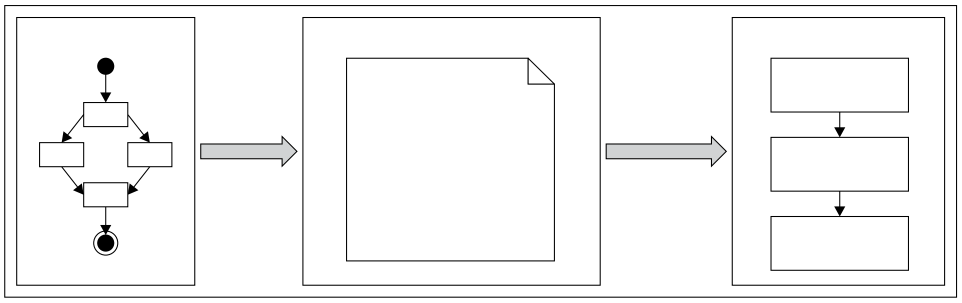
jPDL structure

The main goal of this section is to know how to write and express processes with jPDL XML syntax. This is important because we will do a deep analysis of each element that can be represented inside our process.

However, before that, we need to know where all the elements will be contained. If you remember, in Chapter 2, *jBPM for Developers*, we discussed about something called **process definition** (or just **definition**) that will contain the list of all the nodes which will compose our process. In this case, we will have a similar object to represent the same concept, but more powerful and with a complete set of functionalities to fulfill real scenarios. The idea is the same, but if jPDL has XML syntax, how are these XML tags translated to our Definition object?

This translation is achieved in the same way that we graph our defined process in our simple GOP implementation. However, in this case we will read our defined process described in the `processdefinition.xml` file in order to get our object structure. In order to make this situation more understandable, see the following image:



We could say that the `processdefinition.xml` file needs to be translated to objects in order to run.

In the rest of this chapter, we will see how all these "artifacts" (graph representation of our process → jPDL XML description → Object Model) come into play. Analyzing the basic nodes, starting from the design view and the properties window, to the translation to XML syntax, and how this XML becomes running objects.

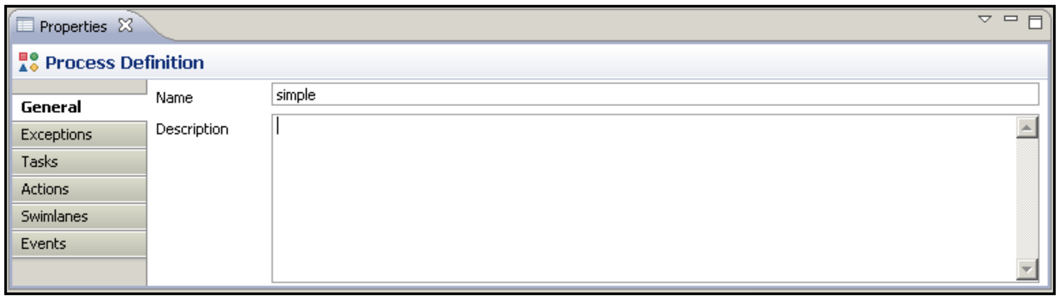
It's necessary to understand this transformation, in order to know how our processes need to be designed. This will also help us to understand each property's meaning for each particular node; showing us how each property will influence the design and execution stage of our process.

Process structure

It's important to note that one `processdefinition.xml` file, generated or not with GPD, will represent just one business process. There is no way to put more than one process in one file, so do not try it at home.

The process designed with GPD will be automatically translated to jPDL XML syntax, so if you don't have the plugin, you will need to write this jPDL XML syntax by hand (a common practice for advanced developers who know jPDL). This XML will be the first technical artifact that we need to know in depth. So, here you will find how this file is internally composed and structured.

If you create a new process and select the background (not an element), you will see the following **Properties** window:



Name: simple
Description:

This panel will represent all of the global properties that can be set to a `process-definition` element. Remember that this element will contain all the nodes in our process, so all the global information must be placed here. As you can see in the image, global exceptions, tasks, actions, and events can be configured here.

