

BPMN METHOD

& Style

WITH BPMN
IMPLEMENTER'S GUIDE

2nd Edition

BRUCE SILVER

Table of Contents

Preface to the Second Edition

Changes since the First Edition

Method and Style Evolution

New BPMN Implementer's Guide

Executable BPMN

Structure of the Book

BPMN Training

BPMN Tools

Acknowledgments

Part I: What is BPMN?

1. Bad BPMN, Good BPMN

The Paradox of BPMN

Method and Style

The Long Road to BPMN 2.0

Business Process Modeling Is More Than BPMN!

2. How Does A Model Mean?

BPMN's Hidden Conceptual Framework

What Is an Activity?

What Is a Process?

Process Logic

Orchestration

The Questions BPMN Asks

BPMN Levels and Process Modeling Conformance Subclasses

Part II: Method and Style – Level 1

3. BPMN by Example

A Simple Order Process

Exceptions and End States

Swimlanes and Activity Types

Subprocesses

Process Levels and the Hierarchical Style

Parallel Split and Join

Collaboration and Black-Box Pools

Start Events and the Process Instance

The Top-Level Diagram

4. The Level 1 Palette

Activity

Task

Send and Receive Task

Manual vs. User Task

Script vs. Service Task

Business Rule Task

Subprocess
Parallel Box and Ad-Hoc Subprocess
The Value of Subprocesses
Call Activity
Gateway
Exclusive Gateway
Parallel Gateway
Start Event
None Start Event
Message Start Event
Timer Start Event
Multiple and Multiple-Parallel Start Event
Alternative Start Events
End Event
None End Event
Message End Event
Terminate End Event
Multiple End Event
Sequence Flow
Message Flow
Pool
Lane
Data Object and Data Store
Documentation, Text Annotation, and Group

5. The Method

Goals of the Method
Hierarchical Top-Down Modeling
End State
Step 1. Determine Process Scope
Scenario: Car Dealer Order-to-Cash
Step 2: The High-Level Map
Scenario: Car Dealer Order-to-Cash
Step 3: Top-Level Process Diagram
Scenario: Car Dealer Order-to-Cash
Step 4: Child-Level Expansion
Scenario: Car Dealer Order-to-Cash
Step 5: Add Message Flows
Scenario: Car Dealer Order-to-Cash
Method Recap

6. BPMN Style

The Basic Principle of BPMN Style
Style Rules
Official BPMN 2.0 Rules

Part III: Method and Style – Level 2

7. Events

[Event-Triggered Behavior](#)

[Timer Event](#)

[Catching Timer Event](#)

[Timer Boundary Event](#)

[Timed Interval](#)

[Timer Event vs. Gateway](#)

[Message Event](#)

[Message and Message Flow](#)

[Send Task and Throwing Message Event](#)

["Sending" Within a Process](#)

[Receive Task and Catching Message Event](#)

[Asynchronous and Synchronous Messaging](#)

[Event Gateway](#)

[Message Boundary Event](#)

[Error Event](#)

[Other Level 2 Events](#)

[Escalation Event](#)

[Signal Event](#)

[Conditional Event](#)

[Link Event](#)

[Event Subprocess](#)

8. Iteration and Instance Alignment

[Loop Activity](#)

[Multi-Instance Activity](#)

[Using Repeating Activities](#)

[Using Multiple Pools](#)

[Batch Processes](#)

[Instance Alignment](#)

9. Process Splitting and Merging

[Conditionally Parallel Flow](#)

[OR Gateway Split](#)

[Conditional Sequence Flow](#)

[Merging Sequence Flows](#)

[Merging Alternative Paths](#)

[AND Gateway Join](#)

[Multi-Merge](#)

[OR Gateway Join](#)

[Discriminator Pattern](#)

10. Transactions

[ACID Transactions](#)

[Business Transactions](#)

[Compensation Boundary Event and Compensating Activity](#)

[Cancel Event](#)

[Compensation Throw-Catch](#)

[Using Compensation](#)

[11. The Rules of BPMN](#)

[Sources of BPMN Truth](#)

[BPMN Rules for Level 2 Process Modeling](#)

[Sequence Flow](#)

[Message Flow](#)

[Start Event](#)

[End Event](#)

[Boundary Event](#)

[Throwing or Catching Intermediate Event](#)

[Gateway](#)

[Process \(Pool\)](#)

[Style Rules for Level 2 Process Modeling](#)

[Labeling](#)

[End Event](#)

[Subprocess Expansion](#)

[Message Flow](#)

[Model Validation](#)

[Part IV: BPMN Implementer's Guide – Non-Executable BPMN](#)

[12. BPMN 2.0 Metamodel and Schema](#)

[XSD Basics](#)

[BPMN Schema Fundamentals](#)

[XSD Files](#)

[Semantic and Graphical Models](#)

[IDs and ID References](#)

[Import, targetNamespace, and Remote ID References](#)

[13. Process Modeling Conformance Subclasses](#)

[Descriptive Subclass](#)

[Analytic Subclass](#)

[Common Executable Subclass](#)

[14. BPMN Serialization Basics](#)

[definitions](#)

[targetNamespace](#)

[expressionLanguage and typeLanguage](#)

[exporter and exporterVersion](#)

[Global Namespace Declarations](#)

[schemaLocation](#)

[import](#)

extension
rootElement
BPMNDiagram
relationship
documentation and extensionElements
collaboration
participant
messageFlow
process
Example: Simple Process Model
Example: Simple Collaboration Model
Example: Simple Import and Call Activity

15. Serializing Process Elements

flowElement and flowNode

activity

subProcess

gateway

event

startEvent

intermediateCatchEvent

intermediateThrowEvent

Link Event Bug

boundaryEvent

endEvent

sequenceFlow

Expressions

Formal Expressions

laneSet and lane

Artifacts

textAnnotation

group

16. Serializing Data Flow

Non-Executable Data Flow

dataObject

dataInput and dataOutput

dataInputAssociation and dataOutputAssociation

dataStore and dataStoreReference

Example: Non-Executable Data Flow

More on Data Inputs and Data Outputs

17. The BPMNDI Graphical Model

BPMNDI Basics

Process Levels and Pages

BPMNDiagram

BPMNPlane

[BPMNShape](#)
[BPMNEdge](#)
[BPMNDI Examples](#)

18. BPMN-I

[BPMN-I Profile Serialization Rules](#)
[Schema Validation](#)
[definitions](#)
[import](#)
[Non-Standard Elements and Attributes](#)
[Remote Element References](#)
[Page Structure](#)
[participant and Pool](#)
[collaboration](#)
[process](#)
[laneSet and lane](#)
[flowNode](#)
[activity](#)
[startEvent](#)
[boundaryEvent](#)
[intermediateCatchEvent and intermediateThrowEvent](#)
[endEvent](#)
[Gateway](#)
[sequenceFlow](#)
[messageFlow](#)
[textAnnotation and association](#)
[group](#)
[Data Flow](#)
[BPMNShape](#)
[BPMNEdge](#)

Part V: BPMN Implementer's Guide – Executable BPMN

19. What Is Executable BPMN?

[Common Executable Subclass](#)

20. Variables and Data Mapping

[itemDefinition](#)
[message](#)
[Importing Structure Definitions](#)
[Example: Data Flow with Imported Item Definitions](#)
[Properties and Instance Attributes](#)
[Data Mapping](#)
[Identity Mapping](#)
[Assignment From/To Mapping](#)

[Transformation Mapping](#)

[Script Task Mapping](#)

[21. Services, Messages, and Events](#)

[Services](#)

[interface](#)

[operation](#)

[Messages](#)

[Automated Tasks](#)

[serviceTask](#)

[sendTask](#)

[receiveTask](#)

[businessRuleTask](#)

[Events](#)

[Message Events](#)

[Signal Events](#)

[Error and Escalation Events](#)

[Timer Events](#)

[22. Human Tasks](#)

[userTask](#)

[Performer Assignment](#)

[Task Assignment by Parameterized Query](#)

[Task Assignment by Expression](#)

[23. Executable BPMN in Practice](#)

[Handling Java Data](#)

[Referencing Java Data](#)

[UEL](#)

[Groovy](#)

[Are XPATH Data Access Functions Needed?](#)

[Services and Service Adapters](#)

[Example: Bonita Open Solution](#)

[Defining Process Variables](#)

[Saving the Request Message](#)

[Service Task – Database Lookup](#)

[Branching at the Gateway](#)

[Script Task – Calculating the Invoice Amount](#)

[Service Task – Email Connector](#)

[Timer Boundary Event](#)

[24. Aligning Executable Design with BPMN Method and Style](#)

[End State Variables](#)

[Gateway Conditions](#)

[Messages](#)

[Errors](#)

[Index](#)

[**25. About the Author**](#)

BPMN Method and Style

Second Edition

with BPMN Implementer's Guide

Bruce Silver

CODY-CASSIDY PRESS

BPMN Method and Style, Second Edition, with BPMN Implementer's Guide

By Bruce Silver

ISBN 978-0-9823681-1-4

Copyright © 2011 by Bruce Silver. All rights reserved. No part of this work covered by the copyright hereon may be reproduced or used in any form or by any means – graphic, electronic, or mechanical, including photocopying, scanning, or transcribing – without written permission of the publisher, except in the case of brief quotations embodied in critical articles and reviews.

Published by Cody-Cassidy Press, Aptos, CA 95003 USA

Contact info@cody-cassidy.com +1 (831) 685-8803

The author and publisher accept no responsibility or liability for loss or damage occasioned to any person or property through use of the material, instructions, methods, or ideas contained herein, or acting or refraining from acting as a result of such use. The author and publisher disclaim all implied warranties, including merchantability or fitness for any particular purpose.

Library of Congress Control Number: 2011918027

Library of Congress Subject Headings

Workflow -- Management.

Process control -- Data processing -- Management.

Business -- Data processing -- Management.

Management information systems.

Reengineering (Management)

Information resources management.

Agile software development.

Cover design by Leyba Associates

For Amelie

*****ebook converter DEMO Watermarks*****

Preface to the Second Edition

BPMN, which stands for Business Process Model and Notation, is a diagramming language for business process models. It is important not because it is superior in every way to other process notations, but because it is a *multi-vendor standard*, maintained by the Object Management Group (OMG), and widely adopted by modelers and tool vendors alike. This book is more than a dictionary of BPMN's shapes and symbols. It offers a unique approach to understanding and mastering the process modeling standard, based on two fundamental principles:

- *The Method and Style Principle* – A given BPMN diagram should have one and only one interpretation. The process logic should be completely and unambiguously described by the diagram alone.
- *The BPMN-I Principle* – A given BPMN diagram should have one and only one XML serialization. Otherwise model interchange between tools cannot be achieved.

The first principle applies to modelers, the second one to implementers, such as BPMN tool vendors... but they are closely related. Unfortunately, strict adherence to the BPMN 2.0 specification is insufficient to guarantee either one. Satisfying those principles requires additional *conventions*, which I call *style rules* and *BPMN-I rules*, respectively. *BPMN Method and Style* not only explains the meaning and usage of the important diagram elements but provides prescriptive guidance, including style rules and BPMN-I rules, for constructing BPMN models that are clear, complete, consistent, and interchangeable between tools.

Changes since the First Edition

The first edition of the *BPMN Method and Style* was published in June 2009, at the time of completion of the BPMN 2.0 “beta” specification. When I began writing the second edition, I thought that a significant portion of it could be copied and pasted from the original, but that turned out not to be the case. In fact, the new edition has been rewritten almost entirely. The central ideas are the same, but the exposition, emphasis, and examples are all new. I have taught Method and Style to hundreds of students since publication of the first edition, and the new edition benefits greatly from that experience. It is clearer, more concise, better organized.

The unique approach of the first edition, including segregation of the palette into “levels,” a prescriptive modeling methodology, and principles of BPMN “style”, have all been carried forward in the new edition. The essential goal remains the same as well: BPMN diagrams that are unambiguous, understandable by others, and complete, fully revealing the process logic even in the absence of attached documentation.

Both the goal and its associated principles and rules have formed the basis of my BPMN training since early 2007. The approach rests on three key pillars:

- **Focus on the important shapes and symbols.** Although BPMN’s critics point to the complexity of the complete BPMN 2.0 element set to “prove” its unsuitability for business people, only a fraction of the full set is used in practice. Method and Style takes a *levels-based approach*. *BPMN Level 1* is a basic working set of shapes almost entirely familiar from traditional flowcharting. *BPMN Level 2* broadens the palette a bit, most notably by the event and gateway types most commonly used for describing exception handling in the diagram. Level 2 is still just a fraction of the complete element set, but few modelers will ever have a [1] need to go beyond the Level 2 palette.
- **A prescriptive methodology**, a step-by-step recipe leading from a blank page to a complete process model that reveals the process logic clearly from the printed diagram. The goal of the Method is not creativity but structural consistency: Given the same set of facts about how the process works, any modeler should create (more or less) the same model structure.
- **BPMN style**, a set of modeling conventions that make the process logic unambiguous from the diagram alone. Like spelling and grammar checks in Microsoft Word, style rule violations can be flagged in a modeling tool.

Method and Style Evolution

While these Method and Style pillars remain intact, the new edition reflects two years’ evolution of both the methodology and style through repeated interaction with students in my BPMN training. For example, the Method’s *High-Level Map* has evolved to better align with business process architecture. The book now clarifies how fundamental BPMN concepts like *process* and *activity* relate to published business process frameworks. Also, BPMN style, once taught as a list of recommended best practices, is now more effectively presented as a set of *style rules* that can be validated in software [2]. These changes

*****ebook converter DEMO Watermarks*****

are reflected in the new edition.

Another change is the evolution of BPMN Level 1. In the original edition it implied not simply a limited working set of shapes and symbols but a more relaxed attitude toward the rules of BPMN, more akin to traditional flowcharting. Today I don't teach it that way, and the new edition reflects the change. One reason is the availability of automated style rule validation, mentioned earlier. That makes a huge difference, since even beginners can quickly learn to avoid style errors. Also, I have come to see that hiding BPMN's conceptual foundations from Level 1 modelers makes it more difficult in the end to create a common language shared between business and IT. Since the shapes and symbols of the Level 1 palette are mostly familiar to business users, it is better to expose BPMN's foundations early on. The ultimate goal, remember, is a language that spans the business and technical worlds.

In the new edition, the Level 1 and Level 2 palettes have been adjusted to correspond *exactly* with the Descriptive and Analytic subclasses of the final BPMN 2.0 specification. The Level 1 section of the book now covers the entire Descriptive subclass, and the Level 2 section the entire Analytic subclass.

New BPMN Implementer's Guide

While the *graphical notation* of BPMN 2.0 is virtually unchanged since the first edition, OMG's Finalization Task Force made several changes to the *XML serialization*. The XML serialization is important not only for executable BPMN but for interchange of non-executable models between BPMN tools. Addition of the Descriptive and Analytic Process Modeling Conformance subclasses, mentioned previously, was also of great significance. Without them, BPMN interoperability between tool vendors would be near impossible without side agreements. Another key addition was a proper XML schema for *diagram layout information*, important not only for preserving some semblance of the original layout on model interchange, but for defining the page structure of the end-to-end model.

While the final specification stabilized the XML structure, BPMN serialization is still poorly understood by implementers. For that reason, the second edition adds an entirely new *BPMN Implementer's Guide* aimed at BPMN tool vendors and developers. It explains the BPMN 2.0 metamodel, proper serialization of process models, and conventions that promote interoperability between BPMN tools. As most tool vendors are still in the early stages of implementing the final BPMN 2.0 specification, the timing is right for such a guide.

Like the Method and Style sections of the book, the BPMN Implementer's Guide addresses gaps in the official BPMN 2.0 spec by introducing *conventions* that act as additional constraints. In principle, *the XML serialization of a BPMN model should be uniquely determined by the diagram alone*. This is similar to the Method and Style principle that the process logic should be evident from the diagram alone, but it is not exactly the same. While Method and Style conventions impose constraints on *modelers*, such as requiring certain labels, model interoperability imposes constraints on *BPMN tool vendors*, such as which elements and attributes should be included or omitted. Remember, *a given BPMN Level 1 or Level 2 diagram should have one and only one XML representation...* but the spec allows more than one. Those constraints, collectively called the *BPMN-I Profile*, are intended to define an interoperable format for any non-executable model containing only members of the Analytic subclass of BPMN 2.0. And like the rules of Method and Style, BPMN-I Profile constraints can be expressed as *rules* that can be validated in a [3] tool.

Neither the style rules nor the BPMN-I Profile are part of the official BPMN 2.0 standard from OMG, but both are consistent with its goals of semantic precision, visual clarity, and interoperability between tools. Just as some Method and Style ideas, such as the Level 1 and Level 2 palettes, eventually found their way into the official BPMN 2.0 spec, it is my hope that style rules and BPMN-I Profile rules will eventually be codified in future versions of the official BPMN standard. But realistically, no revision of BPMN 2.0 is likely to come out before 2013.

Executable BPMN

While BPMN-I is specific to non-executable BPMN, the BPMN Implementer's Guide also includes a section on *executable BPMN*, beginning with what that phrase means in the context of the BPMN 2.0 standard. We'll show how process data is represented in the BPMN XML and how it is mapped to variables, task I/O parameters, gateway conditions, message payloads, event definitions, service interfaces, and human task assignment rules in "executable BPMN". The basic structure has not changed since the first edition of the book, but the XML schema has changed significantly. The new edition describes the proper serialization of BPMN 2.0 execution-related details in accordance with the final BPMN 2.0 specification, and relates that to the way these details are defined in real BPMN 2.0-based process automation tools.

Structure of the Book

Part I, *What Is BPMN?*, discusses the importance of BPMN to business process management overall, its similarities to and differences from traditional flowcharting, and what distinguishes “good BPMN” from “bad BPMN.” It discusses BPMN’s conceptual foundations, and explains how BPMN’s notions of *activity* and *process* relate to business process architecture.

Part II, *Method and Style – Level 1*, is a detailed exposition of the Method and Style approach to process modeling. We start with *BPMN by Example*, walking the reader through construction of a complete process model using only a limited working set of BPMN shapes and symbols familiar from traditional flowcharting – the *Level 1 palette*. Afterward, the book goes back and discusses the meaning and proper usage of each of the Level 1 elements.

Then we explain *the Method*, a cookbook recipe for creating consistent end-to-end BPMN models that reveal at a glance the meaning of the *process instance*, the process’s various *end states*, and its *touchpoints with the external environment*: the customer, service providers, and other internal processes.

Following that we discuss *BPMN style*, the grammar rules of BPMN that make the process logic clear from the diagram alone and traceable through the model hierarchy. The style section is patterned after Strunk and White’s *The Elements of Style*, still a reliable set of principles for writing effective English prose. Even though that book goes back to Professor Strunk’s lecture notes of 1919, its continued popularity demonstrates that basic principles of style can stand the test of time. The book applies similar principles to the creation of BPMN models, with the goals of clarity, expressiveness, and consistency with BPMN’s precise technical meaning.

Part III, *Method and Style – Level 2*, expands the palette of shapes and symbols. The primary focus is on *Events*, primarily the “big 3”— Timer, Message, and Error. We will also discuss the other events in the Analytic subclass, as well as *branching and merging* with gateways and conditional sequence flows. We discuss *iteration* using loop and multi-instance activities and multi-participant pools, and we’ll see how certain business processes cannot be modeled as a single BPMN process but require multiple interacting pools. We conclude Part III by reviewing *the rules of BPMN*, both the official rules and the *style rules*, and show how to use *validation* to maintain model quality and make the process logic easily traceable.

If you are looking for information about BPMN 2.0 Choreography and Conversation models, you won’t find it in this book. So far I have seen little interest in these additions to the BPMN standard, focused on B2B interactions.

Parts IV and V, comprising the *BPMN Implementer’s Guide*, shift attention from the graphical notation to the XML serialization. This is primarily of interest to developers and tool vendors, but business analysts and architects will find it valuable as well. In Part IV, dealing with *non-executable BPMN*, we discuss the *BPMN 2.0 metamodel* and its representation in XML Schema, with focus on proper XML serialization of *process elements* in the Descriptive and Analytic *Process Modeling Conformance subclasses*, including *data flow*. We’ll see how the *graphics model* connects to the *semantic model*, and how multi-page hierarchical models are defined. And we’ll look at how to reference reusable subprocesses and tasks imported from independent BPMN files, and the importance of the model’s *targetNamespace*. Part IV also describes the *BPMN-I Profile*, a set of conventions intended to make non-executable model interchange possible. Where the BPMN 2.0 specification allows multiple ways to serialize a given

diagram, BPMN-I tries to limit it to one way.

Part V, *Executable BPMN*, discusses how execution-related information, particularly *process data*, is defined, referenced, and mapped in executable BPMN models. We discuss BPMN concepts of *services*, *messages*, and *events* in an execution-related context, and we'll see how performer assignment is modeled in human tasks.

We conclude with a discussion of *executable BPMN in practice*, as it is being implemented by the first wave of BPMN 2.0-compliant BPM Suites, with examples using Bonita Open Solution. We'll see how point-click design in the BPMS tool is reflected in the BPMN 2.0 XML export. And finally, we'll discuss *alignment of executable design with BPMN Method and Style*, with guidance for tool developers on how to provide elements of that alignment out-of-the-box.

BPMN Training

The book provides many process diagrams, and I encourage readers to reproduce them using a BPMN tool. It is difficult, if not impossible, to become proficient at BPMN simply from reading a book. Like any skill, you really learn BPMN only by *doing* it, working through the creation of diagrams that clearly express your intended meaning. But this book is just a reference; it is not a substitute for real BPMN *training*.

There is an old joke about sex education vs. sex training that I won't repeat here. But you get the idea. Training involves practice, exercises and discussion of solutions, why certain ways work better than

others. I provide such training myself, both online and in the classroom, through several channels [\[4\]](#). This book could be used as a reference for that training, or as a textbook in a college course on BPMN, but by itself it is not training.

BPMN Tools

The simplest BPMN diagrams can be drawn by hand, but BPMN assumes use of a software tool. The good news is that there are many such tools to choose from, and the meaning of the diagram does not change from one tool to the next. But even though BPMN is a standard, the tools are not all equally good. Some are little more than drawing tools. They can produce diagrams containing the standard shapes and connectors, but they do not “understand” their meaning. They cannot, for example, validate the model, or save it in XML interchangeable with another BPMN tool.

Some tools support all of the BPMN shapes and symbols, while others – particularly those offered as part of a BPM Suite – include just those that the Suite’s process engine can execute. Tools mostly adhere to the symbols, markers, and semantics specified by the standard, but some take liberties here and there. Some tools allow you to draw pools and message flows, while others do not. Some naturally support the *hierarchical* modeling style, in which subprocesses are expanded on separate hyperlinked diagrams, while others are geared toward “flat” process models with inline subprocess expansion. Also, BPMN tools vary widely in the *non-BPMN information* they describe, such as problems and goals, KPIs, organizational roles, and systems.

Prior to version 2.0, the BPMN spec did not even attempt to describe requirements for “conformance”. As a consequence, many tools claim to support BPMN but really do not. The BPMN 2.0 spec does spell out requirements for Process Modeling Conformance. The Descriptive and Analytic subclasses, equivalent to our Level 1 and Level 2 palettes (in fact, borrowed from my BPMN training!), specify the elements in non-executable models that *must* be supported to claim conformance. The BPMN-I Profile, described in Part IV of the book, provides serialization rules for non-executable models that allow tool vendors to interchange those models automatically. At this writing, no BPMN tool vendor yet claims full conformance with the BPMN-I Profile, but some are close.

All this is a long way of saying that *even though BPMN is a standard, BPMN tools are not all the same*, and your choice of tool may significantly impact your ability to create “good BPMN” consistently.

The diagrams used in this book were created using Process Modeler for Visio, an add-in to Microsoft Visio from itp commerce ltd, of Bern, Switzerland.^[5] This is the tool I primarily use in my BPMN training and certification. A key reason is it has the style rule validation built in. Also, it supports the full BPMN 2.0 element set with proper XML serialization and model export and import, and simplifies hierarchical modeling as recommended by the Method.

A major strength of BPMN is that users enjoy a wide choice of tools. Nevertheless, some readers will surely find that the BPMN tool they are currently using does not support some of the shapes, symbols, and patterns described in this book. One possible reason is that the tool is based on BPMN 1.x, while the book is based on BPMN 2.0. Tool vendors are often loath to advertise which version of the standard they support, so here is an easy way to tell.

If your tool includes a shape that looks like this



then it is based on BPMN 1.0. That was obsolete in 2008, so if you are serious about process modeling, I would recommend upgrading your tool.

*****ebook converter DEMO Watermarks*****

If your tool can draw a shape that looks like this



or a shape with a black envelope, like this



but not shapes like these



then it is probably based on BPMN 1.2. This is all right for Level 1 modeling, but it doesn't support important shapes such as non-interrupting event and data store. More important, it doesn't support the formal BPMN metamodel and XML interchange format.

The last three shapes above are new in BPMN 2.0, and a tool that supports BPMN 2.0 is best. If it supports the Analytic subclass, that means it can draw non-interrupting boundary events (the dashed double ring), Escalation events, and data stores, all useful and part of the Level 2 palette. In the book, we will indicate which shapes and symbols are new in BPMN 2.0. If your tool does not yet support BPMN 2.0, all is not lost... but you should encourage your tool vendor to move up to BPMN 2.0 soon!

Acknowledgments

I would like to acknowledge the efforts of two individuals in advancing the standard and the Method and Style approach since publication of the original edition. Without those efforts, this book could not have been written.

Robert Shapiro of Process Analytica, as member of the BPMN 2.0 Finalization Task Force in OMG, succeeded where I could not, getting the Descriptive and Analytic Process Modeling Conformance subclasses included in the compliance section of the final BPMN 2.0 spec. Not only does this provide the only practical basis for model interchange between tools, but it serves as official endorsement of the Level 1 and Level 2 palettes of BPMN Method and Style.

Stephan Fischli of itp commerce, maker of Process Modeler for Visio, has continually added features to his BPMN tool that embrace and automate the Method and Style approach. Notable examples include built-in support for style rule validation, proper XML serialization, and import of global tasks and processes from external Visio files. Good BPMN requires good tools, and Stephan's is the best.

I also want to acknowledge the helpful responses of several individuals to my questions on technical aspects of the BPMN 2.0 specification, including Matthias Kloppmann of IBM, Ralf Mueller of Oracle, Denis Gagne of Trisotech, and Falko Menge of Camunda.

I wish to thank Charles Souillard, Nicolas Chabanoles, and Aurelien Pupier at BonitaSoft for providing the executable BPMN 2.0 model used in Chapter 23 and answering my many questions about it. Thanks also to Carol Leyba of Leyba Associates for great work on the cover design.

Finally, I would like to recognize those whose efforts will ultimately make this book available in other languages and formats: Stephan Fischli and his team at itp commerce (German translation), Miguel Valdes Faura and his team at BonitaSoft (French translation), Brian Reale and his team at Colosa (Spanish translation), and Declan Chellar for preparation of the Kindle version. Thanks to all of them for helping spread the word on BPMN Method and Style.

*Bruce Silver
October 2011*

Part I: What is BPMN?

Chapter 1

Bad BPMN, Good BPMN

BPMN stands for *Business Process Model and Notation*. For the vast majority of BPMN users, the most important part is the N – the graphical *notation* – a diagramming language for business process flows. The most important thing about it is that it is a *standard*, maintained by the Object Management Group (OMG). That means it is not owned or controlled by a single tool vendor or consultancy. You pay no fee or royalty to use the intellectual property it represents. Today, virtually every process modeling tool supports BPMN in some fashion, even though a few vendors may grumble that their own proprietary notation is better or more business-friendly.

A key benefit of a process modeling standard is that understanding is not limited to users of a particular tool. The semantics are defined by the standard, not by each tool. BPMN is an *expressive* language, able to describe nuances of process behavior compactly in the diagram. At the same time, the meaning is precise enough to describe the technical details that control process execution in an automation engine! Thus BPMN bridges the worlds of business and IT, a common process language that can be shared between them.

The Paradox of BPMN

BPMN's popularity begins with its *outward familiarity*, especially to business people. Its boxes and arrows, diamonds and swimlanes look a lot like traditional flowcharts, which have been around for 25 years. And that was by design. But here is the paradox of BPMN. While outwardly familiar, BPMN's unique capabilities come from ways in which it *differs* from traditional flowcharting.

One difference, as mentioned above, is that modelers may not make up their own meaning for the standard shapes and symbols. BPMN is based on a formal *specification*, including a metamodel and rules of usage. Its expressiveness derives from the extensive variety of markers, icons, and border styles that precisely refine the meaning of the basic shapes. It has *rules* that govern the use of each shape, what may connect to what. Thus you can *validate* a BPMN model, and any BPMN tool worth using can do that in one click of the mouse.

A second key difference from traditional flowcharts is that BPMN can describe *event-triggered behavior*. An event is “something that happens” while the process is underway: The customer calls to change the order; a service level agreement is in danger of being violated; an expected response does not arrive in time; a system is down. These things happen all the time. If your model represents the “real” process it needs to say *what should happen* when those exceptions occur. BPMN lets you do that, and visualize that behavior in the diagram itself.

Third, in addition to the solid *sequence flow* connectors depicting flow *within* a process, BPMN describes the communications *between* the process and external entities like the customer, external service providers, and other internal processes. Those communications are represented by a dashed connector, called a *message flow*. The pattern of message flows, called *collaboration*, reveals how the process fits in the global environment.

Thus, using BPMN correctly and effectively requires learning the parts of it that are *unfamiliar*. It's not hard, and that is what this book is about. Nevertheless, following the release of BPMN 2.0 in 2010, we began to hear some people say that understanding BPMN is “too hard for business people.” Usually the people saying it were tool vendors or consultants wedded to their own legacy notations.

But let's concede one point: there is a lot of “bad BPMN” out there, diagrams that are invalid, incomplete, or ambiguous. But that does not mean that creating “good BPMN” is beyond the reach of business users. I suspect that if you looked at a sampling of college application essays you would see a correspondingly high frequency of misused words, grammar errors, and garbled sentence structure. Should you conclude from that that English is just “too hard for high school students”? No, a language *requires* richness in order to express complex ideas. But you need to teach people how to use it correctly and effectively, and provide tools to help them do it.

Method and Style

That's what this book is about. It shows you how to create "good BPMN," meaning models that are:

- **Correct.** The diagram should not violate the rules laid out in the BPMN specification.
- **Clear.** The process logic should be unambiguous and obvious from the diagram alone. It should not depend on attached documentation. Note the term *process logic* means the logic of proceeding from one task to the next, not the details of how individual tasks are performed.
- **Complete.** In addition to the activity flow, the diagram should indicate how the process starts, all of its significant end states, and its communications with external entities, including the requester, service providers, and other internal processes.
- **Consistent.** Given the same set of facts about the process logic, all modelers should create more or less the same BPMN model, or at least models that are similarly structured. Consistency across the organization makes models easier to share and understand.

The BPMN spec demands only correctness, but this is insufficient for good BPMN. Good BPMN requires adopting conventions that go beyond the requirements of the specification. I call those conventions *BPMN Method and Style*.

The Method I present in the book is a prescriptive recipe for turning a blank page into a BPMN model that is correct, clear, complete, and consistent. It is less important that you adopt my Method exactly than you establish some kind of prescriptive process modeling methodology for use across your organization. Consistent model structure maximizes shared understanding as well as model reuse across the organization.

The Method and Style approach is *top-down* and the resulting models have a *hierarchical* structure. *Style* refers to basic principles of composition and element usage that go beyond the official rules of the spec. I used to teach BPMN style as "best practices," but I have found that it is more effective to reduce it to a list of *rules that can be validated in the tool*, just like the official BPMN rules.

Model clarity is directly related to the style rules, many of which simply have to do with labeling. For some reason, beginning modelers are extremely stingy with labels, and the BPMN spec does not require any labels at all. But when you think about it, all you have in the diagram to communicate meaning are shapes and labels. In hierarchical models, where process levels are represented on separate pages, labeling is what makes the process logic traceable from the top level down to the lowest level of detail.

The Long Road to BPMN 2.0

For most modelers the important part of BPMN is the *N*, the graphical *notation*. But most of the effort in creating BPMN 2.0 involved the *M*, the *model*. That means the formal semantics of the element definitions and their inter-relationships, as defined by a formal *metamodel* and its corresponding XML representation. The notation, the shapes and symbols, actually changed very little from BPMN 1.2 to BPMN 2.0.

One key motivation for the shift in focus in BPMN 2.0 was to provide an official *XML interchange format* for process models. A second was the wish on the part of major BPM Suite vendors – notably IBM, Oracle, and SAP – to make BPMN models *executable* in a process engine. Actually, many BPM Suites were already basing executable process design on BPMN 1.x, but they each modeled the execution-related details in their own proprietary way. BPMN 2.0 would standardize the representation of process data, messages, services, task assignment, and the like, not in the diagram but in the XML underneath.

OMG's focus on process execution led to a bit of a backlash against BPMN 2.0 from some in the BPM community, but the fact remains that the vast majority of BPMN 2.0 use is still for diagramming non-executable processes. BPMN 2.0 adds only a couple important new graphical elements to the notation: non-interrupting events and data store. But the tension between BPMN as a business-friendly diagramming notation and BPMN as an executable process language has been there from the beginning.

BPMN originated in 2002 as the visual design layer of a new type of “transactional workflow” system from a consortium called BPMI.org, led by a startup named Intalio. Leveraging the distributed standards-based architecture of the web and web services, this new type of BPM would be a radical break from the proprietary workflow systems of the client/server era. One key difference would be making the process execution language a *vendor-independent standard*. As it developed the language, called BPML, BPMI.org reached a peak of 200 members, essentially all the major software vendors except IBM and Microsoft.

Another difference would be *business empowerment*. “In a nutshell,” recalls BPMI’s founder Ismael Ghalimi, “it would allow less-technical people to build transactional applications by drawing simple flow

[\[6\]](#)

charts.” BPML would not be coded by hand but *generated from a diagram*, which would also be standardized: BPMN. Existing process diagramming standards like UML were rejected as too IT-centric. BPMI demanded something more business-friendly. Howard Smith and Peter Fingar fleshed out the promise of business-empowerment through BPMN in a seminal 2002 book, *BPM: The Third Wave*, which correctly predicted that empowering business people to manage their own processes was critical to the evolution of BPM.

BPMI.org produced a spec for BPMN 1.0 in 2004. Ghalimi continues: “Among [the BPMI members were] very many process modeling tool vendors who loved the idea of a standard notation for processes, and very many workflow vendors who hated the idea of a standard language for executing them. The former understood that they could provide a lot of value around the core process modeling tool. The latter knew all too well that fragmentation of the market helped preserve the status quo...”

As it turned out, no one had anything to fear from BPML. IBM and Microsoft countered with BPEL, a slightly different language layered on top of the new web services standard called WSDL. In an instant

those two vendors trumped 200, and BPML was effectively wiped out. In 2005, needing a new home, BPMI.org was absorbed into the Object Management Group, ironically home of the spurned alternative, UML. OMG formally adopted the BPMN 1.0 spec in 2006, adding a minor update, BPMN 1.1, in January 2008, and BPMN 1.2, a bug fix, a year after that.

It's a familiar cycle in the world of IT standards, one that normally leads to quiet oblivion. By the end of 2008, however, against all odds, BPMN turned out to be not on the road to oblivion but approaching a tipping point of mass adoption. The explanation is simple. Smith and Fingar were right... sort of. Business empowerment *is* the key to BPM, and BPMN provides that – not the executable code, but the precise flow logic that code would have to implement. Even though the BPMN specification made no explicit distinction between its elements that are part of the non-executable *model* and those required for executable *implementation*, it is pretty obvious which is which. The “model” elements are displayed in the diagram; the execution-related elements are not.

The BPM Suite vendors simply adopted the *modeling* parts of BPMN 1.x – the diagram – and ignored the *execution* parts. Executable details could be added to the process model, but each BPMS would do this in its own way. Thus BPMN 1.x – as implemented by the majority of modeling tool and BPMS vendors – is not by itself executable, although it is incorporated in many vendor-specific executable design tools. And that suits the vast majority of process modelers just fine. Few are even thinking about execution in a BPMS, anyway. They are business analysts and process architects, not developers.

BPMN 1.x failed, however, to deliver on a key promise, interchange between modeling tools. It's amazing that BPMN has achieved wide adoption without that, but somehow it was never a priority at either BPMI.org or OMG. Standardizing the XML serialization, based on a formal *metamodel*, would be a key goal of BPMN 2.0.

In fact, OMG originally intended simply to take its own *Business Process Definition Metamodel* (BPDM) and rebrand it as BPMN 2.0. It would de-emphasize the graphical notation and focus on abstract semantics that could be mapped to *any* process modeling language. However, that was a mistake. It would not only abandon existing BPMN 1.x users but did not sit well with IBM, Oracle, and SAP, who needed to bridge the gap between SOA and business-oriented BPM. They wanted to *extend* the BPMN 1.2 notation, popular with business, to include executable design. In the end, their rival proposal [7] carried the day.

In the world of BPM tools, BPMN 2.0 marks a tipping point. BPMN 1.x adoption was spearheaded by the small BPMS “pureplay” vendors. With BPMN 2.0, the biggest software companies in the world are leading the charge. Today any other notation is seen as “proprietary” or “legacy.” Somehow, against all odds, BPMN has become *the* important standard in BPM.

Business Process Modeling Is More Than BPMN!

Business architects and other BPM practitioners never cease to remind me that the activity flow logic, as defined by BPMN, is only one component of the modeling needed to properly describe, analyze, transform, and optimize a company's business processes. I don't disagree with this. BPMN really just describes the sequencing of process activities. That encompasses quite a lot, but admittedly a lot more information is needed to do BPM properly.

So what is missing? I asked Brett Champlin, president of the Association of BPM Professionals (www.abpmp.org) and a BPM practitioner himself at a major insurance company, what additional information should be modeled to support a process manager in an enterprise BPM program. He gave me a long list, which I have rearranged as follows:

Enterprise or Line of Business

- High-level business context, describing the business's relationships to Competitors, Regulators, Suppliers, Business Partners, Customers, Community, etc.
- Strategic Objectives and Performance Metrics
- Controls and Constraints
- Markets, Customers
- Products (goods and services)
- Locations

Operational, Cross-Process

- Value Chains and Process Portfolios
- Operational Goals and Objectives
- Policies
- Performance Metrics and KPIs
- Organizational Structure and Roles.

Process-Specific

- Activity resource requirements
- Revenue and Costs, both activity-based and resource-based
- Job Aids (instructions for human performers)

Technical

- IT Systems
- Services
- Data

Each of these items can be described by one or more models and attachments, linked in some relationship to the BPMN model. The fact that BPMN itself does not include them is not, in my view, a deficiency. In fact, a single vendor-neutral standard describing all of these models and their interrelationships would be nearly impossible to create. A key reason why BPMN is so widely accepted as a standard is that it does not attempt to do too much.

BPM at the enterprise level requires a suite of tools built around a *repository*, a database that maintains the relationships between all of the different models, along with governance and change impact analysis. So-called *Business Process Analysis (BPA)* suites link the process model to models of business rules, organizational roles, strategic goals and problems, and master data. *Enterprise Architecture* *****ebook converter DEMO Watermarks*****

(EA) suites link the BPMN to technical models and executable artifacts. Today, many BPA and EA suites are either replacing their legacy process modeling tools with BPMN or adding BPMN as an alternative format.

Chapter 2

How Does A Model Mean?

A process model is more than a drawing. Its purpose is to convey meaning, specifically the logic of the activity flow from process start to end. From the diagram alone, the process logic should be clear and understandable to a business person but semantically precise, as required by a developer. By *process logic*, we mean a description of all the paths from a single initial state of a process instance to each of its possible end states.

The BPMN specification and most books about it focus on classifying the BPMN elements in isolation, defining the meaning of each shape and symbol. But, as John Ciardi wrote in his classic, *How Does a Poem Mean?*, “the language of experience is not the language of classification.” Effectively communicating the process logic requires understanding how the elements fit together, not just as isolated words but as sentences, paragraphs, a complete story. That requires attention to the overall structure of the model, following a consistent set of conventions, what I call Method and Style. If you do that correctly, the most important features of the process are obvious at a glance: what the instance represents, how the process starts, its various possible end states and their corresponding status messages, and its touchpoints with external entities.

The BPMN diagram is both a visualization and a data entry device for the underlying XML *semantic model*. When you draw a diagram, the tool transparently translates each shape into its corresponding semantic element: a start event, a User task, an end event, etc. In the BPMN 2.0 specification, the BPMN metamodel, element definitions, and associated rules all reference the semantic elements, not the shapes in the diagram. In fact, BPMN allows a semantic model without an associated graphical model. That is, the process logic is defined in the XML, but there is no diagram. However, the converse is not true: In BPMN 2.0, you cannot have a graphical model without an associated semantic model.

A computer may be able to comprehend complex process logic expressed as pages of XML, but people cannot. We need the diagrammatic representation to make sense of what is going on. But here's the problem: Just a tiny fraction of the information defined by the semantic model is visible in the diagram: the basic element type, indicated by its shape and associated icons and markers, and a text label. If we are viewing a multi-page diagram within a BPMN tool, hyperlinks can indicate certain relationships between the pages, and property sheets can display various attributes of a selected shape. But we cannot assume that the diagram is always accessed through the tool. In most cases users view the BPMN diagram as a hard copy or pdf, in which the hyperlinks and property sheets do not appear.

That means we need to convey as much meaning as possible from the diagram by itself, as it would appear in printed form, where all we have are shapes and labels. *Labels* are very important. A key piece of the Method and Style approach deals with consistent labeling, so the process logic is not only clear on *****ebook converter DEMO Watermarks*****

the page but traceable from page to page in a hierarchical model.

We don't want to guess the modeler's intent. It should be obvious from the diagram alone. That's what we mean by "good BPMN," and fortunately, it is a readily learnable skill.

BPMN's Hidden Conceptual Framework

While BPMN is widely adopted, few process modelers know how to use it correctly or effectively. Bad BPMN is the norm rather than the exception. One reason is the BPMN specification itself. It fails to explain clearly the meaning of BPMN's most fundamental concepts, like *activity* or *process*. That failure creates problems not only for beginning process modelers but for experienced business process architects.

What Is an Activity?

Let's start with *activity*. An activity in BPMN is an *action*, a unit of work performed. It is the only BPMN element that has a *performer*. But the meaning of a BPMN activity is more specific than that. A BPMN activity is an action that is performed *repeatedly* in the course of business. Each *instance* of the activity represents the same action (more or less) on a different piece of work. The modeler needs to have clarity on the *meaning* of the activity instance, such as an order, a service request, or a monthly review.

A BPMN activity is a discrete action with a *well-defined start and end*. Once an instance of the activity has ended, it's over, complete. It's not just lying dormant, ready to suddenly reawaken and do a bit more if it discovers something wrong. It is possible for the process to do those things... but in a different activity, or possibly another instance of the same activity.

In the broader realm of BPM architecture, the term "activity" is used more broadly, and this causes some confusion regarding BPMN. Some "activities" described by BPM architecture do not fit BPMN's definition because they are really *functions* performed *continuously*, not discrete actions performed *repeatedly*. They often have names like *Manage X* or *Monitor Y*, and don't operate on instances with a well-defined start and end.

What Is a Process?

Similarly, a *process* in BPMN is a sequence of activities leading from an initial state of the process instance to some defined end state. The start of a process is marked by a triggering event, such as receipt of a request. The *process model* is a map of all the possible paths – sequences of activities – from that initiating event to any defined end state, success or exception. Like activity, a process is discrete not continuous. It is performed repeatedly in the course of business, and has a well-defined start and end. Each instance of the process follows some path in the process model from start to end.

Like activity, BPMN's definition of a process is sometimes at odds with the term "process" as used by BPM or enterprise architects. For example, enterprise BPM often refers to *business process frameworks*, such as SCOR, ITIL, or eTOM, that enumerate the major processes and activities for a particular industry,

typically for cross-company benchmarking^[8]. One organization, called APQC, publishes a cross-industry Process Classification Framework^[9], a hierarchy consisting of Categories, Process Groups, Processes, and Activities. Unfortunately, very few of the processes and activities listed in the PCF match

BPMN's notion of process and activity. Most are of the *Manage X* variety, ongoing business functions rather than actions on discrete instances with well-defined start and end.

For example, below is a brief excerpt from the PCF for the process called *Process Expense Reimbursements* [10]. Here three-digit headings represent processes and four-digit headings represent activities.

- 8.6.2 Process expense reimbursements (10757)
 - 8.6.2.1 Establish and communicate expense reimbursement policies and approval limits (10880)
 - 8.6.2.2 Capture and report relevant tax data (10881)
 - 8.6.2.3 Approve reimbursements and advances (10882)
 - 8.6.2.4 Process reimbursements and advances (10883)
 - 8.6.2.5 Manage personal accounts (10884)

Within a process, activity instances need to align with each other and align with the process instance as well. If we interpret 8.6.2 as the BPMN process of handling employee expense reports, that is certainly not the case here. The first activity is really two separate BPMN activities, since establishing the policies and communicating them probably occur at different times and frequencies. Also, neither one of them is part of this process. The second activity is also two BPMN activities. Tax data might be captured with each expense report, but reporting it to the government would be done quarterly or annually. The next two could possibly be BPMN activities in this process, assuming processing advances and reimbursements use the same process. The last one is an ongoing function, not a BPMN activity at all.

From BPMN's conceptual framework, a better activity list for processing employee expense reimbursements might be as follows, where an instance of each activity is a single expense report:

- 8.6.2 Process expense reimbursements
 - 8.6.2.1 Review expense report and supporting documentation
 - 8.6.2.2 Approve reimbursement
 - 8.6.2.3 Capture tax data
 - 8.6.2.4 Issue payment

My intent here is not to pick on APQC, specifically. The problem is rampant in the literature of business architecture and enterprise BPM. I have come across situations where a BPM architecture team has defined a list of major "activities" that are not discrete actions performed repeatedly on instances with well-defined start and end points, and has then tasked process modelers with wiring them together to describe end-to-end processes. But it is impossible.

Process Logic

When a process modeler begins to document an as-is, or current-state, process, the procedure typically involves meeting with the people directly involved with the process, so-called subject matter experts. And the SMEs might be inclined to describe the process like this: *First X happens, and then it typically goes to Y, and then finally we do Z.* That's fine. It describes what *usually* happens, leading to a successful end state. Or maybe that's just how it happened in one recent instance.

But the process model is more than documentation of one instance of the process. It is a complete map of all the paths from the triggering event to any defined end state. That does not mean every conceivable possibility, no matter how remote, only those end states and paths that occur with significant frequency.

So the first questions for the SME should be things like this:

How does the process actually start? What event triggers it? Is there more than one possible way it could start?

What determines when the process is complete? Are there different end states for the process, such as one signifying successful completion and others signifying failed or abandoned attempts?

How does the process get from X to Y? Does the person doing Y somehow just "know" it's supposed to happen? You said it "typically" goes to Y, but where else might it go? And why?

How do you know when X is done? Does X always end in the same way? Or besides the normal end states are there exception end states where you don't go on to Y? Are there rules that govern this?

Answers to these questions define the *process logic*. The process logic defines all possible sequences of activities from the process's initiating event to one of its end states. Each activity is represented in the diagram by a rounded rectangle, and solid arrow connectors called *sequence flows* describe the possible flow paths. There may be branch points in the flow, where an instance could take one path or another based on some condition. There is a diamond shape in BPMN for that, called a *gateway*, and the labels on the gateway and its outgoing connectors show that conditional logic on the process diagram. BPMN also has circle shapes, called *events*, that can divert the flow when some exception occurs or some external message arrives. In fact, all of the process logic in BPMN is composed of just these three primary *flow nodes* – activities, gateways, and events – and the sequence flows connecting them. Each end of a sequence flow must connect to an activity, gateway, or event.

The SME's first reaction to these questions might be, "Nothing is *making* the process go from X to Y. That's just what happens." Of course, something is *always* making it go. The logic is just hidden, probably inside the head of whoever happens to be doing X for that particular instance. And there is tremendous value in surfacing that logic, making it explicit in a diagram that all stakeholders in the process can understand. Without that, you can't really *manage* the process or improve its performance.

Orchestration

BPMN only describes processes in which the process logic – the map of all possible paths from triggering event to one of the process's end states – is *explicit*, defined in advance of the triggering event. BPMN is a language for specifying that explicit process logic. Every instance of the process must follow some path in the process model. BPMN's technical term for such a process is an *orchestration*. In the BPMN 2.0 specification, the terms *process model* and *orchestration model* mean the same thing.

It is reasonable to ask, "*How can the process logic be defined in advance when an Approval is completely arbitrary?*" Ahh, but how the performer decides to approve or reject is not part of the *process logic*. It is part of the internal *task logic* of the Approval step. For most activity types, BPMN does not describe the task logic, only the process logic, the logic of *what happens next when the task is complete*. The process logic says, "If Approval ends in the state *approved*, follow this path; if it ends in the state *rejected*, take this other path." So orchestration does not mean you know in advance the particular path an instance is going to take, only that the *conditions* for taking any possible path in the model are known in advance.

In contrast, a purely *ad-hoc process* is not an orchestration. By ad-hoc, I mean a process in which the performer of each task determines the task to perform next, and the list of possible next tasks is wide open, not enumerated in the model. (If the list could be enumerated in advance, you could just show them all in the diagram, and let the task end state determine which path to follow. Some so-called ad-hoc processes are like this, in fact. BPMN is not a good fit for them, not because it cannot describe the behavior but because the resulting diagram would be difficult to understand and not worth the modeling effort.)

The path taken by any process instance depends on *information* accumulated by the instance as it progresses. That information includes messages received, data produced in process activities, and the end states of completed activities. BPMN implicitly assumes that all this instance data is available to the process logic. With this information, the process model "knows," as each step is completed, where the instance is going to go next. You might even think of the process model as an intelligent force that "guides" the instance from step to step. It is a very short leap from there to an actual process engine in a BPM Suite. Even though the vast majority of BPMN models do not describe automated processes, BPMN treats the process as if it could, in principle, be automated. This helps explain why it is so important that instances of each process activity are aligned with each other and with the process instance itself.

Remember that BPMN originated as a graphical design language for automated process flow. In most processes modeled in BPMN, the process logic is not automated ... but BPMN treats it as if it could be.

The Questions BPMN Asks

In my BPMN training, a student once asked me how to show in the diagram that a certain activity normally completes in five hours. I replied that that is not a question that BPMN asks. Instead, BPMN wants you to say *what action occurs if the activity is not completed in five hours?* Do you send a reminder? Notify the manager? Escalate the task? Cancel and abandon the process as a whole? Those are things that BPMN describes. They are part of the process logic; the average time to complete is not.

A BPMN process model reveals only the *order* of activities, *when* they happen, and under what conditions. It describes what happens next when an activity completes, but may have little to say about what happens inside the activity itself. It does not describe *how* an activity is performed or *where* or *why*. In fact, BPMN barely touches on *what* the activity is or *who* performs it. Those are simply suggested by labels on activities and swimlanes in the diagram. In fact, BPMN has been criticized for omitting this information from process models... often by the same vendors and consultants who complain that the BPMN notation is too complex! It's not that those other questions are unimportant, but they are not part of the process logic, and thus remain outside the domain of BPMN.

It is important to keep in mind that, as a multi-vendor standard, BPMN is a negotiated agreement among many competing interests. In order to get anything at all through the committees, its scope is narrow by necessity. Many BPMN tools do include, in fact, models of organizational roles and groups, problems and goals, simulation parameters, KPIs, and the like, but these models are tool-specific. The only process information described uniformly across tools is the BPMN process logic.

BPMN Levels and Process Modeling Conformance Subclasses

I have been conducting BPMN training since early 2007. I can say from experience that not everyone who wants to learn BPMN is interested in the same level of process detail. While the language excels at expressing exception handling and other event-triggered behavior, to some modelers that is just extraneous clutter; they don't care about it. And they don't see the need for all the subtypes of activities, gateways, and events in the full BPMN element set. In fact, only a small fraction of the full element set is commonly used.

Thus, my training always started out by restricting models to a limited working set of the shapes and symbols that we called the *BPMN Level 1* palette. Day 1 was, and remains, Level 1 only. Business users easily understand it, and it makes learning the basics of BPMN easier. Moreover, it is a palette that almost every BPMN tool supports. The Level 1 palette is essentially the shapes and symbols carried over from traditional flowcharting, and it is sufficient for describing most process behavior in a compact business-friendly way. In fact, if you are willing to ignore behavior triggered by timeouts and the arrival of external messages, it may be all the BPMN you ever need.

On Day 2 of the training, we move on to exception handling, with emphasis on Message, Timer, and Error events, plus some additional branching and merging patterns. This requires a slightly larger palette we call *BPMN Level 2*. Since event-triggered behavior is a fact of life in real-world processes, business analysts who want to use BPMN for defining solution requirements need to learn BPMN Level 2. Even though the BPMN Level 2 palette encompasses only about half of the full BPMN 2.0 working set, many BPMN tools still don't support all of it.

Both Level 1 and Level 2 concern *non-executable processes* and rely solely on information visible in the diagram. *Executable BPMN*, in contrast, is all about the XML details that are *not* displayed in the diagram, like data models, conditional data expressions at gateways, and detailed task assignment logic. I call it *BPMN Level 3*; as of this writing, it is still not part of the training. Both Level 1 and Level 2 omit these details. Not only are they not represented in the diagram, but until BPMN 2.0 there was no standard XML representation for them. Consequently, their definition has always been tool-proprietary. Today, with BPMN 2.0, you can do Level 3, that is, define executable process logic using the XML elements defined in the BPMN standard. But tools that do that are just now getting off the ground. As of this writing, just a few have the basics of Level 3 working, and none yet include all of the elements of the Level 2 palette. We will discuss Level 3, or Executable BPMN, in Part V of this book.

Thus BPMN levels originated as a pedagogical strategy in my BPMN training. Even though they were not part of the BPMN specification at the time, OMG included my explanation of the levels as "reference material" for its OCEB BPM certification exam. But it turns out that levels have a second value, important to tool vendors: By limiting the palette of supported shapes, they make model interchange possible. In the end, that is what led to their inclusion in the final BPMN 2.0 specification!

When I left the BPMN 2.0 technical committee in June, 2009, the specification draft said that in order to claim Process Modeling Conformance, a tool had to support the entire set of BPMN process model shapes and symbols. While that might be possible for a pure modeling tool, it was never going to allow interchange with tools used for executable design. Practical BPMN interoperability between tools demands, first and foremost, restricting the working set of shapes and symbols. If a tool vendor could limit import/export to the Level 1 working set, it would be far easier for that tool to interoperate with others.

Even though model interchange was always an explicit goal of BPMN 2.0, the vendors in charge of the spec drafting process were reluctant to commit to a real test of compliance. As a member of the technical committee, I tried very hard to get the levels included in the Conformance section of the June 2009 beta specification, but without success. But Robert Shapiro managed to push them through in the Finalization phase, and they are now officially part of the BPMN 2.0 final specification!

In the spec, Level 1 is called the *Descriptive Process Modeling Conformance subclass*, and Level 2 is called the *Analytic Process Modeling Conformance subclass*. A few BPMN elements switched levels, so if you compare the current edition of this book with the original you will see some minor changes in the palettes. In this edition, the Level 1 palette has been adjusted to match the official Descriptive subclass exactly, and the Level 2 palette matches the official Analytic subclass. There is also a third subclass, called the *Common Executable Process Modeling Conformance subclass*. We'll talk about that one in Chapter 19.

In the specification, members of each subclass are defined in terms of specific XML elements and attributes. You should not be surprised that those elements and attributes represent only the information visible in the diagram: the element type and its icons, markers, border styles, and labels – plus the unique ids and id references needed to hold the model structure together. All the details needed to make the process executable – definitions of data, gateway conditions, messages, services, and task assignment – are outside of the Descriptive and Analytic subclasses.

I believe this reinforces the basic premise of the Method and Style approach: *For non-executable process models, it's the notation – what you see in the diagram – that really matters.* Another way of saying it is this: *If it's not in the diagram it doesn't count.* Method and Style shows you how to convey as much meaning as possible from the BPMN shapes, symbols, and labels alone. To achieve that, Method and Style obeys the rules of the BPMN 2.0 specification but imposes additional conventions on the modeler to ensure the diagram's meaning is unambiguous.

The second half of this book, the BPMN Implementer's Guide, shows tool vendors and developers how to translate that meaning, as reflected by the diagram, into XML that can be imported and understood by any tool supporting the Analytic subclass. If a given diagram has one and only one serialization, then interchange of that model between tools becomes straightforward and automatic.

*****ebook converter DEMO Watermarks*****

Part II: Method and Style – Level 1

Chapter 3

BPMN by Example

A Simple Order Process

Consider the process to handle an order. The company receives the order, checks the buyer's credit, fulfills the order, and sends an invoice. In simplest terms, that looks like this in BPMN:



Figure 3-1. Basic order process

The thin circle at the start of the process is called a *start event*. It indicates where the process starts. The thick circle at the end is called an *end event*, signifying the process is complete. The rounded rectangles are *activities*. An activity like *Check Credit* represents an *action*, a specific unit of work performed, as distinct from a *function* (e.g., *Credit Check*) or a *state* (e.g., *Credit OK*). To reinforce this, activities should have names of the form VERB-NOUN. An element's *name* in the XML is displayed as the *label* of the shape in the diagram.

Exceptions and End States

This diagram does not yet represent a process model. It is just a simple description of the *happy path*, the normal sequence of activities when no exceptions occur. What exceptions could occur? Well, the buyer's credit might not be sufficient, or the goods might not be in stock. Those situations would represent failed orders. So a more complete model of the process might look like this:

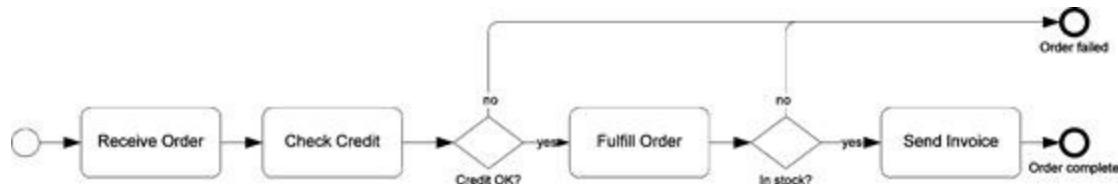


Figure 3-2. Order process with exception paths

The diamond shapes are called *gateways*. They represent branch points in the flow. BPMN provides a number of different gateway types, but this one – the *exclusive data-based gateway* (also called *XOR gateway*), a diamond with no symbol inside – means take one path or the other based on some data condition, such as *Is the buyer's credit OK?* or *Are the order items in stock?* The diagram communicates the process logic by the combination of the gateway label and labels on the sequence flows out of the gateway, called *gates*. Gateways are a common way of splitting *exception paths* from the happy path.

Note we now have two end events, one labeled *Order failed* and the other *Order complete*. BPMN does not require multiple end events like this, but a Method and Style principle requires using separate end events to indicate distinct *end states*, such as one representing success and the other failure, and labeling each with the name of the end state.

Also notice that the diagram now describes three distinct paths from beginning to end. Not all of the model's activities are performed for every instance of the process. If the credit check fails, for example, we do not fulfill the order. If the order items are not in stock, we do not send the invoice. This is common sense, and the BPMN diagram indicates this explicitly.

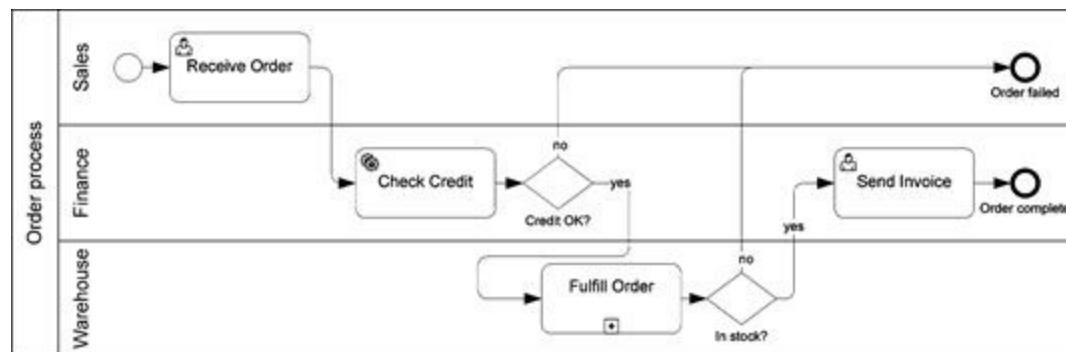


Figure 3-3. Order process in swimlanes

Swimlanes and Activity Types

BPMN also lets us indicate the performer of each activity, using *swimlanes*, or to use the BPMN term, *lanes* (Figure 3-3). Lanes usually represent roles or organizational units that perform activities in the process. They are drawn as subdivisions of the rectangle containing the process, called a *pool*. You sometimes see pools labeled with the name of an organization, but for pools that contain activity flows – some don't, as we will see later – it's best practice to label them with the name of the process.

We can also indicate the *type of activity* through icons and markers inside the rounded rectangle. It is generally useful to distinguish human tasks from automated ones, and these are indicated in the diagram by different *task type icons*. In Figure 3-3, *Receive order* and *Send invoice* are human tasks, called *User tasks* in BPMN. *Check credit* is an automated task, called a *Service task* in BPMN. Automated means executed with no human intervention. If a person pushes a button once and the rest of the task is automated, that is a User task, not a Service task.

Lanes really apply only to User tasks; we can place gateways and events in whatever lane is convenient. Some people like to put Service tasks in their own lanes as well, either one lane for all systems or one lane per system. I tend not to do that, but it's a matter of personal preference.

Subprocesses

What type of activity is *Fulfill Order*? It does not have an icon representing a human or automated task, but a little [+] marker instead. That is a *subprocess*, one of BPMN's most important concepts. A subprocess is an activity containing subparts that can be expressed as a process flow. In contrast, a *task* is an activity with no defined subparts.

A subprocess is simultaneously an *activity*, a step in a process that performs work, and a *process*, a flow of activities from a start event to one or more end events. In the diagram, a subprocess can be rendered either *collapsed*, as a single activity shape, or *expanded* as a process diagram in its own right. BPMN tools typically let you toggle or hyperlink between those two views, allowing zoom in and out to view the process diagram at any level of detail.

One way to represent the expanded view of a subprocess is *inline* in the diagram, as in Figure 3-4. With inline expansion, the process flow is enclosed in an *expanded subprocess shape* (a resizable rounded rectangle). Figure 3-3 and Figure 3-4 mean exactly the same thing, but Figure 3-4 provides an additional level of detail. Note that the expanded view of *Fulfill Order* looks just like a process. It has a start event, a flow of activities, and an end event for each distinct end state. The start of the *Fulfill Order* process is triggered by the sequence flow into the subprocess, which, we can see from the diagram, occurs after *Check Credit* whenever the credit is OK. When the sequence flow arrives at *Fulfill Order*, it continues immediately from the start event of the expansion. When *Fulfill Order* completes, the process immediately continues on the sequence flow out of the subprocess.

In Figure 3-4 we also see the benefit of using multiple end events to distinguish end states, in this case the *Out of stock* end state and the *In stock* end state. By matching the label of the gateway following the subprocess (*In stock?*), it is clear the gateway is asking the question, "Did we reach the *In stock* end event?" Matching the label of a subprocess end state with the label of a gateway immediately following the subprocess is an important Method and Style convention.

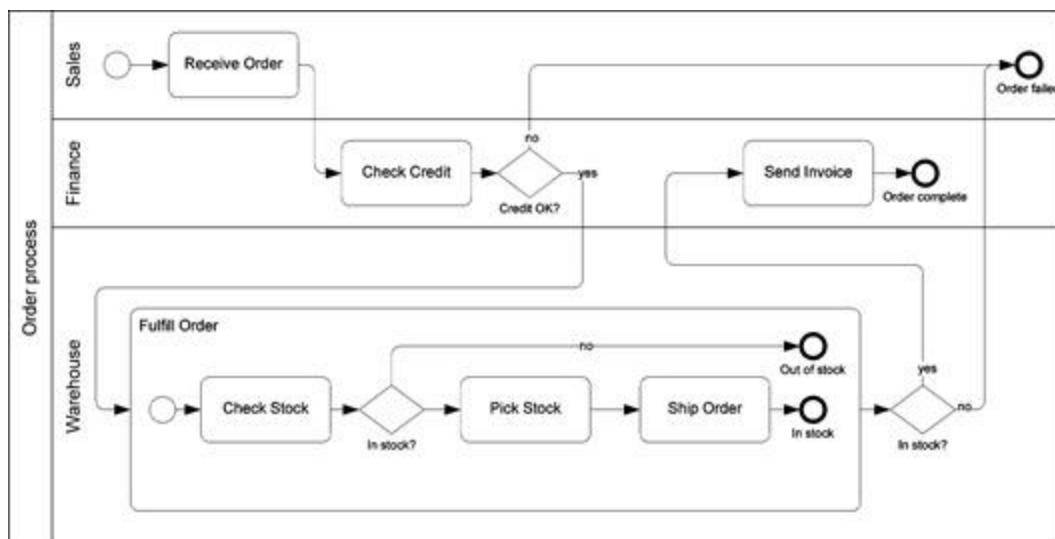


Figure 3-4. Order process including expanded subprocess

Process Levels and the Hierarchical Style

The process depicted inside the *Fulfill Order* activity in Figure 3-4 represents a *child process level* with respect to the level including the overall process start and end and the *Fulfill Order* subprocess, shown in Figure 3-3. The child level could itself contain subprocesses, and there is no limit to the number of levels you can nest in this way.

Inline expansion, as in Figure 3-4, depicts the parent and child levels in the same diagram, but it is not the only way to render the child-level detail. In fact, with most tools, except in simple cases, it is rarely the best way. Note that Figure 3-4 takes up a lot more space on the page than Figure 3-3. For end-to-end processes, showing all the subprocess details on a single page usually isn't possible. One solution is to use *off-page connectors* to link to a continuation of the process level on another diagram. BPMN provides a notation for this, called a *Link event pair*.

But I recommend a different way: depicting the child-level expansion in a separate diagram. I call it *hierarchical expansion*, because it expresses the end-to-end process as a hierarchy of diagrams. In the tool, the parent and child-level diagrams are *hyperlinked* together, but we cannot rely on hyperlinks when the model is printed to paper or pdf. In that case, we need to rely on *matching labels* to link the diagrams together. Let's see how it works, and then talk about why it's the preferred way.

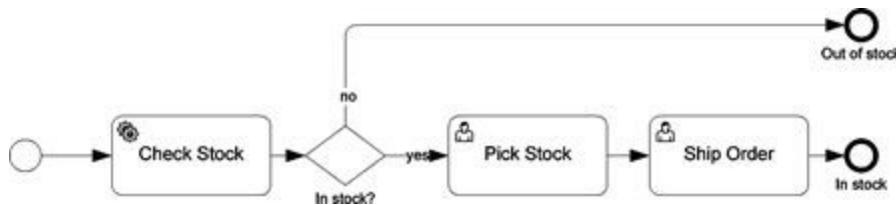


Figure 3-5. Subprocess expansion on a separate page

Figure 3-5 shows the expansion of *Fulfill Order* in the child-level diagram. Note that it omits the pool shape, which is inherited implicitly from the parent. Remember, this is not a new process but a subprocess of *Order Process*. The child-level diagram also omits the expanded subprocess shape surrounding the flow. A child-level expansion may contain lanes, although none are represented in Figure 3-5. If lanes are absent in the child level but present in the parent level, it is implied that activities in the child level inherit the lane of the collapsed subprocess in the parent level. But technically, lanes are defined independently at each process level.

While inline expansion is useful in simple diagrams, in most cases I prefer the hierarchical style. One reason is it allows the top level of a complex process to be represented end-to-end on a single page. That top-level view provides little detail about each major step of the process, but it does reveal at a glance all the possible paths connecting those steps, the meaning of the process instance, how the process starts, its possible end states, and its interactions with external entities. In other words, it expresses on a single page the "big picture" of the end-to-end process.

From the top-level diagram you can then drill down into each child-level subprocess and view its details in a separate linked diagram, which can in turn drill down further to a deeper child level, and so on. With hierarchical modeling, additional detail is provided in layers, and you can zoom in to view detail at any level without losing the integrity of a single end-to-end model. Even though the model is represented visually as separate pages, in the XML it is a single model. That is far better than maintaining

separate high-level and detailed models, and keeping them in sync as the process logic changes over time.

The hierarchical style does add a bit of complexity when viewing the diagrams, since parent and child levels appear on separate pages. For example, with inline expansion (Figure 3-4) the end event *In stock* and the gateway *In stock?* appear on the same page, while in the hierarchical style (Figure 3-3 and Figure 3-5) they appear on separate pages. Once you get used to the hierarchical style, mentally connecting the diagrams becomes easy.

The child-level diagram represents the activity flow *inside* the subprocess. One mistake beginners make is replicating, inside the child-level expansion, activities that occur either *before* the subprocess starts or *after* it ends. For example, Figure 3-6 is *incorrect* as a child-level expansion of *Fulfill Order*:

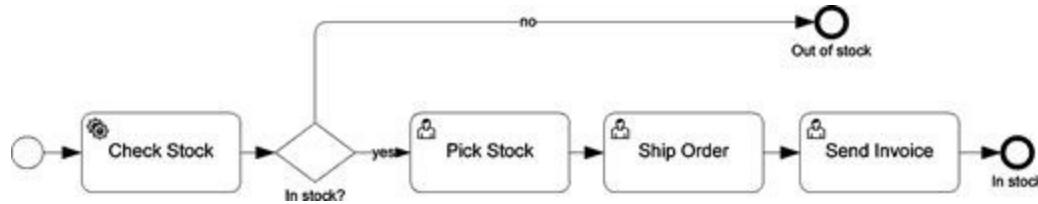


Figure 3-6. Incorrect expansion of *Fulfill Order*

The reason is *Send Invoice* is not part of the subprocess. In the parent level diagram (Figure 3-3), it comes after *Fulfill Order* is complete. Modeling the child level as in Figure 3-6 means the invoice is sent twice, once within *Fulfill Order* and then again afterward. That was not the modeler's intent. Remember, when the child level is complete, the flow immediately continues on the sequence flow out of the collapsed subprocess at the parent level.

Taking another look at Figure 3-3, you might decide that simply ending the process when a requested item is out of stock is not the best way to handle this exception. Perhaps you would contact the customer and offer a replacement item, and if the customer accepts the offer, go on to fulfill the order. That would look something like this:

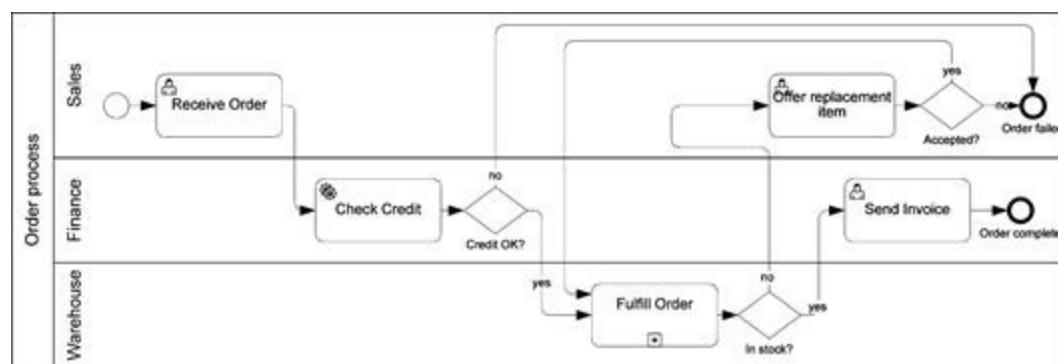


Figure 3-7. Loopback to handle exceptions

In BPMN, unlike block languages such as BPEL, a sequence flow may freely loop back to a previous step. In Figure 3-7, if the replacement offer is accepted, a gateway directs the flow back to *Fulfill Order*. Remember the process is not complete until an end event is reached.

The BPMN spec does not place any significance on whether a sequence flow enters an activity from the left, right, top, or bottom, nor even whether pools and lanes run horizontally or vertically. These are really matters of personal style. I usually try to draw the flow left to right with sequence flows entering

activities from the left and exiting from the right. It takes some rearranging to keep line crossings at a minimum, and sometimes that cannot be avoided. But keeping the diagram as neat and consistently organized as possible is important to the objective of shared understanding. Nothing is more frustrating than looking at a diagram someone else has created and being unsure where exactly the process starts and ends.

Parallel Split and Join

Now let's consider one last detail of our *Fulfill Order* subprocess. In order to expedite shipment, we'd like to make the shipping arrangements concurrently with picking the stock, that is, in parallel. We originally considered making these arrangements to be part of *Ship Order*, but technically that means we don't do it until after *Pick Stock* completes.

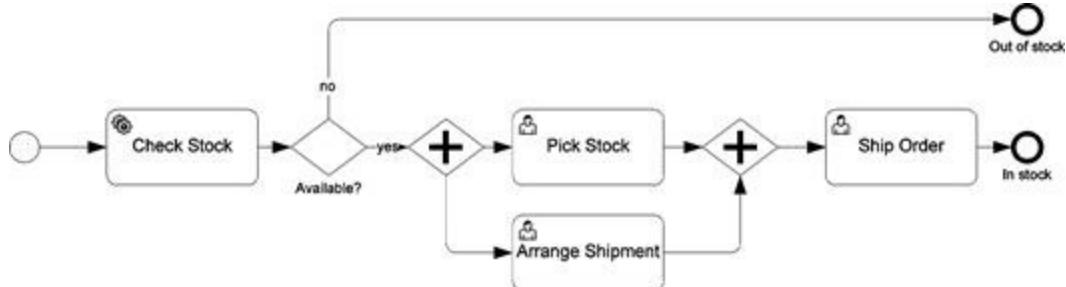


Figure 3-8. Parallel split and join

Figure 3-8 shows how it looks. Again it uses a gateway, in fact two of them, but with a symbol inside. A gateway with a + symbol inside is a *parallel gateway*, also called an *AND-gateway*. A parallel gateway with one sequence flow in and two or more out is called a *parallel split* or *AND-split*. It means unconditionally split the flow into parallel, i.e., concurrent, segments. Both *Pick Stock* and *Arrange Shipment* are enabled to start at the same time. If the same shipping clerk performs them both, they cannot literally be done simultaneously. Concurrent really means it does not matter which is done first.

