Mastering jBPM6

This book provides a complete understanding of the jBPM technology stack. It starts with an introduction to the world of business process management systems, the problem domain addressed by jBPM, explores the main use cases that can be addressed by business process management systems, and illustrates the main design patterns. It takes you through the details of the architecture and available out-of-the-box provisions for customizing, extending, and integrating the features of jBPM to meet the requirements of your application. Moreover, the book will empower you with the knowledge to integrate jBPM with enterprise architecture, debug through the source code of jBPM, and utilize the flexibility provided by a highly modular system. Finally, it introduces you to the provisions available for a jBPM-based application to put the non-functional characteristics of the system, which are of great importance when we deploy our application in production. The book helps you in putting the knowledge at work by providing you with a lot of ready to use examples, both basic and advanced ones.

Who this book is written for

If you are a designer or developer who wants to build and operate business process-centric applications, then this book is for you. Knowledge of the basic concepts of application development in Java will be helpful in following the concepts covered in the book, but is not necessary.

What you will learn from this book

- Understand the jBPM tool stack and business process management systems, the standards, patterns, and use cases
- Build your first BPM application using the jBPM tool stack
- Monitor and manage the daily operations of business process-centric applications
- Get a detailed explanation of all the BPMN constructs supported by jBPM
- Understand the core architecture of jBPM
- Learn about the extension points and pluggable mechanisms which would help you to extend jBPM core
- Enhance your application with business process management functionalities by integrating jBPM into your existing application portfolio
- Configure the jBPM-based application to meet the non-functional requirements in production
- Integrate jBPM to the JEE, SOA, and EDA environments to use its capabilities

In this package, you will find:

- The author's biography
- A preview chapter from the book, Chapter 5 'BPMN Constructs'
- A synopsis of the book’s content
- More information on Mastering jBPM6
About the Authors

Simone Fiorini is a seasoned professional with 20 years of software development experience, specializing in large-scale software systems, mission critical applications, and project management in a wide variety of computing environments and platforms, with a focus on EAI, BPM, and integration-oriented tools. His latest efforts are focused on an online reservation system for a large Middle East railway company and a scalable, reactive, financial market data server for a leading Italian banking group’s investment bank.

A graduate of both Università di Parma (earth science) and University of Milan (engineering of computing systems), Simone resides near Brescia, where he’s trying to grow roses and other fragrant flowers for his wife, Giuliana, and their two sons.

Arun V Gopalakrishnan has more than 9 years of experience in creating architecture, designing, and developing enterprise applications used in the domains of BFSI, supply chain management, and telecom businesses.

He graduated as a bachelor of technology in electronics and communications and holds a master's degree in software systems. Currently, he is playing the role of a software architect in designing and developing an enterprise middleware platform.

He is well versed in service-oriented, event-driven architectures and has experience in integrating jBPM with enterprise architecture. He is passionate about learning new technologies and improving his knowledge base and expertise in JEE, OOAD, SOA, EDA, expert systems, and distributed computing.

Along with this technical expertise, he enjoys engineering software applications and is an avid follower, practitioner, and mentor of agile and continuous delivery models. He believes in tools and is well versed in creating, using, and extending DevOps tools, such as Eclipse, Maven, Git, Gerrit, Jenkins, Sonar, JIRA, and MediaWiki, to enhance the efficiency of development teams.
jBPM is a leading open source BPM and workflow platform whose development is sponsored by Red Hat under Apache Software License (ASL) licensing. The jBPM product has been around for almost 10 years; its strongest points rely on flexibility, extensibility, and lightness, and it is a modular, cross-platform pure Java engine that is BPMN2 compliant.

It features a robust management console and development tools that support the user during the business process life cycle: development, deployment, and versioning. It integrates with widely-adopted frameworks and technologies (SOAP, REST, Spring, Java EE CDI, and OSGi) and provides off-the-shelf support to Git and Maven.

It fits into different system architectures and can be deployed as a full-fledged web application or as a service; it can be tightly embedded into a classical desktop application or loosely integrated into a complex event-driven architecture. In its default configuration, jBPM can be hosted by the enterprise class application server Red Hat EAP 6.x or the bleeding-edge Red Hat WildFly 8 server.

*Masteryng JBPMM6* takes you through a practical approach to using and extending jBPM 6.2. This book provides a detailed jBPM 6.2 overview; it covers the BPM notation supported by the engine and explains the advanced engine and API topics focusing, as much as possible, on several working practical examples.

The book presents the user with solutions to common real-time issues like BAM (which stands for business activity monitoring) and production scenarios.
What this book covers

Chapter 1, Business Process Modeling – Bridging Business and Technology, gives the user an overview of the BPM environment, introduces the jBPM world and give insight to the big picture of business logic integrated platform.

Chapter 2, Building Your First BPM Application, starts by taking the user straight to the jBPM tool stack by providing the reader with a hands-on product installation and configuration tutorial, and then, it tackles beginner topics such as business process modeling and deployment.

Chapter 3, Working with the Process Designer, digs deep into web-based jBPM tools to illustrate to the user the main jBPM web designer features: user forms, scripting, and process simulation.

Chapter 4, Operation Management, describes the new jBPM artifacts architecture, focusing on Maven repositories (modules and deployment), engine auditing and logging analysis, jobs scheduling, and a full working BAM customization example (with Dashboard integration).

Chapter 5, BPMN Constructs, illustrates the BPMN2 constructs implemented by jBPM and provides insights and caveats about their usage by commenting a contextually ready-to-use source code example.

Chapter 6, Core Architecture, covers all the jBPM modules (for example, human task service, persistence, auditing, and configuration) by elaborating on how to leverage engine functionalities with the help of several source code examples.

Chapter 7, Customizing and Extending jBPM, explores engine customization areas with a practical approach; it provides the user with explanations on how to customize persistence, human task service, marshalling mechanism and the work item handler architecture.

Chapter 8, Integrating jBPM with Enterprise Architecture, describes how jBPM can integrate with external applications through SOAP, REST, or JMS either as a client or a server. It offers insights on how to leverage its services in a Java EE application.

Chapter 9, jBPM in Production, explores the jBPM system features when dealing with service availability, scalability, and security; it provides tips and techniques related to engine performance tuning in production environments.


Appendix B, jBPM BPMN Constructs Reference, is a quick reference for the BPMN constructs supported by jBPM.
To classify the level of support that a BPMN software tool provides, the BPMN standard defines the "conformance classes" as follows:

- **Process Modeling Conformance**: This class includes the BPMN core elements, process, collaboration, and conversation diagrams. It defines subclasses that contain a limited set of visual elements (descriptive), an extended set of modeling elements (analytical), and modeling elements that are required to model executable processes (common executable).

- **Process Execution Conformance**: It requires a software tool to support the operational semantics of BPMN.

- **Choreography Modeling Conformance**: The choreography modeling conformance class includes the BPMN core elements and the collaboration and choreography diagrams.

jBPM supports a great part of the Common Executable class, with additional extensions. Please check Chapter 6, Core Architecture, of the jBPM 6.2 user guide for insights into the topic.

jBPM introduced the implementation of the BPMN 2.0 specification with the jBPM 5 release, for both the graphical notation (element visual representation) and the XML serialization, easing the task of exchanging process definitions between developers and the business team (in terms of Eclipse-based BPMN editor and process Web-based designer interoperability).
Other jBPM BPMN notable features are as follows:

- Compliance with the BPMN process execution semantics ("Common Executable" subclass specification)
- The BPMN DI (which stands for Diagram Interchangeability) specification for storing diagram information
- The BPMN I/O specification for input/output mapping

In Chapter 1, Business Process Modeling – Bridging Business and Technology, we already had an overview of the main BPMN concepts, constructs, and modeling patterns. We selected the topics for this chapter not to provide you with a BPMN modeling or reference guide, but as hands-on, example-driven explanation of all BPMN constructs supported by jBPM, without completely hiding away the underlying technical details.