If you now switch to the Source tab, you will find that basically, one process definition represented in XML jPDL syntax needs to have the following structure:

```
<process-definition name="simple">
  <node>
    <transition></transition>
      ...
  </node>
  ...
</process-definition>
```

As you can notice, the root node will be a `<process-definition>` XML tag that will accept a collection of nodes, where each of these nodes will accept a collection of leaving transitions.

This structure will help us to quickly understand how the process flow works without having a graphical representation. We only need a little bit of imagination.

As you can imagine, this process definition tag and all of the elements inside it will be parsed and transformed into objects by the jBPM framework. The **Java** class that will represent this `<process-definition>` tag will be the `ProcessDefinition` class.

Here we will analyze this class, but only how the basic concepts are implemented.

The `ProcessDefinition` class can be found in the `org.jbpm.graph.def` package, inside the core module's `src/main/java` directory.

Here we are talking about the checked out project from the SVN repository, not the binary distribution.

This class is in charge of containing and representing all the data described in the `processdefinition.xml` file. It also includes a few extra meta data that will be useful in the execution stage of our processes.

If you open this class (recommended, as you will learn a lot about the internal details of the framework and you will also start feeling comfortable with the code). The first thing that you will notice is that the class inherits functionality from a class called `GraphElement` and implements the `NodeCollection` interface.

```
public class ProcessDefinition extends GraphElement
implements NodeCollection
```

The `GraphElement` class will give the `ProcessDefinition` class all the information needed to compose a graph and also some common methods for the execution stage. The most common properties that we, as developers, will use are the following:

- `long id = 0;`
- `protected String name = null;`
- `protected String description = null;`

These properties will be shared through all the elements that can be part of our business process graph (Nodes and the Process Definition itself).

It is also important to see all the methods implemented inside the `GraphElement` class, because they contain all the logic and exceptions related to events inside our processes. But some of these concepts will be discussed later, in order not to confuse you and mix topics.

NodeCollection methods

The `NodeCollection` interface will force us to implement the following methods to handle and manage collections of nodes:

```
List<Node> getNodes();
Map<String, Node> getNodesMap();
Node getNode(String name);
boolean hasNode(String name);
Node addNode(Node node);
Node removeNode(Node node);
```

Feel free to open the `GraphElement` class and the `NodeCollection` interface in order to take a look at other implementations' details.

ProcessDefinition properties

Now it is time to continue with the `ProcessDefinition` properties.

Right after the class definition, you will see the property definition section, all these properties will represent the information about the whole process. Remember that the properties inherited for the `GrapElement` class are not shown here. The most meaningful ones are as shown in the following table. These properties represent core information about a process that you will need to know in order to understand how it works:

Property	**Description**
`Node startState`	It represents the node that will be the first node in our process, as you can see, this property is not restricted to the `StartState` type. It is this way, because this `Node` class could be reused for another language that could define another type of start node.
`List<Node> nodes`	It represents the collection of nodes included between the `<process-definition>` tags in the `processdefinition.xml` file.
`transient Map<String, Node> nodesMap`	This property allows us to query all the nodes in our process by name, without looping through all the nodes in the list. With just one string, we can get the node that we are looking for. As this property is transient, it will not be persisted with the process status. This means that this property will be filled when the process is in the execution stage only.
`Map<String, Action> actions`	It represents global actions (custom code) that will be bonded to a name (String) and could be reused in different nodes of our process. This feature is very helpful to reuse code and configurations. It also keeps the `processdefinition.xml` file as short as possible.
`Map<String, ModuleDefinition> definitions`	It represents different internal/external services that could be accessed by the process definition, we will learn more about these modules later.

This is all that you need to know about the information maintained as process definition level.

Functional capabilities

Now we need to see all the functionality that this class provides in order to handle our process definitions.

