The Process Virtual Machine

A library for building executable state machines. It can can serve as the foundation for any form of BPM, workflow and orchestration.

Table of Contents

1

Introduction

1.1. Scope and target audience

This is a tutorial that introduces the Process Virtual Machine library to Java developers.

1.2. Processes and executions

With this library you can build executable process graphs. The key features of this library are

- Create executable processes that are based on a diagram structure
- Runtime behaviour of the nodes can be provided as Activity implementations
- Activities can be wait states
- There are no constraints on the process graph structure
- Processes diagrams can be based on composition (aka block structured)
- Processes diagrams can be a mix of graph based and composition
- During wait states, the runtime state of a process execution can be persisted
- Persistence is optional

Process definitions are static and define an execution analogue to a Java class. Many executions can be run against the same process definition. One execution is also known as a process instance and that is analogue to a Java object. An execution maintains the current state for one execution of the process, including a pointer to the current node.

1.3. Overview

1.3.1. Part One

The first part of this manual gives a thorough introduction on how to implement Activity's. This means creating the runtime implementation for the process constructs (aka activity types) that are defined in the process languages.

Chapter 2 explains how to create process graphs, how process graphs are executed and how Activities can be build

that implement the runtime behaviour of nodes in the process graph.

Chapter 3 uses the basic graph execution techniques to show how concrete activities are implemented in meaningfull setting.

Chapter 4 explains the more fine grained details of graph execution like the relation to threads, looping, sub processes and so on.

Chapter 5 are Java classes that are used as part of the process execution, but are not part of the pvm library.

Chapter 6 captures contextual information related to a process execution. Think of it as a `Map<String, Object>` that is associated with a process execution.

Chapter 7 shows the infrastructure for generating auditable events from the process. This is the information that will be fed into the history database that can be queried for statistical information about process execution (aka Business Intelligence).

1.3.2. Part Two

The second part explains the embeddable infrastructure. That infrastructure makes it possible to use multiple transactional resources inside the process execution and configure them to operate correctly in standard and enterprise Java.

Chapter 8 is the core abstraction layer for the specific Java environment in which the process operates. Transactional resources can be fetched from the environment. The environment will take care of the lazy initialization of the transactional resources based on the configuration.

Chapter 9 shows how process definitions and process executions can be stored in a relational database. It is also explained how hibernate is integrated into the environment and how concurrency is handled.

Chapter 10 are the session facades that are exposed to programmatic clients using the PVM functionality. They are based on commands and use the environment infrastructure.

1.3.3. Part Three

Part three explains two PVM infrastructure features that are based on transactional resources and require the execution in separate a thread. The job executor that is part of the PVM can execute jobs in a standard Java environment. Alternatively, there are implementations for messaging and timers that can be bound to JMS and EJB Timers respectively in an enterprise environment.

Chapter 11 are declarative transaction demarcations in a process. This functionality depends on an asynchronous messaging service.

Chapter 12 can fire pieces of user code, related to an execution in the future.

1.3.4. Part Four

In pPart four, Chapter 13 describes the main steps involved in building a complete process language implementation.

1.4. JVM version

jbpm-pvm.jar requires a JVM version 5 or higher.

1.5. Library dependencies

For building and executing processes the jbpm-pvm.jar does not have any other dependencies then on the JVM. If you're using DB persistence, then there is a dependency on hibernate and it's dependencies. More information about the optional depedencies can be found in the lib directory [../../lib/optional-dependencies.html].

1.6. Logging

All jBPM modules use standard java logging [http://java.sun.com/j2se/1.4.2/docs/guide/util/logging/overview.html]. If you don't like the verbosity of the 2-line default logging output, Here's how you can configure a single line logging format in the code without using the `-Djava.util.logging.config.file=...` command line parameter:

```
InputStream stream = YourClass.class
        .getClassLoader()
        .getResourceAsStream("logging.properties");

try {
  LogManager.getLogManager().readConfiguration(stream);
} finally {
  stream.close();
}
```

Typically such code would be put in a static block in one of the first classes that is loaded in your application. Then put a logging.properties file in the root of the classpath that looks like this:

```
handlers = java.util.logging.ConsoleHandler
java.util.logging.ConsoleHandler.level = FINEST
java.util.logging.ConsoleHandler.formatter = org.jbpm.util.JbpmFormatter

# For example, set the com.xyz.foo logger to only log SEVERE messages:
# com.xyz.foo.level = SEVERE

.level = SEVERE
org.jbpm.level=FINEST
org.jbpm.tx.level=FINE
org.jbpm.wire.level=FINE
```

1.7. Debugging persistence

When testing the persistence, following logging configurations can be valuable. `SQL` shows the SQL statement that is executed and `type` shows the values of the parameters that are set in the queries.

```
org.hibernate.SQL.level=FINEST
org.hibernate.type.level=FINEST
```

And in case you get a failed batch as a cause in a hibernate exception, you might want to set the batch size to 0 like this in the hibernate properties:

```
hibernate.jdbc.batch_size = 0
```

Also in the hibernate properties, the following properties allow for detailed logs of the SQL that hibernate spits out:

```
hibernate.show_sql = true
hibernate.format_sql = true
hibernate.use_sql_comments = true
```

2

Basic graph execution

2.1. Activity

The PVM library doesn't have a fixed set of process constructs. Instead, runtime behaviour of a node is delegated to an `Activity`. In other words, `Activity` is an interface to implement the runtime behaviour of process constructs in plain Java. Also, `Activity` implementations can be subscrribed as listeners to process events.

```
public interface Activity extends Serializable {
  void execute(Execution execution) throws Exception;
}
```

`Activity`'s can be used as node behaviour and as listeners to process events. When an activity is used as the node behaviour, it is in full control of the further propagation of the execution. In other words, a node behaviour can decide what the execution should do next. For example, it can take a transition with `execution.take(Transition)`, go into a wait state with `execution.waitForSignal()`. Or the node behaviour can not invoke any of the above, in that case the Process Virtual Machine will just proceed the execution in a default way.

Events are only fired during process execution. Since during an event the execution is already 'in motion', event listeners can not control the propagation of execution. Therefore, `Activity` implementations can only be used as event listeners if they don't invoke any of the execution propagation methods.

This way, it is very easy to implement automatic activities that can be used as node behaviour as well as event listeners. Examples of automatic activities are sending an email, doing a database update, generating a pdf, calculating an average, etc. All of these can be executed by the process system and they can be used both as node behaviour as well as event listeners. In case they are used as node behaviour they can rely on the default proceed behaviour.

2.2. Activity example

We'll start with a very original hello world example. A Display activity will print a message to the console:

```
public class Display implements Activity {

  String message;

  public Display(String message) {
    this.message = message;
  }

  public void execute(Execution execution) {
    System.out.println(message);
  }
}
```

Let' build our first process definition with this activity:

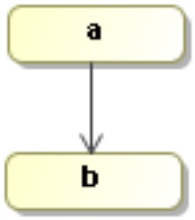

Figure 2.1. Activty example process

```
ProcessDefinition processDefinition = ProcessFactory.build()
    .node("a").initial().behaviour(new Display("hello"))
      .transition().to("b")
    .node("b").behaviour(new Display("world"))
.done();
```

Now we can execute this process as follows:

```
Execution execution = processDefinition.startExecution();
```

The invocation of `startExecution` will print hello world to the console:

```
hello
world
```

One thing already worth noticing is that activities can be configured with properties. In the Display example, you can see that the message property is configured differently in the two usages. With configuration properties it becomes possible to write reusable activities. They can then be configured differently each time they are used in a process. That is an essential part of how process languages can be build on top of the Process Virtual Machine.

2.3. ExternalActivity

External activities are activities for which the responsibility for proceeding the execution is transferred externally, meaning outside the process system. This means that for the system that is executing the process, it's a wait state. The execution will wait until an external trigger is given.

For dealing with external triggers, `ExternalActivity` adds two methods to the `Activity`:

```
public interface ExternalActivity extends Activity {

  void signal(Execution execution,
              String signal,
              Map<String, Object> parameters) throws Exception;

  Set<SignalDefinition> getSignals(Execution execution) throws Exception;

}
```

Just like with plain activities, when an execution arrives in a node, the `execute`-method of the node behaviour is in-

voked. In external activities, the execute method typically does something to transfer the responsibility to another system and then enters a wait state by invoking `execution.waitForSignal()`. For example in the execute method, responsibility could be transferred to a person by creating a task entry in a task management system and then wait until the person completes the task.

In case a node behaves as a wait state, then the execution will wait in that node until the execution's `signal` method is invoked. The execution will delegate that signal to the behaviour Activity of the current node.

So the Activity's `signal`-method is invoked when the execution receives an external trigger during the wait state. With the signal method, responsibility is transferred back to the process execution. For example, when a person completes a task, the task management system calls the signal method on the execution.

A signal can optionally have a signal name and a map of parameters. Most common way on how node behaviours interprete the signal and parameters is that the signal relates to the outgoing transition that needs to be taken and that the parameters are set as variables on the execution. But those are just examples, it is up to the activity to use the signal and the parameters as it pleases.

The `getSignals`-method is optional and if a value is returned, it is the set of signals that this node accepts. The meaning and usage is analogue to how in Java reflection, it's possible to inspect all methods and method signatures of a Java class.