In this chapter, we will discuss the following:

- The concept behind the BPMN construct
- How to use it in a business process (with examples)
- Best practices for when and where to use BPMN constructs

**Parameters, variables, and data**

Most of the time, business processes are data-driven processes: tasks handle variables, and rules handle facts; you will not be asked to draw a BPMN diagram without handling variables, parameters, objects, and states coming from external systems, user input, and other sources. A majority of the jBPM constructs are useless without data. Let us clarify the basics:

- **Parameters**: These are the data input coming from the user through the API. The user can pass parameters during process creation, at a human task completion, or into a service task for a Web service call.
- **Variables**: Variables are objects living in the scope of a single process instance. Variables can be created directly inside a process instance construct (for example, Script Activity and Data Object) or can be mapped from/to other variables (Data Input/Output Mapping) defined in another scope, for example, from the main process to a subprocess, from the process to a human task, and so on.
- **Globals**: Static variables shared across different process instances for a single Kie working session.
• **Facts**: Data that can be added to the Kie session and then updated or removed (retracted). This information is inserted, technically speaking, into the session through channels named **entry points**, and evaluated according to the Drools business rules, for activation. Drools Agenda manages the rule activation and firing mechanism.

Please refer to Drools reference documentation for additional details on facts, rules, entry points, Agenda, and the Drools rule engine in general: https://docs.jboss.org/drools/release/6.2.0.Final/drools-docs/html. Drools and jBPM are complementary projects that integrate together very nicely.

Variables and globals are accessed through context-type implicit references made available to the jBPM constructs at runtime:

- **ProcessContext (kcontext)**: This gives you access to variables
- **KieRuntime (kcontext.getKieRuntime())**: This gives you access to globals and facts

There are no implementation constraints on parameters, variables, and global class types apart from implementing the java.io.Serialization interface. Remember in fact that jBPM uses the standard in-memory serialization mechanism (readObject/writeObject). When we enable persistence, it features an additional custom object marshalling mechanism to and from the store for session and process instances (see **Marshalling in Chapter 7, Customizing and Extending jBPM**). Furthermore, when there are persisting process variables for auditing and logging (**VARIABLEINSTANCELOG** table), jBPM stores the values by calling the process variable **toString()** method.

jBPM does not provide out-of-the-box process variable persistence in any of its schema tables. We need to implement our ad-hoc variable serialization strategy (we will cover variables persistence with **Marshalling in Chapter 7, Customizing and Extending jBPM**).
**Sequence flow**

The sequence flow is the connector between two elements of the process. It represents a flow of execution. A sequence flow may optionally have a condition defined (conditional sequence flow). The engine always evaluates a task node's outgoing sequence flows: If the condition evaluates to true then the engine selects and follows that sequence flow; a sequence flow with no condition defined is always followed by the engine. A **diamond shaped** connector (see Appendix B, jBPM BPMN Constructs Reference, Gateways section for some pictorial examples) indicates a conditional sequence flow. Multiple sequence flows represent branching and merging without the usage of a gateway. Gateways, depending on their nature, handle conditional sequence flows in specific ways as we are about to see.

jBPM allows you to enable multiple outgoing conditional sequence flows from a task by setting the `jbpm.enable.multi.con` system property to `true` (default is `false`).

The following example process (see the figure) shows how the `jbpm.enable.multi.con` property affects the sequence flow behavior.

![Sequence flow diagram](image)

Example test class:

`com.packt.masterjbpm6.SequenceTest`

Example process:

`sequenceflows.bpmn`
Description: The test creates the process instance with an Order variable with different cost values. The process, thanks to the jbpm.enable.multi.con system property set to TRUE, allows the execution of multiple (here, we have two) conditional sequence flows that diverge from a single Script Activity. The first sequence flow is taken if the Order costs more than 10, while the second one is taken when the Order cost is ≤10.

**Gateways**

Gateways are elements that allow you to create branches in your process. These branches can be, conceptually, diverging or converging. You can model the behavior of the different types of business process sequence flows: conditional branching (inclusive and exclusive), forking, merging, and joining.

Let us first review the key gateway concepts and the practical examples in the upcoming sections:

- **Fork** (split) indicates a flow dividing into two or more paths that should execute in a logically parallel (concurrent) way: jBPM, for implementation reasons, never executes parallel flows concurrently (at the thread level) but always sequentially, one step at a time
- **Join** (or synchronization) refers to the combining of two or more parallel paths into one path
- **Branch** (or decision) is a point where the control flow can take one or more alternative paths
- **Merge** refers to a process point where two or more alternative sequence flow paths are combined into a single sequence flow path

Hence, the gateway direction property is defined as follows:

- **Unspecified**: May have both multiple incoming and outgoing connections
- **Mixed**: Multiple incoming and outgoing connections
- **Converging**: Multiple incoming connections and only one outgoing connection
- **Diverging**: Only one incoming connection and multiple outgoing sequence flows

Unspecified and mixed directions are not implemented

Let us now see how these BPM concepts translate into jBPM modeling elements.
Parallel (AND) gateway
This gateway allows us to fork into multiple paths of execution or to join multiple incoming paths of execution. When used to fork a sequence flow (diverging or AND-split), all outgoing branches are activated simultaneously. When joining parallel branches (converging or AND-join), it waits for all incoming branches to complete before moving to the outgoing sequence flow. This gateway must be used when many activities have to be carried out at the same time in any particular order.

Example test class:

    com.packt.masterjbpm6.gateway.GatewayParallelTest

Example process:

    gateway_parallel.bpmn

Description: The plan route script task calculates the order delivery route, while the Prepare Ingredients human task adds some mozzarella to the order bill of materials. The closing Done Script task displays the result after all outgoing flows are complete.

Conditional branching
These gateways introduce the condition expression. The condition expressions linked to each of the outgoing/incoming sequence flows are evaluated during process execution using process data (data-based gateways). Optionally, one of the gateway outgoing paths can be flagged as the default flow (its condition is ignored): this path is taken only if none of the other path flows can be selected. The default (sequence) flow is visually marked with a slash mark as shown in the following image:
The "default flow" property is supported in the Exclusive and Inclusive Gateway elements.

**Drools**

We briefly introduced Drools facts and rules in the first section. Conditional branching based on Drools expressions works with facts but not with process variables. If we want to leverage the Drools expression features in the gateway constructs, we have to insert the process variable as a Drools fact, for example, given the process variable `order`:

```java
Order order = new Order();
order.setNote("urgent");
order.setCost(110);
```

From inside the process definition (by a Script task, a Task on exit Script, and so on), we insert the following fact:

```java
kcontext.getKnowledgeRuntime().insert(order);
```

Alternatively, we can do so by using the API as follows:

```java
ksession.insert(order);
ksession.fireAllRules();
```

**Exclusive (XOR) gateway**

It is used to model a decision in the process. More than one path cannot be taken; the paths are mutually exclusive, hence, the name. In case multiple sequence flows have a condition that evaluates to true, the first one defined in the XML is selected for continuing the process. In an exclusive gateway, all outgoing sequence flows should have conditions defined on them. The default sequence flow is an exception to this rule.
**BPMN Constructs**

Example test class:

```java
com.packt.masterjbpm6.gateway.GatewayExclusiveTest
```

Example process:

```java
gateway_exclusive.bpmn
```

Description: Different paths are taken for successful Pizza deliveries; the default path is chosen when other conditions are not met.