An array of strings is defined to store the events supported by the process definition itself.

```
// event types ////////////////////////////////////////////////////
////////
public static final String[] supportedEventTypes = new String[]{
  Event.EVENTTYPE_PROCESS_START,
  Event.EVENTTYPE_PROCESS_END,
  Event.EVENTTYPE_NODE_ENTER,
  Event.EVENTTYPE_NODE_LEAVE,
  Event.EVENTTYPE_TASK_CREATE,
  Event.EVENTTYPE_TASK_ASSIGN,
  Event.EVENTTYPE_TASK_START,
  Event.EVENTTYPE_TASK_END,
  Event.EVENTTYPE_TRANSITION,
  Event.EVENTTYPE_BEFORE_SIGNAL,
  Event.EVENTTYPE_AFTER_SIGNAL,
  Event.EVENTTYPE_SUPERSTATE_ENTER,
  Event.EVENTTYPE_SUPERSTATE_LEAVE,
  Event.EVENTTYPE_SUBPROCESS_CREATED,
  Event.EVENTTYPE_SUBPROCESS_END,
  Event.EVENTTYPE_TIMER
};
public String[] getSupportedEventTypes() {
  return supportedEventTypes;
}
```

These events will represent hook points to attach extra logic to our process, which will not modify the graphic representation of the process. These events are commonly used for adding technical details to our processes and have a tight relationship with the graph concept. This is because each `GraphElement` will have a life cycle that can be defined where events will be fired. We will continue talking about events in the following chapters.

Constructing a process definition

In this section, we will find different ways to create and populate our process definition objects. This section will not describe the `ProcessDefinition` constructors because they are rarely used, we will directly jump to the most used methods in order to create new `ProcessDefinition` instances.

In most cases, instances of the `parseXXX()` method will be used to create `ProcessDefinition` instances that contain the same structure and information as a `processdefinition.xml` file.

Similar methods are provided to support different input types of process definitions, such as the following ones:

- `parseXmlString(String)`
- `parseXMLResource(String)`
- `parseXMLReader(Reader)`
- `parseXMLInputStream(InputStream)`
- `parseParZIPInputStream(ZipInputStream)`

The only difference between all of these methods is the parameters that they receive. All of these methods will parse the resource that they receive and create a new `ProcessDefinition` object that will represent the content of the XML jPDL `processdefinition.xml` file.

The most simple parse method will take a `String` representing the process definition and return a brand new `ProcessDefinition` object created by using the string information. This `String` needs to represent the correct jPDL process definition in order to be correctly parsed.

The most commonly used will be the one that uses a path to locate where the `processdefinition.xml` file is and creates a brand new `ProcessDefinition` object.

It is good to know how we will construct or create a new process definition object that will reflect the process described in the XML jPDL syntax. It is also important to know how this generation is done. It could be helpful to know how the framework works internally and where all the information from the XML process definition is stored in the object world.

When we finish parsing all the elements inside the XML file, if our process definition doesn't have any errors, a brand new `ProcessDefinition` object will be returned.

Just for you to know, this kind of parsing in real applications is only used a few times. As you can imagine, this parsing process could take a while when you have a large number of nodes inside it, but this is just a note, don't worry about that.

Adding custom behavior (actions)

If you jump to the `actions` section (in the `ProcessDefinition` class, marked with the comment `// actions`), you will find methods for adding, removing and getting these custom technical details called actions. These actions could be used and linked in different stages (graph events) and nodes in the current process. These actions are global actions that must be registered with a name, and then referenced by each node that wants to use them. These process-defined actions are commonly used for reuse code and configuration details. This will also keep your process definition XML file clean and short. If you look at the code, you will find that a bidirectional relationship is maintained between the action and the process definition.

```
public Action addAction(Action action) {
  if (action == null) throw new IllegalArgumentException
     ("can't add a null action to an process definition");
  if (action.getName() == null) throw new IllegalArgumentException
    ("can't add an unnamed action to an process definition");
  if (actions == null) actions = new HashMap<String, Action>();
    actions.put(action.getName(), action);
    action.processDefinition = this;
    return action;
}
```

The bi-directional relationship between Actions and the process definition will allow finding out how many action definitions the process contains, to be able to dynamically define actions in different places in the runtime stage.

I think that no more notes could be written about `ProcessDefinition`. Let's jump to the basic nodes section. But feel free to analyze the rest of the process definition class code, you will only find Java code, nothing to worry about.