We cannot combine this parallel gateway with the XOR gateway that precedes it (*Available?*) because they mean different things. The *Available?* gateway is an exclusive decision, meaning take one path or the other. After we take the *yes* path, then the AND-split says we do *Pick Stock* and *Arrange Shipment* in parallel.

The second parallel gateway, with multiple sequence flows in and one out, is called an *AND-join* or *synchronizing join*. It means wait for *all* of the incoming sequence flows to arrive before enabling the outgoing sequence flow. In plain English, it means *Ship Order* cannot occur until both *Pick Stock* and *Arrange Shipment* are complete.

Labels on AND-splits and joins (and sequence flows connecting them) add no new information, so it is best to omit them.

Unlike BPEL, BPMN does not require all the paths out of a parallel split to be merged in a downstream AND-join. They could even lead to separate end events. In that case, the process level is not complete until *all* parallel segments have reached an end event.

Collaboration and Black-Box Pools

It is not uncommon for experienced flowcharters, new to BPMN, to make the Customer a lane inside the process, and start the process with tasks in that lane like *Fill out order form* and *Submit order...* but that would be incorrect. Actually, the Customer is *external* to the process, not part of it. Think about an online store like Amazon.com. Have you ever put a book or some other item in your shopping cart but, in the end, decided not to order it after all? Of course you have! Now in that situation, have you created an instance of Amazon's order process? I think not. Amazon's order process starts when they receive the order, even though Amazon itself provides the shopping site. The order process includes securing payment, retrieving the order items from the warehouse, and delivering them to the Customer.

This is a fundamental point, and we will discuss it further, but for now please just accept that the requester of a process is usually best modeled as an *external participant*, not as a lane inside the process pool.

We model an external entity like the Customer as a separate *pool* in our diagram. But unlike the pool that contains the *Order Process*, the Customer pool is empty. It contains no flow elements whatsoever. We call it a *black-box pool* – meaning Customer's internal process is invisible to us. Technically, in the XML, a black-box pool represents a *participant* – an external business entity – that has no process. (It doesn't literally mean that the Customer has no defined buying process, but that the Customer's internal process logic is invisible to the Seller.) While we label a process pool with the name of a *process*, we label a black-box pool with the name of the *role or entity*, in this case *Customer* (Figure 3-9).

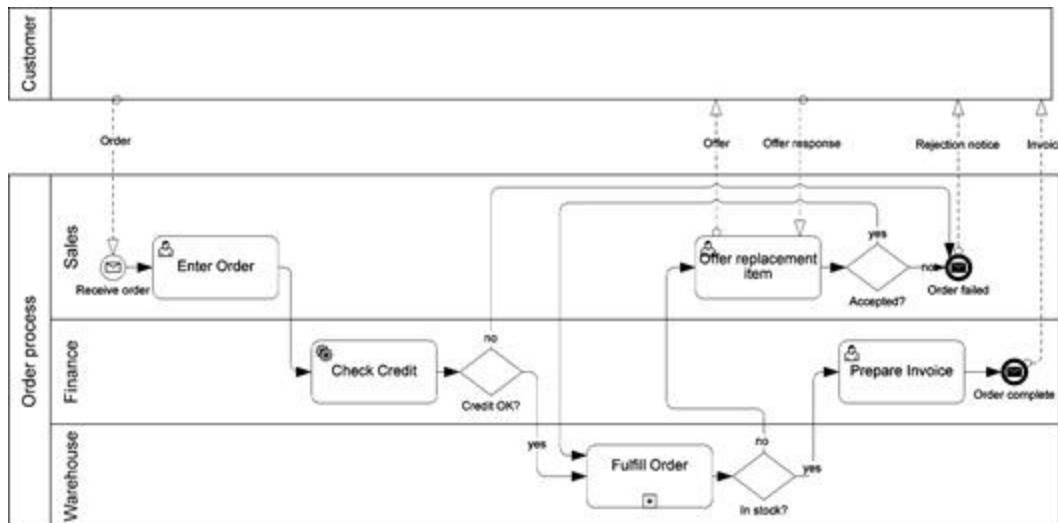


Figure 3-9. Order process in collaboration diagram

The Customer (like other external participants) interacts with the process by exchanging messages. In BPMN, the term *message* means any communication between the process and an external participant. We can indicate these communications in the diagram with another type of connector, called a *message flow*. A sequence flow is represented by a solid line connector and can only be drawn *within a pool*; a message flow, a dashed line with an unfilled arrowhead and little circle on the tail, can only be drawn *between two pools*.

In BPMN 2.0, Figure 3-9 is called a *collaboration diagram*. In addition to the activity flow of our

internal *Order Process*, it shows the interaction of our process with external participants by means of message flows. Note that the message flows attach to the boundary of the black-box pool and directly to activities and events in the process pool.

The envelope icon inside the start and end events indicate that these events receive and send messages. In BPMN terminology, the start event has a *Message trigger*, and the end event has a *Message result*. A *Message start event* has special meaning in BPMN, and we will see it again and again. It signifies that a new *instance* of the process is created upon receipt of a message, in this case *Order*. If a second *Order* message arrives immediately after the first, it creates a second instance of this process. You can only have a *Message start event* in a top-level process; a subprocess must have a *None start event*, meaning no trigger icon.

A *Message end event* signifies that the process sends a message when the end event is reached. In BPMN, the black event icons mean the process *sends* a signal, in this case a message; a white event icon means the process *receives* the signal. Here the process sends an *Invoice* message on reaching the *Order complete* end state, and sends a *Failure notice* on reaching *Order failed*.

Now, since it is the *Message start event* that is “receiving” the order, we will rename the first User task *Enter order*. Similarly, since the end event is now sending the invoice, we will rename the User task *Prepare invoice*. We don’t want to duplicate the action of a *Message event* with an activity that does the same thing.

Finally, we see message flows out of and into the human task *Offer replacement item*. A message flow signifies any communication between the process and external participants – a phone call, fax, or paper mail. Activities can send and receive message flows just as events can.

As we said at the start of this section, the idea that the Customer in Figure 3-9 is not part of the process, but external to it, is a surprise to many experienced flowcharters. But actually, this idea goes all the way back to the Rummler-Brache diagrams of the 1980s, what business people today call swimlane diagrams. Geary Rummler was one of the first analysts of business performance from a process perspective and a great influence on the management discipline of BPM. Paul Harmon, editor of [\[11\]](#) BPTrends and a former colleague of Rummler’s, recounts:

An IBM researcher took Rummler’s courses and was so impressed with the power of Rummler-Brache diagrams that he created an IBM process methodology called LOVEM. The acronym stood for Line of Vision Enterprise Methodology. The “line”, in this case, referred to the swimlane line at the top of a Rummler-Brache diagram that divided the customer from the process and allowed the analyst to see exactly how the process interacted with the customer.

Inherent in analysis of process performance is the interaction of a process with its “customer.” In Rummler-Brache and derivatives like LOVEM, the customer was drawn in the top swimlane, and communications across that line represented the customer’s perspective on the process. In BPMN, the notation has changed slightly – we show external participants in separate pools – but the concept remains the same.

The same modelers who initially want to make Customer a lane inside the process often insist on inserting activities like *Fill out order form* or *Submit order* inside the Customer pool. That is not only unnecessary but incorrect. A pool containing flow elements is, by definition, a process pool, not black-box. As such, it has to represent a *complete* process from start to end. So if you put an end event after

Submit order, how do you receive the replacement offer, rejection notice, or invoice? You cannot draw those message flows to the boundary of a process pool, only to the boundary of a black-box pool. To draw those message flows you would be forced to draw a complete buyer process for the Customer. But if you are the seller, do you even know the buyer's process? Probably not.

In my BPMN training, I normally leave the discussion there. But technically, BPMN 2.0 does define something called a *public process* (in BPMN 1.2 it was called an *abstract process*). A public process lies in between a black-box pool (no process) and a fully defined process (called a *private process*). A *public process* contains only activities that *send or receive messages*; all other activities are omitted. The intent is to represent the kind of message interactions defined in B2B standards like RosettaNet or ebXML. There the buyer and seller do not know the full details of each other's process logic, but the allowed types and sequences of message exchanges – like quotes, orders, confirmations, ship notices, and invoices – may be established in advance through industry standards and trading partner agreements. That rarely applies in BPMN collaboration diagrams, so except when the interaction is based on a defined B2B exchange pattern, you should simply use a black-box pool, not a public process, to represent the requester.

Start Events and the Process Instance

The Message start event in Figure 3-9 is significant in another way. A Message start event indicates that the process starts upon receipt of a *request*. Here the request takes the form of an order, but a loan application, an insurance claim, or customer service request are all examples of requests issued to a process provider. I recommend labeling Message start event *Receive [name of message]*, such as *Receive Order*. Not all processes are triggered by a request, but most are. A Message start event always signifies a process started by an external request, and the pool – usually black-box – at the tail of the message flow identifies the requester, in this case *Customer*.

A Message start event also signifies that the *process instance* represents the fulfillment of that single request. That in turn implies that each activity in the process is just related to that one request as well. In particular, it may not describe the fulfillment of other requests for the same process, such as another order. This is an extremely important point, and we will return to it in Chapter 8.

A Message start event always signifies that the process is started by *external* request, even when the requester is not a customer. It could be, for example, another internal process. Employee-facing processes are a gray area. Is the Employee external or internal to the process? It depends. Sometimes it is better to model the Employee as an external black-box pool, and other times better to make Employee a lane inside the process pool. Here are some guidelines, or rules of thumb, about modeling the requester.

1. If the process starts upon receipt of a form or other document and either
 - a. the requester has no further interaction with the process other than receiving some form of final result or status notice, or
 - b. the requester has occasional intermediate interactions with the process on an exception basis, but has no predefined process tasks to perform,

then model the requester as an *external black-box pool* sending a message flow to the process's Message start event.

2. If the requester has defined tasks to perform as a normal part of the process, model the requester as a *lane* within the process pool, and use a None start event (no trigger) for the process. There is no black-box pool for the requester in this case.

In Figure 3-10, the requester is external. Even though there are intermediate interactions with the process, they are on an exception basis.

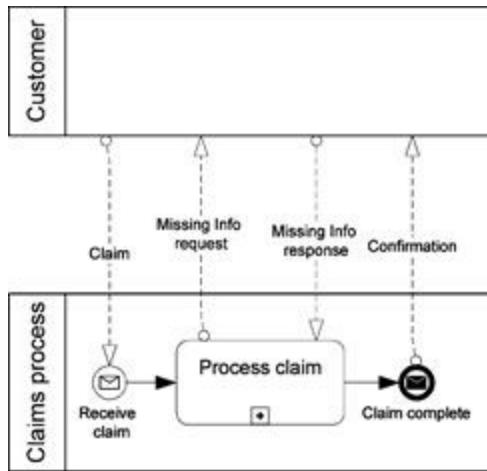


Figure 3-10. External participant as black-box pool

In Figure 3-11, the Employee has specific tasks to perform in the process, preparing the requisition and justification documents, securing management approval, and verifying the equipment is in working order when it arrives. Normally in this case you would use a *None start event*, as shown here, signifying manual start by a task performer.

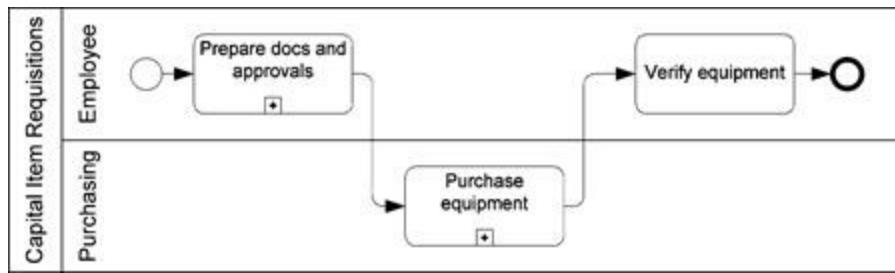


Figure 3-11. Initiation by an internal task performer

But if your focus is on what happens in Purchasing, the Procure-to-Pay process, you could just as easily make the Employee an external pool here, as in Figure 3-12. In that case the Employee is just another external requester. It's all a matter of perspective.

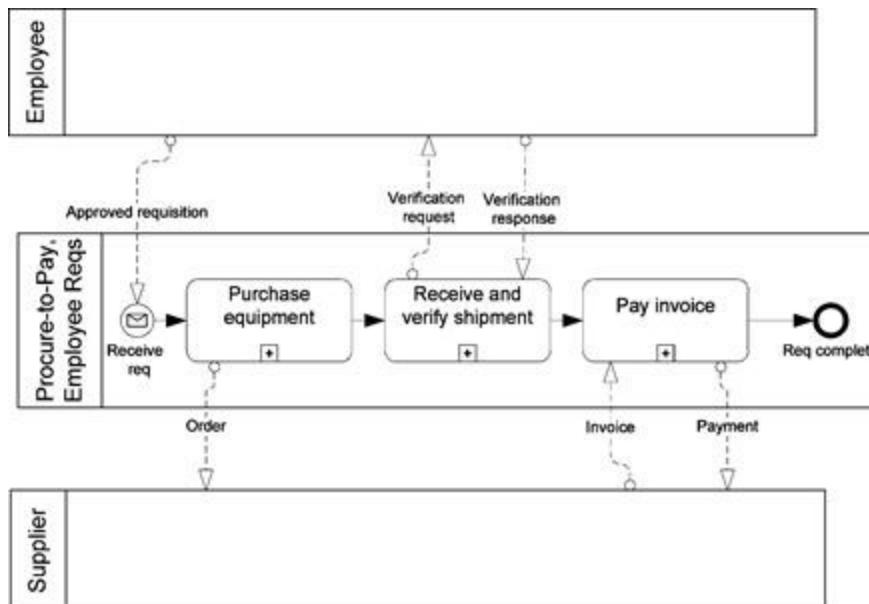


Figure 3-12. Another perspective on the Employee purchase requisition

The Top-Level Diagram

Let's take another look at what we have created so far in our Order process (Figure 3-13). At this point, we have a fairly complete top-level BPMN diagram. In this diagram, the details of *Fulfill Order* are hidden, but we can drill down to see the child-level expansion in a separate hyperlinked diagram.

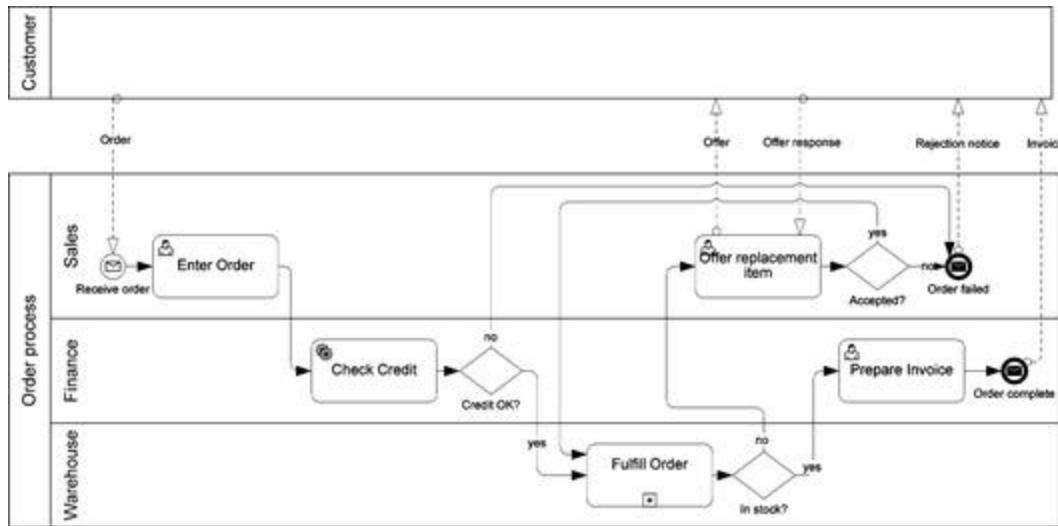


Figure 3-13. Order process, top-level diagram

But note how much the top-level diagram reveals about this process. We see that the instance represents an order, since it starts upon receipt of the *Order* message. It has two *end states*, *Order complete* and *Order failed*. The source of order failure is either bad credit or out of stock with non-acceptance of the offered replacement. Each end state returns a different *final status message* to the Customer. With processes initiated by a request message, it is good practice to return final status to the requester from message end events.

This is admittedly a simple process, but it is not too different from the top-level diagram of typical real-world end-to-end processes. All the shapes and symbols we used are members of the Level 1 palette. In the next chapter, we'll take a closer look at the complete Level 1 element set.

Chapter 4

The Level 1 Palette

All of the shapes and symbols used in the last chapter are part of the *Level 1 palette*, what BPMN 2.0 calls the *Descriptive Process Modeling Conformance subclass*. If you are willing to ignore event-triggered behavior, you can model almost any process without going beyond the Level 1 palette. With the exception of message flows and Message events, the Level 1 notation is basically carried over from traditional flowcharting.

The following is a complete list of the elements in the Level 1 palette, members of the Descriptive subclass in the BPMN 2.0 spec, including some we didn't use in the last chapter:

- Activity: Task (User, Service, None), Subprocess, Call Activity
- Gateway: Exclusive, Parallel
- Start event: None, Message, Timer
- End event: None, Message, Terminate
- Sequence flow and Message flow
- Pool and Lane
- Data object, Data store, and Data association
- Documentation
- Artifact: Text annotation, Association, and Group

In this chapter we'll review each of these elements. If you have done flowcharting before, you may find that BPMN's meaning is slightly different from what you are used to.

Activity

An *activity* represents a unit of work performed in the process. It is always represented by a rounded rectangle. It is the only BPMN element that has a *performer*. Every activity is either a *task* or a *subprocess*. A *task* is *atomic*, meaning it has no internal subparts described by the process model; the actions and end states of a task are merely suggested by its *name*. A *subprocess* is *compound*, meaning it has subparts defined in the model. Those subparts are modeled as a *child-level process*, an activity flow from start to one or more explicit end states.

Task

A *task* is represented in the diagram by the activity shape, rounded rectangle, with the *task type* indicated by a small icon in the upper left corner. A task represents an action, not a function or state. It should be labeled VERB-NOUN.

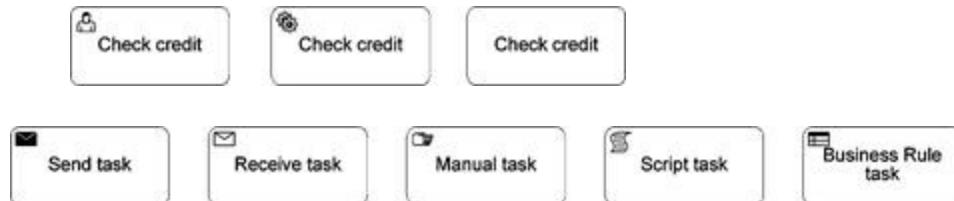


Figure 4-1. Top row, left to right: User task, Service task, Abstract task. Bottom row, left to right: Send task, Receive task, Manual task, Script task, Business Rule task

BPMN 2.0 defines eight task types, but the Level 1 palette just includes the three most commonly used (Figure 4-1, top row).

- A *User task* (left), with the head-and-shoulders icon, means a task performed by a person.
- A *Service task* (center), with the gears icon, means an automated activity. Automated means when the sequence flow arrives, the task starts automatically, with zero human intervention. If a person has to just click a button and the rest is automatic, that is a User task, not a Service task.
- An *Abstract task* (right), with no task type icon, means the task type is undefined.

Send and Receive Task

Send and *Receive* tasks are part of the Level 2 palette. They are similar to Message events and are discussed in Chapter 7. The others are outside of the Level 2 palette.

Manual vs. User Task

A *Manual task*, with the hand icon, should only be used in an executable process, that is, an automated workflow. In that context, a Manual task is one performed without any connection to the workflow engine, as contrasted with a *User task*, which is managed by the engine. *If your process model is not executable, it should not include Manual tasks.* For non-executable processes, just use a User task for any human task.

Script vs. Service Task

A *Script task*, with the scroll icon, also should only be used in an executable process. In non-executable processes, a *Service task* signifies any automated process activity, but in an executable process it means the process issues a *service request* to some external system or entity to perform that function. The implementation of the service is not defined by BPMN, but by the internals of the system that

*****ebook converter DEMO Watermarks*****

performs it.

A Script task, in contrast, means an automated function *performed by the process engine itself*. The implementation is a short program, typically Javascript or Groovy, embedded in the process definition XML. Because the process engine is usually busy executing the process logic, it does not have time to perform complex tasks, so Script tasks are typically used for simple computations such as data mapping.

If your process is not executable, it should not include Script tasks. For non-executable processes, just use a Service task (or equivalent Send/Receive pair) for any automated task.

Business Rule Task

The *Business Rule task*, with the grid icon, is new in BPMN 2.0. It signifies a task that executes a complex decision on a business rule engine. A Business Rule task is effectively a special type of Service task.

Subprocess

A *subprocess* is a compound activity, meaning an activity with subparts that can be described as a child-level process. A subprocess can be represented in multiple ways in the diagram. A *collapsed subprocess* is drawn in the parent-level diagram using a normal-size activity shape with a [+] symbol at the bottom center (Figure 4-2, top); the child-level expansion is drawn in a separate hyperlinked diagram (Figure 4-2, bottom).

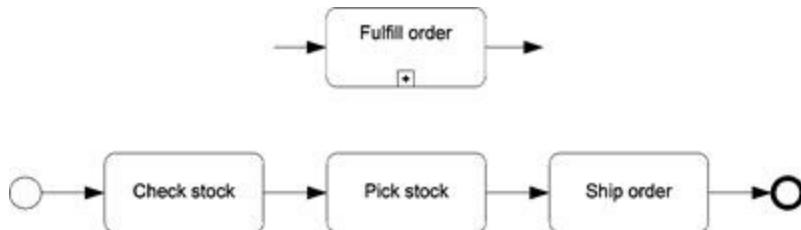


Figure 4-2. Hierarchical expansion: Collapsed subprocess in parent level (top) corresponds to child-level expansion (bottom) in separate hyperlinked diagram.

Alternatively, an *expanded subprocess* (Figure 4-3) is drawn as an enlarged activity shape in the parent-level flow that encloses the child-level expansion in the same diagram.

There is no semantic difference between Figure 4-2 and Figure 4-3. They mean exactly the same thing: When the sequence flow arrives at the collapsed subprocess in the parent level, the process immediately continues out of the start event at child level. And when it reaches the child-level end event, it resumes on the sequence flow out of the subprocess in the parent level. In fact, in the semantic model, there is no distinction at all; the XML for both is exactly the same. In BPMN 2.0, the only difference is in the graphical model (see Chapter 17).

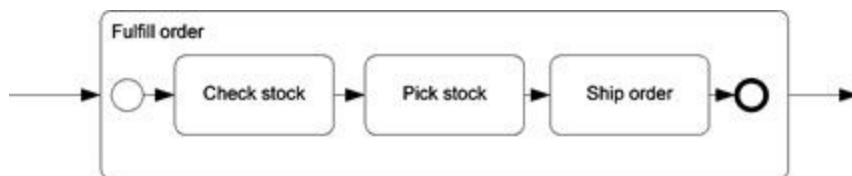


Figure 4-3. Inline expansion: Expanded subprocess shape in parent level encloses the child-level process, all on the same diagram.

An important BPMN rule to keep in mind when using inline expansion is that a sequence flow cannot cross the subprocess boundary. The incoming and outgoing sequence flows must connect to the subprocess boundary, and there should be start and end events in the child-level expansion inside the rounded rectangle. Figure 4-4 is incorrect; Figure 4-3 is correct.

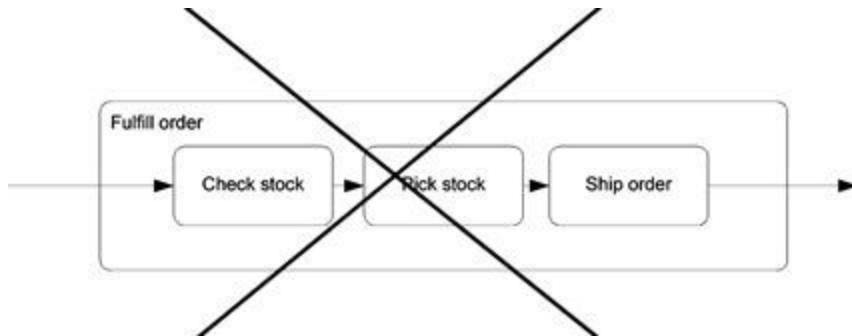


Figure 4-4. A sequence flow cannot cross subprocess boundary.

A subprocess start event must have a None trigger. You may not use a Message start event or Timer start event in a subprocess. That is a BPMN rule, not a style rule. The reason is that the start of the subprocess is not triggered by an event; it is *always* triggered by the same thing – arrival of the incoming sequence flow [\[12\]](#).

Parallel Box and Ad-Hoc Subprocess

With one exception, a subprocess should always have a single start event. The one exception is when the child level is composed of a set of activities with no sequence flows interconnecting them (Figure 4-5, left). This representation, which has no start or end events, is called a *parallel box*. It means that when the subprocess starts, all of its child activities are enabled to start in parallel. They can be completed in any order, but all must be complete in order for the subprocess to be complete.

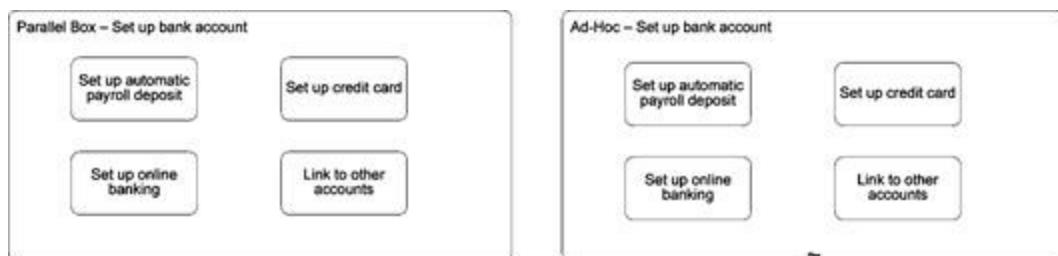


Figure 4-5. Parallel Box (left) and Ad-Hoc Subprocess (right)

A variant of the parallel box is the *ad-hoc subprocess*, denoted by a tilde marker at the bottom (Figure 4-5, right). It is essentially the same, except that not all of the child activities must be performed in order to complete the ad-hoc subprocess. It is complete when the performer declares it to be complete.

Both parallel box and ad-hoc subprocess are legacies of BPMN 1.0 and not particularly useful. In BPMN 2.0, ad-hoc subprocess is not included in either the Descriptive (Level 1) or Analytic (Level 2) subclass.

The Value of Subprocesses

Subprocesses are a valuable feature of BPMN and one of the least appreciated. Their value has several dimensions:

1. Visualize end-to-end process

BPM as a management discipline emphasizes managing and monitoring the business from the perspective of “end-to-end” processes, meaning customer-facing flows that cut across traditional organizational and system boundaries. To do that you need to understand the end-to-end process as a *single thing*, not multiple things. The ability to visualize the end-to-end process on a single page greatly aids that understanding, and collapsed subprocesses enable that. The details of each subprocess are visualized on separate hyperlinked diagrams, but they are all part of a single semantic model. From the end-to-end view you can zoom in to see as much or as little detail as you want, without the need to create and maintain multiple process models.

Visualizing the end-to-end process on a single page assumes the hierarchical style, in which the parent and child process levels are rendered in separate diagrams. The end-to-end view is just the *top-level diagram* in the hierarchy. It reveals at a glance not only the major steps of the process but the meaning of the process instance, the process’s possible end states, and its interactions with the customer, service providers, and other internal processes.

Hierarchical modeling is not required by the BPMN specification. In fact, for many traditional BPM practitioners, used to working with stickies on the wall to capture process flows from the bottom up, it might be a significant change. But flat models stretching over twenty feet of wall space make it difficult to appreciate the end-to-end process as a single thing. The traditional solution to this problem is to create separate high-level and detailed models, but this requires keeping those models in sync as the process changes. Hierarchical modeling in a BPMN tool does not have this problem, because a single semantic model contains both high-level and detailed graphical views. For this reason, the Method and Style approach relies on hierarchical modeling.

2. Enable top-down modeling

Subprocesses are also valuable to “top-down” process modeling. That means starting with the top-level diagram, in which collapsed subprocesses represent the major steps of the process, and then adding the details of each step in child-level diagrams. A collapsed subprocess with no child-level expansion can serve as a placeholder for unknown details while maintaining the integrity of a valid end-to-end model.

3. Clarify governance boundaries

Subprocesses facilitate distributed process ownership and governance. End-to-end processes frequently cross governance boundaries within the enterprise. Different parts of the process may be controlled by different executives who jealously guard their turf. Problems can arise when the boundaries between process activities are fuzzy.

Subprocesses provide unambiguous demarcation of those governance boundaries. If the top-level diagram accurately describes the interaction between independently governed subprocesses, then each subprocess can be modeled and maintained independently. Distributed process governance is aided by a model repository with authorization and versioning features.

4. Scope event handling

Event-triggered exceptions are important in real-world processes, and subprocesses are useful for defining the boundaries of a single exception handler. An event attached to a subprocess defines an *event handler* that is initiated if the event trigger occurs *at any step within the subprocess*. If the same trigger – say an order cancellation message from the customer – is handled differently in different parts of the process, each of those parts may be enclosed in a subprocess with an attached event representing its *****ebook converter DEMO Watermarks*****

distinct event handler. We will see examples of this in Chapter 7.

*****ebook converter DEMO Watermarks*****

Call Activity

BPMN 2.0 distinguishes a *subprocess*, in BPMN 1.2 called *embedded subprocess*, from a *call activity*, formerly called *Reusable subprocess*. This distinction has to do with whether the subprocess detail – the child-level expansion – is defined within the parent-level process or independently. If you have some subprocess that is used in more than one process, it is best to define it independently – in its own file – and then *call* it from each process that uses it, rather than replicate and embed the definition within each calling process.

In the diagram, *call activity* has a thick border, while *subprocess* has a thin border (Figure 4-6).



Figure 4-6. Subprocess and Call Activity

For example, suppose you sell both widgets and widget maintenance. The *Widget Order* process is different from the *Maintenance Order* process, but they share a common *Billing* subprocess. If you use a regular subprocess, you would need to replicate the *Billing* definition inside both *Widget Order* and *Maintenance Order*, and maintain that synchronization whenever *Billing* changed. A better way is to make *Billing* an *independent top-level process* defined in a separate file, and invoke it from *call activities* in *Widget Order* and *Maintenance Order*. The call activity points to a *process* element in the called model, in this case *Billing*. With subprocess, the calling and called processes are defined in the same model; with call activity they are independent.

You can similarly use call activity to call a reusable *task*, in BPMN 2.0 called a *global task*. This call activity looks like a regular task, except for the thick border. For Level 1 and Level 2 (i.e., non-executable) modeling, global tasks add little value, since the only “detail” included in the task definition is its task type and name. But in executable BPMN a User task definition, for example, would include task data, its user interface, and similar details. To reuse that task definition in multiple places (in the same process or across processes), you would define it as a global User task, with multiple call activities pointing to it.

Gateway

A *gateway*, the diamond shape, “controls” process flow, splitting it into alternative paths. Without a gateway, when a BPMN activity has more than one outgoing sequence flow, the process splits into multiple *parallel* paths. Giving them alternative labels may have worked in flowcharting, but it doesn’t work in BPMN (Figure 4-7). If you mean the process should take one path or the other, you need a gateway (Figure 4-8).

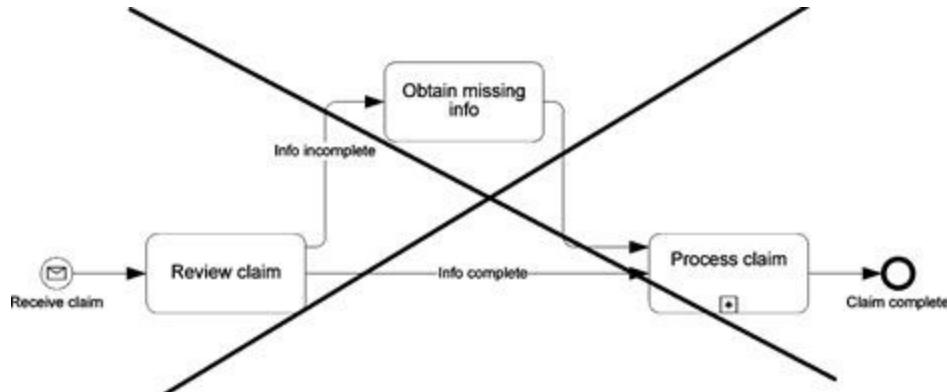


Figure 4-7. Incorrect: Alternative paths require a gateway in BPMN

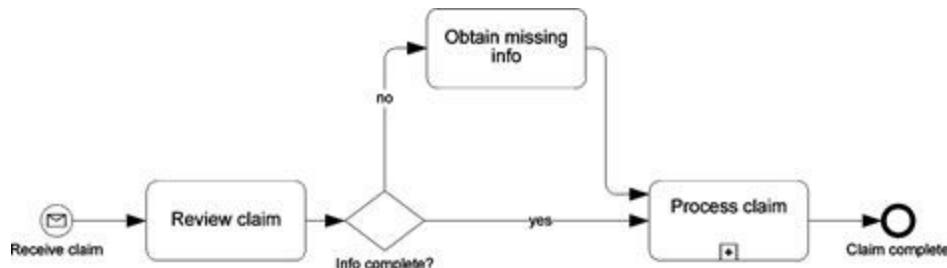


Figure 4-8. Correct: Alternative paths require a gateway in BPMN

Exclusive Gateway

BPMN defines several types of gateways, distinguished by the symbol inside the diamond, but the one shown here, with no symbol inside, is the most common. Officially named the *exclusive data-based gateway*, it is more commonly known as the *XOR gateway*. “Exclusive” means only one of its outgoing sequence flows, or *gates*, is enabled in any instance. “Data-based” means the enabled gate is determined by evaluating an expression of process data. Level 1 or Level 2 BPMN does not include formal data expressions, so the gateway conditions are expressed in the diagram by the *labels* of the gateway and gates.

When a gateway has two gates, I like to label the gateway as a *question* and label the gates *yes* and *no*. As we saw in Chapter 3, matching such a gateway label to an end state of the preceding activity helps you trace the logic from parent to child process levels in a hierarchical model.

There are two alternative ways to draw the XOR gateway. One has no symbol inside the diamond; the other has an X inside (Figure 4-9). There is no difference in meaning between the two, but the spec

asks that you pick one way and use it consistently. I favor the one with no symbol inside.



Figure 4-9. Exclusive (XOR) gateway, shown in alternative representations

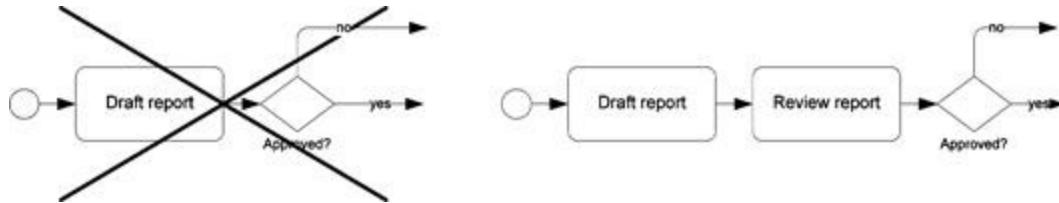


Figure 4-10. A gateway cannot *make* a decision; it only *tests* a data condition.

An important difference between a BPMN gateway and the similar-shaped “decision box” in flowcharting is that *a gateway does not “make” a decision; it just tests a data condition*. A gateway cannot approve or reject, for example. You need a task to do that. Then a gateway following the task can test the decision task end state and route the subsequent flow based on the result. The left diagram in Figure 4-10 is incorrect in BPMN; the one on the right is correct.

Parallel Gateway

A *parallel gateway* (Figure 4-11), also called an *AND-gateway*, with one sequence flow in and multiple sequence flows out, signifies a *parallel split* or *AND-split*. It means that *all* of the outgoing sequence flows are to be followed in parallel, unconditionally. It is distinguished from the exclusive gateway by the + symbol inside the diamond.



Figure 4-11. Parallel gateway

Each outgoing path thus represents a concurrent thread of process activity, meaning they overlap in time. Parallel paths may either be joined downstream or they may lead to separate end events. In the latter case, each parallel path must reach an end event in order for the process level to be complete. Following an activity or start event, multiple outgoing sequence flows means parallel split, so an AND-gateway is unnecessary in that case. The two diagrams in Figure 4-12 have identical semantics.

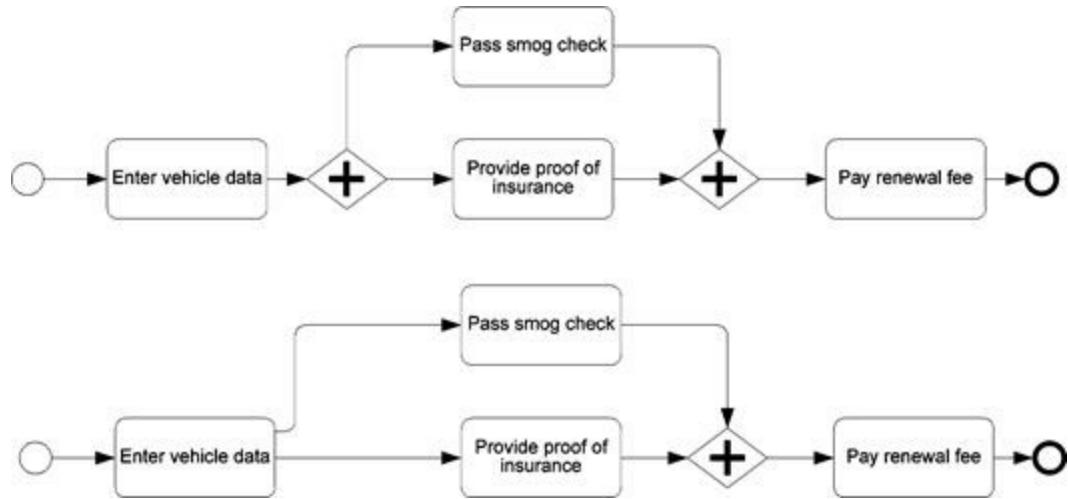


Figure 4-12. Parallel split gateway is technically redundant; both diagrams mean the same thing

In Figure 4-12, the parallel gateway drawn with multiple sequence flows in and one out is called a *parallel join* or *AND-join*. It is a type of *synchronizing join* because it requires *all* of its incoming flows to arrive before enabling the outgoing flow. An AND-gateway may ONLY be used to join paths that are unconditionally parallel. Typically this occurs only when the paths were originally the result of a parallel split, either using the parallel split gateway or multiple sequence flows out of an activity.

Unlike the AND-gateway split, the AND-join may *not* be omitted. Directly merging parallel sequence flows into an activity, without joining them first, triggers the activity (and everything downstream from it) multiple times. That is usually not what you mean. Since a sequence flow label signifies a condition and AND-gateways are unconditional, you should not label an AND-gateway or its gates.

Start Event

A *start event* is always represented as a circle with a single thin border. Its purpose is to indicate where and how a process or subprocess starts. Normally a process or subprocess has only one start event. We saw how a parallel box or ad-hoc subprocess may have no start events, and we will see in this section how a top-level process (not a subprocess) may have more than one.

In a top-level process, the icon inside the circle, called the *trigger*, identifies the type of signal that instantiates the process. Just as important, the trigger identifies the *meaning* of the process instance as the handling of that single triggering event. A subprocess MUST have a *None* trigger, no icon inside, because a subprocess is not initiated by an event but by an incoming sequence flow.

BPMN 2.0 defines seven start event triggers, but the Level 1 palette includes only four of them (Figure 4-13).



Figure 4-13. Level 1 Start events

None Start Event

A *None start event* has no trigger. In a top-level process, it either means the process trigger is unspecified or signifies manual start by a task performer, as discussed in the previous chapter. Usually None start events are unlabeled.

A subprocess MUST have a None start event; it is a spec violation to have a triggered start in a subprocess.

Message Start Event

A *Message start event*, discussed in the previous chapter, means that the process is triggered upon receipt of a message, a signal from outside the process. It signifies a process that starts upon external request, and the process instance represents the handling of that single request.

In order to maximize diagram clarity, a Message start event should be labeled *Receive X*, where X is the name of the message. Also, when using Message events you should get in the habit of drawing the message flow and labeling it with the name of the message. These are style rules, not rules of the BPMN specification.

Timer Start Event

The *Timer start event*, with a clock icon, signifies a scheduled process, usually a recurring schedule. The start event should be labeled to indicate the schedule, such as *Monthly* or *Fridays 4pm*.

Like a Message start event, a Timer start event also reveals the meaning of the process instance. Each instance represents exactly one of those scheduled starts. For example, Figure 4-14 shows a *****ebook converter DEMO Watermarks*****

monthly sales reporting process. If some activity, say *Review loss reports*, cannot be completed by the monthly sales report deadline, you cannot simply loop back in this diagram to mean include in next month's report. Next month's report is a separate instance of this process. Every activity in the process pertains only to this month's report.

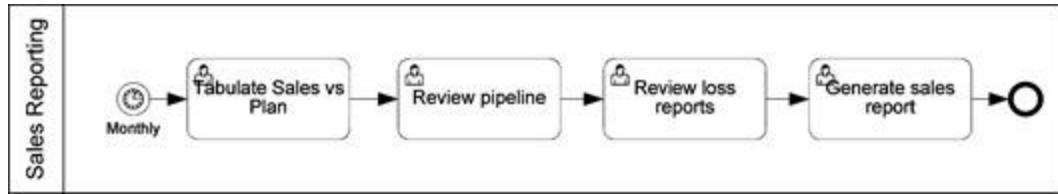


Figure 4-14. Scheduled process

Multiple and Multiple-Parallel Start Event

The *Multiple start event* (Figure 4-15, left) has a distinct shape – a pentagon – but does not represent a distinct BPMN element in the semantic model. It means that the process could be initiated by *any one of multiple triggers*, say either Message A or Message B, or possibly either by regular schedule (Timer) or on special request (Message). The start event label should indicate all of the possible trigger conditions.

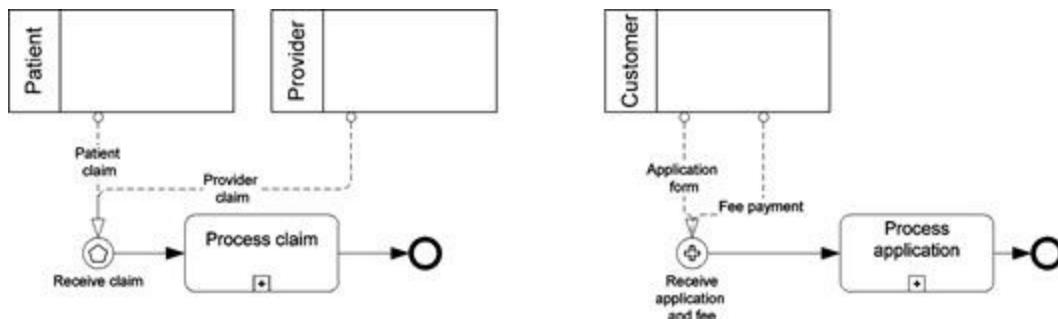


Figure 4-15. Multiple and Multiple-Parallel start events

The *Multiple-Parallel start event* (Figure 4-15, right) was added in the Finalization phase of BPMN 2.0. It is very rarely used, and it is not part of either the Level 1 or Level 2 palette. Like the Multiple start event, it is a distinct shape but not a distinct semantic element. Where the Multiple event means *any* of its multiple triggers will start the process, Multiple-Parallel means the process requires *all* of the triggers to occur before instantiation. They can occur in any order.

Alternative Start Events

The path out of a Multiple start event is the same regardless of which trigger signal is received. But what about the case where the initial process activity depends on which trigger occurs? For that you don't use a Multiple start event; you just use *more than one simple start event*, typically Message.

You can only do this in a top-level diagram. Each start event represents an *alternative* trigger for the process. Once triggered, the process or subprocess instance will *ignore a signal subsequently received by any other start event*. Such a signal would initiate a *new process instance*.

A common use case for this is *channel-dependent start*. For example, a process triggered by customer *****ebook converter DEMO Watermarks*****

request may require different initial step if the request arrives via the call center versus web or fax, but has the same backend processing regardless of the contact channel. The best way to model this is with multiple Message start events, each representing an alternative start point for the process (Figure 4-16). Remember that this is *not* the same as a Multiple start event. You would use Multiple start if any of the triggers initiates the *same* path. You would use more than one start event if each trigger initiates a *different* path.

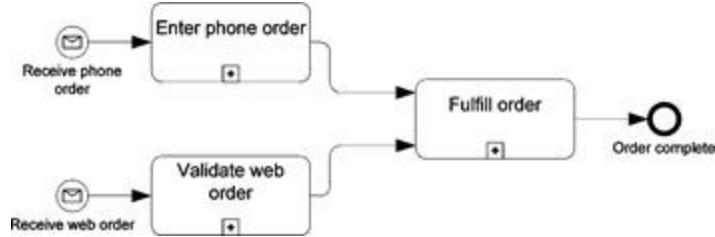


Figure 4-16. Channel-dependent start

End Event

An *end event* is always represented as a circle with a single thick border. It indicates the end of a path in a process or subprocess. An end event in either a process or subprocess may be drawn with a black or “filled” icon inside, indicating the *result* signal thrown when the event is reached. Unlike start events, it is commonplace to see more than one end event in a process or subprocess. In fact, the Method and Style approach requires a separate end event for each distinct end state in a process level.

BPMN 2.0 defines nine end event types, distinguished by their result, but the Level 1 palette includes only three of them, plus Multiple.



Figure 4-17. Level 1 end events

None End Event

A *None* end event (no icon inside) signifies that no result signal is thrown when the end event is reached. In a process level with parallel flow, it is technically allowed to end the parallel paths in separate end events, but they do not represent distinct end states. For that reason, if they are all *None* end events, it is best to merge the paths into a single *None* end event. You don’t need a gateway to join parallel paths at a *None* end event; in fact, you should not use one. Since the process level is not complete until all parallel paths have reached an end event, a join is always implied at a *None* end event.

Message End Event

A *Message* end event (black envelope icon) signifies that a message is sent upon reaching the end event. Best practice is to draw a message flow from the event to the external pool. A common use case is return of a final status response to the Customer. If you merge parallel paths directly into a *Message* end event, the message is triggered multiple times, so use a join gateway if you mean to send the message once.

Terminate End Event

A *Terminate* end event (bulls-eye icon) is a special case. Reaching *Terminate* in a process or subprocess immediately ends that process or subprocess, even if other parallel paths are still running. Reaching *Terminate* in a subprocess only ends that subprocess, not the parent-level process. Some modelers use *Terminate* simply to indicate an exception end state. However, I recommend reserving *Terminate* for the case where its specific semantics are required, an exception in one parallel path of a process level.

Multiple End Event

A *Multiple* end event (pentagon icon) is similar to the Multiple start event, in that it has a distinct shape but does not represent a distinct semantic element. It just implies more than one ordinary result is thrown, for example two different messages.

Sequence Flow

Sequence flow, drawn in the diagram as a solid line connector, represents the sequential execution of process steps: When the node at the tail of a sequence flow completes, the node at the arrowhead is enabled to start. In an executable process, it represents an actual flow of control: When the tail node completes, the arrowhead node is automatically started by the process engine. The only elements that can connect to the tail or head of a sequence flow are activities, gateways, and events, called *flow nodes* in the BPMN 2.0 metamodel. In other words, sequence flow represents *orchestration*.



Figure 4-18. Sequence flow

All activities, gateways, and events in a process level must lie on a continuous chain of sequence flows from start event to end event. (The spec does not absolutely require this, but Method and Style, with a few exceptions like the parallel box, does require it.) The chain of sequence flows is confined within a process level, so *a sequence flow may not cross a subprocess or pool boundary*. This is a fundamental rule of BPMN. Also, both ends of a sequence flow must be connected to a flow node. If one end is left unconnected, the model will not be valid.

Message Flow

Message flow, drawn in the diagram as a dashed line connector, represents communication between the process and an external entity. A message flow can connect to any type of activity, a Message (or Multiple) event, or black-box pool. Note: You may not connect a message flow to the boundary of a process pool; you must directly connect to an activity or event inside the pool. Elements connected to the head and tail ends of a message flow may not be part of the same process (including its child levels).



Figure 4-19. Message flow

In some cases, a message flow indicates the *possibility* of message communications, not the certainty of it. For example, a User task with an outgoing message flow means the task *may* send the message, not *must* send the message. If you want to indicate the certainty of sending or receiving of a message, you should use a Message event or a Level 2 *Send* or *Receive* task.

Pool

The *pool* shape is a rectangular box (Figure 4-20). It can be either horizontal, with the label boxed off on the left, or vertical, with the label boxed off on the top. (Boxing off the label distinguishes a pool from a lane, which does not have its label boxed off.) A pool containing flow elements, called a *process pool* or *white-box pool*, should be labeled with the name of the *process*. An empty pool, called a *black-box pool*, should be labeled with the name of a *business entity* or *role* such as Customer or Seller.

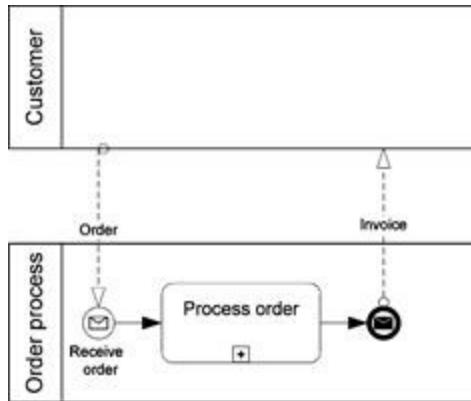


Figure 4-20. Black-box pool (top) and process pool (bottom)

In BPMN 1.2, a *pool* represented a container for a *process*. BPMN 2.0 changed the definition in a way that muddies the waters but does not fundamentally affect how pools are used in practice. Effectively, a pool is still a container for a single process, but technically it represents a *participant* in a *collaboration*. That might suggest you cannot use a pool in a diagram unless you have two or more of them exchanging message flows, and until the end of the BPMN 2.0 drafting period, that was indeed the case! But common sense prevailed at the end. A diagram may show only a single process, enclosed in a pool. In the semantic model, it is defined as a collaboration with a single participant... the BPMN equivalent, I guess, of the sound of one hand clapping.

In the XML there is no *pool* semantic element; there is only *participant*. Pool just means a shape in the graphical model that points to a *participant* in the semantic model. But since a participant can reference just one BPMN *process*, not more than one, it is effectively equivalent to a process. A black-box pool is a participant that has no process reference.

Although I advocate labeling a process pool with the name of the process, it is not uncommon to see BPMN diagrams in which process pools are labeled with the name of an organization, such as a company or department. I disagree with this practice for several reasons:

1. There is no other BPMN element in the diagram where the process name appears.
2. A collaboration diagram could contain two internal processes, interacting via message flows. Such a collaboration requires two *participants*, each referencing a different process, although the members of both participants may be exactly the same people. We'll see an explicit example of this in Chapter 8.
3. Labeling a process pool with the name of an organization, such as a department, encourages splitting a single process into multiple independent processes. There are occasions where

modeling an end-to-end business process as multiple BPMN processes is appropriate, but most of the time it is best to model departments or other organizational units as *lanes* within a single process, not separate pools.

If a diagram depicts only a single process with no message flows, it is not required to draw a pool at all. However, in any diagram that shows a collaboration between multiple processes, at most *one* of them may omit the pool shape. In BPMN 1.2, that pool was considered “invisible”; in BPMN 2.0 the pool does not exist in the model.

In hierarchical modeling, where a child-level expansion is drawn on a separate hyperlinked diagram, it is best to omit a pool shape enclosing the child process level. (Some *tools* automatically draw a pool in the child level if you want to show lanes, but this a tool issue not a BPMN requirement.) If you enclose the child-level expansion in a pool, its label should match that of the *top-level process*; it should *not* be labeled with the name of the subprocess. In the tool I use for my BPMN training, even if you give the same names to the parent and child-level pools, two separate participants will be created in the XML unless you tell the tool they represent the same entity. It’s easy to do, and it makes the XML come out right... but it’s also easy to forget. We’ll come back to this in the BPMN Implementer’s Guide section of this book.

Lane

In BPMN 2.0, a *lane* (Figure 4-21) is an optional subdivision of a process level. Like pool, a lane is drawn as a rectangular box, but its label – at the left for a horizontal lane or at the top for a vertical lane – is not boxed off. BPMN allows you to draw lanes without enclosing them inside a pool (although some tools do not).

Lanes are a holdover from traditional swimlane flowcharts, where they were used to associate process activities with particular actors – departments or roles. They are still typically used for that purpose, but BPMN 2.0 actually allows them to be used for *any* type of categorization, for example value-adding vs. non-value-adding activities. You can even have multiple sets of categorizations, called *lanesets*, in the semantic model, all associated with the same process level. One laneset might indicate the performer *role*, for example, and an alternative one might indicate the responsible *department*. A particular diagram in the graphical model may reference just one of the lanesets.



Figure 4-21. Lanes

BPMN 1.2 was vague about how lanes in a child-level diagram relate to lanes in the parent level, but it is clearer in BPMN 2.0. Each laneset definition applies to a specific process level. If you want to reference the same lanes in parent and child-level diagrams, you need to replicate the laneset at both levels of the model.

A lane in a process level may contain a *child laneset*. The child lanes, also called *sublanes*, are drawn nested inside its parent lane. For example, a parent lane might represent a department, and sublanes roles within that department.

If you use lanes in a process level, all its flow nodes must be associated with one lane or another. You may not depict some in a lane and others not in a lane. BPMN has no rules about sequence flows crossing lane boundaries.

Data Object and Data Store

One of the biggest changes from BPMN 1.2 to BPMN 2.0 concerns the modeling of data and data flow. In BPMN 1.2, data objects were considered *artifacts*, diagram annotations with no semantics or rules. In BPMN 2.0, *data object* was upgraded to a first-class semantic element, along with a new element, *data store*. Even though both data object and data store are part of the Level 1 palette, the new definitions treat data from the perspective of a developer doing executable process design.

The *data object* shape looks like a dog-eared page (Figure 4-22, left). Besides the *name* of the data object, the label may indicate its *state* by enclosing it in square brackets. The *data store* shape (Figure 4-22, right) is a cylinder, similar to the symbol for a database or storage device.



Figure 4-22. Data object and data store

A *data object* is really a programming construct. It represents a *local variable* in a process level, a piece of temporary data stored inside the process instance while it is running. Its value is visible to other elements in the same process level or one of its children – for example, it can be passed to the input of a process activity or be tested by a gateway condition – but is invisible to a sibling or parent-level element. And when the process level (top-level process or subprocess) ends, the data object goes away. In other words, it works like a variable in a computer program, not what a modeler normally means by “data”.

A *data store* represents *persistent data*, such as information stored in a database or business system. It can be queried or updated both by the process and by entities outside the process. It does not disappear when a process level, or the process as a whole, ends. It is more often what a process modeler means by “data”.

Data object and data store connect to other model elements through *data associations*, dotted line connectors that look a bit like message flows except the lines are dots not dashes and the arrowhead is a V, not a triangle. With data objects, one end of the data association is connected to an activity or event, and the other to the data object. In that case, the data association represents a *mapping* between that variable and a data input or output of the activity or event. The mapping may be a simple copy or a transformation, but only the data association connector and label are visible in the diagram.

Data flow within a process is thus represented by a *data output association* from an activity or event to a *data object*, followed by a *data input association* from the *data object* to another activity or event (Figure 4-23, right). You are allowed to use a non-directional data association (drawn without the arrowhead) between the data object and a sequence flow connecting the source and target objects as a “visual shortcut” (Figure 4-23, left). In other words, the semantics are those of the right diagram, even if you draw it as on the left. (Not all BPMN tools will do this for you; best to model it as in the right-hand diagram.)

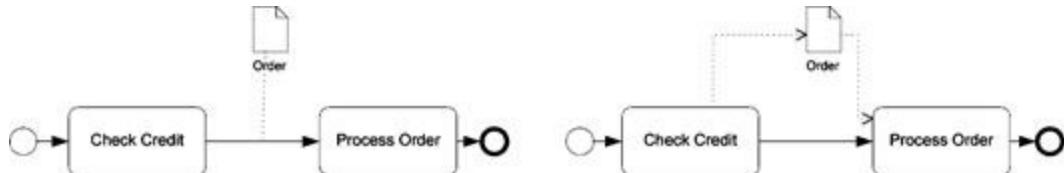


Figure 4-23. The left diagram is considered a “visual shortcut” for the data flow in the right one.

A data store represents a single unit of information stored in a system, such as a database record, not the system or database as a whole. Data association directed into the data store represents an *update* operation, while data association directed out of the data store represents a *query*. In Figure 4-24, the task *Process Order* updates the account balance in the data store *Customer Account*.

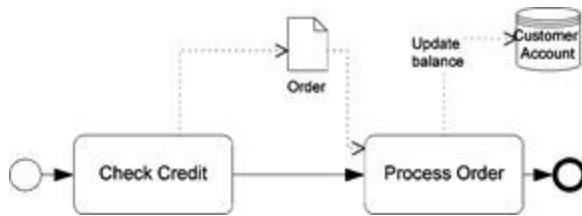


Figure 4-24. Data store represents persistent data accessible to the process.

The BPMN metamodel imposes one more bit of complexity here. The data store itself is a root element in the semantic model; it is defined outside of any particular process. The element drawn in the diagram is actually a *data store reference*, which is contained within a process level (otherwise you could not connect a data association to it). If you interact with the same data store from two different parts of your model you may need to draw separate data store references. But the BPMN tool should hide this complexity from the modeler.

Documentation, Text Annotation, and Group

The BPMN model as a whole and most of its individual elements each contain a *documentation* element in the XML, into which you can stuff as much information as you please, either directly or via links to external documents. These documentation elements are part of the Descriptive subclass (i.e., Level 1), meaning any tool that claims conformance is expected to be able to import and display them. However, *documentation* has no associated graphical element. In other words, it doesn't show up in the diagram.

If you want to put an annotation in the diagram itself, use *text annotation*, indicated in the diagram by a square bracket shape framing a bit of user-entered text (Figure 4-25). Text annotations are not supposed to be free-floating but attached to some graphical element via a non-directional *association*. Association looks the same as a data association without the arrowhead. Text annotation and association are *artifacts*, meaning supporting information that does not affect process flow.

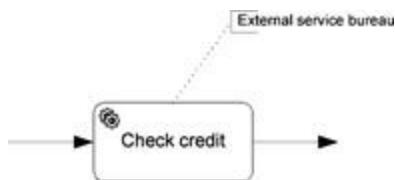


Figure 4-25. Text annotation and association

Finally, there is *Group*, drawn as a rounded rectangle with a dot-dash border (Figure 4-26). Group is also an artifact. Essentially it is just a box drawn around a set of elements in the diagram to indicate some relationship between them. Officially, the spec says this: "The grouping is tied to the CategoryValue supporting element. That is, a Group is a visual depiction of a single CategoryValue. The graphical elements within the Group will be assigned the CategoryValue of the Group." However, I have *never* seen this CategoryValue mechanism used in practice. If you are inclined to use it at all, you should consider Group just a visual highlighter in the diagram.

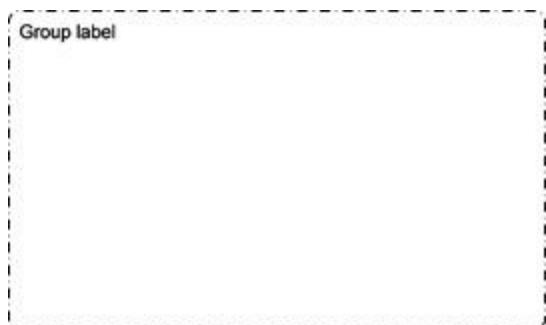


Figure 4-26. Group

Chapter 5

The Method

We've now covered the full Level 1 working set. There is a lot more left to discover at Level 2, but we know enough already to handle the majority of process modeling requirements. So we are now ready to discuss the Method.

The Method is not part of the BPMN specification. OMG proudly declares that BPMN has no official methodology, since it is intended for a wide variety of uses by people with divergent interests and skills. But in my experience, most people using BPMN are trying to do the same thing – *create a non-executable process diagram that conveys the process logic in a meaningful way*. Whether they are simply trying to document an as-is process or create business requirements for an automated to-be solution, the attributes of a “good BPMN” model are pretty much the same. The Method is an attempt to standardize the structure of such a model in order to maximize shared understanding of the diagram. If everyone in the organization structures their process models according to the same principles, they are more likely to understand models created by others.

Goals of the Method

The Method is a recipe for going from a blank page to a complete BPMN model in a consistent, well-structured way. It is based on a *hierarchical modeling style* that reveals important basic facts about the process as a whole from the top-level diagram, and lets you add as much detail as you want in child-level diagrams. It leverages *label-matching* as an aid to tracing the process logic from the top level diagram down to the lowest level of detail, even if the hyperlinking of the “live” model in a tool is not available. The Method is prescriptive, and it will help get you started on the right foot. However, following my Method to the letter is less important than establishing a prescriptive methodology of your own and deploying it consistently across your organization.

Let’s review once more the overarching principles of “good BPMN”. They include:

- **Completeness.** The essential elements of the end-to-end process logic should be captured in the diagram, including how the process starts, its distinct end states, what the instance represents, and its interaction with external entities such as the requester, service providers, and other internal processes.
- **Clarity.** Details of the process flow – which activities are conditional, which are performed in parallel with others, how various exceptions are handled – should be unambiguous from the diagram alone, even to those unfamiliar with your process or even your terminology. That means using label-matching to make the logic traceable from the top level down in a hierarchical model, even when working from paper copies.
- **Shareability between business and IT.** BPMN as a *language* can be shared by business users, business analysts, and developers. But we’re aiming higher. We want to create individual *BPMN models* that can be shared between business and IT. That’s not easy. It demands that business users and business analysts apply more rigor and attention to detail than they may be used to, and it demands that developers describe process activities in terms of the business functions they perform rather than their specific implementation.
- **Structural consistency.** Given the same set of facts about how the process works, all modelers ideally should create more or less the same process model, at least the same overall structure. If you can achieve that kind of consistency across your organization, it greatly enhances the ability to understand models created by others.

These principles of “good BPMN” are the goals of the Method.

Hierarchical Top-Down Modeling

The Method describes a hierarchical top-down modeling style. But what does that mean, and why do I recommend it?

Hierarchical means graphically representing the end-to-end model as a set of linked process diagrams representing distinct *process levels*. That is, a collapsed subprocess in the parent-level diagram is expanded in a separate child-level diagram. Collapsed subprocesses in that child level may be further expanded in yet another diagram with a “grandchild” relationship to the first. A single top-level diagram stands atop the hierarchy, and the number of levels nested below it is unlimited.

In contrast, a *flat* process model places all steps of the process, even the finest details, in a single diagram. If subprocesses are used at all – and sometimes they are not – they are shown expanded “inline,” as described in Chapter 3. A flat end-to-end process model rarely fits on a single printed page, unless output by a large format plotter. Depending on the tool, it may be possible to print it in several Letter/A4-size pieces and tape up the mosaic on a wall. There is also a BPMN element, called a *Link event pair*, that can be used to split a single process level in the semantic model across multiple diagrams. These diagrams are *siblings*; they do not have a parent-child relationship.

Hierarchical modeling means the semantic model is represented visually by multiple diagrams. A *diagram* in BPMN is equivalent to a *logical page* (even if more than one page of paper is required to print it). The diagrams are not separate models, just separate *views* of a single semantic model.

A *single semantic model* means a single process definition describes the entire process end-to-end. In fact, the semantic model by itself does not distinguish between hierarchical and flat representations; the XML is identical. The difference between them is in the *graphical model*, the diagram layout information. We’ll talk more about the graphical model in Chapter 17.

Top-down means beginning by understanding the end-to-end process as a whole, enumerating its major steps in a *high-level map*, and then arranging those steps in a *top-level process diagram* that fits on a single page. From there you proceed to drill down to define the internal logic of each high-level map activity in a *child-level diagram*, revealing only as much detail as required for your purpose. Top-down forces the modeler to start with the big picture, adding only the details needed for the immediate purpose. This stands in contrast to the traditional approach in which the process definition is gradually built up from the bottom based on SME interviews: *First we do this, and then they do that, and then....* That approach can get bogged down in unnecessary detail, and may even lead to wasted effort modeling details that are not even part of the process.

Top-down usually implies a hierarchical modeling style, whereas bottom-up often leads to flat models. I favor BPMN tools, like Process Modeler for Visio or native Visio Premium 2010, that naturally support the hierarchical top-down approach by automatically creating a linked child-level diagram from a collapsed subprocess.

End State

A key concept in the Method is the notion of *end state*. You can search the BPMN 2.0 specification from cover to cover and never see that phrase once. Actually it's not a BPMN term, but a common sense business term. Recall that an activity in BPMN is an action performed repeatedly in the conduct of business. Each instance of the activity has a well-defined start and end. When each instance of the activity is complete, you could ask the question, *how did it end?* Did it complete *successfully* or in some *exception* condition?

Perhaps the activity has more than one exception end state, or possibly more than one success end state. How many are there? That's up to you. How many do you want to distinguish? If the flow branches following the activity, usually the activity end state determines which path is taken. If there are three different possible next steps in the process, then you need to distinguish three end states. A gateway following the activity then tests the end state. Did the activity end in state A, B, or C? If A, go here next; if B, go there next; if C, go to that one next. If the subsequent flow is the same no matter how the activity ends, then you only need one end state.

If the activity is a *task*, its end states are invisible in the model. However, they may be *implied* by the labels on the gateway following the task. For example, a gateway *Credit OK?* implies the end states *Credit OK* and *Credit Not OK*. However, if the activity is a *subprocess*, you can make its end states *visible* in the diagram by defining a *separate end event for each distinct end state*, and labeling each one with the name of the end state. This technique is not required by the BPMN spec, but it is central to the Method because it makes the process logic traceable from the top level down.

If the subprocess has two end states, I recommend labeling the gateway as a question, and labeling its gates *yes* and *no*. The gateway label (minus the question mark) should match the label of one of the subprocess end states. That means that instances following the *yes* path out of the gateway in the parent-level diagram are the same ones that reach the end event with the matching label. Instances following the *no* path are those that reach the other end state. What if there are three end states? In that case, I recommend matching the label of each *gate* of the gateway immediately following the subprocess with the label of one of the end events.

Multiple end states do not always imply exceptions. They could just signify some aspect of the instance that affects the subsequent flow. For example (Figure 5-1), you might have an activity *Determine customer type* that identifies a buyer as either a premier customer or a regular customer, followed by a gateway labeled *Premier customer?* with *yes* and *no* gates leading to separate fulfillment activities. If *Determine customer type* is a subprocess, it should have two end events, one of which is labeled *Premier customer*. Any process instance reaching the *Premier customer* end state will, by this convention, always follow the *yes* path out of the gateway, and any instance reaching the other end state will follow the *no* path.

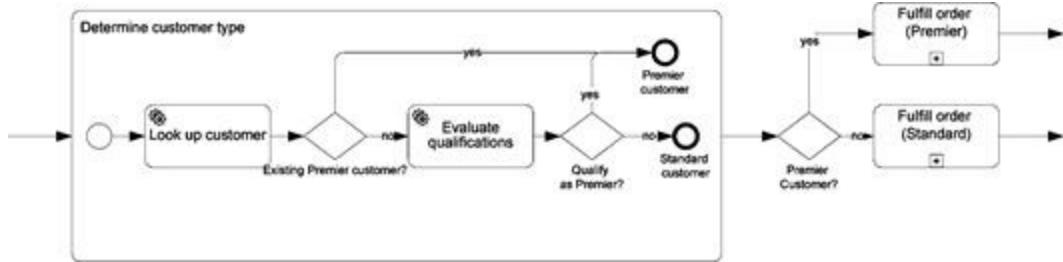


Figure 5-1. Distinguishing subprocess end states as separate end events aids top-down traceability.

All this adding of end events and attention to labeling might seem at first like needless bother. Modelers often assume that everyone that will ever look at their BPMN diagram is already familiar with the process and the terminology used in the diagram. However, that is not always the case. Matching end state and gateway labels creates a persistent visual link between parent and child-level diagrams that makes the logic traceable from the top level down even if the viewer is unfamiliar with the process or its terminology.

Step 1. Determine Process Scope

Top-down modeling starts with agreeing on the *scope* of the process, where it starts and ends. The process does not have to be customer-facing, what we sometimes call end-to-end. It could be an internal function performed entirely within a single department. The important thing is that there is agreement on the scope before modeling begins. You don't want your modeling efforts, after weeks of interviewing subject matter experts and other stakeholders, to break down in a dispute over when the process is actually complete. That is often a difficult question to answer, but you want to have those discussions *before* diving into the process details.

In this first step of the Method, I am using the term "process" a bit loosely, since it possibly could require more than one BPMN process. We'll discuss that more in Chapter 8. But it always means a repeated action with a well-defined start and end, not a continuously ongoing business function. The key questions for now are these:

1. How does the process start? For example, is it on request, either from an external entity or internal task performer, or is it a regularly scheduled process?
2. What determines when it is complete? Once an instance is complete, there are no further actions on it possible within this process definition. Those would have to be part of a separate process model. For example, if your *Order* process ends upon sending the invoice, activities related to collection of payment are not part of this process.
3. What does each instance of the process represent? Normally this is related to the start event, as we have discussed. If the start event represents a request, then the instance normally represents the fulfillment of that request.
4. Are there different ways that the process could end? In other words, does the process have more than one end state?

As we have seen, an *Order* process could fail because of problems with the buyer's credit, or the ordered item is out of stock, or for various other reasons. Should we say that such a process has two end states, or more than two? There is no right or wrong answer. It comes down to how many distinct end states do you want to identify for analysis purposes or possibly monitor in actual operation? If a possible end state occurs very infrequently or is not worth distinguishing from some other end state, don't represent it in the model. BPMN is there to serve *you* in your modeling needs, revealing as much or as little detail as you require.

There is no process diagramming to be done in Step 1. You have completed it once you have general agreement among your stakeholders as to the answers to the four questions above.

Scenario: Car Dealer Order-to-Cash

To illustrate the Method, we will use a process familiar to many of you from the buyer's perspective, purchasing a new car. But here we will imagine it from the car dealer's perspective, the seller's order-to-cash process. Let's go through the four questions one by one.

When does this process start? We use this example in my BPMN training, and a student may suggest that it starts when a customer walks into the showroom. But I don't think that is correct.

*****ebook converter DEMO Watermarks*****

Certainly there are sales activities that occur when the customer walks into the showroom, but they are not part of the order process. There is no “order” when the customer first walks in. In fact, there may not even be a “process” in the BPMN sense.

Then a student typically offers that it starts with an order. I agree with that. OK, what is an order in that context? What form does it take? What information does it include? Does any money change hands?

As I write this, I am in the process of buying a new car myself, so I can tell you exactly what it meant in my case. An order is an agreement from a particular buyer to buy a particular car, or a detailed specification for a car – make, model, color, and options – for an agreed price. If that car is not in the dealer’s possession, it could be acquired by trade with another local dealer or custom-ordered from the manufacturer. In any case, the buyer’s agreement is always with the dealer, and what we are concerned with is the dealer’s order-to-cash process.

At the time of the order, a small refundable deposit might be required in order to reserve the car or to secure the purchase from another dealer or the factory. However, the process is not complete until the full amount of the purchase is paid and the buyer receives the car. For a new car, that usually occurs days or weeks later.

An instance of this process is a single order. But what if the buyer purchase two cars? Is that one instance or two? It depends. If this is considered a single financial transaction – there is a single closing with payment and delivery of both cars together – then it is one instance. If the two cars are treated as separate financial transactions with possibly different closing dates, then it is two instances. It is best to consider and resolve such “gray areas” when scoping the process to be modeled.

Successful closing of the transaction represents the normal, successful end state of this process. Let’s call it *Transaction complete*. But are there other end states we want to consider? In this case there are. Here we only want to count those exceptions that occur with sufficient frequency that they affect the overall business. It could be that the customer is unable to secure financing. In that case the process will complete in an exception end state we call *Financing unavailable*. And there is another exception that could occur when the car must be ordered from the factory. It could turn out that the projected delivery date is later than estimated at the time of the order, and the buyer cancels the order. We’ll call that end state *Delivery date unacceptable*.

We could consider these two exceptions just technical variations of a single end state, *Transaction failed*. But our dealer in this case wants to distinguish them because they suggest problems in separate parts of the organization. *Financing unavailable* suggests a potential problem in the Finance department, since an order should be initiated only if the buyer is thought to be credit-worthy. *Delivery date unacceptable* suggests a potential problem in the Sales department, since the actual delivery date for the ordered vehicle was not estimated properly at the time of order. Identifying these as distinct end states implies we want to understand their cause and handling as individual exceptions. For example, we might want to think about specific improvement actions that could reduce their frequency of occurrence, or actions that could reduce their impact on the business when they occur.

Step 2: The High-Level Map

The next step in the Method is to define the High-Level Map. This is simply an enumeration of the major activities of the process, ideally ten or fewer in order to represent the top-level BPMN diagram, which is generated from the high-level map, on a single page. In the first edition of this book, I drew the high-level map as a linear sequence of BPMN activities, but now I think it might be better simply to think of it as a *list*. We will turn it into a top-level BPMN diagram in the next step of the Method.

Since the high-level map is just a list of the process's major activities, this step should be very simple. In practice, however, you will probably spend a while on it. The activities in the map are not a sampling of process activities, with more details to be added in between them later on. It is better to think of them as *containers* into which those details will be added.