2.4. ExternalActivity example

Here's a first example of a simple wait state implementation:

```
public class WaitState implements ExternalActivity {

  public void execute(Execution execution) {
    execution.waitForSignal();
  }

  public void signal(Execution execution,
                     String signal,
                     Map<String, Object> parameters) {
    execution.take(signal);
  }

  public Set<SignalDefinition> getSignals(Execution execution) {
    return null;
  }
}
```

The `execute`-method calls `execution.waitForSignal()`. This call is necessary to prevent automatic propagation of the execution. By calling `execution.waitForSignal()`, the node will behave as a wait state.

`signal`-method takes the transition with the signal parameter as the transition name. So when an execution receives an external trigger, the signal name is interpreted as the name of an outgoing transition and the execution will be propagated over that transition.

The `getSignals`-method is for introspection. Since it's optional, it is not implemented in this example, by returning null. So with this implementation, tools cannot inspect the possible signals that can be given for this node behaviour. The proper implementation that would match this node's signal method is to return a list of SignalDefinition's that correspond to the names of the outgoing transitions.

Here's the same simple process that has a transition from a to b. This time, the behaviour of the two nodes will be WaitState's.

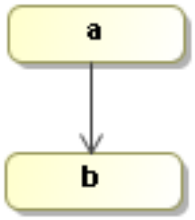

Figure 2.2. Process diagram

```
ProcessDefinition processDefinition = ProcessFactory.build()
    .node("a").initial().behaviour(new WaitState())
      .transition().to("b")
    .node("b").behaviour(new WaitState())
.done();
```

```
Execution execution = processDefinition.startExecution();
```

```
execution.signal();
```

2.5. Basic process execution

In this next example, we'll combine automatic activities and wait states. This example is a simplified version of a loan approval process. Graphically, it looks like this:

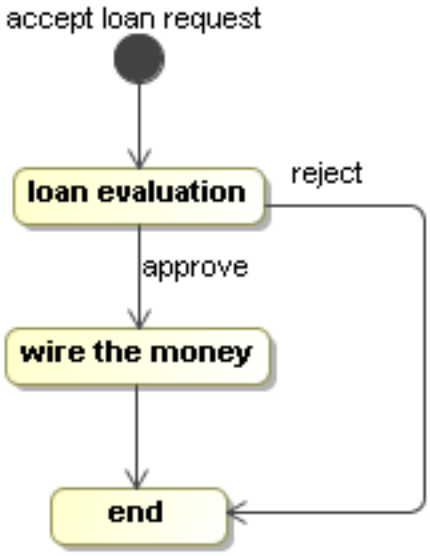

Figure 2.3. The first graph process

Building process graphs in Java code can be tedious because you have to keep track of all the references in local variables. To resolve that, the Process Virtual Machine comes with a ProcessFactory. The ProcessFactory is a kind of domain specific language (DSL) that is embedded in Java and eases the construction of process graphs. This pattern is also known as a fluent interface [http://martinfowler.com/bliki/FluentInterface.html].

```
ProcessDefinition processDefinition = ProcessFactory.build()
    .node("accept loan request").initial().behaviour(new WaitState())
      .transition().to("loan evaluation")
    .node("loan evaluation").behaviour(new WaitState())
      .transition("approve").to("wire the money")
      .transition("reject").to("end")
    .node("wire the money").behaviour(new Display("automatic payment"))
      .transition().to("end")
    .node("end").behaviour(new WaitState())
.done();
```

For more details about the ProcessFactory, see the javadocs. An alternative for the ProcessFactory would be to create an XML language and an XML parser for expressing processes. The XML parser can then instantiate the classes of package `org.jbpm.pvm.impl` directly. That approach is typically taken by process languages.

The node `wire the money` is an automatic node. The `Display` implementation uses the Java API's to just print a message to the console. But the witty reader can imagine an alternative `Activity` implementation that uses the Java API of a payment processing library to make a real automatic payment. All the other nodes are wait states.

A new execution for the process above can be started like this

```
Execution execution = processDefinition.startExecution();
```

Starting a new execution implies that the initial node is executed. Since in this case it's a wait state, the new execution will be positioned in the node 'accept loan request' when the `startExecution`-method returns.

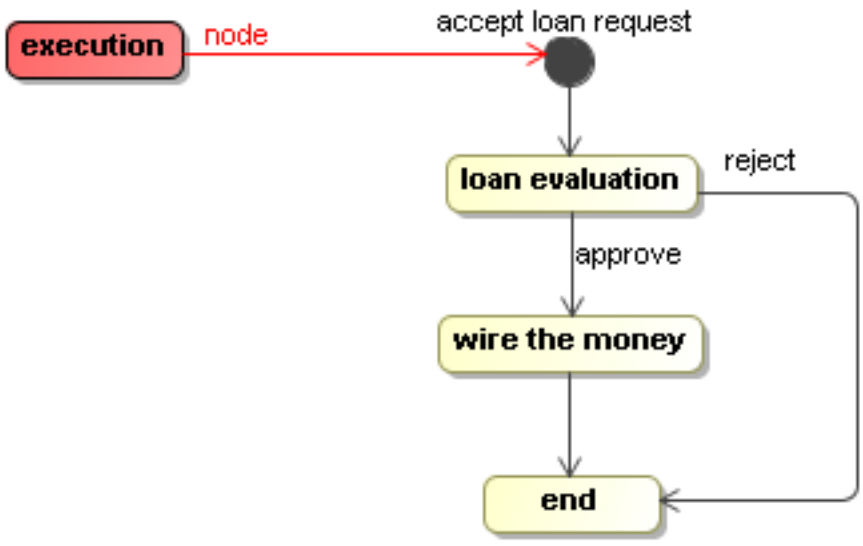

Figure 2.4. Execution positioned in 'accept loan request'

Now we can give this execution an external trigger with the `signal`- method on the execution. Invoking the signal

method will take the execution to the next wait state.

```
execution.signal();
```

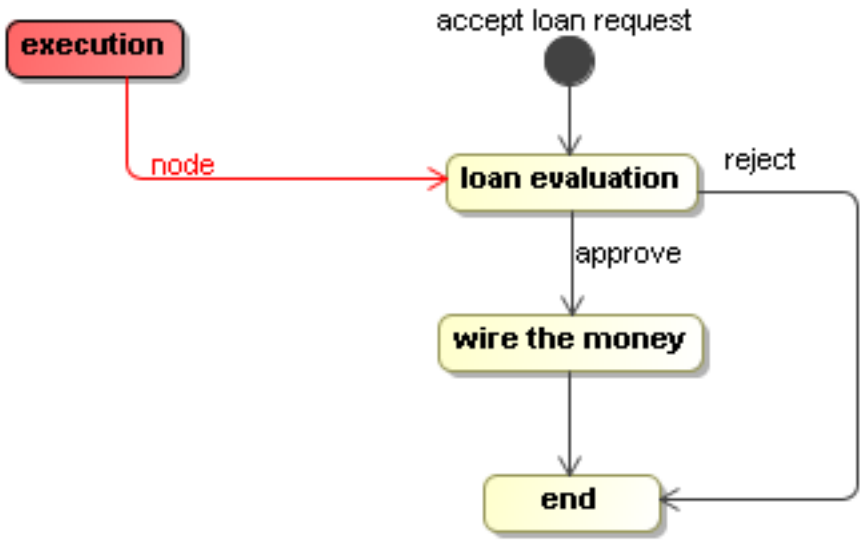

Figure 2.5. Execution positioned in 'loan evaluation'

Now, the execution is at an interesting point. There are two transitions out of the state 'loan evaluation'. One transition is called 'approve' and one transition is called 'reject'. As we explained above in the WaitState implementation, the transition taken corresponds to the signal that is given. Let's feed in the 'approve' signal like this:

```
execution.signal("approve");
```

The 'approve' signal will cause the execution to take the 'approve' transition and it will arrive in the node 'wire the money'.

In `wire the money`, the message will be printed to the console. Since, the `Display` activity didn't invoke the `execution.waitForSignal()`, nor any of the other execution propagation methods, the default behaviour will be to just proceed.

Proceeding in this case means that the default outgoing transition is taken and the execution will arrive in the `end` node, which is a wait state.

So only when the `end` wait state is reached, the `signal("approve")` returns. That is because all of the things that needed to be done between the original state and this new state could be executed by the process system. Executing till the next wait state is the default behaviour and that behaviour can be changed with

TODO: add link to async continuations

asynchronous continuations in case transactions should not include all calculations till the next wait state. For more about this, see Section 4.4.

Another signal invocation will bring it eventually in the end state.