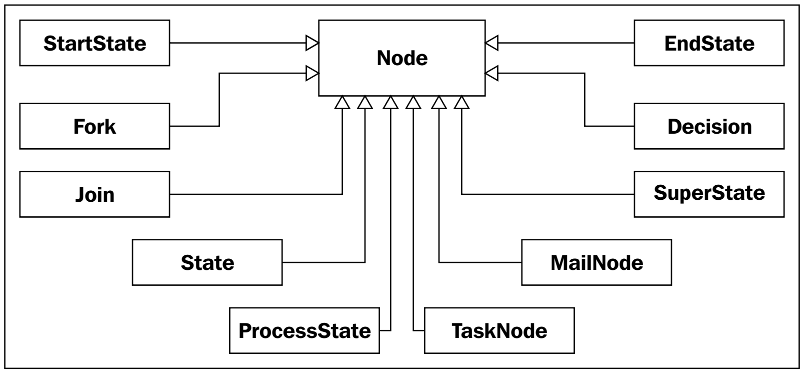
Nodes inside our processes

Inside our `<process-definition>` tag, we will have a collection (set) of nodes. These nodes could be of different types and with different functionalities. You should remember Chapter 2, *jBPM for Developers*, where we discussed about GOP and created a new GOP language. This custom language used node hierarchy to achieve this multiple behavior and functionalities. It expands language capabilities by adding new words to our language, which are represented by different types of nodes.

jPDL is basically that; a main node which implements the basic functionality and then a set of subclasses that conform the language.

This language (jPDL) contains 12 words/nodes in this version (in the GPD palette). These nodes implement some basic and generic functionalities that, in most cases, it's just logic about whether the process must continue the execution to the next node or not. This logic is commonly named *propagation policies*.

If we want to understand how each word behaves, how it is composed, and which "parameters" need to be filled in order to work correctly, firstly we will need to understand how the most basic and generic node behaves. This is because all the functionalities inside this node will be inherited, and in some cases overridden, by the other words in the language.



For this reason, we will start a deep analysis about how the `Node` class is implemented and then we will see all the other nodes, just mentioning the changes that are introduced for each one.

To complete this section, we will just mention some details about the parsing process.

ProcessDefinition parsing process

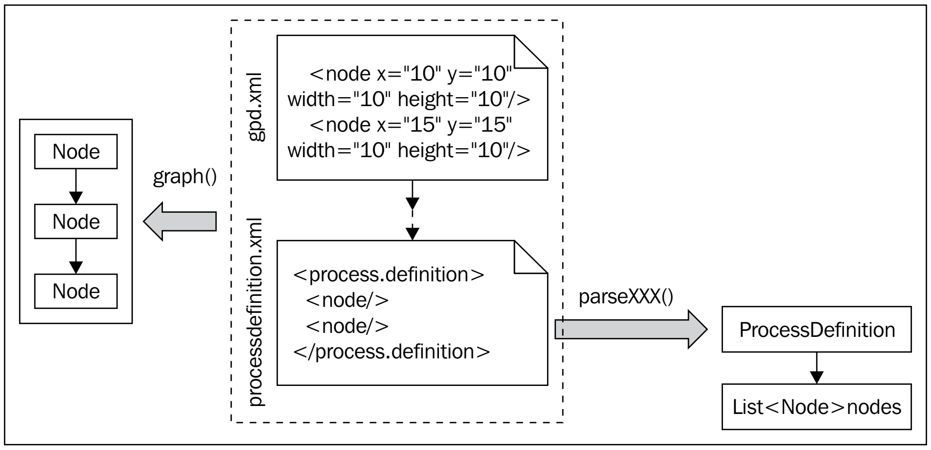
This parsing process begins when we load the `processdefinition.xml` file using some of the `parseXXX()` methods of the `ProcessDefinition` class. These methods internally use the `JpdlXMLReader` class to parse all the content of the `processdefinition.xml` file. It's important to know that this `JpdlXMLReader` class is designed to work with DOM elements. One of the core methods of the `JpdlXMLReader` class is the following method:

```
public ProcessDefinition readProcessDefinition()
```

This method is in charge of parsing all of the process definition XML elements and creating all the Objects needed to represent the process structure.

In this method, we will find the section that will read each part of the process definition shown as follows:

```
readSwimlanes(root);
readActions(root, null, null);
readNodes(root, processDefinition);
readEvents(root, processDefinition);
readExceptionHandlers(root, processDefinition);
readTasks(root, null);
```



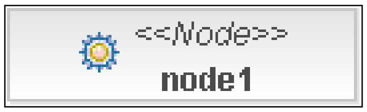
It is important to note that the graphical information stored in the `gpd.xml` file is neither parsed nor stored in the `ProcessDefinition` object. In other words, it is lost in this parsing process and if you don't keep this file, the elements' position will get lost. Once again, the absence of this file will not influence the definition and the execution of our defined process.

