

At the top of the stack is the visual process design layer, based on BPMN. Below it is BPSM, in which BPMN visual models are represented in a common, interchangeable metamodel form, suitable for import into the execution layer based on the OASIS group's BPEL (discussed in [Chapter 5](#)), which is extended by BPXL. BPEL, in turn, runs atop a web services messaging and transport layer, whose major standards include the W3C's WSDL and the OASIS UDDI. BPEL shares the web services base with the standard BAM query service BPQL, as well as the W3C's WS-CDL choreography (discussed in [Chapter 8](#)). BPQL enables business monitoring of BPEL processes, and WS-CDL defines the global contract governing the partner interactions of BPEL processes.

This stack is intriguing for several reasons:

- Not all pieces are based on BPMI standards. (The BPMI pieces are shaded.) BPMI embraces standards—such as BPEL, WS-CDL, and the core web services standards—of other organizations.
- Some of the BPMI pieces (dotted in the figure) are not currently published. If the stack were to be implemented today, the implementers would need to find suitable substitutes for BPSM, BPXL, and BPQL.
- BPML is nowhere to be found! Its place—the web service-based execution spot—seems to have to been stolen by BPEL. The BPMI position paper^[*] concedes that the BPEL standard, BPML's most formidable competitor, has won the XML execution language war, and by virtue of "might makes right" is a better fit than BPML for the stack.

[*] BPMI, "BPMN and BPEL4WS: A Convergence Path Toward a Standard BPM Stack," <http://www.bpmi.org>, August 2002.

- The same BPMI position paper recommended Web Services Choreography Interface (WSCI) as the choreography piece, but since then BPMI has adopted WS-CDL, the W3C's official approach.

This chapter describes BPMN and BPML in detail and introduces the main aspects of each language through several feature-rich examples; each language is also rated on its support for the P4 patterns introduced in [Chapter 4](#).^[*]

[*] Interestingly, the BPMN specification provides a BPEL mapping, which facilitates BPEL XML representation of BPMN diagrams; this mapping is explored at a high level later in the chapter. The BPMN specification makes no mention of BPML, though one would expect a BPMN-to-BPML mapping to resemble the BPMN-to-BPEL mapping.

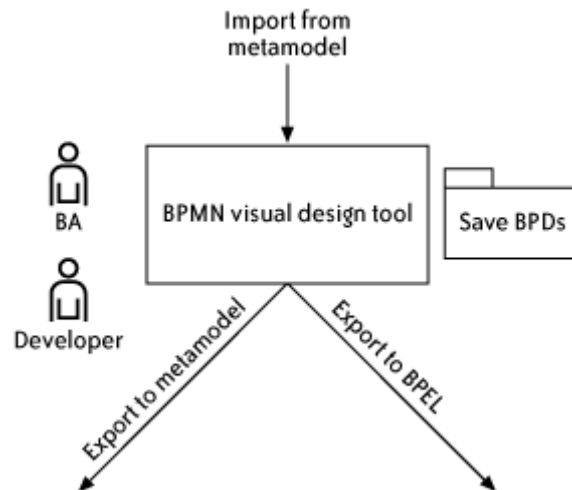
6.1. BPMN

BPMN is a graphical flowchart-like language intended for use by business analysts and developers to build business process diagrams (BPDs). A BPD conveys in pictures what BPML and BPEL encode in XML, but it serves a different purpose: BPMN is for graphical design, whereas BPML and BPEL are for execution. The BPMN specification^[†] attempts to bridge the gap by providing a mapping from BPMN to BPEL (but not, interestingly, to BPML); the mapping specifies rules to generate BPEL from a BPD, enabling the execution of a BPD. [Figure 6-2](#) shows how a typical BPMN tool is used in the design process.

[†] S. White, "Business Process Modeling Notation," Version 1.0. <http://www.bpmi.org>, May 2004.

In addition to BPEL export, the tool also supports BPSM metamodel import and export, allowing the BPMN tool to exchange processes with those developed in other tools. The message broker example developed in [Chapter 11](#) uses ITpearls' MS Visio-based BPMN tool for the design of message broker processes. ITpearls, alas, does not currently include any of these import and export features.

