



BPMN and the Business Process Expert, Part 5: Handling Errors and Business Exceptions

Summary: If 80% of the cost and problems with business processes comes from 20% of the instances – the exceptions – shouldn't business have a role in specifying exception handling? BPMN supports that view. Here we'll learn the diagram patterns to use in the most common exception-handling scenarios. Fifth of six parts.

Author: Bruce Silver

Company: Bruce Silver Associates

Created on: 17 December 2007

Author Bio



Dr Bruce Silver is an independent industry analyst and consultant focused on business process management software. He provides training on process modeling with BPMN through BPMessentials.com, the BPM Institute, and Gartner conferences, and is the author of The BPMS Report series of product evaluations available from the BPM Institute.

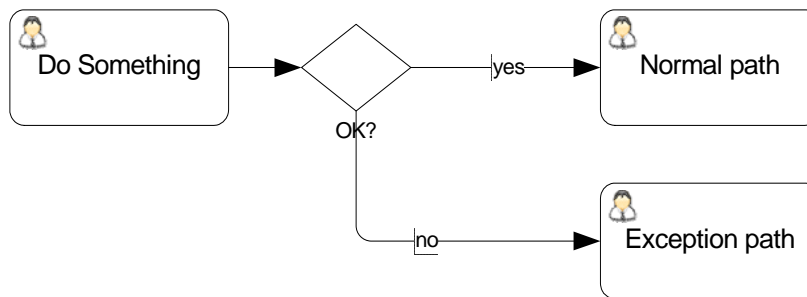
When you begin modeling your process, it is customary to begin with the “happy path,” the activity flows appropriate when nothing goes wrong. But things often do go wrong, in a wide variety of ways. Some problems are technical – a step returns an error code rather than a business result. Others reflect business exceptions, such as the inability to complete the process without additional information. BPMN stands out from traditional modeling notations in its explicit support for exception handling in the diagram. As always, the key to effective modeling is making the diagram as clear and expressive as possible. In this part, we'll focus on specific diagram patterns to use for modeling the most common types of exceptions.

One way to categorize exceptions is by their source. Is the exception detected internally by process logic, or is it a signal received from outside the process? Another way is to distinguish business exceptions from system faults. A system fault means that a model activity cannot provide a business result because of some technical problem – an error code is returned by the implementation, or a communications link is down. A business exception means that the model activity can complete successfully but returns a bad business result – an item is out of stock, or requested information does not arrive in time, or the customer cancels the order. These categories – business exception or system fault, internally detected or externally signaled – provide a useful framework for selecting diagram patterns and building a base of common understanding shared by all in your organization. These patterns are not mandated by the BPMN spec, but should be considered best practice usage of the notation.

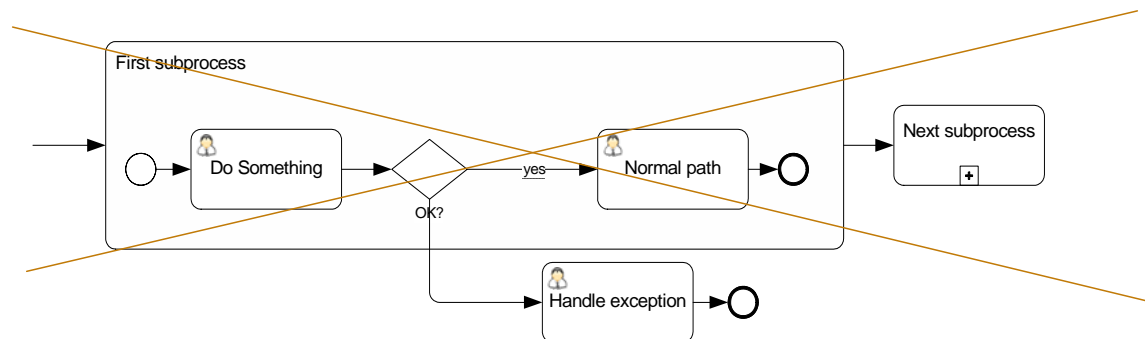
Other important considerations in modeling exception handling are *when* the exception can be detected, and *what should happen* when it is. For exceptions detected internally to the process, the “when” is usually understood to be an end state of a particular activity. However, exceptions signaled from outside could occur when the process is in any number of states, so it is important to think about what the proper response to that signal should be whenever it could occur.