Base node

As we have seen before, this node will implement logic that will be used by all the other words in our language, but basically this class will represent the most common lifecycle and properties that all the nodes will have and share.

With these nodes' hierarchy, our process definition will contain only nodes causing that all the nodes in the palette will be of the `Node` type as well as all of its subclasses.

First of all, we will see how the node concept is represented in jPDL XML syntax. If you have GPD running, create a new process definition file and drag a node of the `Node` type inside your process.



Note the icon inside the node rectangle, a **gear**, is used to represent the node functionality, meaning that the base functionality for a `Node` is the generic work to be done, which can probably be represented with a piece of Java code.

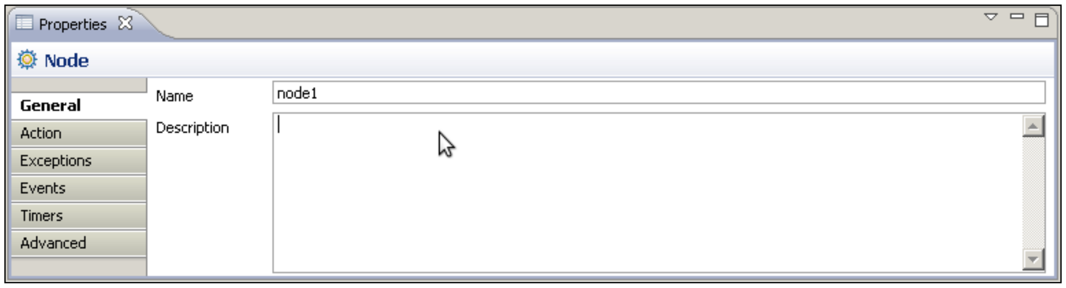
This means that something technical is needed in our business process. That is why the **gear** appears there, just to represent that some "machine working" will happen during this activity of our business process. As you will see a few sections later, if the technical details are not important for the business analysts, you can add them in other places, which are hidden from the graphical view. This will help us to avoid increasing the complexity of the graphical view with technical details, that doesn't mean anything to the real business process.

An example of that could be a backup situation—if one activity in our process takes a backup of some information, we will need to decide if the backup activity will be shown in the graphical representation (as a node) of our process depending on the context of the process, or if it will be hidden in the technical definitions behind the process graph.

In other words, you will only use these type of nodes if the Business Analyst team tells you that some important technical details are part of the process, and these technical details need to be displayed in the process diagram as an activity.

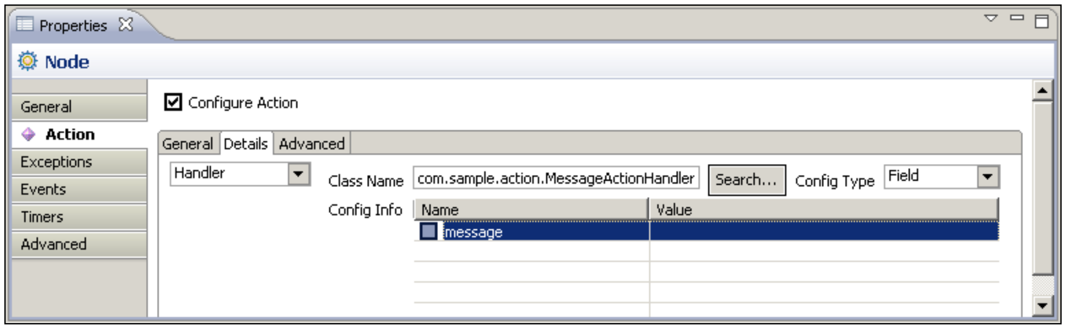
In the jBPM context, we will use "technical detail" to refer to all of the code needed to be able to run our process inside an execution environment. Do not confuse this with something minimal or less important.

Let's analyze this node—if you have GPD plugin installed, select the node dropped before, and go to the properties panel. Here you will see that some basic information could be inserted as name, description, and so on. Just add the basic information, save the process definition, and go to the sources tab to see how this information is translated inside the node tag.