Figure 6-2. Use of BPMN



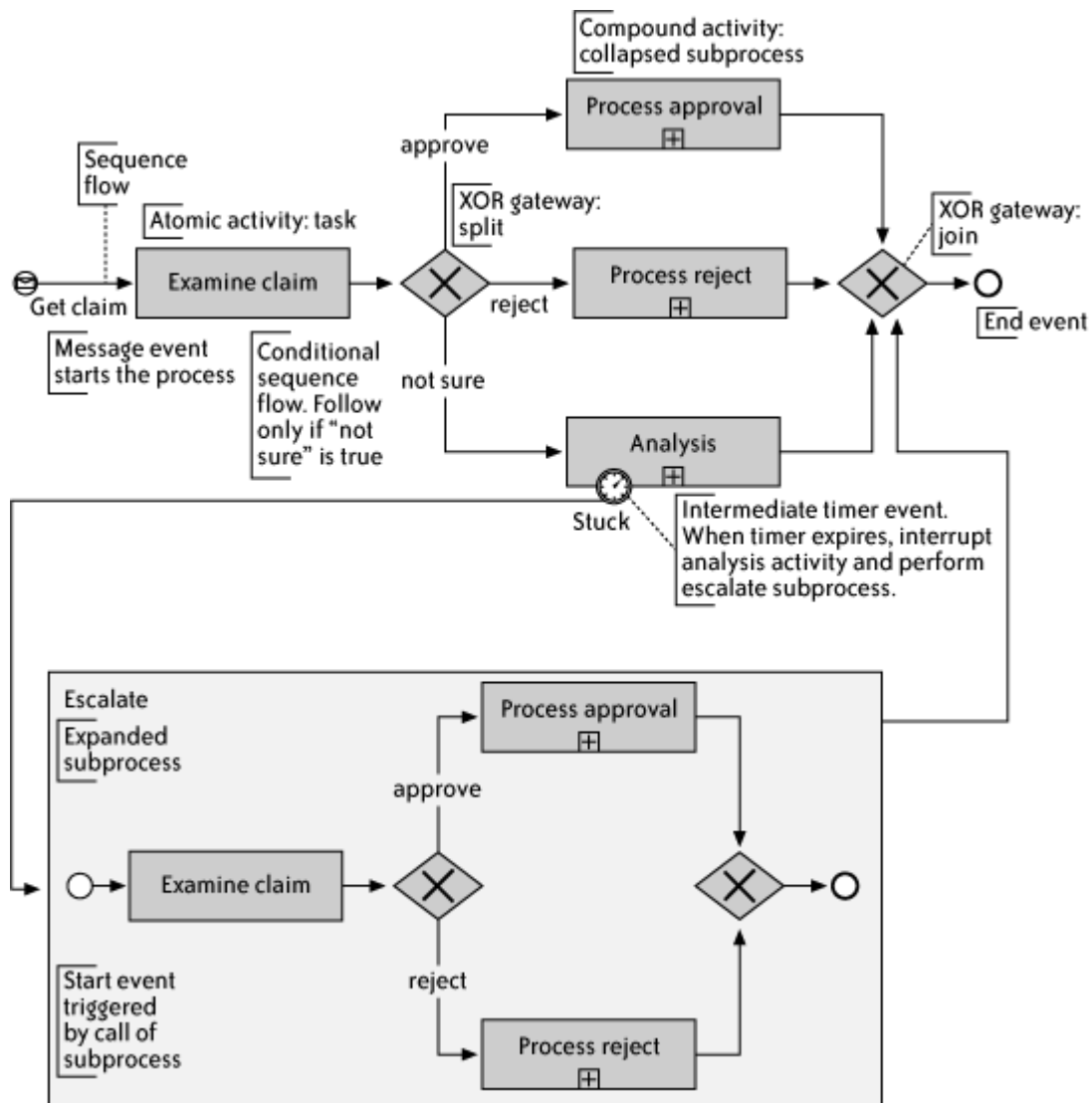
6.1.1. BPMN Example

The first step in learning a new language is to look at the implementation of "Hello, World!" In our case, we study "Hello, Claim!," which is an insurance claims handling process and is shown in [Figure 6-3](#).

The process receives a claim (*Get claim*), examines it (*Examine claim*), and then splits into one of three directions, depending on whether the claim has been approved (*Process approval*), rejected (*Process reject*), or passed along for further analysis (*Analysis*). The analysis option has a time limit; if it is not performed quickly enough, it is aborted (*Stuck*), and a special escalation process (*Escalate*, whose steps are enclosed in the *Escalate* box) is run. The escalation process behaves much like its parent: it begins by examining the claim, then either approves or rejects it; no further analysis is permitted in an escalation. The parent process completes when its conditional path—approval, rejection, analysis, or escalation—completes.

Several types of symbols are used in this diagram: events, gateways, atomic activities (or tasks), compound activities (or subprocesses), sequence flow, and text annotations. Events, drawn as small circles, mark the start (e.g., *Get claim*) and end points of the process, as well as the intermediate timeout condition (*Stuck*) that occurs during analysis. Gateways (diamonds) help mark the conditional split and join portion of the process. Activities (boxes) represent actual work performed. Tasks (e.g., *Examine claim*) are single actions, whereas subprocesses perform arbitrarily complex logic. A subprocess can be drawn either collapsed or expanded; a collapsed process (e.g., *Process reject*) is drawn with a plus sign, its details hidden, but assumed to be documented in another diagram; an expanded process (e.g., *Escalate*) has its internal logic drawn inside of it. Sequence flow is the set of arrows connecting together the other pieces; arrows labeled with text (e.g., arrow between gateway and *Process approval*) are conditional, followed only if the condition is true. Text annotations (open-ended boxes) present instructional comments.

Figure 6-3. BPMN insurance claims process



6.1.2. BPMN in a Nutshell

This section examines the essential language constructs that designers need to understand in order to create a BPMN process: the basic process structure, variables and assignments, exception handling and compensation, split and join, loops, participant exchange, transactions, and extensions. Before delving into the details of the language, we'll first introduce the basic elements in a BPMN process—events, activities, sequence flows, and gateways—in a bit more detail.

An *event*, the first basic element of BPMN, is an occurrence that triggers a business process. Events are categorized by the stage at which they occur in a process (start, intermediate, or end) and by type (basic, message, timer, rule, exception, cancellation, compensation, link, multiple, or termination). The shape of an event is a small circle; a start event has a thin border, an end event a thick border, and an intermediate event has a double border. Figure 6-4 illustrates the complete set of events and how they are depicted.

Table 6-1 describes the role of events in more detail.

Figure 6-4. BPMN events
























Start	Intermediate	End	Name
			Basic
			Message
			Timer
			Rule
			Exception
			Cancellation
			Compensation
			Link
			Multiple
			Termination

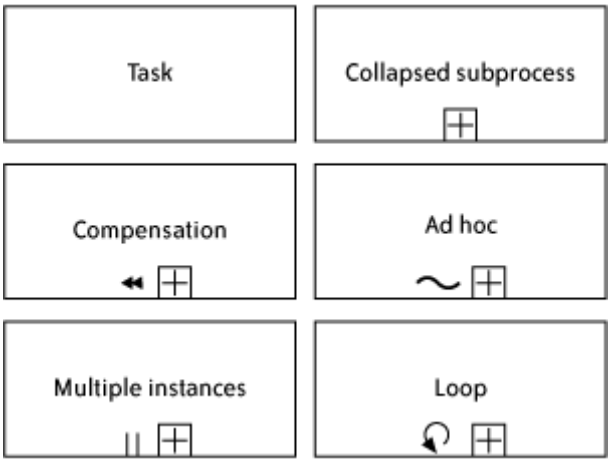
Table 6-1. BPMN event description

Type	Start	Intermediate	End
Basic	Placeholder event or the start of a called subprocess.	Placeholder	Placeholder or end of a subprocess.
Message	Process is started by receipt of a message (e.g., the invocation of a web service method implemented by the process).	Process is waiting for a message (e.g., wait for response from a participant to which this process has sent a request).	A message is to be sent to a participant process (e.g., call its web service).
Timer	The start event defines a schedule for when it triggers (e.g., every Tuesday at midnight).	A point in a defined schedule has been reached.	
Rule	A condition, defined by the process, is met (e.g., process starts when a stock's price hits its 52-week high).	A condition is met. Used only for exception handling.	
Exception		Throw or catch an error.	Generate an error.
Cancellation		Perform cancellation for a given activity.	Cancel the transaction.
Compensation		Trigger and perform compensation handling.	Perform compensating action.
Link	The link start event connects to the link end event of a sibling process.	Link to or from another activity.	Connect to the link start of a sibling process.
Multiple	Two or more triggers can start the process; if any one of them occurs, the process starts. These triggers can be message, timer, rule or link types.	Two or more triggers can continue a waiting process; if any one of them occurs, the process resumes.	When the process ends, several results are required (e.g. several messages need to be sent).
Termination			Terminate all activities in the process. Perform no exception handling or compensation.