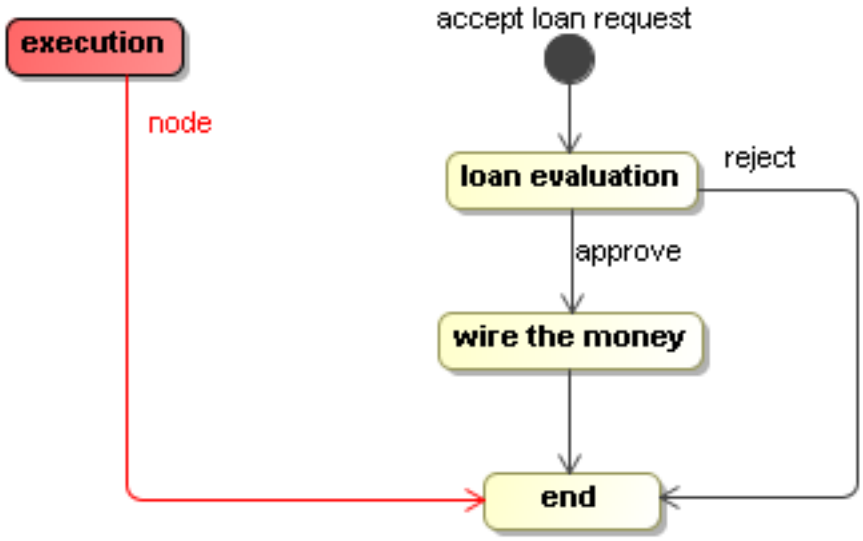

Figure 2.6. Execution positioned in 'end'

2.6. Motivation

There are basically two forms of process languages: graph based and composite process languages. First of all, this design supports both. Even graph based execution and node composition can be used in combination to implement something like UML super states.

In this design, control flow activity implementations will have to be aware of whether they are dependent on transitions (graph based) or whether they are using the composite node structure. The goal of this design is that all non-control flow activities can be implemented in the same way so that you can use them in graph based process languages as well as in composite process languages.

2.7. Events

Events are points in the process definition to which a list of activities can be subscribed as listeners. The motivation for events is to allow for developers to add programming logic to a process without changing the process diagram. This is a very valuable instrument in facilitating the collaboration between business analysts and developers. Business analysts are responsible for expressing the requirements. When they use a process graph to document those requirements, developers can take this diagram and make it executable. Events can be a very handy to insert technical details into a process (like e.g. some database insert) in which the business analyst is not interested.

Most common events are fired by the execution automatically:

- `Transition.EVENT_TRANSITION_TAKE = "transition-take"` : fired on transitions when transitions are taken.
- `Node.EVENT_NODE_ENTER = "node-enter"` : fired on the node when execution enters that node. This happens when execution takes a transition to that node, when a child node is being executed with `execution.execute(Node)` or when a transition is taken from a node outside that node to a contained node. The latter

refers to super states in state machines.

- `Node.EVENT_NODE_LEAVE = "node-leave"` : fired on the node when a transition is taken out of that node or when a child node execution is finished and the execution is propagated to the parent node.
- `ProcessDefinition.EVENT_PROCESS_START = "process-start"` : fired on a process when a new process is started.
- `ProcessDefinition.EVENT_PROCESS_END = "process-end"` : fired on a process when a new process is ended. This might include a executions that are ended with a cancelled or error state.

Events are identified by the combination of a process element and an event name. Users and process languages can also fire events programmatically with the fire method on the Execution:

```
public interface Execution extends Serializable {
  ...
  void fire(String eventName, ProcessElement eventSource);
  ...
}
```

A list of `Activity`s can be associated to an event. But activities on events can not influence the control flow of the execution since they are merely listeners to an execution wich is already in progress. This is different from activities that serve as the behaviour for nodes. Node behaviour activities are responsible for propagating the execution. So if an activity in an event invokes any of the following methods, then it will result in an exception.

- `waitForSignal()`
- `take(Transition)`
- `end(*)`
- `execute(Node)`

We'll reuse the `Display` activity from above in a simple process: two nodes connected by a transition. The Display listener will be subscribed to the transition event.

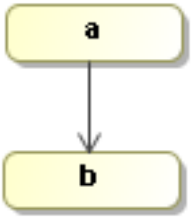

Figure 2.7. The process to which a listener activity will be associated

```
ProcessDefinition processDefinition = ProcessFactory.build()
    .node("a").initial().behaviour(new WaitState())
      .event("node-leave")
        .listener(new Display("leaving a"))
        .listener(new Display("second message while leaving a"))
      .transition().to("b")
```

```
        .listener(new Display("taking transition"))
    .node("b").behaviour(new WaitState())
      .event("node-enter")
        .listener(new Display("entering b"))
.done();
```

The first event shows how to register multiple listeners to the same event. They will be notified in the order as they are specified.

Then, on the transition, there is only one type of event. So in that case, the event type must not be specified and the listeners can be added directly on the transition.

A listeners will be called each time an execution fires the event to which the listener is subscribed. The execution will be provided in the activity interface as a parameter and can be used by listeners except for the methods that control the propagation of execution.

2.8. Event propagation

Events are by default propagated to enclosing process elements. The motivation is to allow for listeners on process definitions or composite nodes that get executed for all events that occur within that process element. For example this feature allows to register a listener on a process definition or a composite node on `node-leave` events. Such action will be executed if that node is left. And if that listener is registered on a composite node, it will also be executed for all nodes that are left within that composite node.

To show this clearly, we'll create a `DisplaySource` activity that will print the message `leaving` and the source of the event to the console.

```
public class DisplaySource implements Activity {

  public void execute(Execution execution) {
    System.out.println("leaving "+execution.getEventSource());
  }
}
```

Note that the purpose of event listeners is not to be visible, that's why the activity itself should not be displayed in the diagram. A `DisplaySource` activity will be added as a listener to the event `node-leave` on the composite node.

The next process shows how the `DisplaySource` activity is registered as a listener to to the 'node-leave' event on the `composite` node:

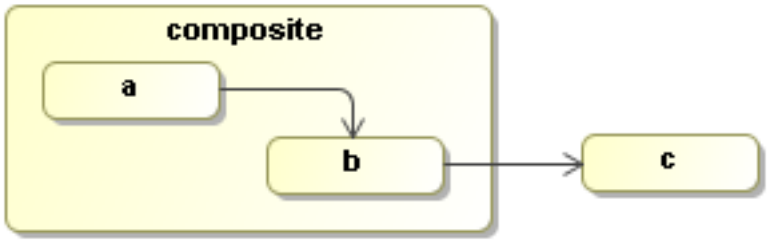

Figure 2.8. A process with an invisible activity on a node-leave event on a composite node.

```
ProcessDefinition processDefinition = ProcessFactory.build("propagate")
    .compositeNode("composite")
      .event(Node.EVENT_NODE_LEAVE)
        .listener(new DisplaySource())
      .node("a").initial().behaviour(new WaitState())
        .transition().to("b")
      .node("b").behaviour(new WaitState())
        .transition().to("c")
    .compositeEnd()
    .node("c").behaviour(new WaitState())
.done();
```

Next we'll start an execution.

```
Execution execution = processDefinition.startExecution();
```

After starting a new execution, the execution will be in node `a` as that is the initial node. No nodes have been left so no message is logged. Next a signal will be given to the execution, causing it to take the transition from `a` to `b`.

```
execution.signal();
```

When the signal method returns, the execution will have taken the transition and the node-leave event will be fired on node a. That event will be propagated to the composite node and to the process definition. Since our propagation logger is placed on node composite it will receive the event and print the following message:

```
leaving node(a)
```

Another

```
execution.signal();
```

will take the transition from b to c. That will fire two node-leave events. One on node b and one on node composite. So the following lines will be appended to the console output:

```
leaving node(b)
leaving node(composite)
```

Event propagation is build on the hierarchical composition structure of the process definition. The top level element is always the process definition. The process definition contains a list of nodes. Each node can be a leaf node or it can be a composite node, which means that it contains a list of nested nodes. Nested nodes can be used for e.g. super states or composite activities in nested process languages like BPEL.

So the even model also works similarly for composite nodes as it did for the process definition above. Suppose that 'Phase one' models a super state as in state machines. Then event propagation allows to subscribe to all events within that super state. The idea is that the hierarchical composition corresponds to diagram representation. If an element 'e' is drawn inside another element 'p', then p is the parent of e. A process definition has a set of top level nodes. Every node can have a set of nested nodes. The parent of a transition is considered as the first common parent for it's source and destination.

If an event listener is not interested in propagated events, propagation can be disabled with `propagationDisabled()`. The next process is the same process as above except that propagated events will be disabled on the event listener. The graph diagram remains the same.

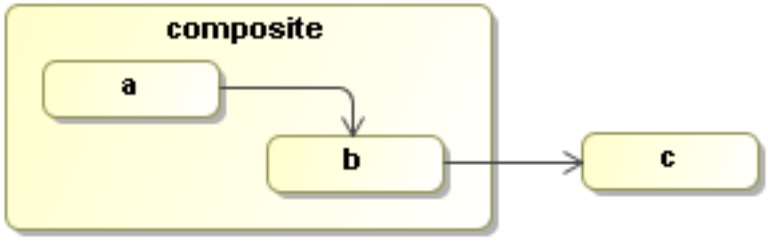

Figure 2.9. A process with a listener to 'node-leave' events with propagation disabled.