Name: node1
Description:

In order to see these basic node properties, you could open the `Node` class to see how these properties are reflected in the Java code. As we discussed before, this class will represent the execution of technical details. So, if we select the node in GPD and see the properties window, we will see that we have an **Action** tab that has a checkbox to activate the configuration from this action. This will represent the added technical details that will be executed when the process execution enters into this node. These technical details could be anything you want. When you activate this option, you will see that new fields appear asking about information that will describe this action.



☑ Configure Action

Handler

Class Name: com.sample.action.MessageActionHandler

Config Type: Field

Config Info:

Name	Value
message	

If you read some documentation about this, you will see that these actions are called **delegated actions**. This name is because these actions are in some way delegated to external classes that will contain the logic to execute. These delegated classes are external to the framework. In other words, we will implement these delegated classes and we will just tell the process the class name that contains the action, then the framework will know how to execute this custom logic.

```
<node name="node1">
    <action class="com.sample.action.MessageActionHandler"></action>
</node>
```

In order to achieve this functionality, the command design pattern (click on `http://en.wikipedia.org/wiki/Command_pattern` for more information) is applied. Therefore, we only need to implement a single method interface called `ActionHandler` in our class. We will see more about how to do this in the next chapter where we build two real, end-to-end applications. You must keep in mind that this action can include custom logic that you will need to write. This can be done by implementing the `ActionHandler` interface that the framework knows how to execute.

Until this point, we have a node (of the `Node` type) graphed in GPD, also expressed in jPDL XML syntax with the tag `<node>` that is kept in sync with the graphical diagram by the plugin. When we load the `processdefinition.xml` file in a new `ProcessDefinition` object, our node (written and formally described in jPDL) will be transformed in one instance of the `Node` class. The same situation will occur with all of the other node types, because all the other nodes will be treated as `Node` instances.

Here we will analyze some technical details implemented in the node class that represent the node generic concepts and the implementation of nodes that can be used in our processes.

This class also implements the `Parseable` interface that forces us to implement the `read()` and `write()` methods in order to understand and be able to write the jPDL XML syntax, which has been used in our process definitions.

```
public interface Parsable {
  void read(Element element, JpdlXmlReader jpdlReader);
  void write(Element element);
}
```

Information that we really need to know about each node

Leaving transitions are one of the most important properties that we need to store and know.

```
protected List<Transition> leavingTransitions = null;
```

This list is restricted only to store `Transition` objects with generics. This list will store all of the transitions that have the current node as the source node.

The `action` property is also an important one, because this property will store the action that will be executed when the current node is in the execution stage.

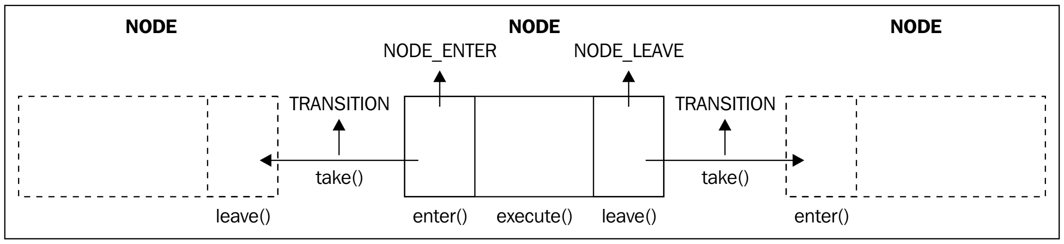
It is important to note that a public `Enum` is defined here to store each type of node that could be defined using this super class.

```
public enum NodeType { Node, StartState, EndState, State, Task, Fork,
Join, Decision };
```

This `enum` specifies the built-in nodes inside the framework. If you create your own type of node, you will need to override the `getNodeType()` method to return your own custom type.

Node lifecycle (events)

The following section, the events section, marked with a comment `//event types////..` in the `Node` class, is used to specify the internal points where the node execution will pass through. These points will represent hook points where we can add the custom logic that we need. In this case, the base node, support events/hook points called `NODE_ENTER`, `NODE_LEAVE`, `BEFORE_SIGNAL`, and `AFTER_SIGNAL`. This means that we will be able to add custom logic to these four points inside the node execution.