An *activity*, the second basic element of BPMN, is a step in a process that performs work. In BPMN, an activity is either atomic or compound. An atomic activity, also known as a task, performs a single action. A compound activity, also known as a process, has its own set of atomic or compound activities, as well events, gateways, and all other BPMN constructs. Processes are hierarchical: a process can have subprocesses, each of which can have subprocesses, and so on.

An activity is drawn as a box with rounded edges. When shown in a parent process, a child process is drawn as a single box bearing a plus sign (+). The plus sign represents the collapsed view of the process; the full detail is drawn in a separate diagram. Any activity—task or process—can be marked up with symbols representing compensation, multiple instances, and loops; additionally, a process can have the markup symbol tilde (~) for ad hoc processing. These possibilities are depicted in [Figure 6-5](#).

Figure 6-5. BPMN activities



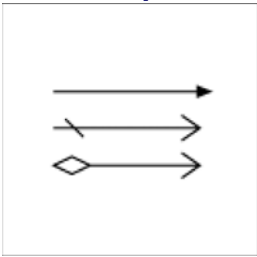
A compensated activity is one that has special compensation logic to revert it (undo its effect) after completion. An ad hoc process contains a set of activities that can occur in any order; the control flow is unstructured. Loop and multiple instance activities are described in the later section "Loops." Table 6-2 summarizes the BPMN specification's suggested task types.

Table 6-2. BPMN task types

Use	Description
Service	Calls a web service
Receive	Waits for a message (an alternative to an event construct)
Send	Sends a message
User, Manual	Task is performed by a human participant (e.g., approval)
Script	Logic encoded in a programming or scripting language (e.g., run a piece of Java code)
Reference	Uses the definition of another task in the process; shares the definition rather than duplicating it

Sequence flow, the third basic element of BPMN, is the flow of control in a process, and is represented by arrows connecting source and target activities, events, or gateways. Figure 6-6 shows the three types of sequence flow arrows.

Figure 6-6. BPMN sequence flow arrows

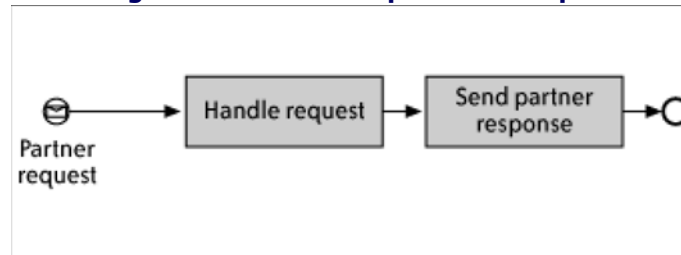


The first arrow represents normal, unguarded flow from source to target. The second symbol is default flow, used in cases where control splits into multiple directions, each path depending on the evaluation of a condition; it fires only if none of the other guarded transitions fired. The third is a guarded transition, traversed only if its associated conditional expression evaluates to true; the diamond at the end of the arrow is not required when the transition originates from an XOR split.

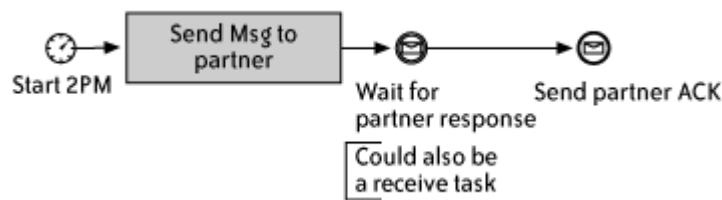
A gateway, the final basic BPMN element, is a special controller of splits and joins. This element is discussed in more depth in the later section "Split and join."

6.1.2.1. Basic process structure: start, end, activities, sequence

A basic BPMN process has a start event , one or more activities, and an end event. The process in Figure 6-7, for example, starts with a message event that receives a partner request, and then executes activities to handle the request and send a response to the partner, before closing with a basic end event.

Figure 6-7. BPMN sequence example

Besides activities, intermediate events can also be key steps in the mainline sequence of a process. A typical example is the process shown in [Figure 6-8](#), which sends a message to a partner application and then needs to wait for response before continuing. This example also shows that the end event can perform useful work; in this case, sending an acknowledgment message to the partner.

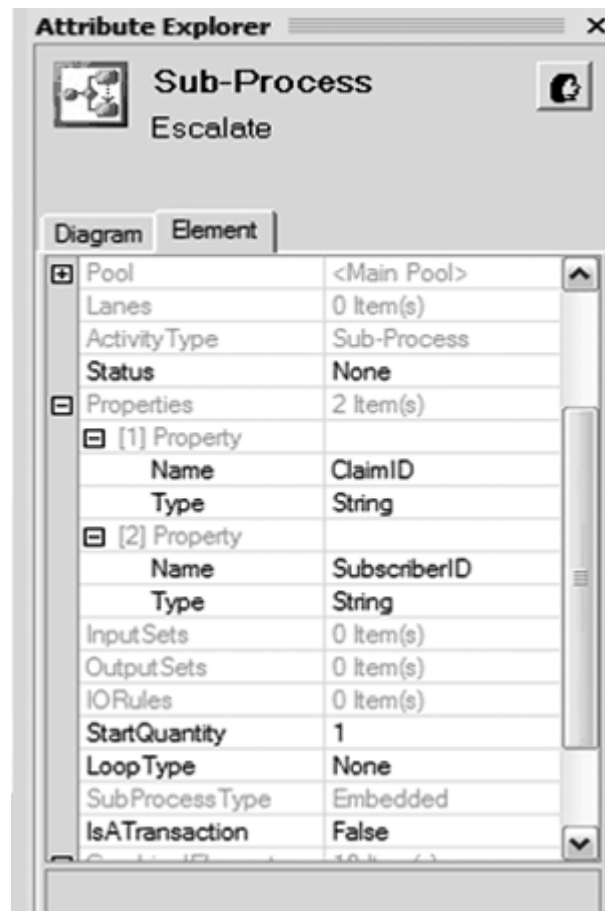
Figure 6-8. BPMN sequence example with intermediate event

6.1.2.2. Variables and assignments

In BPMN, processes and activities can have variables (known as properties), assign values to them, and make decisions based on their values. Though variables are not shown graphically in a BPD, BPMN includes them in its object model, chiefly to facilitate a mapping to BPEL.