![BPMN Diagram](image)

**Inclusive (OR) gateway**

An inclusive gateway is a branching point of the business process. Unlike the exclusive gateway, an inclusive gateway may trigger more than one outgoing flow and execute them in parallel (such as the parallel gateway). So, with diverging behavior, the gateway will always evaluate all outgoing sequence flow conditions, regardless of whether it already has a satisfied outgoing flow or not (unlike the exclusive gateway). In the case of converging behavior, the gateway will wait until all the incoming active sequence flows have reached it (merging). We can usually use this construct in a pair of splitting/merging gateways (see the following example) when we need to fork executions depending on certain conditions and then rejoin them.

Example test class:

```java
com.packt.masterjbpm6.gateway.GatewayInclusiveTest
```

Example process:

```java
gateway_inclusive.bpmn
```
Description: Multiple different paths are taken for evaluation of the order delivery status; the testIssues test is set up so as to make the process take both the delivery not on time (deliveryDate > dueDate) and the retries > 1 path. The default path is chosen when other conditions are not met (see the testNoIssues test).

**Event-based gateways**

Event-based gateways are similar to exclusive gateways, but the gateway trigger is based on event occurrence instead of condition evaluation. When our process arrives at an event-based gateway, we will have to wait until something happens. A specific event, usually the receipt of a message, determines the path that will be taken. Basically, the decision is made by another actor on the basis of data that is not visible to a process. This gateway is always a diverging gateway and must have at least one event attached.

Example test class:

```java
com.packt.masterjbpm6.gateway.GatewayEventAndTaskTest
```
Example process:

```
gateway_event_and_task.bpmn
```

**Description:** The event gateway has a timer attached; when the timer expires, the `send alert` script is executed, bringing the process to termination.

**Instantiating gateway**

The instantiating gateway is a specialized event-based gateway, which triggers the process instantiation as soon as an attached event is received. The "instantiate" option (as of jBPM 6.2 the option is available in the jBPM Eclipse plug-in only) configures the gateway as a diverging gateway with no incoming connections: this gives you a way to instantiate a process by using an event, such as timer expiration or a catching signal event (see the following sections for timers and signals). jBPM does not support a pure instantiating gateway with no incoming connection: you always have to link it to a Start "None" event (see the following figure) or the process compilation will fail (complaining with a "missing incoming connection" error)

Example test class:

```
com.packt.masterjbpm6.gateway.GatewayEventTest
```

Example process:

```
gateway_event.bpmn
```
Description: Depending on events sent from an external (API call), different paths are taken (the `testCustomerPhoneCallEvent` and `testDeliveredEvent` methods); the timer triggers after 15 s if no event is caught (the `testTimerExpired` method). Note that both catching events pass the signal data (a randomly generated `orderid` string) to the process parameter `orderid`, which is later printed from the script tasks.

Complex gateway
This gateway can be used to model complex synchronization behavior. The construct options are available at the designer level, but jBPM has no implementation for this construct.

Events
Events are elements used to model something that happens during the process lifetime. BPMN 2.0 defines two main event categories: catching and throwing events.

- **Catching**: This event represents a pausing point in the process execution: Once the process flow reaches the catching event node, it stops in the wait state, waiting for a specific trigger to happen.
- **Throwing**: This event represents an action generating an event. When process execution reaches the event construct, an action is performed and a trigger is fired. For this throwing event, depending on the event type, there could be a matching catching event or not, that is, a send signal (throwing)/catch signal or send error (throwing)/catch error. On the other hand, the compensate throw event does not have a catch companion, while the timer event is always a catching event.
Events are also categorized according to other criteria:

- An event can appear at the beginning of a process (Start event), within a process (Intermediate event), or at the end of a process (End event)
- An event can be generic or one of the different predefined types: time-based, message-based, signal-based, rule-based, exception-based, and so on
- An event can be positioned within a sequence flow or attached at the boundary of an activity (Boundary event)
- An event can exit the current process execution or not

A note before we start:
To facilitate reading, we’ll go through the events by grouping them by event type (Start, Boundary, End) and then illustrating the supported variations (catching/throwing and start/intermediate/boundary/end) for each type of event (Signal, Message, Timer…).
For additional information and a complete jBPM constructs reference (ordered the same way as you will find in both the Eclipse BPMN modeling tool palette and the KIE console palette), please refer to Appendix B, jBPM BPMN Constructs Reference.

**Start events**
The start event defines where (and how) the process is started; Start events are catching-only events. When a specific start event trigger fires (timer, messages, signal, and so on) the process is started. We will now see the None Start event; the other start event types are discussed in their respective sections.

Supported start events are: None, Message, Timer, Escalation, Conditional, Error, Compensation, Signal

**None Start event**
The simplest form of a Start event is the None Start event. It technically means that the trigger for starting the process instance is not specified; in other words, the engine does not know when the process instance is to be started. The only way to start the process is by invoking the startProcess method on a Kie session reference.

```
ProcessInstance.startProcess(String processId, Map<String, Object> parameters);
```
End events

The End events are meant to express the end of a process or subprocess, and they are always throwing events. When the process execution arrives in the End event node, the associated event type is thrown. A process definition can have one or more End events defined. In this section, we will see the None and the Terminate End event; the other End event types are discussed in their respective sections.

Supported end events are: None, Message, Escalation, Error, Cancel, Compensation, Signal, Terminate

(None) End event

The None End event throws no events, and the engine just ends the current process instance sequence flow execution. If there are no more active sequence flows or nothing else to be performed (activities), the process instance is completed.

Terminate End event

The Terminate End event brings the process instance to the Completed state; all pending tasks, active sequence flows, and subprocesses are aborted.

Boundary events

Boundary events are events (always catching) that are graphically attached to an activity (subprocesses included) boundary (see the following figure). The event is registered for a certain type of trigger (see the following supported boundary events) and reacts only within the scope of the execution of the attached activity, with slight variations depending on the event type. In case the event triggers, it can optionally cancel the activity that it is attached to (by its cancelActivity property), and the event's outgoing sequence flow is executed. The boundary events are activated when the attached activity is started; in other words, they are bound to the activity instance life cycle. When the engine process execution path leaves the activity, all its attached boundary events are deactivated and their triggering is cancelled.

Supported boundary events are: Conditional, Error, Escalation, Message, Signal, Timer

See the Boundary Message event section for a working example.
Signal events
A signal is a generic, simple form of communication, such as messages (see below). We can use signals to synchronize and exchange information. A catching signal may not have a corresponding throwing signal construct. It can also be sent programatically from an external source (API). In contrast to other events (error event), if a signal is caught, it is not consumed. If there are two active intermediate catching events firing on the same signal event name, both events are triggered, even if they are part of different process instances and definitions. If the signal is sent and there are no catching signals registered for this event, the event is lost.

Scope
Signals can have visibility between different parts of the same process or broadcast processes (scope across all process instances), or targeted to a specific process instance. You can throw a signal event in a process instance, and other process instances with a different process definition can react to the event. Please keep in mind that this behavior (broader or narrower signal scope) can be affected by the runtime strategy chosen to create your Kie session (the subject is discussed in Chapter 6, Core Architecture).

Signal ID and signal name tips
You may notice some issues with signals when creating/modifying process signals in BPMN processes shared between the KIE jBPM console editor and the Eclipse BPMN modeler. The generated BPMN differs, and this may lead to bugs and unexpected behavior.

When creating the process definition from the Eclipse BPMN editor, the signal is assigned an internal ID of the form: Signal_{number}. Therefore, the actual signal ID to use is the same signal ID that you see in the Signal property editor and not the user-assigned signal name in the process definition panel (signal list table). Keep in mind this additional signal name referencing when coding against the org.kie.api.runtime.KieSession.sendSignal method.

```xml
<bpmn2:signal id="Signal_1" name="customerPhoneCall"/>
<bpmn2:signalEventDefinition id="SignalEventDefinition_1" signalRef="Signal_1"/>
```

Therefore, with an Eclipse-generated process, the Signal_1 ID must be used with the API.

```xml
<bpmn2:signal id="customerPhoneCall" name="customerPhoneCall"/>
<bpmn2:signalEventDefinition id="_05nSUW_YEeSWR_CUOywjQQ" signalRef="customerPhoneCall"/>
```
With a process generated from a jBPM Web console editor, the signal ID is equal to the name attribute; `customerPhoneCall` must be used with the API.