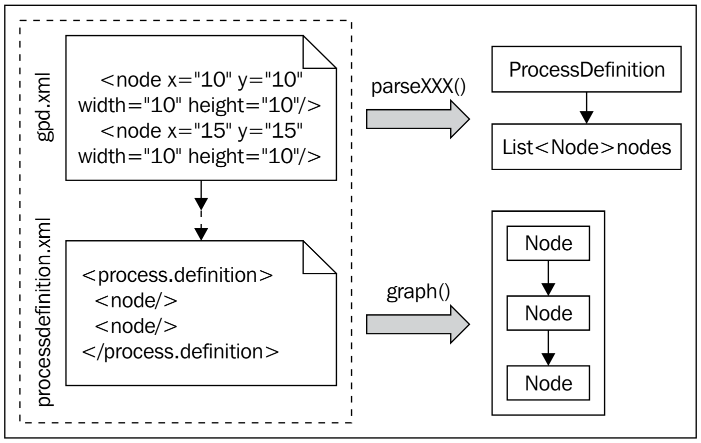
The `BEFORE_SIGNAL` and `AFTER_SIGNAL` events will be described later when we discuss external events/triggers that could influence the process execution.

Constructors

The node class instances will rarely be constructed using the following constructors:

```
public Node()  {  }
public Node(String name)  {
  super(name);
}
```

In most of the cases the instances of node class will be created by the method `parseXXX()` that reads the whole process definition and all the nodes inside it. So, in most cases we don't need to create nodes by hand. However, it is important for us to know how this parsing process is done.



Managing transitions / relationships with other nodes

If we observe the section delimited with the `//leaving transitions//` and `//arriving transitions//` comments, we will find a few methods to manage all of the transitions related to some nodes in our process.

As we have seen before, the transitions for a node are stored in two properties of type list called `leavingTransitions` and `arrivingTransitions`. We have also a helper map to locate each transition inside a particular node by name. In this section of the node class, we will find wrapper methods to these two lists that also add some very important logic.

For example, if we take a look at the method called `addLeavingTransition(Transition)`, we can see the following piece of code:

```
public Transition addLeavingTransition(Transition
                                 leavingTransition)
{
  if (leavingTransition == null)
    throw new IllegalArgumentException("can't add a null
                         leaving transition to an node");
    if (leavingTransitions == null)
      leavingTransitions = new ArrayList<Transition>();
      leavingTransitions.add(leavingTransition);
      leavingTransition.from = this;
      leavingTransitionMap = null;
      return leavingTransition;
}
```

Where the first few lines of this method check to see if the list of `leavingTransitions` is null. If this is true, it will only create a new `ArrayList` to store all the transitions from this node. This is followed by the addition of new transitions to the list, and then the node reference is added to the recently added transition. At last, the `leavingTransitionMap` is set to `null` in order to be generated again, if the method `getLeavingTransitionMap()` is called. This is done in order to keep the transition map updated with the recently added transition.

Another important method is called `getDefaultLeavingTransition()`, this method logic will be in charge of defining which transition to take if we do not specify a particular one. In other words, you must know how this code works in order to know which transition will be taken.

```
public Transition getDefaultLeavingTransition()
{
  Transition defaultTransition = null;
  if (leavingTransitions != null)
  {
    // Select the first unconditional transition
    for (Transition auxTransition : leavingTransitions)
    {
      if (auxTransition.getCondition() == null)
      {
        defaultTransition = auxTransition;
        break;
      }
    }
  }
```

```
    else if (superState != null)
    {
      defaultTransition = superState.getDefaultLeavingTransition();
    }
    return defaultTransition;
  }
```

If you see the code inside this method, you will see that the first unconditional transition will be chosen if no other transition is selected. It is also important to see that if there is a situation with nested nodes, the parent node will be also queried for a default transition.

Runtime behavior

Up until this point, we have seen how and where the information is kept, but from now on, we will discuss how this node will behave in the execution stage of our processes.

The first method in the `//Behavior methods//` comment delimited section is the method called `enter(ExecutionContext)`.

> The `ExecutionContext` class is used by the `enter()`, `execute()`, and `leave()` methods in order to know all the contextual information needed to execute each phase inside the node.