Most BPMN editors, including ITpearls, provide an attribute editor to manipulate variables and other data associated with processes, activities, or other graphical nodes. [Figure 6-9](#) shows how two `String` type properties—`ClaimID` and `SubscriberID`—are defined for the `Escalate` subprocess.

Figure 6-9. BPMN process properties in ITpearls' attribute editor

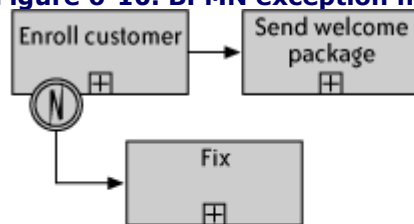


For the mechanics of variable usage, consult the BPMN specification.

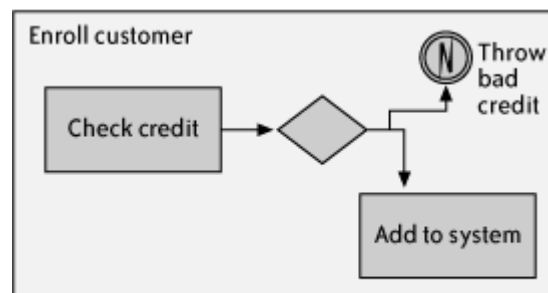
6.1.2.3. Exception handling and compensation

Figure 6-10 illustrates the BPMN approach to exception handling.

Figure 6-10. BPMN exception handling



(a) Exception in Enroll customer



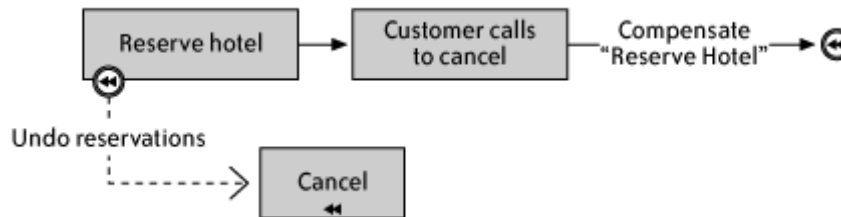
(b) Closeup of Enroll customer subprocess

Part (a) shows the catching and handling of an exception in the subprocess `Enroll customer`. When `Enroll customer` completes normally, it transitions to `Send welcome package`. But if an exception occurs during its

execution, the intermediate error event on the boundary of `Enroll customer` catches the error and passes control to the subprocess `Fix`, which serves as a fault handler for `Enroll customer`. Part (b) shows that an intermediate error event, when not on the boundary of an activity, effectively throws an exception, which triggers exception handling of the parent process. The event `Throw bad credit` breaks the normal flow of the parent process, `Enroll customer`, and precipitates the handler in `Fix` from part (a).

In compensation, an activity is run to reverse the effects of another activity. For example, in Figure 6-11, `Cancel` compensates `Reserve hotel`.

Figure 6-11. BPMN compensation



The notation is to place a compensation intermediate event (resembling a "rewind" symbol) on the boundary of the activity to be compensated, draw a dotted arrow (known as an "associated" in BPMN parlance) from the compensation event to the boundary of the compensating activity, and mark a compensation symbol inside the boundary of the compensating activity. The compensating activity must be self-contained; it cannot have any inbound or outbound sequence flow connections. The job of a compensating activity is strictly to perform the required reversal logic.

Compensation is triggered in one of two ways:

- With an explicit compensation event, as with the event `Compensate "Reserve Hotel"` shown earlier.
- If the activity to be compensated is part of transaction subprocess that is cancelled. This scenario is discussed in the later section "[Transactions](#)."

Compensation can apply to transactional and nontransactional activities alike. For transactional activities, compensation is not the same as rollback. Only a completed activity can be compensated; if that activity is transactional, because it has completed, its transaction has already committed.

6.1.2.4. Split and join

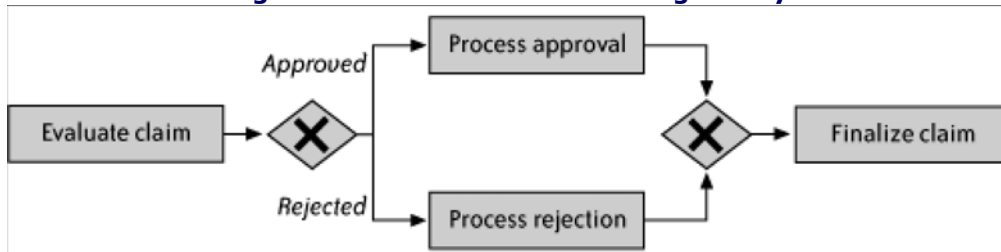
BPMN uses the *gateway* element to model split and join patterns, which represent common programming control structures such as `if-then`, `switch`, and `all`. A gateway branches and merges paths in a process. The diamond-shaped symbol is well known in flowchart languages as a decision point, but BPMN expands its use to model patterns such as AND split and join and deferred choice. Furthermore, in BPMN a gateway has two modes: it splits one incoming path into multiple outgoing paths (which we will refer to as split mode), and merges several incoming paths into one outgoing path (join mode). The BPMN gateway symbols are shown in Figure 6-12.