**Signal data mapping**

Signals can carry optional object data; for each triggered catching signal, you can get this signal data and map it to a process variable. When operating with the jBPM Web designer, in order to successfully map the signal data to a process variable, you have to configure the `DataOutput` signal and assign it the name `event` as you can see in the following screenshot. The picture shows the event's data mapping for the `start_signal.bpmn` process signal events (see the `Start Signal event` section example for a detailed event data mapping example).

![DataOutput Screenshot](image)

This is a very flexible mechanism. By delivering data with your signals, you can update process variables, convey extra information, or change the process flow very easily.

**Start Signal event**

With a named Start Signal, we can programmatically start a process instance. The signal can be fired from within an existing process instance by using the intermediary signal throw event or through the API (the `sendSignal` method). In both cases, all process definitions that have a Signal Start event with the same name will be started. You can have multiple Start Signal events in a single process definition.

Example test class:

```java
com.packt.masterjbpm6.event.StartTest (method testSignalStart)
```
Example process:

```
start_signal.bpmn
```

Description: Different Start Signal events are sent so as to create different process instances. The signal data is mapped to the process variable (see the previous section for an explanation of event data mapping).

**Intermediate Signal event**

An Intermediate catching Signal event catches signals sent from a throwing intermediate signal or through the API call (KieSession or ProcessInstance.sendSignal) and continues the process instance flow. The catching signal has no incoming connections.

**Boundary Signal event**

See the Boundary events section.

**End Signal event**

This kind of signal event is sent at the completion of the process. It can be a handy way to track process instance completions across the system.

**Message events**

Message events reference a name and can optionally have a payload. Unlike a signal, a message event is always targeted at a single catching message. The name of the catch and throw messages must be exactly the same in order to make the message flow work properly. Let us point out some differences between messages and signals:
• Inside the BPMN diagram, the message flow is drawn linking the sender to the receiver, while signals are never directly connected on the diagram. The throwing and the catching signal are implicitly connected only by their name.

• Messages should only be thrown/caught in the same process instance; there is no such limitation for signals. Messages work at the process instance scope only and are point-to-point links. A signal can travel from one process instance to many process instances (broadcast scope).

Message data mapping
See the Signal data mapping section.

Start Message event
A Start Message event is used to start a process instance as a direct consequence of catching a message; a process can have multiple Message Start events. This allows us to choose the process creation method simply by changing the Message event name to send (see the following image). Make sure that the message event name is unique across all loaded process definitions to avoid unwanted process creations.

When sending the message from the API (sendSignal), we have to prefix the message name with the Message-string.

Message Start events are supported only with top-level processes and not with embedded subprocesses.
BPMN Constructs

**Intermediate Message event**
If a process is waiting for the message, it will either be paused until the message arrives or change the flow for exception handling. For using a throw message, there has to be a catch message event that catches the message. It can be a message intermediate event or a message start event.

**Boundary Message event**
The following example shows task cancellation and message data passing by using two boundary events (a timer and a message) attached to a human task. The timer has the cancel activity property set to FALSE, while the message has it set to TRUE. The boundary message event maps the event data to a process variable in order to log the cancellation reason passed by the throwing (external) message sent by the test class.

Example test class:

```java
com.packt.masterjbpm6.event.BoundaryTest (method testBoundaryWithCancel)
```

Example process:

```xml
boundary.bpmn
```

Description: A process with a human task is created. The timer event's duty is to cycle and expire every 15 s calling the script task "time out warning" (its timer expression is 15s###15s, and it is not flagged as "cancel activity"; therefore, the task will not be cancelled as the timer triggers). When the user continues with the test (the test class asks the user to press a key to proceed), a message is sent (sendSignal), the process message boundary event is triggered, and the activity is cancelled (since the boundary message event has the "cancel activity" flag enabled). Note that the message is sent by our test class with some data that serves as the task cancellation reason ("cancelled by ADMIN"):

```java
sendSignal("Message-messageCancelService", "cancelled by ADMIN");
```

The boundary message (id=messageCancelService) catches the sent message, and the message event data, which is bound to the process variable reason, is printed in standard output by the cancel log script task.
End Message event
A message is sent to a specific process at the conclusion of a process.

**jBPM throwing message implementation**
The jBPM throwing message default implementation is just a placeholder. You must provide your own WorkItemHandler definition and register it with the name `Send Task` to the jBPM Runtime, providing a hook to the working Kie session (identified by `ksession` in the following code fragment):

```java
SendMessageTaskHandler messagehandler = new SendMessageTaskHandler();
messagehandler.setKnowledgeRuntime(ksession);
ksession.getWorkItemManager().registerWorkItemHandler("Send Task",messagehandler);
```

Throughout this chapter, you will find several references to "workItem" and "workItem handler and manager." These are the jBPM component part of a feature that lets you define a custom Java class and bind it with a specific process activity type in the engine runtime. Every time the engine activates this activity type, your handler will be invoked and passed the control. Please refer to Chapter 7, Customizing and Extending jBPM for detailed explanation and examples.

From the custom workItemHandler, you can then send signals:

```java
public void executeWorkItem(WorkItemListWorkItem, WorkItemManager manager) {
    ksession.signalEvent("Message-startmessage", "processdata");
```
Example test class:

```java
com.packt.masterjbpm6.event.StartTest (method
  testMessageStartFromMessageThrow)
```

Example processes:

```java
start_message_catch.bpmn, start_message_throw.bpmn
```

Description: The process created sends a message by a custom WorkItemHandler
starting a new instance of the `start_message_catch` process (by a start
message event).

## Timer events

Timer events are events that are triggered when a timer construct expression is met;
the timer properties are as follows:

- **Time Duration**: Single trigger delay value (for example: 10 m, 25 s).
- **Timer Cycle**: The time expression that shall be evaluated. It can be a string
  (interval-based 20 s or 5 m##35 s, where the first value is the initial delay
  and the second value is the delay between repeated fires), a string `cron`
  expression, or a process variable. In the case of JBPMM 6.x, it can also be
  an ISO-8601 formatted date.
- **Timer Cycle Language**: Can be a default interval (empty value and time
duration set) or `cron`.

## Start Timer event

A Start Timer event is used to create a process instance at a given time. It can be used
for processes that should start only once and for processes that should start at specific
time intervals. Note the following points:

- A subprocess cannot have a start timer event.
- A Start Timer event is registered as soon as the process is deployed.
  There is no need to call the `startProcessInstance` API.
- When a new version of a process with a Start Timer event is deployed,
  the job corresponding to the old timer will be removed.
Intermediate Timer event
This event is a catching event only. The timer value triggers the execution of the outgoing sequence flow. You can use the timer to insert a generic delay or a timed-out sequence flow execution; for example, you could add a timer to manage a due date for a human task completion (see the Event-based gateway section example for a timer that acts this way).

Boundary Timer event
See the Boundary Message event section for an example.

Error events
Error events are used to model business exceptions. They are triggered by an exception that might be generated during the execution of an activity. Intermediary throw/catch error events do not apply.

Boundary Error event
This boundary error event must be attached to an activity. As the error event triggers, the activity is always canceled and the error event's outgoing sequence flow is taken.

Example test class:

```java
com.packt.masterjbpm6.event.ErrorTest (method testBoundaryErrors)
```

Example process:

```xml
errorboundary.bpmn
```
Description: Two different boundary error events are attached to the same user task registered on different errorCode properties (FileNotFoundException or RuntimeException); the error handler logs the exception message. Depending on the process parameter (triggerexceptionflag) value passed, the user task throws a different exception upon completion (the onExit script), which triggers the appropriate boundary error event.