The steps in the high-level map must be "activities" in the BPMN sense, meaning actions performed repeatedly, each with a well-defined start and end. Moreover, the *instances* of each activity in the high-level map must be aligned, having one-to-one correspondence with each other and with the process instance.

Other factors help guide the selection of high-level map activities. Remember that in BPMN, the start of one activity is usually triggered by the *completion* of a previous activity, not at reaching some point in the *middle* of the activity. Also, if the governance of the process is distributed across multiple parts of the organization, steps in the high-level map should ideally match up with those governance boundaries. And, of course, we would like to restrict the count of high-level map activities to ten or less. These considerations guide definition of the high-level map, but it will require considerable time and discussion with process stakeholders.

Finally, when the outcome of an activity affects the subsequent path of the process instance, we need to think about *the end states of each activity* in the high-level map. End state names should be brief but descriptive.

Scenario: Car Dealer Order-to-Cash

In our car dealer order-to-cash scenario, the Owner meets with Sales Manager, the Service Manager, and the Finance Manager to come up with the high-level map. They collectively agree on the following activities:

- *Finalize order.* There are slight differences in procedure and price depending on whether the car is available from dealer stock, acquired by trade with another dealer, or custom ordered from the factory. This activity is governed and performed by the Sales department. End states: *Reserved from stock; Dealer trade; Order from factory*.
- *Acquire car from local dealer.* This activity is conditional, performed in some fraction of process instances, not all of them. This activity is also governed and performed by the Sales department. End state: *Car received*.
- *Acquire car from factory*, also conditional and performed by Sales. If the factory delivery date is later than the one estimated at the time of *Finalize Order*, the customer could cancel the transaction. End states: *Car received; Order cancelled*.

- *Prepare car for delivery.* This activity includes dealer-installed options and cleaning up the car for delivery to the customer, regardless of whether the car comes from stock, dealer trade, or factory order. It is governed and performed by the Service department. End state: *Ready*.
- *Arrange financing*, performed by the Finance department. It can start as soon as *Finalize order* is complete, running in parallel with acquiring and preparing the car. End states: *Financing confirmed; Financing unavailable*.
- *Close and deliver.* This activity, performed by the Finance department, completes the financial transaction and delivers the car and registration materials to the customer. It may not start until both *Arrange financing* and *Prepare car for delivery* are complete. End state: *Transaction complete*.
- *Handle order cancellation.* This activity is only performed when the order is cancelled before the closing. It is performed by the Finance department. After handling the cancellation, we still want to distinguish the process-level end states. End state: *Delivery date unacceptable, Financing unavailable*.

Step 3: Top-Level Process Diagram

Now that we have our high-level map, we can turn it into a top-level BPMN diagram. The process starts on request from the customer, so we will use a Message start event, *Receive order*. Each high-level map activity becomes a subprocess in the diagram. In the hierarchical modeling style, we will later expand each of these activities in hyperlinked child-level diagrams to show the details of each step.

In the Method and Style approach, each activity that is *conditional* in the high-level map will be drawn following a gateway that tests the end state of the preceding activity. If the gateway has two outputs (gates), we label the gateway as *[endstate1]?*, where *[endstate1]* is the name of one of the end states of the preceding activity, and label the gates *yes* and *no*. If there are more than two gates, we simply label the gates themselves *[endstate1]*, *[endstate2]*, etc. We don't need a gateway to merge alternative paths; just connect the sequence flows directly into the downstream activity.

If an activity is performed *concurrently* with other activities in the high-level map, we can split the flow into parallel paths either using a parallel gateway or simply two sequence flows out of the preceding activity. If a downstream activity requires completion of two or more parallel activities, we must use a gateway join.

In this way, construction of the top-level BPMN diagram from the high-level map becomes a fairly mechanical exercise.

Scenario: Car Dealer Order-to-Cash

We'll start by considering only the "happy path" of the process, leading to the successful *Transaction complete* end state, and ignoring the exception end states of *Order car from factory* and *Arrange financing*. The result is shown in Figure 5-2. Since *Finalize order* has three end states, we don't label the subsequent gateway as a question, but instead match the name of each gate to an end state. Two sequence flows out of *Finalize order* means both paths are initiated in parallel. The AND-gateway join means both *Prepare car for delivery* and *Arrange financing* must be complete before *Close and Deliver* can start.

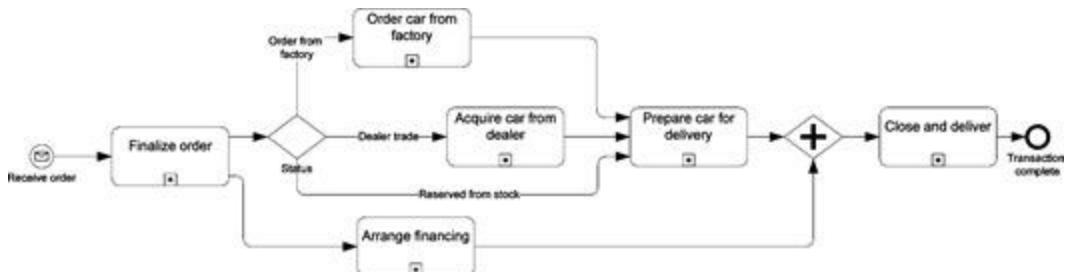


Figure 5-2. Top-level BPMN diagram, happy path

That was the happy path. Now let's add the exception paths. We want to show all the end states of the process as separate end events in the top-level diagram, each labeled with the name of the end state. The result is shown in Figure 5-3.

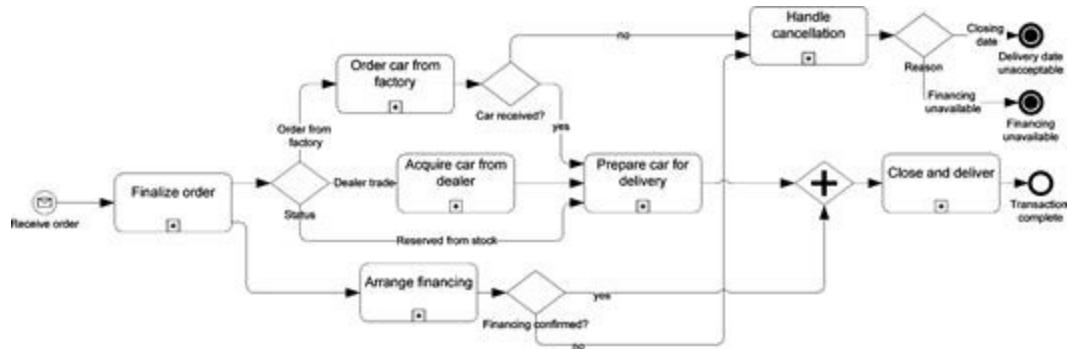


Figure 5-3. Top-level BPMN diagram, including exception paths

Recall that the exception end state *Delivery date unacceptable* is the consequence of an exception in *Order car from factory*. If that exception occurs, we need to perform *Handle cancellation* and then end the process. Because there is another process path in parallel to this one, we need to use a *Terminate* end event at *Delivery date unacceptable*. Otherwise, the financing path would continue and wait forever at the join.

The same thing applies with *Financing unavailable*. If *Arrange financing* does not end in *Financing confirmed*, we need to abort the process by performing *Handle cancellation* and then ending in a *Terminate*. Otherwise the *Prepare car for delivery* path would hang at the join.

We could have drawn lanes in the top-level diagram, but it makes the diagram slightly more convoluted and tedious to make “pretty.” You can see this from Figure 5-4, which is semantically equivalent to Figure 5-3. And we have not added any message flows yet! It is often better to omit lanes in the top-level diagram and just put them in the child-level diagrams.

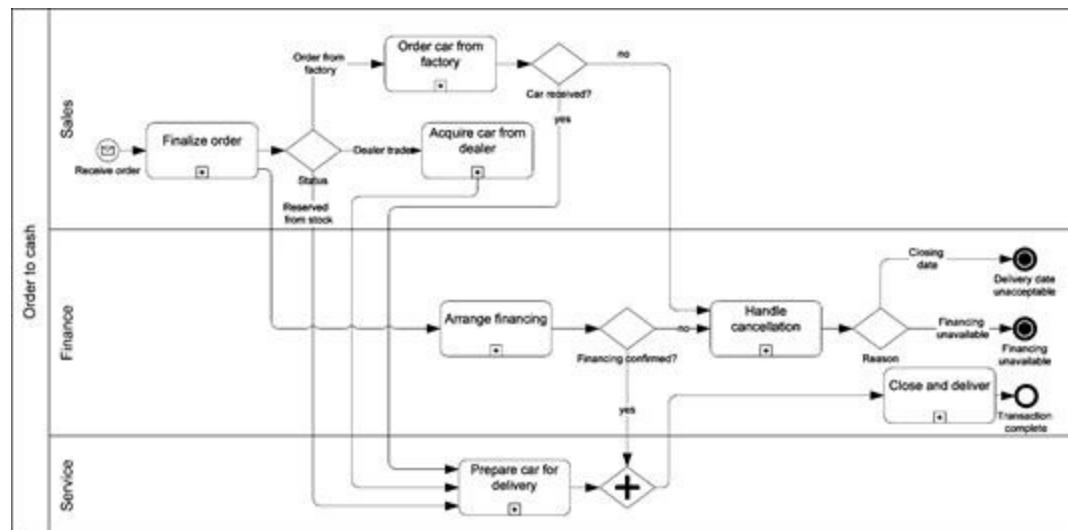


Figure 5-4. Top-level diagram showing pool and lanes

Step 4: Child-Level Expansion

The top-level diagram tells you how the process starts and ends, but it reveals little about the internal details. For that you need to show the child-level expansion of each top-level activity. In hierarchical modeling, each is drawn on a separate diagram, hyperlinked to a collapsed subprocess in the top-level diagram. Good BPMN tools create those hyperlinks automatically.

The child-level expansion must have a None start event. Activities in the child-level expansion can include collapsed subprocesses, which would then be expanded in another diagram two levels down from the top. You may either enclose the child-level process in a pool or not. If you draw the pool, it must be named the same as the pool of the parent level, i.e., the name of the process. You may either include lanes in the child-level diagram or not. Lanes are defined independently at each process level.

Remember to create a separate end event for each end state of the high-level map activity identified in Step 2, and label it with the name of the end state. If you want to modify your list from Step 2, it is OK to do that here. Just make sure that if a subprocess is followed by a gateway, the gateway (or gate) label matches the label of one of the subprocess end states.

Scenario: Car Dealer Order-to-Cash

To illustrate, we'll consider here the first activity only, *Finalize order*. We know that it starts upon receipt of an order, a document from the customer that identifies the buyer, the car (or specifications for a car), and the agreed price. And we know that it could end in one of three states: *Reserved from stock*, meaning it is available from dealer inventory; *Dealer trade*, meaning we will acquire it from a local dealer; and *Order from factory*, meaning we will custom order it from the manufacturer. The child-level expansion must have flows leading to those three end events in order to be consistent with the parent-level diagram.

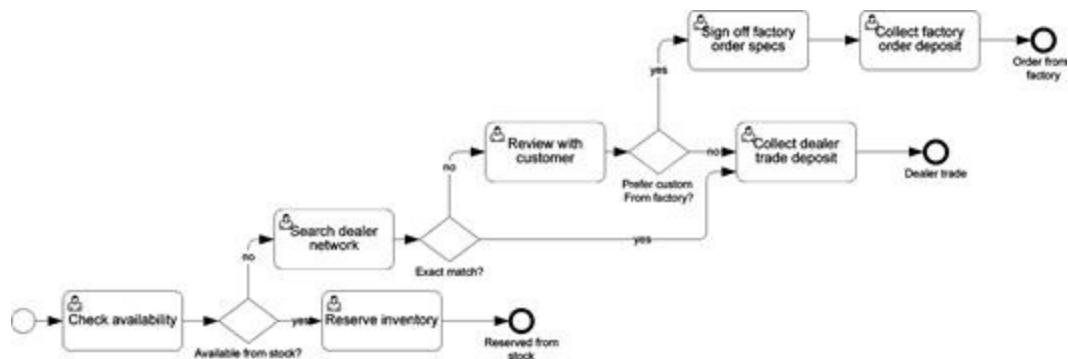


Figure 5-5. Child-level expansion, *Finalize Order*

Step 5: Add Message Flows

In Chapter 3 we added the customer request and final status message flows before we did the child-level expansion, and I still often do it that way. But here I created the child-level expansion first in order to talk about the general issue of showing “collaboration”, i.e., message flows, in your BPMN models.

Message flows are not required in BPMN. The spec says you can draw them or not, as you choose. However, Method and Style says you should draw them, because they add valuable information to the diagram. They show how your process interacts with the customer, service providers, and other internal processes. In other words, they provide valuable *business context* for the process.

The downside is that message flows also add visual clutter to your diagrams. Through years of BPMN training I have found that students with an “architectural” inclination love them, but others may find them annoying. The solution that I use in training is to include them in the model, but allow them to be hidden from users who don’t want to see them. This is easy to do in Visio and similar tools that support drawing “layers,” but it may not be possible in other tools.

Message flows always connect to either an activity or a Message event in the process. Usually the other end of the message flow is connected to the boundary of a black-box pool. Even if that pool represents another internal process, it is best to represent it here as a black-box pool. (In the model of the other process, *this* process becomes a black-box pool.) In any case, you can’t simply leave one end of the message flow dangling in space; the XML schema demands a valid connection at both ends.

In some cases, message flows indicate the *possibility* of a message rather than the *certainty* of it. In other words, a message flow out of a User task does not mean that the message *must* be sent but that it *could* be sent. And similarly, a message flow into a User task does not require the arrival of the message in order to complete the task. (In the Level 2 palette, we’ll see other task types, Send and Receive, that do require the sending and receiving, but we are not there yet.) Also, if there are multiple message flows connected to an activity, their order of occurrence is ambiguous. Sometimes it is implied by the label – for example, an *Info Request* message would generally precede *Info Response* – but in general you can’t tell.

In hierarchical models, a basic Method and Style principle is that the message flows should be consistent between parent and child levels. If a collapsed subprocess at parent level has three outgoing and two incoming message flows, then the child-level expansion should have the same number, and their labels should match the parent diagram as well. This is another example of top-down logic traceability. In the parent-level diagram, you cannot tell by inspection the order of those messages, or whether some of them are conditional. You get a much better idea from the child-level expansion, where those five messages are replicated.

Consistent application of this principle means that all the message flows in the entire process model are present in the top-level diagram, and that can be quite a few. If they don’t all fit, it’s acceptable to use visual shortcuts like combining multiple message flows into one, with appropriate labeling, but it’s best to show them all if you can.

Scenario: Car Dealer Order-to-Cash

Figure 5-6 shows the top-level diagram with message flows. You see that while they make the interactions with other entities visible, they also add visual clutter. Take a look at *Order Car from Factory*. There are four message flows here, but their order is not obvious from the top-level diagram. By replicating them in the child-level expansion, the message order, along with the rest of the process logic, becomes immediately apparent (Figure 5-7).

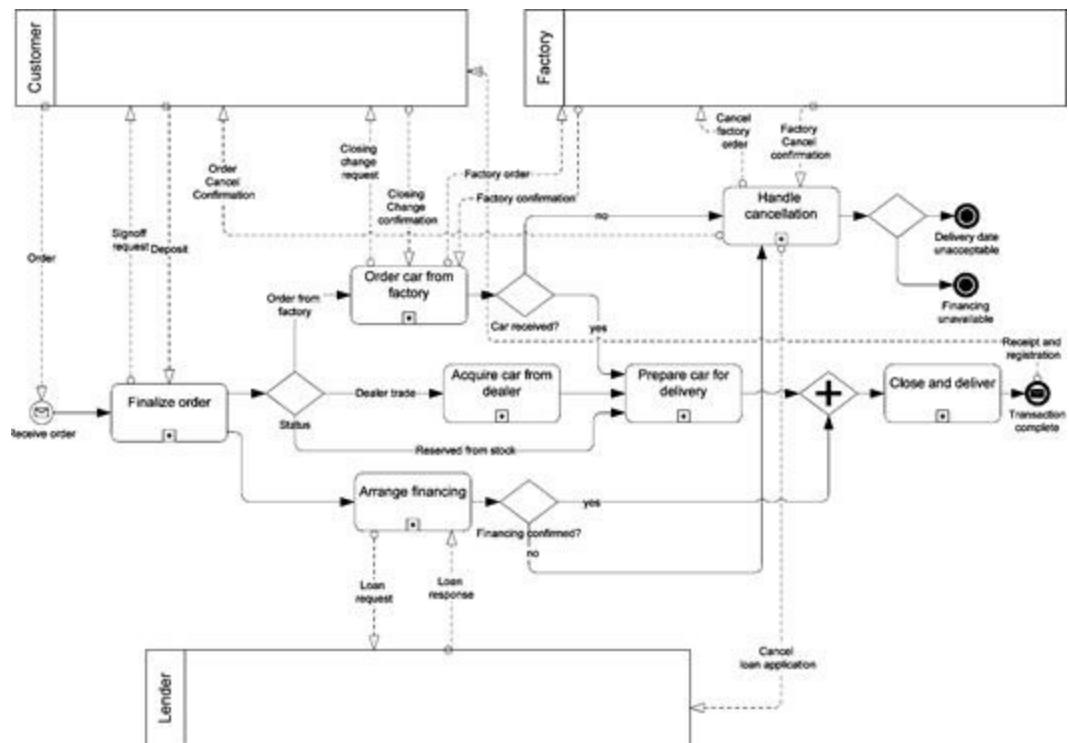


Figure 5-6. Top-level diagram, with message flows

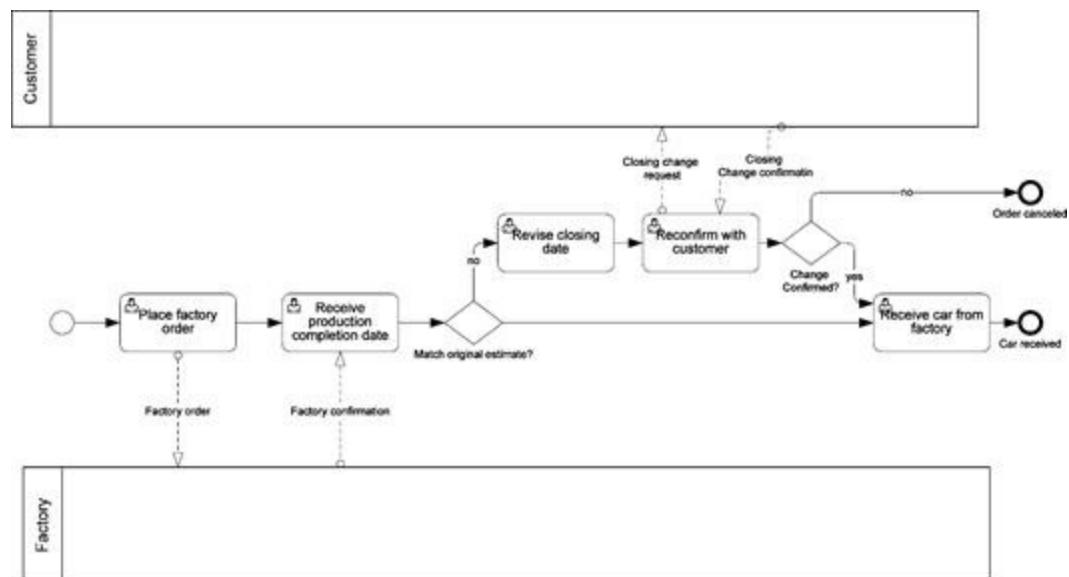


Figure 5-7. Order car from factory, child level, with message flows

In Microsoft Visio, you can selectively suppress the display of message flows or any other diagram element by placing them in a hidden diagram layer. This is far better than creating separate high-level
*****ebook converter DEMO Watermarks*****

and detailed models and trying to keep them in sync. Just add a new layer, select the shapes and connectors to put in that layer (Figure 5-8), and set the properties of that layer to be invisible or, by coloring them light gray, barely visible. In Figure 5-9, only the Customer pool and message flows connected to it are left in the visible layers.

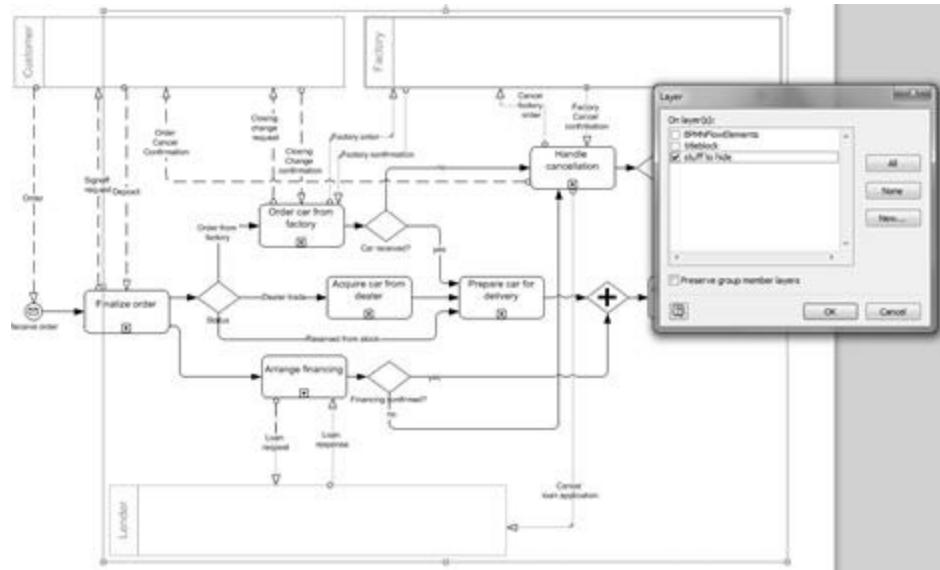


Figure 5-8. Placing selected shapes in an invisible Visio layer

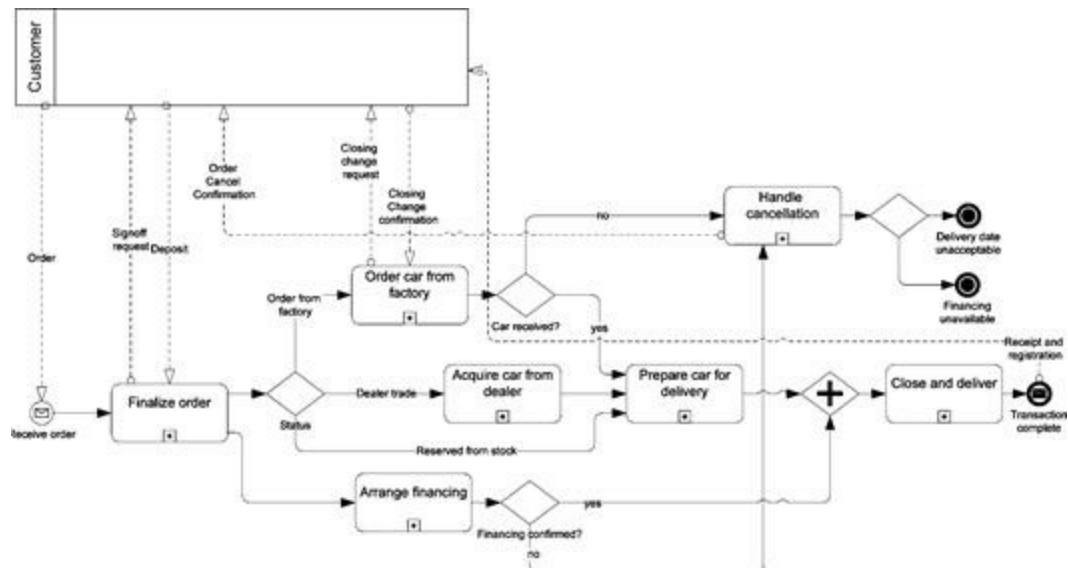


Figure 5-9. Top-level diagram with Factory and Lender collaboration hidden

Method Recap

We've now covered the Method. It's pretty simple, really. Let's review the steps:

1. Agree on process scope, when it starts and ends, what the instance represents, and possible end states.
2. Enumerate major activities in a high-level map, ten or fewer, each aligned with the process instance. Think about possible end states of each activity.
3. Create top-level BPMN diagram. Arrange high-level map activities as subprocesses in a BPMN process diagram, with one top-level end event per process end state. Use gateways to show conditional and concurrent paths.
4. Expand each top-level subprocess in a child-level diagram. If a subprocess at parent level is followed by a gateway, match subprocess end state and gateway (or gate) labels.
5. Add business context by drawing message flows between the process and external requester, service providers, and other internal processes, drawn as black-box pools. Message flows connecting to collapsed subprocess at parent level should be replicated with same name in the child-level diagram.
6. Repeat steps 4 and 5 with additional nested levels, if any.

Chapter 6

BPMN Style

The Method helps you establish consistency in the structure of your BPMN models, but by itself it does not ensure that your diagrams can stand on their own, revealing the process logic clearly and completely without the need for supplementary documentation. The rules of the BPMN specification won't do that, either. Maximizing shared understanding of BPMN diagrams requires application of additional conventions that I call *BPMN style*.

In my BPMN classes, I used to teach BPMN style as recommended "best practices," since these conventions, after all, are not required by the BPMN 2.0 specification. But, like childhood admonitions to "eat your vegetables," best practices have a way of being ignored by modelers, especially when they are in a hurry. So now I have distilled the elements of BPMN style to a set of rules – I call them *style rules* – that can be used to validate models in a tool. Students in my training, for example, can validate against

[\[13\]](#)

them directly in Process Modeler for Visio from itp commerce¹³, and I have made style rule validation [\[14\]](#) available through my own online tool¹⁴. Before students in my training can submit their certification exercises for approval, I now insist that they validate the diagrams against both the official rules and the style rules, and fix all the errors. This has made a huge difference in both the quality of the submissions and the speed of student learning.

In this chapter we'll look at some of the basic principles of BPMN style and the important style rules applicable to the Level 1 palette.

The Basic Principle of BPMN Style

The basic principle of BPMN style is simply this: *The process logic should be unambiguous from the diagram alone.* That's what we mean by "good BPMN." Remember "process logic" is not the internal logic of a process task. Task logic is important, of course, but BPMN has little to say about it. Process logic is the logic of the sequence flows: When an activity ends, what happens next, under what conditions? It is about the *sequencing* of process activities, not the inner workings of individual tasks. We can show the inner workings of an activity by modeling it as a subprocess, but BPMN cannot "look inside" a task.

In the diagram we have only a few visual elements available to convey the process logic: the basic shapes, their internal icons and markers, border style, diagram placement, and – last but not least – their labels. BPMN style depends on using those elements to the fullest. *Labeling* is a particularly important aspect of BPMN style, but many modelers are uncommonly stingy with labels. Style rules not only require labels for certain diagram elements, but may require matching the label text with the label of another diagram element. In the XML, BPMN uses pointers to element IDs to link them together, but these IDs and pointers don't show up in the diagram; only the labels do.

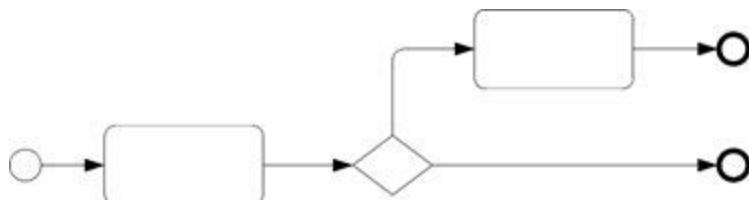


Figure 6-1. A "valid" but meaningless process

It may surprise you to know that Figure 6-1 is *valid* according to the BPMN 2.0 specification. It breaks no official rules, but it provides no useful information. The activities are not labeled. The gateway, its gates, and the end states are similarly unlabeled.

6 infos			
Violation	Type	Element	Message
120701	Task	(unnamed)	Activities should be labeled.
120701	Task	(unnamed)	Activities should be labeled.
230701	Exclusive Ga...	(unnamed)	An exclusive gateway should have at most one unlabeled gate.
230702	Exclusive Ga...	(unnamed)	An exclusive or inclusive gateway with an unlabeled gate should be labeled.
312702	End Event	(unnamed)	If there is more than one End Event in a process level, all should be labeled with the name of the end state.
312702	End Event	(unnamed)	If there is more than one End Event in a process level, all should be labeled with the name of the end state.

Figure 6-2. Style rule violations in Figure 6-1

That is why style rules are important. Figure 6-2, the validation report from the itp commerce tool, lists six violations from this simple diagram alone, all related to labeling. (In the tool, each violation is hyperlinked to the shape it references.) Like spelling and grammar checking in word processing, validation – including style rule validation – is something every modeler should perform on a regular basis. Many violations do not imply ignorance of the rules, but simply a hurry to finish.

Of course, the diagram must also obey the official rules of the BPMN spec. That's obvious, and even more important than following the style rule conventions. But that is not as easy as it seems. For one thing, the spec does not enumerate its rules. It has no Appendix where they are all listed and numbered. Instead, the rules are sprinkled throughout the narrative of this 508-page document, where

they refine and override various other requirements imposed by the BPMN metamodel (UML class diagrams) and its associated XML schema. In a sense, BPMN 2.0 has three sources of truth. They are intended to be aligned, but that is not always the case. Each tool, therefore, must make its own interpretation of the rules.

In any case, if you are serious about process modeling, you should avoid any BPMN tool that cannot validate your diagrams against some interpretation of the rules of the BPMN specification. Fortunately, most BPMN tools offer such validation. Good BPMN always starts with adherence to the rules of the spec.

Style Rules

A number of style rules are basic principles of composition, while others are specific rules of usage supporting validation in a tool. The important style rules applicable to Level 1 modeling are listed below.

1. Use icons and labels to make the process logic clear from the printed diagram.

Maximize use of BPMN's visual elements, including icons, markers, and especially labels. Label all activities, even subprocesses. Label end states. Label the sequence flows out of an exclusive gateway. Label pools and message flows. Identify task types and event triggers with icons. If some aspect of the process logic cannot be conveyed unambiguously from the BPMN elements alone, use a text annotation.

2. Make models hierarchical, fitting each process level on one page.

This principle essentially says use the Method or an equivalent methodology that results in a hierarchical model structure. The top-level diagram should capture the end-to-end process on one page and show its interactions with external entities using message flows. Each subprocess in a process level should be expanded in a separate child-level diagram, and this nesting can go on as deep as you'd like. With hierarchical modeling, as child-level detail is added, no change is needed to the parent level diagrams.

3. Use a black-box pool to represent the Customer or other external requester or service provider.

A common beginner mistake is to insert activities in the Customer or other requester pool (Figure 6-3, left).

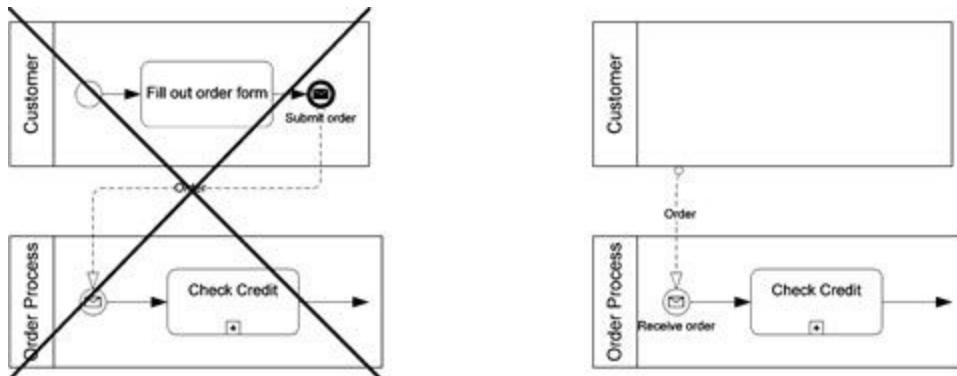


Figure 6-3. Customer and other external participants should be modeled as black-box pools

It's a mistake because you don't know the logic of the Customer's process. Submitting the order is not the end of the interaction. Other messages may be exchanged downstream – confirmation, invoice, failure notice, perhaps other notifications and requests. You cannot anticipate the Customer's internal process for all of that, and you are not allowed to connect message flows to the boundary of a process pool. The solution is to make the Customer a black-box pool.

4. Begin customer-facing processes with a Message start event receiving a message flow from the Customer pool.

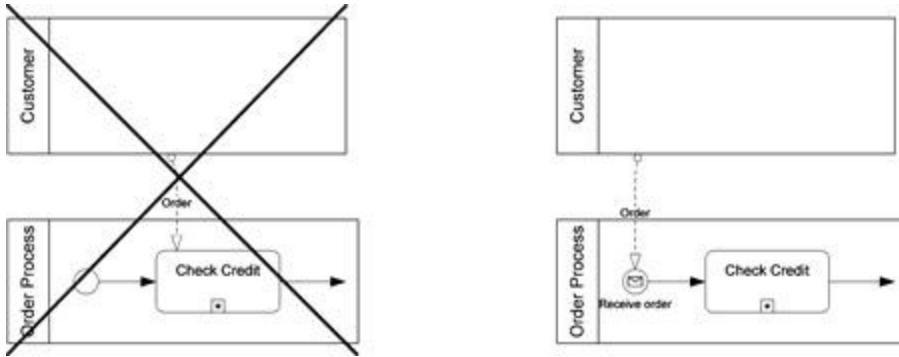


Figure 6-4. Message start event means the message instantiates the process

A process that is initiated by request should be modeled with a Message start event that receives a message flow from the requester pool. Message start (Figure 6-4, right) implies that a new instance of the process is created whenever the message is received. Receiving the message in an activity following a None start (left) implies manual start by a task performer, followed by a wait for the message. Also, message flow to an activity implies the *possibility* of the message, whereas a Message start event implies the *certainty* of the message.

5. If you can, model internal organizational units as lanes within a single process pool, not as separate pools. Separate pools imply independent processes.

There are a few occasions when your internal business process should be modeled as multiple pools, meaning multiple BPMN processes, but most of the time it is best to model it as a single BPMN process, in a single pool. Representing each organizational unit that performs process activities as a separate pool (Figure 6-5, left) is usually incorrect. This implies each unit's process is independent of the others, not a fragment of a single end-to-end process. Representing the organizational units as lanes within a single pool (Figure 6-5, right) signifies a single BPMN process end-to-end.

If this cannot be done, it is usually because there is not alignment of the process instance across the organizational units. For example, Figure 6-5 says that each order is invoiced separately. You could not model it this way if the *Billing* process was based on monthly statements rather than invoices for each order. In that case, you could use separate pools for the *Order* and *Billing* processes (Figure 6-6). Here the processes communicate via a shared data store.

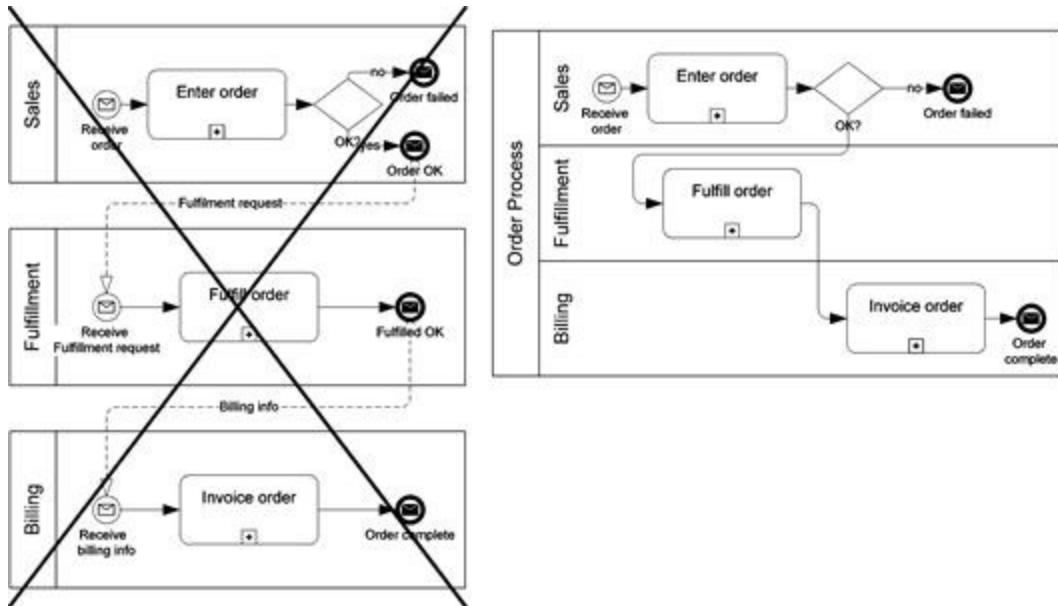


Figure 6-5. Organizational units performing process activities should normally be represented as lanes within a single pool, not separate pools

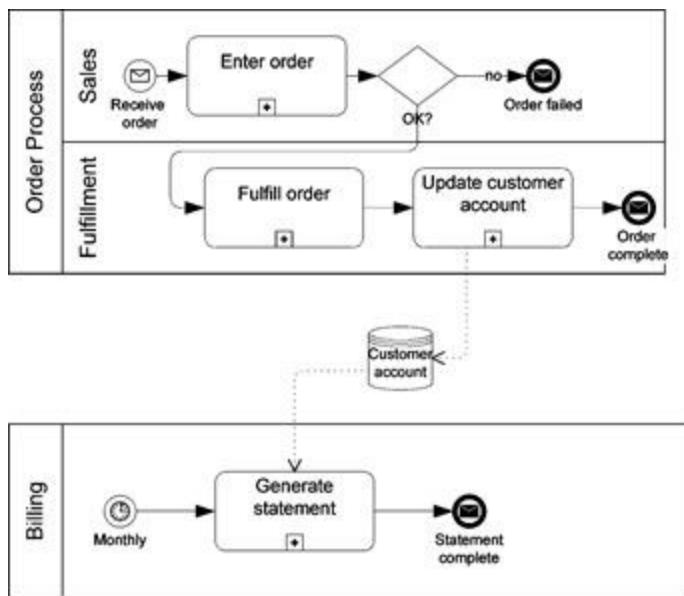


Figure 6-6. Multiple pools may be required if there is not 1:1 correspondence between the process instances

6. Label process pools with the name of a process; label black-box pools with a participant role or business entity.

Labeling black-box pools using generic role or entity names like Seller, Manufacturer, or Lender is good practice, but it is better to label process pools with the name of the process. Unfortunately, the BPMN 2.0 specification itself encourages the practice of labeling process pools with the name of an organization or role. So let me elaborate on why it is a bad thing.

Part of the reason is technical. In the BPMN metamodel and XML, the semantic element directly referenced by a pool shape is called a *participant*. But "participant" is not the same as a task *performer*. It merely identifies a counterparty in the transaction relating the process requester and provider. Each

participant, however, is associated with at most *one* BPMN process. Thus a pool simultaneously represents *both* a participant and a process.

So the question becomes, what does the pool *label* represent? The spec is silent on this, but the normal convention is that it represents the *name* attribute of the associated semantic element, in other words the *participant name*. So each pool with the name *My Company* defines a participant named *My Company*. Even though they have the same name, technically those could represent distinct participants (and point to different processes), because the unique identifier of any semantic element is not its *name* but its *id* attribute.

But here is where Method and Style comes in. The *id* is not visible in the diagram; what you see is only the label, the *name*. Method and Style says that what you see in the diagram is what counts, not information hidden in invisible XML. By that principle, a model should never have two pools with the same name that secretly mean different semantic entities. In fact, this should apply not only across the diagrams of a single BPMN model, but across all of your organization's BPMN models, if they interact with each other.

I believe the only way this works is if the pool label also names the *process*. That implies that for a process pool the *participant name is the same as the process name*, not the name of a department or company. That might seem odd, but consider why this is a good thing. First, there is no other place in the diagram where the process name appears. Each distinct process should have a different name in the diagram. If you label white-box pools with the name of your organization, many will have the same name. Interconnecting them with message flows then suggests the message flow source and target participants are the same participant... which is not allowed and makes no sense.

7. Indicate success and exception end states of a process or subprocess with separate end events, and label them to indicate the end state.

This principle of composition is part of the Method, as we saw in Chapter 5. More than any other single characteristic, attention to activity and process end states distinguishes the Method and Style approach. Most BPMN modelers typically use a single end event to represent process level completion regardless of end state. That hides valuable information, however, and makes it harder to trace the process logic from the top level down in a hierarchical model. It's better to use a separate end event for each end state you want to distinguish (Figure 6-7). If the end state has a bearing on the subsequent flow, then it is especially important to show the relevant end states as separate end events.

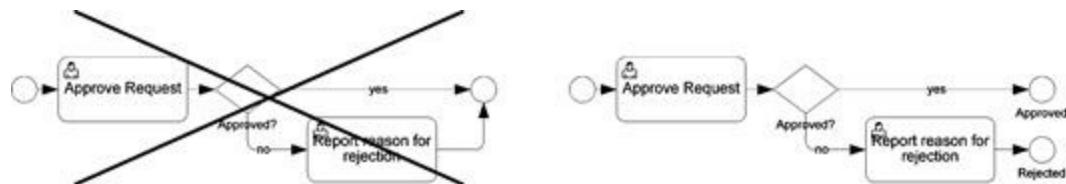


Figure 6-7. Represent distinct end states with separate end events, labeled with the end state

8. Label activities VERB-NOUN.

Activities, including subprocesses, represent *work* or *actions* performed in the process, not functions or states. Therefore you should give them names of the form VERB-NOUN. For example:

- *Check credit* (action), not *Credit check* (function) or *Credit OK* (state)
- *Approve loan* (action), not *Loan approval* (function) or *Loan rejected* (state)

- *Receive report* (action), not *Report received* (state)
9. ***Use start event trigger in top-level process to indicate how the process starts.***
- Use a Message start event to signify a process triggered by external request. The event should be labeled *Receive [message flow name]*.
 - Use a Timer start event to signify a scheduled process, typically recurring. The event should be labeled with the recurring schedule, such as *Monthly* or *Mondays at 8am*.
 - Use a None start event to signify a process manually started by a task performer. It may be left unlabeled.
10. ***If a subprocess is followed by a gateway labeled as a question, the subprocess should have multiple end events, and one of them should match the gateway label.***

Another way of saying this is if the flow following a subprocess branches into two alternative paths, the gateway should be labeled *[end state 1]?*, where *[end state 1]* is the name of one of the child-level end states, and the gates should be labeled *yes* and *no*. Instances reaching *end state 1* of the subprocess follow the *yes* path out of the gateway, and those reaching the other end state follow the *no* path.

The top diagram in Figure 6-8 is incorrect because the *Approved* and *Rejected* end states of the subprocess are combined in a single end event. Even though the logic is easy to follow in this example, the style rule says there should be *two* end events, and one of them should be named *Approved*, matching the gateway label *Approved?* With more complex models, this assists top-down traceability of the process logic through the diagram hierarchy.

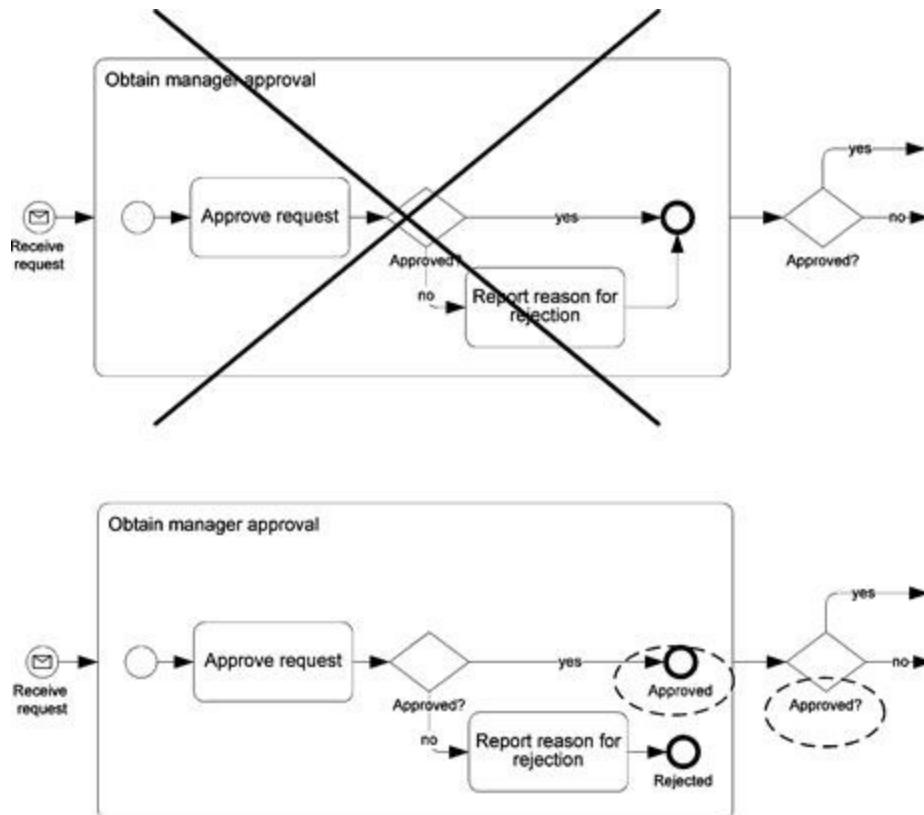


Figure 6-8. Subprocess followed by XOR gateway should have two end states, one matching the name of the gateway

11. Show message flow with all Message events.

Message flows are optional in BPMN, and even in the illustrations in this book, where they do not add value I may not show them. But in a real, finished BPMN model, I think it is best to show the message flow connected to all Message events. In the Level 1 palette, we have seen Message start and end events, and we will see a few more at Level 2. Style rule validation flags any Message event without an attached message flow.

12. Match message flows in parent- and child-level diagrams.

A second top-down traceability rule requires replicating in the child-level diagram all message flows connecting to a collapsed subprocess. The count and labels of message flows should match at parent and child levels.

Figure 6-9, taken from the chapter on the Method, shows four message flows connecting to *Order car from factory* in the parent level. The rule says those same four message flows, with the same names, should be replicated in the child-level expansion, Figure 6-10.

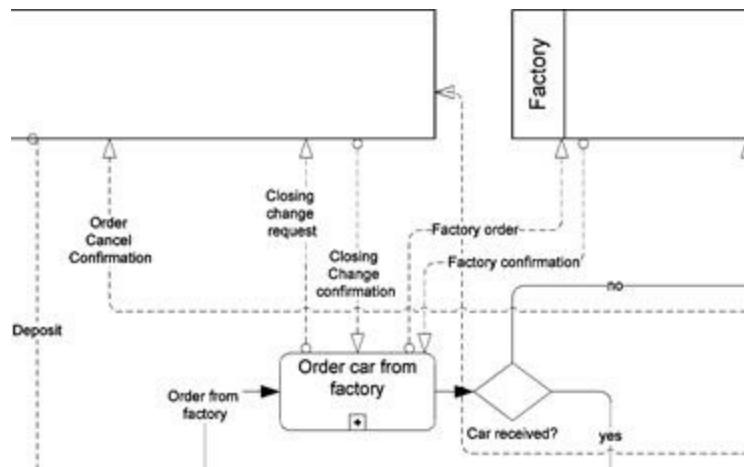


Figure 6-9. Four message flows connect to *Order car from factory* in parent-level diagram

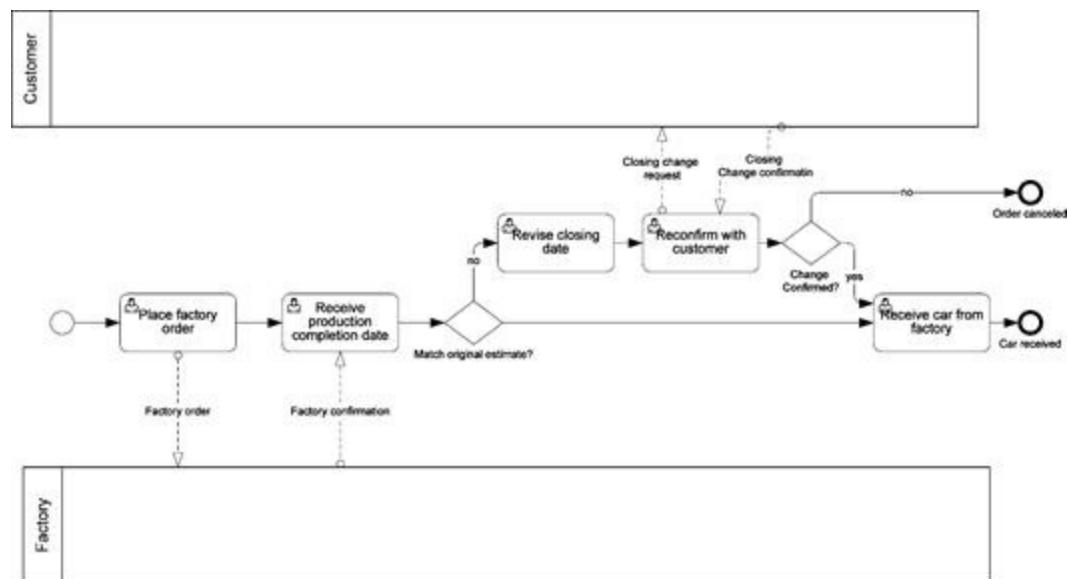


Figure 6-10. Message flows replicated in expansion of *Order car from factory*

13. Label message flows directly with the name of the message.

It is not enough simply to draw the message flows. You also need to label them. The label should be the name of the message, such as *Rejection notice*. It should not be the name of a state, such as *Rejected*, or the action of sending or receiving, such as *Send rejection*.

It is incorrect to leave the message flow unlabeled and identify the message by an associated data object (Figure 6-11, left). In BPMN 2.0, a data object cannot be associated to a message flow. It is technically legal to use a Message icon attached to the message flow (Figure 6-11, center), but the best way to label a message flow is directly (Figure 6-11, right).

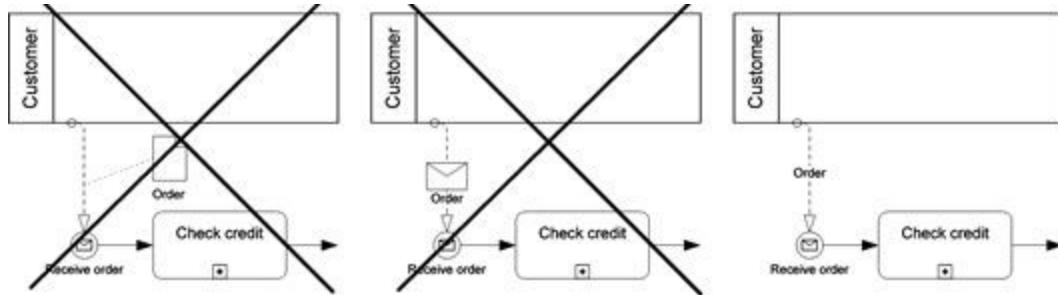


Figure 6-11. Label message flows directly with the name of the message

14. Two end events in a process level should not have the same name.

If they represent the same end state, combine them in a single end event. If they represent distinct end states, give them different names.

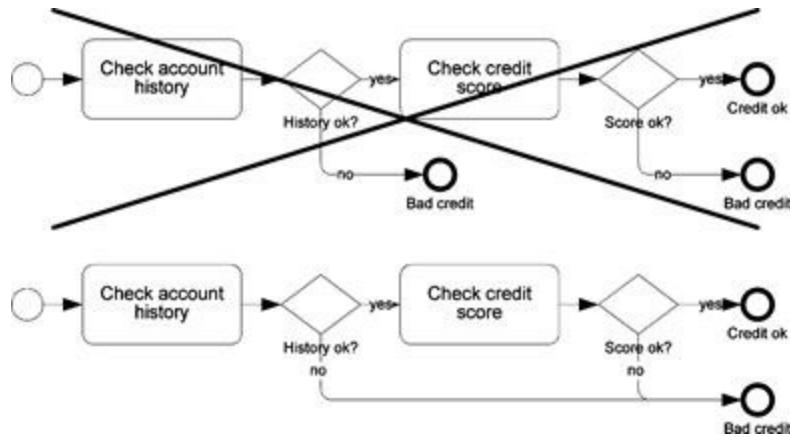


Figure 6-12. Two end events in a process level should not have the same name

15. Two activities in a process model should not have the same name.

If they represent the same activity, use a call activity referencing the same global task or process. If they represent different activities, give them different names. This is self-explanatory, but I sometimes see a subprocess containing a task of the same name. That is incorrect; just give them different names (Figure 6-13).

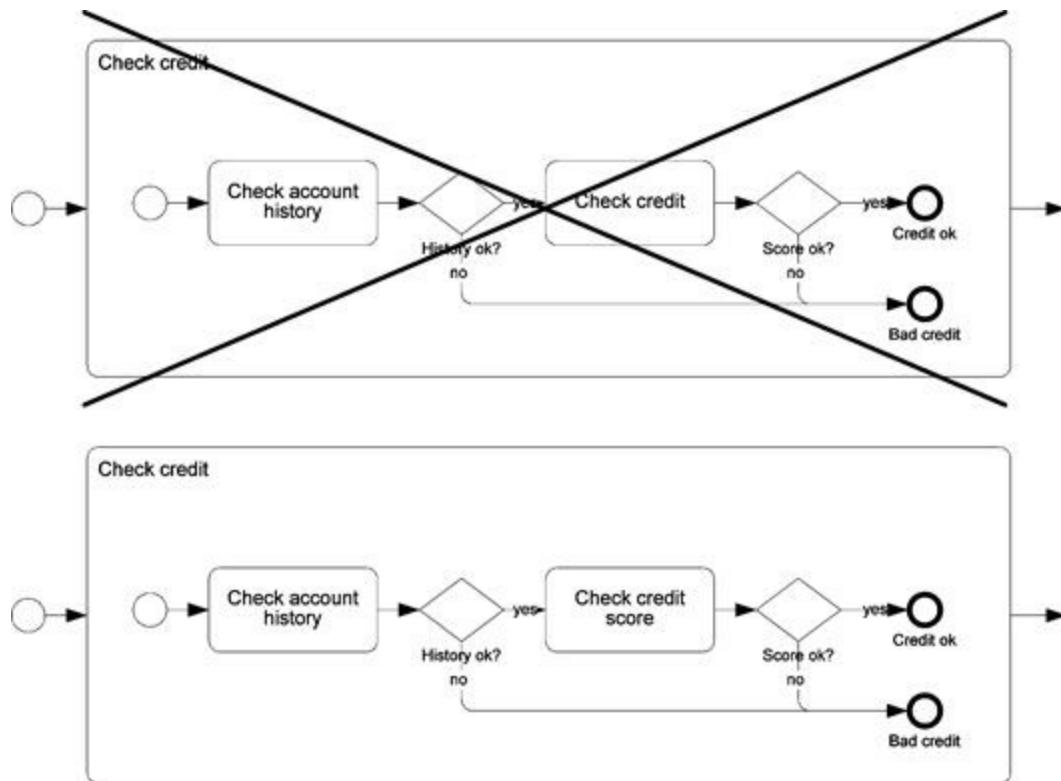


Figure 6-13. Two activities should not have the same name

16. A subprocess should have a single None start event.

Except for “parallel box” subprocesses, which have no start event, a subprocess should contain *exactly one start event*, and it must be None type (no trigger). In a top-level process, multiple start events are used to represent alternative triggers, but triggered start events are not allowed in a subprocess. (Note: this does not apply to an *event subprocess*, a type of exception handler that is not part of the Level 1 or Level 2 palette. We’ll talk about event subprocesses in Chapter 7.)

The spec does not specifically say that a subprocess may have only one None start event, but a subprocess with two None start events is ambiguous. Do those events represent parallel or alternative start nodes? In a top-level diagram, they mean alternative start points. In a subprocess, always use a single start event, removing the ambiguity. For example, the left diagram in Figure 6-14 is ambiguous, but the right diagram indicates *Receive application* and *Receive payment* are alternative, not parallel, activities. A parallel split following a single start event (with a parallel join before *Complete registration*) would indicate parallel activities.

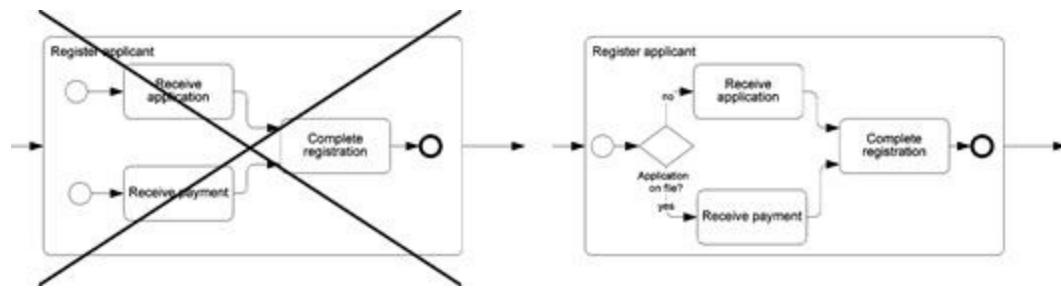


Figure 6-14. A subprocess should have a single None start event

17. A process pool in child-level diagram (if drawn) should be labeled with name of the top-level

*****ebook converter DEMO Watermarks*****

process, not the name of the subprocess

I see this all the time: A collapsed subprocess *Check Credit* in the pool labeled *Order process* in the top-level diagram is expanded in a separate diagram, following the hierarchical style. The child-level expansion is enclosed in a pool labeled *Check Credit*. That is incorrect. You could omit the process pool in the child-level diagram, but if you draw it, it must also be labeled *Order process*.

The reasoning behind this is the same as number 6. It is really implied by the metamodel and XML schema, but since the spec doesn't talk about this explicitly, I'll call it a style rule. The pool shape on the child level points to a single *participant*, and that participant points to a single top-level *process*. It cannot point to a subprocess. Method and Style assumes that the modeler creates the diagram, and the diagram generates the XML, so a pool called *Check Credit* would generate a new participant with that name and (probably, but tool-dependent) also a process with that name. But *Check Credit* is not an independent process, just a subprocess of *Order process*.

The ambiguous relationship between pool and process names, inherent in the BPMN 2.0 XML, actually requires the tool to prompt the modeler for additional information. (The reason this has been so little remarked up to now, I think, is that very few tools have thought about the XML serialization details yet, even though they are crucial to model interchange. This is exactly the subject of the BPMN Implementer's Guide section of this book.)

In the tool I use for most of my BPMN training, Process Modeler for Visio from itp commerce, the modeler can tell the tool that two (or more) pools in the model reference the same participant. When you do this, the pool label is changed automatically to that of the referenced participant, and the XML structure is produced correctly. If you don't do that, labeling the child-level expansion of a subprocess may give a structure that is not be what you intended. In the example described above, the child-level diagram of *Check Credit* is, in the XML, actually another top-level process named *Check Credit*, and in the original *Order process* the contents of the *Check Credit* subprocess are empty! That structure would make sense if the collapsed subprocess were converted to a call activity... but it's not.

Bottom line: Follow the style rule as described above.

18. In a hierarchical model, a child-level diagram may not contain any top-level processes.

This is a technical point, discussed more fully in the BPMN Implementer's Guide section. I add it here as well because violating the rule can create XML structures that are either ambiguous or not what the modeler intended. From the modeler's perspective, it's best to consider each diagram (page) of a hierarchical model as either a top-level diagram or a child-level diagram; it may not be both simultaneously. A child-level diagram may contain as many black-box pools as you want, but it may not contain activities of any process other than that of the parent-level subprocess. A top-level diagram may contain elements of more than one process.

19. Don't use an XOR gateway to merge alternative paths, unless into another gateway. Just connect the sequence flows directly.

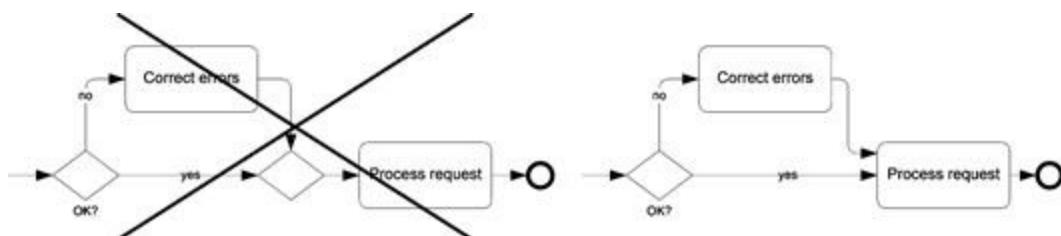


Figure 6-15. Don't use XOR gateway to merge alternative paths

20. *Don't use an AND gateway to join parallel paths into a None end event. A join is always implied at a None end event.*

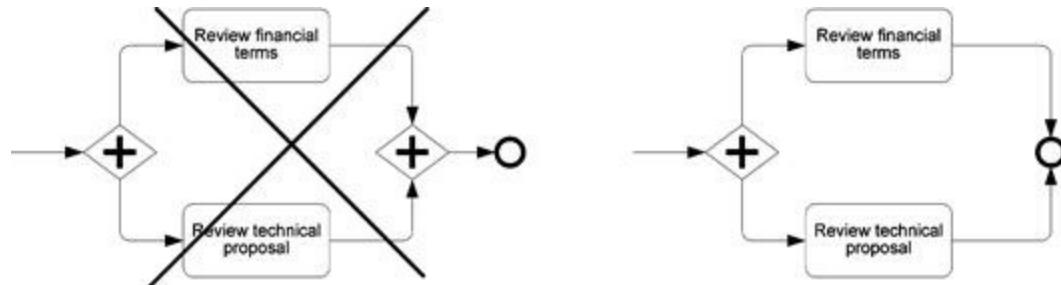


Figure 6-16. Don't use gateway join into None end event

Official BPMN 2.0 Rules

One principle of BPMN style is so obvious I shouldn't need to say it: The model should not violate any official rules of the BPMN 2.0 specification. I mentioned this at the beginning of the chapter, along with the reason why different tools may give different validation results on the same BPMN model. Here are some of the basic rules that apply to the Level 1 palette. We'll revisit a more complete list in Chapter 11.

21. *A sequence flow may not cross a pool (process) boundary.*

You cannot, for example, connect the end event of *Process 1* to the start event of *Process 2* using a sequence flow. You can do it, however, with a message flow.

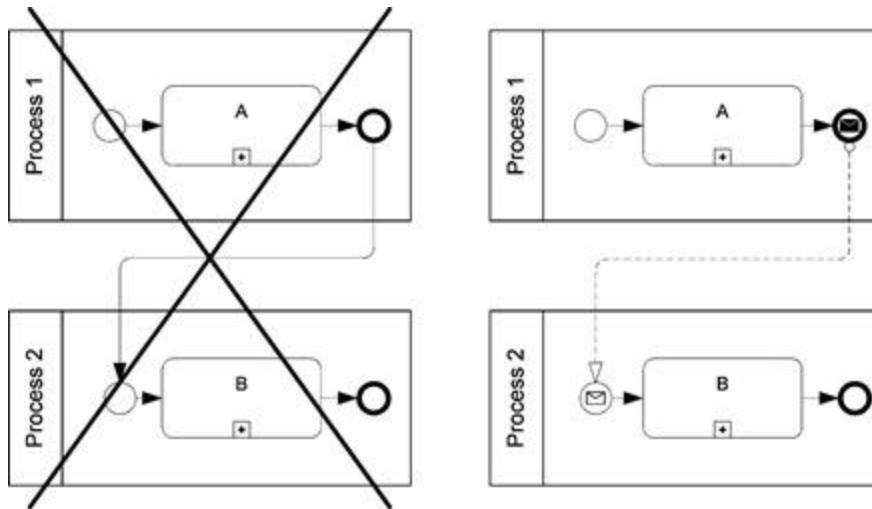


Figure 6-17. A sequence flow may not cross a pool boundary

22. *A sequence flow may not cross a subprocess boundary*

I most often see this error when the modeler tries to wrap a process fragment in an expanded subprocess shape after the fact. The left diagram in Figure 6-18 is incorrect, since sequence flow cannot cross the subprocess boundary. All sequence flows in the child-level expansion must be completely contained *inside* the subprocess shape.

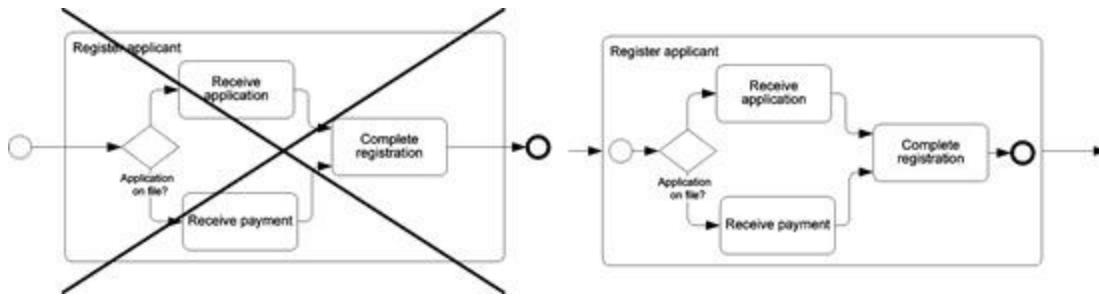


Figure 6-18. Sequence flow may not cross a subprocess boundary

23. *A message flow may not connect nodes in the same pool*

A “message” in BPMN does not mean the same as a “message” in English. For instance, an email between two tasks in the same process is *not* a BPMN message. A BPMN message is, by definition,

exchanged between the process and an entity *outside* the process. Consequently, the head and tail of a message flow may not be in the same pool.

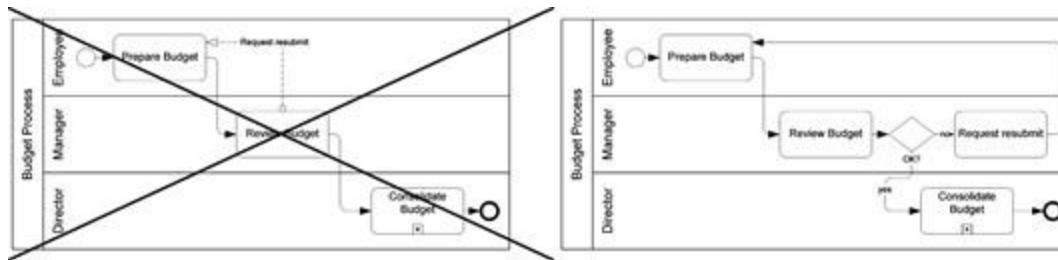


Figure 6-19. A message flow may not connect nodes in the same pool

24. *A sequence flow may only connect to an activity, gateway, or event, and both ends must be properly connected.*

You may not connect a sequence flow, for example, to a pool, a data object, or another sequence flow.

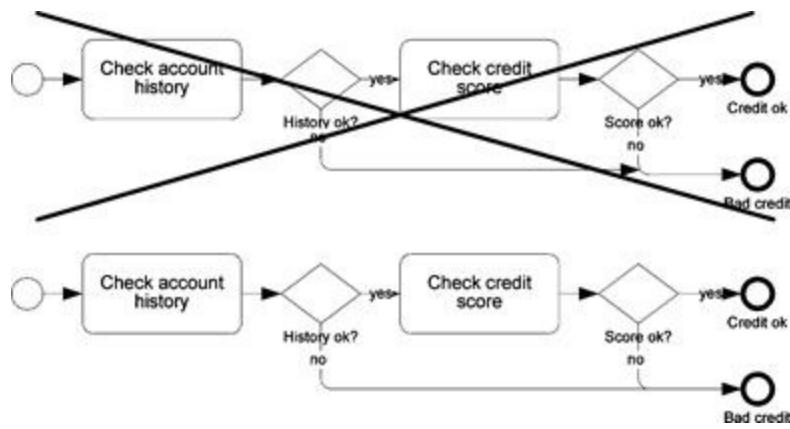


Figure 6-20. A sequence flow may not connect to another sequence flow, only to an activity, gateway, or event

25. *A message flow may only connect to an activity, Message (or Multiple) event, or black-box pool, and both ends must be properly connected.*

You may not connect a message flow, for example, to a process pool boundary, a data store, or a gateway, or leave one end unconnected.

Part III: Method and Style – Level 2

Chapter 7

Events

The most significant change between BPMN Level 1 and Level 2 is the emphasis on *events*, the circle shapes in the diagram. The BPMN spec defines an event as “something that happens” in a process. It would be more accurate to say that a BPMN event describes *how the process responds to a signal that something happened*, or – in the case of a throwing event – *how the process generates a signal that something happened*. The type of signal, called the *trigger* for catching events and the *result* for throwing events, is indicated by the icon inside the circle.

At Level 1, each step in the process is triggered by the completion of the prior step. When an activity completes, the sequence flow out of it initiates the next step in the process. That is the usual way a process moves along, but events let you describe additional behaviors. For example, you can say the process *pauses* until the trigger occurs, and then *resumes*. Or you can say that if the trigger occurs while an activity is running, the activity is terminated immediately and some other exception activity is initiated immediately. Or, alternatively, the activity continues but something else is initiated in parallel with it. BPMN provides a visual language for all these event-triggered behaviors.

When you hear people say that “BPMN is too complicated for business people,” usually what they are talking about is its bewildering array of event types. In fact, they are specifically talking about the table shown in Figure 7-1, clipped directly from the BPMN 2.0 specification. That table has 13 rows, one for each trigger/result type, and 8 columns, a total of 104 distinct combinations. Also, half of those cells are *empty*, meaning the combination is not allowed!

Types		Start			Intermediate			End	
		Top-Level	Event Sub-Process Interrupting	Event Sub-Process Non-Interrupting	Catching	Boundary Interrupting	Boundary Non-Interrupting	Throwing	
None		○						○	○
Message		✉	✉	✉	✉	✉	✉	✉	✉
Timer		⌚	⌚	⌚	⌚	⌚	⌚		
Error		⚠			⚠			⚠	⚠
Escalation		Ⓐ	Ⓐ		Ⓐ	Ⓐ	Ⓐ	Ⓐ	Ⓐ
Cancel					☒			☒	☒
Compensation		⤠			⤠			⤠	⤠
Conditional		☰	☰	☰	☰	☰	☰		
Link					⤠			⤠	
Signal		△	△	△	△	△	△	△	△
Terminate									●
Multiple		○	○	○	○	○	○	○	○
Parallel	Multiple	⊕	⊕	⊕	⊕	⊕	⊕		

Figure 7-1. BPMN 2.0 events – full element set

I agree that if you needed to memorize this table, BPMN would indeed be too complicated for anyone. Fortunately you don't. In Level 1 we learned about None, Message, and Timer start events, and None, Message, and Terminate end events. (We also saw Multiple start and end events, which are distinct shapes but not additional semantic elements; they just mean "more than one trigger or result".)

The Level 2 palette now adds *intermediate events*, the ones with the double ring, and a few additional triggers. The Analytic subclass of BPMN 2.0, i.e., the *official* Level 2 palette, includes the Level 1 triggers plus Error, Escalation, Conditional, Signal, and Link. We are going to mainly focus on the "Big 3" event

*****ebook converter DEMO Watermarks*****

types – Timer, Message, and Error. Those are the ones you really need to know, and it's a small and readily learnable subset (Figure 7-2). Afterward we'll also briefly discuss Escalation, Signal, Conditional, and Link, as well as event subprocesses. We'll defer discussion of Cancel and Compensation events until Chapter 10.

Types	Start		Intermediate			End
	Top-Level		Catching	Boundary <i>Interruption</i>	Boundary <i>Non-Interruption</i>	
None	○					○
Message	✉		✉	✉	✉	✉
Timer	⌚		⌚	⌚	⌚	
Error			⚡			⚡

Figure 7-2. BPMN 2.0 events – the ones you need to know

Event-Triggered Behavior

Event-triggered behavior refers to process actions initiated immediately upon occurrence of a specific trigger signal. In BPMN Level 1 we saw one example of this in the triggered start event, which always creates a new instance of the process. A Message start event creates a new process instance whenever it receives the message represented by the incoming message flow. A Timer start event creates a new process instance whenever the recurring schedule dictates. Instantiation is presumed to occur *immediately* upon detection of the trigger. In an executable process, instantiation is immediate and automatic. In a non-automated process, we use Message and Timer start events even when instantiation is human-mediated, as long as it is effectively triggered by the arrival of the message or timer signal.

Here we turn our attention to *intermediate events*, the ones with the double ring. As the name suggests, intermediate events occur after the start of a process level and before the end. But the precise meaning of an intermediate event depends on the details of its representation – the icon inside, the color of that icon, the double-ring line style, and its placement in the diagram. The four columns of intermediate events in Figure 7-1 (plus the two event subprocess start columns) actually signify different triggered behaviors for a given trigger signal.

- A *throwing* intermediate event, with the black icon inside, means the process *generates* the trigger signal. Only a few intermediate events support the throwing behavior. By convention, throwing the signal occurs immediately and automatically as soon as the incoming sequence flow arrives, and the process continues immediately afterward on the sequence flow out of the throwing event. (In a non-automated process, we just pretend it is automatic and immediate.) A Message intermediate event supports throwing behavior, but not Timer or Error.
- A *catching* intermediate event, with the white icon inside, drawn with sequence flow in and sequence flow out, means the process *waits* for the trigger signal. When the trigger signal arrives, the process resumes on the sequence flow out of the event. Most intermediate events support this behavior, but not all. Message and Timer events do, for example, but not Error. In other words, a process can wait for a message or timer signal, but it cannot wait for an error.



Figure 7-3 Catching (left) and throwing (right) Message event

- A catching intermediate event drawn on the boundary of an activity, called a *boundary event*, does not signify waiting. It means while the activity is running, the process *listens* for that signal. If it occurs before the activity completes, the sequence flow out of the event, called the *exception flow*, is triggered. On the other hand, if the activity completes without the occurrence of the boundary event signal, the exception flow is ignored and the process continues on the sequence flow out of the activity, called the *normal flow*.