Figure 6-12. BPMN gateways

Symbol	Name
	Exclusive OR
	Exclusive OR
	Exclusive OR (Event-based)
	Exclusive OR
	Complex
	Parallel

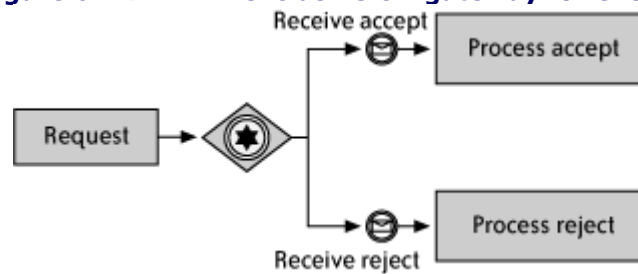
The first gateway, the *exclusive OR*, uses *if-then-else* and *switch* with mutually exclusive cases as a control structure. In split mode, it evaluates a separate condition on each of its outgoing paths and lets through the first path whose condition evaluates to true; all others are ignored. Exactly one condition must be true; a default branch can be specified in case none of the other branches fire. In join mode, the exclusive OR gateway lets through the first of its multiple incoming branches and discards all others. Figure 6-13 illustrates the behavior by showing an activity (*Finalize claim*) that runs when either *Process approval* or *Process rejection* completes, and an activity (*Evaluate claim*) that splits to *Process approval* if the condition *approved* is satisfied and to *Process rejection* otherwise.

Figure 6-13. BPMN exclusive OR gateway



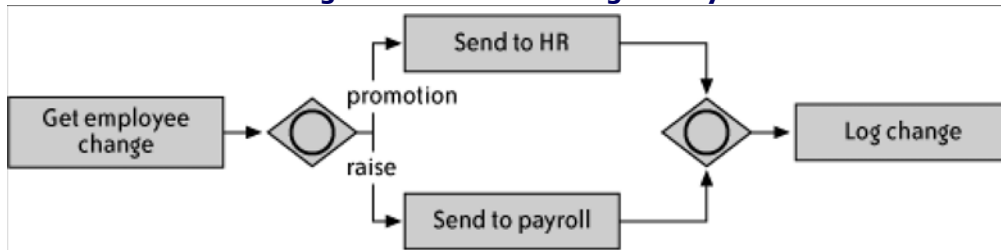
The second gateway, the *exclusive OR (event-based)*, uses a *pick* control structure. In split mode, each outgoing branch leads to an event node. The gateway lets through the branch having the first triggered event, and ignores all others. The join mode is not commonly used. For example, in the process in Figure 6-14, when the activity *Request* completes, the process waits for one of the two events—*Receive accept* or *Receive reject*—to occur.

Figure 6-14. BPMN exclusive OR gateway for events



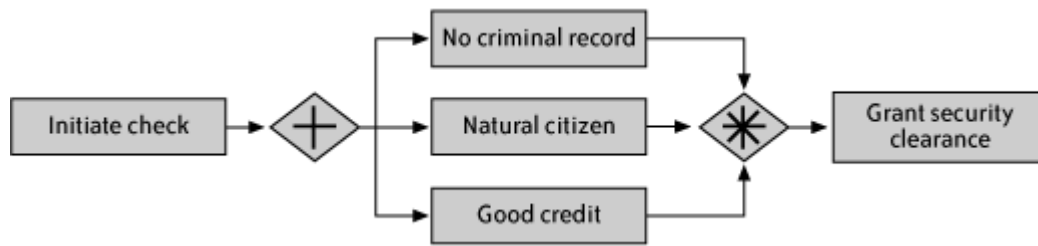
The third gateway, the *inclusive OR*, uses a *switch* with overlapping cases as a control structure. The split mode is similar to exclusive OR but lets through each outgoing path whose condition evaluates to true. The join mode blocks passage until each expected executing incoming path enters it. The gateway knows in advance how many active inputs to expect. Figure 6-15 illustrates both behaviors: activity *Get employee change* splits to *Send to HR* or *Send to payroll* or both, depending on the evaluation of conditions *promotions* and *raise*; activity *Log change* waits until *Send to HR* or *Send to payroll* or both of these complete, depending on which paths the splitting gateway let through.

Figure 6-15. BPMN OR gateway



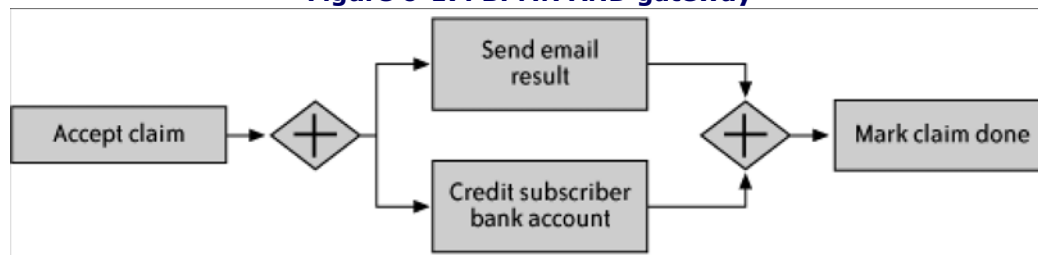
The fourth gateway, *complex*, uses a control structure that is quite unique to BPMN. The split mode is not commonly used. The join mode evaluates an expression to determine which of the incoming paths to let through. As an example, in Figure 6-16 the gateway waits for two of the three parallel activities—*Good credit*, *Natural citizen*, and *No criminal record*—before granting security clearance.

Figure 6-16. BPMN complex gateway



The final type, the *parallel* gateway, uses `all` control structures in BPML and `flow` in BPEL. In split mode, it lets through each outgoing path. In join mode, it blocks until each incoming path completes. Figure 6-17 illustrates these two behaviors: when activity `Accept claim` completes, the activities `Send email result` and `Credit subscriber bank account` are run in parallel; `Mark claim done`, however, starts only when both `Send email result` and `Credit subscriber bank account` complete.

Figure 6-17. BPMN AND gateway



NOTE

The BPMN specification is laden with references to token passing, betraying its dependency on the ideas of the Petri net. For example:

- A parallel gateway sends a token for each of its outgoing arrows. A parallel gateway waits for a token on each of its incoming arrows.
- An exclusive gateway sends a token on exactly one of its outgoing arrows, namely the one whose condition is true. An exclusive gateway waits for exactly one token from its incoming arrows.
- When multiple arrows converge on an activity without first passing through a gateway, each token that comes through will trigger the activity.
- When multiple arrows emanate from a single activity (known as "uncontrolled" flow in BPMN), a token is generated on each arrow.