The process is started with a variable whose value affects the type of exception to be thrown:

```java
Map<String, Object> params = new HashMap<String, Object>();
// "1" for a runtime exception; "2" for a FileNotFoundException
String trigger = "1";
params.put("triggerexceptionflag", trigger);
ProcessInstance processInstance = ksession.startProcess("errorboundary", params);
```

The user task's onExit script evaluates the process variable and throws the exception accordingly:

```java
String trigger=(String)context.getVariable("triggerexceptionflag");
if (trigger.equals("1"))
{
    throw new RuntimeException("a runtime exception");
}
else
{
    throw new FileNotFoundException("a filenotfound exception");
}
```
The engine triggers the appropriate boundary error event depending on the exception thrown; the event, in fact, must be configured with the `errorCode` property set to the exception classname: `java.lang.RuntimeException` (see the following screenshot).

![Screenshot of BPMN diagram](image)

Note that the boundary error can bind the exception to a process variable. In the example, this variable (`exceptionvar`) is logged to the console by the script task:

```java
Throwable exc=(Throwable )context.getVariable("exceptionvar");
System.out.println("log error message:"+exc.getMessage());
```

**Error Start event**

The Error Start event can only be used to trigger an Event Subprocess and cannot be used to start a process instance. This is a feature you could consider using when activating alternative subprocesses on error exceptions.

**Error End event**

When the process execution reaches an Error End event, the current path of execution is ended and an error event is thrown. This error is caught by a matching intermediate boundary Error event or a subprocess Start Error event. If no Error event is found, an exception is thrown.

The following example uses the Error End event to trigger a subprocess by its Error Start event.
Example test class:

```java
com.packt.masterjbpm6.event.ErrorEndTest (method testSubprocessStartError)
```

Example process:

`errorsubprocess.bpmn`

Description: The main process features a human task and an Error End event, which triggers an embedded subprocess Script task by an Error Start event.

---

**Compensation**

Complex business processes may involve a number of heterogeneous parties and systems such as modern transactional systems, legacy systems (not transactional), and Web services. In order to preserve business consistency, when something fails and no transactional protocols are available, these systems may require you to perform programmatic corrective actions by invoking some dedicated API or by any other means. The compensation is the action of post-processing trying to remedy (not properly undoing or rolling-back) the effects produced by an action.

We want to stress the fact that jBPM compensations are not a transactional feature or a `try/catch` error mechanism. The compensation is a BPM business feature, which models an activity as the compensating counterpart for an already completed activity.

Here you have the common steps, which take place during a compensation event (see the following process example figure for a visual reference of the sequence).

- An activity (A1) whose boundary is attached to a compensation event (E1) is completed
- A compensate event (E2) is thrown somewhere in the process
• The compensate event (E1) catches E2
• jBPM activates the compensation handler (A2), which is connected to E1

The engine is ignorant of what the compensating activity will do since it is up to the developer to define the compensating business logic.

**Intermediate Compensation event**
The throwing Compensation event (E2) and the boundary Compensation event (E1) are implicitly connected by the same event name (we have already seen this with signals and messages). What we have explained for boundary events still applies here: when the Compensation event (E2) is triggered, the boundary Compensation event (E1) reacts by invoking the linked compensating activity (A2), which is marked with the typical compensation FastBackward-like symbol.

**Boundary Compensation event**
The Compensation boundary event (E1) must reference one Compensation handler (A2) only through the direct association line. The Compensation boundary event is activated only when the activity (A1) has been completed (unlike the default boundary event behavior where the event is activated depending on the Activity start state). The Compensation catch event (E1) is removed after either the parent process instance completes or the Compensation event itself is triggered. If a Compensation boundary event is attached to a multiple-instance subprocess, a compensation event listener will be created for each instance. jBPM does not seem to support this last feature.

**Compensating activity**
This activity (also called a compensation handler) is directly connected to the triggering boundary compensation event and must have no outgoing sequence flows.

Example test class:

```java
com.packt.masterjbpm6.event.CompensationTest (method
testCompensationEvent)
```

Example process:

```java
compensateorder.bpmn
```
Description: We used this example process to explain the typical compensation "workflow," so you should already be familiar with it. Let us just add that the Compensate event is thrown when the human task (H1) is completed and the cancelOrder variable evaluates to "y." This activates the exclusive gateway sequence flow, which triggers the event (E2). This activates the boundary Compensate event (E1), which in turn calls the cancel order script task (A2). The cancel order task acts as a "compensating" activity.

Triggering compensations with signals

jBPM offers additional ways to trigger compensations inside a process instance by using signals: general (implicit) and specific compensation handling. An implicit compensation triggers all of the compensation handlers for the process instance:

```java
ksession.signalEvent("Compensation",
    CompensationScope.IMPLICIT_COMPENSATION_PREFIX
    + "compensateorder", pi.getId());
```

You must use the compensation signal type and pass the signal data a string that results from concatenating the CompensationScope class constant and the process definition ID resulting in the following:

"implicit:compensateorder"

The specific compensation triggers a specific compensation handler inside a process instance. You must pass the activity node ID attached to the boundary compensation event, along with the process instance ID:

```java
ksession.signalEvent("Compensation", "_2", pi.getId());
```
Our example process script task XML element follows:

```
<bpmn2:scriptTask id="_2" name="prepare order"
scriptFormat="http://www.java.com/java">

	No new signal event needs to be defined at the process definition level.

```

For working examples, please refer to the following:

Example test class:

```
com.packt.masterjbpm6.event.CompensationTest (methods
testGlobalCompensationWithSignal and
testSpecificCompensationWithSignal respectively).
```

### End Compensation event

The end compensation event works the same way as the intermediate one (please see the example process figure). A compensation end event is thrown \((E1)\), and the compensation handler triggered \((A1)\). This kind of event is useful when there is a need to perform housekeeping or remediation business logic at the end of a process, but only when your bounded activity \((S1)\) is in the COMPLETE state. Note in fact, as we already stressed, that the compensation handler kicks in only when the subprocess \((S1)\) is already in the completed state.

Example test class:

```
com.packt.masterjbpm6.event.CompensationTest (method
testSubprocessCompensationEndEvent)
```

Example processes:

```
compensateendsubprocess.bpmn
```
Description: The process has a subprocess \(S1\) with an attached boundary Compensate event \(E2\). The subprocess triggers the throwing compensate end event \(E1\). The Compensate boundary catch event \(E2\) invokes the compensation handler \(A1\), which rolls back the process variable to the initial value.

**Multi-instance compensation**
Compensation catching events attached to a multi-instance subprocess are not implemented. See the Subprocess section for details about multi-instance activities.

**Escalation**
Escalation, according to the common policies of an institution, organization, or corporate, refers to the existing relationships between the working personnel and their duties. The presence of an escalation event indicates that there is a condition that requires the business process flow to be diverted to a different user group. For instance, if an order above a certain price threshold is received, the approval task must be performed by a user in a higher role (for example, a manager); otherwise, it can also be approved by a clerk user.

In the case of jBPM 6.2.0, escalation events seem to be partially implemented and it is not clear what part of the BPMN specification is supported at this stage. You can partially overcome the lack of an escalation event with Deadlines and Notifications (see the User Task section).
Conditional events

Conditional events are a jBPM feature extension. They are triggered by an evaluation of user-provided expressions of Drools rules and facts properties. Conditional Start and Boundary events are supported.

Example test class:

```java
com.packt.masterjbpm6.event.ConditionalTest (method testSubprocessStartError)
```

Example process:

```bpmn
conditional.bpmn
```

Description: The main process is started when the Fact order note property matches "urgent"; the following script task `ordercost` is cancelled if Order cost > 100.