Internal Business Exception Pattern

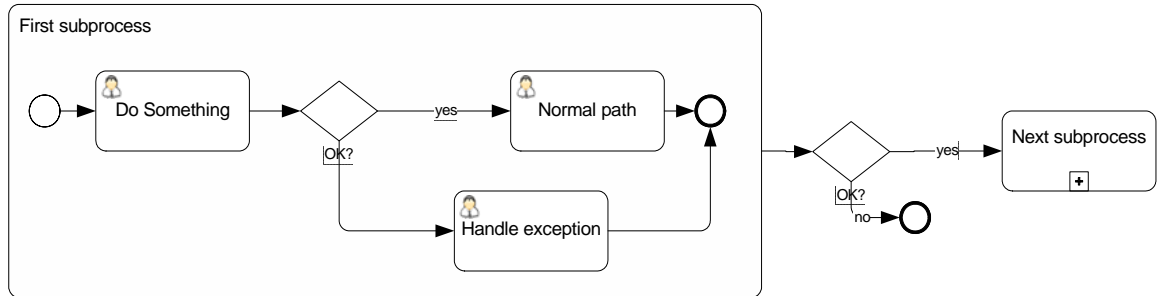
Many BPMN beginners instinctively associate exception handling with events. But by far the most common exception-handling pattern doesn’t use events at all, just a basic gateway. That is the *internal business exception pattern*, meaning the exception is detected internally by logic in a process activity that completes normally but with a bad business result. In that case, simply follow that activity with a gateway that tests the business result. One branch out of the gateway leads to the happy path, and the other branch leads to the exception path.



BPMN places almost no restrictions on where the exception path can go. It can lead directly to an end event, or loop back to the activity preceding the gateway, or somewhere else. One significant restriction is if the exception is enclosed in a subprocess, the exception path cannot cross the subprocess boundary. For example, the following diagram, intended to terminate the top-level process based on a business exception in a subprocess, is illegal in BPMN:

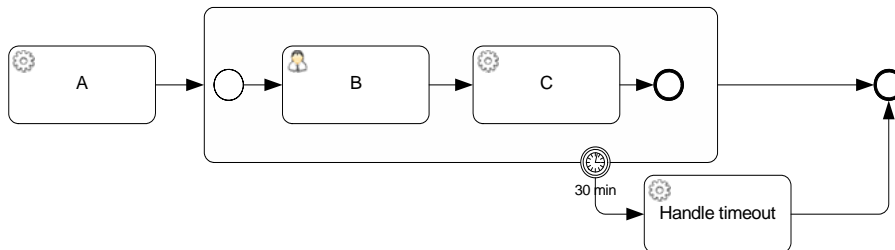


Instead you need to end the exception path inside the subprocess, and if necessary insert another gateway after the subprocess to end the parent process. While the second gateway may seem redundant, it allows the diagram to remain valid whether the subprocess is collapsed or expanded.



Timeout Exception Pattern

A second common exception is when a process activity is not completed by a particular date and time or by some specified duration after it starts. For that, use the *timeout exception pattern* based on an attached timer event. In an attached timer event, the clock starts when the activity it is attached to is first enabled. If the activity completes before that duration expires, the normal flow out of the activity is enabled. If not, the activity is aborted and the exception flow out of the timer event is enabled.