6.1.2.5. Loops

BPMN's approach to looping is powerful but obscure. In most process languages, a loop is a specific type of compound activity that iterates over the set of activities inside of it. For example, the BPEL `while` loop in the following code sample repeats a sequence of `invoke` activities (A and B) for as long as its specified condition evaluates to true:

```

<while condition=". . .">
  <sequence>
    <invoke name="A" . . . />
    <invoke name="B" . . . />
  </sequence>
</while>

```

In BPMN, looping is an attribute of an activity. To make an activity loop, simply play with the attributes of the activity, and it will loop as directed. If BPEL were designed this way, its code would resemble the following, in which the sequence activity itself controls whether and how to loop:

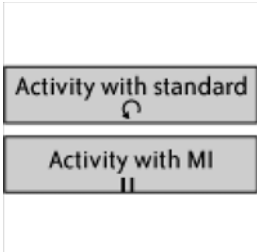
```

<sequence looping="true" loopcondition=". . .">
  <invoke name="A" . . . />
  <invoke name="B" . . . />
</sequence>

```

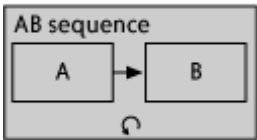
In BPMN, any activity (either a task or a subprocess) can be configured either to not loop, to loop in standard mode (`while` or `until` loops), or to support multiple instances (`foreach` looping). The notation for standard and multiple instance (MI) loops is shown in [Figure 6-18](#). (An activity with standard looping has a circular arrow mark in the bottom center of its box; an activity with MI has a pair of vertical parallel bars.)

Figure 6-18. BPMN notation for standard and multiple instance (MI) loops



[Figure 6-19](#) shows the BPMN representation of the BPEL `while` loop: activities A and B run sequentially in a process `AB sequence` that is configured with standard looping.

Figure 6-19. BPMN while loop for a sequence of activities



[Table 6-3](#) summarizes the settings for a standard loop.

Table 6-3. Standard loop settings

Setting	Description
Condition	An expression that determines whether to continue looping.
Test time: before, after	When to test the condition: before the activity is run, or after it is run. In the former case, the loop will behave as a <code>while</code> ; in the latter, as an <code>until</code> .
maxLoops	An upper boundary on the number of iterations.
Loop counter	Used internally. Starts at zero, is incremented by one for each iteration, and is compared with maxLoops.

If the test time is set to `before`, the logic of the loop is the following:

```
While (cond and loopCounter < maxLoops)
  Perform Activity
```

Otherwise, the logic of the loop is the following:

```
Do
  Perform Activity
Until (cond and loopCounter >= maxLoops)
```

In contrast to the standard loop, the MI loop is rather complicated. [Table 6-4](#) summarizes the MI loop settings.

Table 6-4. MI loop settings

Setting	Description
MI condition	An expression that determines the number of instances to run.

Setting	Description
Loop counter	Used internally as loop counter.
Ordering: sequential, parallel	Determines whether the instances are run sequentially or in parallel.
Flow condition: none, one, all, complex	Used only if the order is parallel. <code>none</code> means that as soon as each instance of the activity is executed, the next activity in the process is executed. <code>one</code> means that the next activity is executed only after the first instance completes; subsequent iterations execute, but do not continue onto the next activity. <code>all</code> means that the next activity executes only once all instances have completed. <code>complex</code> means that the "complex condition" expression must be evaluated to determine the rule for how to handle the next activity; <code>complex</code> can be used to model patterns such as N-out-of-M join.
Complex condition	A condition that determines when and how many times to execute the next activity.

Sequential processing is simple:

```
For counter = 0 to MI condition
  Perform activity
Perform next activity
```

Parallel processing with a flow condition of `all` resembles the following:

```
For counter = 0 to MI condition
  Spawn activity
Wait for all activities; when an activity completes, do nothing
Perform next activity
```

Parallel processing with a flow condition of `one` resembles the following:

```
For counter = 0 to MI condition
  Spawn activity
Wait for one activity; when it completes, perform the next activity
```

Parallel processing with a flow condition of `none` resembles the following:

```
For counter = 0 to MI condition
  Spawn activity
Wait for all activities; for each completed activity, perform next activity
```

Parallel processing with a flow condition of `complex` resembles the following:

```
For counter = 0 to MI condition
  Spawn activity
Wait for all activities
For each completed activity
  If complex condition says perform next activity, do so
```

Figure 6-20 shows scenarios in which each type of looping would be used.

Figure 6-20. BPMN loop scenarios

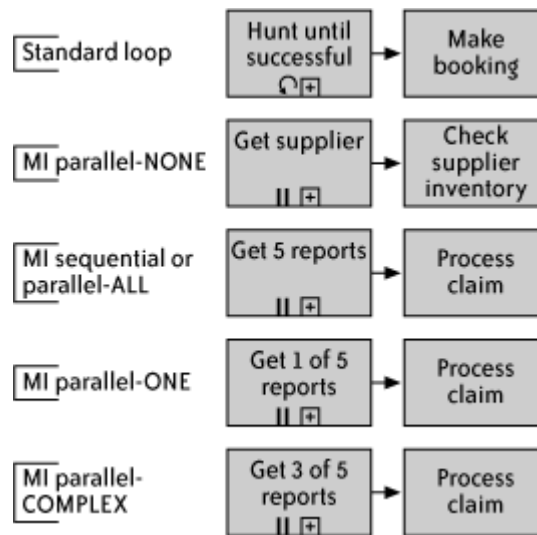


Table 6-5 summarizes each of these scenarios.