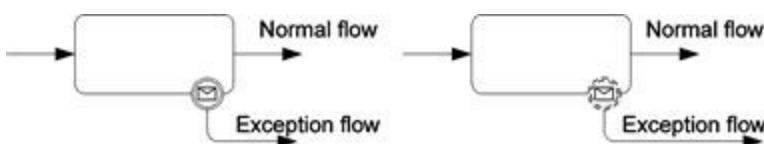


Figure 7-4. Interrupting (left) and non-interrupting (right) Message boundary event

A boundary event has no incoming sequence flow and must have *exactly one* outgoing sequence flow, the exception flow. There are two types of boundary event. An *interrupting boundary event*, denoted by the solid double ring, means the activity that the event is attached to is terminated immediately upon occurrence of the trigger signal. The process does not exit on the normal flow but continues immediately on the exception flow, the sequence flow out of the event. Message, Timer, and Error events all support interrupting boundary event behavior.

A *non-interrupting boundary event*, denoted by the dashed double ring, does not terminate the activity. That activity continues uninterrupted, and when it completes, the process continues on the normal flow, the sequence flow out of the activity. But, in addition, upon occurrence of the trigger a new parallel path of the process is instantiated immediately on the exception flow. In this case, the exception flow represents actions taken *in addition* to those on the normal flow. Non-interrupting events are new in BPMN 2.0, probably the most significant addition to the palette. Message and Timer events support non-interrupting boundary event behavior, but Error boundary events are always interrupting.

Understanding how to use Timer, Message, and Error events correctly is the key to BPMN Level 2. Let's take them one at a time.

Timer Event

Catching Timer Event

Drawn with sequence flow in and out, a catching Timer intermediate event represents a *delay*. It means either *wait for [specified duration]* or *wait until [specified date/time]*. For example, you might want to wait for a short while before retrying an activity such as polling for posted data (Figure 7-5, top). You can also use a catching Timer event to model a wait for a scheduled action, such as a semi-monthly check run (Figure 7-5, bottom).

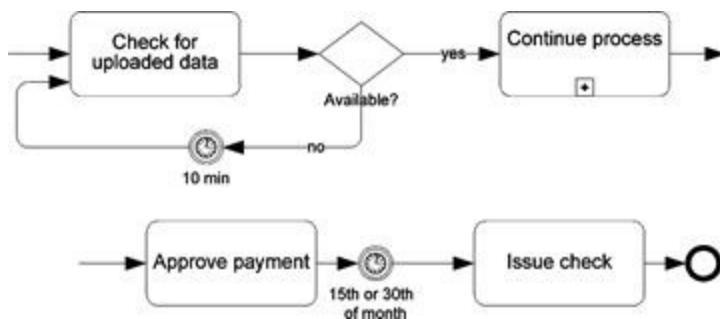


Figure 7-5. Delay using Timer event

A catching Timer event does NOT mean wait for something to occur, such as a response to a request; that would be a Message event. And you don't use a catching Timer event to signify that an activity "usually" takes three days; you can use a Timer boundary event to say *what happens if the activity takes longer* than three days.

BPMN provides XML attributes for the Timer event to specify the duration or a specific date/time, but these are not directly visible in the diagram (and not in the Analytic subclass). Therefore the duration or date/time value is represented in the diagram by the label (*name*) of the Timer event.

Timer Boundary Event

A Timer boundary event acts like a combination stopwatch and alarm clock. By convention, the stopwatch starts when the activity the event is attached to starts. An activity "starts" when the sequence flow into it arrives, not when the performer decides to begin work on it. If the activity is not complete by the Timer event's specified duration or date-time parameter, the alarm is triggered. Remember, BPMN does not have a way to say how long something *usually* takes, but it does let you say what happens if it takes too long to complete.

What happens then depends on whether the event is interrupting or non-interrupting. An interrupting Timer event aborts the activity, and the process continues immediately on the exception flow. A non-interrupting Timer event immediately triggers a parallel thread of execution on the exception flow without aborting the activity or the normal flow out of it.

For example (Figure 7-6), you could use an interrupting Timer event in a hiring process to indicate that if a search for internal candidates does not complete within two weeks, you want to abandon it and

engage an external search firm. Note that because the exception flow and normal flow are exclusive alternatives, they can be merged at *Screen resumes* without a gateway.

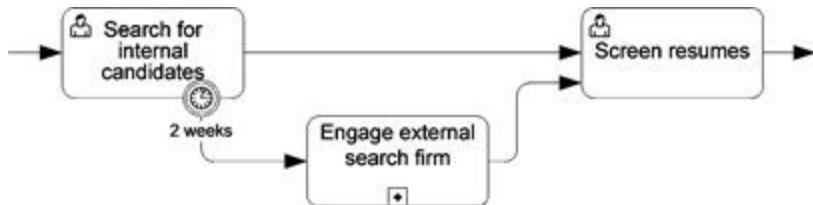


Figure 7-6. Interrupting Timer boundary event

The non-interrupting Timer event is generally more useful than the interrupting variety. If something takes too long, you usually want to keep doing it but do something else in addition, such as notify the requester, notify the manager, or get additional help.

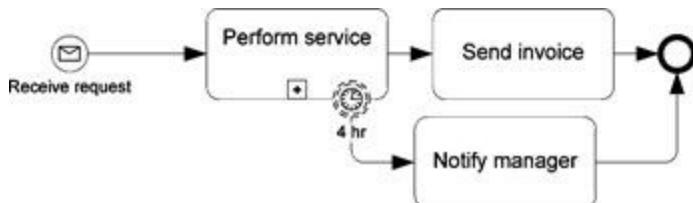


Figure 7-7. Non-interrupting Timer boundary event

For example (Figure 7-7), if it takes more than 4 hours to complete a service request, you want to notify the manager but keep performing the service. The exception flow is triggered at 4 hours after *Perform service* starts, but it does not terminate *Perform service*. That activity continues on, and when it is complete, the process continues to *Send invoice*. With non-interrupting events, the normal flow and exception flow are performed in parallel, or logically in parallel, since *Notify manager* is probably finished before *Send invoice* starts. Here we join the normal and exception flow paths directly to the end event, since a gateway is not used to join into a None end event.

Since it does not abort the activity it is attached to, a non-interrupting Timer event could be triggered multiple times. For example, you could send a reminder or notification on the exception flow every hour until the activity is complete. In that case, label the event *Every hour*.

Timed Interval

A Timer boundary event measures the time from start to completion of a *single activity*, but what if you want to time the interval from point A to point B in the process, spanning *multiple activities*? That's easy. Just wrap the fragment from point A to point B in a *subprocess*, and attach the Timer event to the subprocess boundary.

Figure 7-8 illustrates a fast food process: Take the order, collect the money, in parallel prepare the burger, fries, and drink, and when all those are complete deliver to the customer.

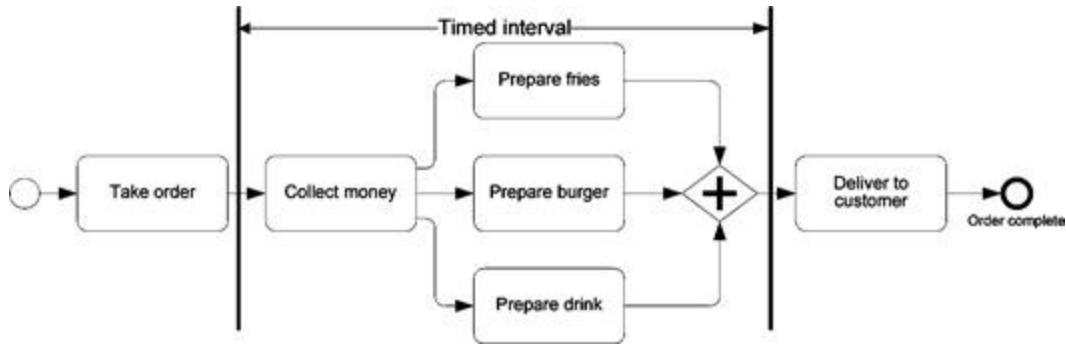


Figure 7-8. To time an interval spanning multiple activities...

Now we'd like to say if the order isn't ready to deliver to the customer within 5 minutes of taking the order, the restaurant will refund the money. In other words we want to time an interval spanning multiple activities, as shown in Figure 7-8. We can do that by enclosing that interval in a subprocess, and attaching a non-interrupting Timer event to it (Figure 7-9). In the inline expansion representation, it looks like this:

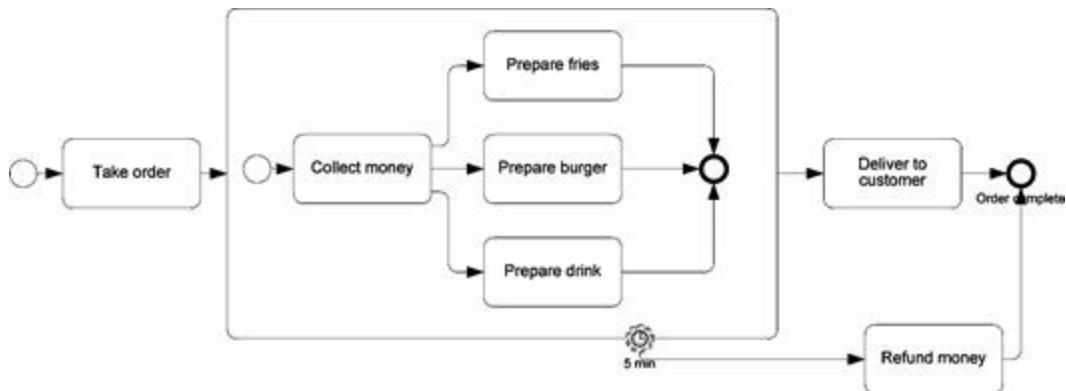
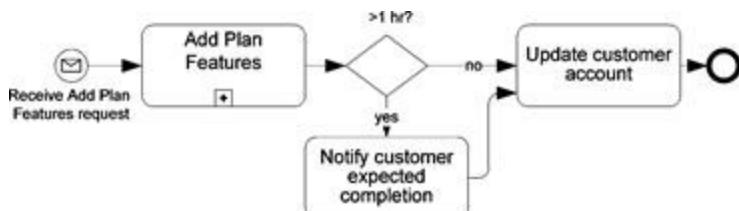


Figure 7-9. ...wrap the interval in a subprocess and attach Timer boundary event.

Timer Event vs. Gateway

Beginners sometimes try to test the duration of an activity using a gateway following the activity. That is usually incorrect, because the process does not arrive at the gateway until *after* the activity finishes, and by then the triggered action is too late. The whole point of a boundary event is that its action occurs *immediately* upon the timeout, *before* the activity it is attached to is complete. Here is an illustration.

Consider the two diagrams in Figure 7-10, both intended to represent a wireless carrier's *Add Plan Features* process. Adding the features is supposed to take no longer than one hour. If it takes longer, we want to notify the customer with the expected completion time. The question is this: Does the customer notification occur at the same time in both diagrams?



*****ebook converter DEMO Watermarks*****

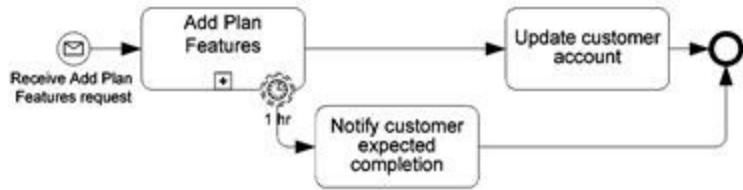


Figure 7-10. Does customer notification occur at same time in both diagrams?

No. In the diagram on the top, with the gateway, the customer is not notified until *after* the plan features are added, no matter how long that takes. That is probably not what the modeler intended. In the diagram on the bottom, the customer is notified after exactly 1 hour if the activity is not yet complete. Even though the Timer event here does not interrupt the task, its action is immediate. It does not wait for the activity to complete. That's the value of a Timer boundary event.

Message Event

Before diving into the details of Message events, we need to say more about what BPMN means by “sending” and “receiving.”

Message and Message Flow

The terms *send* and *receive* should be considered “keywords” in BPMN, reserved specifically for sending and receiving a *message*, represented in the diagram by a *message flow*. In BPMN a “message” means any communication between the process and an outside entity – a customer or service provider, another internal process, or possibly even an IT system. The BPMN 2.0 specification defines a message as simply “the content of a communication between two participants.” That communication could take any form. It does not have to be a SOAP or JMS message, as it might typically be in an executable process. In most process models it is more likely some form of human communications, such as a letter, fax, email or phone call. The only requirement is that the sender and receiver of the message are different “participants,” meaning not part of the same process.

In fact it is even possible that a BPMN message represents a *material flow*, delivery of some physical object. The BPMN metamodel specifies the message content or “payload” through its *item definition*. Early drafts of BPMN 2.0 used the term *data definition*, but it was changed to item definition to allow messages (and data objects, as well) to represent both physical and information objects.

What distinguishes a *Message* from another form of BPMN inter-process communication called *Signal* is that a Message must be addressed to a particular process, or possibly a particular instance of that process, whereas a Signal may be broadcast to *any* process that might be listening. (We’ll talk more about Signal later in this chapter.) Because a message flow identifies a particular process activity or event that sends or receives the message, it is possible that a single message is represented by more than one message flow in the diagram, each representing receipt of the message at a different point in the flow.

Send Task and Throwing Message Event

The term *send* in BPMN implies a message, and thus a message flow. A message may be sent from a black-box pool, a throwing Message event, or any type of activity.

Recall that a message flow out of an activity such as a User task signifies the *possibility* of sending the message, not the *certainty* of it. We’d like to have a way to say that a step in the process *always* sends the message, and BPMN Level 2 provides that, in two different ways. One is a *Send task*, denoted by a black envelope icon (Figure 7-11, left). A Send task is a task that does only one thing, sends a message. By convention, the send is *immediate* upon arrival of the incoming sequence flow, after which the flow immediately continues. In that sense it is implicitly automatic, although it could be used for human communications as well if the intent is to show the certainty, not the possibility, of sending the message. (Alternatively, you could attach a text annotation to a message flow out of a User task to specify that the message is always sent or only sent under certain conditions.)



Figure 7-11. Send task and throwing Message intermediate event

Alternatively, a *throwing Message intermediate event* (Figure 7-11, right) does the same thing. Effectively it is the same as a Send task. When the incoming sequence flow arrives, it sends a message and then immediately continues. You might ask why BPMN has two different elements that do exactly the same thing. Good question. I don't know.

Actually, there is a tiny difference between a throwing Message event and a Send task. As a type of activity, a Send task has a *performer*; an event does not. Also, you can attach a marker to a Send task to signify that it is performed multiple times, i.e., sends multiple messages; you cannot do that with an event. And you could attach an Error boundary event to a Send task, which you cannot do for a Message event. But for all practical purposes they are identical.

When a process is initiated by a Message start event, I like to show the return of final status in Message end events, a separate one for each end state. In IT terms, the start and final status messages in a sense define the "signature" or "interface" of the process. And I just like the symmetry of it in the diagram. For an executable process, the start and end event message flows effectively represent its WSDL.

However, in non-executable modeling, the fact that a Message event has no performer makes some modelers hesitant to use it to return final status when identifying the sender of the message is important. This often comes up in my BPMN training, and here is how I try to resolve it.

One way is by using *lanes*. Lanes usually identify the human performer of an activity as a role or organizational unit. Technically, an event has no performer, so you could argue lanes do not apply. But in BPMN 2.0, lanes can actually be used for *any* type of element categorization that you want. It's purely up to the modeler. So if you want to have an internal convention in your organization that says the lane of a Message end event identifies the sender, that is perfectly in accord with the BPMN spec.

A second way is more technical, and it applies mostly to email. Even if you say that lanes do not apply to Message events, that does not mean the identity of the sender is invisible to the recipient. It is part of the *content* of the message. In the standard email message structure, it's the *From* field. Nevertheless, some modelers have a hard time drawing a Message end event in Lane A when the sender is really someone in Lane B. That might come up, for example, where multiple process paths merge to a single end event in Lane A, because it represents a single end state, and the sender of the final status message could be in Lane B. To modelers who still cannot come to terms with that I advise sending final status from separate *tasks* in the respective senders' lanes, and afterward merge to a single None end event.

"Sending" Within a Process

A common beginner mistake is to use a Send task to forward work to a downstream task (Figure 7-12). Since the "sender" and "receiver" are part of the same process, you may not use a message. Thus a Send task is incorrect. In fact, since "send" is a sort of BPMN keyword, you should not even use the word "send" in the label of a User task!

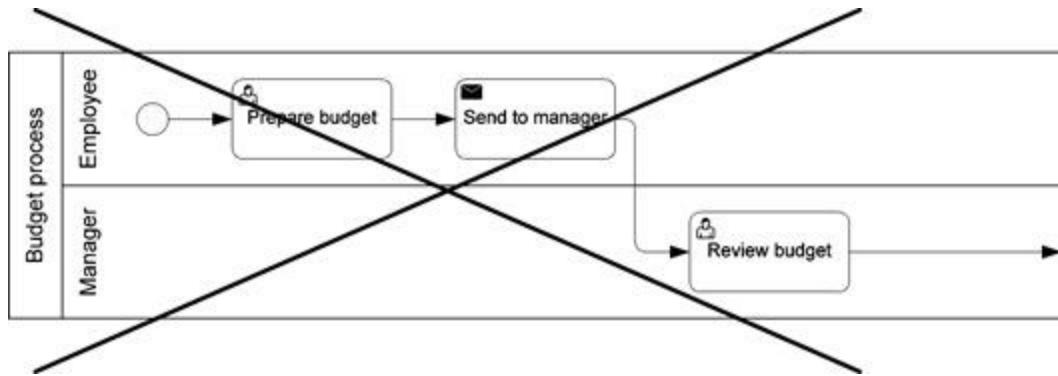


Figure 7-12. Don't use a Send task to communicate within a process

So how do you “send” work to a downstream task performer, or simply notify a Manager in another lane of the process?

In the case of forwarding work downstream, usually the best choice is not to model the “sending” action explicitly at all. It is simply *implied* by the sequence flow (Figure 7-13). In an automated workflow, the sequence flow delivers not only the work item notification to the downstream task performer, but all of the instance data available at that point in the process, including documents, forms, etc. Unless there is specific reason to call attention to the effort of sending, it is best to just imagine that similar delivery occurs somehow even in non-automated processes.

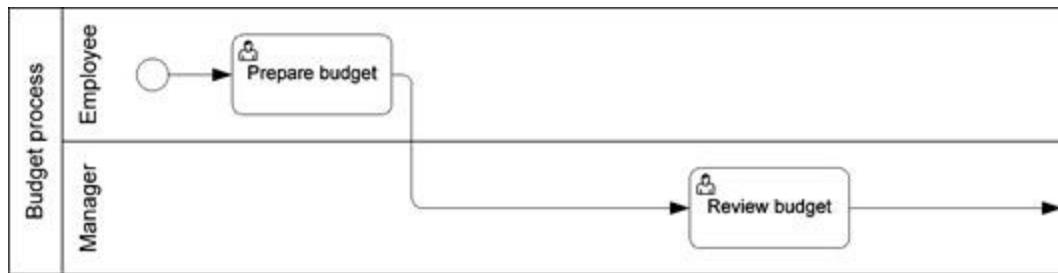


Figure 7-13. “Sending” work downstream is implied by sequence flow alone

It could be, however, that forwarding the budget materials to the Manager is not just attaching a spreadsheet to an email. Let's say it requires packing up two drawers of a file cabinet and carting it off to Fedex. And let's say that effort is exactly the kind of thing you want to improve upon in the to-be process, so you don't want to hide it. In such a case, you should make it a task, but it is a User task not a Send task (Figure 7-14). Since there is no BPMN message involved, don't even use the verb “Send” in the label. Instead use names like *Forward...* or *Pack and Ship....* And if you want to call attention to the materials being shipped, you can use a data object.

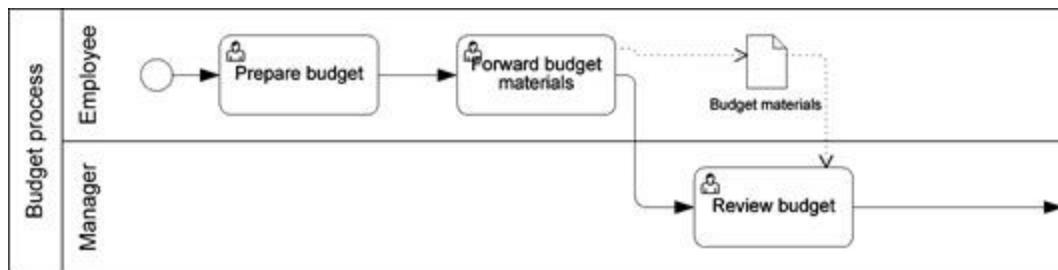


Figure 7-14. User task can represent the work of “sending” within a process

Notifications, at least those where no action on the recipient's part are required to advance the process, are slightly different. Here you don't want to add a task to the recipient lane, since that implies a required action on the part of the recipient. Instead just add a User task in the sender's lane, identifying the recipient in the task label (Figure 7-15).

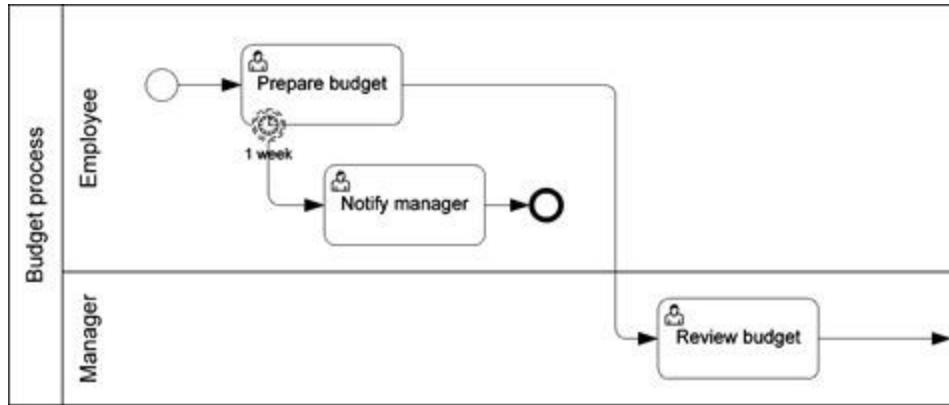


Figure 7-15. Notification within a process

Receive Task and Catching Message Event

Receiving is closely related to sending. Again, the term technically applies only to messages, communications from external participants. We have already seen that a Message start event creates a new process instance when the message is received. We can also receive a message in the middle of a process, but as we discussed with sending, a message flow into a User task only suggests the *possibility* of incoming message, not the *certainty* of it.

BPMN provides a task type that *only* receives a message, called *Receive task*, with the white envelope icon (Figure 7-16, left). A Receive task *waits* for a message. That is the only thing it does. When the incoming sequence flow arrives, the process instance pauses; when the message arrives, the process immediately resumes on the outgoing sequence flow.



Figure 7-16. Receive task and catching Message intermediate event

Technically, a Receive task immediately following a None start event in a top-level process may be designated as *instantiating*, meaning the message creates the process instance. However, to show a request-triggered process it is better to use a Message start event instead, since the None start plus instantiating Receive task construction is indistinguishable in the diagram from a manual start followed by waiting for the message. Two different meanings for the same diagram construct violates the basic Method and Style principle, so you should avoid instantiating Receive tasks.

A *catching Message intermediate event* (Figure 7-16, right), drawn with sequence flow in and out, has the same meaning as a Receive task. It waits for a message, and immediately resumes when the message is received. As with sending, again there is a tiny difference between a Receive task and a catching Message event. You can attach a Timer boundary event to a Receive task, but you cannot do that with a

Message event. As it turns out, there is another way to accomplish the same thing, and we'll see that shortly.

Asynchronous and Synchronous Messaging

Send and Receive tasks, or throwing and catching Message events, represent *asynchronous communications*. As soon as the process sends the message, the flow continues on the outgoing sequence flow. It does not wait for a response message. *Synchronous communications*, on the other hand, means when the process sends a message it waits for a response before continuing.

A *Service task* is an example of synchronous communications. Recall that a Service task represents an automated action. In the BPMN 2.0 metamodel, the Service task actually means an *automated request* for an action performed by some external system, with receipt of that system's response. The request and response are really *messages*, but usually we do not represent them as message flows in the diagram. They are simply implied. The Service task is not complete until it receives the response from the system that performs the action. That is what synchronous means.

In an executable process, synchronous tasks are *short-running*, completing in milliseconds or seconds. If an automated task is *long-running*, meaning it takes minutes, hours, or even weeks to complete, it is modeled in BPMN as an asynchronous request, using a Send task or throwing Message event, not a Service task. While this distinction is important for executable processes, it is a good convention to apply to non-executable BPMN as well: If an automated function is long-running, represent it with separate Send and Receive tasks (with message flows). Reserve Service task for short-running actions (Figure 7-17).



Figure 7-17. Use Send and Receive tasks for long-running services, Service task for short-running

Event Gateway

Figure 7-18 illustrates use of throwing and catching Message intermediate events in a process for issuing a credit card. If the Customer's application is missing required information, the process sends a request for it and waits for the response. Whenever you use a Message event you should draw the message flow, and label both the event and the message flow. The event should be labeled with the action – *Request X*, for example – and the message flow should be labeled with the name of the message.

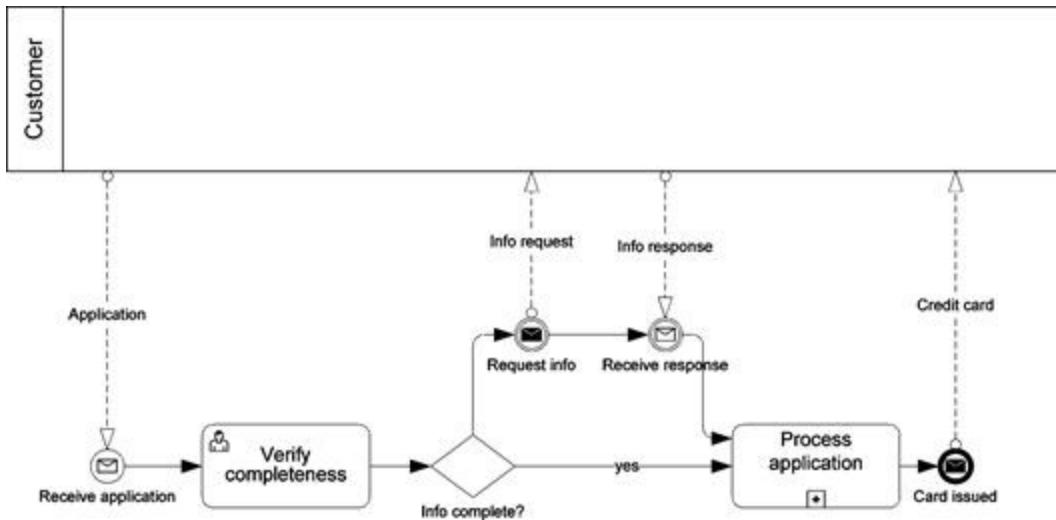


Figure 7-18. Throwing and catching Message intermediate events

When there is a possibility that the response may not be returned before some deadline, you should not wait for it using a “naked” Message event as in Figure 7-18. If the customer decides not to respond to the request, the process instance will wait forever at the catching Message event. Real processes don’t work that way. They will wait up to some maximum time, and then do something else. You can model that behavior with a Timer boundary event on a Receive task, but there is a way to do the same thing with a catching Message event.

Figure 7-19 is a better way to wait for the response message. It’s called an *event gateway*. The symbol inside the gateway shape is the *Multiple intermediate event*, and on each gate there is a catching intermediate event, usually a Message event and a Timer event. (You may also attach a gate to a Receive task with no boundary events, but better to just use a Message event.)

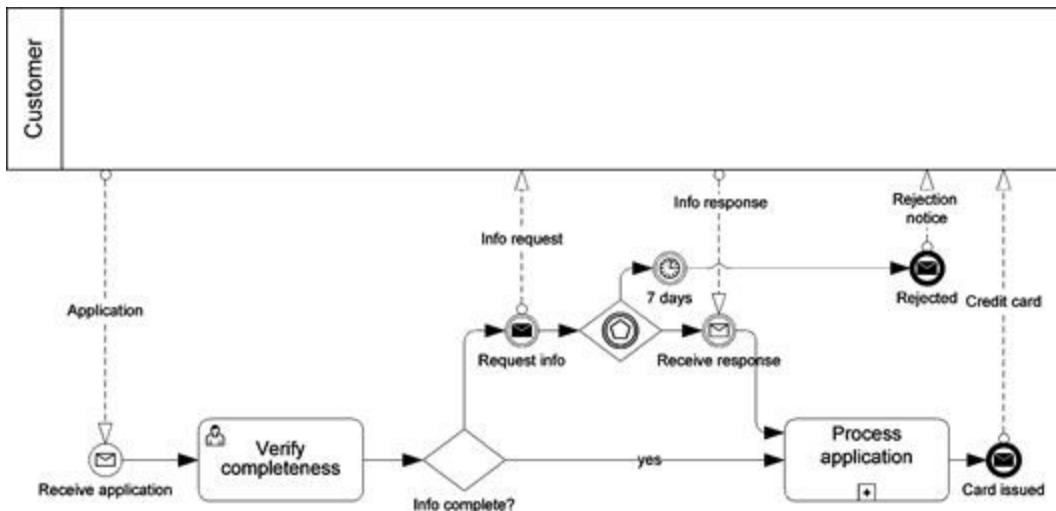


Figure 7-19. Event gateway waits for response or timeout, whichever occurs first

Like the regular XOR gateway, an event gateway represents an exclusive choice – i.e., only one of the gates is enabled – but the choice is not based on a process data condition. The gate that is enabled is the event that occurs *first*. An event gateway may have two or more gates, each representing an event, and it’s a race between them. In Figure 7-19 it’s a race between the response message and a timeout. If the *Info response* message is received within 7 days, the Message event gate is enabled and the instance

continues to *Process application*. If it is not received in 7 days, the Timer event gate is enabled and the instance continues to the *Rejected* end state.

In a BPMN tool, you usually must construct the event gateway in pieces – the gateway element itself and the event on each gate – but it's best to think of the whole assembly as the event gateway.

You can also use an event gateway to wait for *alternative messages*. For example, if you model *Approval* and *Rejection* as separate messages, you can receive them on separate gates of an event gateway (Figure 7-20). However, you could just as well model *Approval* and *Rejection* as simply different content of a single *Response* message. In that case, you can test the content value in an XOR gateway *after* receiving the message (Figure 7-21).

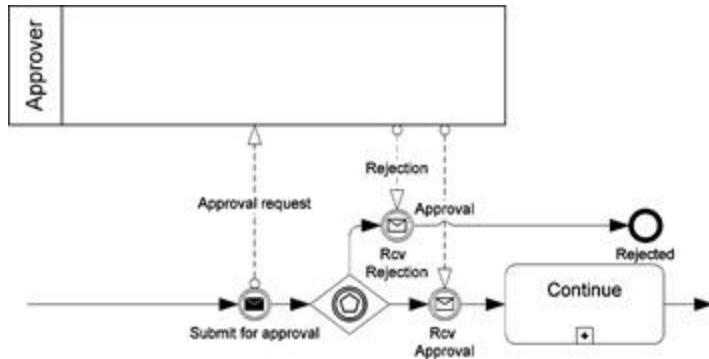


Figure 7-20. Branching on distinct messages with event gateway

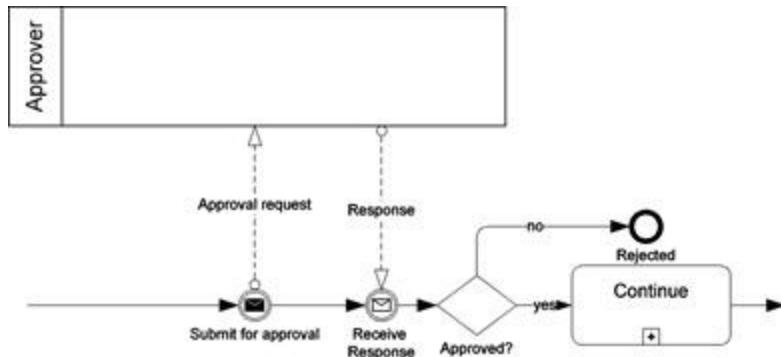


Figure 7-21. Branching on received message content with XOR gateway

The distinction between Figure 7-20 and Figure 7-21 is mostly notational, not significant in business terms. (In fact, both message flows in Figure 7-20 could technically point to the same message element in the underlying XML.) If instead of coming from a black-box pool, *Approval* and *Rejection* messages came from separate end events of another process, you would need to use Figure 7-20.

Message Boundary Event

A message you are waiting for usually implies a response to a prior request. But BPMN provides a way to respond to *unsolicited messages* as well. In that case, the process is not paused waiting for the message, but listening for it while running. A *Message boundary event* attached to an activity initiates the response to the message if it arrives while the activity is running. An *interrupting boundary event* aborts the activity immediately and exits on the *exception flow*, the sequence flow out of the event. A *non-*

interrupting boundary event continues the activity but immediately initiates a parallel action on the exception flow. If the activity completes without the message arriving, the exception flow is not triggered. The process simply continues on the *normal flow*, the sequence flow out of the activity.



Figure 7-22. Interrupting (left) and non-interrupting (right) Message boundary events

For example, if the customer cancels an order while it is being fulfilled (Figure 7-22, left), an interrupting Message boundary event immediately terminates *Fulfill Order* and exits on the exception flow. On the other hand, if the customer updates shipping information while the order is being fulfilled (Figure 7-22, right), you do not want to terminate *Fulfill Order* but initiate something else in addition, such as adding the updated shipping information to the order. The exception-triggered action is on the exception flow. When *Fulfill Order* completes, processing continues on the normal flow. With non-interrupting events, the normal flow and exception flow exits represent parallel paths.

The same physical message may be represented in the process model by more than one Message boundary event, each representing a different triggered behavior, depending on the state of the process when the message arrives. How cancellation is handled immediately after the order is placed may not be the same as when it is ready to ship.

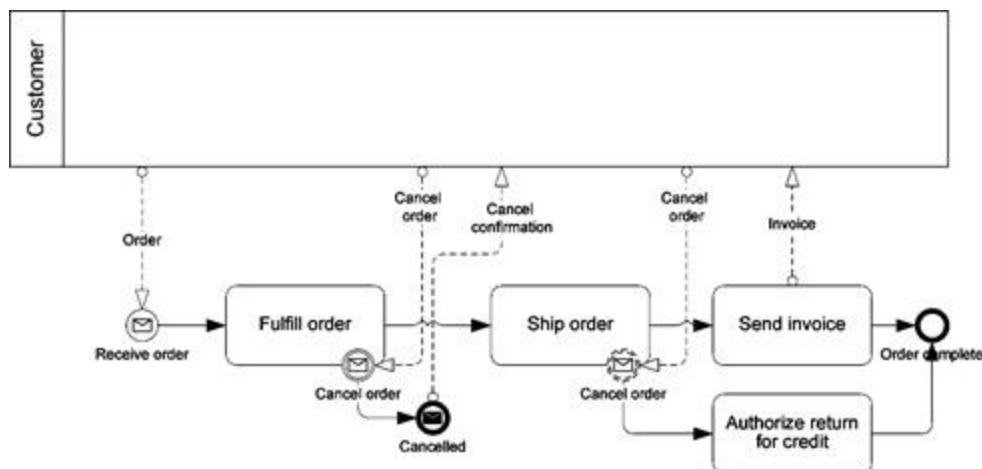


Figure 7-23. The same message may be received in multiple boundary events

For example, in Figure 7-23 we see that the message *Cancel order* aborts the order process and returns a *Cancel confirmation* message if it is received before *Ship order* starts. However, if the same message is received during *Ship order*, the process in this case cannot be terminated. A new action, *Authorize return for credit* is triggered, but *Ship order* continues, and when it finishes the process goes to *Send invoice*. Both the exception flow and normal flow are enabled in this case.

A couple matters of BPMN style are worth noticing in Figure 7-23. You should always draw the incoming message flow to a Message boundary event and label both the event and the message flow. Since both boundary events handle the same physical message, usually it's best to give them the same name (and give the message flows the same name as well). It is not required that one boundary event is interrupting and the other non-interrupting; they both could equally well be interrupting or both non-

interrupting, as the semantics of the behavior dictate.

If the response to the message is the same for every activity within a contiguous segment of the process, you should not attach a Message boundary to each of those activities and merge the exception flows. The correct way to model it is to *enclose that segment in a subprocess* and attach a single Message event to the subprocess boundary (Figure 7-24).

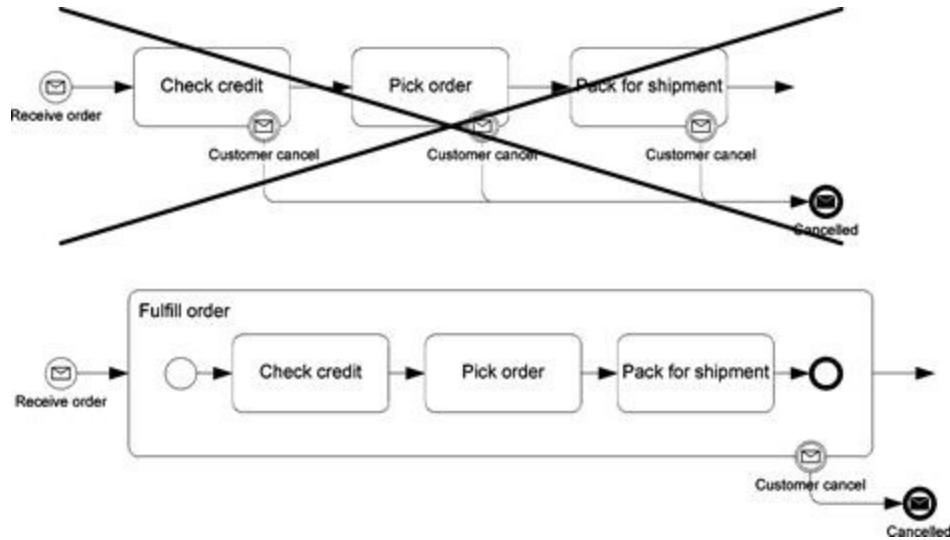


Figure 7-24. Use a single Message boundary event on a subprocess enclosing a process fragment when the event-triggered action is the same for all child-level activities

Error Event

The last of the Big 3 event types is the *Error event*, representing an *exception end state* of a process activity. Error events only come in two flavors: an *interrupting Error boundary event* and an *Error end event*. You cannot throw or wait for an Error signal in an intermediate event, and there is no Error start event (except in an event subprocess, which we will discuss later).

An Error event on the boundary of a task simply represents the exception end state exit from the task. The normal flow, the sequence flow out of the task, represents the exit when the task completes successfully, and the exception flow, the sequence flow out of the Error event, is the exit when it does not. Its meaning is exactly the same as an XOR gateway following the task with a success gate and an exception gate (Figure 7-25).



Figure 7-25. An Error boundary event on a task is equivalent to testing the task end state with a gateway

In the first edition of the book, I advocated reserving Error events for *technical exceptions*, and using the gateway end state test we saw in Level 1 to handle *business exceptions*. However, in the time since that edition was published, feedback from students and others has caused me to revise my opinion. Now I say it is perfectly fine to use Error events for any type of exception, business or technical. There is no implied semantic distinction between testing the end state in a gateway and using an Error event, although you could make such a distinction as a convention for your organization.

You can have more than one Error event on the boundary, representing distinct exception end states, although if the exception flows all take the same path it is best to consolidate all the exceptions in a single Error event.

What if the task *Check credit* in Figure 7-25 were a *subprocess* instead? As before, the boundary event *Bad credit* signifies that the activity has an exception end state *Bad credit*. But, unlike a task, a subprocess exposes its end states explicitly. So a *Bad credit* Error boundary event on a subprocess implies the child-level expansion must contain an end state *Bad credit*. This is not just Method and Style; the BPMN spec here agrees. Not only must there be a *Bad credit* end event in the child-level expansion, but it must be an *Error end event*.

An Error end event in a subprocess throws an error signal to the boundary of the subprocess, where it is caught by the Error boundary event and exits on the exception flow. This is called the *Error throw-catch pattern*. You could think of it as *propagating an exception from child to parent levels* in a hierarchical BPMN model.

This is illustrated in Figure 7-26. The *Bad credit* error is thrown from a child-level end event to a boundary event at the parent level. In the BPMN metamodel, both the Error end event and Error boundary event reference the same *error code*, but since the error code does not appear in the diagram, we apply the usual Method and Style principle and say that the *labels* of the error thrower and catcher must match. Following some exception handling at the child level (*Update customer info*), the error *****ebook converter DEMO Watermarks*****

throw-catch propagates the exception to the parent level for additional exception handling at that level (*Contact customer* and end the process).

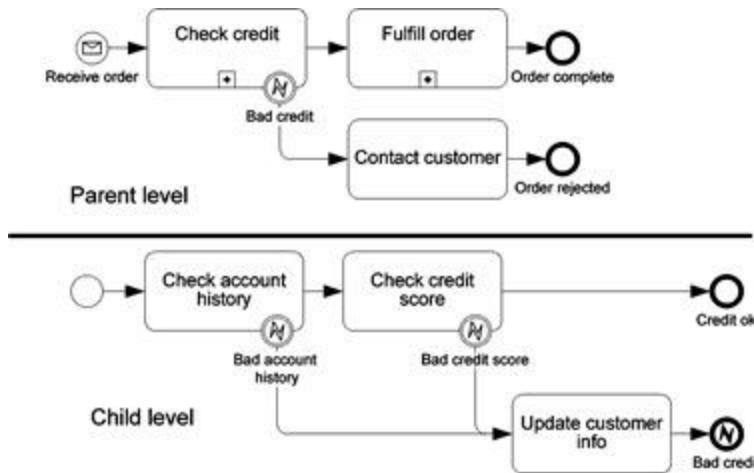


Figure 7-26. Error throw-catch

When you use them to model business exceptions, Error events are really just a notational convenience, since we can describe the same behavior with gateways. Figure 7-27, the gateway end state test from Level 1, means exactly the same thing as Figure 7-26, using Error throw-catch. The gateway end state test also propagates exceptions from child level to parent level.

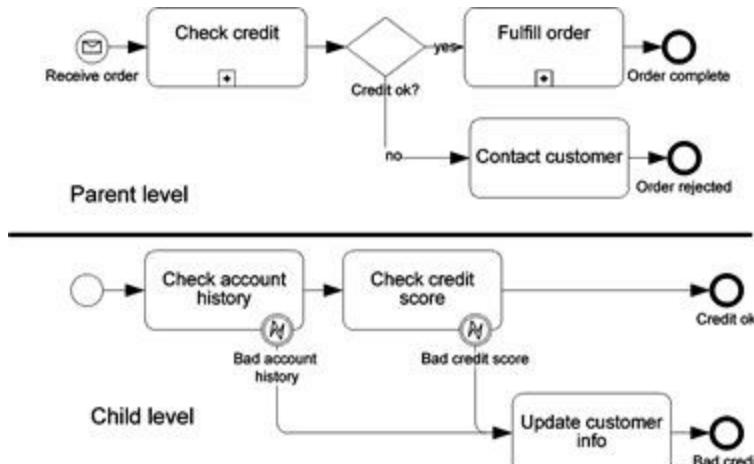


Figure 7-27. Gateway end state test

In the examples presented so far, the error is thrown when the child level is already complete, so the “interrupting” Error boundary event doesn’t really interrupt anything – the subprocess is already over. But it is possible that the child-level expansion has parallel paths reaching separate end events. If one of them is an Error end event, then throwing the error terminates the subprocess even if the other path has not yet reached its end event. With parallel paths, Error throw-catch acts like the gateway end state test where the exception end state is a Terminate, not a None end event.

Other Level 2 Events

The Analytic subclass of BPMN 2.0, what we have been calling the Level 2 palette, contains a few more event types. Chances are you will never need to use them, but I will describe them briefly here.

Escalation Event

Escalation is another one of those terms that has a specific meaning in BPMN that is not the same as its general meaning in English, or even in business process management. In BPMN, Escalation is the non-interrupting counterpart of Error, with similar throw-catch behavior. An Escalation boundary event simply signifies a *non-interrupting exception inside an activity*. That activity could be either a task or subprocess.

A valuable use case for an *Escalation boundary event on a User task* is *ad-hoc user action*. That means while the performer is in the *middle* of the task, the performer *may possibly* initiate another parallel path of action. Since BPMN does not describe the internals of a task, the performer's logic is invisible to the model, so triggering the non-interrupting exception flow is effectively ad-hoc. For example (Figure 7-28), if a technical configuration issue comes up during order entry, the salesperson may need to consult with a specialist before completing the task. On a task, the Escalation event does not imply that the exception flow *will* be triggered, only that it *may* be triggered.

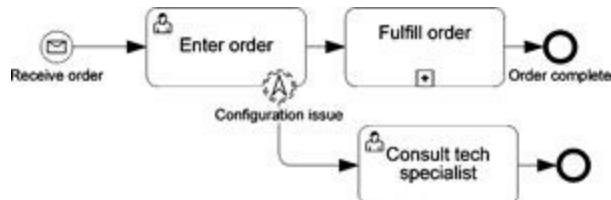


Figure 7-28. Ad-hoc user action with Escalation event

The Escalation-triggered exception flow runs in parallel with the original activity and possibly with activities on the normal flow exit. Figure 7-28 only "works" if *Enter order* always waits for *Consult tech specialist* to complete before continuing to *Fulfill order*. But the diagram itself does not ensure that. Technically, as modeled in Figure 7-28, *Fulfill order* could start before the technical consultation is complete. If you want to say that can never happen, it might be better to enclose *Enter order* and *Consult tech specialist* in a subprocess, as in Figure 7-29.

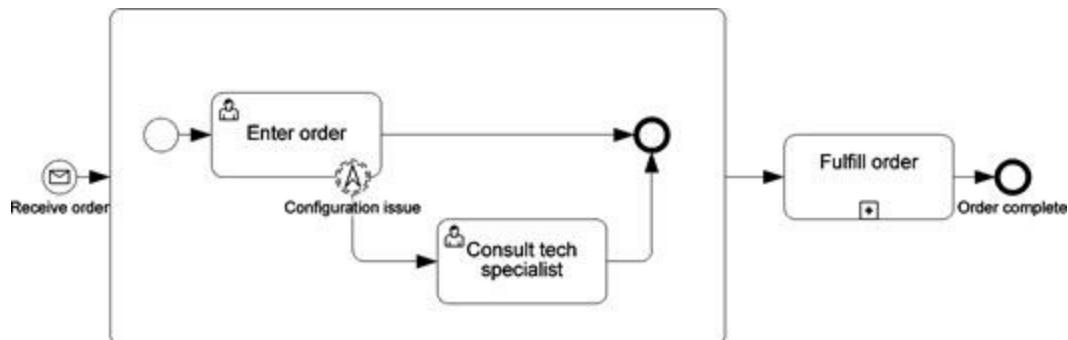


Figure 7-29. Joining non-interrupting exception flow and normal flow with a subprocess

It is important that the action be initiated from the *middle* of the User task, not at the end. A parallel thread of action initiated at *the end* of the task is better modeled as an OR gateway following the task, which we will see in Chapter 9. You might argue this is inconsistent with Error, where I say Error throw-catch and gateway end state test are equally acceptable. But few tools support Escalation events, which are new in BPMN 2.0, and even fewer modelers know what they mean. So it's best to save Escalation for where it is really needed.

In a subprocess, an Escalation end event in the child-level expansion can throw a signal caught by an Escalation event on the subprocess boundary in the parent-level diagram. (Such an *Escalation throw-catch* signal can also be thrown by an Escalation intermediate event, something you cannot do with Error.) Technically, an Escalation boundary event may be either interrupting or non-interrupting, but there is no semantic difference between an interrupting Escalation event and an Error event, so in the interrupting case I would just use Error.

As with Error, an Escalation event on the boundary of a collapsed subprocess implies a matching throwing Escalation event in the child-level expansion. Unlike Error, however, the normal flow and exception flow exits from the activity are not alternative paths but parallel.

Signal Event

Message, Error, and Escalation events all constrain the relationship between thrower and catcher. Error and Escalation can only throw to the boundary of the parent subprocess; Message can only throw to another pool. One motivation for Signal events, added in BPMN 1.1, was to provide throw-catch signaling without those constraints, in particular, the inability to communicate using a Message with an activity on a parallel path of the process.

But Signal was also given a second, completely unrelated, property. The signal itself would be *broadcast* rather than targeted at a particular process or process instance, as Message is. Broadcasting the signal rather than addressing it to a particular process has the advantage of *loosely coupling* the thrower and catchers. Such behavior, known as *publish-subscribe integration*, allows a process or system to announce an event, such as the addition of a new customer, without having to know about all the processes that might be triggered by that event. *Any* process listening for that particular event could trigger a new instance using a *Signal start event* (Figure 7-30).



Figure 7-30. Signal start event generally signifies publish-subscribe integration

The Signal event properties needed for intra-process signaling and publish-subscribe integration actually work against each other, since communication with a parallel node in the process requires addressing a particular process instance, not just broadcasting a signal. But the BPMN spec conveniently ignores this problem. When used to communicate within a process, we assume that the Signal provides the necessary details to target the right instance. When the catcher is a Signal start event, usually the broadcast (also known as *publish-subscribe*) behavior is assumed.

Signal can be thrown from either a throwing intermediate event or end event, and may be caught in a start event, catching intermediate event (including event gateway), or boundary event. This flexibility

*****ebook converter DEMO Watermarks*****

is one of its key benefits. We have seen previously how Terminate or Error end events can be used to end a parallel path within the process level. But those patterns immediately end the entire process level. Signal throw-catch to an interrupting boundary event on a parallel path does not have this limitation. In Figure 7-31, if contract negotiations fail, we can stop *Develop specs* using Signal throw-catch without immediately terminating the process..

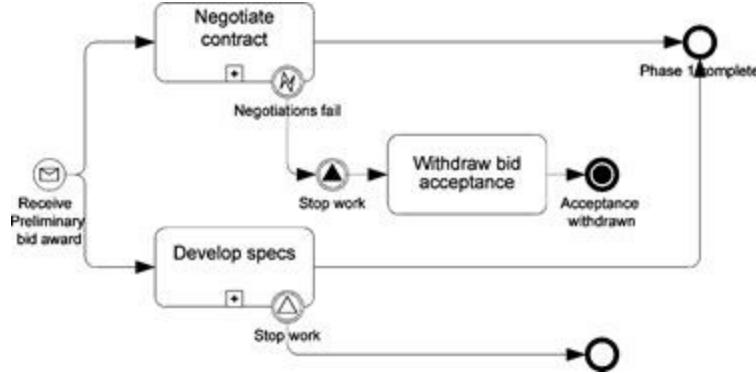


Figure 7-31. Signal throw-catch across parallel paths is more flexible than simple Terminate

It is incorrect to attach a message flow to a Signal event. The link between the Signal thrower and catcher is suggested only by matching labels. In fact, many times only one half of a Signal throw-catch pair is part of the model at all. For that reason, Signal throw-catch should only be used when Message, Error, or Escalation throw-catch cannot be used.

Conditional Event

The Conditional event signifies a continuously monitored data condition. When the condition, defined by a data expression, becomes true, the event is triggered. For example, a Conditional start event can trigger a stock replenishment process when inventory becomes low (Figure 7-32, left).

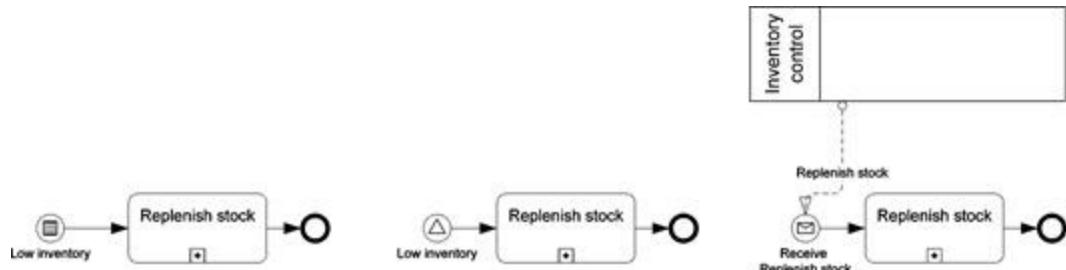


Figure 7-32. Conditional event (left) signifies a monitored data condition. Often Signal or Message events can describe the same behavior.

In Figure 7-32, all three diagrams effectively do the same thing, with only slight differences in interpretation. With Conditional start (left), the process model defines the monitored data condition. With Signal or Message start, detection of low inventory is a function of an external system or database. Signal start indicates a broadcast *Low inventory* signal, while Message start indicates a specific request sent by the external system.

Conditional may be used with catching intermediate and boundary events, as well as start events. The label of the Conditional event should indicate the monitored condition.

Link Event

The Link event is more of a drawing aid than a true event. It does not really throw or catch a trigger signal. Link only supports throwing and catching intermediate events, not start, end, or boundary events. A *Link throw-catch pair* is simply a *visual shortcut for a sequence flow between the throwing Link event and the catching Link event*. It may only be used where BPMN allows a sequence flow, so the Link throw-catch may not cross a subprocess or pool boundary.

One use for Link events is as an *off-page connector*, where a single process level does not fit on a single page. It may not be used between pages representing parent and child process levels; both pages must represent the same process level. Thus Link event pairs are seen more often in flat (single-level) BPMN models than in hierarchical models.

A second use is as an *on-page connector*, simply to reduce the clutter of crossing sequence flows. It is used, for example, in tools like IBM Blueworks Live that draw the sequence flow layout automatically. Tools supporting manual layout, like Microsoft Visio, let you precisely arrange sequence flows to minimize crossing, but auto-layout tools are less good at this.

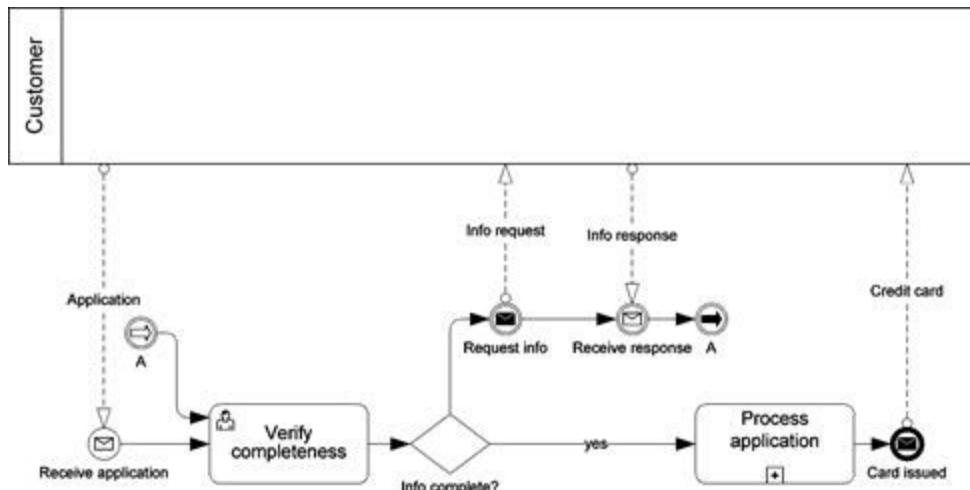


Figure 7-33. Link event pair used as on-page connector

In Figure 7-33, the Link events labeled A mean that the sequence flow out of the catching Link event is really an extension of the sequence flow into the throwing Link event. Whether visually connecting Link throw-catch pairs in this way is clearer than crossing sequence flows is a matter of opinion.

Event Subprocess

In BPMN Level 2, you can think of the actions on the exception flow of a boundary event as the *handler* of that event. BPMN 2.0 introduced a second type of event handler called an *event subprocess*. Event subprocesses are more easily mapped to BPEL than boundary event handlers and, unlike boundary event handlers, are able to access the *context* (i.e., data and state values) of the process level in which the event occurs. These are both really developer issues related to executable BPMN, and thus beyond the scope of BPMN Level 2. However, event subprocesses are useful constructs for non-executable modeling as well, if you care to use them.

An event subprocess is defined within a particular process level, either the top-level process or a regular subprocess. It works similar to a boundary event. If the trigger occurs while the process level containing the event subprocess is running, the event subprocess is started. Unlike a regular subprocess, an event subprocess has no incoming or outgoing sequence flows. Instead, it has a *triggered start event*, such as a Message, Timer, or Error. Since the trigger is only active while the process level is running, it acts more like an intermediate event than a regular start event. An event subprocess may be either *interrupting* or *non-interrupting*, as indicated by the start event border: solid for interrupting, dashed for non-interrupting, just like a boundary event.

In the notation, an event subprocess is visually distinct from a regular subprocess. A collapsed event subprocess is denoted by a *dotted line boundary* and a *trigger icon* in the top left corner (Figure 7-34, top). Expansion of the event subprocess may be either inline, inside an enlarged rounded rectangle, or hierarchical, on a separate diagram (Figure 7-34, bottom).

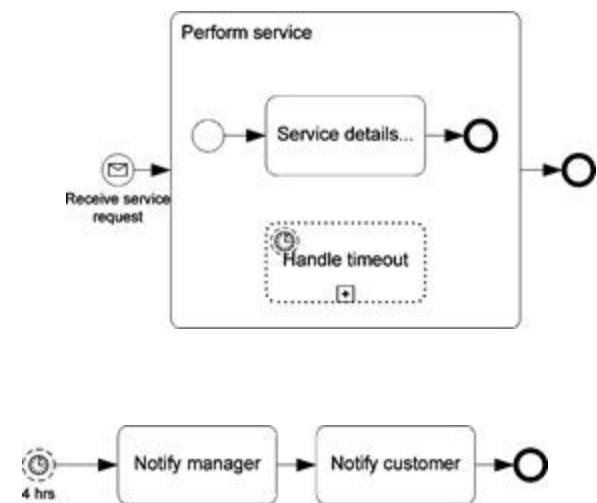


Figure 7-34. Event subprocess *Handle timeout* defined for regular subprocess *Perform service* (top); child-level expansion of *Handle timeout* (bottom)

In Figure 7-34, *Handle timeout* is an event subprocess defined inside *Perform service*, a regular subprocess. It is triggered by a non-interrupting timer. The semantics are similar to a boundary event. If *Perform service* takes longer than 4 hours, do not interrupt it but trigger the event subprocess in parallel. When a non-interrupting event subprocess is triggered, the containing process level is not complete until both the regular child-level process (*Service details...*) and the event subprocess have

completed. Processing then resumes on the normal flow out of the regular subprocess.

An interrupting event subprocess works in a similar manner, except that the regular subprocess is aborted when the event subprocess is triggered. When the event subprocess completes, the process resumes on the normal flow out of the regular subprocess.

There is one exception to process continuation on the normal flow out of the regular subprocess. An Error or Escalation end event of the event subprocess may throw an exception to a matching boundary event on the regular subprocess (not on the event subprocess). In that case, processing continues on the exception flow out of the boundary event. (If a non-interrupting Escalation is thrown, processing continues in parallel on the normal and exception flows.) This behavior is not clearly explained in the specification, but I checked it out with other members of the BPMN 2.0 technical committee, and this is their consensus opinion.

Chapter 8

Iteration and Instance Alignment

BPMN provides a way to say that an activity, either task or subprocess, is not complete until it is performed multiple times. In fact, it has two different ways to say that. In some cases, even such repeating activities are inadequate to properly align an activity instance with the process instance. We'll discuss iteration in all its aspects in this chapter.

Loop Activity

A *loop activity*, indicated by a circular arrow marker at bottom center (Figure 8-1, left), is like Do-While in programming. It means the same thing as the explicit gateway-loopback diagram on the right: perform the activity once, and then evaluate the *loop condition*, a Boolean data expression. If the condition is *true*, perform the activity a second time, and then evaluate the loop condition again. This iteration can continue indefinitely, or you can establish an upper limit. When the loop condition is *false*, the sequence flow out of the loop activity is enabled.

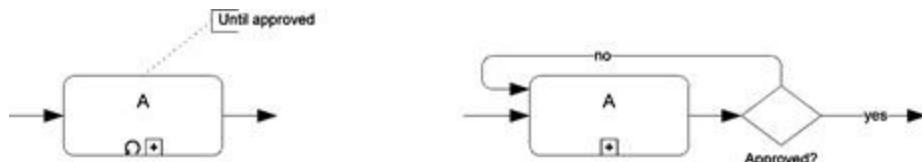


Figure 8-1. Loop activity A (left) means the same as non-loop activity A with gateway-loopback

Don't use the loop marker and gateway-loopback together. That's a loop within a loop, probably not what you mean. The loop marker provides a more compact representation than the gateway-loopback, but it hides the loop condition. For that reason, it is best to indicate the condition in a *text annotation*. Note: A condition of the form *Until X* corresponds to the loop condition *if Not X*; when X is true, *Not X* is false and the looping ends.

With loop activities, the iteration is always sequential. You can't start the second iteration until you have finished the first, and the loop condition is true. Also, with loop, the *number of iterations is unknown* when the first iteration starts. It is determined by evaluating the loop condition at the end of each iteration.

Multi-Instance Activity

A *multi-instance (MI) activity*, denoted by a marker of 3 parallel bars at the bottom center, is like *For-Each* in programming. It means perform the activity once for each item in a list. In a single process instance there are multiple instances of the activity, and each activity instance acts on one item in the list. What list? A multi-instance activity only makes sense when the process instance data contains some kind of collection, such as items in an order. In an order process, MI activity *Check stock* means check the stock of each order item.

Each order does not have the same number of items, but when *Check stock* begins for a particular order, you already know how many iterations will be required. It's the number of items in the order. Often the list involved is obvious from the activity name, but if not, best to indicate it in a text annotation, such as *For each X*. Knowing the number of iterations in advance is one fundamental difference between multi-instance and loop activities. Another is that MI instances may be performed in parallel. If so, the MI marker is 3 vertical bars. If, on the other hand, the instances are always performed sequentially, the marker is 3 horizontal bars. A sequential MI activity is *not* the same as a loop.

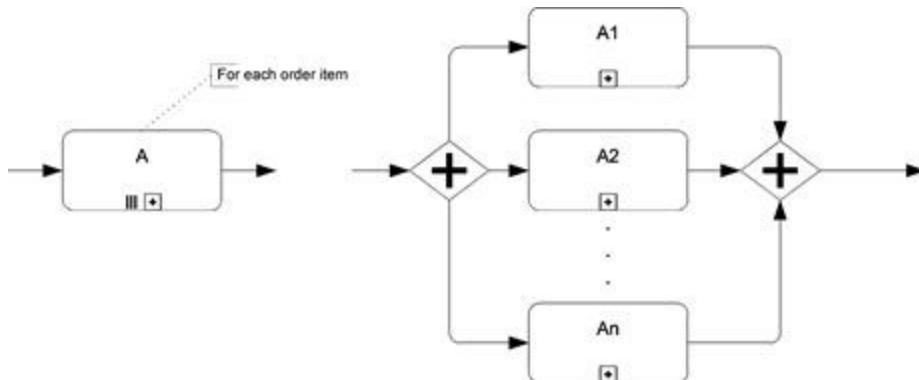


Figure 8-2. MI activity A (left) is the same as n parallel instances of non-MI activity A followed by a join (right).

In Figure 8-2, MI activity A (left) means the same thing as n parallel non-MI activities (right). The MI activity is not complete until all its instances are complete. Technically, other completion conditions are allowed by the spec, but I have never seen them in the wild and they are indistinguishable in the diagram from the usual *all-complete* condition. BPMN 2.0 actually allows you to say that a Signal event is generated, either as each instance completes or just when the first instance completes, and caught on the boundary of the MI activity.

So perhaps it is just a Method and Style convention, but I think best to interpret MI as requiring *all* instances to complete before the MI activity is complete. This is almost always the modeler's intent. A Terminate or interrupting boundary event on a multi-instance activity will immediately abort all running instances.

Using Repeating Activities

In my BPMN training, one of the exercises formerly used for certification involved a hiring process. Each instance of the process is a job opening. The process starts with defining and approving the position, followed by posting the job, accepting applications, interviewing candidates, and ultimately hiring one of them. It's a familiar process to most students, but even so, I would see a disheartening number of certification submissions that looked like this:

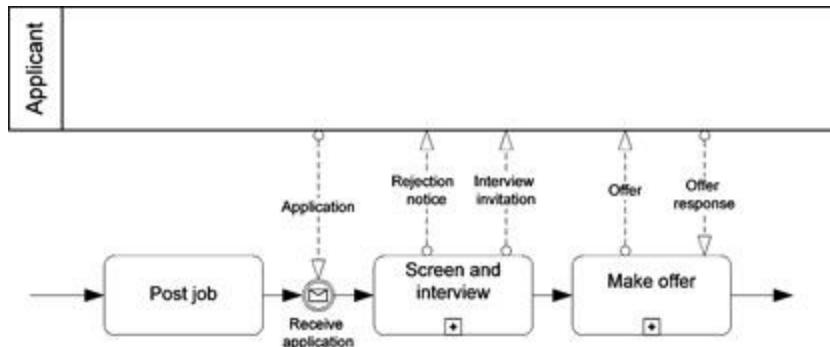


Figure 8-3. A common beginner mistake

What is the problem here? Take a look until you see it.

Each instance of the process represents a separate job opening. How many applications can each instance handle? In Figure 8-3, it's just one! After the first *Application* message advances the instance to *Screen and Interview*, there is nothing to accept the next one. We need some way to indicate that there are multiple incoming *Application* messages for each instance of our process. Maybe repeating activities can help.

How many applications will we receive? Well, we don't know. We don't have a list of them before we begin, so we must use a *loop activity*, not MI. The simplest thing would be to wrap the Message event and *Screen and Interview* in a loop subprocess with the loop condition *Until ready to make offer*:

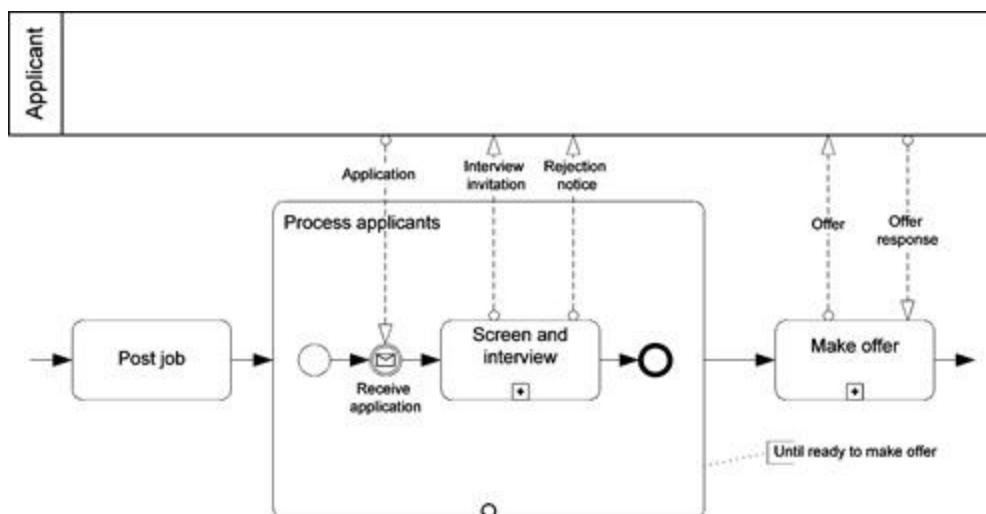


Figure 8-4. A valid but impractical solution

This technically works, but it has a serious practical problem. The second iteration cannot begin

until the first is complete. That means completing the interview process, which might take two or three weeks. If each iteration takes too long before we even look at the next applicant, this process is not going to work. If we are going to use a loop, it has to be relatively fast.

A more practical approach might be to have a fast *Receive and Screen* loop, followed by an MI *Interview* subprocess. *Receive and Screen* just sorts applicants into viable candidates – those who match the basic qualifications – and non-viable ones. Let's say the loop condition is *Until 5 viable candidates*. Then *Interview* can be MI because we have a list. We can conduct the interviews of all five candidates in parallel. When they are all done, we go on to *Make offer*. Now it looks like this:

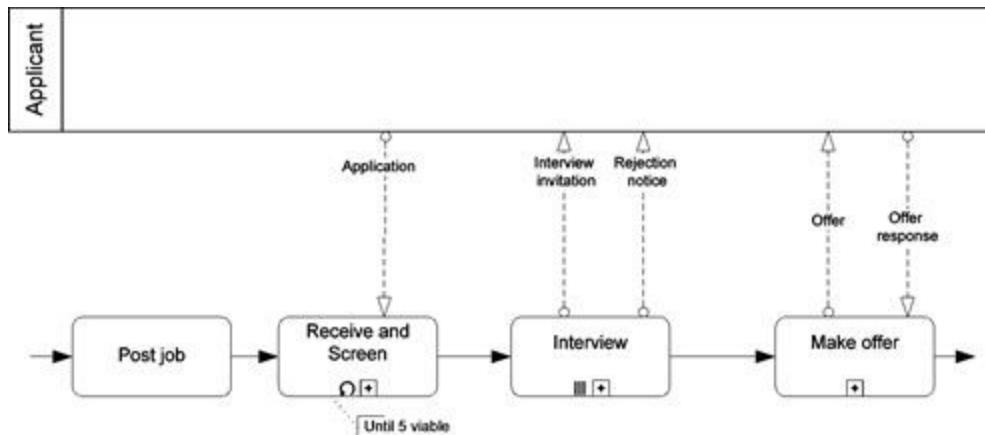


Figure 8-5. A more practical process model

Figure 8-5 is a practical solution to the hiring process problem, but it too is imperfect. The basic problem is that *Receive and Screen* and *Interview* cannot overlap in time. We cannot start any interviews until we have all five viable candidates, and once we begin the interviews we cannot look at any more applicants. This may be the way your process actually works, but most people say, no, we'd like to begin interviews as soon as we have one viable candidate, and keep looking at new applicants after we've started interviewing.

You cannot do that with repeating activities. In fact, you cannot do that in one BPMN process. You need more than one.

Using Multiple Pools

Method and Style generally recommends you model an end-to-end business process as a single BPMN process, if you can. But sometimes you cannot do that, and the reason is that activity instances are not aligned across the whole end-to-end process. There is no 1:1 correspondence between them. In that case, you may need to model the end-to-end process as multiple pools, that is, multiple BPMN processes. Our hiring process scenario provides a good example.

Recall that a Message start event has that magical ability to create a new process instance whenever the start message arrives. We don't need to know how many start messages will arrive. The Message start event creates a new instance for each one. And there is no rule that one instance must complete before the next one starts. The instances may overlap in time in any manner. This combines the best parts of loop and multi-instance activities, without their constraints.

This suggests the alternative solution to the hiring process problem shown in Figure 8-6. Its principal feature is the hiring has been split into two pools instead of the normal one. The reason for two pools is not because the "participants" are different. In fact, the activity performers – the hiring department and HR – are *exactly the same people in both*. They are separate pools because they are *independent BPMN processes*. And the reason they are separate processes is that their respective instances do not have 1:1 correspondence.

In *Hiring Process*, the instance corresponds to a single job opening, just as we had in the repeating activity structure. In *Evaluate Candidate*, the instance is a single applicant. A new instance of *Evaluate Candidate* is created whenever the *Resume* start message is received. These instances can overlap in time in any fashion, and we don't need to know how many there are.

Note that *Evaluate Candidate* has what looks like a multi-instance marker at bottom center, indicating a *multi-instance participant*. This marker only has significance in a collaboration between pools. It signifies that in the collaboration diagram there are multiple instances of this pool with respect to each instance of the other pool. In Figure 8-6 that means multiple instances of *Evaluate Candidate* for each instance of *Hiring Process*, i.e., multiple applicants for each job opening.

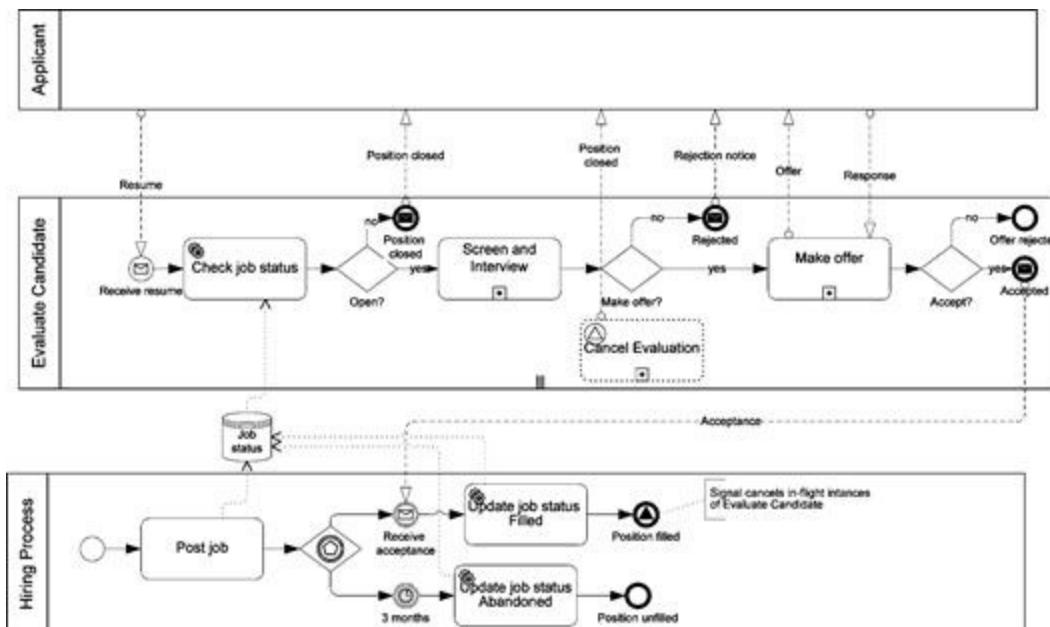


Figure 8-6. Multi-pool solution to the hiring process problem

In the multi-pool structure, after posting the job, *Hiring Process* just waits for a message from *Evaluate Candidate* indicating that an applicant has been selected and has accepted the offer.

We can put a timeout on waiting for that message by using an event gateway. Each instance of *Evaluate Candidate* could take weeks from beginning to end, but unlike the simple loop in Figure 8-4, this works because instances of *Evaluate Candidate* can overlap in time.

