Supplier*

The second advantage is based on business considerations. Business processes are a significant asset of a company, so that the company is not willing to expose its internal processes to the outside world. Since only the communication behaviour of a process can be observed from the outside, a process restricted to its communication activities is called *public process*.

We can also provide public processes for both communication partners. In this case, message flows are no longer associated with borders of pools, but with the actual send and receive tasks. This view provides details about the communication activities of both collaborating processes. To illustrate this, Figure 4.110 shows also the communication tasks of the manufacturer and their process flow.

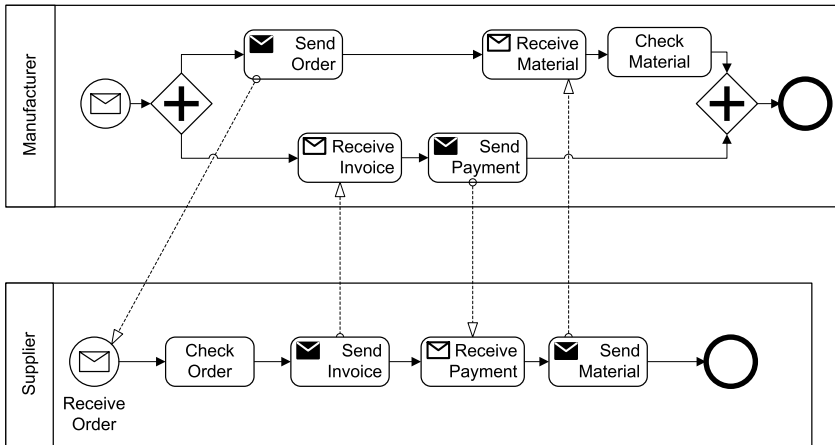


**Fig. 4.110.** Collaborating business processes with public processes of both partners

The process of the manufacturer starts by sending an order. The process continues with concurrent branches. In one branch the manufacturer waits for

the ordered material; in the other branch, it waits for receiving the invoice. After the invoice is received, the payment is sent.

A partner might also choose to expose its complete internal process. This is done by adding activities and potentially also control structures to its public process. The resulting process is called *private process*. A private business process contains all activities that are enacted within a company; it realizes a process orchestration.



**Fig. 4.111.** Collaborating business processes with private processes of both partners

A private business process of the manufacturer shown in Figure 4.111 contains an activity *Check Material*, so that the manufacturer can check the material after receiving it. This is a typical example of an activity that is executed in the process orchestration of a partner, but which has no implication on the externally visible behaviour of the process. Therefore, it is part of the private process, but not of its public process.

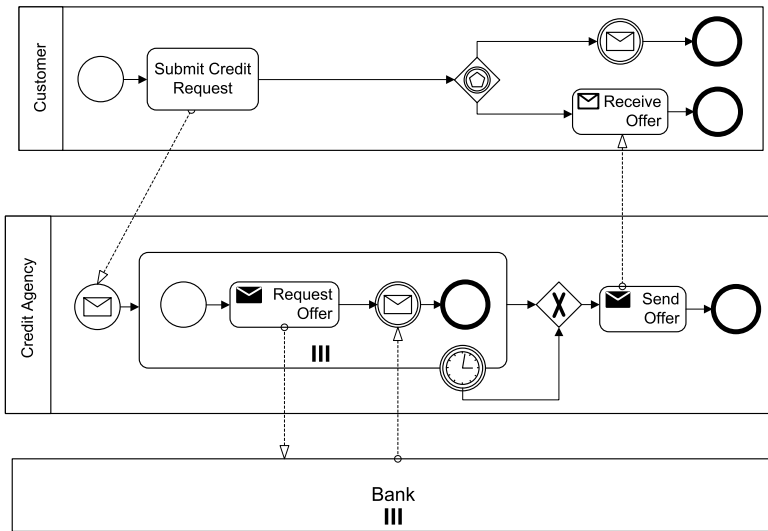
So far, each pool represented a single organization, for instance a concrete supplier or a concrete manufacturing company. The BPMN also provides means to express pools that represents multiple organizations that participate in the process collaboration.

An example involving multiple instance pools is given in Figure 4.112, where a credit request process is shown. There are three pools in this collaboration, a customer, a credit agency, and a bank. As indicated by the multiple instances marker in the bank pool, multiple banks participate in this collaboration, while only one customer and only one credit agency participate.

The process starts by the customer filling a credit request and sending it to the credit agency, spawning off a new process instance. The credit agency requests offers from several banks, represented by the multiple instances subprocess. In each instance of the subprocess, one offer message is sent to a

concrete bank, and a response is received from that bank. The subprocess instances are created concurrently, so that the requests are sent out concurrently and the respective messages are collected from the banks, as they come in.

The BPMN states that the number of multiple instances of a subprocess matches the number of instances of a multiple instances pool it communicates with. In our example, the number of subprocess instances that send the request messages and receive the response matches the number of banks.



**Fig. 4.112.** Collaborating processes with a multiple instances pool

The process continues as follows. Once all responses are collected or a timer elapses, an offer is selected and submitted to the customer. If the customer is still patiently waiting to receive this offer, it does so.

While the BPMN can graphically represent the interaction of business processes, there are no formal properties defined on the relationship between a business process and its externally visible behaviour. Correctness criteria for process choreographies that consist of a set of interacting business processes are also not part of the BPMN. These aspects will be discussed in the context of process choreographies in Chapter 5.

#### 4.7.4 Executability and Exchange Format

One of the points of critique regarding earlier versions of the BPMN was the lack of executable processes, which resulted in the need to translate BPMN diagrams to executable languages, like WS-BPEL. In the current version,

executability is addressed in BPMN, and first process engines that natively support that standard are available.

Maybe the most important aspect of the BPMN in its current version is the standardization of the exchange format. By providing XML Schema definitions for the standard, tool vendors can provide a serialization format for BPMN diagrams, so that process models can be exported from one tool to be imported in another tool. This is a very important feature, since it allows the automatic transfer of process models between tools that are rather on the domain aspect to tools that are focusing on executable processes.



<http://www.springer.com/978-3-642-28615-5>

Business Process Management  
Concepts, Languages, Architectures  
Weske, M.  
2012, XVI, 404 p., Hardcover  
ISBN: 978-3-642-28615-5