Building the process with the process factory:

```
ProcessDefinition processDefinition = ProcessFactory.build("propagate")
    .compositeNode("composite")
      .event(Node.EVENT_NODE_LEAVE)
        .listener(new DisplaySource())
        .propagationDisabled()
      .node("a").initial().behaviour(new WaitState())
        .transition().to("b")
      .node("b").behaviour(new WaitState())
        .transition().to("c")
    .nodesEnd()
    .node("c").behaviour(new WaitState())
.done();
```

So when the first signal is given for this process, again the node-leave event will be fired on node a, but now the listener on the composite node will not be executed cause propagated events have been disabled. Disabling propagation is a property on the listener and doesn't influence the other listeners. The event will always be fired and propagated over the whole parent hierarchy.

```
Execution execution = processDefinition.startExecution();
execution.signal();
```

Next, the second signal will take the transition from b to c.

```
execution.signal()
```

Again two node-leave events are fired just like above on nodes b and composite respectively. The first event is the node-leave event on node b. That will be propagated to the composite node. So the listener will not be executed for this event cause it has propagation disabled. But the listener will be executed for the node-leave event on the composite node. That is not propagated, but fired directly on the composite node. So the listener will now be executed only once for the composite node as shown in the following console output:

```
leaving node(composite)
```

2.9. Process structure

Above we already touched briefly on the two main process constructs: Nodes, transitions and node composition. This section will elaborate on all the basic combination possibilities.

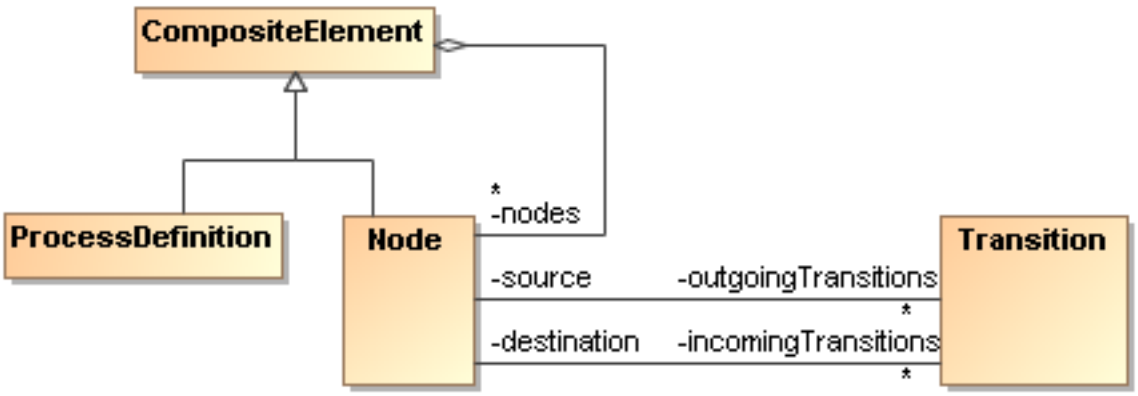

Figure 2.10. UML class diagram of the basic process structure

Figure 2.11. Any two nodes can be connected with a transition.

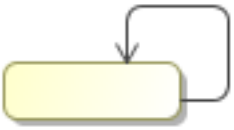

Figure 2.12. A self transition.

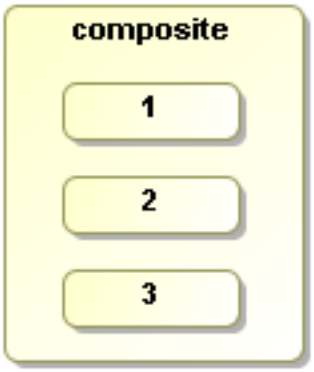

Figure 2.13. Composite node is a list of nested nodes.

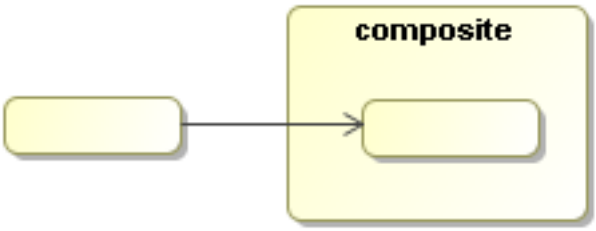

Figure 2.14. Transition to a node inside a composite.

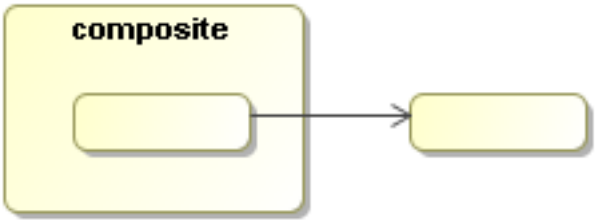

Figure 2.15. Transition from a node inside a composite to a node outside the composite.

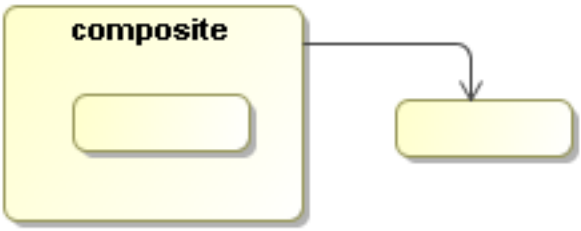

Figure 2.16. Transition of composite nodes are inherited. The node inside can take the transition of the composite node.

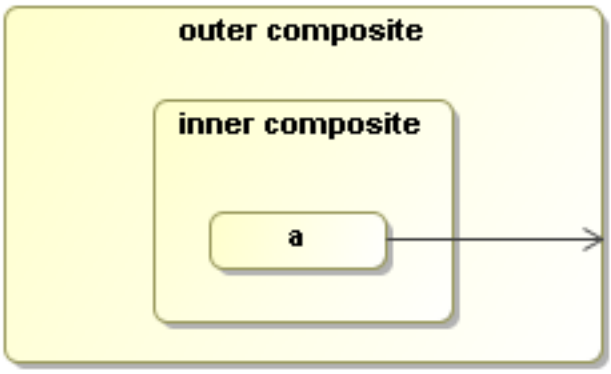

Figure 2.17. Transition from a node to an outer composite.

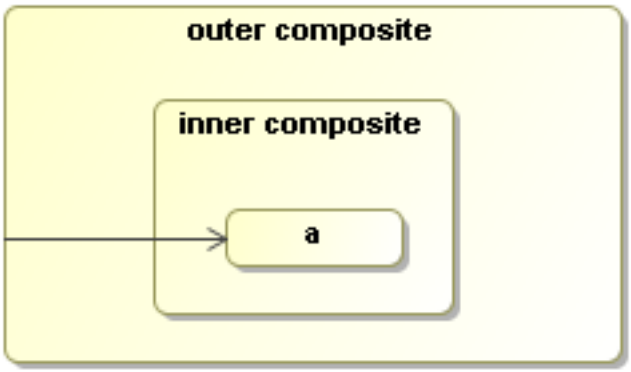

Figure 2.18. Transition from a composite node to an inner composed node.

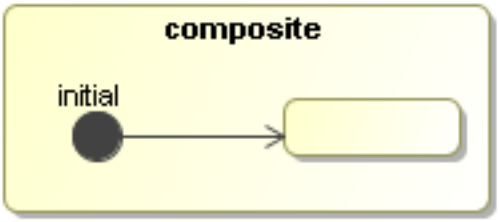

Figure 2.19. An initial node inside a composite node.

3

Examples

3.1. Graph based control flow activities

3.1.1. Automatic decision

This example shows how to implement automatic conditional branching. This is mostly called a decision or an or-split. It selects one path of execution from many alternatives. A decision node should have multiple outgoing transitions.

In a decision, information is collected from somewhere. Usually that is the process variables. But it can also collect information from a database, a file, any other form of input or a combination of these. In this example, a variable `creditRate` is used. It contains an integer. The higher the integer, the better the credit rating. Let's look at the example implementation:

Then based on the obtained information, in our case that is the `creditRate`, an outgoing transition has to be selected. In the example, transition `good` will be selected when the creditRate is above 5, transition `bad` will be selected when creditRate is below -5 and otherwise transition `average` will be selected.

Once the selection is done, the transition is taken with `execution.take(String)` or the `execution.take(Transition)` method.

```
public class AutomaticCreditRating implements Activity {
    public void execute(Execution execution) {
      int creditRate = (Integer) execution.getVariable("creditRate");

      if (creditRate > 5) {
        execution.take("good");

      } else if (creditRate < -5) {
        execution.take("bad");

      } else {
        execution.take("average");
      }
    }
  }
```

We'll demonstrate the `AutomaticCreditRating` in the following process:

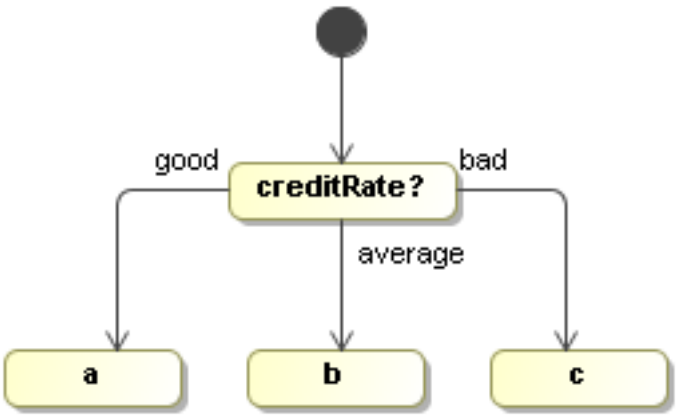

Figure 3.1. The decision process

```
ProcessDefinition processDefinition = ProcessFactory.build()
    .node("initial").initial().behaviour(new WaitState())
      .transition().to("creditRate?")
    .node("creditRate?").behaviour(new AutomaticCreditRating())
      .transition("good").to("a")
      .transition("average").to("b")
      .transition("bad").to("c")
    .node("a").behaviour(new WaitState())
    .node("b").behaviour(new WaitState())
    .node("c").behaviour(new WaitState())
.done();
```

Executing this process goes like this:

```
Execution execution = processDefinition.startExecution();
```

`startExecution()` will bring the execution into the `initial` node. That's a wait state so the execution will point to that node when the `startExecution()` returns.