Table 6-5. Description of BPMN loop scenarios

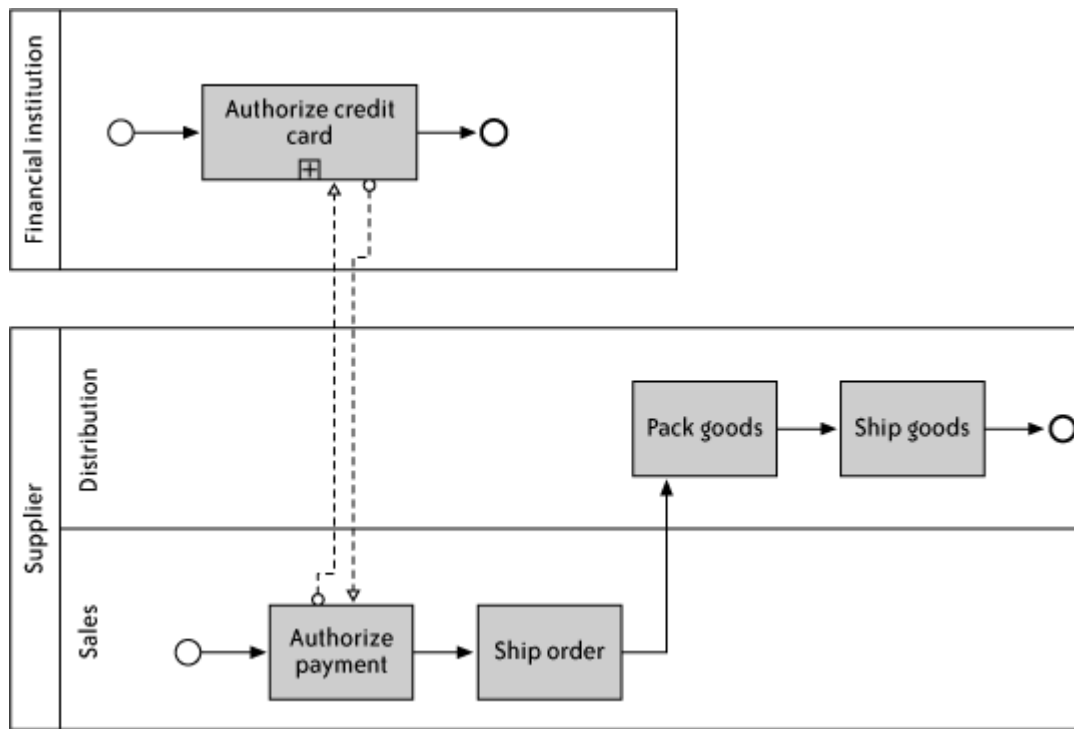
Loop type	Description
Standard	Hunt airline schedules for an available flight. When one is found, book it.
MI parallel-NONE	Loop through a list of suppliers. For each supplier, query its inventory.
MI sequential or parallel-ALL	For the investigation of an insurance claim, take reports from five witnesses (either sequentially or in parallel). When all five have completed, process the claim.
MI parallel-ONE	For the investigation of an insurance claim, take reports from five witnesses in parallel. As soon as one completes, process the claim, but in the meantime, let the others complete too.
MI parallel-COMPLEX	Similar to the previous case, but do not process the claim until three reports have completed.

6.1.2.6. Participant exchange

BPMN provides a rich framework for modeling interparticipant processing, which includes swim lanes and pools, message flow, message events, send and receive tasks, and message correlation.

A *swim lane* is a pool and each of its lanes. A *pool* represents the activities of one participant—often a company—in collaboration; a lane in a pool represents a subdivision of the participant—often a department or division of the company. Swim lanes help convey the sense that a process spans multiple participants; it depicts who does what and how the interactions are structured. For example, consider the collaboration of a supplier and a financial institution. The supplier calls the financial institution to authorize payment. The supplier, in turn, is divided into sales and distribution departments, which manage different parts of the supplier's process. Figure 6-21 illustrates this scenario.

Figure 6-21. BPMN swim lane, adapted from BPMN specification, V.1, p.104



Message flow, symbolized by a dashed arrow, such as the arrow between supplier and financial institution in [Figure 6-21](#), shows the flow of messages, or the data flow, between participants. The solid arrows of sequence flow, by contrast, capture process flow, or inter-activity control movement.

NOTE

The BPMN authors favor showing both kinds of flow in the same diagram, but this practice has drawbacks, the most obviously of which is clutter. Showing both types of flow is also common in UML activity diagrams.

Message events and send and receive tasks are also supported. [Table 6-6](#) summarizes the five main BPMN message-exchange process elements and their BPEL mapping.

Table 6-6. BPMN message exchange elements

Type	Meaning	BPEL equivalent	From/to participant
Start message event	Inbound web service	Receive and create instance	From
Intermediate message event	Inbound web service	Receive	From
End message event	Outbound web service	Invoke or reply	To
Send task	Outbound web service	Invoke	To
Receive task	Inbound web service	Receive	From

BPMN's support for *message correlation* (which allows a process to determine whether a given inbound message, based on key data, belongs to its conversation) is something of an afterthought. Specifically, BPMN allows a process or activity property to be designated as a correlation set. Such a property can have child properties that represent members of the correlation set. Refer to the BPMN specification for the mechanics of the approach.

6.1.2.7. Transactions

BPMN also supports the notion of the *transaction*, in which a subprocess can be marked to allow it to be executed as a single unit of work. If the subprocess reaches its end point successfully, it is committed. If it receives a cancellation event, the transaction is rolled back, compensation is applied to any subactivities that require it, and a special cancellation handler is executed. A transactional subprocess is drawn with a thick border line. The cancellation handler is connected by a sequence flow to an intermediate cancel event on the border of the subprocess.

As an example, in [Figure 6-22](#), if a cancellation occurs in the transactional subprocess *Invite to conference*, the

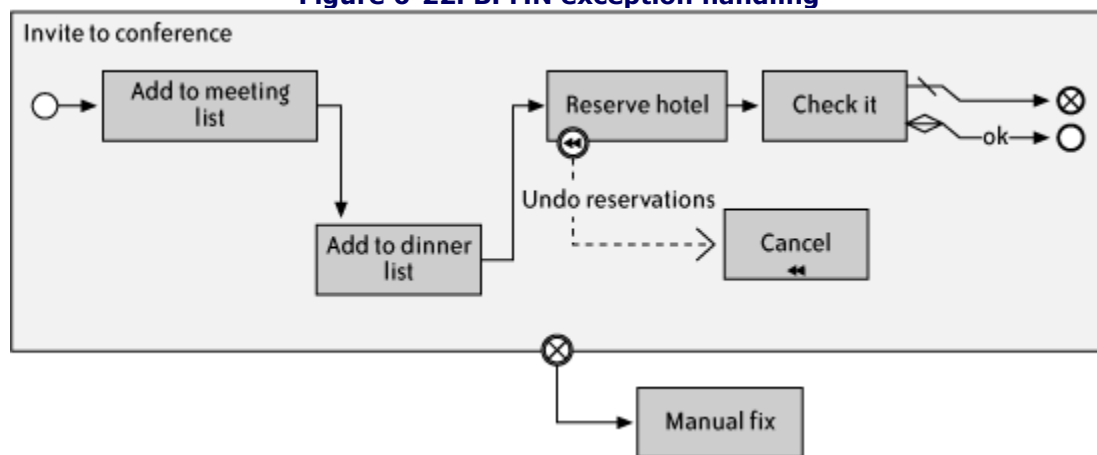
updates in `Add to meeting list` and `Add to dinner list` are rolled back, the `Reserve hotel` activity is compensated, and the cancellation handler `Manual fix` is executed.