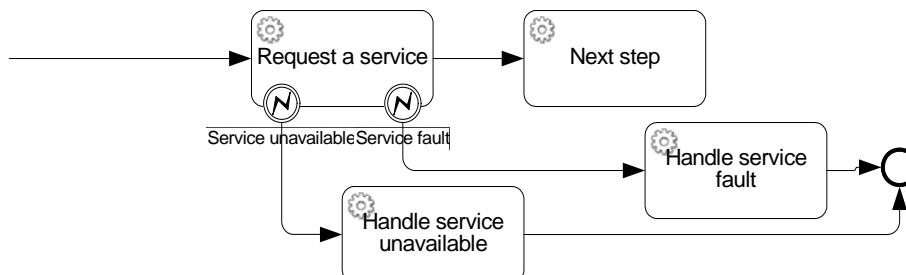


You can use subprocesses to specify when the timer starts and stops. For example, the diagram above says that if activity C is not completed within 30 minutes of the start of activity B, the exception flow is triggered.

System Fault Pattern

A system fault means the process activity cannot complete successfully because of a technical problem. It is not the same as completing with a bad business result. For example, a database query that returns no matching records is a business exception. A database query that cannot be executed because of bad SQL syntax or because the server is down would be a system fault. Typically system faults are exceptions on automated activities, or in BPMN parlance, service tasks.

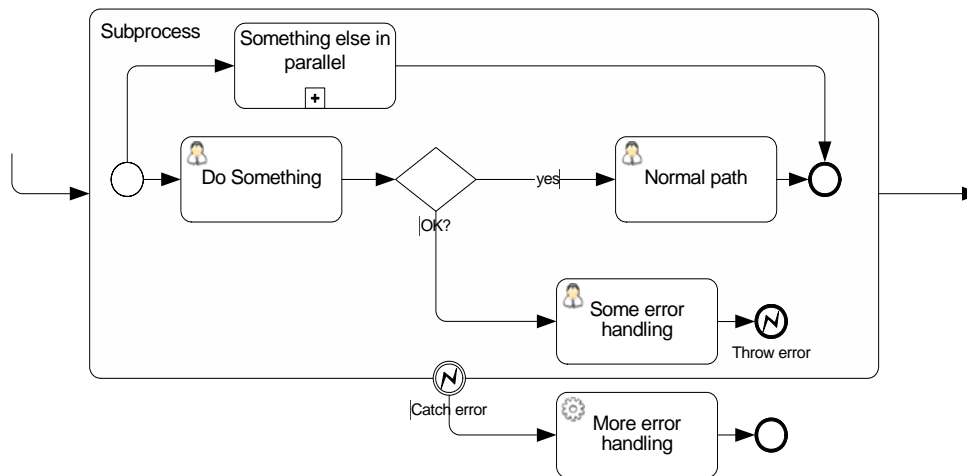
An error intermediate event attached to a service task indicates the *system fault pattern*. It signifies that the task could not complete successfully. If the task completed successfully but returned a bad business result, best practice would be to call that a business exception, but many modelers would use an attached error event for that as well.



In BPMN each fault is specified by an ErrorCode attribute, so there may be more than one, each with its own exception flow.

Business Exception Throw-Catch Pattern

An error event can also be used to handle business exceptions when the simple internal business exception pattern described earlier is insufficient. Typically that occurs when the detected business exception in a subprocess needs to end a parallel thread of the subprocess before beginning the exception flow. In that case, an error end event in the subprocess can “throw” the error result signal that is “caught” by an error intermediate event attached to the subprocess boundary. In this *business exception throw-catch pattern*, paired events are linked by a common ErrorCode attribute.