Then we have a chance to set the `creditRate` to a specific value like e.g. 13.

```
execution.setVariable("creditRate", 13);
```

Next, we provide a signal so that the execution takes the default transition to the `creditRate?` node. Since process variable `creditRate` is set to 13, the `AutomaticCreditRating` activity will take transition `good` to node `a`. Node `a` is a wait state so them the invocation of `signal` will return.

Similarly, a decision can be implemented making use of the transition's guard condition. For each outgoing transition, the guard condition expression can be evaluated. The first transition for which its guard condition evaluates to true is taken.

This example showed automatic conditional branching. Meaning that all information is available when the execution arrives in the decision node, even if it may have to be collected from different sources. In the next example, we show how a decision is implemented for which an external entity needs to supply the information, which results into a wait state.

3.1.2. External decision

This example shows an activity that again selects one path of execution out of many alternatives. But this time, the information on which the decision is based is not yet available when the execution arrives at the decision. In other words, the execution will have to wait in the decision until the information is provided from externally.

```
public class ExternalSelection implements ExternalActivity {

  public void execute(Execution execution) {
    execution.waitForSignal();
  }

  public void signal(Execution execution, String signalName, Map<String, Object> parameters) throws Excep
    execution.take(signalName);
  }

  public Set<SignalDefinition> getSignals(Execution execution) throws Exception {
    return null;
  }
}
```

The diagram for this external decision will be the same as for the automatic decision:

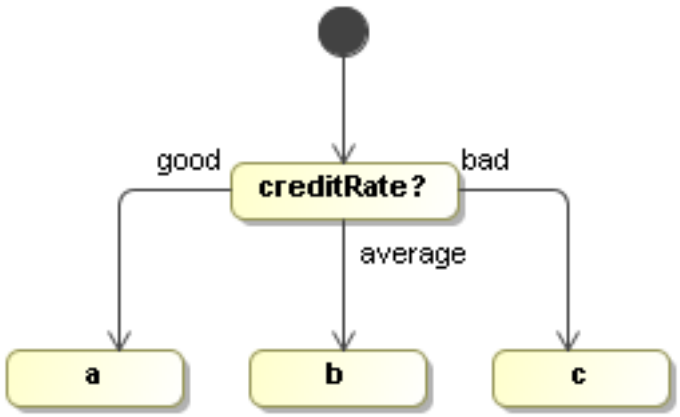

Figure 3.2. A decision

```
ProcessDefinition processDefinition = ProcessFactory.build()
    .node("initial").initial().behaviour(new WaitState())
      .transition().to("creditRate?")
    .node("creditRate?").behaviour(new ExternalSelection())
      .transition("good").to("a")
      .transition("average").to("b")
      .transition("bad").to("c")
    .node("a").behaviour(new WaitState())
    .node("b").behaviour(new WaitState())
    .node("c").behaviour(new WaitState())
.done();
```

The execution starts the same as in the automatic example. After starting a new execution, it will be pointing to the initial wait state.

```
Execution execution = processDefinition.startExecution();
```

But the next signal will cause the execution to take the default transition out of the `initial` node and arrive in the `creditRate?` node. Then the `ExternalSelection` is executed, which will result into a wait state. So when the invocation of `signal()` returns, the execution will be pointing to the `creditRate?` node and it expects an external trigger.

Next we'll give an external trigger with `good` as the signalName. So supplying the external trigger is done together with feeding the information needed by the decision.

```
execution.signal("good");
```

That external trigger will be translated by the `ExternalSelection` activity into taking the transition with name `good`. That way the execution will have arrived in node a when `signal("good")` returns.

Note that both parameters `signalName` and `parameters` can be used by external activities as they want. In the example here, we used the signalName to specify the result. But another variation might expect an integer value under the `creditRate` key of the parameters.

But leveraging the execution API like that is not done very often in practice. The reason is that for most external functions, typically activity instances are created. Think about `Task` as an instance of a `TaskActivity` (see later) or analogue, a `ServiceInvocation` could be imagined as an instance of a `ServiceInvocationActivity`. In those cases, those activity instances make the link between the external activity and the execution. And these instances also can make sure that an execution is not signalled inappropriately. Inappropriate signalling could happen when for instance a service response message would arrive twice. If in such a scenario, the message receiver would just signal the execution, it would not notice that the second time, the execution is not positioned in the service invocation node any more.

3.2. Composite based control flow activities

3.2.1. Composite sequence

Block structured languages like BPEL are completely based on composite nodes. Such languages don't have transitions. The composite node structure of the Process Virtual Machine allows to build a process with a structure that exactly matches the block structured languages. There is no need for a conversion to a transition based model. We have already discussed some examples of composite nodes. The following example will show howw to implement a sequence, one of the most common composite node types.

A sequence has a list of nested activities that need to be executed in sequence.

This is how a sequence can be implemented:

```
public class Sequence implements ExternalActivity {

  public void execute(Execution execution) {
    List<Node> nodes = execution.getNode().getNodes();
    execution.execute(nodes.get(0));
  }

  public void signal(Execution execution, String signal, Map<String, Object> parameters) {
    Node previous = execution.getPreviousNode();
    List<Node> nodes = execution.getNode().getNodes();
    int previousIndex = nodes.indexOf(previous);
```

```
    int nextIndex = previousIndex+1;
    if (nextIndex < nodes.size()) {
      Node next = nodes.get(nextIndex);
      execution.execute(next);
    } else {
      execution.proceed();
    }
  }

  public Set<SignalDefinition> getSignals(Execution execution) {
    return null;
  }
}
```

When an execution arrives in this sequence, the **execute** method will execute the first node in the list of child nodes (aka composite nodes or nested nodes). The sequence assumes that the child node's behaviour doesn't have outgoing transitions and will end with an `execution.proceed()`. That proceed will cause the execution to be propagated back to the parent (the sequence) with a signal.

The **signal** method will look up the previous node from the execution, determine its index in the list of child nodes and increments it. If there is a next node in the list it is executed. If the previous node was the last one in the list, the proceed is called, which will propagate the execution to the parent of the sequence in case there are no outgoing transitions.

To optimize persistence of executions, the previous node of an execution is normally not maintained and will be to null. If a node requires the previous node or the previous transition like in this Sequence, the property `isPreviousNeeded` must be set on the node.

Let's look at how that translates to a process and an execution:

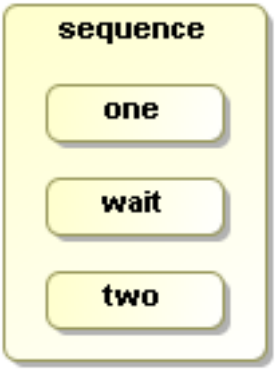

Figure 3.3. A sequence.

```
ProcessDefinition processDefinition = ProcessFactory.build("sequence")
    .compositeNode("sequence").initial().behaviour(new Sequence())
      .needsPrevious()
      .node("one").behaviour(new Display("one"))
      .node("wait").behaviour(new WaitState())
      .node("two").behaviour(new Display("two"))
    .compositeEnd()
.done();
```

The three numbered nodes will now be executed in sequence. Nodes `1` and `2` are automatic `Display` activities,

while node `wait` is a wait state.

```
Execution execution = processDefinition.startExecution();
```

The `startExecution` will execute the `Sequence` activity. The `execute` method of the sequence will immediately execute node 1, which will print message `one` on the console. Then the execution is automatically proceeded back to the sequence. The sequence will have access to the previous node. It will look up the index and execute the next. That will bring the execution to node `wait`, which is a wait state. At that point, the `startExecution()` will return. A new external trigger is needed to complete the wait state.

```
execution.signal();
```

That signal will delegate to the `WaitState`'s signal method. That method is empty so the execution will proceed in a default way. Since there are no outgoing transitions, the execution will be propagated back to the sequence node, which will be signalled. Then node `2` is executed. When the execution comes back into the sequence it will detect that the previously executed node was the last child node, therefore, no propagation method will be invoked, causing the default proceed to end the execution. The console will show:

```
one
two
```

3.2.2. Composite decision

In a composite model, the node behaviour can use the `execution.execute(Node)` method to execute one of the child nodes.

Figure 3.4. A decision based on node composition

```
ProcessDefinition processDefinition = ProcessFactory.build()
    .compositeNode("creditRate?").initial().behaviour(new CompositeCreditRating())
      .node("good").behaviour(new ExternalSelection())
      .node("average").behaviour(new ExternalSelection())
      .node("bad").behaviour(new ExternalSelection())
    .compositeEnd()
.done();
```

The `CompositeCreditRating` is an automatic decision, implemented like this:

```
public class CompositeCreditRating implements Activity {

  public void execute(Execution execution) {
    int creditRate = (Integer) execution.getVariable("creditRate");

    if (creditRate > 5) {
      execution.execute("good");

    } else if (creditRate < -5) {
      execution.execute("bad");

    } else {
      execution.execute("average");
    }
  }
}
```

So when we start a new execution with

```
Map<String, Object> variables = new HashMap<String, Object>();
variables.put("creditRate", 13);
Execution execution = processDefinition.startExecution(variables);
```

The execution will execute the `CompositeCreditRating`. The `CompositeCreditRating` will execute node `good` cause the process variable `creditRate` is 13. When the `startExecution()` returns, the execution will be positioned in the good state. The other scenarios are very similar.