![Conditional BPMN process diagram]

Activities

An activity is a unit of work that is executed within a business process; it can be atomic or non-atomic (Compound activity, Call activity, or Subprocess). Activities can be of the Task, Call activity, or Subprocess type.

Task

A task is the smallest atomic activity unit that can be included within a process. Usually, the performer of the task can be an end user (called human) using a UI-based application, a participating external service, or a generic set of business statements. Tasks have their local scope and can accept input parameters from their container and return output parameters.
User Task
A user task is used to model work that needs to be done by a human actor. When the process execution arrives at the user task node, a new task instance is created in the work list of the actor(s) or group(s) defined for this task (the Actors and Groups properties). Human tasks can transition to several different states and involve human stakeholders depending on the action taken on the task itself and the defined human roles.

Human roles
Human roles define what a person or a group of actors can do with tasks. Let us review the roles defined for the human task activities:

- **Task initiator**: The person who creates the task instance. Depending on how the task has been created, the task initiator may not be defined.
- **Actual owner**: The person who owns the task and is performing it. A task always has one actual owner.
- **Potential owners**: Persons who are given a task so that they can claim and complete it. A potential owner can become the actual owner of a task by claiming it.
- **Excluded owners**: Actors may not transition to be an actual or potential owner, and they may not reserve or start a task.
- **Business administrators**: Business administrators are able to perform the same operations as task users since they are always potential owners of every task. jBPM provides a default business administrator user (Administrator) and group (Administrators).

State transitions
The task remains in the Created state until it is activated. When the task has a single potential owner, it transitions into the Reserved state (it is assigned to a single actual actor); otherwise, it transitions into the Ready state; this state indicates that the task can be claimed by one of its potential owners. After being claimed, the task transitions into the Reserved state, elevating the potential owner to the actual owner actor. At this point, the actor can start the task that is in either the Ready or the Reserved state and make it transition to the InProgress state. The InProgress state means that the task is being worked on. If the actor completes the work, the task transitions into the Completed state. If the completion of the work goes wrong (exception), the task is put into the Failed state. Alternatively, the user can release the task, bringing it back to the Ready state. No transition is allowed from the Complete state and the Failed state.
For detailed information on task state transitions, please refer to the Web Services – Human Task (WS-HumanTask) Specification by Oasis at http://docs.oasis-open.org.

Deadlines and escalations

The jBPM concept of a task deadline is bound to the task start-complete time interval duration; deadlines are associated with task escalations: the task escalation may exist in either a task reassignment or a task notification action. The task deadline is calculated on the task expiry date: it is reset when the task is started, and it expires when the task is completed over the allowed time boundary. Deadlines are physically stored in the DEADLINE table while notifications are stored in the NOTIFICATION set of tables.

The Reassignment and Notifications property editor is available in the KIE Web process editor only.
Task reassignment
Task reassignment is a jBPM mechanism that lets you change a task ownership by setting specific rules, which are based on the task state transition and a deadline time expression, for example: "if Luigi (a named task actor) does not start the task in 60 seconds then reassign the task instance to Mario." The nominated user is replaced by the new user as the potential owner for the task. The resulting reassignment rule syntax is as follows:

[users:mario|groups:@[@60s]@not-started

You can define multiple reassignment rules on a single task instance.

Task event type conditions can be not-started and not-completed.

The BPMN XML task parameters are NotStartedReassign and NotCompletedReassign. The reassignment information is persisted by the engine into the REASSIGNMENT and REASSIGNMENT_POTENTIALOWNERS tables.

Example test class:

com.packt.masterjbpm6.activity.TaskTest (testReassign method)

Example process:

reassign.bpmn

Description: The main process is started, and the task is assigned to Luigi. The reassign rule states that "if Luigi (named task actor) does not start his task in 60 seconds then the task should be assigned to Mario."

Notifications
A notification is the action of alerting someone (actor, group) when a task deadline expires. The default jBPM notification is e-mail based, and the default e-mail configuration is read from the userinfo.properties and email.properties files. The userinfo.properties file lists the user/group information in the following form:

entityId=email:locale:displayName: [member,member]

e.g., for an entity of type actor, we have:

nino=nino@domain.com:en-UK:nino

Member data is optional and is used for listing members belonging to a group organizational entity.
Please refer to the official jBPM 6.2 documentation for the configuration details.

The BPMN XML task parameters are `NotStartedNotify` and `NotCompletedNotify`.

An example `NotStartedNotify` parameter value follows:

```
from:mario|tousers:simo|togroups:|replyTo:|subject:warning|body:the task has not been started in 10s !@10s@not-started
```

**Delegation**

Delegation is the process of setting a task's potential owners. The actual owners, potential owners, or business administrators can delegate a task to another user, adding this user to the potential owners (if he/she isn't already) and making the user the task owner. A task can be delegated when it is in an active state (Ready, Reserved, or InProgress) and transitioned into the Reserved state, and its `skippable` property can be flagged to `true` (the target actor/owner can skip the task). The task's state and parameters will not change after delegation.

**Forward**

Task forwarding is the process performed by a potential owner on an active task who replaces himself in the potential owner list, passing the task to another person. The potential owner can only forward tasks when in the Ready state. If the task is in the Reserved or InProgress state, the task is transitioned to the Ready state again.

**Suspend/resume**

A task can be suspended in any of its active states (Ready, Reserved, or InProgress), transitioning it into the Suspended state. The Suspended state has sub-states to indicate the original state of the task. When resumed, the task transitions back to the original state from which it had been suspended.

**Skip**

A stakeholder working on a human task or a business administrator may decide that a task is no longer needed and hence, skip this task. This makes the task transition into the Obsolete state. The task can only be skipped if this capability is specified during the task configuration (the `skippable` property).
For delegate, forward and skip, and suspend/resume examples have a look at a test class:

```java
com.packt.masterjbpm6.task.TaskTest (methods
testDelegateReadyStateAndSkip, testForwardAndSkip,
testSuspendAndResume)
```

**Example process:**

```xml
delegate_forward.bpmn
```

Description: The main process is started, and a human task is reserved to Luigi. The test methods check for task delegation, forwarding, and suspend/resume.

**Release**

A task may be released by the current owner as a human task, making it available for other potential owners. From active states that have an actual owner (Reserved or InProgress), a task can be released and transitioned into the Ready state. Task data associated with the task is kept unchanged.

If a task is currently InProgress, it can be stopped by the actual owner, transitioning it into the Reserved state. Business data associated with the task as well as its actual owner is kept unchanged.

**Script Task**

A Script task is an automatic activity. When a process execution arrives at the Script task, the corresponding script is executed. All process variables that are accessible through the execution context (the `kcontext` variable) can be referenced within the script. It has the following properties:

- It is executed by the business process engine
- The script is defined in a language supported by the engine (Java or MVEL)
- The script task execution is always immediate
- The script task transitions to the complete state after the script execution

For a complete MVEL reference, please visit [http://mvel.codehaus.org/](http://mvel.codehaus.org/).

**Example test class:**

```java
com.packt.masterjbpm6.activity.ScriptTaskTest
```
Example process:

```java
script.bpmn
```

Description: The process script activity updates the process variable `order` description property:

```java
Order order=(Order)kcontext.getVariable("order");
order.setNote("order modified");
```

**Service Task**

The service task indicates the work that is to be automatically performed by a service provider. Usually, all work that has to be executed outside the engine should be designed as a service task. jBPM supports two types of service task implementations: plain Java class and Web service. The service task is backed by a WorkItemHandler implementation (org.jbpm.process.workitem.bpmn2.ServiceTaskHandler) registered with the name **Service Task**.