The multi-pool solution avoids the limitations of repeating activities, but it is harder for many people to understand. Also, you must deal with the problem of *coordinating the state* of the two pools. Remember, in this structure these are independent processes. While *Hiring Process* and *Evaluate Candidate* are technically peers, *Hiring Process* is effectively the parent. It needs to enable *Evaluate Candidate* when the job is posted and disable it when the job is filled.

Figure 8-6 illustrates two ways of synchronizing the state. One uses a data store representing the job status in a database. *Hiring Process* updates the data store when the job is opened, filled, or abandoned, and *Evaluate Candidate* queries it immediately upon instantiation. Once the job is filled, new applicants just receive a *Position Closed* message. But when the job is filled we also need to terminate any running instances of *Evaluate Candidate*. For that we throw a Signal event (possibly a Message event would work just as well). We cannot attach a boundary event to a top-level process, but we can use an interrupting event subprocess. Upon receiving the Signal, it terminates *Evaluate Candidate* and provides any cleanup actions, such as sending the *Position Closed* message.

Batch Processes

The multiple pool solution may seem obscure, but for end-to-end process modeling you may find yourself using it frequently. A common use case is where one part of the process operates on “batches” of items that are processed one at a time in another part of the process. For example, in the order process examples used in this book, the process instance is a single order, meaning that end-to-end processing is one order at a time. But in real order processes, there may be a mainframe batch program that runs one or more times a day to post all orders received since the previous batch. It is not really correct to insert an activity *Post order batch* in the middle of a process where the instance is a single order, since that suggests *Post order batch* is repeated for each individual order.

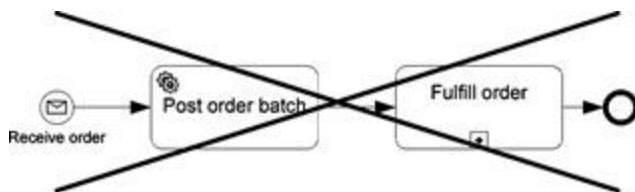


Figure 8-7. Instance mismatch between activity and process

Post order batch is better represented as an independent top-level process, with a Timer start event signifying a scheduled process, that interacts with the order process. As we saw with the hiring process example, there are two ways to model the interaction, data store (Figure 8-8) and Message (or Signal) events (Figure 8-9).

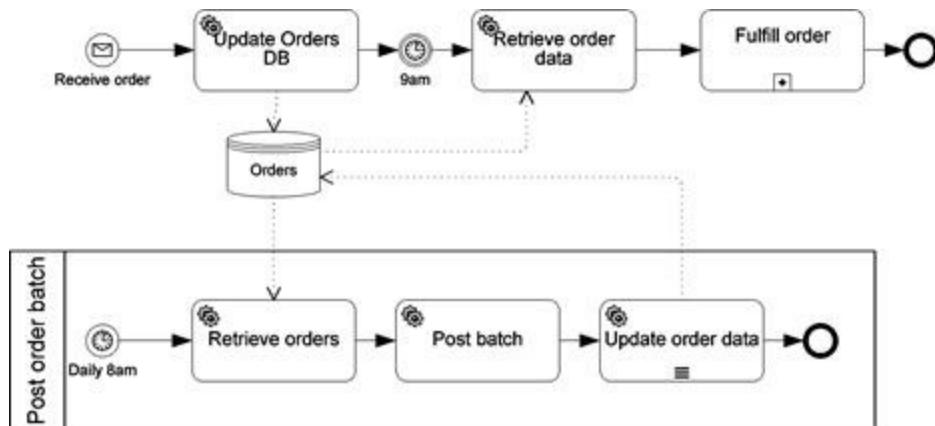


Figure 8-8. Two pools interacting via data store

In Figure 8-8, the *Order* process updates the *Orders* database with each order as it is received. Once a day, the *Post order batch* process retrieves all the new orders, runs the batch, and updates the *Orders* database with the posting data. The *Order* process waits until the batch posting is scheduled to be complete, retrieves the posting data for that order, and continues.

In Figure 8-9, collection of daily orders is the same, but the posting info is returned to the *Order* process in a message. The process waits for the message, and continues as soon as the message arrives. With either the structure of Figure 8-8 or Figure 8-9, it is not necessary to show the process logic of *Post order batch* if your objective is modeling the *Order* process. You could model it as a black-box pool. The key thing is you cannot model the batch posting as an activity inside the *Order* process.

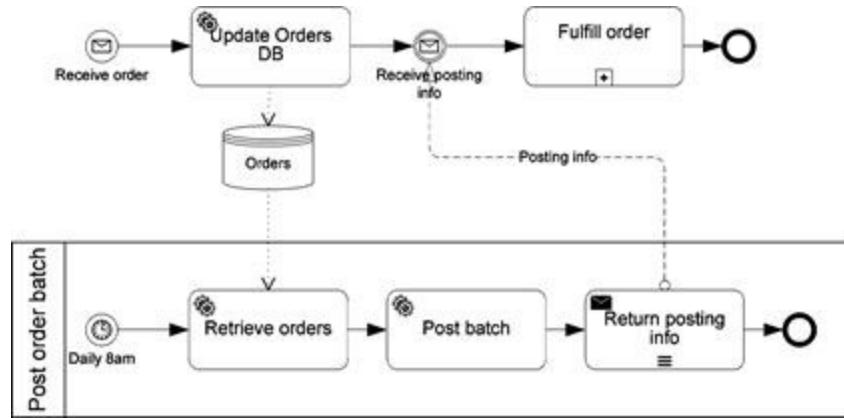


Figure 8-9. Two pools interacting via data store and message

Instance Alignment

The multi-pool structure works in other cases besides mainframe batch programs. For example, in many of the examples in this book, an invoice is sent to the customer with each order. But for regular customers it is not uncommon to send a bill every month, not with every order. In that case, you cannot make *Send monthly statement* an activity in the *Order* process. It must be part of a separate *Billing* process that runs every month (Figure 8-10).

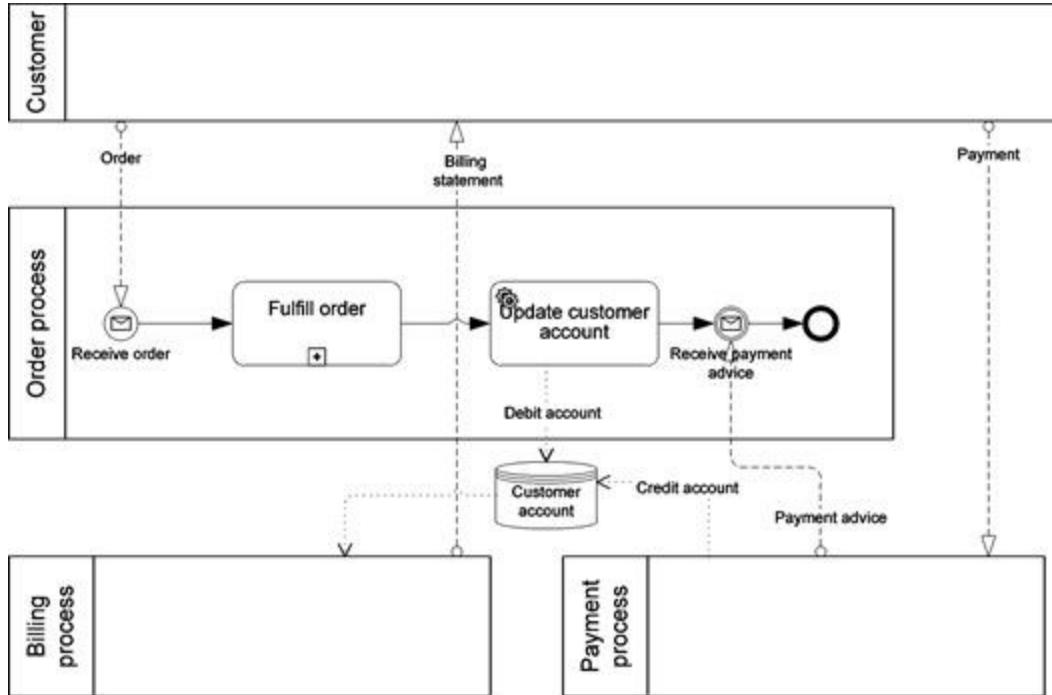


Figure 8-10. Billing and Payment are separate pools because the instance is not an order

Similarly, customer payments are not once per order or even necessarily once per month. An instance of the *Payments* process is once per payment. Thus if the *Order* process does not end until the order is paid for, multiple interacting pools are required.

Chapter 9

Process Splitting and Merging

We have already covered in detail the most common splitting and merging behaviors in BPMN:

- Exclusive split based on a data condition, using the XOR gateway
- Exclusive split based on the first event to occur, using the event gateway
- Unconditional parallel split, using either the AND gateway or multiple sequence flows out of an activity or start event
- Merge of exclusive alternatives by direct connection (no gateway)
- Join of parallel paths, using AND gateway

In this chapter we will cover a few additional splitting and merging behaviors.

Conditionally Parallel Flow

The parallel (AND) gateway represents an *unconditional split*, meaning in every instance the process splits into two or more parallel paths, one for each gate. But what if you want to say that the parallel split is *conditional*, meaning each path may or may not be enabled for a particular process instance? It does not happen often, but BPMN has a way to say it... in fact, two different ways!

OR Gateway Split

The *inclusive gateway*, also called the *OR gateway*, with the O symbol inside, represents *conditional split*. Like the exclusive (XOR) gateway, each gate has a Boolean condition, but here the conditions are *independent*. More than one of them could be true, and each gate with a true condition is enabled. If two or more are enabled, those paths run in parallel.

In Figure 9-1, after *Draft contract* we always *Conduct financial review* but only *Conduct financial review* if it is a technical contract. If we do both, the financial review and technical review occur in parallel. An OR gateway split requires a condition on each of its gates. If the gate is always enabled, use the label *Always*.

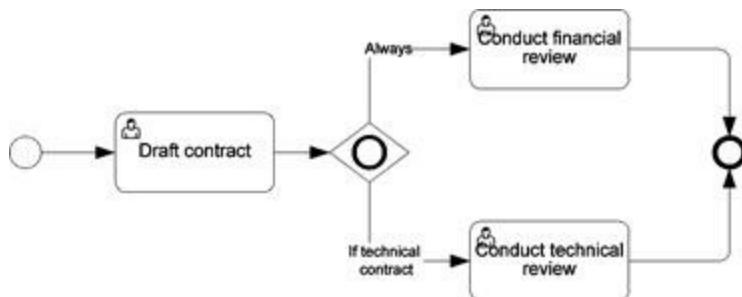


Figure 9-1. Conditional split using OR gateway

Figure 9-2 is slightly different. Now we only *Conduct financial review* if the cost is over \$10,000, and we only *Conduct technical review* if it is a technical contract. If we do both, they occur in parallel. The gate with the tickmark is called the *default flow*. Default flow in BPMN does not mean always or even usually; it means *otherwise*. The default flow is enabled if and only if no other gate is enabled for the process instance. In this example, *Conduct quick review* only occurs if the cost is *not* over \$10,000 and it is *not* a technical contract. A gateway may have at most one default flow.

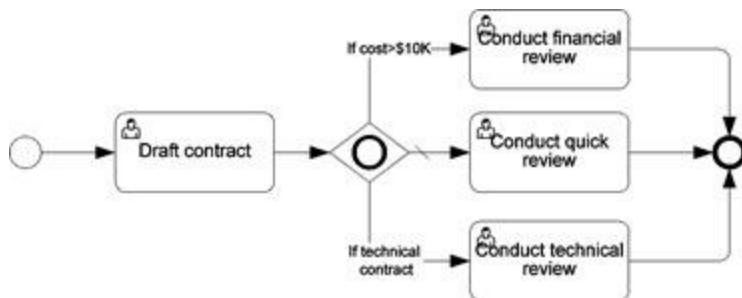


Figure 9-2. Default flow means *otherwise*

Conditional Sequence Flow

Figure 9-3 illustrates a second way to show conditionally parallel flow, called *conditional sequence flow*. Here we have no gateway at all, but two of the sequence flows have a little diamond on the tail, indicating enablement only if its condition is true. This diamond-on-the-tail notation is only allowed for sequence flows out of an activity. In the XML, sequence flows out of an XOR gateway or OR gateway are also conditional, but in the notation there are no diamonds on the tail for sequence flows out of a gateway. (I have seen BPMN tools that put the diamond on the tail of a sequence flow out of a gateway, but this is incorrect.) The default flow, indicated by the tickmark, means the same thing as it does out of a gateway – *otherwise*. The default flow is enabled only if none of the other outgoing sequence flows are enabled.

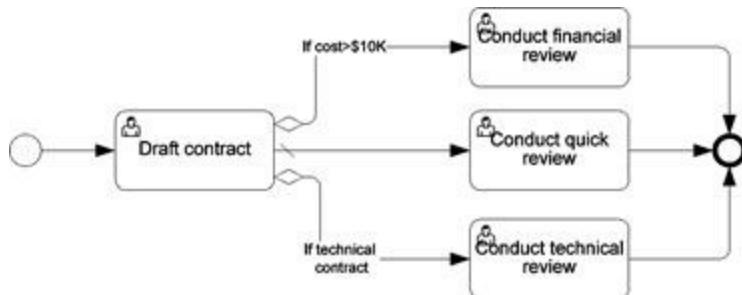


Figure 9-3. Conditional sequence flow

It is best to reserve conditional sequence flows for conditionally parallel flow. If you mean exclusive choice, use an XOR gateway instead. In Figure 9-4, the left diagram implies *Approved* and *Rejected* could both be true, which is incorrect. The middle diagram is technically correct, but it only works with two outgoing sequence flows. If there are two conditional sequence flows plus a default flow, does the modeler mean exclusive choice? In my experience, usually the answer is yes... which is incorrect. To eliminate ambiguity, use XOR gateway (Figure 9-4, right) when you mean exclusive choice.

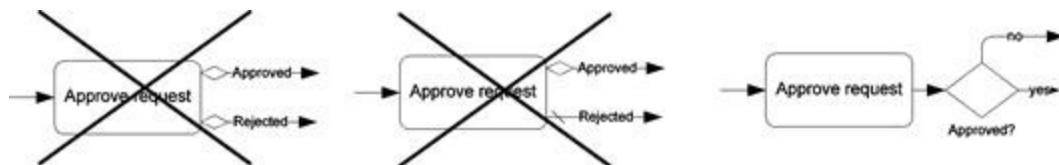


Figure 9-4. Don't use conditional sequence flow when you mean exclusive choice

Merging Sequence Flows

Proper modeling of the merge of multiple sequence flows into one depends on two factors: 1) whether the flows are exclusive alternatives, unconditionally parallel, or conditionally parallel, and 2) the intended merging behavior.

Merging Alternative Paths

If the paths to be merged represent *exclusive alternatives*, just merge them directly (Figure 9-5, left). In order to tell if they are exclusive alternatives, you need to look upstream to see how they were split in the first place. If they were split by an XOR gateway, event gateway, or an interrupting boundary event, they are exclusive alternatives.

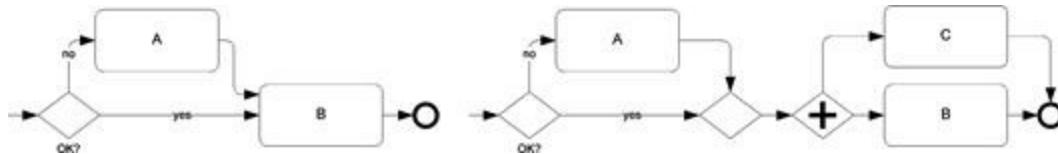


Figure 9-5. Merge alternative paths directly into an activity. You may use XOR gateway to merge into another gateway.

An XOR gateway used as a merge is the same as no gateway at all. It simply passes through each incoming sequence flow as it arrives. For merging alternative paths into an activity it is completely redundant, so best to omit it. However, you may want to use it to merge alternative paths into another gateway (Figure 9-5, right), since the behavior of a gateway with multiple inputs and multiple outputs may be ambiguous from the diagram.

AND Gateway Join

If paths are *unconditionally parallel*, usually you want to *join* them (Figure 9-6, right). A *parallel join* is modeled as an AND gateway with multiple sequence flows in and one out. It waits for *all* incoming sequence flows to arrive before continuing. We don't need to use a gateway to join into a None end event, but a parallel join into an activity always requires one.

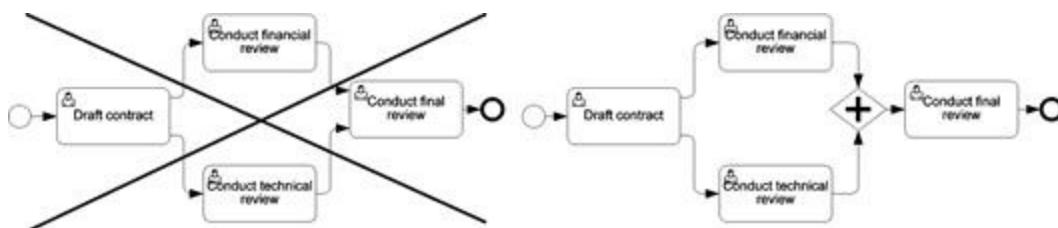


Figure 9-6. To join parallel paths into an activity, use AND gateway

Multi-Merge

Even though the parallel gateway is optional for a split, you should not omit the gateway for the join (Figure 9-6, left). The spec technically allows omitting the gateway – it's called *multi-merge* – but it means the activity following the merge (*Conduct final review*) is triggered multiple times, once for each incoming sequence flow, and the same goes for all downstream activities. Using an XOR gateway as a merge passes through each sequence flow as it arrives, the same as no gateway at all. *Thus with parallel inputs an XOR gateway also signifies multi-merge, not a join.* Multi-merge is almost never what you mean, and I recommend avoiding it.

OR Gateway Join

If some of the parallel paths to be joined are *conditional*, meaning not enabled in every process instance, you may still join them, but you use an *OR gateway*, not an *AND gateway*. An OR gateway join is like an AND gateway join except that it ignores incoming sequence flows that are not enabled for this process instance.

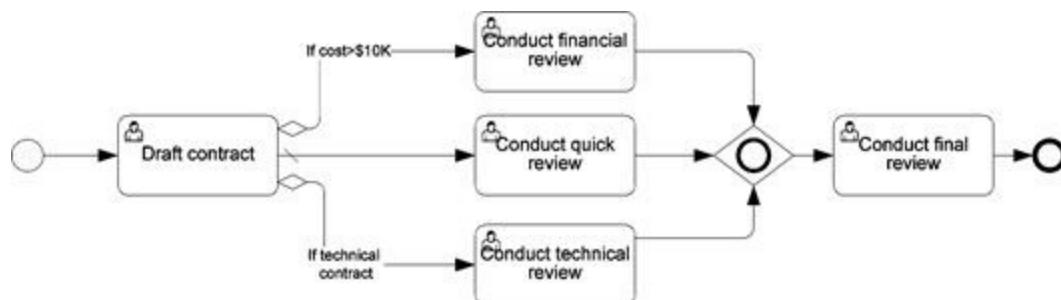


Figure 9-7. OR gateway join of conditional sequence flows

For example, in Figure 9-7 either one or two of the flows could be enabled in any process instance. The OR gateway join ignores the incoming sequence flows that are not enabled in this instance. Note we did not need the OR gateway in Figure 9-3 because a join is always implied into a None end event.

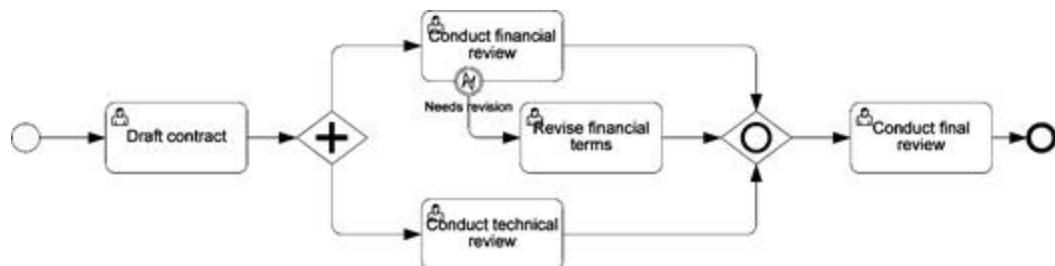


Figure 9-8. Another use case for OR gateway join

Figure 9-8 provides another example. Even though we have an unconditional parallel split, only two of the three sequence flows into the join can arrive in any process instance. An AND gateway join requires all three; the OR gateway join ignores the “dead path” in this process instance.

A third example is joining the exception flow path, in the case of a non-interrupting boundary event, with the normal flow path. The normal flow exit always occurs, but the exception flow occurs only if the event is triggered. Thus the flows are conditionally parallel and cannot be joined by an AND gateway; the gateway must be OR.

Discriminator Pattern

*****ebook converter DEMO Watermarks*****

There is one more merging behavior worth discussing. It is called the *Discriminator pattern*, and it uses the *complex gateway*, with the asterisk symbol inside the diamond. A complex gateway does not necessarily mean Discriminator. It means some user-defined behavior other than that described by AND, OR, or XOR gateways. There are very few of those, and the only one that occurs with any frequency at all is Discriminator.

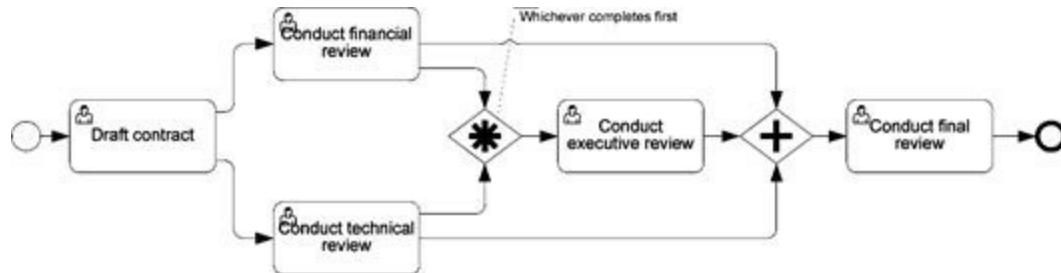


Figure 9-9. Discriminator pattern

The Discriminator pattern passes the *first* incoming sequence flow to arrive and blocks all the rest. When multiple activities are running in parallel, Discriminator lets you start something else when any one of them completes. For example (Figure 9-9), we can start the executive review with the output of either the financial or technical reviews, whichever comes first. That's Discriminator. If we had no gateway at all into *Conduct executive review*, that task would be triggered twice (multi-merge), not what we want.

Note the complex gateway requires a text annotation to explain the intended behavior. The complex gateway is *not* part of the Level 2 palette, as defined by the Analytic subclass of BPMN 2.0.

Chapter 10

Transactions

Although it goes beyond the scope of the Level 2 palette, BPMN provides native support for transactions. The term *transaction* refers to the coordinated execution of multiple activities such that they either *all* complete successfully or the system is restored to a state equivalent to *none* of them completing. An example familiar from everyday experience is electronic funds transfer in a bank. The transaction debits one account and credits another account, perhaps at another financial institution, an equal amount. This requires the coordinated action of two databases, possibly two independent systems. If, for some reason, the debit and the credit cannot both be executed simultaneously, neither of them should be executed. What absolutely must not happen is that one account is debited without the corresponding credit in the other account (or vice versa).

ACID Transactions

This example and similar distributed database operations are known in computer science as *ACID transactions*. Here ACID stands for:

- Atomic – indivisible, all-or-nothing behavior
- Consistent – preventing an inconsistent state of the system, such as a debit with no corresponding credit
- Isolated – the systems managing each account are locked during execution of the transaction
- Durable – the state of the participating systems is stored in a database, not just in memory, so it can be restored in case of a crash

In IT systems, ACID transactions are typically implemented using a special protocol called *two-phase commit*. In two-phase commit, a piece of software called a transaction manager first communicates with the various resources performing each side of the transaction, in this case, the debit and the credit, to ensure they are all ready to execute. Only if all resources report that they are ready is the transaction *committed*. Otherwise they are *rolled back* to the state before the transaction was initiated.

Business Transactions

BPMN implements a similar idea for business processes. In BPMN, a subprocess marked as *transactional* means that its component activities must either all complete successfully or the subprocess must be restored to its original consistent state. However, *business transactions* are usually not ACID transactions coordinated via two-phase commit. The reason is they fail the *I*, or isolation, requirement. In order to isolate, or lock, the resources performing the component activities of the transaction, the transaction must be *short-running*, taking milliseconds to complete. For business transactions you cannot usually make that assumption. Business transactions are *long-running*, and the resources associated with their component tasks are not locked while the transaction is in progress. Instead, each activity in the transaction executes normally in its turn, but if the transaction as a whole fails to complete successfully, each of its activities that has completed already is *undone* by executing its defined *compensating activity*.

Examples of transaction recovery by compensation are familiar from everyday experience. Suppose you purchase some item online via credit card, but it turns out later that the item is unavailable from the company's suppliers. You will see on your credit card statement both a charge for the item and a subsequent matching credit cancelling the charge. The credit is the compensating activity for the charge. This is not the same as an ACID transaction that reserves the item in inventory before it charges the credit card. In that case you would see neither the charge nor the credit on your statement, because the transaction was never committed in the first place.

BPMN provides built-in support for business transactions. A subprocess with a double border (Figure 10-1) denotes it as a *transaction*. Activities within the transaction that need to be undone if the transaction fails are linked with their respective compensating activities in the BPMN diagram by *Compensation boundary events*. And BPMN provides other events that signal transaction failure and initiation of compensation.

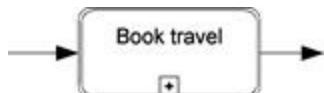


Figure 10-1. Transactional subprocess

Compensation does not include the handling of the exception that caused the transaction to fail. It just means restoring the original consistent state of the system before the transaction began, by undoing those parts of the transaction that completed before the point of failure. Once compensation is complete, exception handling continues in the normal manner.

Compensation Boundary Event and Compensating Activity

The *Compensation boundary event* is used to link an activity to its undoing, or compensating, activity. It is not a normal boundary event, however. It has no outgoing sequence flow. Instead it has an *association* linking it to a single *compensating activity* (Figure 10-2). Both the Compensation event and compensating activity are identified by the rewind symbol. The purpose of the Compensation boundary event is simply to link an activity with its associated compensating activity.

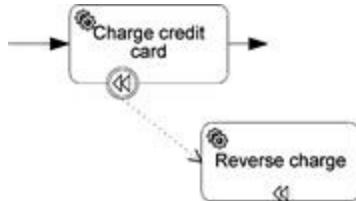


Figure 10-2. Compensation boundary event and compensating activity

Unlike a regular boundary event, a Compensation boundary event can only be triggered *after* the activity to which it is attached completes successfully. If the activity has not started or is still running when the transaction fails, or if the activity itself completes unsuccessfully, its compensating activity is not run when the transaction fails.

Alternatively, a *Compensation event subprocess* – an event subprocess with a Compensation start event – may be used as the compensating activity.

Cancel Event

The *Cancel event*, with the X icon, is a special form of Error event that may only be used with transactional subprocesses (Figure 10-3). It is used when the source of transaction failure is *within* the transaction subprocess, not after completion. Like Error, Cancel supports throw-catch from an end event of the transactional subprocess to a boundary event or event subprocess. Also, like Error it is always interrupting; there is no non-interrupting variant of Cancel. Its meaning is identical to Error except that before beginning the error handling, represented by exception flow or event subprocess, Cancel implicitly commands compensation.

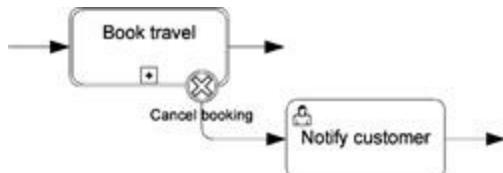


Figure 10-3. Cancel boundary event on transactional subprocess

When the transaction is Canceled, all successfully completed activities within it that have defined compensating activities are undone by executing those compensating activities. Once compensation is complete, error handling commences by executing the exception flow or event subprocess associated with the Cancel event.

Any other type of interrupting boundary event, such as Error, on a transactional subprocess aborts the transaction *without compensation*.

Compensation Throw-Catch

In addition to Cancel, BPMN provides an alternative way to directly command compensation by a throwing *Compensation intermediate event* or *Compensation end event*. Unlike Cancel, the throw-catch target is not a boundary event, but the *activity* to be compensated. This *Compensation throw-catch* does not require a transactional subprocess.

A use case for Compensation throw-catch is when the need to undo the transaction is determined *after* the transaction is complete. The following examples illustrate the use of Cancel and Compensation events.

Using Compensation

To properly define compensating activities you need to think about the various points, either within the transactional subprocess or after its conclusion, where the transaction could possibly fail, and which possibly completed activities would need to be undone if that occurs.

Consider a simple travel booking example in which the transaction consists of two activities, reserving the seat and charging a credit card, always performed in that order (Figure 10-4). The only time this transaction requires compensation is if the credit card charge fails. In that case, the airline reservation must be undone using a compensating activity. If the activity reserving the seat fails (e.g., no seats available) there is no successfully completed activity to undo. Even though a compensating activity is defined, it is not executed unless the original activity completes successfully. Also, no compensating activity need be defined for the charge, since if it completes successfully the transaction as a whole completes successfully.

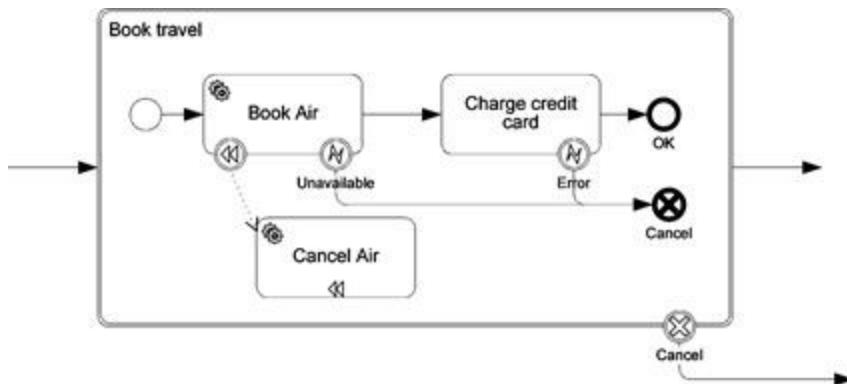


Figure 10-4. Transaction compensation, simple case

Now let's consider a more complex case, in which multiple flights and hotels must be booked to complete the itinerary (Figure 10-5). The order of booking each leg, hotel and air, is indeterminate. If any leg of the itinerary cannot be booked successfully, the transaction fails. If all legs of the itinerary can be booked, then the credit card is charged. If the credit card charge fails, the transaction also fails.

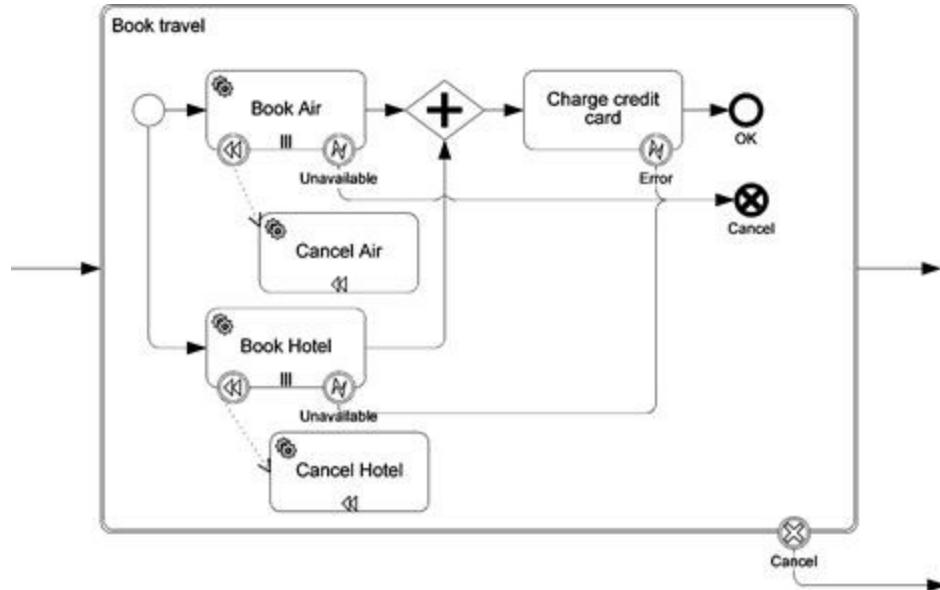


Figure 10-5. Transaction compensation, complex case

Here you can see the convenience of compensation, since there are many potential points of failure in this transaction. The state of each of the individual leg bookings at the point of failure cannot be known in advance. If you had to consider all the possible combinations and add paths to the diagram describing what to do if failure occurs in one state versus another, it would be a nightmare. With compensating activities, BPMN just applies a simple rule: if the activity has completed successfully when compensation is commanded, then execute its compensating activity; if it has not, do not execute the compensating activity.

In Figure 10-5, if any of the leg booking activities – that is, any *instances* of the multi-instance Book Hotel and Book Air activities – fail, the resulting Cancel throw-catch undoes just those instances that have already completed. If all of the leg booking instances complete but the credit card charge fails, then that Cancel undoes all of the bookings.

Now let's take it one step further. Suppose that after the transaction is complete, the customer for some reason decides to cancel the trip. This might not be modeled as part of the same process, but let's say in this case it was. The *Book travel* transaction is complete, but we want to undo it after the fact. This is a good use case for the *direct Compensation throw-catch*. We can't use Cancel because a Cancel throw must come from within the transactional subprocess.

Figure 10-6 illustrates. For argument's sake, we'll say that once all the bookings are complete, the travel agency requests final confirmation from the customer before charging the credit card. The travel agency waits 24 hours before charging the customer in case of cancellation. To wait for either 24 hours or the cancellation message, we use an event gateway. If a cancellation message is received, we can still undo the bookings using the previously defined compensating activities, but we cannot use a Cancel event because the transactional subprocess is already complete. Here we use a throwing Compensation event *Undo Book Travel*, targeted at the transactional subprocess, *Book Travel*. This triggers all of the compensating activities within that subprocess.

[15]

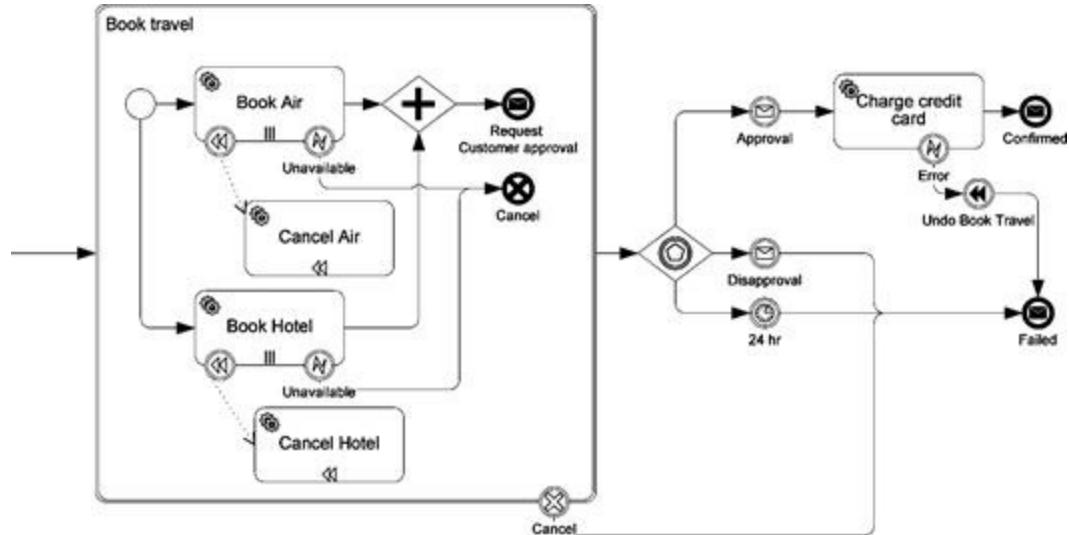


Figure 10-6. Throwing Compensation event

Note that compensation does not handle the exception. It merely rolls back the transaction to its initial state. The exception handling notifying the customer, perhaps adding a cancellation fee to the invoice, must be added in the exception flow.

Chapter 11

The Rules of BPMN

Before concluding Level 2 modeling, we return again to the topic of BPMN style. In the end, it's all a matter of following the rules, both the rules of the BPMN spec and the style rules. Most of the bad BPMN in the world – diagrams that are confusing, ambiguous, or just make no sense at all – could be eliminated if the modeler would simply *follow the rules*. That is easy to say, but less easy in practice. Modelers depend on tools to *validate* their BPMN models and warn them about violations, and tool vendors depend on the BPMN specification to define the validation rules. The difficulties start with the spec itself.

Sources of BPMN Truth

It is inexcusable, but the BPMN 2.0 specification – all 500+ pages of it – never enumerates its rules. There should be an Appendix where the rules are listed in one place, but there is not. One member of the BPMN 2.0 Finalization Task Force admitted to me that they had wanted to provide such a list but “ran out of time.” I don’t know, but seven years seems like enough time to me.

Moreover, the spec does not even provide a single source of truth! It provides, by my count, three separate sources, and they do not agree in every case. The first source is the BPMN *metamodel*, expressed in UML class diagrams and their serialization in OMG’s *XML Metadata Interchange* (XMI) format. The second source is the *schema*, an alternative formulation of the metamodel in “normal” XML, i.e., based on the *XML Schema Definition* (XSD) language.

Both the metamodel and its XSD equivalent define the various BPMN elements, their attributes, and the relationships between them. The XSD and XMI serializations of the metamodel are nominally equivalent, but differences between the two languages prevent perfect agreement. For example, the metamodel says a sequence flow may only connect to a *flow node*, meaning an activity, gateway, or event, while the XSD allows connection to any BPMN element. That means the rule that a sequence flow may only connect to a flow node cannot be tested by schema validation... but it remains a rule nonetheless.

OMG’s Model Driven Architecture emphasizes XMI, but XSD is far more often used by software tools and developers. It is the basis of XML standards, model interchange, SOA, and web services. Thus, in the BPMN Implementer’s Guide section of this book, most of the discussion is based on the XSD, not XMI.

In any case, both the XMI and XSD only define basic rules for each BPMN element class as a whole (e.g., boundary events), not all the rules specific to individual elements (e.g., Error boundary events). Most rules about individual element types are sprinkled throughout the spec narrative, in 500 pages of tables and text that refine and override the metamodel. The spec narrative thus constitutes the third source of truth, a sometimes ambiguous one. Some rules are stated plainly in the text, while others must be inferred.

Further muddying the waters is the fact that the spec references the BPMN *semantic elements* not the shapes and symbols. Usually this presents no problems since, for the most part, each semantic element corresponds to a single shape/symbol combination. But some shapes have no corresponding semantic element. For example, there is no semantic *Multiple event* element, even though there is a distinct shape for it in the notation. A Multiple event shape simply means an event with more than one event definition in the semantic model.

Finally, some of the “requirements” stated in the BPMN spec are applicable only to executable processes. They involve technical details omitted in most BPMN models and not represented in the notation.

For all these reasons, *each BPMN tool is forced to make up its own list of validation rules*. Even where tools agree on the content of the rule, they will differ on the text of it. There simply is no official list of BPMN rules.

BPMN Rules for Level 2 Process Modeling

If OMG won't provide a list, I will. Below is my list of the most important official rules for Level 2 (non-executable) process modeling. Some readers may say about one or two of them, "I don't see that rule in the BPMN spec." I have two responses to that. First, many of the rules of the spec are implied by other parts of the specification. And second, if it makes them feel better about it, those readers could always consider it a style rule. Good BPMN means conforming to both the official rules and the style rules.

Sequence Flow

1. A sequence flow must connect to a flow node (activity, gateway, or event) at both ends. Neither end may be unconnected.
2. All flow nodes other than start events, boundary events, and catching Link events must have an incoming sequence flow, if the process level includes any start or end events. [Exceptions, not part of the Level 2 palette: compensating activity, event subprocess.]
3. All flow nodes other than end events and throwing Link events must have an outgoing sequence flow, if the process level includes any start or end events. [Exceptions, not part of the Level 2 palette: compensating activity, event subprocess.]
4. A sequence flow may not cross a pool (process) boundary.
5. A sequence flow may not cross a process level (subprocess) boundary.
6. A conditional sequence flow may not be used if it is the only outgoing sequence flow.
7. Sequence flow out of a parallel gateway or event gateway may not be conditional. [Note: On sequence flows out of gateways, conditional is an invisible attribute; the conditional tail marker is suppressed on sequence flows out of gateways.]
8. An activity or gateway may have at most one default flow.

Message Flow

9. A message flow may not connect nodes in the same process (pool).
10. The source of a message flow must be either a Message or Multiple end event or throwing intermediate event; an activity; or a black-box pool.
11. The target of a message flow must be either a Message or Multiple start event, catching intermediate event, or boundary event; an activity; or a black-box pool. [Exceptions, not part of the Level 2 palette: event subprocess Message or Multiple start event.]
12. Both ends of a message flow require a valid connection. Neither end may be unconnected.

Start Event

13. A start event may not have an incoming sequence flow.
14. A start event may not have an outgoing message flow.
15. A start event with incoming message flow must have a Message or Multiple trigger.
16. A start event may not have an Error trigger. [Exceptions, not part of Level 2 palette: event subprocess start event].
17. A start event in a subprocess must have a None trigger. [Exceptions, not part of Level 2 palette: event subprocess start event].

End Event

18. An end event may not have outgoing sequence flow.
19. An end event may not have incoming message flow.
20. An end event with outgoing message flow must have Message or Multiple result.

Boundary Event

21. A boundary event must have exactly one outgoing sequence flow. [Exception, not part of the Level 2 palette: Compensation.]
22. A boundary event trigger may include only Message, Timer, Signal, Error, Escalation, Conditional, or Multiple. [Exceptions, not part of Level 2 palette: Cancel, Compensation, Multiple-Parallel.]
23. A boundary event may not have incoming sequence flow.
24. An Error boundary event on a subprocess requires a matching Error throw event.
25. An Error boundary event may not be non-interrupting.
26. An Escalation boundary event on a subprocess requires a matching Escalation throw event.

Throwing or Catching Intermediate Event

27. An intermediate event with incoming message flow must be catching type with Message or Multiple trigger.
28. An intermediate event with outgoing message flow must be throwing type with Message or Multiple trigger.
29. A throwing intermediate event result may include only Message, Signal, Escalation, Link, or Multiple. [Exceptions, not part of Level 2 palette: Compensation.]
30. A catching intermediate event trigger may include only Message, Signal, Timer, Link, Conditional, or Multiple.
31. A throwing Link event may not have outgoing sequence flow.

32. A catching Link event may not have incoming sequence flow.

Gateway

33. A gateway may not have incoming message flow.
34. A gateway may not have outgoing message flow.
35. A splitting gateway must have more than one gate.
36. Gates of an event gateway may include only a catching intermediate event or Receive task.

Process (Pool)

37. A process must contain at least one activity.
38. Elements of at most one process may be contained in a single pool.
39. A pool may not contain another pool. If a child-level subprocess expansion is enclosed in a pool, that pool must reference the same participant and its associated process as the parent level.

Style Rules for Level 2 Process Modeling

The official rules of the spec allow a diagram to be “valid” but ambiguous in meaning. Style rules are Method and Style *conventions*, consistent with the official rules, intended to make the process logic clear from the diagram alone. The most important style rules are listed below.

Labeling

The label of a diagram shape corresponds to the *name* attribute of the semantic element.

1. An activity should be labeled, ideally VERB-NOUN.
2. Two activities in the same process should not have the same name, unless they are both call activities.
3. A triggered start event should be labeled to indicate the trigger condition.
 - a. A Message start event should be labeled “Receive [message name]”.
 - b. A Timer start event should be labeled to indicate the process schedule.
 - c. A Signal start event should be labeled to indicate the Signal name.
 - d. A Conditional start event should be labeled to indicate the trigger condition.
4. A boundary event should be labeled.
5. The label of an Error boundary event on a subprocess should match the label of a child-level Error end event.
6. The label of an Escalation boundary event on a subprocess should match the label of a child-level throwing Escalation event.
7. A throwing intermediate event should be labeled.
8. A catching intermediate event should be labeled.
9. Paired Link events should have matching labels.
10. Throwing and catching events corresponding to the same Signal event definition should have matching labels, if they occur in the same BPMN model.
11. An end event should be labeled with the name of the end state.
12. A splitting XOR gateway should have at most one unlabeled gate.
13. A splitting XOR or inclusive gateway should be labeled if any of its gates are unlabeled.
14. The label of a child-level diagram (page) should match the name of the subprocess.

End Event

15. Two end events in a process level should not have the same name. If they signify the same end state, combine them; otherwise give them distinct names.

16. If a subprocess is followed by a yes/no gateway, at least one end event of the subprocess should be labeled to match the gateway label.

Subprocess Expansion

17. Only one start event should be used in a subprocess, unless it is a parallel box.
18. A child-level expansion should not be enclosed in an expanded subprocess shape if parent and child process levels are represented by separate diagrams.

Message Flow

19. A message flow should be labeled directly with the name of the message.
20. A Send task should have an outgoing message flow.
21. A Receive task should have an incoming message flow.
22. A Message start event should have an incoming message flow.
23. A catching Message event should have incoming message flow.
24. A throwing Message event should have outgoing message flow.
25. A message flow from a collapsed subprocess should be replicated in the child-level diagram.
26. A message flow to a collapsed subprocess should be replicated in the child-level diagram.
27. An incoming message flow in child-level diagram should be replicated in the parent level.
28. An outgoing message flow in child-level diagram should be replicated in the parent level.

Model Validation

It is far easier to comply with the rules of BPMN when your tool can validate models against them and list all the violations. Many BPMN tools provide some type of model checking against the official BPMN rules. There is only one, to my knowledge, that has implemented the style rules: Process Modeler for Visio, from [itp commerce](#).

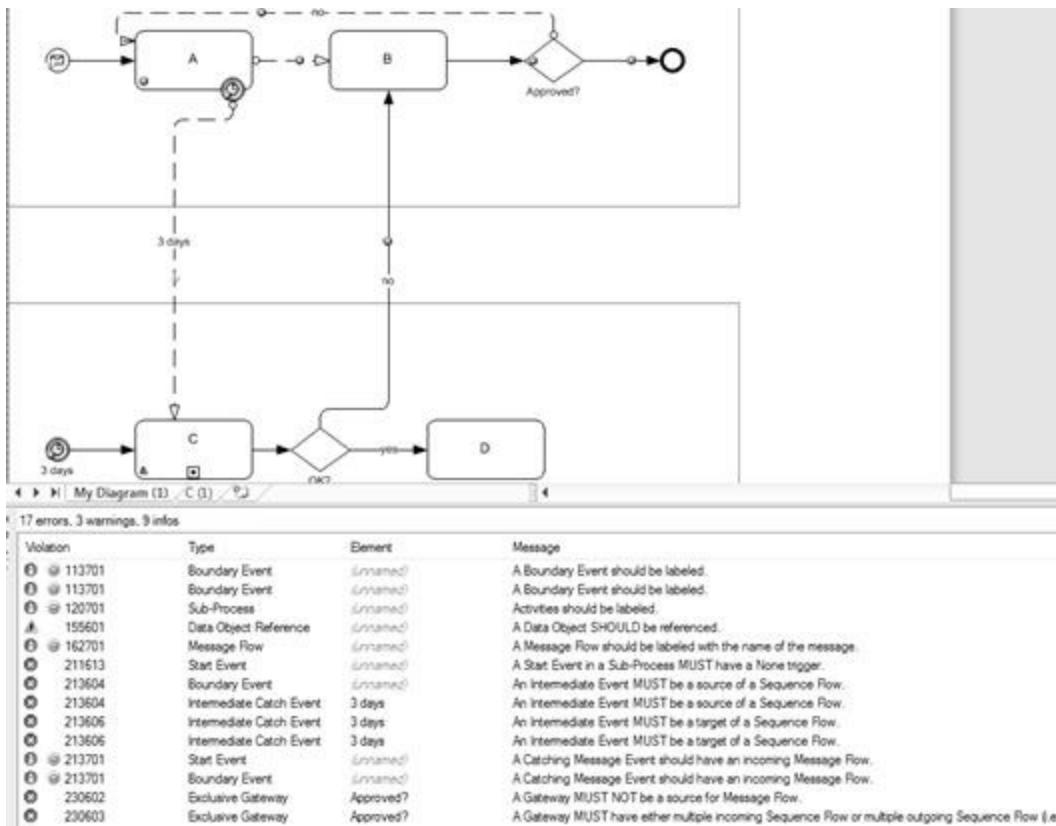


Figure 11-1. Validation against both official rules and style rules in [itp commerce](#) tool.

Figure 11-1 illustrates validation of a particularly error-filled model in the [itp commerce](#) tool. Elements with violations are tagged with icons in the diagram – *x* for a spec violation, *i* for a style rule violation – and you can navigate in the tool easily between a selected shape and its associated violations, or from a violation in the list to its shape.

I liken model validation to spelling and grammar checking in a word processing program. Many of the violations are the equivalent of “typos,” inadvertent or careless errors. You don’t need to continuously validate as you model, but it’s a good idea to do validate before you declare the model ready for release to others... and, of course, you must fix all of the reported errors.

If your tool can export a model in the BPMN 2.0 XML interchange format, I have created an online tool that will validate it against both the spec rules and the style rules. You upload the XML to the website and it creates the validation report. For further details, see the website for this book, www.bpmnstyle.com.

Part IV:
BPMN Implementer's Guide –
Non-Executable BPMN

Chapter 12

BPMN 2.0 Metamodel and Schema

The world generally understands BPMN to mean Business Process Modeling Notation, and that is what it stood for in BPMN 1.2. But actually OMG changed the acronym in version 2.0 to stand for Business Process Model *and* Notation. In fact, most of the work that went into the BPMN 2.0 specification had nothing to do with the notation, the shapes and symbols, which were left mostly unchanged from BPMN 1.2. It had to do with defining a *metamodel* for BPMN, a formal specification of the semantic elements comprising a BPMN model and their relationships to each other. All valid BPMN models must conform to the specifications of the metamodel.

Metamodel elements are defined as *object classes* with defined required and optional *attributes*. Some classes are subtypes of other classes and inherit their attributes, while adding more of their own. A model element may be a subtype of more than one class, and inherits the attributes of all of them. Some classes, like *Root Element* or *Base Element*, are purely abstract, not used directly in BPMN models. Their purpose is merely to provide a single point of definition of attributes shared among its subclasses.

[\[16\]](#)

In the BPMN 2.0 specification document [\[16\]](#), the metamodel is represented by *UML class diagrams*, augmented by tables and text in the narrative. For example, Figure 12-1 depicts the *Definitions* class. The classes are organized in sets called *packages*. The packages are layered for extensibility, each layer building on and extending lower layers. Many elements of the four Core packages are shared by BPMN's three types of models: Process, Collaboration, and Choreography. In this book we are concerned only with Process and Collaboration models.

The metamodel is also published in two alternative XML formats, OMG's XML Metadata Interchange

[\[17\]](#)

(XMI) and W3C's XML Schema Definition (XSD) [\[17\]](#). They are nominally equivalent representations of the BPMN metamodel, although XSD cannot represent certain relationships of the UML, such as multiple inheritance. XSD is the language of "normal" XML used by the Web, SOA, and application software. It is also the language most BPMN tool vendors will use to interchange models. For that reason, in this book we will focus on the XSD representation of the BPMN metamodel.

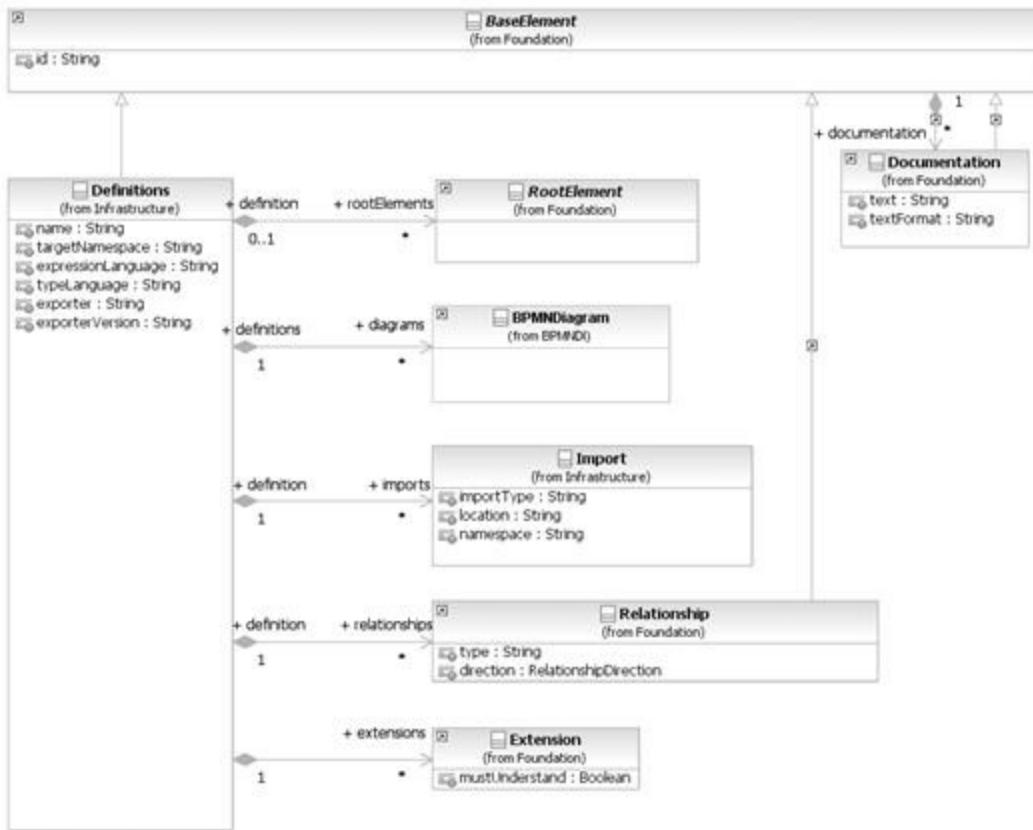


Figure 12-1. Definitions class diagram. Source: OMG

XSD Basics

[18]

A full explanation of the XSD language^[18] is beyond the scope of this book, but for those unfamiliar with it, a few basics here will be helpful to understand the discussion of BPMN model serialization.

An XML schema is itself an XML document. You can view or edit it as tagged text, but many XML tools also provide a graphical view that is more helpful for understanding the schema structure. Figure 12-2 illustrates, for example, a fragment of both the text and graphical representations of the BPMN root

[19]

definitions element in XML Spy^[19] from Altova, the tool I use.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:bpmndi="http://www.omg.org/spec/BPMN/20100524/DI"
  targetNamespace="http://www.omg.org/spec/BPMN/20100524/MODEL"
  elementFormDefault="qualified" attributeFormDefault="unqualified">
  <xsd:import namespace="http://www.omg.org/spec/BPMN/20100524/DI"
    schemaLocation="BPMDI.xsd"/>
  <xsd:include schemaLocation="Semantic.xsd"/>
  <xsd:element name="definitions" type="tDefinitions"/>
  <xsd:complexType name="tDefinitions">
    <xsd:sequence>
      <xsd:element ref="import" minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element ref="extension" minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element ref="rootElement" minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element ref="bpmndi:BPMDiagram" minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element ref="relationship" minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:ID" use="optional"/>
    <xsd:attribute name="name" type="xsd:string"/>
    <xsd:attribute name="targetNamespace" type="xsd:anyURI" use="required"/>
    <xsd:attribute name="expressionLanguage" type="xsd:anyURI" use="optional"
      default="http://www.w3.org/1999/XPath"/>
    <xsd:attribute name="typeLanguage" type="xsd:anyURI" use="optional"
      default="http://www.w3.org/2001/XMLSchema"/>
    <xsd:attribute name="exporter" type="xsd:string"/>
    <xsd:attribute name="exporterVersion" type="xsd:string"/>
    <xsd:attribute namespace="##other" processContents="lax"/>
  </xsd:complexType>
  <xsd:element name="import" type="tImport"/>
  <xsd:complexType name="tImport">
    <xsd:attribute name="namespace" type="xsd:anyURI" use="required"/>
    <xsd:attribute name="location" type="xsd:string" use="required"/>
    <xsd:attribute name="importType" type="xsd:anyURI" use="required"/>
  </xsd:complexType>
</xsd:schema>
```

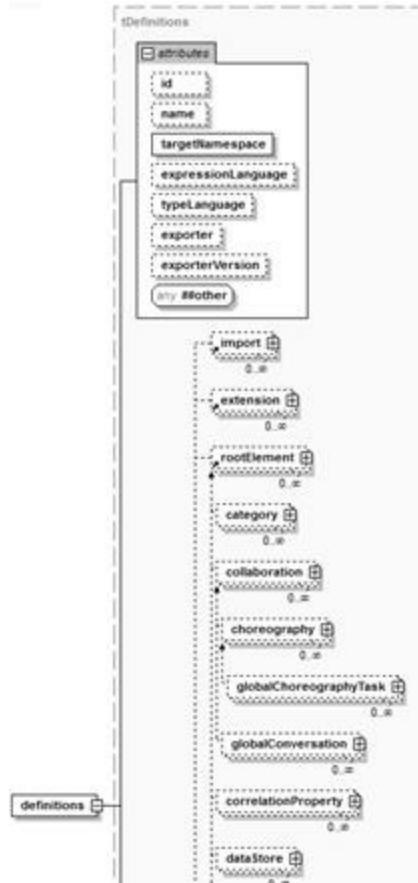


Figure 12-2. Text and Graphical views of the BPMN schema in XML Spy

The XSD defines the element names and datatypes, their attributes and child elements. A BPMN 2.0 model is, by definition, incorrect unless it is a *valid* instance of the schema. Most XML editors contain a *schema processor* that lets you validate a process model against the BPMN schema in a single mouse click. Some tools will allow you to save a BPMN model even if it is not schema-valid, but some may not. And tools that manipulate XML, such as XSLT editors, may require the input to be schema-valid in order to work at all. Thus creating schema-valid BPMN is an absolute requirement for any implementer. Not all the rules of BPMN are enforced by schema validation, but passing schema validation is an absolute minimum requirement for BPMN model correctness.

Note in the text representation of Figure 12-2 that each tag name has two parts, a *prefix* separated by a colon from the *local name*. The prefix is shorthand for the *namespace*, typically specified as a URL. Usually all the namespaces used in the schema are declared using *xmlns* attributes of the root *xsd:schema* element. The prefix *xsd*, for example, represents the namespace <http://w3.org/2001/XMLSchema>, which is the namespace for the XSD language itself. Sometimes you see the prefix *xs* declared for that namespace, or both *xsd* and *xs* in the same schema document. It wouldn't matter if the prefix were *qwp*; the thing that counts is the declared namespace URL that corresponds to the prefix.

The *targetNamespace* attribute of the root *xsd:schema* element identifies the namespace associated with this particular schema. Here, for example, the namespace <http://www.omg.org/spec/BPMN20100524/MODEL> signifies the *BPMN 2.0 namespace*. All BPMN 2.0 models must reference this namespace.

Another thing to notice about Figure 12-2 is that certain elements in the graphical representation, such as *category* or *collaboration*, appear to be missing in the text representation. They are actually defined in other XSD files that are *included* or *imported*. The text view on the left represents just the file *BPMN20.xsd*, but note that it *includes* another file, *Semantic.xsd*, and *imports* a third one, *BPMDI.xsd*. *Include* means the other XSD has the same *targetNamespace* as *BPMN20.xsd*; *import* means it has a different one, in this case <http://www.omg.org/spec/BPMN20100524/DI>. The graphical view on the right automatically combines the original file with its included and imported elements, while the text view does not.

The *xsd:sequence* element enclosing a list of child elements specifies the required *order* of those elements in an instance document. The *minOccurs* attribute specifies whether the element is required or not. In the XML Spy graphical view, *optional* elements (*minOccurs="0"*) have a dotted border, required elements a solid border. The *maxOccurs* attribute specifies whether the element may be repeated. If an XML instance omits a required element or puts them in the wrong order, it is a schema violation.

Attributes of an element may occur in any order. They may be either required or optional, but may not be repeated. If an attribute has a defined *default* value, omission of the attribute means exactly the same thing as presence of the attribute with a value equal to the default.

Each element and attribute in the schema has a datatype, or *type*. The XSD language defines a large number of basic types, and additional simple and complex types may be defined within the schema itself. From the text view of Figure 12-2, note that the element *definitions* is assigned to the type *tDefinitions*, defined just below it in the schema.

Finally, in the graphical view of Figure 12-2, note the dotted vertical arrow from *category*, *collaboration*, *dataStore*, and others to *rootElement*. That signifies these elements are subtypes of *rootElement*, which in XSD is called a *substitutionGroup*. Substitution groups are as close as XSD comes to UML subclasses. In XSD, an element may have only one *substitutionGroup*, whereas in UML an element may be a subclass of many different classes.

BPMN Schema Fundamentals

XSD Files

The BPMN 2.0 schema is distributed as a set of five XSD files: *BPMN20.xsd*, *Semantic.xsd*, *BPMNDI.xsd*, *DI.xsd*, and *DC.xsd*. Implementers should store them locally in the same folder. *BPMN20.xsd* is the top level. It includes *Semantic.xsd* and imports *BPMNDI.xsd*, which in turn imports *DI.xsd* and *DC.xsd*.

By itself, *BPMN20.xsd* represents the *Infrastructure* package of the BPMN metamodel Core. It contains just two elements, *definitions* and *import*. A single *definitions* element is always at the root of any BPMN XML instance document. The *import* element allows a single BPMN *model* to be composed of multiple BPMN XML *documents* (files), supporting reuse of independently maintained global tasks and processes.

Semantic and Graphical Models

In the BPMN XSD, the *graphical model* – information concerning the graphical layout of shapes, such as position, size, and connection points – is entirely separate from the semantic model. Both semantic and graphical models are enclosed within a single *definitions* element. The graphical model, called *BPMNDI*, specifies no semantic information at all; it says only that shapes with a bounding box of some particular size exist at some location on a page. You cannot tell from BPMNDI whether the shape is an activity or event, except by tracing its *bpmnElement* attribute, a pointer to the *id* of an element in the semantic model. A valid BPMN model may omit BPMNDI entirely, but you may not omit the semantic model. BPMNDI without semantic model information is meaningless.

IDs and ID References

Most elements in the BPMN 2.0 XSD have an *id* attribute of type *xsd:ID*, a type defined by the XSD language for use in attributes only. ID types have special requirements. Their values must start with either a letter or underscore, and can contain only letters, digits, underscores, hyphens, and periods. More important, their values must be *unique within an XML instance*, regardless of the attribute's name. In other words, there can at most one element in a BPMN model with an *id* value of `_12345`.

This uniqueness is critical because relationships between model elements are maintained by *pointers* to other elements via their *id* value. For example, a sequence flow's *sourceRef* attribute matches the *id* of the flow node connected to the tail of the sequence flow. Elements and attributes with "Ref" in their names are typically of type *IDREF*, a pointer to an attribute of type *ID*. An XML instance document will not pass schema validation if any *IDREF* elements or attributes point to an *id* value that is missing in the document, or if duplicate *id* values exist anywhere in the document.

Import, targetNamespace, and Remote ID References

Recall that a BPMN instance document may *import* other BPMN instance documents. This is not the same as an XSD file importing another XSD file, but it works in a similar manner. One of the documents represents the top level or root of the BPMN model, but all the documents together constitute a single BPMN model. This *import* feature is the key to BPMN modularity and reuse.

A *reusable subprocess*, for example, is defined as a *top-level process* in its own BPMN document. Let's call it *Billing*. The *Billing* process may be invoked as a reusable subprocess using a *Call Activity* from another BPMN instance document. The BPMN document containing the Call Activity must *import* the document defining the *Billing* process. This allows *Billing*, which is called by multiple end-to-end processes, to be maintained independently of its various calling process definitions. In a mature BPM environment, such modularity is the rule rather than the exception, but few BPMN tool vendors have yet considered its implications for model serialization.

When one BPMN document imports another, some "Ref" elements or attributes will point to an *id* in another file. And since the imported file, say a called process, was defined without knowledge of other BPMN documents that might someday import it, there is the possibility that an *id* value is duplicated between the imported and importing documents. It is not clear whether that would be a schema violation or not, since ID types must be unique only within an instance document, but an ambiguity would definitely exist for any IDREF pointing to the *id*: Which element does it point to?

Here is where the BPMN spec does something unusual. To avoid the potential problem of pointers to duplicate *ids*, the BPMN XSD defines many "Ref" elements and attributes not as IDREF types but QName types. In XSD, QName normally means a namespace prefix-qualified *name*, but BPMN uses it for a namespace prefix-qualified *id* value. The namespace here is the *targetNamespace* declared by the model's *definitions* element.

This is very strange indeed. In the BPMN 2.0 XSD, *targetNamespace* is a *required* attribute of the root element *definitions*. Normally in XML a targetNamespace is defined for a *schema*, but here we are talking about a targetNamespace for an *instance document*, a particular BPMN model. It's not the same thing at all. Its only purpose here is to support *id* references to elements in imported documents, using the *targetNamespace* prefix to unambiguously identify the referenced element. Unlike with IDREF, a schema processor cannot validate the presence of the *id* value referenced by QName.

Here is an example. The *sourceRef* attribute of a message flow is possibly a reference to an element in an imported BPMN file, so it is defined in the XSD as a QName. Let's say the source of the message flow is a task in the imported *Billing* model with *id* value *Task001*, and the *Billing* model targetNamespace is mapped to the prefix *billing*. In that case, the *sourceRef* value should not be simply *Task001* but *billing:Task001*. This resolves any possible ambiguity between *Task001* in *Billing* and *Task001* in the calling process model.

In a model where there is no import or where the importing and imported documents have the same targetNamespace, it is perfectly acceptable to omit the prefix on QName references, and this is the most common situation.

Most tool vendors that support BPMN 2.0 export today populate the targetNamespace with a fixed value for all BPMN models, something that identifies the vendor or tool, not the particular model. *But that could defeat its intended purpose.* The spec intends that tool vendors populate targetNamespace with a value that uniquely identifies the particular BPMN model.

Using a fixed value for targetNamespace is OK if the tool guarantees uniqueness of *id* values globally, across all documents, not just within an instance document. Tools that use hashing or similar techniques to generate *globally unique IDs* can get away with a fixed value for the targetNamespace. Those are tools where the *id* value is some long, seemingly random string of characters. But tools that use simple *ids* like *Task001* must define unique targetNamespace values for each BPMN document if they want to avoid ambiguous remote references. (Today, most vendors avoid the problem because they do not yet support *import*. But that is a temporary artifact of an immature BPMN 2.0 market. Ultimately, any serious BPMN tool must support *import* and remote *id* references, because they are required for task and process reuse.)

Chapter 13

Process Modeling Conformance Subclasses

[20]

Here is what the BPMN 2.0 spec says about conformance:

“Software can claim compliance or conformance with BPMN 2.0 if and only if the software fully matches the applicable compliance points as stated in the specification. Software developed only partially matching the applicable compliance points can claim only that the software was based on this specification, but cannot claim compliance or conformance with this specification. The specification defines four types of conformance namely Process Modeling Conformance, Process Execution Conformance, BPEL Process Execution Conformance, and Choreography Modeling Conformance....

The implementations claiming Process Modeling Conformance MUST support the following BPMN packages:

- The BPMN core elements, which include those defined in the Infrastructure, Foundation, Common, and Service packages.
- Process diagrams, which include the elements defined in the Process, Activities, Data, and Human Interaction packages.
- Collaboration diagrams, which include Pools and Message Flow.
- Conversation diagrams, which include Pools, Conversations, and Conversation Links.”

As an alternative to *full* Process Modeling Conformance, there are three *Process Modeling Conformance subclasses* defined:

- Descriptive
- Analytic
- Common Executable

Without the addition of these Process Modeling Conformance subclasses in the Finalization phase of BPMN 2.0, it is doubtful we would ever see software that could claim compliance or conformance under the terms stated. Some of the packages referenced for full conformance contain elements only used in executable models, and all of them contain many obscure and rarely used elements. I cannot imagine a tool vendor supporting every one of them.

But common sense prevailed in the end. The Descriptive and Analytic conformance subclasses are explicitly for *non-executable models*, and include *only the information visible in the diagram itself*. Sound familiar? It should, because these subclasses were based on BPMN Method and Style Level 1 and Level 2

palettes!

*****ebook converter DEMO Watermarks*****

Descriptive Subclass

The Descriptive subclass corresponds to the Level 1 palette. The elements and attributes in the table below are referenced by their XML names in the XSD. Some “attributes” in Figure 13-1 are actually *elements* in the XSD.

Element	Attributes
participant (pool)	id, name, processRef
laneSet	id, lane with name, childLaneSet, flowElementRef
sequenceFlow	id, name, sourceRef, targetRef
messageFlow	id, name, sourceRef, targetRef
exclusiveGateway	id, name
parallelGateway	id, name
task (None)	id, name
userTask	id, name
serviceTask	id, name
subProcess	id, name, flowElement
callActivity	id, name, calledElement
dataObject	id, name
textAnnotation	id, text
association	id, name, sourceRef, targetRef, associationDirection
dataAssociation	id, name, sourceRef, targetRef
dataStoreReference	id, name, dataStoreRef
startEvent (None)	id, name
endEvent (None)	id, name
messageStartEvent	id, name, messageEventDefinition
messageEndEvent	id, name, messageEventDefinition
timerStartEvent	id, name, timerEventDefinition
terminateEndEvent	id, name, terminateEventDefinition
documentation	text
Group	id, categoryValueRef

Figure 13-1. Descriptive Subclass elements and attributes

Note that the elements in the left column match up exactly with the Level 1 palette from the Chapter 4 of this book. Of more significance is the right column, which specifies the details of each element that a tool must support in order to conform to the Descriptive subclass. It is just the *name* (the label in the diagram), the *id* and *id references*, and a few elements that determine the icon or marker, such as *messageEventDefinition* – in other words, just the information that is visible in the diagram!

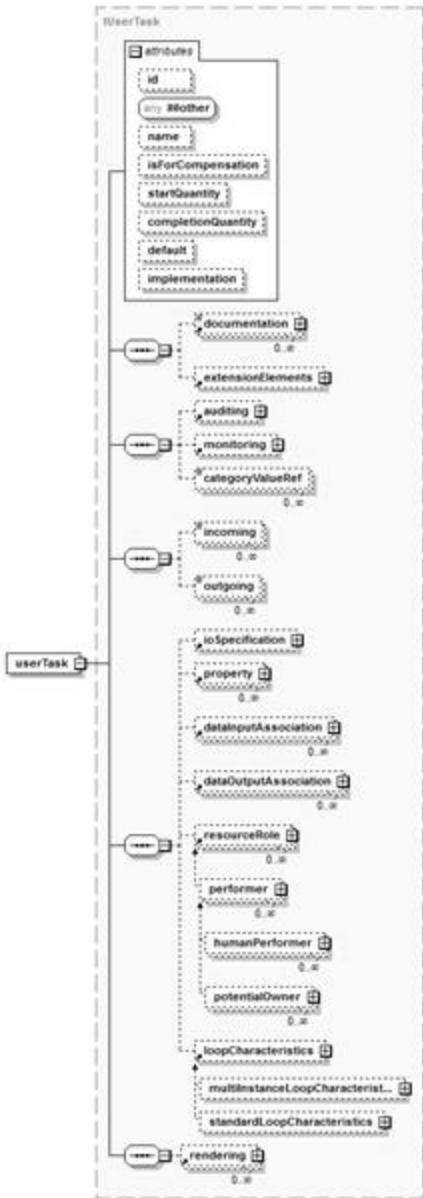


Figure 13-2. User task schema

That is just a tiny fraction of the elements and attributes defined in the XSD. Figure 13-2 is a condensed view of the schema for a single element, *userTask*. It is condensed because each of the boxes with the [+] marker – and this is most of them – can be further expanded to reveal additional child elements and attributes.

Now you see why the Descriptive and Analytic subclasses are so important to interoperability between tools. Full conformance, according to the spec, would demand a tool must “support” all of these elements, that is, be able to export and import them and understand their meaning. That is simply not realistic. Conformance with the Descriptive subclass, however, demands a tool support only *documentation* and the attributes *id* and *name*. Actually there are a couple more, related to data flow connections, but it’s still a tiny fraction of the full schema.

Analytic Subclass

The Analytic subclass corresponds to the Level 2 palette. Like the Descriptive subclass, Analytic also just reflects information visible in the diagram, not the execution-related details underneath each shape and symbol. The subclass includes everything in the Descriptive subclass plus the elements and attributes shown in Figure 13-3.

Element	Attribute
sequenceFlow	conditionExpression, default
sendTask	id, name
receiveTask	id, name
Looping activity	standardLoopCharacteristics
Multi-instance activity	multiinstanceLoopCharacteristics
exclusiveGateway	Default
inclusiveGateway	id, name, default
eventBasedGateway	id, name, eventGatewayType
Link event pair	id, name, linkEventDefinition/@name
Signal start/end event	id, name, signalEventDefinition
Signal throw/catch intermediate event	id, name, signalEventDefinition
Signal boundary event	id, name, signalEventDefinition, attachedToRef, cancelActivity
Message throw/catch intermediate event	id, name, messageEventDefinition
Message boundary event	id, name, messageEventDefinition, attachedToRef, cancelActivity
Timer catching event	id, name, timerEventDefinition
Timer boundary event	id, name, timerEventDefinition, attachedToRef, cancelActivity
Error boundary event	id, name, errorEventDefinition, attachedToRef
Error end event	id, name, errorEventDefinition
Escalation throw intermediate event	id, name, escalationEventDefinition
Escalation end event	id, name, escalationEventDefinition
Escalation boundary event	id, name, escalationEventDefinition, attachedToRef, cancelActivity (false only)
Conditional start event	id, name, conditionalEventDefinition
Conditional catch intermediate event	id, name, conditionalEventDefinition
Conditional boundary event	id, name, conditionalEventDefinition, attachedToRef, cancelActivity
message	id, name
Message flow	messageRef

Figure 13-3. Analytic Subclass elements and attributes

Common Executable Subclass

The spec defines a third process modeling conformance subclass called Common Executable. The palette is in between Descriptive and Analytic, but it contains additional attributes related to executable details. We will discuss it more fully in Chapter 19.

Chapter 14

BPMN Serialization Basics

definitions

The top-level element in any BPMN model instance document is *definitions*. In this book I use the terms BPMN *document* and BPMN *file* interchangeably. Because a BPMN document can *import* another one, a single BPMN *model* may be composed of multiple BPMN documents. In that case, one of the documents is the top level of the hierarchy; the import references may not be circular. Each document must be enclosed in a *definitions* element.