3.3. Human tasks

This section will demonstrate how support for human tasks can be build on top of the Process Virtual Machine.

As we indicated in Section 4.4, for each step in the process the most important characteristic is whether responsibility for an activity lies within the process system or outside. In case of a human task, it should be clear that the responsibility is outside of the process system. This means that for the process, a human task is a wait state. The execution will have to wait until the person provides the external trigger that the task is completed or submitted.

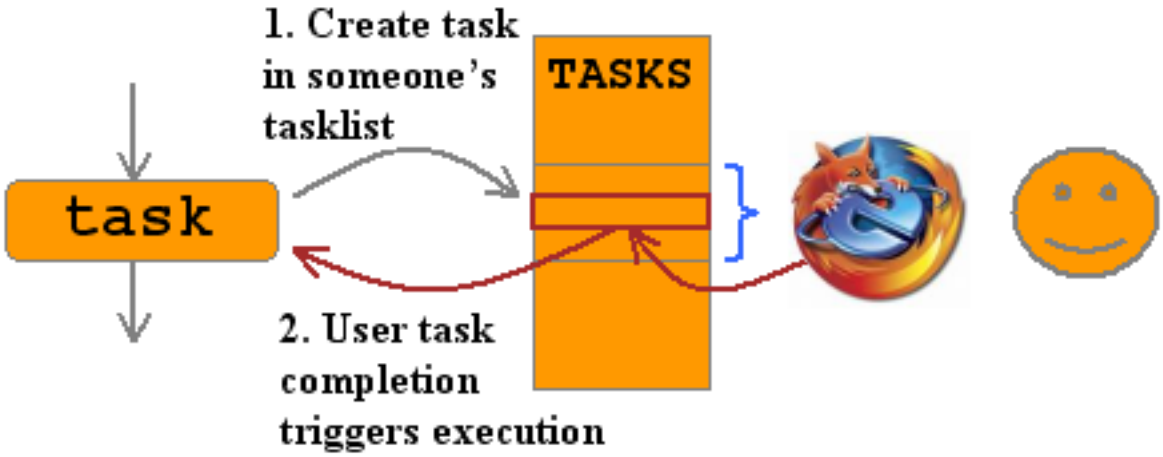

Figure 3.5. Overview of the link between processes and tasks.

In the picture above, the typical link between process execution and tasks is represented. When an execution arrives in a task node, a task is created in a task component. Typically such a task will end up in a task table somewhere in the task component's database. Then users can look at their task lists. A task list is then a filter on the complete task list based on the task's assigned user column. When the user completes the task, the execution is signalled and typically leaves the node in the process.

A task management component keeps track of tasks for people. To integrate human tasks into a process, we need an API to create new tasks and to get notifications of task completions. The following example might have only a rudimentary integration between between process execution and the task management component, but the goal is to show the interactions as clearly as possible. Real process languages like jPDL have a much better integration between process execution and tasks, resulting in more complexity.

For this example we'll first define a simplest task component with classes `Task` and `TaskComponent`:

```
public class Task {
  public String userId;
  public String taskName;
  public Execution execution;

  public Task(String userId, String taskName, Execution execution) {
    this.userId = userId;
    this.taskName = taskName;
    this.execution = execution;
  }

  public void complete() {
    execution.signal();
  }
}
```

This task has public fields to avoid the getters and setters. The taskName property is the short description of the task. The userId is a reference to the user that is assigned to this task. And the execution is a reference to the execution to which this task relates. When a task completes it signals the execution.

The next task component manages a set of tasks.

```
public class TaskComponent {

  static List<Task> tasks = new ArrayList<Task>();

  public static void createTask(String taskName, Execution execution) {
    String userId = assign(taskName, execution);
    tasks.add(new Task(userId, taskName, execution));
  }

  private static String assign(String taskName, Execution execution) {
    return "johndoe";
  }

  public static List<Task> getTaskList(String userId) {
    List<Task> taskList = new ArrayList<Task>();
    for (Task task : tasks) {
      if (task.userId.equals(userId)) {
        taskList.add(task);
      }
    }
    return taskList;
  }
}
```

To keep this example short, this task component is to be accessed through static methods. The assigning tasks is done hard coded to "johndoe". Tasks can be created and tasklists can be extracted by userId. Next we can look at the node behaviour implementation of a TaskActivity.

```
public class TaskActivity implements ExternalActivity {

  public void execute(Execution execution) {
    // let's use the node name as the task id
    String taskName = execution.getNode().getName();
    TaskComponent.createTask(taskName, execution);
  }

  public void signal(Execution execution, String signal, Map<String, Object> parameters) {
    execution.takeDefaultTransition();
  }

  public Set<SignalDefinition> getSignals(Execution execution) {
    return null;
  }
}
```

The task node works as follows. When an execution arrives in a task node, the execute method of the TaskActivity is invoked. The execute method will then take the node name and use it as the task name. Alternatively, 'taskName' could be a configuration property on the TaskActivity class. The task name is then used to create a task in the task component. Once the task is created, the execution is not propagated which means that the execution will wait in this node till a signal comes in.

When the task is completed with the Task.complete() method, it will signal the execution. The TaskActivity's signal implementation will take the default transition.

This is how a process can be build with a task node:

```
ProcessDefinition processDefinition = ProcessFactory.build("task")
    .node("initial").initial().behaviour(new AutomaticActivity())
      .transition().to("shred evidence")
    .node("shred evidence").behaviour(new TaskActivity())
      .transition().to("next")
    .node("next").behaviour(new WaitState())
.done();
```

When a new execution is started, the initial node is an automatic activity. So it will immediately propagate to the task node the task will be created and the execution will stop in the 'shred evidence' node.

```
Execution execution = processDefinition.startExecution();

assertEquals("shred evidence", execution.getNode().getName());

Task task = TaskComponent.getTaskList("johndoe").get(0);
```

Next, time can elapse until the human user is ready to complete the task. In other words, the thread of control is now with 'johndoe'. When John completes his task e.g. through a web UI, then this should result into an invocation of the complete method on the task.

```
task.complete();

assertEquals("next", execution.getNode().getName());
```

The invocation of the complete method cause the execution to take the default transition to the 'next' node.

4

Advanced graph execution

4.1. Loops

Loops can be based on transitions or on node composition. Loops can contain wait states.

To support high numbers of automatic loop executions, the Process Virtual Machine tranformed the propagation of execution from tail recursion to a while loop. This means that all the methods in the `Execution` class that propagate the execution like `take` or `execute` will not be executed when you call them. Instead, the method invocations will be appended to a list. The first invocation of such a method will start a loop that will execute all invocations till that list is empty. These invocations are called atomic operations.

4.2. Sub processes

TODO: sub processes

4.3. Default proceed behaviour

When an `Activity` is used as node behaviour, it can explicitely propagate the execution with following methods:

- `waitForSignal()`
- `take(Transition)`
- `end(*)`
- `execute(Node)`
- `createExecution(*)`

When `Activity` implementations used for node behviour don't call any of the following execution propagation methods, then, after the activity is executed, the execution will just proceed.

By default proceeding will perform the first action that applies in the following list:

- If the current node has a default outgoing transition, take it.
- If the current node has a parent node, move back to the parent node.

- Otherwise, end this execution.

Process languages can overwrite the default proceed behaviour by overriding the `proceed` method in `ExecutionImpl`.

4.4. Execution and threads

This section explains how the Process Virtual Machine boroughs the thread from the client to bring an execution from one wait state to another.

When a client invokes a method (like e.g. the signal method) on an execution, by default, the Process Virtual Machine will use that thread to progress the execution until it reached a wait state. Once the next wait state has been reached, the method returns and the client gets the thread back. This is the default way for the Process Virtual Machine to operate. Two more levels of asynchonous execution complement this default behaviour: Asynchronous continuations and the asynchronous command service.

The next process will show the basics concretely. It has three wait states and four automatic nodes.

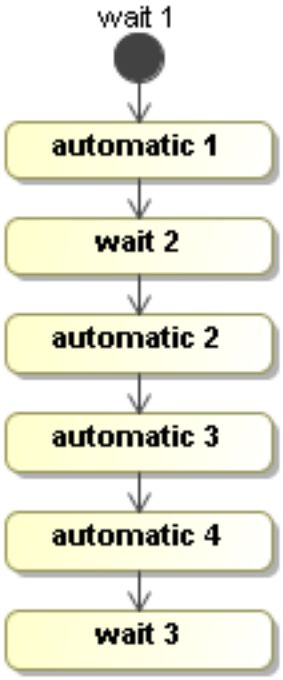

Figure 4.1. Process with many sequential automatic activities.