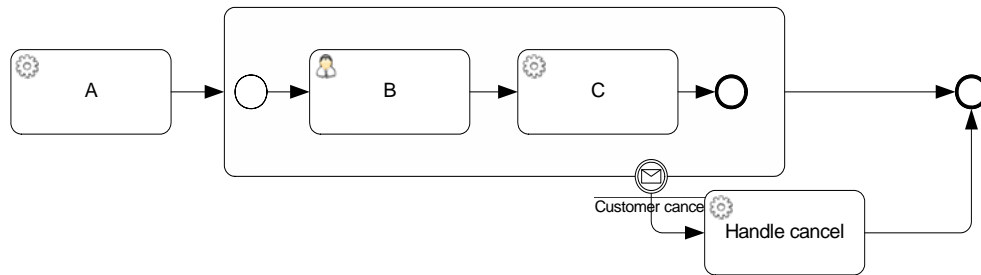
For example, here an internal business exception in the task Do Something is handled with the simple internal business exception pattern described earlier. Following the gateway there is some error handling and that path ends. The exception path out of the gateway doesn't end in a None end event but in an error end event. That is because there is a concurrent path as well, here labeled Something Else in Parallel. For this error, we want to abort that activity and then do more error handling. The thrown error signal is caught by the event labeled Catch Error on the subprocess boundary. Like all attached events, it aborts the activity – the entire subprocess – and then triggers the exception flow.

Thus an error event attached to a subprocess may have one or more error end events in the expanded view of the subprocess that specify where the error signal is thrown.

Unsolicited External Business Exception Pattern

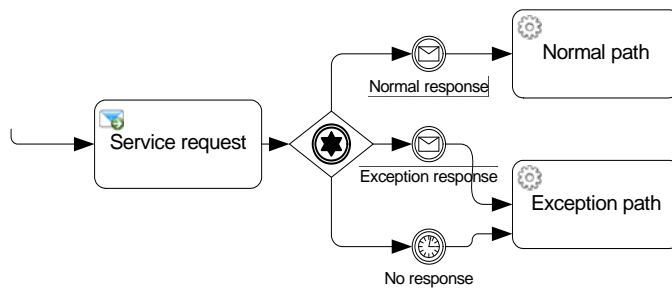
So far all the exceptions have been internal to the process. What about external business exceptions, such as an order change or cancellation? The *unsolicited external business exception pattern* uses an attached message event, which catches the signal from the external entity. If the exception signal occurs while the activity it is attached to is running, the activity is aborted and processing continues on the exception flow.

As with timeouts, subprocesses are essential for scoping these events. Think about the earliest point in the process, and also the latest, where the external exception is handled in a particular way. Now you can enclose that fragment in a subprocess, attach the message event to it, and use the exception flow to lead to that handler. For example, in the diagram below a customer can cancel the order after B has started but before C has ended.



Solicited Response Exception Pattern

The preceding pattern is used to respond to unsolicited external events. But what about exception responses solicited from external entities? Typically for these you would use an event gateway, since you are *waiting* for the event, not *aborting* on the event. A common scenario is a request for additional information, or perhaps a service request. An event gateway pauses the process to wait for a response. The normal response on one gate represents the happy path. An exception response, such as *item out of stock*, could be placed on another gate to define the exception path. No response at all within a specified timeout interval would be modeled as a timer event on a third gate. It could lead to a separate exception path or one shared with the exception response, as shown below.

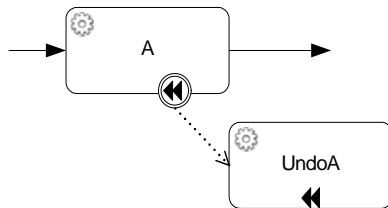


Transaction Compensation

BPMN goes beyond these basic exception handling patterns to model business transaction recovery through *compensation*. A business transaction is a set of activities that must complete atomically, meaning either *all* activities in the set are performed successfully or the state of the system must be restored as if *none* of them were. Unlike classic ACID transaction protocols like two-phase commit, business transactions are inherently long running, so their resources cannot be locked for the duration of the transaction. Instead, each activity in the business transaction is executed normally, but if any part of the transaction fails to complete, those activities already completed are undone by executing a *compensating activity*. For example, the compensating activity for a debit charge would be a credit for the same amount.