We already see the Node life cycle graph, where this method will be the first method called when the node is reached.

It's very important to see all the code in this method, because it give us the first phase in the execution life cycle of our node.

```
  public void enter(ExecutionContext executionContext)
  {
    Token token = executionContext.getToken();
    // update the runtime context information
    token.setNode(this);
    // fire the enter-node event for this node
    fireEvent(Event.EVENTTYPE_NODE_ENTER, executionContext);
    // keep track of node entrance in the token,
    so that a node-log can be generated at node leave time.
    token.setNodeEnter(Clock.getCurrentTime());
    // remove the transition references from the runtime context
    executionContext.setTransition(null);
    executionContext.setTransitionSource(null);
    // execute the node
    if (isAsync)
    {
      ExecuteNodeJob job = createAsyncContinuationJob(token);
      MessageService messageService = (MessageService)Services.
                getCurrentService(Services.SERVICENAME_MESSAGE);
      messageService.send(job);
      token.lock(job.toString());
    }
    else
    {
      execute(executionContext);
    }
  }
```

Here in the first lines of the method appears the concept of `Token` that will represent where the execution is, at a specific moment of time. This concept is exactly the same as the one that appears in Chapter 2, *jBPM for Developers*.

That is why, this method gets the `Token` from the `Execution Context` and changes the reference to the current node. If you see the following lines, you can see how this method is telling everyone that it is in the first phase of the node life cycle.

```
  // update the runtime context information
  token.setNode(this);
  // fire the enter-node event for this node
  fireEvent(Event.EVENTTYPE_NODE_ENTER, executionContext);
```

If you analyze the `fireEvent()` method that belongs to the `GraphElement` class, you will see that it will check whether some action is registered for this particular event. If there are some actions registered, just fire them in the defined order.

As you can see at the end of this method, the `execute()` method is called, jumping to the next phase in the life cycle of this node. The `execute()` method is called as follows:

```
  public void execute(ExecutionContext executionContext)
  {
    // if there is a custom action associated with this node
    if (action != null)
    {
      try
      {
        // execute the action
        executeAction(action, executionContext);
      }
      catch (Exception exception)
      {
        raiseException(exception, executionContext);
      }
    }
    else
    {
      // let this node handle the token
      // the default behaviour is to leave the
              node over the default transition.
      leave(executionContext);
    }
  }
```

In this `execute()` method, the base node functionality implements the following execution policy. If there is an action assigned to this node, execute it. If not, leave the node over the default transition.

This node functionality looks simple, and if I ask you if this node behaves as a wait state, you will probably think that this node never waits. Having seen the code above, we can only affirm that if no action is configured to this type of node, the default behavior is to continue to the next node without waiting. However, what happens if there is an action configured? The behavior of the node will depend on the action. If the action inside it contains a call to the `executionContext.leaveNode()` method, the node will continue the execution to the next node in the chain (of course, passing through the `leave()` method). But if the action does not include any call to the `leave()` method, the node as a whole will behave like a wait state.

If this node does not behave like a wait state, the execution will continue to the next phase calling the `leave()` method.

```
  public void leave(ExecutionContext executionContext,
                               Transition transition)
  {
    if (transition == null)
      throw new JbpmException("can't leave node '" + this + "'
                          without leaving transition");
      Token token = executionContext.getToken();
      token.setNode(this);
      executionContext.setTransition(transition);
      // fire the leave-node event for this node
      fireEvent(Event.EVENTTYPE_NODE_LEAVE, executionContext);
      // log this node
```

```
        if (token.getNodeEnter() != null)
        {
          addNodeLog(token);
        }
        // update the runtime information for taking the transition
        // the transitionSource is used to calculate
                                events on superstates
        executionContext.setTransitionSource(this);
        // take the transition
        transition.take(executionContext);
    }
```

This method has two important lines:

- The first one is the one that fires the `NODE_LEAVE` event telling everyone that this node is in the last phase before taking the transition out of it, where actions could be attached like the `NODE_ENTER` event
- The second line is the one at the end of this method where the execution gets out of the current node and enters inside the transition:

  ```
  transition.take(executionContext);
  ```

This is the basic functionality of the Node class. If the subclasses of the Node class do not override a method, the functionality discussed here will be executed.