Here's how to build the process:

```
ProcessDefinition processDefinition = ProcessFactory.build("automatic")
    .node("wait 1").initial().behaviour(new WaitState())
      .transition().to("automatic 1")
    .node("automatic 1").behaviour(new Display("one"))
      .transition().to("wait 2")
    .node("wait 2").behaviour(new WaitState())
      .transition().to("automatic 2")
```

```
    .node("automatic 2").behaviour(new Display("two"))
      .transition().to("automatic 3")
    .node("automatic 3").behaviour(new Display("three"))
      .transition().to("automatic 4")
    .node("automatic 4").behaviour(new Display("four"))
      .transition().to("wait 3")
    .node("wait 3").behaviour(new WaitState())
.done();
```

Let's walk you through one execution of this process.

```
Execution execution = processDefinition.startExecution();
```

Starting a new execution means that the initial node is executed. So if an automatic activity would be configured as the behaviour in the initial node, the process will start executing immediatly in the startExecution. In this case however, the initial node is a wait state. So the startExecution method returns immediately and the execution will be positioned in the initial node 'wait 1'.

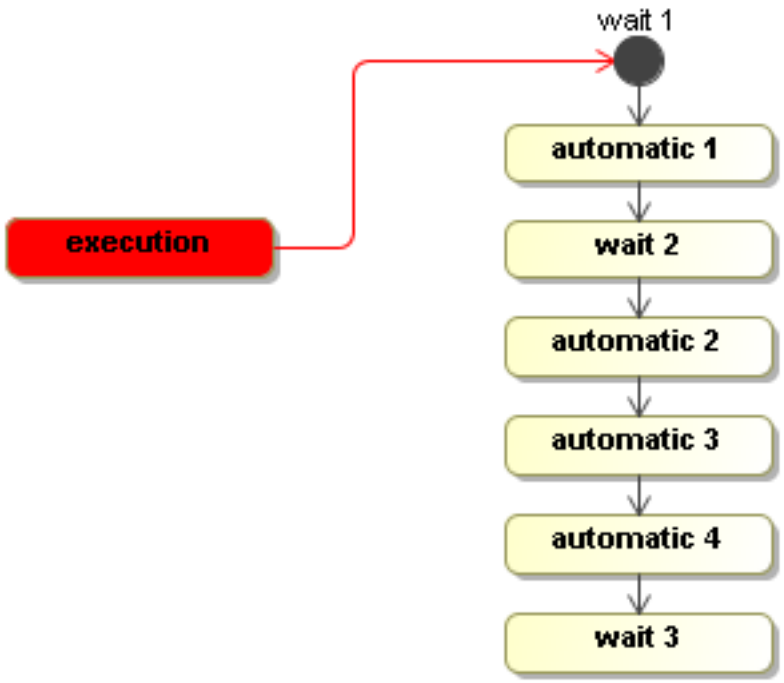

Figure 4.2. A new execution will be positioned in 'wait 1'.

Then an external trigger is given with the signal method.

```
execution.signal();
```

As explained above when introducing the WaitState, that signal will cause the default transition to be taken. The transition will move the execution to node `automatic 1` and execute it. The execute method of the `Display` activity in `automatic 1` print a line to the console and it will **not** call `execution.waitForSignal()`. Therefore, the execution will proceed by taking the default transition out of `automatic 1`. The signal method is still blocking cause this action and the transitions are taken by that same thread. Then the execution arrives in `wait 2` and executes the `WaitState` activity. That method will invoke the `execution.waitForSignal()`, which will cause the signal method

to return. That is when the thread is given back to the client that invoked the signal method.

So when the signal method returns, the execution is positioned in `wait 2`.

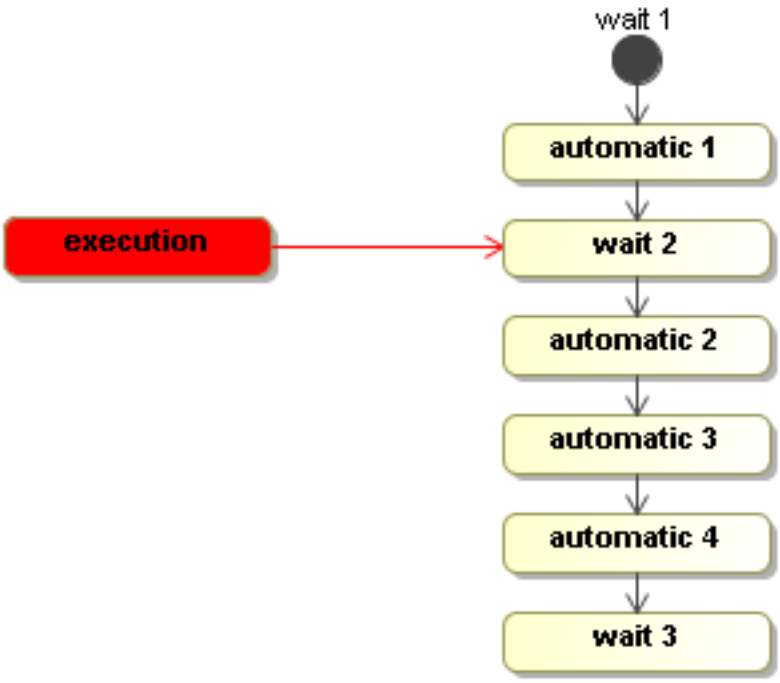

Figure 4.3. One signal brought the execution from 'initial' to 'wait 2'.

Then the execution is now waiting for an external trigger just as an object (more precisely an object graph) in memory until the next external trigger is given with the signal method.

```
execution.signal();
```

This second invocation of signal will take the execution similarly all the way to `wait 3` before it returns.

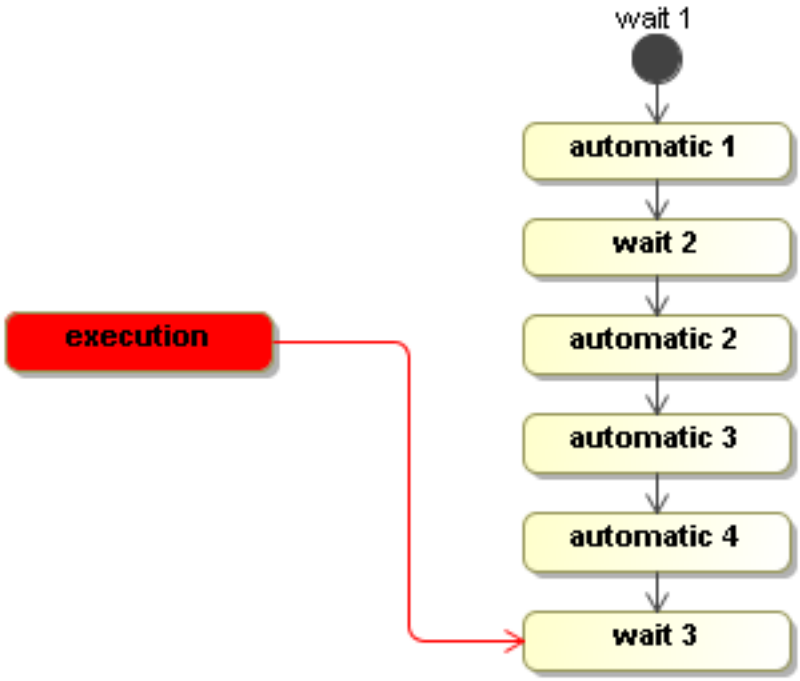

Figure 4.4. The second signal brought the execution all the way to 'wait 3'.

To make executable processes, developers need to know exactly what the automatic activities, what the wait states are and which threads will be allocated to the process execution. For business analysts that draw the analysis process, things are a bit simpler. For the activities they draw, they usually know whether it's a human or a system that is responsible. But they typically don't not how this translates to threads and transactions.

So for the developer, the first job is to analyse what needs to be executed within the thread of control of the process and what is outside. Looking for the external triggers can be a good start to find the wait states in a process, just like verbs and nouns can be the rule of thumb in building UML class diagrams.

4.5. Process concurrency

To model process concurrency, there is a parent-child tree structure on the execution. The idea is that the main path of execution is the root of that tree. This implies that on the level of the Process Virtual Machine, there is no differentiation between complete process instances and paths of execution within a process instance. One of the main motivations for this design is that the API actually is not made more complex then necessary for simple processes with only one single path of execution.

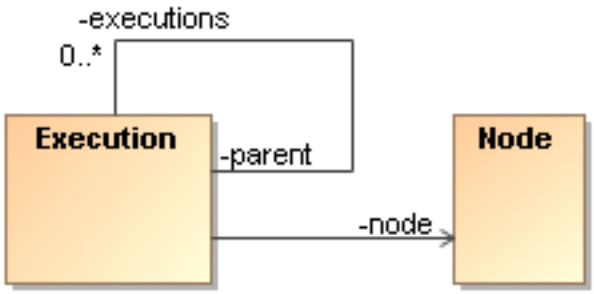

Figure 4.5. UML class diagram of the basic execution structure

To extablish multiple concurrent paths of execution, child executions can be created. Only leaf executions can be active. Non-leave executions should be inactive. This tree structure of executions doesn't enforce a particular type of concurrency or join behaviour. It's up to the forks or and-splits and to the joins or and-merges to use the execution tree structure in any way they want to define the wanted concurrency behaviour. Here you see an example of concurrent executions.