PRIVATE, ABSTRACT, AND COLLABORATIVE PROCESSES IN BPMN

BPMN supports three types of processes:

- Private (internal), representing the full orchestration logic of particular participant. Private processes can be used to model BPEL executable processes.
- Abstract (public), representing the publicly observable interparticipant exchange of a particular participant. Abstract processes can be used to model BPEL abstract processes .
- Collaborative (global), which show the overall public exchange between a set of participants. These processes are similar to the collaborations of BPSS (see [Chapter 9](#)) or WS-CDL ([Chapter 8](#)).

Figure 6-22. BPMN exception handling



A cancellation can be triggered either explicitly, by a cancellation end event in the subprocess, or implicitly, by a cancellation message from the transaction manager of the engine. The use of a cancellation event is shown in [Figure 6-22](#): when the `Check it` activity is executed, it transitions to one of two end events: a normal end, if the variable `ok` evaluates to true, or a cancellation event, which cancels the subprocess.

6.1.2.8. Extensions

BPMN can be extended in two ways: by adding new symbols or by modifying existing symbols. Considering the importance of BPMN's mapping to BPEL (discussed in the next section), a key BPMN extension follows the BPEL extension of BPEL by providing Java variables and expressions, Java-based participants, and Java-based message events and send and receive tasks. These changes would be noninvasive, affecting only the behind-the-scenes attributes of core BPMN constructs, with no impact to the visual representation.

6.1.3. BPEL Mapping

The BPMN specification includes a 64-page chapter on BPEL mapping , which bridges the gap between graphical design and executability. BPMN diagrams are of little consequence unless they can actually be deployed and run. The BPEL mapping allows the generation of BPEL XML from BPMN diagrams, thus making it possible to run BPMN on BPEL engines. Significantly, the BPMN specification omits, and makes no mention of, a mapping to BPML; the BPMI, as described previously, places BPEL on its stack in place of BPML.

The details of the mapping are difficult. This section is a very high-level overview; you should consult the BPMN specification for the complete picture. BPEL has fewer constructs than BPMN, which complicates the mapping. One of the weaknesses of BPMN is its excess of features (e.g., multiple instance activities and the complex gateway); this excess makes BPMN extremely expressive, but a chore to fit into the BPM stack. BPSM, described at the beginning of this chapter, might help close the gap by providing a common metamodel for these languages. [Table 6-7](#) presents the highlights of the mapping presented in the BPMN specification.

Table 6-7. BPMN mapping to BPEL

BPMN	BPEL
Start event: all except Multiple	<code>receive</code> element with <code>createInstance</code> set to <code>yes</code> .
Start event: Multiple	<code>pick</code> element with <code>createInstance</code> set to <code>yes</code> .
End event: Message	<code>invoke</code> or <code>reply</code> element.
End event: Error	<code>throw</code> element.
End event: Compensation	<code>compensate</code> element.
End event: Terminate	<code>terminate</code> element.
End event: Link	<code>invoke</code> element.
Intermediate event: Error	Error handler if on the boundary of an activity, throw otherwise.
Activities: MI	Not a direct mapping. Use a variety of constructs.
Activities: Subprocesses	Use <code>invoke</code> to spawn. Embedded subprocesses implemented in <code>scope</code> .
Gateway: Exclusive data-based	<code>switch</code> element.
Gateway: Exclusive event-based	<code>pick</code> element.
Gateway: Inclusive	Multiple <code>switch</code> elements within a <code>flow</code> element.
Gateway: Complex	No obvious mapping.
Gateway: Parallel	<code>flow</code> element.
Sequence Flow	Model the flow of control from one node to another either explicitly with flow <code>link</code> elements or implicitly with control structures such a <code>sequence</code> , <code>while</code> , or <code>switch</code> .
Message Flow	No mapping.

6.1.4. BPMN and Patterns

BPMN is designed to implement most of the P4 patterns discussed in [Chapter 4](#). White's paper^[*] describes the BPMN implementation for each of the patterns. [Table 6-8](#) summarizes White's findings.

[*] Stephen White, "Process Modeling Notations and Workflow Patterns," *BPTrends*, March 2004.

Table 6-8. BPMN support for the P4 patterns

Pattern	Compliance Approach	
	(+, -, +-)	
Sequence	+	Normal sequence flow
Parallel Split	+	Parallel gateway as splitter
Synchronization	+	Parallel gateway as joiner
Exclusive Choice	+	Exclusive gateway as splitter
Simple Merge	+	Exclusive gateway as joiner
Multi-Choice	+	Inclusive gateway as splitter
Sync Merge	+	Inclusive gateway as joiner
Multi-Merge	+	Uncontrolled split and join

Pattern	Compliance Approach (+, -, +-)	
Discriminator	+	Complex gateway as joiner to accept one out of M or N out of M incoming paths
Arbitrary cycles	+	Sequence flow allows loops
Implicit Termination	+	Multiple end events allowed; first to trigger ends process
Multiple Instances (MI) Without Synchronization	+	MI marker on activity
MI With Design Time Knowledge	+	Use MI
MI With Runtime Knowledge	+-	More advanced use of MI
MI Without Runtime Knowledge	-	Relatively difficult coding
Deferred Choice	+	Exclusive data-based gateway as splitter
Interleaved Parallel Routing	+	Yes! Ad hoc process
Milestone	+	
Cancel Activity	+	Exception handling
Cancel Case	+	Exception handling or implicit termination