```
<xsd:element name="definitions" type="tDefinitions"/>
<xsd:complexType name="tDefinitions">
<xsd:sequence>
  <xsd:element ref="import" minOccurs="0" maxOccurs="unbounded"/>
  <xsd:element ref="extension" minOccurs="0" maxOccurs="unbounded"/>
  <xsd:element ref="rootElement" minOccurs="0" maxOccurs="unbounded"/>
  <xsd:element ref="bpmndi:BPMDiagram" minOccurs="0" maxOccurs="unbounded"/>
  <xsd:element ref="relationship" minOccurs="0" maxOccurs="unbounded"/>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID" use="optional"/>
<xsd:attribute name="name" type="xsd:string"/>
<xsd:attribute name="targetNamespace" type="xsd:anyURI" use="required"/>
<xsd:attribute name="expressionLanguage" type="xsd:anyURI" use="optional" default="http://www.w3.org/1999/XPath"/>
<xsd:attribute name="typeLanguage" type="xsd:anyURI" use="optional" default="http://www.w3.org/2001/XMLSchema"/>
<xsd:attribute name="exporter" type="xsd:string"/>
<xsd:attribute name="exporterVersion" type="xsd:string"/>
<xsd:anyAttribute namespace="#<other" processContents="lax"/>
</xsd:complexType>
```

Figure 14-1. *definitions* schema

Figure 14-1 shows the schema for *definitions*. The attributes *id* and *name* are optional and rarely used. The latter would represent the name of the BPMN model.

targetNamespace

The attribute *targetNamespace*, discussed earlier, is required. The datatype is *anyURI*, which usually is a URL. No file or web page is required to exist at the URL; it simply identifies a namespace. Most tools today use the same URL for all BPMN models, something that identifies the tool or tool vendor. However, as discussed earlier, this only works when globally unique *id* values are assigned to model elements, since there is the risk of ambiguous remote references due to duplicate *ids* in imported BPMN documents. In general, it is better to generate a model-specific *targetNamespace* value, perhaps related to the model *name*.

expressionLanguage and typeLanguage

The attributes *expressionLanguage* and *typeLanguage* are optional. The first identifies the language used in data expressions, such as gateway conditions. If the attribute is omitted, the default XPath 1.0 (<http://www.w3.org/1999/XPath>) is implied. The global *expressionLanguage* value provided here may be overridden on individual expression elements. *typeLanguage* identifies the language used to specify datatypes of model elements. The default value is the XSD language. The global value may be overridden on individual data elements.

These attributes are not part of the Analytic class, which assumes the defaults.

exporter and exporterVersion

*****ebook converter DEMO Watermarks*****

String attributes *exporter* and *exporterVersion* identify the tool and tool version used to serialize the model. The attributes are optional in the XSD but are recommended if the export is intended to be interoperable with other tools.

Global Namespace Declarations

Since *definitions* is the root element in the BPMN document, it should provide namespace declarations for all of the namespaces used in the document. Namespaces may be declared locally in elements where they are used, but it is better in general to declare them in *definitions*. Namespace declarations are attributes of the form *xmlns[:prefix]=[namespace URI]*. The *default namespace* – implied for elements with no prefix – is usually set to the BPMN 2.0 namespace, i.e., *xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"*. In addition, the XSD and BPMNDI namespaces must be declared, as well as those of imported documents.

schemaLocation

The *xsi:schemaLocation* attribute is used by XML tools to validate your model against the BPMN 2.0 XSD. The value of this attribute is constructed by concatenating the namespace with a filepath or URL pointing to the file BPMN20.xsd, separated by a space. The attribute's prefix *xsi*: indicates the XML Schema Instance namespace, which is used for schema locations. (If the schemaLocation is provided, the XSI namespace also needs to be declared, as described above.)

For example, *xsi:schemaLocation="http://www.omg.org/spec/BPMN/20100524/MODEL schemas/BPMN20.xsd"* says that the schema for the BPMN 2.0 namespace is located at the path *schemas/BPMN20.xsd* (relative to the BPMN file). Instead of a local file path, you may point to the official schema location on the web, *http://www.omg.org/spec/BPMN/20100501/BPMN20.xsd*.

import

Element *import* identifies another XML document imported into the model. The element is optional and unbounded, meaning any number of import elements are allowed. Although very few tools do it today, the spec says support of *import* is *required* for conformance. BPMN specifically identifies three types of imports that MUST be supported – BPMN documents, XSD files, and WSDL files – but allows others in addition. Each import is defined by three *required* attributes:

- *importType*. The value (an absolute URI) MUST be set to *http://www.w3.org/2001/XMLSchema* when importing XML Schema 1.0 documents, to *http://www.w3.org/TR/wsdl20/* when importing WSDL 2.0 documents, and *http://www.omg.org/spec/BPMN/20100524/MODEL* when importing BPMN 2.0 documents. Other types of documents MAY be supported.
- *location*. The value (string) is the file location or URL of the imported document.
- *namespace*. The value (an absolute URI) must match the *targetNamespace* of the imported file.

extension

Child element *extension* (optional, unbounded), according to the spec, “allows BPMN adopters to attach additional attributes and elements to standard and existing BPMN elements.” In addition to a child *documentation* element, each *extension* includes two attributes:

- *definition*. A QName reference to an element in an imported XSD.
- *mustUnderstand*. A Boolean.

The *extension* element in *definitions* binds the imported data definition globally to the model. I have never seen this construct used. In practice, proprietary tool vendor extensions most often use *extensionElements* within specific model elements.

rootElement

The *rootElement* children of *definitions* represent reusable elements of the BPMN *semantic model*. These include the basic model types *process*, *collaboration*, and *choreography*, plus any other globally reusable elements, such as global tasks, event definitions, data store, and message. In the XSD, root element is designated as *abstract* – meaning you should never see an element named *rootElement* in a BPMN instance. Concrete root elements that designate *rootElement* as its *substitutionGroup* automatically inherit the properties of the root element class in the metamodel. For example, only the root elements of a BPMN document may be referenced by another BPMN document that *imports* the first one. For example, a call activity’s *calledElement* may point to an imported *process* (root element) but not to an imported *subProcess* (not a root element). There is no prescribed order of root elements. Specific root elements are defined in *Semantic.xsd*.

BPMNDiagram

The *bpmndi:BPMNDiagram* children of *definitions* comprise the BPMN *graphical model*, specifying the location, size, and page organization of the shapes in the diagram. Each *BPMNDiagram* element represents a different page or *diagram* in the model. The element is prefixed because it is in a separate namespace. We will discuss the graphical model in more detail in Chapter 17.

relationship

Child element *relationship* provides another BPMN extension mechanism specifying user-defined relationships between *source* and *target* model elements, such as between as-is and to-be process models. I have never seen this used in practice.

documentation and extensionElements

Most BPMN model elements contain child elements *documentation* and *extensionElements*.

- *documentation* is part of the Descriptive and Analytic subclasses. It has no graphical representation in the diagram. It allows embedding any documentation content in the process model XML.
- *extensionElements* is not part of the Analytic subclass, but is the normal way BPMN tools insert proprietary information used by the tool itself, including information beyond the scope of the BPMN standard, such as simulation parameters. Children of *extensionElements* should be prefixed with the namespace of the tool or tool vendor.

collaboration

What was called in BPMN 1.2 a *Business Process Diagram* is in BPMN 2.0 called a *collaboration model*. In the diagram it contains one or more processes interacting via message flows. In the semantic model, the root element *collaboration* merely defines the participants, message flows, and artifacts. Each *process* referenced by a *participant* is a separate root element. (Technically, collaboration also contains a number of elements related to Choreography and Conversation models, but those are outside the scope of this book.)

participant

A *pool* in the diagram is a shape that references a *participant* in the semantic model. In the Method and Style section of this book I said that a pool is primarily a container for a *process*, as it was officially in BPMN 1.2, and only secondarily a partner role or entity involved in a business-to-business interaction, which is how the term *participant* is described in the BPMN 2.0 spec narrative. By equating pool to participant, the BPMN 2.0 spec muddies the waters but effectively changes very little. The reason is simple: A participant element may either reference *no process*, in which case we call it a black-box pool, or a *single process*. Within any BPMN model, a single participant may not be associated with more than one process. Thus, in reality, except for black-box pools, the terms *participant* and *process* signify the same thing.

There is also the issue of compatibility with existing BPMN models. In BPMN 1.2, it was common to draw a pool enclosing a single BPMN process, even when no other pool or message flows was drawn. If a pool means a role or business entity engaged in a collaboration, would a diagram with a single pool even be legal in BPMN 2.0? In fact, in early drafts of BPMN 2.0, it was not; the XSD required a minimum of *two participants*. Fortunately that requirement was later dropped.

The *participant* element has three attributes in the Analytic subclass:

- *id*. You need to specify this if you want to draw a pool shape in the diagram. It is the unique value pointed to by the *bpmnElement* attribute of a pool shape in the graphical model.
- *name*. This is the label displayed in the pool shape. In the Method and Style section of this book I advise labeling a process pool with the name of the *process*. In the BPMN XML, that value becomes the *participant name*. In the BPMN 2.0 graphical model, there is no shape associated with the semantic element *process*. A pool, meaning a participant, is the closest thing we have. For that reason, I recommend applying the pool label value to both the participant and process *name* attributes.
- *processRef*. This is a QName pointer to a *process* element. It is QName because the *process* and *collaboration* elements could be in different BPMN files. Omission of this attribute indicates a *black-box pool*, i.e., no process. And if present, there can be at most *one* of them.

Child element *participantMultiplicity*, if present, is visualized through the *multi-instance participant marker* discussed in Chapter 8.

messageFlow

*****ebook converter DEMO Watermarks*****

The *messageFlow* semantic element has five important attributes:

- *id*. Optional if no graphical model is provided, but required as *bpmnElement* reference for the graphical connector.
- *name*. This is the connector label, identifying the message.
- *sourceRef* and *targetRef*. These are *required* QName pointers to the semantic elements at the message flow tail and head, respectively. They must be valid sources and targets for messages, as discussed in Chapter 7.
- *messageRef*. I tend not to use it, but it is included in the Analytic subclass. If you show the Message shape on a message flow, *messageRef* is a QName pointer to a root *message* element that specifies the shape label and (in executable models) the technical details of the message.

The location and endpoints of the message flow connector are defined in the graphical model, not in the semantic *messageFlow* element.

process

The root element *process* describes a BPMN process, that is, an *orchestration*, as discussed in detail in Chapter 2. Attributes of this element include:

- *id*. Required as target for *participant* attribute *processRef*, indicating a white-box pool.
- *name*. Actually, the process *name* appears on no shape label, unless you follow the Method and Style convention of making the participant and process names identical. A good reason to follow this convention is to support process reuse via *call activity*. Although the call activity's *calledElement* reference in the XML is the process *id*, in a BPMN tool the modeler is most likely to browse and select the called process by *name*.
- *processType* (not in Analytic subclass). Optional enumerated string attribute *processType* specifies whether a process is Public or Private. A *Public process*, called an *abstract process* in BPMN 1.2, contains only nodes that interact with outside entities via messages. A *Private process*, in contrast, contains the complete activity flow logic. *None*, the default value of *processType*, signifies undefined.
- *isExecutable* (not in Analytic subclass). Optional Boolean attribute for Private processes. If this attribute is omitted, the process is *implicitly non-executable*. Certain rules in the spec narrative apply only to executable processes.

Example: Simple Process Model

The simple model depicted in Figure 14-2 is serialized in Figure 14-3.

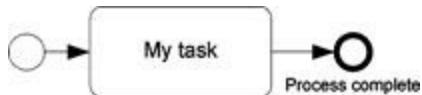


Figure 14-2. A simple process model

```
<definitions targetNamespace="http://www.itp-commerce.com"
  xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"
  xmlns:itp="http://www.itp-commerce.com/BPMN2.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.omg.org/spec/BPMN/20100524/MODEL schemas/BPMN20.xsd"
  exporter="Process Modeler 5 for Microsoft Visio" exporterVersion="5.2742.13663 SR6"
  itp:name="My Diagram" itp:version="1.0" itp:author="bruce" itp:creationDate="8/3/2011 2:42:47 PM"
  itp:modificationDate="8/3/2011 3:07:20 PM"
  id="_4adb855a-76f3-4539-8a1d-60102fbb12e7">
  <process id="_4188bf61-cb2f4f72-a84f-9f4f70b41a6b" name="My Process" processType="None">
    <startEvent id="_3f808752-02dd-42d5-b4aa-2015031c7cc7"/>
    <task id="_0532502d-31db-4fa5-920b-65c173652055" name="My task"/>
    <endEvent id="_7986530a-fb47-4918-83fb-ad6c4f7d7656" name="Process complete"/>
    <sequenceFlow id="_6e913629-e553-47bf-875a-ce53cc167bcd" sourceRef="_3f808752-02dd-42d5-b4aa-2015031c7cc7" targetRef="_0532502d-31db-4fa5-920b-65c173652055"/>
    <sequenceFlow id="_bfcf1cad-0c47-40df-9bd3-0e744bfe5bd2" sourceRef="_0532502d-31db-4fa5-920b-65c173652055" targetRef="_7986530a-fb47-4918-83fb-ad6c4f7d7656"/>
  </process>
</definitions>
```

Figure 14-3. Serialization of a simple process model

Several things are worth noting about the serialization, generated by Process Modeler for Visio from itp commerce ltd.

- *ids* for all elements are tool-generated globally unique values.
- The *targetNamespace* declaration is not model-specific but the same for all models serialized by this tool. That is acceptable since the tool generates globally unique element *ids*.
- The default (unprefixed) namespace is declared to be the BPMN 2.0 namespace, <http://www.omg.org/spec/BPMN/20100524/MODEL>. Some tools use a prefix for this namespace.
- Two other namespaces are declared in the *definitions* element, the prefix *xsi* to reference the *schemaLocation* attribute, and the prefix *itp* to reference tool vendor proprietary elements and attributes.
- The *xsi:schemaLocation* attribute indicates that this instance document is to be validated against the BPMN 2.0 schema found at the relative file location *schemas/BPMN20.xsd*.
- The *exporter* and *exporterVersion* identify the tool and version used to create the serialization.
- Vendor-proprietary attributes in with the *itp* prefix are used to hold non-standard information about the model, such as the model name, version, author, creation date and modification date.
- The *process* element has a *processType* value of *None*. Since that is the default, this attribute could have been omitted.
- The *name* attribute of the *task* and *endEvent* elements match their labels in the diagram.
- The sequenceFlow *sourceRef* and *targetRef* values match the *id* values of the source and target

nodes.

Example: Simple Collaboration Model

Figure 14-4 illustrates a simple collaboration model, serialized in Figure 14-5.

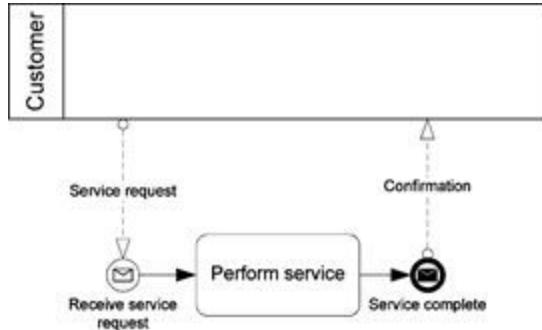


Figure 14-4. Simple collaboration model

```
<definitions targetNamespace="http://www.itp-commerce.com" xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"
  xmlns:itp="http://www.itp-commerce.com/BPMN2.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.omg.org/spec/BPMN/20100524/MODEL schemas/BPMN20.xsd"
  exporter="Process Modeler 5 for Microsoft Visio" exporterVersion="5.2742.13663 SR6"
  itp:name="My Diagram" itp:version="1.0" itp:author="bruce" itp:creationDate="8/3/2011 3:41:57 PM"
  itp:modificationDate="8/3/2011 3:47:47 PM" itp:createdWithVersion="5.2742.13663 SR6"
  id="_1f2848e9-2fd8-49ab-96ae-1411838c1e70"
  <process id="_98663b88-a518-493a-96f4-e1b2b7c3aace" name="Main Process" processType="None">
    <startEvent id="_d028c241-0061-4c86-99a4-b8e9ab4e3a54" name="Receive service request">
      <messageEventDefinition />
    </startEvent>
    <task id="_94aa77d8-a54d-4000-aa0-b00cfbbb652d" name="Perform service">
      <endEvent id="_94f1e4b4-d44e-4fc8-8d02-fd9320c4ace0" name="Service complete">
        <messageEventDefinition />
      </endEvent>
    </task>
    <sequenceFlow id="_006c73de-346f-4111-8824-e687db8210c6" sourceRef="_94aa77d8-a54d-4000-aa0-b00cfbbb652d" targetRef="_94f1e4b4-d44e-4fc8-8d02-fd9320c4ace0"/>
    <sequenceFlow id="_3d9765f0-4006-4998-a5ec-438ffa29aa3a" sourceRef="_d028c241-0061-4c86-99a4-b8e9ab4e3a54" targetRef="_94aa77d8-a54d-4000-aa0-b00cfbbb652d "/>
      <process>
        <collaboration id="_fd9acbee-264b-44dc-bae0-d3d3e74f751">
          <participant id="_8eea715d-f551-4487-9a64-6226dea487cd" name="Customer"/>
          <participant id="p_98663b88-a518-493a-96f4-e1b2b7c3aace" name="Main Process" processRef="_98663b88-a518-493a-96f4-e1b2b7c3aace"/>
        <messageFlow id="_e817a1b2-f0dd-4a49-b33d-25da322872ae" name="Service request" sourceRef="_8eea715d-f551-4487-9a64-6226dea487cd" targetRef="_d028c241-0061-4c86-99a4-b8e9ab4e3a54"/>
        <messageFlow id="_566cf079-8ac8-4ca4-9b01-a0dac679962d" name="Confirmation" sourceRef="_94f1e4b4-d44e-4fc8-8d02-fd9320c4ace0" targetRef="_8eea715d-f551-4487-9a64-6226dea487cd"/>
      </collaboration>
    </process>
  </process>
</definitions>
```

Figure 14-5. Serialization of a simple collaboration model

Noteworthy differences from Figure 14-3 include:

- In addition to the *process* element there is a *collaboration* element. The *process* is not contained in the *collaboration*, but is another root element.
- The *collaboration* identifies two *participants*. One, named *Customer*, has no *processRef* attribute, indicating it is a black-box pool. The *participant name* is taken from the pool label. The second *participant* has no pool shape available to name it, so by default it takes the *name* of the *process*. In this case, since I did not assign a process name, the tool gave it the default name *Main Process*. The participant takes the same name. The *processRef* of the *participant* points to the *process id*.
- The empty child element *messageEventDefinition* identifies the start and end events as Message events.
- The *messageFlow sourceRef* and *targetRef* values point to a Message event at one end and the

*****ebook converter DEMO Watermarks*****

participant Customer at the other end.

*****ebook converter DEMO Watermarks*****

Example: Simple Import and Call Activity

Figure 14-6 illustrates a process that calls *My Process* as a *reusable subprocess* using a *call activity*. The called process is the same as the one depicted in Figure 14-2 and serialized in Figure 14-3. In order to reference *My Process*, the calling process must first *import* the BPMN file defining the called process.

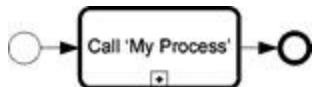


Figure 14-6. Simple process with call activity

```
<definitions targetNamespace="http://www.itp-commerce.com" xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"
  xmlns:i="http://www.itp-commerce.com/BPMN2.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.omg.org/spec/BPMN/20100524/MODEL schemas/BPMN20.xsd"
  exporter="Process Modeler 5 for Microsoft Visio" exporterVersion="5.2742.13663 SR6"
  itp:name="My Diagram" itp:version="1.0" itp:author="bruce" itp:creationDate="8/4/2011 11:02:21 AM"
  itp:modificationDate="8/4/2011 11:07:18 AM" itp:createdWithVersion="5.2742.13663 SR6"
  itp:conformanceSubClass="Full"
  id="c66bdc7-22fb-4b94-ac58-c28d0fc76c16">
<import namespace="http://www.itp-commerce.com" location="C:\Users\Bruce\Documents\book\draft\14-2.bpmn" importType="http://www.omg.org/spec/BPMN/20100524/MODEL"/>
<process id="_c77810f5-3926-40fc-81fd-1bb409bc5c91" name="Main Process" processType="None">
<startEvent id="708b45b8-bd58-4a33-b629-ee96e4a785f0"/>
<callActivity id="_0c280062-dd03-4f62-ae45-db61a2b5cb93" name="Call 'My Process'" calledElement="_4188bfa1-cb2f-4f72-a84f-9f4f70b41a6b" itp:isCollapsed="true"/>
<sequenceFlow id="_8843eeb9-faeb-4bbe-ab0-214831acc38b" sourceRef="_708b45b8-bd58-4a33-b629-ee96e4a785f0" targetRef="_0c280062-dd03-4f62-ae45-db61a2b5cb93"/>
<endEvent id="_800e79a2-ed8e-4f69-8fab-c4cc4d38b53c"/>
<sequenceFlow id="_33364eeb-f363-40f5-b543-8335095abca0" sourceRef="_0c280062-dd03-4f62-ae45-db61a2b5cb93" targetRef="_800e79a2-ed8e-4f69-8fab-c4cc4d38b53c"/>
</process>
</definitions>
```

Figure 14-7. Serialization of import and call activity

There are several things to note about Figure 14-7:

- The imported file is not contained in the serialization. *Import* does not copy the contents of the imported process, but simply points to it.
- The *import* attribute *namespace* is the *targetNamespace* of the imported BPMN file. Recall that the itp commerce tool uses a fixed *targetNamespace* for all its models, which it can do because it uses globally unique element *ids*.
- The *import* attribute *location* is the relative filepath to the imported .bpmn (XML) file.
- The *import* attribute *importType* identifies the import as a BPMN 2.0 file.
- The *callActivity* attribute *calledElement* matches the *id* of the *process* element of the imported file, which you can verify from Figure 14-3. If the imported file had had a different *targetNamespace*, the *calledElement* value would be prefixed.

Now that we have seen the basic XML structure of the BPMN semantic model, the next chapter explains how to serialize the flow elements of a process.

Chapter 15

Serializing Process Elements

flowElement and flowNode

flowElement, an optional unbounded child of *process*, represents the abstract class (called *substitutionGroup* in XSD) of elements that belong to a *process*. These include *sequenceFlow*, *dataObject*, *dataObjectReference*, *dataStoreReference*, and the *flowNode* elements, meaning those that can connect to sequence flows, namely *activity*, *gateway*, and *event*.

All members of the *flowElement* class have an optional *name* attribute that appears as the label of the corresponding shape, and three rarely-used child elements: *auditing*, *monitoring*, and *categoryValueRef*. The first two of those, which are not in the Analytic subclass, are undefined placeholders for some future standard; it's unclear why they are part of the specification at all. *categoryValueRef* is a QName pointer to a *categoryValue* element, a child of a *category* root element. In the Analytic subclass its only purpose is to associate a *group* shape with elements of a particular category value. I have never seen it used.

The *flowNode* class adds two more optional unbounded child elements, *incoming* and *outgoing*. These are QName pointers to incoming and outgoing sequence flows, respectively. In that sense they are completely redundant to the *sourceRef* and *targetRef* attributes of the sequence flows themselves, which are *required* in the XSD. I believe it is incorrect to make these QName, as a sequence flow may not connect to a flowNode in another BPMN document, and the *sourceRef* and *targetRef* of a sequence flow are local IDREFs not QName. Thus, since they have no apparent purpose, I recommend omitting *incoming* and *outgoing*.

Each distinct type of activity, gateway, and event is represented by a separate XSD element in the semantic model. The elements *activity*, *gateway*, and *event* themselves are *abstract* classes, not used directly in the XML but from which the specific subtypes inherit various attributes and child elements. In the BPMN XML you must use the concrete subtype elements like *userTask* or *exclusiveGateway*.

activity

The *activity* abstract class adds several attributes and child elements to *flowNode*:

- Optional IDREF attribute *default* identifies a *default sequence flow* outgoing from the activity. It is in the Analytic subclass.
- Optional Boolean attribute *isForCompensation* (default value *false*) identifies a compensating activity. It is not in the Analytic subclass.
- Optional integer attributes *startQuantity* and *completionQuantity*, both default value *1*, are used only in executable processes. They are not in the Analytic subclass.
- Optional child elements *property*, *ioSpecification*, *dataInputAssociation*, and *dataOutputAssociation* are related to data flow. The last three are effectively part of the Descriptive and Analytic subclasses, as they are needed to serialize data flow in Level 1 or Level 2 diagrams. We will discuss data flow modeling in more detail in Chapter 16.
- Optional child element *resourceRole* and its subtypes *humanPerformer* and *potentialOwner* are related to human task assignment in executable processes, discussed in more detail in Chapter 22. They are not in the Analytic subclass.
- Optional child element *multiInstanceLoopCharacteristics* signifies a *multi-instance activity*. Boolean attribute *isSequential* indicates whether the bars in the marker should be horizontal (sequential) or vertical (parallel). The default value is *false*, so if omitted the MI behavior is parallel. This is the only detail of *multiInstanceLoopCharacteristics* in the Analytic subclass; there are many more elements and attributes used to describe complex MI behaviors in executable BPMN.
- Optional child element *standardLoopCharacteristics* signifies a *loop activity*, part of the Analytic subclass. The following details are not part of the subclass. Optional Boolean attribute *testBefore*, default value *false*, determines whether the loop condition is evaluated before or after running the activity. Optional integer attribute *loopMaximum* lets you put an upper limit on the iterations. Optional child *loopCondition* is a conditional expression used in executable processes.

The *activity* element is an abstract class and should not be used directly. Instead you must use a concrete element representing a particular task or subprocess type. The following activity type elements are included in the Analytic subclass:

- *task* (called Abstract task in the spec narrative)
- *userTask*
- *serviceTask*
- *sendTask*
- *receiveTask*
- *callActivity*
- *subProcess* (meaning “embedded” subprocess other than a transaction or ad-hoc subprocess)

*****ebook converter DEMO Watermarks*****

The following activity elements are outside the Analytic subclass:

- *scriptTask*
- *businessRuleTask*
- *manualTask*
- *adHocSubProcess*
- *transaction*

Reusable task definitions, called *global tasks*, are not defined within a process but are root (callable) elements like *process* itself. The XSD defines the following global task types:

- *globalTask* (Abstract task)
- *globalUserTask*
- *globalScriptTask*
- *globalManualTask*
- *globalBusinessRuleTask*

serviceTask, *sendTask*, and *receiveTask* are implicitly reusable as-is, so they do not have corresponding *globalTask* types. Although the spec does not say so, we will assume that if a task type is in the Descriptive or Analytic subclass, its corresponding global type is a member of that subclass as well.

Other than those like documentation and loop/multi-instance characteristics, inherited from the base *activity* class, attributes and child elements of specific activity types are outside the Analytic subclass. They are there to support executable processes and discussed further in Part V:

- *userTask* has attribute *implementation*, with allowed values `##unspecified` (the default), `##WebService`, or a URL to indicate a defined implementation such as WS-HumanTask (<http://docs.oasis-open.org/ns/bpel4people/ws-humantask/protocol/200803>). It also has optional child *rendering* that can be used to specify user interface details under *extensionElements*.
- *serviceTask* has attribute *implementation* with the same allowed values as *userTask*, except `##WebService` is the default. Attribute *operationRef* is a remote (QName) reference to a web service operation, typically from an imported WSDL file.
- *sendTask* has the same *implementation* and *operationRef* attributes and defaults as *serviceTask*. In addition, attribute *messageRef* is a QName pointer to a *message* element, typically in an imported XSD or WSDL.
- *receiveTask* has the same attributes as *sendTask*, with the addition of optional Boolean attribute *instantiate*, default value `false`. Immediately following a None start event, a *receiveTask* with an *instantiate* value of `true` means the same thing as a Message start event, i.e., receipt of the message instantiates the process. Since the value of *instantiate* is not visible in the diagram, this construction is visually ambiguous, so Method and Style deprecates its use in favor of Message start event.
- *callActivity* has optional QName attribute *calledElement*, a pointer to either a *process* or *global task*. The *calledElement* is typically defined in an imported BPMN file, but it could be in

the same file as the *callActivity*.

subProcess

subProcess has two differences from standard *activity*. Optional Boolean attribute *triggeredByEvent*, if *true*, signifies an *event subprocess*. The default value *false* signifies a regular subprocess. Event subprocess is not in the Analytic subclass.

In the XML, a *subProcess* element encloses all of the *flowElements* in its *child process level*. A *subProcess* element in the child level encloses its children as well, and this nesting can extend without limit. Just to restate the point, process level containment is not modeled by pointers to element *ids*; the elements themselves are enclosed within the *subProcess* tags. Since all the elements in a process level are physically contained within its parent *subProcess* (or, at the top level, *process*) element, BPMN is *inherently hierarchical*.

It is important to note that there is *nothing* in the semantic *subProcess* element that indicates whether the child-level flow is drawn *inline*, inside an *expanded subProcess* shape on the same page (diagram) as the parent level, or *hierarchically*, in a separate diagram linked to a *collapsed subProcess* shape in the parent level. Notions of expanded vs. collapsed subprocesses or inline vs. hierarchical modeling styles are purely aspects of the *graphical model*. The serialization of the *subProcess* element in the semantic model is exactly the same no matter how it is drawn in the diagram! In BPMN 1.2, some people (including tool vendors) mistakenly thought embedded and expanded subprocesses were the same thing, or at least went hand in hand. The BPMN 2.0 schema should finally put this idea to rest.

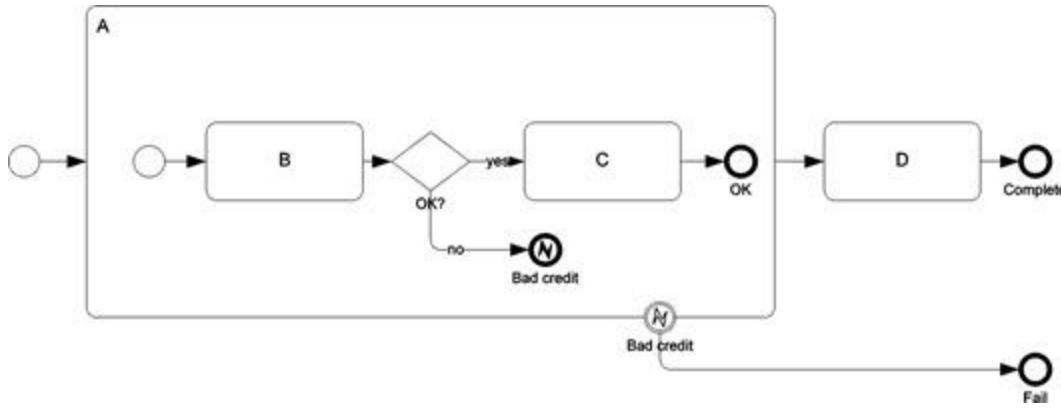


Figure 15-1. Process model with two process levels

Figure 15-1 illustrates a process model with two process levels. The serialization in Figure 15-2 shows the nesting of the child-level elements underneath the *subProcess* element.

```
<definitions targetNamespace="http://www.itp-commerce.com" xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL" xmlns:itp="http://www.itp-commerce.com/BPMN2.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://www.omg.org/spec/BPMN/20100524/MODEL schemas/BPMN20.xsd" exporter="Process Modeler 5 for Microsoft Visio"
  exporterVersion="5.2742.13663 SR6" itp:name="My Diagram" itp:version="1.0" itp:author="bruce" itp:creationDate="8/4/2011 12:17:24 PM" itp:modificationDate="8/4/2011 1:57:52 PM"
  itp:createdWithVersion="5.2742.13663 SR6" itp:conformanceSubClass="Full" id="_1ae6a483-77a8-4eed-be89-f1e343bf9bf6">
  <process id="2fc6601-1839-44ed-af36-5e67811891e1" name="Main Process" processType="None">
    <startEvent id="_3fbcb343-3a7c-4023-805f-d12f747dceb"/>
    <subProcess id="_e7df3b40-d626-4920-bc92-d006dad77502" name="A" itp:isCollapsed="false">
      <startEvent id="_89aa9447-1739-46d4-87eb-c7b459e6f06e"/>
      <task id="_3bc7dc56-15b0-4581-87d8-832427474fc" name="B"/>
      <task id="_ca47d153-84f6-4f12-98a2-6bbc3269e3ae" name="C"/>
      <exclusiveGateway id="_b7941c03-aaa0-4afc-9c32-4d689aa440a8" name="OK?" gatewayDirection="Diverging"/>
      <endEvent id="_ef84c60-db46-404d-822a-b08451799db0" name="OK"/>
      <endEvent id="_06abf31e-4092-43e3-af27-5a419d7b79ab" name="Bad credit">
        <errorEventDefinition/>
      </endEvent>
      <sequenceFlow id="_ad57bdee-f5a2-4221-a9a6-be6f692879c" sourceRef="_89aa9447-1739-46d4-87eb-c7b459e6f06e" targetRef="_3bc7dc56-15b0-4581-87d8-832427474fc"/>
      <sequenceFlow id="_2151821f-1c15-4143-8999-425c4a39876d" sourceRef="_3bc7dc56-15b0-4581-87d8-832427474fc" targetRef="_b7941c03-aaa0-4afc-9c32-4d689aa440a8"/>
    </subProcess>
  </process>
</definitions>
```

*****ebook converter DEMO Watermarks*****

```

<sequenceFlow id="_677cc983-f5c5-47d6-98ff-221b56599a98" name="yes" sourceRef="_b7941c03-aaa0-4afc-9c32-4d689aa440a8" targetRef="_ca47d153-84f6-4f12-98a2-6bbc3269e3ae">
  <conditionExpression>test='yes'</conditionExpression>
</sequenceFlow>
<sequenceFlow id="_56128bff-1d44-4c19-a95b-3eba6ecfdaf3" sourceRef="_ca47d153-84f6-4f12-98a2-6bbc3269e3ae" targetRef="_efd84c60-db46-404d-822a-b08451799db0"/>
<sequenceFlow id="_db96d25e-ab43-407a-a0c6-de2a7fe4fe" name="no" sourceRef="_b7941c03-aaa0-4afc-9c32-4d689aa440a8" targetRef="_06abf31e-4092-43e3-af27-5a419d7b79ab">
  <conditionExpression>test='no'</conditionExpression>
</sequenceFlow>
</subProcess>
<boundaryEvent id="_2a05521f-e21f-4609-bc73-51e458f2f1a2" name="Bad credit" cancelActivity="true" attachedToRef="_e7df3b40-d626-4920-bc92-d006dad77502">
  <errorEventDefinition/>
</boundaryEvent>
<endEvent id="_c81b1121-93e1-430a-a07d-8edc3c756301" name="Complete"/>
<task id="_ac76b036-81f3-4adb-ad81-0be41e226d92" name="D"/>
<endEvent id="_742c449d-0e13-41bc-83b4-dd29c633132d" name="Fail"/>
<sequenceFlow id="_1814f042-9656-4987-bcad-cd326fa07b33" sourceRef="_3fbcb343-3a7c-4023-805f-d12f747fdeeb" targetRef="_e7df3b40-d626-4920-bc92-d006dad77502"/>
<sequenceFlow id="_4ce984e-5584-4ac2-8d86-c9f31a0141f2" sourceRef="_2a05521f-e21f-4609-bc73-51e458f2f1a2" targetRef="_742c449d-0e13-41bc-83b4-dd29c633132d"/>
<sequenceFlow id="_e88e9485-01d4-4256-bcb6-49188bf40e9b" sourceRef="_e7df3b40-d626-4920-bc92-d006dad77502" targetRef="_ac76b036-81f3-4adb-ad81-0be41e226d92"/>
<sequenceFlow id="_37590ff4-87b7-4625-a30c-ac79852561a4" sourceRef="_ac76b036-81f3-4adb-ad81-0be41e226d92" targetRef="_c81b1121-93e1-430a-a07d-8edc3c756301"/>
</process>
</definitions>

```

Figure 15-2. Serialization of process model with two process levels

Note that the *subProcess* XML element physically encloses its child-level *task* elements. Also note that *boundaryEvent Bad credit* is in the parent level and the *endEvent Bad credit* is in the child level.

gateway

The abstract *gateway* class adds the optional attribute *gatewayDirection* to the standard *flowNode* attributes and elements. This attribute, with the enumerated values *Unspecified*, *Converging*, *Diverging*, and *Mixed*, is only used in executable processes and is not part of the Analytic subclass. It seems redundant, as the splitting versus merging semantics are evident from the sequence flows connected to the gateway.

The gateway *conditions* are not defined under the *gateway* element, but in the *sequenceFlow* elements representing the gates.

Like the other abstract elements, *gateway* is not used in the process model XML. Instead, there is a separate XML element for each gateway type. Attributes and child elements of the gateway type elements follow the base gateway class, with the following exceptions:

- *exclusiveGateway* and *inclusiveGateway* add the optional Boolean attribute *default*, an IDREF pointer to an outgoing sequence flow representing the default flow, the sequence flow enabled if no other gate conditions are true.
- *parallelGateway* has no differences from the base gateway schema.
- *complexGateway* has *default* and adds a child element *activationCondition*, a data expression.
- *eventBasedGateway* does not have *default* but adds two other attributes.

Optional Boolean attribute *instantiate*, default value *false*, indicates that the trigger of any of the gates instantiates the process. When this attribute is *true*, the symbol inside the diamond shape is a *start event*, not an intermediate event. In that case, this element must be the first node of a top-level process or immediately follow a None start event. As the semantics are the same as multiple triggered start events, and few tools support the instantiating gateway shape, Method and Style deprecates *instantiate* in favor of multiple triggered start events.

Optional attribute *eventGatewayType*, with enumerated values *Exclusive* and *Parallel*, indicates whether the flow continues when the *first* gate event occurs (*Exclusive*, the default) or when *all* of them occur (*Parallel*). A value of *Parallel* is equivalent to a *Multiple-Parallel catching intermediate event*, which is not in the Analytic subclass. Omitting this element signifies the normal event gateway behavior, and is recommended.

event

The abstract *event* class adds only the child element *property* to the base *flowNode* class. We will discuss *property* in Chapter 16. Like the other abstract classes, *event* is not used directly in process model XML; instead each type of event is represented by a separate element.

startEvent

The *startEvent* element represents the start event of a process, subprocess, or event subprocess.

- Optional Boolean attribute *isInterrupting*, default value *true*, has meaning only in an event subprocess and should be omitted otherwise. It determines whether the event subprocess trigger is interrupting or non-interrupting. Event subprocesses are not in the Analytic subclass.
- Optional Boolean attribute *parallelMultiple*, default value *false*, signifies a *Parallel-Multiple start event*. That means *all* triggers must occur in order to instantiate the process, or trigger the event subprocess. Parallel-multiple start is not in the Analytic subclass, so this attribute normally should be omitted.
- Child elements *property*, *dataOutput*, *dataOutputAssociation*, and *outputSet* relate to data flow, and are discussed in Chapter 16.
- The abstract *eventDefinition* class defines the start event *trigger*. In the XML, you must use one of the concrete *eventDefinition* subtypes as a child of a triggered *startEvent*. These include *timerEventDefinition*, *messageEventDefinition*, *signalEventDefinition*, and *conditionalEventDefinition*. The schema allows additional event definitions valid only for event subprocesses, including *errorEventDefinition*, *escalationEventDefinition*, *compensateEventDefinition*, and *cancelEventDefinition*. Others like *linkEventDefinition* and *terminateEventDefinition* are technically schema-valid, but are actually not allowed for start events. A *None start event* is signified by omission of the *eventDefinition* element. A *Multiple start event* is signified by more than one *eventDefinition* element; if attribute *parallelMultiple* is *true*, it signifies a *Multiple-Parallel start event*. Each *eventDefinition* element has trigger-specific attributes and child elements, but these are meant for executable BPMN and are outside the Analytic subclass. For non-executable models, an empty *eventDefinition*-type element is all you need to specify the trigger.
- As an alternative to embedding an *eventDefinition* as direct child of a *startEvent*, it is possible to point to a reusable *eventDefinition* root element, using *eventDefinitionRef* (QName).

intermediateCatchEvent

The *intermediateCatchEvent* element represents a catching intermediate event with sequence flow in and out; it is not used for a boundary event. The attributes and children of *intermediateCatchEvent* are the same as those of *startEvent*, with the following exceptions:

- There is no *isInterrupting* attribute.
- The same set of *eventDefinition* elements is allowed by the XSD, but the only valid ones are those allowed in Figure 7-1: *messageEventDefinition*, *timerEventDefinition*, *conditionalEventDefinition*, *linkEventDefinition*, and *signalEventDefinition*. Since there is no *None* catching intermediate event, at least one of the above *eventDefinition* elements is required.

intermediateThrowEvent

The *intermediateThrowEvent* element represents a throwing intermediate event. Its attributes and child elements are the same as those of *intermediateCatchEvent*, with the following exceptions:

- There is no *parallelMultiple* attribute.
- The data flow-related child elements are *property*, *dataInput*, *dataInputAssociation*, and *inputSet*. These will be discussed in Chapter 16.
- The same set of *eventDefinition* elements is allowed by the XSD, but the only valid ones (per Figure 7-1) are *messageEventDefinition*, *signalEventDefinition*, *compensateEventDefinition*, *linkEventDefinition*, and *escalationEventDefinition*. More than one *eventDefinition* signifies a *throwing Multiple* event. Omission of *eventDefinition* signifies a *throwing None* event, which is allowed; it can be used in the diagram to indicate a particular state of the instance.

Link Event Bug

Link events are discussed in Chapter 7. The BPMN 2.0 specification contains a bug concerning their serialization. A Link event has *optional* child elements *source* and *target*, each a QName pointer to the other half of the Link event pair. Table 10.98 of the spec says that *name*, *source*, and *target* are all *required*. This is clearly incorrect, as *source* and *target* should be mutually exclusive for Link events; a single Link event cannot have both. However, the XSD says the string attribute *name* is *required*, meaning a *linkEventDefinition* element without the attribute *name* (type *string*) is schema-invalid.

It is bad practice to use a string type like *name* to link model elements together. I believe the original intent was to use *source* and *target* for this purpose, but something got messed up along the way. My recommendation – and this is what the itp commerce tool does, as well – is to populate the *name* attribute of a *linkEventDefinition* with the *id* of its paired Link event. In that case the *names* of the paired *linkEventDefinition* elements will not match each other; each points instead to the *id* of the paired Link event element. This unambiguously connects the Link pair, but it means the *label* of the Link events in the diagram, which should match, must be something other than the *name*. This is inconsistent with the rest of BPMN.

boundaryEvent

The *boundaryEvent* element indicates an interrupting or non-interrupting boundary event. Its attributes and child elements are the same as those of *intermediateCatchEvent*, with the following

exceptions:

- Required attribute *attachedToRef* points to the activity the event is attached to. This attribute is QName, although I do not believe it is possible for the referenced activity to be defined in another file.
- Optional Boolean attribute *cancelActivity* determines whether the event is interrupting (*true*) or non-interrupting (*false*). The default value is *true*, so omission of the attribute signifies interrupting.
- Again the XSD allows all *eventDefinition* elements, but the only valid ones, per Figure 7-1, are *messageEventDefinition*, *timerEventDefinition*, *errorEventDefinition* (interrupting only), *escalationEventDefinition*, *cancelEventDefinition* (interrupting only), *compensateEventDefinition*, *conditionalEventDefinition*, and *signalEventDefinition*. Since there is no None boundary event, at least one *eventDefinition* is required. More than one signifies a Multiple or Multiple-Parallel boundary event.

endEvent

The *endEvent* element indicates an end event. Its attributes and child elements are the same as those of *intermediateThrowEvent*, with the following exceptions:

- The XSD allows all *eventDefinition* elements, but the only valid ones, per Figure 7-1, are *messageEventDefinition*, *errorEventDefinition*, *escalationEventDefinition*, *cancelEventDefinition*, *compensateEventDefinition*, *terminateEventDefinition*, and *signalEventDefinition*. Omission of an *eventDefinition* signifies a None end event. More than one signifies a Multiple end event.

sequenceFlow

The element *sequenceFlow* is a member of the *flowElement* class and inherits its standard attributes and child elements. In addition, it has the following attributes and child elements:

- *sourceRef*, a required IDREF attribute pointing to the *flowNode* at the tail of the sequence flow.
- *targetRef*, a required IDREF attribute pointing to the *flowNode* at the head of the sequence flow.
- *isImmediate*, an optional Boolean attribute indicating whether or not the sequence flow transition occurs immediately upon completion of the *sourceRef* node. This is useful information, but as it is invisible in the diagram, Method and Style recommends omitting it. It has no default value.
- *conditionExpression*, an optional child element of type *tExpression*, discussed below. Presence of this element indicates that the sequence flow is *conditional*. This is allowed only if the *sourceRef* points to an activity or an exclusive, inclusive, or complex gateway. If the *sourceRef* node's *default* attribute points to this sequence flow, *conditionExpression* must be omitted.

Expressions

Conditional expressions on the gates of an exclusive or inclusive gateway or conditional sequence flow represent the most common use of the *tExpression* datatype resulting in a Boolean value. This datatype is also used for certain gateway join conditions, loop conditions, multi-instance activity completion conditions, and conditional events.

The spec says that in non-executable processes, *tExpression* is intended to define a condition or other expression in “natural language,” and is considered “underspecified.” For executable processes, modelers are supposed to use a subclass of *tExpression* called *tFormalExpression*, defining a computable expression in a specified expression language. Indication that the element is *tFormalExpression* is expressed by the attribute *xsi:type="tFormalExpression"* (see Figure 15-3).

In XSD terminology, *tExpression* is a complex type with *mixed content*. That means it has both direct text content and attributes and child elements. (Most XML datatypes have either direct content or attributes and child elements, but not both.) Presumably the direct content of *tExpression* is intended to hold the natural language text of the expression. However, the direct content of an element of type *tExpression* is not what is displayed in the diagram. What is displayed in the diagram is the *name* of the sequence flow, conditional event, or other object to which the expression applies. For that reason, in non-executable models, Method and Style recommends *not* using the content of *conditionExpression* to define natural-language sequence flow conditions, but instead using the *name* (label) of the sequence flow, possibly in combination with the *name* of the gateway.

Formal Expressions

The examples in the spec all use *tFormalExpression*, which extends *tExpression* with two additional optional attributes:

- *language*, a URL that indicates the expression language, if needed to override the default type language declared in *definitions*.
- *evaluatesToTypeRef*, a QName indicating the datatype of the expression output, such as xsd:boolean.

Usage of formal expressions is not clearly defined in the spec. The executable process example [21] shown below is clipped from OMG's non-normative *BPMN 2.0 by Example v1.0* document, and modified slightly:

```

<exclusiveGateway name="Result?" gatewayDirection="Diverging" id="_1-128" />
<sequenceFlow sourceRef="_1-128" targetRef="_1-252" name="2nd level issue" id="_1-402">
  <conditionExpression xsi:type="tFormalExpression" language="http://www.jcp.org/en/jsr/detail?id=245" evaluatesToTypeRef="xsd:boolean">
    ${getDataObject("TicketDataObject").status == "Open"}
  </conditionExpression>
</sequenceFlow>
<sequenceFlow sourceRef="_1-128" targetRef="_1-150" name="Issue resolved" id="_1-396">
  <conditionExpression xsi:type="tFormalExpression" language="http://www.jcp.org/en/jsr/detail?id=245" evaluatesToTypeRef="xsd:boolean">
    ${getDataObject("TicketDataObject").status == "Resolved"}
  </conditionExpression>
</sequenceFlow>

```

Figure 15-3. Serialization of gateway conditions using formal expressions

This fragment illustrates formal expressions defining the gate conditions on an exclusive gateway labeled *Result?* The *language* attribute points to the URL for the Java *Unified Expression Language (UEL)*, overriding the global *expressionLanguage* value in *definitions*. The *evaluatesToTypeRef* indicates a Boolean type in the default XSD type language. The direct content, with the \${ } format, is the formal expression itself. The BPMN XPath extension function *getDataObject* is probably unnecessary for non-XPath expression languages.

In the non-executable form of this process model, the process logic would be fully specified by the gateway label *Result?* in combination with the gate labels *2nd level issue* and *Issue resolved*. However, technically a *conditionExpression* should be present on each gate, either empty as in Figure 15-4 or with a natural language string generated from the gateway and gate labels, as in Figure 15-2.

```

<exclusiveGateway name="Result?" gatewayDirection="Diverging" id="_1-128" />
<sequenceFlow sourceRef="_1-128" targetRef="_1-252" name="2nd level issue" id="_1-402">
<conditionExpression />
</sequenceFlow>
<sequenceFlow sourceRef="_1-128" targetRef="_1-150" name="Issue resolved" id="_1-396">
<conditionExpression />
</sequenceFlow>

```

Figure 15-4. Serialization of non-executable gateway conditions

laneSet and lane

Both *process* and *subProcess* elements may contain *lanes*. Lanes are not purely graphical constructs but semantic elements in their own right. In traditional flowcharting, lanes signify the performer or owner of the activities – primarily human tasks – that they contain. BPMN 2.0 generalizes the concept to support *any* user-defined classification of flow nodes. In fact, a single BPMN model may define multiple classifications, called *laneSets*, for the same set of flow nodes. It is very rare to see multiple *laneSets* – I have not yet come across it in the wild – but the spec allows it. Even if there is only one, all *lane* elements must be enclosed in a *laneSet* element.

In BPMN 2.0, *laneSets* and *lanes* are specified at independently at each process level. Each *lane* element in a process level contains a list of *flowNodeRefs* – pointers to *flowNodes* contained in the lane. Sequence flows, data objects, and any other shapes that are not *flowNodes* should not be referenced by *flowNodeRef*.

BPMN 1.2 had a rule that lanes could not be drawn inside an expanded subprocess shape, but there is no such rule in BPMN 2.0. In BPMN 2.0, *lane* and *subProcess* are semantic elements, regardless of their graphical representation. It might be difficult for a *BPMN tool* to draw lanes inside an expanded subprocess shape, but that is a tool issue not a BPMN issue.

Within a single process level, it is possible to have lanes within lanes. More accurately, a *lane* element may itself contain a *childLaneSet*. The *childLaneSet* has the same datatype as a *laneSet*. It contains *lanes*, each with *flowNodeRefs* and possibly more *childLaneSets* in a recursively nested structure. If nested lanes are drawn in the process diagram, the serialization must use *childLaneSet*.

Artifacts

Artifact is BPMN's term for elements that provide visual annotation of the diagram but do not directly specify sequence flow or message flow behavior. There are two types of artifacts, both in the Analytic subclass: *textAnnotation*, user-defined text linked to an element with an *association* connector; and *group*. Artifacts may belong either to a *collaboration* or to a *process*, as discussed below.

textAnnotation

The content of a *textAnnotation* element is defined by its child element *text*. Assignment of a *textAnnotation* element to a *process* or *collaboration* is determined by the *association* linking it to a model node.

The *association* is non-directional, so its *associationDirection* attribute either has a value of *None* (the default) or is omitted. Attributes *sourceRef* and *targetRef* of association are remote (QName) references. One of them points to the *textAnnotation*. If the other points to a *flowElement*, both the *textAnnotation* and *association* belong to the *flowElement*'s *process*. Otherwise they belong to the *collaboration*.

group

Normally a *group* element just has an *id*, which is referenced by a shape in the graphical model. The Analytic subclass also includes the optional attribute *categoryValueRef*, a QName pointer to a root element *category/categoryValue*. However, *category* is not in the Analytic subclass and invisible in the diagram, so I believe *categoryValueRef* should not be in the Analytic subclass, either.

Chapter 16

Serializing Data Flow

Data and data flow are primarily concerns of executable BPMN, but *data object*, *data store*, and *data association* are members of the Analytic subclass and are sometimes used in non-executable process diagrams. Serializing these process elements properly in non-executable BPMN requires creating *dataInput*, *dataOutput*, and other related elements not listed in Figure 13-1. In this chapter we will discuss serialization of data flow in non-executable Level 2 BPMN. We will discuss data flow in executable BPMN in Chapter 20.

Non-Executable Data Flow

Figure 16-1 illustrates data flow in a non-executable model. The data object *Contract [unsigned]* represents data flow from the start event to task A. Task B updates the data store *Contracts database*. The dotted line connectors are *directional data associations*. In the diagram, the data associations appear to connect to the start event and task elements directly, but in the XML they actually connect to data inputs or outputs of those elements.

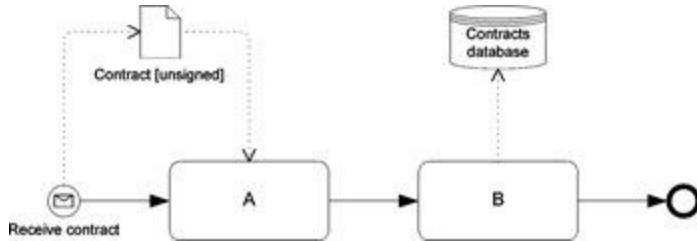


Figure 16-1. Non-executable data flow

dataObject

A data object in BPMN represents a *local instance variable*. It is visible only within the process level in which it is defined and its child levels, and the variable disappears when the process level instance is complete. In the BPMN 2.0 metamodel, *dataObject* is both a *flowElement*, meaning it has *id* and *name* and belongs to a *process*, and an *item-aware element*, meaning it points to an *itemDefinition*. BPMN 2.0 uses the term *item* to allow item-aware elements to describe not just data, i.e. information items, but physical items as well. Because its lifetime is limited to that of the process level instance, it is not likely that a *dataObject* represents a physical item, but it is theoretically possible.

The *itemDefinition*, a root element in the BPMN structure, is typically used only in executable models. In that case, the *dataObject*'s attribute *itemSubjectRef* points to an *itemDefinition*, which in turn points to a datatype, typically imported from an external XSD file. The optional attribute *isCollection* (default value false) indicates the *dataObject* represents an array of data elements. If *isCollection* is *true*, the data object shape carries the three-bar multi-instance marker.

An item-aware element also has optional child *dataState* with string attribute *name*. In Figure 16-1, *Contract* is the *dataObject* name, and *unsigned* is its *dataState* name. The label in the diagram is supposed to concatenate them, wrapping the *dataState* name, if any, in square brackets. Many tools simply make "*Contract [unsigned]*" the *dataObject* name, with no *dataState*; in non-executable BPMN, that is probably OK.

dataInput* and *dataOutput

During the drafting of BPMN 2.0, the technical committee argued for a while whether it would be acceptable for a data association to connect directly to an activity or event element, but in the end decided not to allow it. The BPMN metamodel says that the source and target of a data association must be an *item-aware element*. A *flowNode* element is not item-aware, but its *dataInputs* and *dataOutputs* are.*****ebook converter DEMO Watermarks*****

That means these child elements must be present in the XML in order to create a valid data association connection to an activity or event. That is unfortunate, since *dataInput* and *dataOutput* are not normally displayed in the diagram and are important only in executable models. Requiring them makes the serialization more verbose, but it is fairly straightforward for the implementer.

Tasks and processes have child element *ioSpecification*, which defines their input and output data requirements. The *ioSpecification* contains a list of *dataInput* and *dataOutput* elements, plus at least one *inputSet* pointing to needed *dataInputs* and at least one *outputSet* pointing to needed *dataOutputs*. *ioSpecification* is optional, but if included it must contain both *inputSet* and *outputSet*.

Events do not have *ioSpecification*, but they do have *dataInput* or *dataOutput*. If a non-executable process diagram depicts data flow to or from the event, the serialization must include these item-aware elements.

dataInputAssociation and dataOutputAssociation

A *data association* connector looks just like the regular *association* connector used with *textAnnotation*, except that data association is by default *directional*, drawn with an arrowhead. In the XML both its *sourceRef* and *targetRef* must point to an item-aware element. That means they do not point to a *task* or *event* element directly but to one of its *dataInput* or *dataOutput* elements. Actually, in the XSD there are separate elements for *dataInputAssociation* and *dataOutputAssociation*. The first connects from an item-aware element, such as *dataObject*, to a *dataInput*, and the second from a *dataOutput* to an item-aware element. In executable models, the *dataAssociation* may include a *mapping*.

dataStore and dataStoreReference

A *dataStore* is also an item-aware element, but, unlike *dataObject*, it is persistent and accessible from any process element. In the XSD it is defined as a *root element*, so it does not belong to a particular *process* or *subProcess*. However, data store *interactions* with a process element via data association are part of a particular process and must use the element *dataStoreReference*. *dataStoreReference* is a *flowElement*, i.e. , part of a *process*, that points to the *dataStore* global element. In executable models, the *dataStore* element points in turn to the *itemDefinition*.

Example: Non-Executable Data Flow

At this point it should be obvious that serializing data flow, even in non-executable BPMN, is verbose, involving multiple levels of indirection, and requires elements that are not visible in the diagram. For example, the serialization of the simple flow depicted in Figure 16-1 is shown below.

```
<definitions targetNamespace="http://www.itp-commerce.com" xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL" xmlns:itp="http://www.itp-commerce.com/BPMN2.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://www.omg.org/spec/BPMN/20100524/MODEL schemas/BPMN20.xsd" exporter="Process Modeler 5 for Microsoft Visio" exporterVersion="5.2742.13663 SR6" itp:name="My Diagram" itp:version="1.0" itp:author="bruce" itp:creationDate="8/5/2011 8:43:11 AM" itp:modificationDate="8/5/2011 10:17:52 AM" itp:createdWithVersion="5.2742.13663 SR6" itp:conformanceSubClass="full" id="a26428bb-9287-4346-b659-1d89f5d41217">
  <process id="5c311ebc4-4ae3-41aa-a2f5-a7802720c773" name="Main Process" processType="None">
    <startEvent id="c529a130-7805-4b9e-90b7-8d923e4813ca" name="Receive contract">
      <dataOutput id="do_c529a130-7805-4b9e-90b7-8d923e4813ca"/>
      <dataOutputAssociation id="5f837dfc-6686-4e1c-bb9e-67123e59cadf">
        <sourceRef>do_c529a130-7805-4b9e-90b7-8d923e4813ca</sourceRef>
        <targetRef>_37bfff1e7-a72c-434a-81b9-2873d11b8845</targetRef>
      </dataOutputAssociation>
      <messageEventDefinition/>
    </startEvent>
    <task id="f2509706-84ef-4f59-8fdb-5f25b3102686" name="A">
      <iOSpecification>
        <dataInput id="di_f2509706-84ef-4f59-8fdb-5f25b3102686"/>
        <inputSet>
          <dataInputRefs>di_f2509706-84ef-4f59-8fdb-5f25b3102686</dataInputRefs>
        </inputSet>
        <outputSet/>
      </iOSpecification>
      <dataInputAssociation id="985c2eb0-3265-4f13-a295-e29778b1c973">
        <sourceRef>_37bfff1e7-a72c-434a-81b9-2873d11b8845</sourceRef>
        <targetRef>di_f2509706-84ef-4f59-8fdb-5f25b3102686</targetRef>
      </dataInputAssociation>
    </task>
    <task id="63b74f88-2f16-4808-a953-4a082d28bdb3" name="B">
      <iOSpecification>
        <dataOutput id="do_63b74f88-2f16-4808-a953-4a082d28bdb3"/>
        <inputSet/>
        <outputSet>
          <dataOutputRefs>do_63b74f88-2f16-4808-a953-4a082d28bdb3</dataOutputRefs>
        </outputSet>
      </iOSpecification>
      <dataOutputAssociation id="a9af7e2-fe6e-41b7-9a7b-6ba39d2f63c8">
        <sourceRef>do_63b74f88-2f16-4808-a953-4a082d28bdb3</sourceRef>
        <targetRef>_474935d1-d1bf4244-b5b2-3a3bffa9a4d5</targetRef>
      </dataOutputAssociation>
    </task>
    <endEvent id="846d6306-9380-4e56-aae7-532d1ef96fc5"/>
    <dataObject id="37bfff1e7-a72c-434a-81b9-2873d11b8845" name="Contract [unsigned]" />
    <sequenceFlow id="88c3ac5d-877d-465e-9669-c7fb2443105" sourceRef="c529a130-7805-4b9e-90b7-8d923e4813ca" targetRef="f2509706-84ef-4f59-8fdb-5f25b3102686"/>
    <sequenceFlow id="589e46f9-5213-49fd-8050-4649e6368c1" sourceRef="f2509706-84ef-4f59-8fdb-5f25b3102686" targetRef="63b74f88-2f16-4808-a953-4a082d28bdb3"/>
    <sequenceFlow id="f2d060d3-2725-436b-99e3-6a2169b96365" sourceRef="63b74f88-2f16-4808-a953-4a082d28bdb3" targetRef="846d6306-9380-4e56-aae7-532d1ef96fc5"/>
    <dataStoreReference id="474935d1-d1bf4244-b5b2-3a3bffa9a4d5" name="Contracts database" dataStoreRef="a3b16297-1657-497d-ab57-0f64e38f27a3"/>
  </process>
  <dataStore id="a3b16297-1657-497d-ab57-0f64e38f27a3" name="Contracts database"/>
</definitions>
```

Figure 16-2. Serialization of non-executable data flow

Note the following things from Figure 16-2:

- Out of the start event, the *sourceRef* of the *dataOutputAssociation* is not the *startEvent* itself but its *dataOutput* element. This element must be generated by the tool.
- In Task A, the *targetRef* of the *dataInputAssociation* is not the *task* itself but its *dataInput* element, child of *iOSpecification*. Even though there is only one *dataInput* defined for this task, the XSD requires *inputSet* referencing the *dataInput*. Even though there is no *dataOutput*, the XSD requires an empty *outputSet* element. Similar considerations apply to Task B.
- The *targetRef* of the *dataOutputAssociation* from Task B is the *dataStoreReference*, defined within the *process*, not the *dataStore*, which is a root element. The data store shape in the diagram must generate both the *dataStore* and the *dataStoreReference* elements in the serialization.

- In an executable model, the *dataStore* and *dataObject* elements would have pointers to *itemDefinition* root elements, which in turn would point to their datatype definition. We will revisit this in Chapter 20.

More on Data Inputs and Data Outputs

There remains some disagreement within the BPMN expert community – I would say within the BPMN 2.0 technical committee itself – about certain aspects of *dataInput* and *dataOutput*. The spec is ambiguous on the matter. There was a flurry of debate about it for a while, and in the end the only thing everyone could agree on was that the spec had made a mess of it that would need fixing in some future BPMN 2.1. The following is my view of it.

The specification clearly says *dataInput* and *dataOutput* describe the *data requirements*, or *interface*, of a task or process. That is in contrast to a *dataObject*, which represents a *stored data value*, or *variable*. The value of a *dataObject* is stored and may be mapped and communicated via *dataInputAssociation* to any process element, such as a task, in the same process level (or one of its child levels). On the other hand, a populated *dataInput* value, when received, is used *immediately* by the element in which it is defined; it is not stored for mapping and communication to other elements in the process level.

In the semantic model, the spec says, “Data Inputs MAY have *incoming Data Associations*” [italics in original]. We have already seen in Figure 16-2 examples of *dataInputAssociation* targeting a *dataInput*. The spec does *not* say a data input may have *outgoing* data associations but it does not explicitly rule it out. This is at the heart of the controversy over the meaning and use of a *dataInput* of a *process*. Some people say maintain that the *dataInput* of a *process* is not simply an interface but also a stored input variable, just like a *dataObject*. Their justification is that the metamodel and XSD allow a data association to link *any* pair of item-aware elements, and you can see examples of this serialization in the non-normative *BPMN 2.0 by Example* document on the OMG website [\[22\]](#).

But I disagree with that view. In fact, the spec is full of cases where the narrative disallows constructs that are schema-valid. My view is that a *dataInput* is an interface only, not a stored value, and may *only* have incoming data associations, not outgoing data associations. Further evidence of this for a *process* *dataInput* is the fact that its incoming data associations may not come from inside the process, only from outside.

subProcess *dataInput* and *dataOutput* are confusing as well. According to the spec narrative, a *subProcess* may not have *dataInput* or *dataOutput*, even though the XSD allows it. However, this makes little sense, as there is no apparent difference between the input data requirements of a subprocess and a task. And if *Task A* in Figure 16-1 were *Subprocess A* instead, there would be no way to serialize the data flow! I suggest ignoring this statement of the spec narrative.

dataInput and *dataOutput* can be represented visually in the diagram. The shape looks like a data object with a white arrow (*dataInput*) or black arrow (*dataOutput*) arrow icon inside (Figure 16-3). However, the spec appears to limit this graphical representation to a *dataInput* or *dataOutput* of a *process*, either a top-level process or a called process, i.e., a *callActivity* to a reusable process. The *dataInput* of a task is not supposed to be displayed graphically.



Figure 16-3. Data input and Data output

Also, the spec suggests that the only time a *dataInputAssociation* to a *process* *dataInput* would be displayed graphically is in a called process represented as an *expanded call activity* shape. And even in that case, the semantic element containing the *dataInput* should be the *callActivity*, not the called *process*.

In short, the spec is a mess when it comes to *dataInput* and *dataOutput* of a *process*. Fortunately, to serialize data flow depicted in the diagram, implementers only need concern themselves with the data inputs and outputs of activities and events, where the rules are clearer.

Chapter 17

The BPMNDI Graphical Model

So far we have covered only serialization of the BPMN *semantic model*. BPMN 2.0 also provides an XML schema for the *graphical model*, called *BPMN Diagram Interchange*, or *BPMNDI*. It describes the location and size of shapes and connectors, as well as the linked page structure of the model diagrams.

A proper XML schema for BPMNDI did not exist until the Finalization phase of BPMN 2.0. In the beta specification, OMG's push for a single metamodel supporting *any* diagram type, including both UML and BPMN, prevented definition of a proper XSD for the BPMN graphical model. However, that universal graphical metamodel, which can still be seen in Appendix B of the BPMN 2.0 spec, made XML interchange of BPMN diagrams between modeling tools impractical, if not impossible. BPMNDI, as defined in the final version of the spec, not only makes BPMN model interchange possible but allows individual BPMN models to specify their page structure in a usable way.

In BPMN, the graphical model can never stand alone. It *must* be accompanied by the semantic model information. For example, the only way BPMNDI distinguishes a task shape from a Timer boundary event, or a sequence flow from a data association, is via the shape's *bpmnElement* attribute, a pointer to the corresponding semantic element. Information is not duplicated between the BPMNDI and the semantic model, but instead is split between them. For example, the *text* of a shape label is in the semantic model; the label's *position* and *font* information is in BPMNDI.

It is rare that one BPMN tool will be able to *exactly* reproduce the graphical layout created and serialized by another tool. Most tools have their own graphics libraries that constrain the size, aspect ratio, and label position of each shape. On import, such a tool cannot arbitrarily scale the shapes to match the original diagram exactly, but BPMNDI allows them at least to approximate the layout. Also, BPMNDI reveals the page structure of the original model – for example, whether a child process level is drawn inside an expanded subprocess shape or in a separate hyperlinked diagram.

BPMNDI Basics

BPMNDI does not use the BPMN 2.0 namespace. It has its own namespace, or rather its own three namespaces. The principal one, <http://www.omg.org/spec/BPMN/20100524/DI> (usually assigned the prefix *bpmndi*) is used for most BPMNDI elements, including the top-level *BPMNDiagram* element. But the *dc:Bounds* element describing the location and size of a shape uses a second namespace, <http://www.omg.org/spec/DD/20100524/DC> (prefixed *dc*), and the *di:waypoint* element describing the bendpoints of a connector uses a third one, <http://www.omg.org/spec/DD/20100524/DI> (prefixed *di*).

The graphical model comprises multiple pages, or diagrams, each serialized in BPMNDI with a *BPMNDiagram* and its child *BPMNPlane* element. (The purpose of using both *BPMNDiagram* and *BPMNPlane* elements is unclear, since every *BPMNDiagram* element MUST have exactly one child *BPMNPlane*.) Unlike a page in Visio, a diagram in BPMNDI has no *page size*; it is semi-infinite in extent. The origin of the coordinate system is the top left corner, and the page extends to infinity in the x and y directions. Negative coordinates are not allowed.

The *location* of a shape is defined as the x,y coordinates of the top left corner of a rectangular bounding box enclosing the shape. The *size* of a shape is likewise the width and height of that bounding box. Tools do not necessarily employ the BPMN coordinate system internally. For example, Visio's native coordinate system has the origin at the bottom left corner of the page and defines shape locations as the center of the rectangular bounding box.

The implementer must convert between the tool's native coordinates and BPMN coordinates upon export or import.

Each page contains a list of two-dimensional shapes (*BPMNShape*) and connectors (*BPMNEdge*), with location and size information for each. The specific shape or connector represented is defined by attribute *bpmnElement*, a remote (QName) pointer to the id of a BPMN semantic element. This attribute is effectively *required* for all shapes and connectors, even though that requirement is not enforced by the XSD.

Process Levels and Pages

A BPMN model may include multiple *process* elements. Each represents a *top-level BPMN process*, which can either stand alone or be invoked by a *callActivity* as a reusable subprocess. Each *process* is a hierarchical structure comprising multiple *process levels*. Each process level is enclosed in the XML structure by a *subprocess-type element*, either a *subProcess*, *adHocSubProcess*, *transaction*, or *callActivity*. The process level that includes the subprocess-type element itself is the *parent* of the process level that includes its expanded activity flow.

The child-level flow may be represented graphically either on the *same page* as the parent level, enclosed in an *expanded subprocess shape*, or on a *separate linked page*, using a *collapsed subprocess shape* on the parent-level page and no subprocess shape on the child-level page. In the Method and Style section of this book, we called the former the *inline modeling style* and the latter the *hierarchical modeling style*, and we indicated a preference for the hierarchical style. Again we need to emphasize that this choice of graphical representation is specified *only in BPMNDI*; it is not part of the semantic model.

Here is how BPMNDI makes that distinction. Of critical importance is the *bpmnElement* attribute of *BPMNPlane*, that is, the semantic element referenced by the *page* in the graphical model. This is a *required QName* pointer to a *process*, *collaboration*, or subprocess-type element (*subProcess*, *callActivity*, *transaction*, or *adHocSubProcess*). These pointers for each *BPMNPlane* describe the *page structure* of the graphical model. A *BPMNPlane* that points to a subprocess-type semantic element is, by definition, a *child-level page*. Any other page is, by definition, a *top-level page*.

Internal consistency demands that a *child-level page* may contain only flow elements belonging to the referenced subprocess-type element. It may not, for example, also include elements of a separate top-level *process* ("Process2") enclosed in a pool shape on the page. If it could, there would be no link between that page in BPMNDI and *Process2* in the semantic model.

On the other hand, a *top-level page* may contain elements belonging to more than one *process*, as long as shapes belonging to at most one *process* are not enclosed in a pool shape. If a top-level page contains flow elements of more than one *process*, the *bpmnElement* attribute of its *BPMNPlane* should point to a *collaboration*. If it contains elements of a single *process* only, *bpmnElement* should point either to a *process* or *collaboration*.

BPMNDiagram

The top-level element in BPMNDI is *BPMNDiagram*, representing a page. A model may have any number of *BPMNDiagram* elements. A semantic-only BPMN model, by definition, is one with no *BPMNDiagram* elements. Each *BPMNDiagram* has a required child element *BPMNPlane*, which serves as the container for the shapes and connectors on the page. A *BPMNPlane* is not like a layer in Visio or Autocad; a *BPMNDiagram* may not have more than one. Thus there is no apparent reason why a separate *BPMNPlane* element is necessary; it's just the way the XSD works.

Both *BPMNDiagram* and *BPMNPlane* effectively stand for the page as a whole. Both elements have an *id*, but only *BPMNDiagram* has a *name*.

- The *name* attribute of *BPMNDiagram* should contain the name of the page in the BPMN tool.
- The *resolution* attribute of *BPMNDiagram* is a number defining the scale in *pixels per inch*. This attribute is needed to convert BPMNDI location and size values in pixels to lengths on the page. For some strange reason, there is nothing in BPMNDI that allows an alternative scale unit such as pixels per cm.
- Unlike *documentation* in the semantic model, which is an element, *documentation* for *BPMNDiagram* is an attribute.

In addition to the required child element *BPMNPlane*, *BPMNDiagram* has child *BPMNLabelStyle* (optional, unbounded), which specifies font styles used in labels on the page.

BPMNPlane

BPMNPlane contains an ordered list of *BPMNShape* and *BPMNEdge* child elements representing the shapes and connectors on the page. The order of the shapes and edges inside a *BPMNPlane* determines their *Z-order*, from back to front.

The attribute *bpmnElement* of a *BPMNPlane* defines the page as either top-level or child-level, as discussed previously. If it points to a subprocess-type element, it is a child-level page. Otherwise it is a top-level page.

BPMNShape

The *BPMNShape* element represents the visualization of a single BPMN semantic element other than a connector.

- Attribute *bpmnElement* is a QName pointer to a BPMN semantic element. It is the only indication of the type of shape represented. The spec narrative says this attribute is *required*, although that is not enforced by the XSD.
- Optional Boolean attribute *isHorizontal* applies to *pool* and *lane* shapes only. There is no default value. A pool shape is one for which *bpmnElement* points to a *participant* in the semantic model.
- Optional Boolean attribute *isExpanded* applies to subprocess-type elements only (*subProcess*, *transaction*, *adHocSubProcess*, or *callActivity*). There is no default value. A *BPMNShape* element with attribute *bpmnElement* pointing to a subprocess-type element represents a *collapsed subprocess shape* if attribute *isExpanded* equal to *false*, and an *expanded subprocess shape* if attribute *isExpanded* is *true*.
- Optional Boolean attribute *isMarkerVisible* applies only to *exclusiveGateway* elements. There is no default value. A *true* value indicates the X symbol is displayed inside the gateway diamond.
- *dc:Bounds* is a *required* child element defining the location and size coordinates of a rectangular bounding box surrounding the shape. To convert to inches, divide the *dc:Bounds* coordinate values by the *BPMNDiagram* attribute *resolution*. Location coordinates *x* and *y* are *required*, type *xsd:double*, and locate the top left corner of the bounding box. Size coordinates *width* and *height* are also *required*, type *xsd:double*.
- Optional child element *BPMNLabel* is used to define location and font style of diagram labels. The label text is defined by the corresponding semantic element. Attribute *labelStyle* is a QName pointer to *BPMNLabelStyle* child of *BPMNDiagram*. Child element *dc:Bounds* defines the label location and size.

In the graphical model, a *BPMNShape* element may not contain another *BPMNShape* element, even if one shape is drawn inside the other. For example, a task shape may be drawn inside a pool shape, but their *BPMNShape* elements are siblings, children of the same *BPMNPlane* element.

BPMNEdge

A *BPMNEdge* element is the graphical representation of a single BPMN *connector*.

- Attribute *bpmnElement* is a QName pointer to a semantic connector element. It is the only indication of the type of connector represented. The spec narrative says it is *required*, although that is not enforced by the XSD.
- Optional QName attributes *sourceElement* and *targetElement* are pointers to BPMNDI elements. The spec says these are only to be used when those shapes are NOT the same as those whose *bpmnElement* references point to the semantic connector's *sourceRef* and *targetRef* elements. An example might be the "visual shortcut" linking a data object to a sequence flow in the diagram.
- Optional attribute *messageVisibleKind* (enumerated values *initiating*, *non-initiating*) applies only to message flows referencing a message. It should be used only if the *message* symbol is displayed on the message flow. A value of *initiating* should be displayed with white envelope, *non-initiating* with shaded envelope.
- Required child element *di:waypoint* is an ordered list of x,y coordinates from the source to the target of the connector. At least two *di:waypoint* elements are required for each *BPMNEdge*. Waypoints between the first and last are bendpoints of the connector. Each *di:waypoint* has required child elements *x* and *y*, type *xsd:double*.
- Optional child element *BPMNLabel* is the same as in BPMNShape.

BPMNDI Examples

Figure 17-1 illustrates a simple BPMN process. The serialization, including BPMNDI, is shown in Figure 17-2.

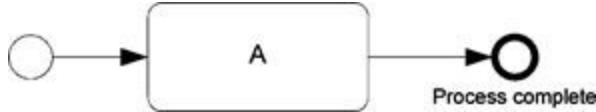


Figure 17-1. A simple process model