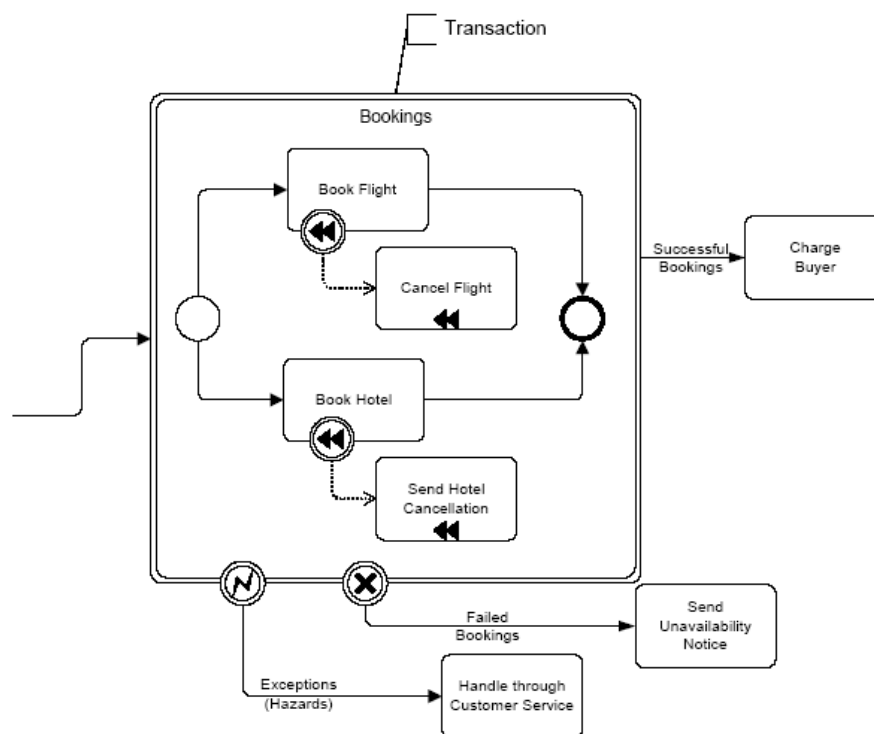
BPMN provides a way to specify all of this in the diagram. Actually, it specifies two alternative ways. We'll talk about one of them.

A subprocess in BPMN can be specified as *transactional*, in which case the rounded rectangle is drawn with a double border. That signifies that if the entire subprocess does not complete successfully, it must be compensated to restore a consistent state of all participating resources. Any activity participating in the transaction that needs to be undone if the transaction fails can be linked to its compensating activity via a compensation intermediate event, drawn with a rewind icon inside.



The compensation event is unlike all other attached events. For one thing, it has no sequence flow out, but only an association to the compensating activity, also marked with a rewind icon. For another, the event is only effective if the activity it is attached to has already completed successfully. (Regular attached events are only effective if the activities they are attached to are still running.) The sole purpose of the attached compensation event is to link an activity with its compensating activity.

A cancel event, drawn with an X icon, attached to the border of a transactional subprocess, is a special type of error event. Like a regular error event, it aborts the subprocess when triggered, but before starting on the exception flow it implicitly commands compensation of the transaction. That means that all activities in the transactional subprocess that have completed and have defined compensating activities should execute those compensating activities to restore a consistent state of all the resources. Once compensation is done, the exception flow proceeds. As with the regular error event, throw of the cancel signal can optionally be shown explicitly by a cancel end event inside the transactional subprocess.



In the above example from the BPMN spec, the transactional subprocess labeled Bookings contains concurrent activities Book Flight and Book Hotel. If either one of them fails, the error (here thrown implicitly as in our system fault pattern) is caught by the cancel event. If Book Hotel fails after Book Flight has completed, invoking compensation executes Cancel Flight to restore a consistent state. Any other event type attached to the subprocess, like the regular error event here labeled Hazards, aborts the transaction without invoking compensation.

Transaction compensation is a neat solution to modeling transaction recovery explicitly in the diagram. Unfortunately, few if any BPMN-based execution environments today provide direct implementation.

Bruce Silver