The parameters are as follows:

- **Interface**: Java class name or WSDL WebService service interface
- **Operation**: Java method name or WSDL WebService operation
- **Parameter**: Method name (to invoke)
- **ParameterType**: Method (to invoke) parameter type (only 1 parameter supported)
- **Mode (WS only)**: **SYNC** (default), **ASYNC**, or **ONEDAY**

In case of a service task of type Java, jBPM uses Java reflection to load the Java class type (by using an **Interface** parameter), instantiate it, and invoke the specified method (searched by **Operation** and **ParameterType**) with the value provided by **Parameter**. Only method signatures with a single parameter are supported, and the result of the invoked method is mapped in the activity **Results** output parameter.

The **Mode** parameter applies to a Web service only and describes the way a request has to be performed:

- **Synchronous (SYNC)**: Sends a request and waits for a response before continuing
- **Asynchronous (ASYNC)**: Sends a request and uses callback to get a response
- **Oneway**: Sends request without blocking (ignore response)

The Web service runtime leverages the "dynamic clients" features of the Apache CXF framework in order to generate Java classes at runtime.
A Service task can be really useful for rapid prototyping, but when it comes to complex external service integration, it falls short in meeting common development needs: multiple parameter passing, additional Web service configuration, and so on.

The following example demonstrates how to override the standard jBPM service task component by adding a custom workItem handler. Note, however, that the input/output parameters of the custom service task handler cannot be changed from the process designer because the task interface is defined in the configuration files of the jBPM workItem handlers.

WorkItem handlers are thoroughly explained in Chapter 7, Customizing and Extending jBPM.

Example test class:

```
com.packt.masterjbpm6.test.ServiceTaskTest
```

Example process:

```
servicetask.bpmn
```

Description: The first test (testJavaServiceTask) launches the process with a standard Java Service task (Interface: ServiceJavaTask, Operation: processOrder, Parameter: order, ParameterType: Order). The Service task changes the note field of the order and returns it to the main process whose script activity traces the change to the console. The second test (testJavaCustomServiceTask) features a custom Service task handler (PacktServiceTaskHandler) that overrides the default handler and processes the order parameter, setting its note property with a specific value.

**Rule Task**

The (Business) Rule tasks let us execute rules and get output from the embedded rule engine (Drools). Remember that process variables can be shared with the Rule tasks by using global variables or Drools session facts.

Example class:

```
com.packt.masterjbpm6.task.RuleTaskTest
```
Example knowledge artifacts:

rule.bpmn, rule.drl

Description: The main process is started, and the rule task triggers when the order cost is >100, and as a result, it changes the order's note property to URGENT. Look at the rule.drl file:

global StringBuffer newnote;
global com.packt.masterjbpm6.pizza.model.Order orderglobal;

rule "checkorder" ruleflow-group "masterRuleGroup"
when
  $o: com.packt.masterjbpm6.pizza.model.Order (cost>100)
then
  {
    System.out.println("checkorder triggered");
    String desc="big order ! (cost=+"+$o.getCost()+") ";
    orderglobal.setNote("URGENT");
    newnote.append(desc);
  }
End

The order variable (with cost > 100) is inserted into the knowledge session to activate the rule that triggers when Order (cost > 100); see the RuleTaskTest.testRule() method:

  ksession.insert(order);

While the shared orderglobal variable is used to get the result back:

  ksession.setGlobal("orderglobal", order);

Send/Receive Task

Send/Receive tasks are general-purpose messaging tasks since they do not provide a default implementation. They are handled as workItem and it is up to the implementer to back them with a working implementation through the WorkItemHandler interface, registering it with the jBPM WorkItemManager.

The workItem name of the receive task must be Receive Task. Receive Task refers to the message ID through the messageRef attribute; the handler receives the message ID value with the MessageId parameter.
BPMN Constructs

The workItem name of the send task must be **Send Task**. **Send Task** refers to the message ID through the `messageRef` attribute; for additional reference, check the Intermediate Message event.

Example class

```java
com.packt.masterjbpm6.task.TaskTest (method testSendReceive)
```

Example process artifacts:

```java
send_receive.bpmn
```

Description: The subprocess send task passes data to the receive task of the parent process. The test registers two custom workItem handlers, and the Send task and the Receive task share a message by using a global process variable.

**Manual Task**

A manual task defines a task that is to be performed externally to the engine. It is used to model work that is done by a stakeholder without interacting with the system; the engine does not know anything about the task, and it does not need to. There is no UI interface or system available for the manual task completion. For the engine, a manual task is managed as a passthrough activity. It continues the process from the moment process execution arrives into it.

**Ad hoc (Custom or None) Task**

The custom task is an empty, generic, unspecialized unit of work. The implementer is requested to provide a `WorkItemHandler` implementation for the task and register it with `WorkItemManager`

```java
Void registerWorkItemHandler(String workItemName, WorkItemHandler handler);
```
See Chapter 7, Customizing and Extending jBPM for detailed sections on the WorkItemHandler architecture.

The handler is registered for all workItems of the given workItemName and is called every time the process activates a node with that name. Further, workItemName must match the taskname attribute of the task element. WorkItemHandler is responsible for completing or aborting the task instance.

See the Conditional events section for a working example.

**Async tasks**

We are now going to take a closer look at some peculiar usage of the custom task. In Chapter 4, Operation Management we introduced the new jBPM executor service and the job scheduling features of the KIE console. The custom task can be conveniently configured to instruct the executor to call service-oriented components in an asynchronous fashion by scheduling an execution job in the background. The jBPM handler responsible for the job submission is org.jbpm.executor.impl.wih.AsyncWorkItemHandler (more on this in Chapter 7, Customizing and Extending jBPM).

The jBPM process designer gives you the ability to toggle a wait-for-completion flag on the workitem handler node. This flag does not reflect the sync/async nature of the handler invocation. It does tell the engine to evaluate (by an event listener) the handler results and map them back to the process context variables, using the task output mapping. If the flag is set to false, the custom task results will be ignored.

We can configure an async task by doing the following:

- Specifying async as the task taskName property
- Adding a data input parameter called CommandClass, and assigning a fully qualified Java class name to the schedule
- (Optional) adding a data input parameter called Retries, which tells the executor how many times the execution should be retried (default = 3)

Chapter 4, Managing Jobs and Asynchronous Command Execution explains in detail how to write Command classes.

The example that we discuss sets our AsyncTaskCommand as CommandClass, starts the executor service, and registers AsyncWorkItemHandler.
**Call Activity Task**

The call activity task is a general-purpose means to reuse existing, externally defined business constructs (process) simply by specifying their ID (the `calledElement` attribute of `bpmn2:callActivity`) or Name (`calledElementByName`). The execution of the called element can be synchronous/asynchronous (`waitForCompletion=true`) or independent (`independent=true`). You can set `independent` to false only if `waitForCompletion` is true.

All these properties are easily set, as usual, through both the jBPM Eclipse plugin or the KIE process editor; we extract from the process definition, for reference purposes, the relevant XML for the `callActivity` construct:

```xml
<bpmn2:callActivity drools:waitForCompletion="true"
    drools:independent="true" name="CallActivity"
    calledElement="callactivitySubprocess">
```

The following figure shows the main process on the left and the `callactivitySub1` process "zoomed out" from the CallActivity node:
The callee construct supports, like other activity nodes (tasks), data input and output mappings from/to the caller process, as we are going to see in the following example.

Example class:

```java
com.packt.masterjbpm6.task.CallactivityTaskTest
testIndependentSubprocess method
```

Example process artifacts:

```plaintext
callactivity.bpmn (parent process), callactivitySub1.bpmn (subprocess called by the callActivity construct)
```

Description: The main process is started and callActivity is executed; the main process passes the process order variable to callActivity. The callActivity subprocess modifies the order variable and returns it to the calling process definition.