```
<definitions targetNamespace="http://www.itp-commerce.com"
  xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"           xmlns:itp="http://www.itp-commerce.com/BPMN2.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"           xsi:schemaLocation="http://www.omg.org/spec/BPMN/20100524/MODEL
  exporter="Process Modeler 5 for Microsoft Visio" exporterVersion="5.2742.13663 SR6" itp:name="My Diagram"
  itp:version="1.0" itp:author="bruce" itp:creationDate="8/11/2011 3:26:19 PM" itp:modificationDate="8/11/2011
  3:27:51 PM" itp:createdWithVersion="5.2742.13663 SR6" itp:conformanceSubClass="Full" id="_a84f7a92-b55d-4de1-a18f-901ae69cfce7">
  <process id="_9c6890e1-cb48-4996-bbcc-93f7932018d8" name="Main Process" processType="None">
    <startEvent id="_3d4ea3bc-62fe-4db2-af78-565fff63f442"/>
    <task id="_a904e6fa-2864-4c6f-9bf3-806387908aaf" name="A"/>
    <endEvent id="_cbd876c6-f3a7-4ed5-a27d-48e75d5ced83" name="Process complete"/>
    <sequenceFlow id="_6ef08698-2d78-4357-a843-08eebc32b64d" sourceRef="_3d4ea3bc-62fe-4db2-af78-565fff63f442" targetRef="_a904e6fa-2864-4c6f-9bf3-806387908aaf"/>
    <sequenceFlow id="_413a3714-a3dd-4cc2-8bbe-1f6c9448b7ec" sourceRef="_a904e6fa-2864-4c6f-9bf3-806387908aaf" targetRef="_cbd876c6-f3a7-4ed5-a27d-48e75d5ced83"/>
  </process>
  <bpmndi:BPMNDiagram name="My Diagram" (1)" resolution="72"
    xmlns:bpmndi="http://www.omg.org/spec/BPMN/20100524/DI">
    <bpmndi:BPMNPlane id="_1" bpmnElement="_9c6890e1-cb48-4996-bbcc-93f7932018d8">
      <bpmndi:BPMNShape id="_8A224598-E150-4114-8679-BB572A629081" bpmnElement="_3d4ea3bc-62fe-4db2-af78-565fff63f442" itp:label="(unnamed)" itp:elementType="startEvent">
        <dc:Bounds x="209.763779527559" y="232.44094488189" width="17.007874015748"
          height="17.007874015748" xmlns:dc="http://www.omg.org/spec/DD/20100524/DC"/>
      </bpmndi:BPMNShape>
      <bpmndi:BPMNShape id="_2A8B7F30-0E05-44B0-A282-B93B71A197AF" bpmnElement="_a904e6fa-2864-4c6f-9bf3-806387908aaf" itp:label="A" itp:elementType="task">
        <dc:Bounds x="263.622047244095" y="219.685039370079" width="85.0393700787402"
          height="42.5196850393701" xmlns:dc="http://www.omg.org/spec/DD/20100524/DC"/>
      </bpmndi:BPMNShape>
      <bpmndi:BPMNShape id="_D37840EE-7D36-4296-98D7-14A30BE6E5AE" bpmnElement="_cbd876c6-f3a7-4ed5-a27d-48e75d5ced83" itp:label="Process complete" itp:elementType="endEvent">
        <dc:Bounds x="396.850393700787" y="232.44094488189" width="17.007874015748"
          height="17.007874015748" xmlns:dc="http://www.omg.org/spec/DD/20100524/DC"/>
      </bpmndi:BPMNShape>
      <bpmndi:BPMNEdge id="_AE5164CB-05B3-428F-94B6-BA3B272D7F75" bpmnElement="_6ef08698-2d78-4357-a843-08eebc32b64d" itp:label="(unnamed)" itp:elementType="sequenceFlow">
        <di:waypoint x="226.771653543307" y="233.858267716535"
          xmlns:di="http://www.omg.org/spec/DD/20100524/DI"/>
        <di:waypoint x="263.622047244095" y="233.858267716535"/>
    </bpmndi:BPMNEdge>
  </bpmndi:BPMNDiagram>

```

*****ebook converter DEMO Watermarks*****

```

xmlns:di="http://www.omg.org/spec/DD/20100524/DI"/>
  </bpmndi:BPMEEdge>
    <bpmndi:BPMEEdge id="_894F23B9-22F7-40B7-B767-96B01D48C677" bpmnElement="_413a3714-a3dd-4cc2-8bbe-1f6c9448b7ec" itp:label="(unnamed)" itp:elementType="sequenceFlow" >
      <di:waypoint x="348.661417322835" y="233.858267716535" />
    </bpmndi:BPMEEdge>
  </bpmndi:BPMPNPlane>
</bpmndi:BPMDiagram>
</definitions>

```

Figure 17-2. Serialization of simple process model, including BPMNDI

Note the following about Figure 17-2:

- The *name* attribute of *BPMDiagram* corresponds to the page name in Visio.
- The *bpmnElement* attribute of *BPMPNPlane* points to the process, indicating a top-level page.
- The *di* and *dc* namespaces were not declared in *definitions*, but instead declared in each BPMNDI element where used. This is allowed but results in verbose XML.
- Private attributes in the *itp* namespace identifying the shape type and label are there for the tool's own use. Such extensions are allowed by the XSD but are not required for model interchange.

Figure 17-3 shows the top-level diagram of a hierarchical model; Figure 17-4 shows the child-level expansion of *Process order*. The serialization, including BPMNDI, is shown in Figure 17-5.

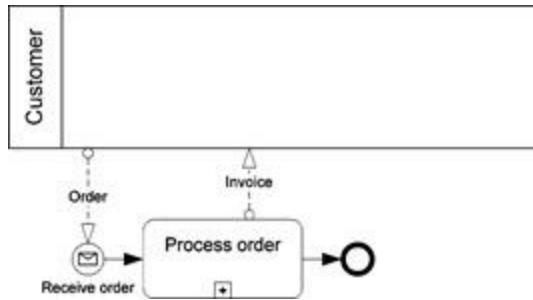


Figure 17-3. Simple hierarchical model, top level

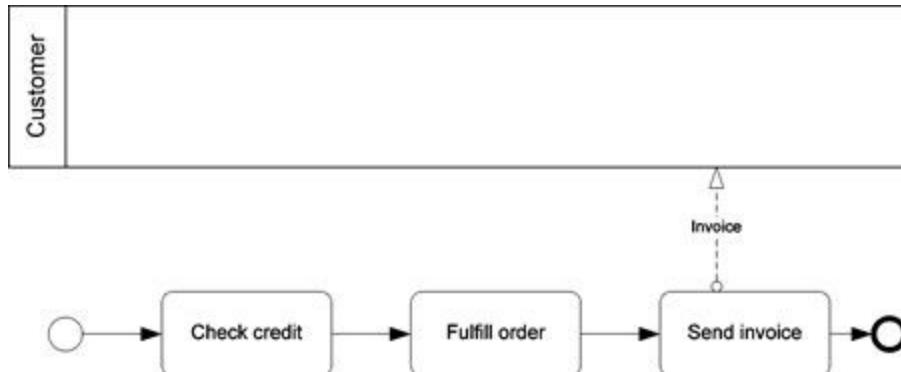


Figure 17-4. Simple hierarchical model, child level

```

<definitions
  targetNamespace="http://www.itp-commerce.com"
  xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"           xmlns:itp="http://www.itp-
  commerce.com/BPMN2.0"           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.omg.org/spec/BPMN/20100524/MODEL           schemas/BPMN20.xsd"
  exporter="Process Modeler 5 for Microsoft Visio" exporterVersion="5.2742.13663 SR6" itp:name="My Diagram"
  itp:version="1.0" itp:author="bruce" itp:creationDate="8/11/2011 5:27:45 PM" itp:modificationDate="8/11/2011
  5:38:23 PM" itp:createdWithVersion="5.2742.13663 SR6" itp:conformanceSubClass="Full" id="_f008c590-be03-
  4ed9-8923-a3c80b07121c">
  <process id="_ab4160fa-a43a-40bb-8c7e-b26919f97deb" name="Main Process" processType="None">
    <startEvent id="_75454980-4128-4083-95d7-0ef85b52ecba" name="Receive order">
      <messageEventDefinition/>
    </startEvent>
    <subProcess id="_31f4992c-a912-4828-b67b-3c430d841189" name="Process order" itp:isCollapsed="true"
    itp:logicalSheetId="f5203bf0-5def-4d91-91ae-1e25ae5e4403">
      <startEvent id="_7883fb8-a2c9-469f-8084-2bab5e877326"/>
      <task id="_88e44689-8fed-476a-b4bc-42e894c23fab" name="Check credit"/>
      <task id="_817bf4cf-4ede-407c-8624-da8dc56d78c4" name="Fulfill order"/>
      <task id="_bd2d2300-de55-4aa0-baf3-f43398a36666" name="Send invoice"/>
      <endEvent id="_da813ae8-7300-4d0c-8cb3-032a84f4d77f"/>
      <sequenceFlow id="_f544e6ce-dbfe-4e01-a942-581ea7a76d17" sourceRef="_bd2d2300-de55-4aa0-baf3-
      f43398a36666" targetRef="_da813ae8-7300-4d0c-8cb3-032a84f4d77f"/>
      <sequenceFlow id="_5c3838c3-1fa1-45b2-a4c6-a0f66d592f3f" sourceRef="_7883fb8-a2c9-469f-8084-
      2bab5e877326" targetRef="_88e44689-8fed-476a-b4bc-42e894c23fab"/>
      <sequenceFlow id="_6a7a06bb-bb53-4861-818c-8bb0f7a2a942" sourceRef="_88e44689-8fed-476a-b4bc-
      42e894c23fab" targetRef="_817bf4cf-4ede-407c-8624-da8dc56d78c4"/>
      <sequenceFlow id="_fecb259c-2083-4d9f-919b-bec391354605" sourceRef="_817bf4cf-4ede-407c-8624-
      da8dc56d78c4" targetRef="_bd2d2300-de55-4aa0-baf3-f43398a36666"/>
    </subProcess>
    <endEvent id="_5d9f3cba-4787-4420-b1b3-c7666f8a837d"/>
    <sequenceFlow id="_cf8bf2c6-c959-45f4-93e2-cdce3175850e" sourceRef="_75454980-4128-4083-95d7-
    0ef85b52ecba" targetRef="_31f4992c-a912-4828-b67b-3c430d841189"/>
    <sequenceFlow id="_8dadf786-45a3-4594-a56a-02375207af8" sourceRef="_31f4992c-a912-4828-b67b-
    3c430d841189" targetRef="_5d9f3cba-4787-4420-b1b3-c7666f8a837d"/>
  </process>
  <collaboration id="_7fe9461f-f0b3-4beb-a664-b4034c8cf4da">
    <participant id="_357f89aa-eb8f-4014-9548-0928d47192a7" name="Customer"/>
    <participant id="p_ab4160fa-a43a-40bb-8c7e-b26919f97deb" name="Main Process"
    processRef="ab4160fa-a43a-40bb-8c7e-b26919f97deb"/>
    <messageFlow id="_36ae2e66-cbfe-451a-9a7b-52539da0702b" name="Invoice" sourceRef="_bd2d2300-
    de55-4aa0-baf3-f43398a36666" targetRef="_357f89aa-eb8f-4014-9548-0928d47192a7"/>
    <messageFlow id="_daf3f8e0-f6f5-491a-b04f-aa54caf62a39" name="Invoice" sourceRef="_31f4992c-a912-
    4828-b67b-3c430d841189" targetRef="_357f89aa-eb8f-4014-9548-0928d47192a7"/>
  </collaboration>
  <bpmndi:BPMNDiagram name="My Diagram" (1)" resolution="72"
  xmlns:bpmndi="http://www.omg.org/spec/BPMN/20100524/DI">
    <bpmndi:BPMNPlane id="_1">
      <bpmndi:BPMNShape id="_182647EF-1058-488D-9888-45945045C623" bpmnElement="_75454980-4128-
      4083-95d7-0ef85b52ecba">
        <dc:Bounds x="90.7086614173228" y="226.771653543307" width="17.007874015748"
        height="17.007874015748" xmlns:dc="http://www.omg.org/spec/DD/20100524/DC"/>
      </bpmndi:BPMNShape>
    *****
  
```

*****ebook converter DEMO Watermarks*****

```

<bpmndi:BPMNShape id="_126B2C02-E3E6-4E39-B9FF-2F33B3AA3004" bpmnElement="_31f4992c-a912-4828-b67b-3c430d841189" isExpanded="false">
    <dc:Bounds x="128.976377952756" y="214.015748031496" width="85.0393700787402" height="42.5196850393701" xmlns:dc="http://www.omg.org/spec/DD/20100524/DC"/>
</bpmndi:BPMNShape>
<bpmndi:BPMNShape id="_614D8B4E-41C4-4055-9234-0601C778626F" bpmnElement="_5d9f3cba-4787-4420-b1b3-c7666f8a837d">
    <dc:Bounds x="235.275590551181" y="226.771653543307" width="17.007874015748" height="17.007874015748" xmlns:dc="http://www.omg.org/spec/DD/20100524/DC"/>
</bpmndi:BPMNShape>
<bpmndi:BPMEEdge id="_153BA773-386A-417A-83CC-729D21CAEFB7" bpmnElement="_cf8bf2c6-c959-45f4-93e2-cdce3175850e" sourceElement="_75454980-4128-4083-95d7-0ef85b52ecba" targetElement="_31f4992c-a912-4828-b67b-3c430d841189">
    <di:waypoint x="107.716535433071" y="228.188976377953" xmlns:di="http://www.omg.org/spec/DD/20100524/DI"/>
    <di:waypoint x="128.976377952756" y="228.188976377953" xmlns:di="http://www.omg.org/spec/DD/20100524/DI"/>
</bpmndi:BPMEEdge>
<bpmndi:BPMEEdge id="_FE03106D-C683-446A-8098-62E1C78F6D92" bpmnElement="_8dadf786-45a3-4594-a56a-02375207af8" sourceElement="_31f4992c-a912-4828-b67b-3c430d841189" targetElement="_5d9f3cba-4787-4420-b1b3-c7666f8a837d">
    <di:waypoint x="214.015748031496" y="228.188976377953" xmlns:di="http://www.omg.org/spec/DD/20100524/DI"/>
    <di:waypoint x="235.275590551181" y="228.188976377953" xmlns:di="http://www.omg.org/spec/DD/20100524/DI"/>
</bpmndi:BPMEEdge>
<bpmndi:BPMEEdge id="_F56D4F86-0E68-4850-8B01-4CBEB99E2EEF" bpmnElement="_8081ac38-cf1f-4114-9efe-3892f7c3e2db" sourceElement="_357f89aa-eb8f-4014-9548-0928d47192a7" targetElement="_75454980-4128-4083-95d7-0ef85b52ecba">
    <di:waypoint x="106.299200433446" y="175.748031496063" xmlns:di="http://www.omg.org/spec/DD/20100524/DI"/>
    <di:waypoint x="106.299203136775" y="226.771653543307" xmlns:di="http://www.omg.org/spec/DD/20100524/DI"/>
</bpmndi:BPMEEdge>
<bpmndi:BPMNShape id="_6686DC43-2536-4761-8626-75633D08A530" bpmnElement="_357f89aa-eb8f-4014-9548-0928d47192a7" isHorizontal="false">
    <dc:Bounds x="85.2698558897484" y="99.2125984251969" width="254.887635238527" height="76.5354330708661" xmlns:dc="http://www.omg.org/spec/DD/20100524/DC"/>
</bpmndi:BPMNShape>
<bpmndi:BPMEEdge id="_6686B787-328D-4A2A-9591-3374E546F057" bpmnElement="_daf3f8e0-f6f5-491a-b04f-aa54caf62a39" sourceElement="_31f4992c-a912-4828-b67b-3c430d841189" targetElement="_357f89aa-eb8f-4014-9548-0928d47192a7">
    <di:waypoint x="178.582681220347" y="214.015748031496" xmlns:di="http://www.omg.org/spec/DD/20100524/DI"/>
    <di:waypoint x="178.582673110361" y="175.748031496063" xmlns:di="http://www.omg.org/spec/DD/20100524/DI"/>
</bpmndi:BPMEEdge>
</bpmndi:BPMNPlane>
</bpmndi:BMNDiagram>
<bpmndi:BPMNDiagram name="Process" order="1" resolution="72" xmlns:bpmndi="http://www.omg.org/spec/BPMN/20100524/DI">

```

*****ebook converter DEMO Watermarks*****

```

<bpmndi:BPMNPlane id="_2" bpmnElement="_31f4992c-a912-4828-b67b-3c430d841189">
  <bpmndi:BPMNShape id="_E655BE81-98FE-43CE-AFA3-D2B6979C9117" bpmnElement="_7883fbf8-a2c9-469f-8084-2bab5e877326">
    <dc:Bounds x="99.2125984251969" y="252.283464566929" width="17.007874015748" height="17.007874015748" xmlns:dc="http://www.omg.org/spec/DD/20100524/DC"/>
  </bpmndi:BPMNShape>
  <bpmndi:BPMNShape id="_A665A671-D32B-4D77-B38C-BD1D1E070FC9" bpmnElement="_88e44689-8fed-476a-b4bc-42e894c23fab">
    <dc:Bounds x="155.905511811024" y="239.527559055118" width="85.0393700787402" height="42.5196850393701" xmlns:dc="http://www.omg.org/spec/DD/20100524/DC"/>
  </bpmndi:BPMNShape>
  <bpmndi:BPMNShape id="_DF74DAE6-A83A-47F4-82D8-A8C86FE70CEB" bpmnElement="_817bf4cf-4ede-407c-8624-da8dc56d78c4">
    <dc:Bounds x="280.629921259843" y="239.527559055118" width="85.0393700787402" height="42.5196850393701" xmlns:dc="http://www.omg.org/spec/DD/20100524/DC"/>
  </bpmndi:BPMNShape>
  <bpmndi:BPMNShape id="_C070AC5F-8CBF-4554-B705-C4092157AA28" bpmnElement="_bd2d2300-de55-4aa0-baf3-f43398a36666">
    <dc:Bounds x="405.354330708661" y="239.527559055118" width="85.0393700787402" height="42.5196850393701" xmlns:dc="http://www.omg.org/spec/DD/20100524/DC"/>
  </bpmndi:BPMNShape>
  <bpmndi:BPMNShape id="_F7B71203-FF20-4EE5-801E-62568945CDB8" bpmnElement="_da813ae8-7300-4d0c-8cb3-032a84f4d77f">
    <dc:Bounds x="511.653543307087" y="252.283464566929" width="17.007874015748" height="17.007874015748" xmlns:dc="http://www.omg.org/spec/DD/20100524/DC"/>
  </bpmndi:BPMNShape>
  <bpmndi:BPMEEdge id="_DFC3ABB5-AF6F-496B-841E-7D3875A10AED" bpmnElement="_f544e6ce-dbfe-4e01-a942-581ea7a76d17" sourceElement="_bd2d2300-de55-4aa0-baf3-f43398a36666" targetElement="_da813ae8-7300-4d0c-8cb3-032a84f4d77f">
    <di:waypoint x="490.393700787402" y="253.700787401575" xmlns:di="http://www.omg.org/spec/DD/20100524/DI"/>
    <di:waypoint x="511.653543307087" y="253.700787401575" xmlns:di="http://www.omg.org/spec/DD/20100524/DI"/>
  </bpmndi:BPMEEdge>
  <bpmndi:BPMEEdge id="_F2B6D016-13A9-42C6-9AF0-BF3F47D4CC7B" bpmnElement="_5c3838c3-1fa1-45b2-a4c6-a0f66d592f3f" sourceElement="_7883fbf8-a2c9-469f-8084-2bab5e877326" targetElement="_88e44689-8fed-476a-b4bc-42e894c23fab">
    <di:waypoint x="116.220472440945" y="253.700787401575" xmlns:di="http://www.omg.org/spec/DD/20100524/DI"/>
    <di:waypoint x="155.905511811024" y="253.700787401575" xmlns:di="http://www.omg.org/spec/DD/20100524/DI"/>
  </bpmndi:BPMEEdge>
  <bpmndi:BPMEEdge id="_9681136E-B7EF-4F53-8441-24D921FE0034" bpmnElement="_6a7a06bb-bb53-4861-818c-8bb0f7a2a942" sourceElement="_88e44689-8fed-476a-b4bc-42e894c23fab" targetElement="_817bf4cf-4ede-407c-8624-da8dc56d78c4">
    <di:waypoint x="240.944881889764" y="253.700787401575" xmlns:di="http://www.omg.org/spec/DD/20100524/DI"/>
    <di:waypoint x="280.629921259843" y="253.700787401575" xmlns:di="http://www.omg.org/spec/DD/20100524/DI"/>
  </bpmndi:BPMEEdge>
  <bpmndi:BPMEEdge id="_EECB2D7E-44D3-498A-9E7C-F0D65C792C27" bpmnElement="_fecb259c-2083-44d3-498a-9e7c-f0d65c792c27">

```

*****ebook converter DEMO Watermarks*****

```

4d9f-919b-bec391354605" sourceElement="_817bf4cf-4ede-407c-8624-da8dc56d78c4"
targetElement="_bd2d2300-de55-4aa0-baf3-f43398a36666">
    <di:waypoint x="365.669291338583" y="253.700787401575"
    xmlns:di="http://www.omg.org/spec/DD/20100524/DI"/>
    <di:waypoint x="405.354330708661" y="253.700787401575"
    xmlns:di="http://www.omg.org/spec/DD/20100524/DI"/>
</bpmndi:BPMEEdge>
    <bpmndi:BPMEEdge id="_19E735E6-DAB6-470C-8502-7A4790DB47FA" bpmnElement="_36ae2e66-cbfe-451a-9a7b-52539da0702b" sourceElement="_bd2d2300-de55-4aa0-baf3-f43398a36666"
targetElement="_357f89aa-eb8f-4014-9548-0928d47192a7">
        <di:waypoint x="426.614173904179" y="239.527559055118"
        xmlns:di="http://www.omg.org/spec/DD/20100524/DI"/>
        <di:waypoint x="426.614172552514" y="177.165354330709"
        xmlns:di="http://www.omg.org/spec/DD/20100524/DI"/>
    </bpmndi:BPMEEdge>
    <bpmndi:BPMSHape id="_BFA2B743-E00A-46C7-BA20-8694F79E9742" bpmnElement="_357f89aa-eb8f-4014-9548-0928d47192a7" isHorizontal="false">
        <dc:Bounds x="107.947032057394" y="96.3779527559055" width="420.71438526544"
height="80.7874015748032" xmlns:dc="http://www.omg.org/spec/DD/20100524/DC"/>
    </bpmndi:BPMSHape>
</bpmndi:BPMPNPlane>
</bpmndi:BPMDiagram>
</definitions>

```

Figure 17-5. Serialization of simple hierarchical model, including BPMNDI

Here we've made the XML a bit less verbose by removing the tool-proprietary BPMNDI attributes and by consolidating the namespace declarations in the BPMDiagram elements. Note the following points about Figure 17-5:

- There are two *BPMDiagram* elements, signifying two pages.
- Both pages have *resolution* set to 72 pixels per inch. To convert location and size coordinates to inches, divide the pixel values by 72.
- The *BPMPNPlane* with *id* equal to *_1* is a *top-level page*, because its *bpmnElement* points to the *collaboration* element. The *BPMPNPlane* with *id* equal to *_2* is a *child-level page* because its *bpmnElement* points to a *subProcess* element. Tools should populate the attribute *bpmnElement* for all *BPMPNPlane* elements.
- That *subProcess* shape is *collapsed* because its *isExpanded* value is *false*. All shapes referencing subprocess-type elements should populate *isExpanded*. A subprocess-type element referenced by a page (*BPMPNPlane*) should always be collapsed. When *isExpanded* is *true*, both parent and child-level semantic elements should be displayed on the same page.
- There are two *participant* elements in the semantic model, but only one of them – *Customer* – is referenced by a pool shape in BPMNDI. Not all semantic elements have a corresponding shape in BPMNDI.
- Two shapes, one on each page, point to the same *participant*, the one named *Customer*, indicating pool shapes. The *participant* has no *processRef*, so it is a black-box pool. The tool I used to create Figure 17-3 and Figure 17-4 allowed me to indicate that both *Customer* pool

shapes reference the same *participant* element; it did not simply assume that because their *names* are the same (although that would not be a bad idea).

- Both pool shapes have *isHorizontal* set to true. Tools should set the value of this attribute for pool and lane shapes.

Chapter 18

Interchange of process models was an explicit goal of BPMN 2.0, but as of this writing it has not yet been realized in practice. Unfortunately, XML serialization in accordance with all the rules of the BPMN 2.0 XSD, metamodel, and spec narrative still allows enough variation to make interoperation between tools difficult. Even if we limit the problem to interchange of models containing only the elements and attributes in the Analytic subclass – that is, non-executable Level 2 models, including only information visible in the diagram – the BPMN 2.0 spec does not guarantee a unique serialization. In practice, additional conventions and validation checks are needed to facilitate model interchange.

Ideally, such detailed rules for model interchange should be part of the BPMN specification. But they are not there today, and they are unlikely to be added anytime soon, for several reasons:

- It took over three years to complete BPMN 2.0, and a full year after finalization, tool vendors are only now beginning to implement the final standard. A new version of the standard would take a couple years more, at least.
- The current specification does not even include a consolidated list of its existing semantic rules. That would be required before adding any new validation checks to the standard.
- The main focus of the BPMN technical committee in OMG has been (and, I believe, remains) execution semantics, not the non-executable models of the Analytic subclass.
- The consensus-driven OMG standards process is unlikely ever to constrain tools sufficiently to ensure BPMN model interchange. I suspect many tool vendors secretly prefer that interchange with other tools is *not* an easy thing to do. The real beneficiaries of model interchange are *end users*, and they have little influence over the standards.

For these reasons, I am tackling the issue myself. I call the initiative *BPBMN-I*, in analogy with WS-I, a successful grass-roots effort to promote interoperability of web services by defining a *Basic Profile*, constraints on implementations beyond those of the official web service standards. While WS-I is concerned with runtime interoperability, BPBMN-I takes on a much easier problem: design-time interchange of non-executable BPMN models, using only the elements and attributes in the Analytic subclass. I invite BPMN implementers of all types to collaborate with me in the effort. [\[23\]](#)

The guiding principle of BPBMN-I is this: *Any BPMN model conforming to the Analytic subclass should have one and only one XML serialization.* That requires imposing additional constraints on the serialization beyond those of the BPMN 2.0 spec. I call that set of constraints the *BPBMN-I Profile*.

A key lesson from my experience in BPMN training is that getting modelers to conform to best practice conventions works best when those conventions can be reduced to *rules that are validated in a *****ebook converter DEMO Watermarks******

tool. Simply publishing a list of rules is not nearly as effective as implementing those rules in a tool. I have created such a tool for the BPMN-I profile using XSLT 2.0 and I am making it available for [\[24\]](#)

implementers to use. The evolution of Method and Style from “best practices” to rules implemented in the itp commerce modeling tool has made a huge difference in the quality of student models in my BPMN training, and I expect that BPMN-I validation in a tool will similarly accelerate the implementation of interoperable BPMN by tool vendors.

The BPMN-I Profile is a work in progress. Ultimately, its success is dependent on participation and adoption by implementers such as you. If successful, I believe it will eventually be incorporated in some fashion in a future version of the official BPMN standard, just as Level 1 and Level 2 of Method and Style became the Descriptive and Analytic subclasses in BPMN 2.0.

The BPMN-I Profile is primarily a set of rules governing the *export* of BPMN 2.0-compliant XML from tools. The XML serialization of a BPMN 2.0 model can be verified against the BPMN-I Profile using my validation tool, which reports specific violations. *Violation of a BPMN-I Profile rule does not mean the model violates any rules of the BPMN spec, only that it may not be interoperable with other tools claiming to be BPMN-I compliant.*

The purpose of the BPMN-I Profile is to allow modelers to determine in advance Tool B’s ability to import and understand a BPMN model created by Tool A. A BPMN tool may assert the ability to *import* BPMN-I-compliant XML, possibly with specific exceptions. The complete BPMN-I profile includes all elements and attributes of the Analytic subclass, *import* of external BPMN files and remote QName references, hierarchical modeling, and BPMNDI. However, I know of no BPMN tool today that does it all and conforms to all the BPMN-I serialization constraints.

BPMN-I Profile Serialization Rules

The BPMN-I serialization rules apply to the *BPMN 2.0 XML export* of any model in the Analytic subclass. Many of the rules concern elements and attributes populated by the exporting tool, independent of the diagram created by the modeler. However, some rules effectively constrain the actions of modelers themselves, in the sense that certain diagrams that can be drawn in the tool cannot be serialized unambiguously or in a way that is interoperable with other tools. *BPMN-I thus implies that tools should apply certain validation checks prior to export and warn modelers when the diagram cannot be serialized in accordance with the BPMN-I Profile.*

The term *BPMN model* is understood to include multiple *BPMN files* linked by one or more *import* elements. In that case, *one* of the BPMN files is considered the *top-level BPMN file* for the model. My BPMN-I Profile validation tool applies an XSLT 2.0 transform to the top-level BPMN file to generate the error report.

In the list of rules that follows, attributes and child elements are identified using XPATH syntax, in which *A/B* means *child element B of A*, and *A/@B* means *attribute B of A*. The rule numbers in brackets correspond to violations reported by the tool.

Schema Validation

- [R0001] As a prerequisite, all BPMN files in the model must be valid per the final BPMN 2.0 XSD (<http://www.omg.org/spec/BPMN/20100501/BPMN20.xsd>). In addition to verifying presence in the BPMN file and correct document order of required elements and attributes, schema validation checks the uniqueness of all attributes of type *xsd:ID* and validates the presence of elements referenced by attributes and elements of type *xsd:IDREF*.
- [R0003] The BPMN model must include at least one *process* or *collaboration* element.

definitions

- [R0004] The *targetNamespace* of any BPMN file in the model may not be the BPMN 2.0 namespace.
- [R0005] *definitions/@exporter* must be populated in every BPMN file in the model. The value should be the name of the tool creating the serialization.
- [R0006] *definitions/@exportVersion* must be populated in every BPMN file in the model. The value should be the detailed version number, equivalent to that found in the Help/About dialog of the exporting tool.

import

- [R0002] A BPMN file referenced by *import* must be available from the specified *location*.

Non-Standard Elements and Attributes

- [R0007] Model elements not defined by the BPMN 2.0 XSD must be in a declared namespace other than either the BPMN 2.0 namespace or the *targetNamespace* of any BPMN file in the model.
- [R0008] Model elements not defined by the BPMN 2.0 XSD must be enclosed in an *extensionElements* tag.
- [R0009] Model attributes not defined by the BPMN 2.0 XSD must be in a declared namespace other than the BPMN 2.0 namespace or the *targetNamespace* of any BPMN file in the model.

Remote Element References

Schema validation ensures the presence of local (IDREF) references, but does not ensure the presence of elements targeted by BPMN 2.0 remote references of type *xsd:QName*. BPMN-I validation ensures the presence of remote QName references, according to the following rules:

- If the remote reference does not contain a colon, the *targetNamespace* of the referencing and referenced elements must be the same, and an element with *id* matching the remote reference string value must exist in the model.
- If the remote reference contains a colon, the namespace corresponding to the prefix must be declared in the context of the referencing element and must match the *targetNamespace* of the referenced element. In addition, the string following the colon must match the *id* of the referenced element.

The following members of the Analytic subclass are QName remote references subject to “Element not found” errors in BPMN-I validation:

- *flowNode/@default* [*flowNode* stands for any activity, gateway, or event element.]
- *callActivity/@calledElement*
- *boundaryEvent/@attachedToRef*
- *participant/@processRef*
- *messageFlow/@sourceRef*
- *messageFlow/@targetRef*
- *messageFlow/@messageRef*
- *bpmndi:BPMNPlane/@bpmnElement*
- *bpmndi:BPMNShape/@bpmnElement*
- *bpmndi:BPMNEdge/@bpmnElement*

Page Structure

Each *page* in the graphical model is represented by a separate *BPMDiagram* element and its child *BPMNPlane*. The *page structure* of a BPMN model is specified by the *bpmnElement* attribute of *BPMNPlane*. If a *BPMNPlane* references a subprocess-type element, it is a *child-level page*. Otherwise it is a *top-level page*.

Link event pairs used as *off-page connectors*, while allowed by the BPMN 2.0 spec, are not supported by BPMN-I.

- [R9001] A *BPMNPlane* must contain at least one *BPMNShape*. For example, a Visio page that contains only explanatory documentation should not be exported as a *BPMDiagram* in the BPMN graphical model.
- [R9002] A *BPMDiagram* must have a *name*. This attribute should hold the name or title of the page created in the BPMN tool.
- [R9003] A *BPMDiagram* must specify a *resolution*, in pixels per inch.
- [R9004] A *BPMNPlane* must have an *id*. In the XSD, both *BPMDiagram* and *BPMNPlane* have *id* attributes, but in BPMN-I *BPMNPlane/@id* is used to identify the page.
- [R9005] *BPMNPlane/@bpmnElement* must point to a *subProcess*, *callActivity*, *process*, or *collaboration* element. (*transaction* and *adHocSubProcess* are outside the Analytic subclass and should not appear in models conformant with the BPMN-I Profile.)
- [R9006] If *BPMNPlane/@bpmnElement* references a *collaboration*, the page may contain *flowElements* of more than one *process*; if it references a *process*, the page may contain *flowElements* of only that *process*. In either case, this *BPMNPlane* signifies a *top-level page*.
- [R9007] If *BPMNPlane/@bpmnElement* references a *collaboration*, *flowElements* of at most one *process* on that page may be unenclosed by a *pool shape*.
- [R9008] If *BPMNPlane/@bpmnElement* references a *subProcess* or *callActivity*, all *flowElements* on that page must be children of the referenced *subProcess* or *callActivity*. In this case the *BPMNPlane* signifies a *child-level page*.
- [R9009] If *BPMNPlane/@bpmnElement* references a *subProcess* or *callActivity*, the *BPMNShape* referencing that *subProcess* or *callActivity* must have attribute *isExpanded* equal to *false*.
- [R9010] All *flowElements* in a *process* level must be displayed on the same page. Link pair off-page connectors are not supported by the BPMN-I Profile.
- [R9011] *BPMNLabelStyle*, child element of *BPMDiagram*, is not supported by BPMN-I and should not be included in any model conformant with the BPMN-I Profile.

participant and Pool

A *pool* is, by definition, any *BPMNShape* that points to a *participant* in the semantic model. The *pool label* corresponds to *participant/@name*.

- [R3001] A model may not contain two or more *participants* in the same *targetNamespace*

with the same *name*. This can occur when two or more pages in the model have pools with the same label, but the tool does not recognize that they reference the same semantic element.

- [R3002] *participant/@processRef*, if present, must point to a *process* element in the model.
- [R3003] A *participant* element is required for every *process* that sends or receives a *messageFlow*, whether or not *flowElements* of the *process* are enclosed in a pool.
- [R9031] If a *flowNode*'s *process* has a pool shape on the page, the *flowNode* shape must be enclosed within the pool boundary. In other words, all *flowNodes* in the *process* must be drawn inside the pool shape.
- [R9120] A pool shape may not overlap with another pool shape. In particular, a pool may not be nested inside another pool.
- [R9121] A black-box pool shape may not contain or overlap any *flowNode* shapes. It should be completely empty. A black-box pool is a *BPMNShape* whose *bpmnElement* points to a *participant* that has no *@processRef*.
- [R9122] A black-box pool may not contain or overlap any *lane* shapes.
- [R9123] All of the *flowNode* shapes contained in a pool must point to semantic *flowNodes* belonging to the *participant*'s referenced *process*. In other words, one *process*'s pool may not enclose a *flowNode* of another *process*.
- [R9124] A pool shape must populate Boolean attribute *isHorizontal*. A value of *true* means the pool extends the width of the diagram with label on the left edge; a value of *false* means the pool extends top to bottom with label on the top edge.

collaboration

- [R0500] A *collaboration* must contain at least one *participant*.
- [R0501] *collaboration* attributes other than *id* and *name* are not supported by BPMN-I and should not appear in models conforming to the BPMN-I Profile.
- [R0502] *collaboration* child elements other than *documentation*, *extensionElements*, *participant*, *messageFlow*, *association*, *group*, and *textAnnotation* are not supported by BPMN-I and should not appear in models conforming to the BPMN-I Profile.

process

- [R1001] A *process* must contain at least one activity. Some tools always create a “main process” but leave it empty if the modeler encloses *flowElements* in a pool. That would be a BPMN-I Profile violation.
 - [R1002] A *process* must have a *name*. Note: there is no *BPMNShape/@bpmnElement* that points to a *process*, so *process/@name* may be invisible in the diagram. Method and Style recommends populating *process/@name* with *participant/@name* (the pool label).
 - [R1003] The *process name* must not be the same as the *name* of any *subProcess* or *callActivity*
- *****ebook converter DEMO Watermarks*****

contained in the *process*.

- [R1004] *process/@processType* must be omitted or *None*.
- [R1005] *process/@isExecutable* must be omitted or *false*. The BPMN-I Profile applies to non-executable BPMN only.
- [R1006] A *process* must contain at least one *startEvent*.
- [R1007] A *process* must contain at least one *endEvent*.
- [R1008] Two *process* elements in the same *targetNamespace* must not have the same *name*.

laneSet and lane

- [R1102] If a *laneSet* is used in a process level, the node-set *laneSet/lane/@flowNodeRef* must include pointers to all *flowNode* elements in that process level. In other words, if a process level uses lanes, all its *flowNodes* must be referenced by one *lane* or another in the semantic model.
- [R1103] *lane/@flowNodeRef* must point to a *flowNode*, i.e., an activity, gateway, or event. Sequence flows, data objects, and text annotations are not valid targets of *flowNodeRef*.
- [R1101] The shapes representing all of the *lanes* in a *laneset* must be displayed on the same page.
- [R9130] A *lane shape* should populate Boolean attribute *isHorizontal*.
- [R9131] A *lane shape* may not extend beyond its enclosing pool.
- [R9132] A *lane shape* should extend the full length of its enclosing pool. This is a requirement of the BPMN spec, but there is ambiguity concerning the *dc:Bounds* values for the top left corner of a *lane shape*. BPMN-I resolves the ambiguity. For horizontal pools, BPMN-I requires the *lane shape's dc:Bounds/@x* value to be the same as that of the pool shape. However, some BPMN tools put the top left corner of the lane to the right of the *pool label box*. BPMNDI provides nothing to specify the size of the pool label box, however. To repeat, in BPMN-I the *dc:Bounds/@x* value of horizontal lane and pool shapes should be the same.

flowNode

The *flowNode* abstract class includes activity, gateway, and event elements. The following BPMN-I rules pertain to all *flowNodes*:

- [R1200] Only *flowNodes* in the Analytic subclass are allowed in models conforming to the BPMN-I Profile. These include *task*, *userTask*, *serviceTask*, *sendTask*, *receiveTask*, *callActivity*, *subProcess* [*@triggeredByEvent=false()*], *exclusiveGateway*, *inclusiveGateway*, *eventBasedGateway*, *startEvent*, *intermediateThrowEvent*, *intermediateCatchEvent*, *boundaryEvent*, and *endEvent*.
- [R1201] Any *flowNode* in the Analytic subclass other than *startEvent*, *boundaryEvent*, catching Link event, or child of a “parallel box” *subProcess* should have incoming sequence flow.

- [R1202] Any *flowNode* in the Analytic subclass other than *endEvent*, throwing Link event, or child of a “parallel box” *subProcess* should have outgoing sequence flow.
- [R1203] *flowNode/incoming* and *flowNode/outgoing* should be omitted from the serialization. These elements are not in the Analytic subclass, and are redundant to *sequenceFlow/@sourceRef* and *sequenceFlow/@targetRef*.
- [R1204] The only *flowNodes* that may have attribute *default* are *activity* elements, plus *exclusiveGateway* and *inclusiveGateway*.
- [R1209] *flowNode/@default* must point to a *sequenceFlow* outgoing from the *flowNode*.

activity

- [R1300] Only *activity* elements in the Analytic subclass are allowed in models conforming to the BPMN-I profile. These include *task*, *userTask*, *serviceTask*, *sendTask*, *receiveTask*, *callActivity*, and *subProcess* [*@triggeredByEvent=false()*].
- [R1301] An *activity* should have a *name*, displayed as the label of the activity shape.
- [R1302] *activity/startQuantity* and *activity/completionQuantity* should be omitted from the serialization, implying the default value of 1 for both. These elements are not in the Analytic subclass.
- [R1303] Compensating activities (*activity[@isForCompensation = true()]*) are not part of the Analytic subclass and not allowed by BPMN-I Profile.
- [R1330] *callActivity/@calledElement* must point to either a *process* or *global task*.

startEvent

- [R1500] Only *startEvents* in the Analytic subclass are allowed in models conforming to the BPMN-I Profile. These include those with either no *eventDefinition* child or child element *messageEventDefinition*, *timerEventDefinition*, *signalEventDefinition*, or *conditionalEventDefinition*. A *startEvent* may have more than one of these child elements, but may not have attribute *@parallelMultiple = true()*. In other words, the *Multiple start event* is allowed but not the *Parallel-Multiple start event*.
- [R1501] A *startEvent* may not have incoming sequence flow. The legacy construct allowing start events on the boundary of an expanded subprocess is specifically excluded by the BPMN-I Profile.
- [R1502] A *startEvent* may not have outgoing message flow.
- [R1503] A *startEvent* with incoming message flow must have child *messageEventDefinition*.
- [R1505] A *startEvent* in a *subProcess* (not an *event subprocess*) must have None trigger, i.e., must have no child *eventDefinition* elements.
- [R1506] Attribute *startEvent/@isInterrupting* is not part of the Analytic subclass and should

be omitted from models conforming to the BPMN-I Profile. It is only used in event subprocesses, which are not part of the Analytic subclass.

boundaryEvent

- [R1600] Only *boundaryEvents* in the Analytic subclass are allowed in models conforming to the BPMN-I profile. These include only those with child element *messageEventDefinition*, *timerEventDefinition*, *errorEventDefinition*, *escalationEventDefinition*, *conditionalEventDefinition*, or *signalEventDefinition*. A *boundaryEvent* must have at least one of these child elements, but may not have attribute *@parallelMultiple = true()*. In other words, the *Multiple boundary event* is allowed but not the *Parallel-Multiple boundary event*.
- [R1619] Attribute *attachedToRef* must point to an *activity* in the same process level.
- [R1620] A *boundaryEvent* must have exactly one outgoing *sequenceFlow*.
- [R1622] A *boundaryEvent* may not have incoming *sequenceFlow*.
- [R1623] An Error *boundaryEvent* on a *subProcess* requires matching Error *endEvent* in the child-level expansion, unless the *subProcess* contains no child elements.
- [R1624] An Error *boundaryEvent* may not be non-interrupting, i.e., may not have *@cancelActivity=false()*.
- [R1630] An Escalation *boundaryEvent* on a *subProcess* requires matching Escalation *intermediateThrowEvent* or *endEvent* in the child-level expansion, unless the *subProcess* contains no child elements.

intermediateCatchEvent and intermediateThrowEvent

- [R1700] Only *intermediateThrowEvents* in the Analytic subclass are allowed in models conforming to the BPMN-I Profile. These include only those either with no *eventDefinition*, or with child element *messageEventDefinition*, *signalEventDefinition*, *escalationEventDefinition*, or *linkEventDefinition*.
- [R1701] Only *intermediateCatchEvents* in the Analytic subclass are allowed in models conforming to the BPMN-I Profile. These include only those with child element *messageEventDefinition*, *timerEventDefinition*, *signalEventDefinition*, *conditionalEventDefinition*, or *linkEventDefinition*.
- [R1744] A Link *intermediateThrowEvent* may not have outgoing sequence flow.
- [R1745] The target of a Link *intermediateThrowEvent* must be a Link *intermediateCatchEvent* in the same process level. Because of a bug in the BPMN 2.0 XSD, the target is NOT identified by the optional child *linkEventDefinition/target*, but instead by the *required* attribute *linkEventDefinition/@name*. Although this attribute is type *xsd:string* in the XSD, BPMN-I requires its value to be a pointer to the *id* of the target *intermediateCatchEvent* element.
- [R1746] A Link *intermediateCatchEvent* may not have incoming sequence flow.

- [R1747] The source of a Link *intermediateCatchEvent* must be a Link *intermediateThrowEvent* in the same process level. Because of a bug in the BPMN 2.0 XSD, the source is NOT identified by the optional child *linkEventDefinition/source*, but instead by the required attribute *linkEventDefinition/@name*. Although this attribute is type *xsd:string* in the XSD, BPMN-I requires its value to be a pointer to the *id* of the source *intermediateThrowEvent* element.

endEvent

- [R1800] Only *endEvents* in the Analytic subclass are allowed in models conforming to the BPMN-I Profile. These include only those with child element *messageEventDefinition*, *terminateEventDefinition*, *errorEventDefinition*, *signalEventDefinition*, or *escalationEventDefinition*, or no child in the *eventDefinition* class. An *endEvent* may have more than one of these child elements.
- [R1850] An *endEvent* may not have outgoing sequence flow.
- [R1851] An *endEvent* may not have incoming message flow.
- [R1852] An *endEvent* with outgoing message flow must have child *messageEventDefinition*.

Gateway

- [R1900] Only gateway elements in the Analytic subclass are allowed in models conforming to the BPMN-I Profile. These include *exclusiveGateway*, *inclusiveGateway*, *eventBasedGateway*, and *parallelGateway* only.
- [R1901] Attribute *gatewayDirection* is not in the Analytic subclass and should be omitted from the serialization.
- [R1902] Attribute *default* on *exclusiveGateway* or *inclusiveGateway* must point to a sequence flow outgoing from the gateway.
- [R1903] On *eventBasedGateway*, attribute *instantiate* is not part of the Analytic subclass and is not allowed in models conforming to the BPMN-I Profile. (The default behavior corresponds to the value *false()*.)
- [R1904] On *eventBasedGateway*, attribute *eventGatewayType* is not part of the Analytic subclass and is not allowed in models conforming to the BPMN-I Profile. (The default behavior corresponds to the value *Exclusive*.)
- [R1960] A gateway may not have incoming message flow.
- [R1961] A gateway may not have outgoing message flow.
- [R1962] A gateway may not have one incoming and one outgoing sequence flow.
- [R1965] Each gate of an *eventBasedGateway* must be either an *intermediateCatchEvent* or a *receiveTask*.

sequenceFlow

- [R2000] *sequenceFlow/@sourceRef* must point to a *flowNode* in the same process level.
- [R2001] *sequenceFlow/@targetRef* must point to a *flowNode* in the same process level.
- [R2002] Attribute *isImmediate* is not part of the Analytic subclass and should be omitted from models conforming to the BPMN-I Profile.
- [R2003] *sourceRef* and *targetRef* values may not be the same; a sequence flow may not connect a *flowNode* to itself.
- [R2004] If a *flowNode* has only one outgoing *sequenceFlow*, the *sequenceFlow* must be unconditional, i.e., it may not have child element *conditionExpression*.
- [R2005] If *sequenceFlow/@sourceRef* points to *parallelGateway* or *eventBasedGateway*, the *sequenceFlow* must be unconditional, i.e., it may not have child element *conditionExpression*.
- [R2006] If *sequenceFlow* has child *conditionExpression*, the *sequenceFlow* may not be referenced by the *default* attribute of an activity or gateway.
- [R2007] If *sequenceFlow/@sourceRef* points to *exclusiveGateway* or *inclusiveGateway*, it should have child element *conditionExpression*, unless it is the gateway *default* flow. In non-executable BPMN, *conditionExpression* is usually an empty element. The label of the *sequenceFlow* connector is the *sequenceFlow/@name*, not the content of *conditionExpression*.

messageFlow

- [R3102] *messageFlow* attributes *source* and *target* may not point to elements in the same process.
- [R3103] *messageFlow/@source* must point to an *activity*, *intermediateThrowEvent* with child *messageEventDefinition*, *endEvent* with child *messageEventDefinition*, or a black-box pool (*participant[not(@processRef)]*).
- [R3104] *messageFlow/@target* must point to an *activity*, *intermediateCatchEvent* with child *messageEventDefinition*, *boundaryEvent* with child *messageEventDefinition*, *startEvent* with child *messageEventDefinition*, or a black-box pool (*participant[not(@processRef)]*).
- [R3105] *messageFlow/@messageRef*, if present, must point to a *message* element in the model.

textAnnotation and association

A *textAnnotation* is an *artifact*. In the serialization it is a child of either a *collaboration* or a *process* element. A *textAnnotation* is usually linked to a *flowElement* via an *association* connector, but the spec does not require it. If the *association* is present, the node at the other end determines the parent element. If *textAnnotation* is “floating” with no *association*, the *bpmnElement* attribute of the page (*BPMNPlane*) on which it appears points to the parent element of the *textAnnotation*. If *bpmnElement* points to a subprocess-type element, the *process* to which that element belongs is the parent of *textAnnotation*.

- [R4001] If a *textAnnotation* is linked to a *flowElement* via *association*, the parent of the

textAnnotation must be the *process* to which the *flowElement* belongs.

- [R4002] If a *textAnnotation* is linked to a *participant* or *messageFlow* via *association*, the parent of the *textAnnotation* must be the *collaboration* to which the *participant* or *messageFlow* belongs.
- [R4003] If a *textAnnotation* is not connected to an *association* and is drawn on a *top-level page*, the parent of the *textAnnotation* must be the *process* or *collaboration* referenced by *bpmndi:BPMPNPlane/@bpmnElement*.
- [R4004] If a *textAnnotation* is not connected to an *association* and is drawn on a *child-level page*, the parent of the *textAnnotation* must be the *process* parent of the subprocess-type element referenced by *bpmndi:BPMPNPlane/@bpmnElement*.
- [R4005] An *association* connecting to a *textAnnotation* must be non-directional, i.e., attribute *associationDirection* must be either omitted or *None*.

group

Like *textAnnotation*, *group* is an artifact that belongs either to a *process* or a *collaboration*.

- [R4500] If a *group* is drawn on a *top-level page*, its parent is the *process* or *collaboration* element referenced by *bpmndi:BPMPNPlane/@bpmnElement*.
- [R4501] If a *group* is drawn on a *child-level page*, its parent is the *process* containing the subprocess-type element referenced by *BPMPNPlane/@bpmnElement*.
- [R4502] Attribute *categoryValueRef* is not supported by BPMN-I and should not appear in any model conformant to the BPMN-I Profile.

Data Flow

- [R5001] A *dataObject* element is allowed by the BPMN-I Profile only if a data object shape pointing to it exists in the graphical model.
- [R5002] Only *dataObject* attributes in the Analytic subclass are allowed in models conforming to the BPMN-I Profile. These include *id* and *name*.
- [R5003] A *dataStoreReference* element is allowed by the BPMN-I Profile only if a data store shape pointing to it exists in the graphical model.
- [R5004] Only *dataStoreReference* attributes in the Analytic subclass are allowed in models conforming to the BPMN-I Profile. These include *id*, *name*, and *dataStoreRef*.
- [R5005] *dataStoreReference/@dataStoreRef* must point to a *dataStore* element.
- [R5006] A *dataObject* or *dataStoreReference* must be the *sourceRef* of a *dataInputAssociation* or the *targetRef* of a *dataOutputAssociation*, or both. It may not be “unattached”.
- [R5007] The *sourceRef* of a *dataInputAssociation* may only be a *dataObject* or *dataStoreReference*.
- [R5008] The *targetRef* of a *dataInputAssociation* may only be a *dataInput*.

*****ebook converter DEMO Watermarks*****

- [R5009] The *sourceRef* of a *dataOutputAssociation* may only be a *dataOutput*.
- [R5010] The *targetRef* of a *dataOutputAssociation* may only be a *dataObject* or *dataStoreReference*.

BPMNShape

- [R9030] A *BPMNShape* must have attribute *bpmnElement* that points to a semantic element in the model.
- [R9101] A *BPMNShape* must have an *id*.
- [R9102] A *BPMNShape* may not reference a *process*. A pool shape must reference a *participant*.
- [R9103] A *BPMNShape* referencing a *subProcess* or *callActivity* must populate attribute *isExpanded*. No other shape may have this attribute.
- [R9104] A *BPMNShape* referencing a *participant* (pool) or *lane* must populate attribute *isHorizontal*. No other shape may have this attribute.
- [R9105] *BPMNShape/@isMarkerVisible* applies only to *exclusiveGateway*. No other shape may have this attribute.
- [R9106] *BPMNShape/@isMessageVisible* applies only to *message*. No other shape may have this attribute.
- [R9107] *BPMNShape/dc:Bounds/@x* and *dc:Bounds/@y* values may not be negative.
- [R9108] *BPMNShape/dc:Bounds/@height* and *dc:Bounds/@width* values may not be negative.
- [R9109] *BPMNShape* child *bpmndi: BPMNLabel* is not supported by BPMN-I and should not appear in any model conformant with the BPMN-I Profile.

BPMNEdge

BPMNEdge is the graphical representation of sequence flow, message flow, association, and data association elements. Child elements *di:waypoint* are an ordered list of coordinates representing the connector source, bendpoints, and target location.

- [R9050] A *BPMNEdge* must have attribute *bpmnElement* that points to a semantic connector element in the model.
- [R9051] A *BPMNEdge* must have an *id*.
- [R9052] *BPMNEdge* attributes *sourceElement*, *targetElement*, and *messageVisibleKind* and child element *BPMNLabel* are not supported by BPMN-I and should not appear in any model conformant with the BPMN-I Profile.
- [R9053] The first *di:waypoint* of a *BPMNEdge* should lie on or within the bounding box of the *BPMNShape* for the element referenced by the semantic connector's *sourceRef*.

- [R9054] The last *di:waypoint* of a *BPMNEdge* should lie on or within the bounding box of the *BPMNShape* for the element referenced by the semantic connector's *targetRef*.

Part V: BPMN Implementer's Guide – Executable BPMN

Chapter 19

What Is Executable BPMN?

Up to this point in the book, we have focused on *non-executable BPMN*, in which the process diagram describes the process logic in a human-understandable way. The primary emphasis is on the diagram, the visual representation of the process logic. The XML serialization serves primarily the purpose of model interchange between tools as well as to make the semantics more precise. However, most of the effort in developing the BPMN 2.0 specification involved elements related to *executable processes*. In an executable process, a software engine automates the flow of model execution from process instantiation to completion. This requires additional details to be specified for each BPMN element, including:

- Process variables
- Task input and output data, and their mappings to variables
- Task user interface forms and screenflows
- Task performer assignment logic
- Conditional expressions
- Event definitions
- Messages

These details are invisible in the diagram, but BPMN 2.0 provides XML elements to specify them.

BPMN 1.x-based BPM Suites have been available from numerous vendors for several years. They support execution of process logic defined in BPMN, but they are *not* what we mean here by executable BPMN. The reason is that while the process logic in those tools may follow the semantics and rules of BPMN, the execution-related details listed above are specified by each tool in a proprietary manner. Standardization of these execution-related details was an explicit objective of BPMN 2.0.

That does not mean, however, that BPMN 2.0 is a *process execution language* like BPEL, in which the language may be executed directly on the process engine. Some vendors may implement such an engine, but I expect executable BPMN 2.0 to serve primarily as an *interchange format*. Internally, each tool has its own proprietary object model, but will be able to *export* execution-related details using BPMN 2.0 XML, and ideally *import* them as well. Thus, in the context of this book, the term “executable BPMN” refers to a tool’s ability to specify and export execution-related details, such as those listed above, consistent with the BPMN 2.0 metamodel and schema.

Common Executable Subclass

In addition to the Analytic and Descriptive subclasses for non-executable BPMN, the BPMN 2.0 spec enumerates the elements and attributes supported for basic executable BPMN, called the *Common Executable subclass*. In terms of the shapes and symbols included, Common Executable is close to the Descriptive subclass, but it includes additional child elements and attributes to specify the executable details. The Common Executable subclass requires support of XML Schema as the type definition language, WSDL as the definition language for service interfaces, and XPath as the language for referencing data elements.

Element	Attributes
sequenceFlow	id, name, sourceRef, targetRef, conditionExpression, default
exclusiveGateway	id, name, gatewayDirection, default
parallelGateway	id, name, gatewayDirection
eventBasedGateway	Id, name, gatewayDirection, eventGatewayType
userTask	id, name, rendering, implementation, resource, ioSpecification, dataInputAssociation, dataOutputAssociation, loopCharacteristics, boundaryEventRefs
serviceTask	id, name, implementation, operationRef, ioSpecification, dataInputAssociation, dataOutputAssociation, loopCharacteristics, boundaryEventRefs
subProcess	id, name, flowElement, loopCharacteristics, boundaryEventRefs
callActivity	id, name, calledElement, ioSpecification, dataInputAssociation, dataOutputAssociation, loopCharacteristics, boundaryEventRefs
dataObject	id, name, isCollection, itemSubjectRef
textAnnotation	id, text
dataAssociation	id, name, sourceRef, targetRef, assignment
startEvent (None)	id, name
endEvent (None)	id, name
Message startEvent	id, name, messageEventDefinition (ref or contained), dataOutput, dataOutputAssociation
Message endEvent	id, name, messageEventDefinition (ref or contained), dataInput, dataInputAssociation
Terminate endEvent	id, name, terminateEventDefinition
Message intermediateCatchEvent	id, name, messageEventDefinition, dataOutput, dataOutputAssociation
Message intermediateThrowEvent	id, name, messageEventDefinition, dataInput, dataInputAssociation
Timer intermediateCatchEvent	id, name, timerEventDefinition
Error boundaryEvent	id, name, attachedToRef, errorEventDefinition, dataOutput, dataOutputAssociation

Figure 19-1. Common Executable Process Modeling Conformance subclass

The Common Executable subclass includes also the following supporting elements:

*****ebook converter DEMO Watermarks*****

Element	Attributes
standardLoopCharacteristics	id, loopCondition
multiInstanceLoopCharacteristics	id, isSequential, loopDataInput, inputDataItem
rendering	
resource	id, name
resourceRole	id, resourceRef, resourceAssignmentExpression
ioSpecification	id, dataInput, dataOutput
dataInput	id, name, isCollection, itemSubjectRef
dataOutput	id, name, isCollection, itemSubjectRef
itemDefinition	id, structure (complexType) or import
operation	id, name, inMessageRef, outMessageRef, errorRef
message	id, name, structureRef
error	id, structureRef
assignment	id, from, to (complexType)
messageEventDefinition	id, messageRef, operationRef
terminateEventDefinition	id
timerEventDefinition	id, timeDate

Figure 19-2. Common Executable subclass, supporting elements

Note several basic elements from the Descriptive subclass are missing in Common Executable, including *pool*, *lane*, *messageFlow*, and *dataStore*. This is consistent with the fact that few BPM Suites today support collaboration diagrams in their BPMN tools. Also, the only *boundaryEvent* supported by Common Executable is *Error*, and this is presumably on a task only, since *Error endEvent* is not included in the subclass. I believe that *timerEventDefinition* child *timeDuration* was omitted inadvertently and should be added to the subclass. Still, it is clear that the Common Executable subclass supports only the bare minimum of exception handling. Nevertheless, it provides all the elements necessary to specify a basic executable process. We will look at how to do that in the next few chapters.

Chapter 20

Variables and Data Mapping

Serialization of data flow in non-executable models was discussed in Chapter 16. Those models, however, lacked any formal definition of data elements, expressions, and mappings. Process data is at the core of executable BPMN. In this chapter we see how such details are defined in BPMN 2.0.

Below is a brief overview:

1. Process data elements reference their definitions by pointing to an *itemDefinition* element, which in turn points to an element or complex type defined *externally* to the BPMN document and *imported* by it. Support for import of XSD and WSDL files is required by the Common Executable subclass. It is also allowed to define datatypes *internally* to the BPMN document as XSD complex types and reference them by QName from the *structureRef* attribute of *itemDefinition*.
2. *Data objects* represent process *variables* managed by the process engine. A data object is accessible only within the process level in which it is defined and its child process levels. Its lifetime is limited to the active time of the process or subprocess in which it is defined. When that process or subprocess is complete, the data object is no longer accessible.
3. *Activity dataInputs and dataOutputs*, interface parameters defined by the activity's *ioSpecification* element, are mapped to data objects by *data associations*. The mapping details are specified within the BPMN using *assignment* or *transformation*. It is also possible to use *Script tasks* to implement complex data mapping.
4. *Events* with associated *itemDefinition*, including Message, Signal, Error, and Escalation, also may have data associations that store or populate event data. Catching events have *dataOutputAssociation* only, and throwing events have *dataInputAssociation* only.

Now let's take a deeper look.

itemDefinition

In non-executable BPMN, process data is described simply by the *name* of a *dataObject* or *dataStore* element. In executable BPMN, or even in BPMN used to describe business requirements for implementation, more detailed data description is needed, and BPMN 2.0 supports this through the *itemDefinition* element. All item-aware elements have an attribute *itemSubjectRef* that points to an *itemDefinition*. *itemDefinition* is a root element and may be referenced by any item-aware element in the model.

Note that the *name* of the data element is an attribute of the item-aware element, not of the *itemDefinition*.

BPMN does not provide its own data definition language. Data structures are assumed to be defined externally, using standard data definition languages and tools, and *imported* into the BPMN model. The *typeLanguage* attribute of the root *definitions* element specifies the default type language for all *itemDefinitions*; if omitted, the XSD type language is assumed.

Here again the specification confuses matters with a bug. The metamodel (Figure 8.25 and Table 8.47 in the spec) gives *itemDefinition* an additional attribute, *import*, a pointer to a root *import* element in the model. This attribute is not present in the XSD, however, and so we may not use it in the serialization. We don't really need it since the imported schema element name must be unique in its namespace.

Attributes of *itemDefinition* include:

- *id*, the target of the *itemSubjectRef* of an item-aware element.
- *isCollection*, a Boolean (default *false*) indicating a collection of data elements. A *dataObject* referencing an *itemDefinition* must have the same value of *isCollection*.
- *itemKind*, an enumerated value (*information* or *physical*, default *information*) indicating data or a physical item.
- *structureRef*, a QName pointer to the data structure, which must be a single element or complex type in the specified *typeLanguage*. If XSD (the default) is the *typeLanguage*, *structureRef* typically points to an element or complex type in an imported XSD file. Here the QName type is used as a *real* QName – a namespace-qualified element name – not a prefixed *id* value.

message

The root element *message* is also an item-aware element. *messageRef* is an attribute of *messageFlow*, *messageEventDefinition*, *sendTask*, and *receiveTask*, and points to a *message* element.

To support the Message shape in the diagram, the Analytic subclass includes only the *message* attributes *id* and *name*. In executable BPMN, the additional *message* attribute *itemRef* is a prefixed *id* pointer to an *itemDefinition* detailing the message structure. In that case, the *structureRef* of the *itemDefinition* often references an element in an imported WSDL file.

Importing Structure Definitions

We have already seen how the *import* root element is used to import BPMN files into the model. In BPMN meant for execution or detailed business requirements, *import* is also used to reference message and data structures defined in external WSDL and XSD files. Other type languages are allowed by BPMN 2.0, but the spec says that WSDL and XSD import are required for conformance.

When importing XSD files, the *importType* attribute of *import* must be set to <http://www.w3.org/2001/XMLSchema>. When importing WSDL 2.0 files, *importType* must be set to <http://www.w3.org/TR/wsdl20/>. The attribute *location* specifies the URL or filepath of the imported file, and the attribute *namespace* specifies the target namespace of the imported file.

Example: Data Flow with Imported Item Definitions

To illustrate the use of *itemDefinition* and *import*, we return to a simple data flow example, shown in Figure 20-1.

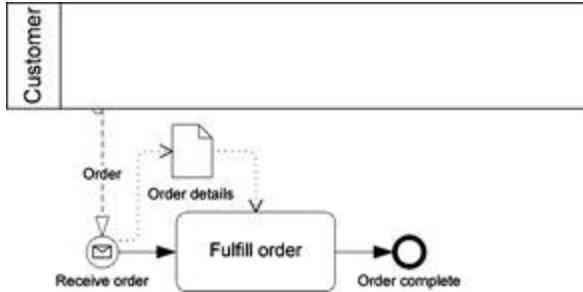


Figure 20-1. Simple data flow with imported item definitions

The serialization is shown below:

```
<definitions targetNamespace="http://www.itp-commerce.com" xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL" xmlns:itp="http://www.itp-commerce.com/BPMN2.0" xmlns:order="http://www.example.org/Order" xmlns:tns="http://www.itp-commerce.com" xmlns:xs="http://www.w3.org/2001/XMLSchema-instance" exporter="Process Modeler 5 for Microsoft Visio" exporterVersion="5.2742.13663" SRG="a26428bb-9287-4346-b659-1d89f5d41217" xsi:schemaLocation="http://www.omg.org/spec/BPMN/20100524/MODEL schemas/BPMN20.xsd">
  <import importType="http://www.w3.org/2001/XMLSchema" location="Order.xsd" namespace="http://www.example.org/Order"/>
  <import importType="http://www.w3.org/TR/wsdl20/" location="OrderProcess.wsdl" namespace="http://www.example.org/Order"/>
  <itemDefinition id="item001" structureRef="order:OrderDetails"/>
  <itemDefinition id="item002" structureRef="order:OrderMsg"/>
  <message id="msg001" name="Order" itemRef="tns:item002"/>
  <collaboration id="2ac611c8-fd55-46eb-8af3-1b3e8229a297">
    <participant id="d4c94914-9ee4-402d-86d2-427956d26872" name="Customer"/>
    <participant id="5c311ebc-4ae3-41aa-a2f5-a7802720c773" name="Order Process" processRef="5c311ebc-4ae3-41aa-a2f5-a7802720c773"/>
    <messageFlow id="a170e302-c697-42fe-b612-d4d286891621" name="Order" sourceRef="d4c94914-9ee4-402d-86d2-427956d26872" targetRef="a5ff783f-b313-46f4-997c-6a5f3bee18e0" messageRef="tns:msg001"/>
  </collaboration>
  <process id="5c311ebc-4ae3-41aa-a2f5-a7802720c773" name="Order Process" processType="None">
    <startEvent id="c529a130-7805-4b9e-90b7-8d923e4813ca" name="Receive order">
      <dataOutput id="do_c529a130-7805-4b9e-90b7-8d923e4813ca" itemSubjectRef="tns:item001"/>
      <dataOutputAssociation id="f837dfc4686-4e1c-bb9e-67123e59cadf">
        <sourceRef>do_c529a130-7805-4b9e-90b7-8d923e4813ca</sourceRef>
        <targetRef>37bff1e7-a72c-434a-81b9-2873d11b8845</targetRef>
      </dataOutputAssociation>
      <messageEventDefinition messageRef="tns:msg001"/>
    </startEvent>
    <task id="f2509706-84ef-4f59-8fdb-5f25b3102686" name="Fulfill Order">
      <iOSpecification>
        <dataInput id="di_f2509706-84ef-4f59-8fdb-5f25b3102686" itemSubjectRef="tns:item001"/>
        <inputSet>
          <dataInputRefs>di_f2509706-84ef-4f59-8fdb-5f25b3102686</dataInputRefs>
        </inputSet>
        <outputSet/>
      </iOSpecification>
      <dataInputAssociation id="985c2eb0-3265-4f13-a295-e29778b1c973">
        <sourceRef>37bff1e7-a72c-434a-81b9-2873d11b8845</sourceRef>
        <targetRef>di_f2509706-84ef-4f59-8fdb-5f25b3102686</targetRef>
      </dataInputAssociation>
    </task>
    <endEvent id="846d6306-9380-4e56-aeef-7532d1ef96fc5" name="Order complete"/>
    <dataObject id="37bff1e7-a72c-434a-81b9-2873d11b8845" name="Order details" itemSubjectRef="tns:item001"/>
    <sequenceFlow id="88c3ac5d-877d-465e-9669-c7fb2443105" sourceRef="c529a130-7805-4b9e-90b7-8d923e4813ca" targetRef="f2509706-84ef-4f59-8fdb-5f25b3102686"/>
    <sequenceFlow id="689e46f9-5213-49fd-8050-4649e6368cf1" sourceRef="f2509706-84ef-4f59-8fdb-5f25b3102686" targetRef="846d6306-9380-4e56-aeef-7532d1ef96fc5"/>
  </process>
</definitions>
```

Figure 20-2. Serialization of simple data flow with imported item definitions

Note the following about the serialization in Figure 20-2:

- There are two *import* elements, one for the schema file *Order.xsd* and the other for a WSDL file *OrderMsg.wsdl*. In this case they are in the same namespace, although it is quite common to use separate namespaces for related XSD and WSDL files.
- The namespace for the imported files is declared in *definitions* and assigned the prefix *order*.

- Also declared in definitions is the prefix *tns*, standing for the BPMN file *targetNamespace*. Since here it is the same as the default (unprefixed) namespace, we don't absolutely need it. But since the *id* values of *itemDefinition* and *message* are not globally unique in this serialization, QName references to them can be made unambiguous with the namespace prefix.

In this simple example, the start event *dataOutput*, the *dataObject*, and the task *dataInput* all reference the same element *OrderDetails* in *Order.xsd*. In the general case they do not need to be identical, as a data association can perform a mapping between them.

Properties and Instance Attributes

Values of data objects, data inputs, and data outputs may be accessed for use in data mappings and condition expressions. In addition, BPMN defines two more data elements for this purpose, *property* and *instance attribute*.

- A *property* is a user-defined data element of a process, activity, or event. It has no graphical representation in the model. For example, a key performance indicator could be defined as a *property*.
- The spec defines various *instance attributes* of a process, activity, or event, representing values that vary by instance at runtime. The currently assigned task *performer*, task *priority*, and current *loop count* of a loop activity are examples of instance attributes.

Data Mapping

Whether data flow is visualized in the diagram or not, *data mapping* is critical to all aspects of executable BPMN. The *dataInputs* and *dataOutputs* of some tasks in the process model may be predetermined by the implementation, while others may be user-defined. In either case, data must be mapped between process variables (*dataObjects*), *properties*, or *instance attributes* and the task *dataInputs* and *dataOutputs*. The mapping may be expressed in the BPMN 2.0 XML in several ways, as described below.

Identity Mapping

Identity mapping means the source and target of a data association reference the same data structure. In that case, only the *sourceRef* and *targetRef* are specified in the XML.

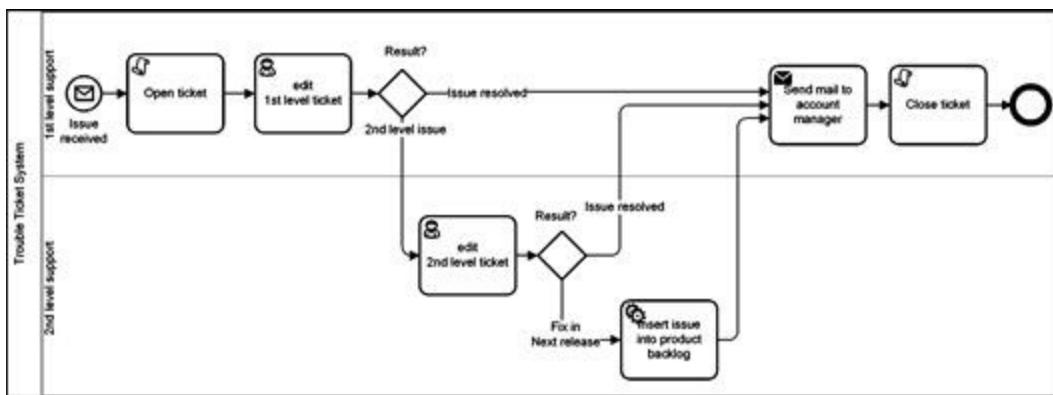


Figure 20-3. Incident Management process. Source: OMG

[25]

For example, the *Incident Management* example from the OMG website (Figure 20-3) shows a identity mapping from the *dataObject* representing the business object *TicketItem* to a *dataInput* of a *serviceTask* named *insert issue into product backlog*:

```
<dataObject id="TicketDataObject" itemSubjectRef="tns:TicketItem" />
...
<serviceTask name="Insert issue into product backlog"
  operationRef="tns:addTicketOperation" id="_1-325">
  <ioSpecification>
    <dataInput itemSubjectRef="tns:TicketItem" id="TicketDataInputOf_1-325" />
    <inputSet>
      <dataInputRefs>TicketDataInputOf_1-325</dataInputRefs>
    </inputSet>
    <outputSet />
  </ioSpecification>
  <dataInputAssociation>
    <sourceRef>TicketDataObject</sourceRef>
    <targetRef>TicketDataInputOf_1-325</targetRef>
  </dataInputAssociation>

```

*****ebook converter DEMO Watermarks*****

```

</serviceTask>
...
<itemDefinition id="TicketItem" isCollection="false" itemKind="Information"
  structureRef="com.camunda.examples.incidentmanagement.TroubleTicket" />

```

Figure 20-4. Identity mapping example. Source: OMG

Identity mapping is indicated by a *dataInputAssociation* without an *assignment* child element. With identity mapping, the *sourceRef* and *targetRef* elements must have the same datatype. Here they clearly do, as both the *dataInput* and *dataObject* have *itemSubjectRef* pointing to the same *itemDefinition*. The details of the *itemDefinition* datatype are not provided in this OMG example. Ideally, the *structureRef* attribute should point to an element or complex type in an imported XSD file.

Assignment From/To Mapping

If the source and target data elements are not identical, the *assignment/from* and *assignment/to* elements of a data association define the mapping. The *assignment/from* and *assignment/to* elements are *expressions* in the default *expressionLanguage* specified in the *definitions* root, unless overridden by the *language* attribute of the *from* or *to* element itself. In the fragment below, also excerpted from the OMG *Incident Management* example, the default expression language is the Java Universal Expression Language (UEL). The BPMN 2.0 Common Executable subclass requires support for XPath 1.0 as the expression language, but Java-based tools may find UEL easier to implement.

```

<dataObject id="TicketDataObject" itemSubjectRef="tns:TicketItem" />
...
<sendTask name="Send mail to account manager" messageRef="tns:AnswerMessage"
  operationRef="tns:sendMailToIssueReporterOperation" id="_1-150">
  <ioSpecification>
    <dataInput itemSubjectRef="tns:AnswerItem" id="AnswerDataInputOfSendTask" />
    <inputSet>
      <dataInputRefs>AnswerDataInputOfSendTask</dataInputRefs>
    </inputSet>
    <outputSet />
  </ioSpecification>
  <dataInputAssociation>
    <sourceRef>TicketDataObject</sourceRef>
    <targetRef>AnswerDataInputOfSendTask</targetRef>
    <assignment>
      <from>${getDataObject("TicketDataObject").reporter}</from>
      <to>${getDataInput("AnswerDataInputOfSendTask").recipient}</to>
    </assignment>
    <assignment>
      <from>
        A ticket has been created for your issue, which is now in
        status ${getDataObject("TicketDataObject").status}.
      </from>
      <to>${getDataInput("AnswerDataInputOfSendTask").body}</to>
    </assignment>
  </dataInputAssociation>

```

```
</sendTask>
```

Figure 20-5. Mapping example using assignment/from and assignment/to in UEL. Source: OMG

Here specific elements the *TicketItem* data object are mapped to elements of the *dataInput* of the *sendTask* named *Send mail to account manager*. Specifically, the *reporter* element of the *TicketItem* is mapped to the *recipient* element of the task *dataInput*, and a text string containing the *status* element of the *TicketItem* is mapped to the *body* element of the *dataInput*. Note here that *from* and *to* expressions do not reference the *dataObject* or *dataInput* directly, but use *accessor functions* *getDataObject* and *getDataInput*. BPMN 2.0 defines these as *extension functions* for XPath expressions accessing elements of data objects, data inputs and outputs, properties, and instance attributes. In the example above, UEL makes use of the same functions, although they may not be necessary. In XPath, the mapping would look like this:

```
<dataInputAssociation>
  <sourceRef>TicketDataObject</sourceRef>
  <targetRef>AnswerDataInputOfSendTask</targetRef>
  <assignment>
    <from>getDataObject("TicketDataObject")/tns:reporter</from>
    <to>getDataInput("AnswerDataInputOfSendTask")/tns:recipient</to>
  </assignment>
  <assignment>
    <from>
      concat("A ticket has been created for your issue, which is now in
      status", getDataObject("TicketDataObject")/tns:status)
    </from>
    <to>getDataInput("AnswerDataInputOfSendTask")/tns:body</to>
  </assignment>
</dataInputAssociation>
</sendTask>
```

Figure 20-6. Mapping example using assignment/from and assignment/to in XPATH

Transformation Mapping

Assignment/from and *assignment/to* map data elements one at a time. Alternatively, a single *transformation* element can be used to map the data association's *sourceRef* element to the *targetRef* element. Unfortunately, *transformation* is defined in the spec as a single expression of type *tFormalExpression*. It is not clear if, say, an XSLT 2.0 transformation could be used here. XSLT is not really an expression language, and there is no way for *transformation* to reference an external XSLT file. The contents of the XSLT could be copied into the *transformation* element as a CDATA section.

Script Task Mapping

A more practical way to implement complex data mapping in BPMN is to use a *scriptTask*. A *scriptTask* is code, embedded in the BPMN, that is executed on the process engine. (That distinguishes it

from a *serviceTask*, in which the process engine invokes some function provided by some other system.) A script is a set of statements, a program, not just a single expression. The script languages supported will vary from one process engine to the next. They could include Javascript or Groovy. BPMN 2.0 Common Executable subclass does not require support for any particular script language.

scriptTask has a *scriptFormat* attribute that specifies the script language as a MIME type string, such as *text/x-groovy* (Groovy) or *application/x-javascript* (Javascript). A child *script* element contains the script text, which may be enclosed in a CDATA section to prevent XML parsing of the script.

The fragment below from the OMG *Incident Management* example illustrates populating a *dataOutput* from a Groovy script.

```
<scriptTask name="Open ticket" scriptFormat="text/x-groovy" id="_1-26">
  <ioSpecification>
    <dataInput itemSubjectRef="tns:IssueItem"
      id="IssueDataInputOfScriptTask" />
    <dataOutput itemSubjectRef="tns:TicketItem" id="TicketDataOutputOfScriptTask"/>
    <inputSet>
      <dataInputRefs>IssueDataInputOfScriptTask</dataInputRefs>
    </inputSet>
    <outputSet>
      <dataOutputRefs>TicketDataOutputOfScriptTask</dataOutputRefs>
    </outputSet>
  </ioSpecification>
  <dataInputAssociation>
    <sourceRef>IssueDataInputOfProcess</sourceRef>
    <targetRef>IssueDataInputOfScriptTask</targetRef>
  </dataInputAssociation>
  <dataOutputAssociation>
    <sourceRef>TicketDataOutputOfScriptTask</sourceRef>
    <targetRef>TicketDataObject</targetRef>
  </dataOutputAssociation>
  <script><![CDATA[
    issueReport = getDataInput("IssueDataInputOfScriptTask")

    ticket = new TroubleTicket()
    ticket.setDate = new Date()
    ticket.setState = "Open"
    ticket.setReporter = issueReport.getAuthor()
    ticket.setDesctiption = issueReport.getText()

    setDataOutput("TicketDataOutputOfScriptTask", ticket)
  ]]></script>
</scriptTask>
```

Figure 20-7. Data mapping example using Groovy script. Source: OMG

Chapter 21

Services, Messages, and Events

Services

Except for scripts embedded in the XML itself, BPMN 2.0 assumes an automated task is a *service* invoked by the process. The BPMN 2.0 metamodel defines the basic elements of a service. Unlike BPEL, BPMN does not require a web service implementation, but it does assume the service has an *interface* with enumerated *operations* invoked by *messages*.

interface

A service *interface* is a root element in the BPMN XML, containing *name* and one or more *operation* elements. The *interface* element also has an optional *implementationRef* attribute that points to a concrete implementation artifact representing the interface, such as a WSDL *portType*.

A *participant* in a collaboration may reference a number of *interface* and *endpoint* elements. The actual definition of the service address is out of scope of BPMN 2.0. The *endPoint* may be specified, via WS-Addressing or equivalent, using *extensionElements*.

operation

An *operation* defines the *message* elements used for the request, response, and errors. Each *operation* must have a *name*, unique in its namespace, and exactly one *inMessageRef*, a pointer to the request (input) message. If the *operation* returns a response, it also specifies an *outMessageRef* as well as zero or more *errorRef* elements. *errorRef* does not point to a *message* but to a root *error* element. It may also provide an *implementationRef* that points to a concrete implementation artifact representing the operation, such as a WSDL *operation*.

Messages

Each *message* used in an executable process should be declared in a root element of the model. The *message* element provides a *name* and an *itemRef* that points, by prefixed *id*, to an *itemDefinition*. The *itemDefinition* in turn has a *structureRef* that points, by *name*, to a data structure definition such as an element or complex type in an imported XSD or WSDL

BPMN supports a range of message implementations, but generally assumes that the message is composed of a *header*, used for endpoint addressing, quality of service, and security, and a *payload* that holds the message content. The *dataInput* and *dataOutput* of BPMN Message events and Send or Receive tasks map to the message *payload* only, not the header.

The BPMN spec describes a *CorrelationKey* mechanism for binding a message to a particular process instance at runtime, but its use is restricted to *Conversation models*, a special form of collaboration oriented to B2B interactions. I have never seen Conversations used in practice, and they are not covered in this book, but the need to identify the target process instance of an incoming message is universal in executable processes, even in BPM Suites that do not support collaboration or message flows at all. For that very common use case, each BPM Suite must provide its own correlation implementation through an instance ID value embedded in the message payload. A standard way to implement message correlation without Conversations would appear to be a major omission in the BPMN 2.0 spec.

Automated Tasks

serviceTask

A *serviceTask* is a task that automatically invokes a service *operation*. Its *implementation* attribute specifies the technology used to send the invocation message and receive the response. If omitted, the default value `##WebService` is implied. Alternatively, *implementation* may contain a URI specifying another messaging technology, or `##unspecified` to leave the implementation open. The optional attribute *operationRef* (required for web service implementation) points by QName to an *operation* in a service *interface*.

A *serviceTask* has a single *dataInput* with *itemDefinition* equivalent to that of the *message* defined by the referenced *operation's inMessageRef*. Similarly, if the service returns output, the *serviceTask* has a single *dataOutput* with *itemDefinition* equivalent to that of the *message* defined by the *operation outMessageRef*. At execution, the process engine copies the task *dataInput* to the input *message* payload, and copies the returned output *message* payload to the task *dataOutput*.

Again, the *Incident Management* example from OMG provides a simple illustration.

```
<process isExecutable="true" id="WFP-1-1">
  ...
  <dataObject id="TicketDataObject" itemSubjectRef="tns:TicketItem" />
  ...
  <serviceTask name="Insert issue into product backlog"
    operationRef="tns:addTicketOperation" id="_1-325">
    <ioSpecification>
      <dataInput itemSubjectRef="tns:TicketItem" id="TicketDataInputOf_1-325" />
      <inputSet>
        <dataInputRefs>TicketDataInputOf_1-325</dataInputRefs>
      </inputSet>
      <outputSet />
    </ioSpecification>
    <dataInputAssociation>
      <sourceRef>TicketDataObject</sourceRef>
      <targetRef>TicketDataInputOf_1-325</targetRef>
    </dataInputAssociation>
  </serviceTask>
  ...
</process>
<interface name="Product Backlog Interface"
  implementationRef="java:com.camunda.examples.incidentmanagement.ProductBacklog">
  <operation name="addTicketOperation" implementationRef="addTicket"
    id="addTicketOperation">
    <inMessageRef>tns:AddTicketMessage</inMessageRef>
  </operation>
</interface>
  ...
*****ebook converter DEMO Watermarks*****
```

```

<message id="AddTicketMessage" name="addTicket Message" itemRef="tns:TicketItem" />
...
<itemDefinition id="TicketItem" isCollection="false" itemKind="Information"
  structureRef="com.camunda.examples.incidentmanagement.TroubleTicket" />

```

Figure 21-1. Service task definition in BPMN 2.0. Source: OMG

The *serviceTask* named *Insert issue into product backlog* has a single *dataInput* that references the *itemDefinition* *TicketItem*. Its *operationRef* points to an *operation* named *addTicketOperation*. That *operation* has the input *message* *AddTicketMessage*. (Note that the example points to the *message* by prefixed *id* rather than by a unique *name*.) Both the *message* data, identified by *itemRef*, and the task *dataInput* point to the same *TicketItem* element.

sendTask

sendTask works almost the same as *serviceTask*, except that there is, by definition, no response *message*. A failed *sendTask* *operation* may, however, return *errorRefs*. Optional attributes *implementation* and *operationRef* are specified exactly as in *serviceTask*. Optional attribute *messageRef* points to the *message* by prefixed *id*. If an *operation* is specified, the *message* datatype must match that of the task *dataInput*.

receiveTask

A *receiveTask* waits for a *message* identified by attribute *messageRef*. It may also reference an *operation*, indicating the *message* is a response to an asynchronous service invoked previously. In that case, the *message* payload datatype must match that of the task *dataOutput*. The optional Boolean attribute *instantiate* is allowed only if the *receiveTask* has no incoming sequence flow, making it an implicit start node of the process. A value of *true* signifies instantiation of the process when the message is received. Method and Style recommends use of *Message startEvent* to signify this behavior instead of *instantiate* on *receiveTask*.

businessRuleTask

A *businessRuleTask* is intended to invoke an automated decision from a business rule engine. In that sense, it sounds like a special use case of a *serviceTask*. However, unlike *serviceTask*, *businessRuleTask* does not specify an *operation*, so its use in practice effectively requires proprietary *extensionElements*.

Events

Message Events

A *Message event* is any event, whether throwing or catching, with a *messageEventDefinition*. The *messageEventDefinition* element has optional attributes *messageRef* and *operationRef* that work exactly the same as in *sendTask* and *receiveTask*. *messageEventDefinition* is usually specified as a child of a specific Message event element, but BPMN allows a single *messageEventDefinition* to be reused by specifying it as a root element and then pointing to it from the *eventDefinitionRef* child of multiple message event elements.

Signal Events

A *Signal event* is any event, throwing or catching, with a *signalEventDefinition*. As with message events, the *signalEventDefinition* may be specified for each Signal event or by reference to a reusable root element. However, the *signalEventDefinition* only provides a *signalRef* pointer to a root *signal* element, which has attributes *id*, *name*, and *structureRef* pointing to an *itemDefinition* by prefixed *id*. Similar to message, a Signal catch event copies the trigger payload to a *dataOutput* of the event, implying that *dataOutput* must be of the same datatype, and a Signal throw event copies the *dataInput* to the thrown signal payload.

Error and Escalation Events

Error and Escalation events work the same way. An *errorEventDefinition* or *escalationEventDefinition* provides merely a pointer to a root *error* or *escalation* element that provides attributes *id*, *name*, *errorCode* or *escalationCode*, and *structureRef*. The *structureRef* is a QName that points to an *itemDefinition* by *id*. (I think this is a bug in the XSD, since *itemDefinition* also has attribute *structureRef* that points to an imported element or complex type by *name*, not *id*. The pointer by *id* to *itemDefinition* should be named *itemRef* or *itemSubjectRef*... not *structureRef*.)

errorCode and *escalationCode* are simple strings used to match throw-catch pairs. Throwing events must provide it, but it is optional for boundary events. (This actually kind of strange, since *errorCode* belongs to the reusable *error* element, not to a specific event.) An Error *boundaryEvent* will catch any *error* signal with matching *errorCode* thrown from a child level event, and similarly for Escalation. If *errorCode* is omitted, the *boundaryEvent* will catch any *error* thrown from the child level.

The referenced *itemDefinition*, if it exists, specifies the structure of the *error* or *escalation* payload. Similar to Message and Signal events, upon execution the *dataInput* of a throwing Error event is copied to the *error* payload, which is then propagated to the *dataOutput* of the Error *boundaryEvent*. Again, the datatype of the *error itemDefinition* must match that of the Error event *dataInput* or *dataOutput*. Escalation works the same way.

Timer Events

Timer events do not transmit or receive data, so they have no *dataInput* or *dataOutput*. The *timerEventDefinition* specifies the deadline though one of three child elements, all of type *tExpression*: *timeDate*, *timeDuration*, or *timeCycle*.

The *timeDate* expression, which in most cases is a literal string, must resolve to a value consistent with ISO-8601 time and date formats. This encompasses quite a wide range of formats; for interoperability, I recommend use of the XSD *date*, *time*, and *dateTime* types, which are consistent with ISO-8601.

The *timeDuration* expression must resolve to a value consistent with ISO-8601 time interval formats. These take the form P[n]Y[n]M[n]D[n]TH[n]M[n]S or P[n]W. Here [n] is replaced by a number indicating the quantity of the units specified by the preceding letter; if the value is zero, the letter and [n] may be omitted. P always starts the expression; Y, M, and D stand for years, months, and days; T starts the time part of the expression; H, M, S means hours, minutes, seconds; and W means weeks. Thus, for example P4M means 4 months, and PT4M means 4 minutes.

The *timeCycle* expression is reserved for *repeating intervals*, such as in a Timer *startEvent* or non-interrupting Timer *boundaryEvent*. (In BPMN 1.2, *durations* used an attribute named *timeCycle*, so this could be a source of confusion in BPMN 2.0.) The *timeCycle* expression value must be consistent with ISO-8601 for repeating intervals. Again, ISO-8601 allows many options for this, all starting with R[n]/, where [n] indicates the number of repetitions (unbounded if omitted), and continuing either with start and end *dateTime* separated by /, or start plus duration separated by /, or duration plus end separated by /, or just duration.

Thus, a Timer *startEvent* that occurs on September 11, 2011, Pacific Time, and every 7 days thereafter would have *timeCycle* evaluate to

R/2011-09-07T14:00:00-07:00/P7D

Chapter 22

Human Tasks

In an executable process, a *userTask* signifies a human task managed by the process engine. A *manualTask* signifies some human activity that is not managed by the process engine. We will focus here on the specification of a *userTask* and its associated *resource*.

userTask

The *implementation* attribute of a *userTask* may be `##WebService`, `##unspecified`, or a URI indicating another technology or coordination protocol. For example, a value of `http://docs.oasis-open.org/ns/bpel4people/ws-human-task/protocol/200803` signifies WS-HumanTask as the implementation.

Optional child element *rendering* provides a hook to specify, via tool-proprietary *extensionElements*, details of the task user interface. Input and output data of a *userTask* is specified in the *ioSpecification* element, as with any other type of activity.

Two *instance attributes* of *userTask* are accessible for use in expressions via the *getInstanceAttribute* function:

- *actualOwner*, a string uniquely identifying a single user who has claimed or is performing the task.
- *taskPriority*, an integer used to sort *userTask* instances in a queue.

Performer Assignment

BPMN allows the modeler to specify any number of *resource* root elements that may be referenced by an activity, whether human or automated, as playing some *resourceRole*. Each *resource* represents a static list of users belonging to a certain role or organizational unit. How the universe of all users is assigned to each *resource* is outside the scope of BPMN. In the metamodel, *resourceRole* is an abstract class. Its only defined subclass is *performer*, which in turn has subclass *humanPerformer*, which in turn again has subclass *potentialOwner*. Each subclass represents a particular specialization of the parent class, and the spec invites implementers to define their own subclasses. However, the only element actually spelled out in the spec is *potentialOwner*, meaning the set of individuals allowed as performers of a particular *userTask*. When one member of that set claims or performs the task, it is identified as the task's *actualOwner*, an instance property.

There are two alternative ways to specify task assignment to *potentialOwner*: by *parameterized query*, or by *expression assignment*. Assignment by expression is more convenient when each *resource* is defined as a very specific role, group, or capability, and the *potentialOwner* of a *userTask* is several of them. On the other hand, if each *resource* represents a broad group of users differing in specific role, organizational unit, or capability, assignment by parameterized query allows task assignment to a subset of the *resource*.

Task Assignment by Parameterized Query

Parameterized query assumes each member of a *resource* exposes a set of *parameters*. The root element *resource* must have a *name* and may contain a list of child *resourceParameter* elements used with parameterized queries. Each *resourceParameter* has attributes *id*, *name*, *type*, and Boolean *isRequired*. Here *type* is either a simple type or a pointer to an *itemDefinition*, identifying the datatype of the parameter.

With selection by parameterized query, *potentialOwner* must contain child *resourceRef* that points to a *resource* element containing *resourceParameters*, plus any number of child *resourceParameterBinding* elements, each a formal expression of *resourceParameters*. If no *resourceParameterBindings* are provided, *all* members of the *resource* become members of *potentialOwner*.

The following parameterized query scenario is an extension of the Incident Management example from OMG:

```
...
<resource id="FirstLevelSupportResource" name="1st Level Support" />
<resourceParameter id="product" isRequired="true" name="Product" type="xsd:string"/>
<resourceParameter id="region" isRequired="true" name="Region" type="xsd:string"/>
</resource>

...
<process isExecutable="true" id="WFP-1-1">
...
<userTask name="edit 1st level ticket" id="_1-77">
  <ioSpecification>
    <dataInput itemSubjectRef="tns:TicketItem" id="TicketDataInputOf_1-77" />

```

```

<dataOutput itemSubjectRef="tns:TicketItem" id="TicketDataOutputOf_1-77" />
<inputSet>
  <dataInputRefs>TicketDataInputOf_1-77</dataInputRefs>
</inputSet>
<outputSet>
  <dataOutputRefs>TicketDataOutputOf_1-77</dataOutputRefs>
</outputSet>
</ioSpecification>
<dataInputAssociation>
  <sourceRef>TicketDataObject</sourceRef>
  <targetRef>TicketDataInputOf_1-77</targetRef>
</dataInputAssociation>
<dataOutputAssociation>
  <sourceRef>TicketDataOutputOf_1-77</sourceRef>
  <targetRef>TicketDataObject</targetRef>
</dataOutputAssociation>
<potentialOwner>
  <resourceRef>tns:FirstLevelSupportResource</resourceRef>
<resourceParameterBinding parameterRef="tns:product">
  getDataInput("TicketDataInputOf_1-77")/product
</resourceParameterBinding>
<resourceParameterBinding parameterRef="tns:region">
  getDataInput("TicketDataInputOf_1-77")/region
</resourceParameterBinding>
  </potentialOwner>
</userTask>
...
</process>

```

Figure 22-1. Human task assignment by parameterized query. Source: OMG

The *userTask* 'edit 1st level ticket' has *potentialOwner/resourceRef* that points by *id* to the *resource FirstLevelSupportResource*, a list of all first level support resources. That *resource* has two required parameters, *product* and *region*, meaning each member of this list must have a *product* value and a *region* value. Here we want the *potentialOwner* of this particular *userTask* to be just specialists in the *product* referenced in the *TicketItem* and in the requester's *region*. The *resourceParameterBinding* elements select members of the *resource* that satisfy the both query conditions, which are XPath expressions of the *TicketItem dataInput*.

Task Assignment by Expression

Alternatively, *potentialOwner* may replace *resourceRef* and *resourceParameterBinding* with child *resourceAssignmentExpression*. This element contains child element *expression*, a formal expression that evaluates to one or more *resources*, e.g., by OR-ing them together.

Chapter 23

Executable BPMN in Practice

From the preceding discussion it would be easy to imagine that executable process design in BPMN 2.0 is a matter of populating values in the XML structure. But that's not at all how it's done. Real BPM Suites employ graphical editors that streamline the work of defining process data, mapping it to and from task input/output parameters, assigning human task performers, and similar aspects of executable design. The BPMN 2.0 XML we've been discussing represents simply an interchange format for the executable design. All the tedious data association mappings required in the XML – *inputSet* to *dataInput*, *dataInput* to *itemDefinition*, *itemDefinition* to imported XSD element – are created automatically under the covers by the tool. The modeler doesn't have to think about it.

If a tool can export its executable models in accordance with the BPMN 2.0 schema, I call that executable BPMN 2.0. If you are using a BPMN-based BPM Suite, you might ask your vendor, "Well, how hard could that be?" But you would be surprised. As of this writing, more than a year after publication of the final BPMN 2.0 spec, commercial BPM Suites are only *almost* there.

In this chapter we'll discuss some of the differences between the way executable processes are designed in real tools and how they are serialized in BPMN 2.0. We'll see examples of how that works with an open source BPMN 2.0-based BPMS called Bonita Open Solution (BOS) ^[26] from BonitaSoft.

Handling Java Data

The BPMN 2.0 specification provides explicit requirements and examples for importing, referencing, and mapping XML data, but is silent on how to handle data as it is most commonly defined by developers, in Java or some similar programming language. The BPMN spec is explicit in allowing non-XML type definitions, but it does not say exactly how to do it.

Referencing Java Data

As a practical matter, if you use a programming language like Java to define process data, you are inevitably bound by the conventions of a particular tool or IDE. It is not quite as “standard” as XML data. Be that as it may, executable process design has traditionally been the domain of developers, and it is not at all uncommon to find that your BPM Suite is using Java types, not XSD, to define BPMN process data.

In that case, how should that data be referenced in the BPMN 2.0 XML? The BPMN spec leaves each tool to define its own conventions. One convention is to use a *pseudo-namespace prefix* such as *java*: to identify elements from a Java namespace, and within that namespace use Java qualification rules to refer to a simple type, Java class or nested class. For example, with simple types like

```
<itemDefinition id="item001" structureRef="java:float"/>
```

the data type of the *itemDefinition* is clear from the BPMN XML. But that is not so with complex business objects:

```
<itemDefinition id="item002" structureRef="java:myClass.nestedClass"/>
```

With complex XML data, the *structureRef* points to an element or complex type from an XSD file referenced by an *import* element, but this is usually not the case with Java data. OMG’s Incident Management example we looked at previously had no such *import*, for instance. According to Falko Menge of Camunda, author of that example, “These *structureRefs* are fully-qualified Java class names, which is the standard way of identifying a Java class. A Java-based Engine is able to load the class using that name. An import would only be needed if the package name, e.g., *com.camunda.examples.incidentmanagement*, is not specified. However, all Java-based engines that I know just use fully-qualified class names. Note that the XML shown in *BPMN 2.0 by Example* has been created before any Java-based Engine existed and is therefore just a suggestion on how Java-Code could be referenced.”

[\[27\]](#)

For a developer in a Java IDE, there is no problem in accessing and inspecting the classes used in the model, whether or not there is an *import* element in the BPMN file. The problem is that outside such a tool, the process data definitions are invisible. While the model is indeed executable, it seems to violate the spirit of a *transparent*, standards-based serialization.

To increase transparency, you could always *import* the Java classes. Here the file *Example.java* defines the data:

*****ebook converter DEMO Watermarks*****

```

package org.bonitasoft.bpmn;
public class Example {
    public String att1;
    public int att2;
    public InternType att3;
    public class InternType{
    }
}

```

and in the BPMN, an *import* element points to it:

```

<import importType="http://jcp.org/en/jsr/detail?id=270" location="Example.java"
        namespace="http://jcp.org"/>

```

Once imported, a class can be referenced by *itemDefinition*:

```

<itemDefinition id="itemX" structureRef="java:org.bonitasoft.bpmn.Example$InternType"/>

```

In any case, referencing Java data in BPMN 2.0 is likely to remain implementation-specific.

Besides the issue of defining process data in Java, there is the question of how to use that data in data association mappings, gateway conditions, and other expressions required by the process model. With XML data, element references and expressions typically use XPATH 1.0, which is the BPMN 2.0 default. With Java data, there is no generally agreed way to do it. Implementations may reference elements using the Java “dot” notation, or use something like XPATH when the structure is XML but the individual element types are Java.

For expressions involving Java data, BPMN 2.0 tools seem to be using either UEL or Groovy.

UEL

[28]

Activiti²⁸, for example uses UEL. It stands for *Unified Expression Language* and is part of the Java EE6 specification^[29]. UEL supports two types of expressions, *value expressions* and *method expressions*. Depending on the implementation, either may be used for expressions in BPMN. These expressions can be used to resolve and compare primitives, beans, lists, arrays and maps. A value expression resolves to a value. In UEL, variables and bean (object) properties are referenced using the following syntax:

```

${myVar}
${myBean.myProperty}

```

A method expression invokes a method, with or without parameters. Parameters may be literal values or expressions. The syntax is as shown below:

```

${printer.print()}
${myBean.addNewOrder('orderName')}
${myBean.doSomething(myVar, execution)}

```

[30]

The following examples, from the Activiti 5.7 User Guide^[30], show how UEL is used in condition expressions on sequence flows out of an XOR gateway. In this example,

```

<conditionExpression xsi:type="tFormalExpression">
<![CDATA[ ${order.price > 100 && order.price < 250} ]]>

```

```
</conditionExpression>
```

UEL defines a value expression referencing process variables.

In this example,

```
<conditionExpression xsi:type="tFormalExpression">
  <![CDATA[ ${order.isStandardOrder()} ]]>
</conditionExpression>
```

UEL uses a method expression that returns a Boolean value.

To prevent XML processors from parsing the UEL, it is best to enclose it in a CDATA section in the BPMN model, as shown above.

Groovy

[31]

BonitaSoft uses Groovy^[31]. Groovy is an object-oriented programming or scripting language for the Java platform. It is a dynamic language with features similar to those of Python, Ruby, Perl, and Smalltalk. Groovy is dynamically compiled to Java Virtual Machine (JVM) bytecode and interoperates with other Java code and libraries. Most Java code is also syntactically valid Groovy.

Groovy can be used both for expressions and for full scripts. As an expression language, it offers advantages over pure Java, as illustrated by this example from the JasperForge website^[32]:

Expression	Java	Groovy
Field	<code>\$F{field_name}</code>	<code>\$F{field_name}</code>
Sum of two double fields	<code>new Double(\$F{f1}.doubleValue() + \$F{f2}.doubleValue())</code>	<code>\$F{f1} + \$F{f2}</code>
Comparision of numbers	<code>new Boolean(\$F{f}.intValue() == 1)</code>	<code>\$F{f} == 1</code>
Comparision of strings	<code>new Boolean(\$F{f} != null && \$F{f}.equals("test"))</code>	<code>\$F{f} == "test"</code>

Figure 23-1. Java versus Groovy expressions. Source: JasperForge

As with UEL, Groovy expressions may include methods. They use a similar \${ } notation and may include special characters; if so, the expression should be enclosed in CDATA. The following Groovy example from BonitaSoft means the condition on this sequence flow is if the variable named *available* has value *false*:

```
<conditionExpression xsi:type="tFormalExpression">
  ${!available}
</conditionExpression>
```

Are XPATH Data Access Functions Needed?

The BPMN 2.0 specification does not allow XPATH to directly reference item-aware elements (*dataObject*, *dataInput*, *property*, etc.). Instead, it requires special XPATH *extension functions* like *getdataObject('[data object id]')*. The reason, supposedly, for it is to establish unambiguously the context

*****ebook converter DEMO Watermarks*****

node of the XPATH reference, although it seems to me that

dataObject[@id='data object id']

accomplishes the same thing. The spec does not say whether such functions are required when other expression languages, such as UEL or Groovy, are employed. I would think they are not.

The Incident Management process in the OMG's non-normative *BPMN 2.0 by Example* document, which we have referenced several times in this book, does use these functions for referencing Java data in UEL expressions. However, the examples from the Activiti website, which also use UEL, do not use these functions.

The test of whether or not the functions are needed comes down to the executable implementation. Elements of type *tFormalExpression* in the BPMN XML should have values as they are required in the executable design. If the process engine does not need *getDataObject()* for proper execution, then it should not appear in the *conditionExpression*.

Services and Service Adapters

[33]

One critical difference between BPMN 2.0 as an executable design language and BPEL^[33], an older process execution language standard from OASIS, is that BPEL assumes all tasks are implemented as *web services* described by WSDL and invoked by SOAP messages, but BPMN does not. In BPMN 2.0, a *serviceTask* may be implemented as a SOAP-based web service, but that is not the only option. It could be a RESTful service, a Java remote procedure call, or any other implementation supported by the process engine.

In particular, most commercial BPM Suites provide *service adapters* (sometimes called *connectors*) that expose a user-configurable *service interface* for any number of functions provided both by the BPMS itself and by external systems. For example, reading or writing a file, sending email, performing a database lookup, and adding a new customer in the ERP system are all functions typically implemented by a service adapter.

BPM Suites vary widely in the architecture and configuration of their adapters. However, BPMN 2.0 does impose certain web service-like constraints on their specification in the process model XML, as described in Chapter 21:

- A service adapter must specify an *interface* with one or more *operations*.
- Each *operation* must specify a single input *message* and single output *message*.
- A *serviceTask* must reference exactly one of those *operations*.
- The *serviceTask* must have a single *dataInput* and (if the *operation* returns a response) a single *dataOutput*.
- The *serviceTask dataInput* must be of the same type as the *operation's* input *message*, and the same goes for the output.

Thus, even if the service adapter implementation has no native concept of input and output messages, the implementer needs to specify those constructs to satisfy the BPMN 2.0 metamodel.

Example: Bonita Open Solution

To illustrate the relationship between executable design using service adapters in a real BPMS and its serialization in the model, we'll use a simple process created in BOS from BonitaSoft. BOS claims to be the only complete open-source BPMS. Its process engine is not built natively on BPMN 2.0 from the ground up, but BonitaSoft is committed to serialization of its process models in a manner fully compliant with the BPMN 2.0 standard. The BPMN 2.0 export from the current version, BOS v5.6, isn't exactly as presented here, but the company plans to implement a BPMN 2.0 export very close to this in BOS v6, scheduled for early 2012.

The BPMN and XSD files excerpted in the following example are provided in downloadable form on the book website www.bpmnstyle.com.

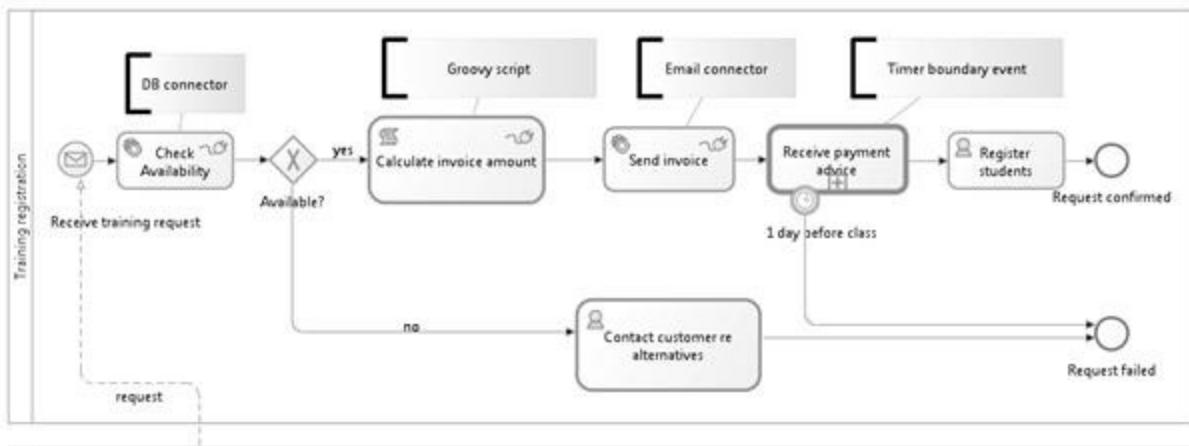


Figure 23-2. Training request process in BOS. Source: BonitaSoft

The example shown in Figure 23-2 illustrates a training request process. It starts upon receipt of a training request message, labeled *request*. This is an XML message that contains the requester name and contact information, the requested course ID and date, and number of students. The first step is a service task that checks the availability of the requested course through a database lookup. The result of the lookup is tested in a gateway, and if the course is available, a script then calculates the invoice amount. Another service task sends the invoice via email, then continues to wait for notification from a payment service provider that the invoice has been paid. Upon receipt of payment, a user task registers the students. If payment notification is not received by one day before the class begins, the registration request fails.

The current implementation of BOS does not support event gateway or a timer boundary event on a receive task, but it does support timer boundary event on a call activity, and that is what is illustrated here. The process called by the call activity just contains a catching Message event (plus start and end events).

The database lookup and the email are implemented as *BonitaSoft connectors*, BOS's term for service adapters. We'll see how their configuration in the tool is expressed in the BPMN 2.0 export, as well as other aspects of executable design.

Defining Process Variables

The variables for this process are illustrated in Figure 23-3, along with the dialog for adding new ones. In addition to simple datatypes like Text, Integer, Float, Date, or Boolean, BOS supports XML or a Java object types. As discussed earlier, I believe the “spirit” of BPMN as a standard is to *expose* the process data definitions in the serialization, not *hide* them. That means the tool must export, in addition to the .bpmn file, *data definition files* such as XSD, and reference those files by *import* in the BPMN XML.

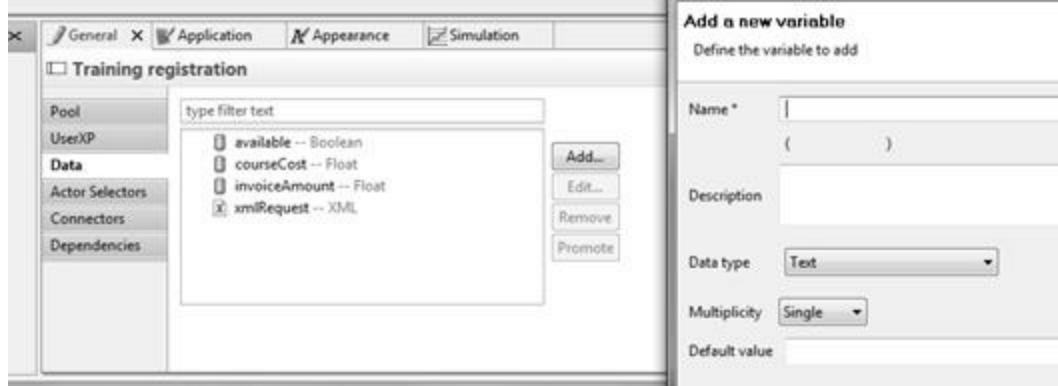


Figure 23-3. Process variable definition in BOS. Source: BonitaSoft

In the BPMN 2.0 XML, each variable is represented by a *dataObject* and its corresponding *itemDefinition*.

Simple types can be specified directly in *itemDefinition/@structureRef* by referencing a basic XSD or Java type. For example the Boolean variable *available* is represented in the BPMN XML as follows [34] :

```
<model:itemDefinition id="item04" structureRef="xsd:boolean"/>
...
<model:process>
...
<model:dataObject id="available" name="available" itemSubjectRef="item04"/>
</model:process>
```

For XML variables, BOS exports a data definition XSD file based on *Ecore* [35], which stands for *Eclipse Modeling Framework core*. EMF is a modeling framework and code generation facility that generates Java classes for modeled business objects. *Ecore.xsd*, imported by this data definition file, defines the data types. In our *Training request process* example, the request message is based on the variable *xmlRequest*, which generates the following data definition file:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<xsd:schema xmlns:XMLRequest="http://www.bonitasoft.org/complexTypes"
  xmlns:.ecore="http://www.eclipse.org/emf/2002/Ecore" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.bonitasoft.org/XMLRequest" ecore:nsPrefix="TrainingRequest"
  ecore:package="TrainingRequest">
  <xsd:import namespace="http://www.eclipse.org/emf/2002/Ecore" schemaLocation="Ecore.xsd"/>
  <xsd:element name="Request" type="XMLRequest:tRequest" ecore:ignore="true"/>
  <xsd:complexType name="tRequest">
    <xsd:attribute name="requesterName" type=".ecore:EString"/>
    <xsd:attribute name="address" type=".ecore:EString"/>
    <xsd:attribute name="email" type=".ecore:EString"/>

```

```

<xsd:attribute name="courseId" type="ecore:EString"/>
<xsd:attribute name="courseDate" type="ecore:EDate"/>
<xsd:attribute name="numStudents" type="ecore:EInt" ecore:unsettable="false"/>
</xsd:complexType>
</xsd:schema>

```

Figure 23-4. Exported data definition file for XML data using Ecore. Source: BonitaSoft

This data definition file is then *imported* by the BPMN file and referenced by *itemDefinition/@structureRef*:

```

<model:import importType="http://www.w3.org/2001/XMLSchema" location="XMLRequest.xsd"
  namespace="http://www.bonitasoft.org/complexTypes"/>
<model:message id="message01" name="request" itemRef="item01"/>
<model:itemDefinition id="item01" structureRef="n2:Request"/>
...
<process>
...
<model:dataObject id="xmlRequest" name="xmlRequest" itemSubjectRef="item01"/>
</process>

```

The XML variable in BOS generates the *dataObject* named *xmlRequest*, which points to the *itemDefinition* that points to the *Request* element in the imported data definition file (namespace prefix *n2:*). The *message* named *request* references the same *itemDefinition*.

For Java types (there are none in this example), *itemDefinition* directly references the Java class, as illustrated earlier in this chapter.

Saving the Request Message

Upon receipt of the *request* message, the process first must save its contents in a variable, the *dataObject* named *xmlRequest*. This is not automatic; it must be explicitly described in the BPMN. What is automatic is copying the *message* payload to the *dataOutput* of the Message *startEvent*. From there you need to use a *dataOutputAssociation* of the *startEvent* to map to the *dataObject*. Since they have the same type, that is simple:

```

<model:startEvent id="Receive_training_request" name="Receive training request">
  <model:dataOutput id="Receive_training_request_out" itemSubjectRef="item01"/>
  <model:dataOutputAssociation>
    <model:sourceRef>Receive_training_request_out</model:sourceRef>
    <model:targetRef>xmlRequest</model:targetRef>
  </model:dataOutputAssociation>
  <model:messageEventDefinition id="msgEvent01" messageRef="message01"/>
</model:startEvent>

```

Note also that the *messageEventDefinition* points to the *message* with id *message01*, which is the start message named *request*.

Service Task – Database Lookup

Next the process executes a database lookup in the *serviceTask Check Availability*. This task is implemented by a *Bonita Connector*, what I have called a *service adapter*. BonitaSoft provides many connectors itself, and receives many more from its open source community. In this case the process uses the *MySQL Connector* to execute a SQL query.

Each connector is configured in point-click fashion through a wizard. Figure 23-5 shows the two input configuration screens for the connector. Figure 23-6 shows mapping of the connector output to process variables.

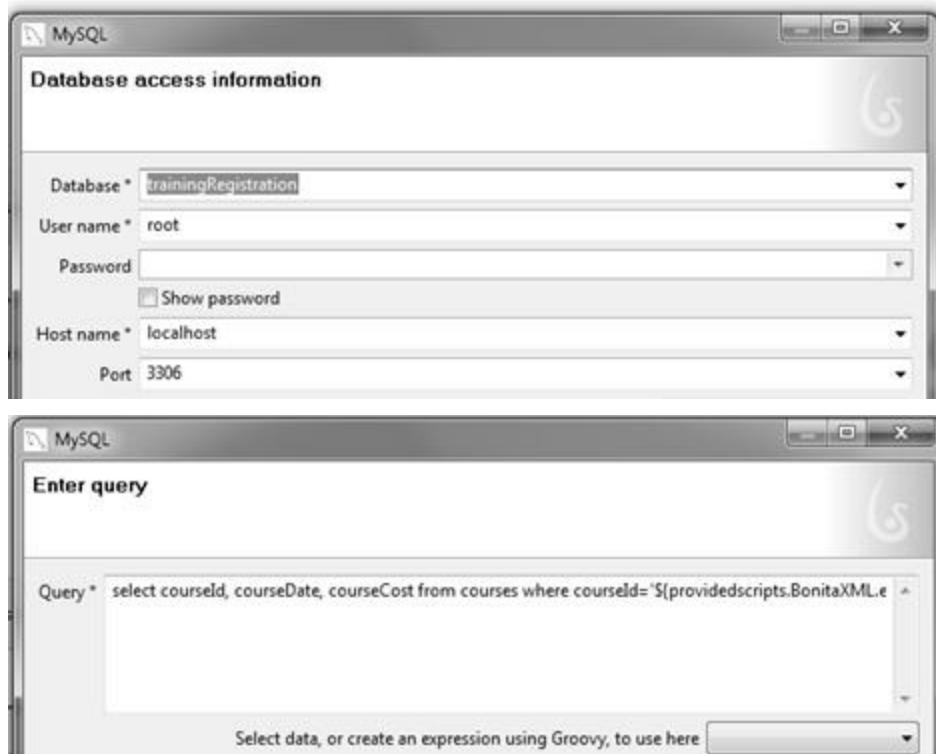


Figure 23-5. BOS MySQL Connector input configuration. Source: BonitaSoft



Figure 23-6. BOS MySQL Connector output mapping configuration. Source: BonitaSoft

Here the database connection information is static text passed to the connector, but the SQL query string requires instance data, such as the *courseId*, *courseDate*, etc. You can see in Figure 23-5 a Groovy expression embedded in the SQL query string. The question is how to represent all this in the BPMN 2.0 XML.

The BPMN 2.0 conceptual model says that a connector such as this represents an *interface*. Each instance of the connector in the process model represents an *operation* with a single input *message* and a single output *message*. And to describe the parameters contained in those messages and their datatypes, we need an *itemDefinition* and *structureRef* for each one.

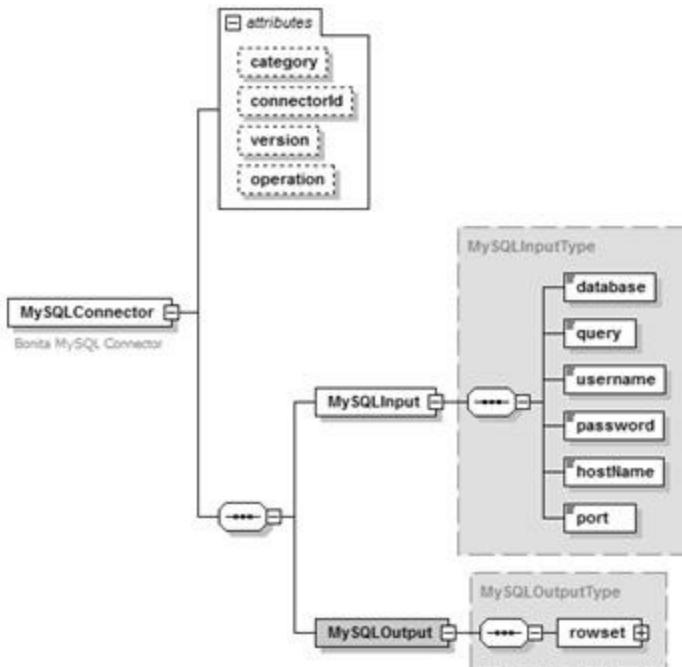


Figure 23-7. MySQL Connector in Connectors.xsd. Source: BonitaSoft

For this, BOS also generates a *Connectors.xsd* file in the BPMN export, containing the input and output parameters. As with the *xmlRequest* data definition file, BPMN *imports* this XSD and references it in the *serviceTask* *dataInput* and *dataOutput*. Figure 23-7 is a graphical representation of the *Connectors.xsd* element for the MySQL connector.

The BPMN 2.0 XML below describes the *serviceTask*, including its input and output mappings as described by the configuration wizard.

```

<!--request message is saved in dataObject id 'xmlRequest'-->
<model:serviceTask id="Check_Availability" name="Check Availability" implementation="BonitaConnector" operationRef="execMySQL">
    <!--operationRef points to the connector operation, which points to a message, which points to an itemDefinition, which points to imported data structure-->
    <model:ioSpecification>
        <model:dataInput id="Check_Availability_input" itemSubjectRef="item02"/>
        <model:dataOutput id="Check_Availability_output" itemSubjectRef="item03"/>
    <!-- dataInput and dataOutput point to same itemDefinition as the service interface inMessage-->
    <model:inputSet>
        <model:dataInputRefs>Check_Availability_input</model:dataInputRefs>
    </model:inputSet>
    <model:outputSet>
        <model:dataOutputRefs>Check_Availability_output</model:dataOutputRefs>
    </model:outputSet>
    </model:ioSpecification>
<!-- Map dataObject to dataInput-->

```

```

<model:dataProvider>
  <model:sourceRef>xmlRequest</model:sourceRef>
  <model:targetRef>Check_Availability_input</model:targetRef>
  <model:assignment>
    <model:from>"trainingRegistration"</model:from>
    <model:to>getDataInput('Check_Availability_input')/n1:database
    </model:to>
  </model:assignment>
  <model:assignment>
    <model:from>"root"</model:from>
    <model:to>getDataInput('Check_Availability_input')/n1:username
    </model:to>
  </model:assignment>
  <model:assignment>
    <model:from>"password"</model:from>
    <model:to>getDataInput('Check_Availability_input')/n1:password
    </model:to>
  </model:assignment>
  <model:assignment>
    <model:from>"localhost"</model:from>
    <model:to>getDataInput('Check_Availability_input')/n1:hostname
    </model:to>
  </model:assignment>
  <model:assignment>
    <model:from>"3306"</model:from>
    <model:to>getDataInput('Check_Availability_input')/n1:port</model:to>
  </model:assignment>
  <model:assignment>
    <model:from>'select courseId, courseDate, courseCost from courses where
courseId='${providedscripts.BonitaXML.evaluateXPathOnVariable(xmlRequest,
"/Request/@courseId")}'</model:from>
    <model:to>getDataInput('Check_Availability_input')/n1:query
    </model:to>
  </model:assignment>
</model:dataProvider>
<!-- Map connector output to variables-->
<model:outputAssociation>
  <model:sourceRef>Check_Availability_output</model:sourceRef>
  <model:targetRef>available</model:targetRef>
  <model:assignment>
    <model:from>!rowSet.getValues().isEmpty()</model:from>
    <model:to>getDataObject('available')</model:to>
  </model:assignment>
  <model:assignment>
    <model:from xsi:type="model:tFormalExpression"
      language="http://groovy.codehaus.org/"
      evaluatesToTypeRef="xsd:float"><![CDATA[ List<List<Object>> courses = rowSet.getValues();
      if(courses!=null &&!courses.isEmpty()) {
        course = courses.get(0);
        return course.get(2);
      }
    </model:from>
  </model:assignment>
</model:outputAssociation>

```

*****ebook converter DEMO Watermarks*****

```

    return 0; ]]></model:from>
    <model:to>getDataObject('courseCost')</model:to>
</model:assignment>
</model:dataOutputAssociation>
</model:serviceTask>

```

Figure 23-8. BOS *serviceTask* serialization for MySQL connector.

Branching at the Gateway

The value of the Boolean variable *available*, set by the database lookup, determines the flow at the XOR gateway. In BOS as in BPMN 2.0, the gate conditions are properties of the outgoing sequence flows, not of the gateway itself. Configuration of the path marked *yes* in Figure 23-2 is illustrated in Figure 23-9. In the Condition field, an expression builder lets the developer select from existing process variables for use in a Groovy expression or a decision table. In this case, the condition is simply *available*, i.e. if the value of this variable is *true*, then the *yes* path is enabled. The BPMN 2.0 XML for the gateway and gate conditions is shown in Figure 23-10.

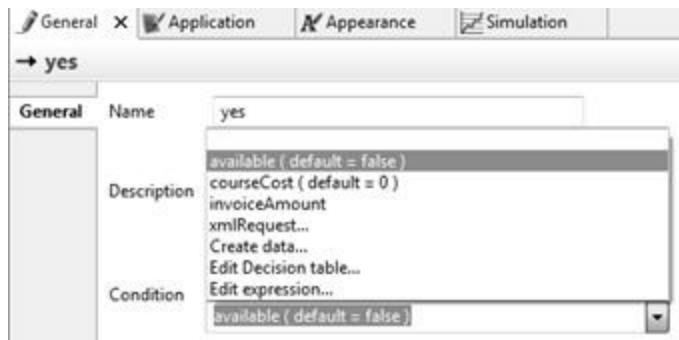


Figure 23-9. Defining a sequence flow condition in BOS. Source: BonitaSoft

```

<model:exclusiveGateway id="Available_" name="Available?" />
<model:sequenceFlow id="yes" name="yes" sourceRef="Available_"
    targetRef="Calculate_invoice_amount">
    <model:conditionExpression xsi:type="model:tFormalExpression" evaluatesToTypeRef="xsd:boolean">
        ${available}
    </model:conditionExpression>
</model:sequenceFlow>
<model:sequenceFlow id="no" name="no" sourceRef="Available_"
    targetRef="Contact_customer_re_alternatives">
    <model:conditionExpression xsi:type="model:tFormalExpression" evaluatesToTypeRef="xsd:boolean">
        ${!available}
    </model:conditionExpression>
</model:sequenceFlow>

```

Figure 23-10. Serialization of gateway and gate conditions

Script Task – Calculating the Invoice Amount

Simple calculations are typically performed in *scriptTask* elements. Here we need to calculate the *****ebook converter DEMO Watermarks*****

invoice amount based on the number of students (from *xmlRequest*) times *courseCost* (from the database lookup). We need to pass those data objects to the *dataInput* of the *scriptTask* using *dataInputAssociation*. The script language Groovy is indicated by the *scriptFormat* MIME type. The BPMN 2.0 XML is shown below:

```

<model:scriptTask      id="Calculate_invoice_amount"      name="Calculate      invoice      amount"
  scriptFormat="text/x-groovy">
<model:ioSpecification>
  <model:dataInput id="Calculate_invoice_amount_input" itemSubjectRef="item01"/>
  <model:dataOutput id="Calculate_invoice_amount_output" itemSubjectRef="item06"/>
  <model:inputSet>
    <model:dataInputRefs>Calculate_invoice_amount_input
    </model:dataInputRefs>
  </model:inputSet>
  <model:outputSet>
    <model:dataOutputRefs>Calculate_invoice_amount_output
    </model:dataOutputRefs>
  </model:outputSet>
</model:ioSpecification>
<model:dataInputAssociation>
  <model:sourceRef>xmlRequest</model:sourceRef>
  <model:targetRef>Calculate_invoice_amount_input</model:targetRef>
</model:dataInputAssociation>
<model:dataOutputAssociation>
  <model:sourceRef>Calculate_invoice_amount_output
  </model:sourceRef>
  <model:targetRef>invoiceAmount</model:targetRef>
</model:dataOutputAssociation>
  <model:script>${courseCost}
* Integer.valueOf(providedscripts.BonitaXML.evaluateXPathOnVariable(xmlRequest,
"/Request/@numStudents"))</model:script>
</model:scriptTask>

```

Figure 23-11. Serialization of the script task.

Service Task – Email Connector

With the calculated invoice amount, the invoice is sent in an email. In a real process, a connector would request a billing system to generate and send the invoice, but in this simple example we illustrate the use of an email adapter. In non-executable BPMN, we model communication to the customer as a *message*, but executable BPMN often restricts a BPMN message to mean a system-to-system message. That is the case here. We don't use a *sendTask*, but instead a *serviceTask* implemented by a Bonita Email Connector. In the BPMN XML there is no *message* element for the email, but there is one for the connector input, as required by the BPMN metamodel.

Figure 23-12 shows the schema for the Email Connector from the Connectors.xsd file. Figure 23-13 and Figure 23-14 illustrate the configuration of the connector in BOS. Figure 23-15 shows the XML serialization of the *serviceTask* using this connector. Note there is no output for this connector, and consequently no *dataOutput* for the *serviceTask*. (The BPMN 2.0 XSD still demands an *outputSet* element,

empty in this case.

A connector like this with many input parameters generates many lines of BPMN 2.0 XML. Fortunately, the process designer doesn't need to worry about that, as it is all generated automatically by the tool on BPMN export.

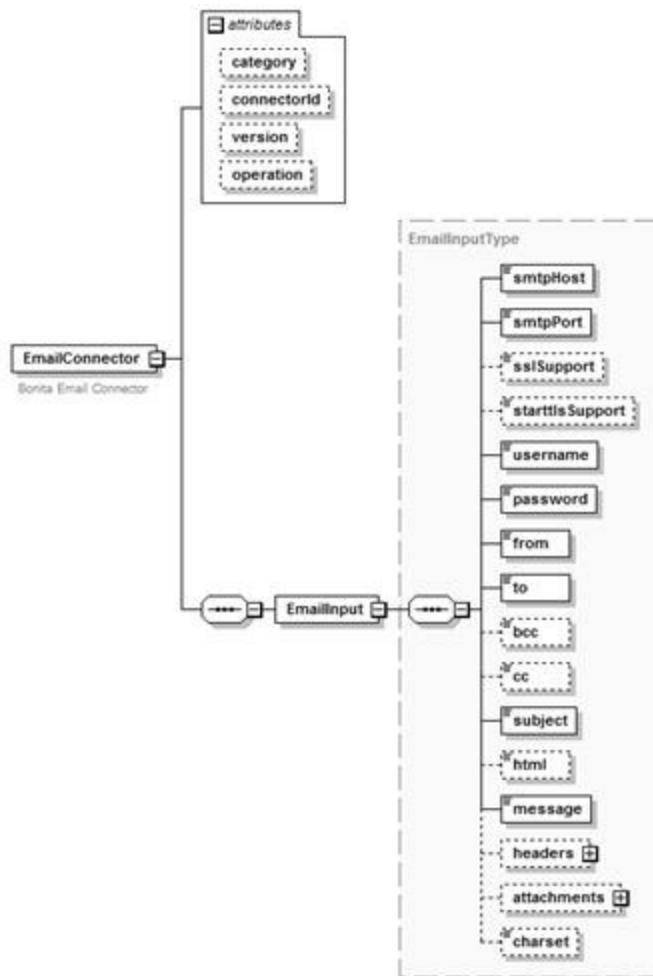


Figure 23-12. Bonita Connectors.xsd schema for Email connector.



Figure 23-13. Email Connector configuration wizard, screen 1. Source: BonitaSoft

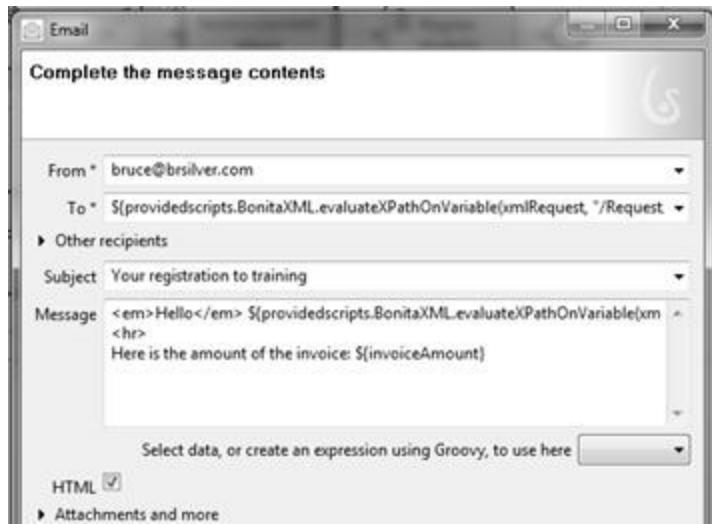


Figure 23-14. Email Connector configuration wizard, screen 2. Source: BonitaSoft

```

<model:serviceTask id="Send_invoice" name="Send invoice" implementation="BonitaConnector"
    operationRef="execEmail">
    <!--this service uses email connector-->
    <model:ioSpecification>
        <model:dataInput id="Send_invoice_input" itemSubjectRef="item07"/>
        <model:inputSet>
            <model:dataInputRefs>Send_invoice_input</model:dataInputRefs>
        </model:inputSet>
        <model:outputSet/>
    </model:ioSpecification>
    <model:dataInputAssociation>
        <model:sourceRef>xmlRequest</model:sourceRef>
        <model:sourceRef>invoiceAmount</model:sourceRef>
        <model:targetRef>Send_invoice_input</model:targetRef>
        <model:assignment>
            <model:from>"smtp.free.fr"</model:from>
            <model:to>getDataInput('Send_invoice_input')/n1:smtpHost</model:to>
        </model:assignment>
        <model:assignment>
            <model:from>"25"</model:from>
            <model:to>getDataInput('Send_invoice_input')/n1:smtpPort</model:to>
        </model:assignment>
        <model:assignment>
            <model:from>"bruce@brsilver.com"</model:from>
            <model:to>getDataInput('Send_invoice_input')/n1:username</model:to>
        </model:assignment>
        <model:assignment>
            <model:from>"password"</model:from>
            <model:to>getDataInput('Send_invoice_input')/n1:password</model:to>
        </model:assignment>
        <model:assignment>
            <model:from>"bruce@brsilver.com"</model:from>
            <model:to>getDataInput('Send_invoice_input')/n1:from</model:to>
        </model:assignment>
        <model:assignment>
    </model:assignment>

```

```

<model:from>${providedscripts.BonitaXML.evaluateXPathOnVariable(xmlRequest,
"/Request/@email")}</model:from>
<model:to>getDataInput('Send_invoice_input')/n1:to</model:to>
</model:assignment>
<model:assignment>
<model:from>"Your registration for training"</model:from>
<model:to>getDataInput('Send_invoice_input')/n1:subject</model:to>
</model:assignment>
<model:assignment>
<model:from><![CDATA[
<em>Hello</em> ${providedscripts.BonitaXML.evaluateXPathOnVariable(xmlRequest,
"/Request/@requesterName")}<br>
Here is the amount of the invoice: ${invoiceAmount}
]]></model:from>
<model:to>getDataInput('Send_invoice_input')/n1:message</model:to>
</model:assignment>
</model:dataInputAssociation>
</model:serviceTask>

```

Figure 23-15. Serialization of serviceTask with Email Connector

Timer Boundary Event

The last part of this example concerns the timeout on waiting for payment notification, modeled as a message from an external payment service provider. Normally we would model this in BPMN as an event gateway or a Timer boundary event on a receive task, but the current version of BOS does not support those. It does support Timer boundary events on a call activity, so our example models it that way. Figure 23-16 shows the dialog for setting the timeout value.

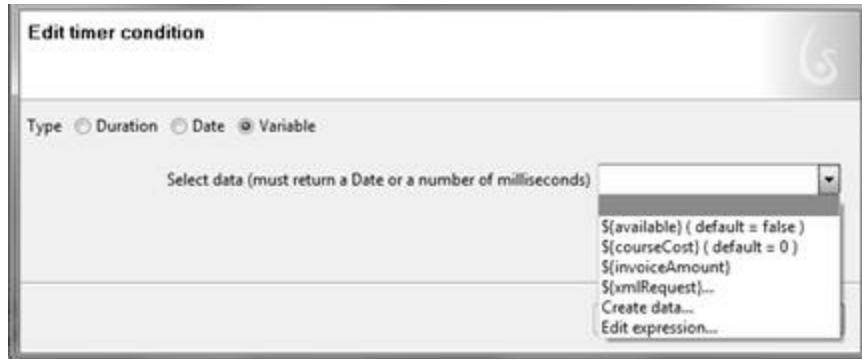


Figure 23-16. Timer event configuration wizard in BOS. Source: BonitaSoft

In this case, the timeout is a date (or dateTime) value calculated from the *courseDate* in the *xmlRequest* variable. The Groovy script expression calculating this is shown in the BPMN 2.0 XML below:

```

<model:boundaryEvent id="_1_day_before_class" name="1" day before class">
  attachedToRef="Receive_payment_advice">
<model:timerEventDefinition>
  <model:timeDate
    xsi:type="model:tFormalExpression" evaluatesToTypeRef="xsd:timeDate">
*****ebook converter DEMO Watermarks*****

```

```
<![CDATA[
import java.text.SimpleDateFormat;
import org.ow2.bonita.util.DateUtil;
String stringDateStart = ((String)providedscripts.BonitaXML.evaluateXPathOnVariable(xmlRequest,
"/Request/@courseDate"));
Date date = DateUtil.parseDate(stringDateStart);
Calendar calendar = Calendar.getInstance();
calendar.setTime(date);
calendar.add(Calendar.DATE, -1);
SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd'T'HH:mm:ssz");
String stringDate = sdf.format(calendar.getTime());
String result = stringDate.substring(0, 19) + stringDate.substring(22, stringDate.length());
return result;
]]>
</model:timeDate>
</model:timerEventDefinition>
</model:boundaryEvent>
```

Figure 23-17. Serialization of Timer boundary event with calculated timeout value

Chapter 24

Aligning Executable Design with BPMN Method and Style

When I began writing this book, I was hoping to describe a methodology that starts from a non-executable Level 2 model created using Method and Style principles and leads to a fully executable BPMN 2.0 model, and illustrate that methodology using real tools. Ideally, the Level 2 model, conforming to the BPMN-I profile, could be *exported* from a tool like Process Modeler for Visio from itp commerce and *imported* into a BPMS like Bonita Open Solution, where the execution-related details would be added.

Unfortunately, the tools are not quite ready to do that yet. I think we are less than a year away. But we can still talk about what it would mean to align executable process design with BPMN Method and Style, and what such a methodology would include.

Recall that BPMN Method and Style is about exposing the process logic clearly in the diagram using nothing but shapes and labels, while executable BPMN is all about defining and mapping process data. Aligning executable design with Method and Style implies a specific connection between the shapes and labels in the diagram and the variables, messages, data inputs, data outputs, and mappings in the executable model. A “methodology” would include cookbook procedures for creating those elements based on specific shapes and labels in the non-executable diagram. But actually, I want the executable design tool do that *automatically* on import. So consider this chapter my “guidance” – more accurately, “wish list” – for architects of executable process design tools.

End State Variables

The notion of *end states* in a process or subprocess is central to Method and Style. The end states of a subprocess are often connected to branching conditions in a gateway following the subprocess. Style rule validation ensures that the end states are properly labeled in the diagram and that the gateway labeling is consistent with it. In an executable model, those same conditions are implemented as *expressions* of process variables (*dataObjects*), using XPATH, UEL, Groovy, or some other expression language.

The executable model thus requires an *end state variable* for each subprocess with multiple end states, with enumerated string values matching the labels of the end events. In the XML, that means creating a *dataObject* in the parent process level, the one that includes the *subProcess* and *exclusiveGateway* elements. Each end event defines a *dataOutput* containing the string value of its label, and a *dataOutputAssociation* mapping that value to the end state variable.

What about a *task* followed by a gateway (or conditional sequence flow)? You could do something similar here as well. Some tools, like Oracle BPM11g, already require enumerated end states of a *userTask*, selected by the task performer through the task user interface, and then tested by a gateway. I would like any task followed by an exclusive or inclusive gateway to define an end state variable (*dataObject*) with enumerated values consistent with the gate labels on the gateway. For a *userTask* or *scriptTask*, where the process designer defines the task implementation, the task *dataOutput* would typically point to the same *itemDefinition* as the *dataObject*, making the *dataOutputAssociation* mapping simple. For a *serviceTask*, with a predefined *interface*, the process designer would need to define a mapping of the *dataOutput* to the end state variable in the *dataOutputAssociation*.

Gateway Conditions

With end state variables, the conditions on most gateway outputs can be generated automatically in the executable design. For an exclusive gateway labeled as a question with gates *yes* and *no*, the *conditionExpression* for the *yes* path takes the form

```
<conditionExpression>
  getDataObject('[endStateVarId]') = "[gatewayLabel without '?']"
</conditionExpression>
```

Messages

Message flows also play an important role in Method and Style. Even if the executable design tool does not display message flows, the *message* elements they represent are important in the model. Most of the message flows in a Level 2 diagram connect a process activity or Message event to a black-box pool. Modelers may attach a Message shape to the message flow, but in the book I recommend labeling the message flow directly. Here we'll assume the message flow has a label but not attached Message shape. Style rule validation ensures that all process nodes that send or receive messages have attached message flows and that all message flows are properly labeled.

In the executable BPMN, we need a *message* element for each message flow, unless it has the same *name* (label) as another message flow in the model. The *name* of the *message* should match the label of the message flow. We will assume that two message flows with the same label represent the same *message*. For example, Method and Style says that a message flow attached to a collapsed subprocess in the parent-level diagram should be replicated in the child-level diagram with matching label. In this case, both *messageFlow* elements in the XML will have *messageRef* pointing to the same *message*. Also, the *messageRef* of an event or task connected to the message flow should point to the same *message*. The executable design tool should generate each required *message* element and all the *messageRef* pointers to it automatically upon import.

Errors

In Method and Style we allow Error events to stand for any type of internally generated exception, whether that is a business exception or a technical exception. Some BPMSs may reserve Error events for technical exceptions and require gateways to handle business exceptions. Here we assume that the non-executable modeling tool and executable design tool follow the same convention.

In the executable BPMN, an *errorEventDefinition* points to a reusable *error* element containing an *errorCode* string. If the infrastructure provides more details about the error, a *structureRef* is available to define the error information structure. Aligning Method and Style with executable design means that on import, the executable design tool should automatically create an *error* element with *errorCode* value matching the Error event label in the diagram, and point to it from *errorEventDefinition*. (If the BPMS supports a pre-defined list of possible *errorCode* values, the user could be prompted to select the best one for each Error event.)

Signal and Escalation events, if supported, could be handled in a similar way.

Obviously, no BPM Suites work this way today, but I believe that a future BPMS that generated these elements automatically (not just in the XML but in the native object model as well) would be welcomed by many BPMN modelers looking to convert their Method and Style Level 2 models quickly and easily to executable processes. There is still plenty of work to do in the executable design environment – designing the task user interfaces, service implementations and parameter mappings, and performance monitoring – but if the BPMS can save time by automatically generating elements that the BPMN diagram implicitly requires, it should do so.

*****ebook converter DEMO Watermarks*****

About the Author

Bruce Silver is principal at Bruce Silver Associates, provider of consulting and training services in the area of Business Process Management. He is founder and principal at BPMessentials, the leading provider of BPMN training and certification. His unique contributions to BPMN include the Method and Style approach and the BPMN-I Profile for model interchange. His website *BPMS Watch* (www.brsilver.com) is well known for reports and commentary about the latest developments in BPM standards, tools, and products. He was a member of the technical team that developed the BPMN 2.0 specification in OMG, and contributed to the BPMN section of OMG's OCEB BPM certification exam.

Prior to founding Bruce Silver Associates in 1994, he was Vice President in charge of workflow and document management at the analyst firm BIS Strategic Decisions, which became Giga (now part of Forrester Research). He has Bachelor and PhD degrees in Physics from Princeton and MIT, and four US Patents in electronic imaging.

To contact the author, email bruce@brsilver.com.

[1]

Our Level 1 and Level 2 were formally included in the final BPMN 2.0 specification, where they are called the Descriptive and Analytic Process Modeling Conformance subclasses, respectively.

[2]

Currently Process Modeler for Visio (www.itp-commerce.com) implements it in the BPMN editor, and I have created an online tool (www.brsilver.com) that validates serialized BPMN 2.0 models.

[3]

For more details, see www.bpmnstyle.com

[4]

For more information, see www.brsilver.com.

[5]

www.itp-commerce.com

[6]

For a firsthand account, see Why All This Matters, Ismael Ghalimi, <http://itredux.com/2008/10/24/why-all-this-matters/>

[7]

In December 2008, somewhat by accident, I joined the IBM-Oracle-SAP submission team, and remained active until publication of the beta specification in the summer of 2009.

[8] See, for example, Paul Harmon, Business Process Change, 2nd edition, Morgan Kauffman, 2007.

[9] <http://www.apqc.org/process-classification-framework>

[10] http://www.apqc.org/knowledge-base/download/31928/a%3A1%3A%7Bi%3A1%3Bs%3A1%3A%222%22%3B%7D/PCF_Cross%20Industry%20destination=node/31928

[11] Paul Harmon, BPTrends Advisor, December 8, 2008, <http://www.bptrends.com/publicationfiles/advisor20081209.pdf>

[12] This is true for a regular subprocess, but an *event subprocess* is an exception handler triggered by an event. Event subprocesses are not included in the Level 1 or Level 2 palettes. They are discussed in Chapter 7.

[13] Currently available only via BPMessentials. See www.bpmessentials.com for more information.

[14] See www.bpmnstyle.com.

[15] Here we use an AND-join before the confirmation Message end event because we want to send the request only once. Connecting *Book Air* and *Book Hotel* directly to the end event would send it twice.

[16] The spec document can be found at <http://www.omg.org/spec/BPMN/2.0/PDF>.

[17] The XSD and XMI may be downloaded from <http://www.omg.org/spec/BPMN/20100501>.

[18] A good reference is Priscilla Walmsley, *Definitive XML Schema*, Prentice Hall PTR, 2002

[19] <http://www.altova.com/xmlspy.html>

[20] <http://www.omg.org/spec/BPMN/2.0/PDF>, page 1.

[21] <http://www.omg.org/cgi-bin/doc?dtc/10-06-02.pdf>

[22] <http://www.omg.org/cgi-bin/doc?dtc/10-06-02.pdf>

[23] Contact bruce@brsilver.com.

[24] For more information, go to www.bpmnstyle.com.

[25] <http://www.omg.org/cgi-bin/doc?dtc/10-06-02.pdf>
*****ebook converter DEMO Watermarks*****

[26] www.bonitasoft.com

[27] Falko Menge, private communication, September 29, 2011

[28] <http://www.activiti.org/>

[29] <http://docs.sun.com/app/docs/doc/820-7627/gjddd?l=en&a=view>

[30] <http://www.activiti.org/userguide/index.html#conditionalSequenceFlowXml>

[31] <http://groovy.codehaus.org/>

[32] http://jasperforge.org/uploads/publish/ireportwebsite/IR%20Website/iReport_groovy.html

[33] <http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html>

[34] BOS exports use the prefix *model*: to signify the BPMN 2.0 namespace.

[35] <http://www.eclipse.org/modeling/emf/?project=emf>

*****ebook converter DEMO Watermarks*****