As a side note, if we examine the PROCESSINSTANCELOG table, we can see the two instances of the processes (the main and the called process) logged; their parentship relation is saved through the PARENTPROCESSINSTANCEID column; it shows that callactivitySubprocess is a child process of the callactivityprocess. This is the output when callActivity has the independent=true and waitForCompletion=true properties set.

<table>
<thead>
<tr>
<th>PROCESSID</th>
<th>PROCESSINSTANCEID</th>
<th>PARENTPROCESSINSTANCEID</th>
<th>STATUS</th>
<th>START_DATE</th>
<th>END_DATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>callactivityprocess</td>
<td>1</td>
<td>null</td>
<td>2</td>
<td>2014-12-02 16:34:14.066</td>
<td>2014-12-02 16:34:14.253</td>
</tr>
<tr>
<td>callactivity/Subprocess</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2014-12-02 16:34:14.144</td>
<td>2014-12-02 16:34:14.191</td>
</tr>
</tbody>
</table>

Let us look at another example and see how the independent property affects the called subprocess.

Example class:

```java
com.packt.masterjbpm6.task.CallactivityTaskTest (method testAbortProcess)
```

Example process artifacts:

```plaintext
callactivityabort.bpmn (parent process), callactivitysubprocessabort.bpmn (subprocess called by the call activity construct)
```
Description: The `callactivityabort` process is started, and `callActivity` (with `independent=false`) is executed. The subprocess referenced by `callActivity` (`callactivitysubprocessabort`) has a human task, so it stops for user interaction. This gives us the time to issue (see the test class code) `abortProcessInstance` on the parent process. The `independent` flag set to `FALSE` forces `callActivity` (that is, the waiting subprocess) to abort contextually to the main process instance; when the flag is set to `TRUE`, the `callActivity` is not affected (see previous example).

This is the output when aborting the parent process instance, which has `callActivity` with the `independent=false` property set. Note also that `status = 3` (ABORTED) for both process instances.

<table>
<thead>
<tr>
<th>PROCESSID</th>
<th>PROCESSINSTANCE</th>
<th>PARENTPROCESSINSTANCE</th>
<th>STATUS</th>
<th>START_DATE</th>
<th>END_DATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>callactivityabort</td>
<td>1</td>
<td>null</td>
<td>3</td>
<td>2014-12-02 13:12:47.199</td>
<td>2014-12-02 13:13:00.647</td>
</tr>
<tr>
<td>callactivitysubprocessabort</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>2014-12-02 13:12:47.277</td>
<td>2014-12-02 13:13:00.667</td>
</tr>
</tbody>
</table>

**Subprocess**

A subprocess, as the name suggests, is a process that is included within another process. It can contain activities, events, gateways, and so on, which form a **boxed** process that is part of the enclosing process. The subprocess can be completely defined inside a parent process (an embedded subprocess) or can be linked through a `CallActivity` element by its ID or Name property. You can link a subprocess (by `callActivity`) across different multiple process definitions, reusing common groups of process elements (activities, gateways, and so on). The embedded subprocess construct can have multi-instance capabilities (see the MultiInstance section).

However, using a subprocess does impose the following constraints:

- Sequence flow cannot cross subprocess boundaries
- Process variables must be mapped for input and/or output

At the designer level, a subprocess can be expanded or collapsed so as to hide or show its details.

**Ad hoc subprocess**

Ad hoc subprocesses are commonly used when a number of tasks can be selected and performed in any order (because unspecified or unknown), and there is no execution dependency between them. Tasks might have unknown dependencies, most often because they are dynamic and managed by a human user on a case-by-case basis. The subprocess can complete even if some of the tasks are not executed at all. An ad hoc subprocess is represented as a subprocess with a tilde (˜) marker at the base.
The jBPM ad hoc subprocess implementation seems to be fairly incomplete. There seem to be some issues when exiting from the subprocess instance. The user is able to start the ad hoc subprocess activities by using the `signal` method by referencing the activity name:

```java
ksession.signalEvent("report1", null, processInstance.getId());
```

Because of their nature, ad hoc subprocesses are hard to design and of little use in real structured business processes; nevertheless, here, we provide you with an example that you can tweak and experiment with:

Example class:

```java
com.packt.masterjbpm6.task.AdHocSubprocessTest
```

Example process artifacts:

```ini
adhocsubprocess.bpmn
```

Description: The ad hoc subprocess has 2 script activities and 1 human task. The script tasks are signaled, and the human task is completed.
Multiple instances
This construct can be used to create multiple instances of a reusable subprocess definition as well as an embedded subprocess. Passing an input parameter collection works as the instantiation loop. jBPM will create one instance of the looping process for each element in the collection. The following figure shows the process with the embedded multi-instance subprocess (Log pizzas, the parallel symbol denotes that it is a multi-instance process) and the subprocess attributes. The loop input is the process variable list and the loop instance parameter (the collection item) is item of type Pizza. The item variable is visible in the instantiated subprocess scope.

Example class:

```java
com.packt.masterjbpm6.task.MultiInstanceTest
```

Example process artifacts:

```text
multiinstance.bpmn
```

Description: The process is created by passing a variable list of pizzas:

```java
List<Pizza> myList = new ArrayList<Pizza>();
myList.add(new Pizza(PizzaType.getType(Types.MARGHERITA), "margherita"));
myList.add(new Pizza(PizzaType.getType(Types.NAPOLI), "assorreta!");
params.put("list", myList);
ProcessInstance processInstance = ksession.startProcess("multiinstance", params);
```

Subsequently, two subprocess instances are created, and each is passed the loop item variable (a Pizza instance). The subprocess script activity simply prints the pizza description, and the subprocess exits.

```java
System.out.println("pizza desc " + item.getDesc());
```
Lanes

A lane is a partitioning box-shaped element used to group activities within the process definition. Lanes can be used to visually point out different group task assignments. For example, you can think of a lane as a company department (IT, business administration, and so on) where all employees have (more or less) the same duties. jBPM will try to assign (making a task reserved for the user) all tasks within the same lane to the same user. For example, if there are several tasks on a lane, the user who claimed and completed the first task will be assigned to the other tasks on the lane. Usually, it is convenient to assign the same group ID to all the tasks in the same lane.

Example class:

```java
com.packt.masterjbpm6.task.LaneTest
```

Example process artifacts:

```xml
lane.bpmn
```

Description: The task1 and task2 (on lane) activities are assigned to the pizzerianapoli group, while Mario's Task is assigned to the actor Mario. taskNotInLane is also assigned to pizzerianapoli but it's not on lane.

After the process is started, the actor Luigi (belonging to the pizzerianapoli group; see the LaneUserCallBack class) has 2 tasks on the list (task1 and taskNotInLane). After he completes task1, he is automatically given the task2 activity (status = Reserved), while the taskNotInLane status remains unchanged (Ready).
Data objects

Data objects are BPMN constructs that represent how data is required or produced by an activity. Data objects can have a direct association to one or more activity providing the input or the target output for that activity.

Example class:

```java
com.packt.masterjbpm6.task.DataObjectTest
```

Example process artifacts:

```
data-object.bpmn
```

Description: The task1 and task2 activities share the same data object (pizza class type); the first task produces the pizza, which then serves as the input of the second task.

Summary

In this chapter, we examined the jBPM BPMN constructs, providing hands-on working examples, tips, and, whenever possible, some details regarding the jBPM internal mechanisms. The chapter is not meant to be a BPMN tutorial or a BPMN best practices modeling guide for which we suggest picking more suitable books and a lot of real-world practice. In the next chapter, we will cover the jBPM subsystems API with several practical examples: the new Kie API, the runtime engine, the human task service, and the persistence engine.
Where to buy this book
You can buy Mastering jBPM6 from the Packt Publishing website.
Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.
Click here for ordering and shipping details.