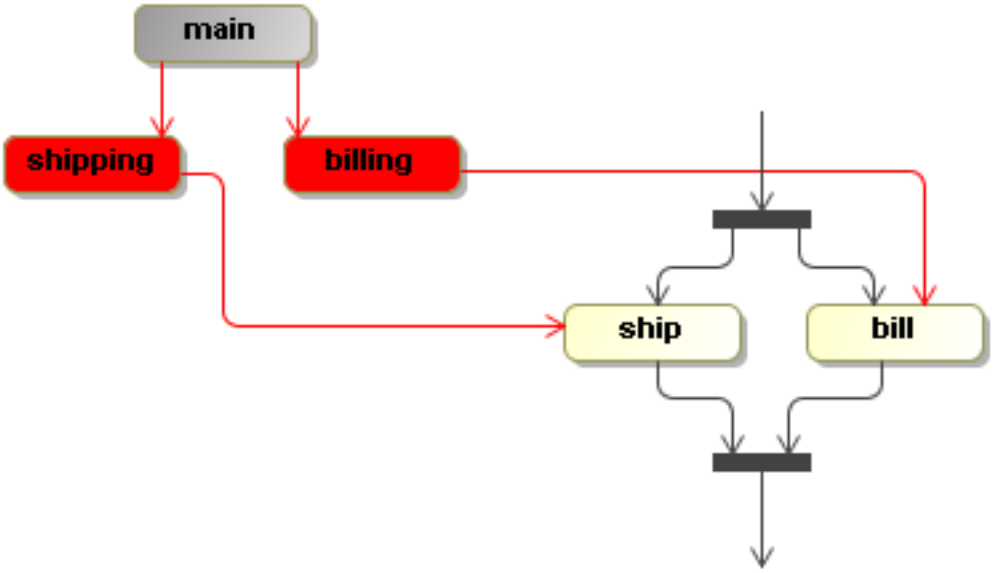

Figure 4.6. Concurrent paths of execution

There is a billing and a shipping path of execution. In this case, the flat bar nodes represent nodes that fork and join. The execution shows a three executions. The main path of execution is inactive (represented as gray) and the billing and shipping paths of execution are active and point to the node `bill` and `ship` respectively.

It's up to the node behaviour implementations how they want to use this execution structure. Suppose that multiple tasks have to be completed before the execution is to proceed. The node behaviour can spawn a series of child executions for this. Or alternatively, the task component could support task groups that are associated to one single execution. In that case, the task component becomes responsible for synchronizing the tasks, thereby moving this responsibility outside the scope of the execution tree structure.

4.6. Exception handlers

In all the code that is associated to a process like Activity's, Actions and Conditions, it's possible to include try-catch blocks in the method implementations to handle exceptions. But in order to build more reusable building blocks for both the delegation classes and the exception handling logic, exception handlers are added to the core process model.

An exception handler can be associated to any process element. When an exception occurs in a delegation class, a matching exception handler will be searched for. If such an exception handler is found, it will get a chance to handle the exception.

If an exception handler completes without problems, then the exception is considered handled and the execution resumes right after the delegation code that was called. For example, a transition has three actions and the second action throws an exception that is handled by an exception handler, then

Writing automatic activities that are exception handler aware is easy. The default is to proceed anyway. No method needs to be called on the execution. So if an automatic activity throws an exception that is handled by an exception handler, the execution will just proceed after that activity. It becomes a big more difficult for control flow activities. They might have to include try-finally blocks to invoke the proper methods on the execution before an exception handler gets a chance to handle the exception. For example, if an activity is a wait state and an exception occurs, then there is a risk that the thread jumps over the invocation of `execution.waitForSignal()`, causing the execution to proceed after the activity.

TODO: exceptionhandler.isRethrowMasked

TODO: transactional exception handlers

TODO: we never catch errors

4.7. Process modifications

TODO: process modifications

4.8. Locking and execution state

The state of an execution is either active or locked. An active execution is either executing or waiting for an external trigger. If an execution is not in STATE_ACTIVE, then it is locked. A locked execution is read only.

When a new execution is created, it is in STATE_ACTIVE. To change the state to a locked state, use lock(String). Some STATE_* constants are provided that represent the most commonly used locked states. But the state '...' in the picture indicates that any string can be provided as the state in the lock method.

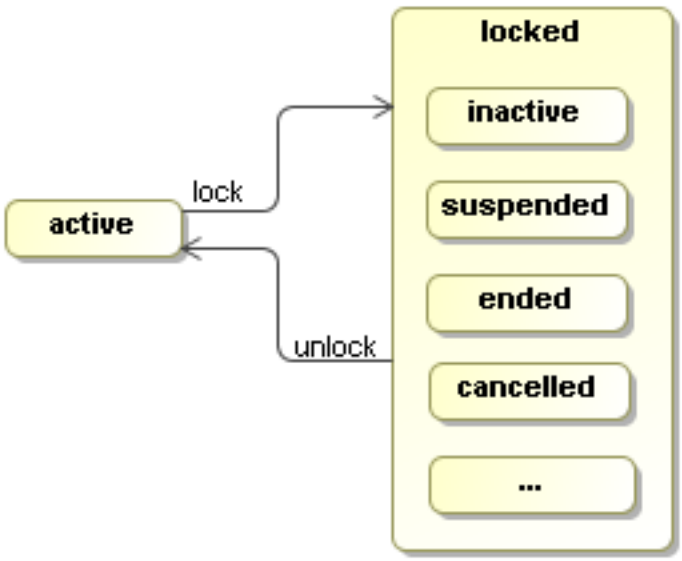

Figure 4.7. States of an execution

If an execution is locked, methods that change the execution will throw a PvmException and the message will reference the actual locking state. Firing events, updating variables, updating priority and adding comments are not considered to change an execution. Also creation and removal of child executions are unchecked, which means that those methods can be invoked by external API clients and node behaviour methods, even while the execution is in a locked state.

Make sure that comparisons between getState() and the STATE_* constants are done with .equals and not with '==' because if executions are loaded from persistent storage, a new string is created instead of the constants.

An execution implementation will be locked:

- When it is ended
- When it is suspended
- During asynchronous continuations

Furthermore, locking can be used by Activity implementations to make executions read only during wait states hen responsibility for the execution is transferred to an external entity such as:

- A human task
- A service invocation
- A wait state that ends when a scanner detects that a file appears

In these situations the strategy is that the external entity should get full control over the execution because it wants to control what is allowed and what not. To get that control, they lock the execution so that all interactions have to go through the external entity.

One of the main reasons to create external entities is that they can live on after the execution has already proceeded. For example, in case of a service invocation, a timer could cause the execution to take the timeout transition. When the response arrives after the timeout, the service invocation entity should make sure it doesn't signal the execution. So the service invocation can be seen as a node instance (aka activity instance) and is unique for every execution of the node.

External entities themselves are responsible for managing the execution lock. If the timers and client applications are consequent in addressing the external entities instead of the execution directly, then locking is in theory unnecessary. It's up to the node behaviour implementations whether they want to take the overhead of locking and unlocking.

5

Delegation classes

5.1. What are delegation classes

Delegation classes are the classes that implement `Activity` or `Condition`. From the Process Virtual Machine's perspective, these are external classes that provide programming logic that is inserted into the PVM's graph execution. Delegation classes can be provided by the process languages as well as by the end users.

5.2. Configuration of delegation classes

Delegation classes can be made configurable. Member fields can contain configuration parameters so that a delegation class can be configured differently each time it is used. For example, in the `Display` activity, the message that is to be printed to the console is a configuration parameter.

Delegation classes should be stateless. This means that executing the interface methods should not change values of the member fields. Changing member field values of delegation classes during execution methods is actually changing the process while it's executing. That is not threadsafe and usually leads to unexpected results. As an exception, getters and setters might be made available to inject the configuration cause they are used before the delegation object is actually used in the process execution.

5.3. Object references

TODO

5.4. Design time versus runtime

TODO: the node behaviour allows for design time as well as runtime behaviour.

5.5. UserCodeInterceptor

TODO: UserCodeInterceptor

5.6. Member field configurations versus properties

TODO: document field configurations versus properties

6

Variables

7

History

7.1. Process logs

7.2. Business Intelligence (BI)

7.3. Business Activity Monitoring (BAM)

8

Environment

8.1. Introduction

The environment component together with the wire context is a kind of Inversion of Control (IoC) container. It reads configuration information that describes how objects should be instantiated, configured and wired together.

The environment is used to retrieve resources and services needed by `Activity` implementations and the Process Virtual Machine itself. The main purpose is to make various aspects of the Process Virtual Machine configurable so that the PVM and the languages that run on top can work in a standard Java environment as well as an enterprise Java environment.

The environment is partitioned into a set of contexts. Each context can have its own lifecycle. For instance, the application context will strech over the full lifetime of the application. The block context only for the duration of a try-finally block. Typically a block context represents a database transaction. Each context exposes a list of key-value pairs.

8.2. EnvironmentFactory

To start working with an environment, you need an EnvironmentFactory. One single environment factory object can be used throughout the complete lifetime of the application. So typically this is kept in a static member field. The EnvironmentFactory itself is the application context.

An `EnvironmentFactory` is typically obtained by parsing a configuration file like this:

```
static EnvironmentFactory environmentFactory =
        EnvironmentFactory.parse(new ResourceStreamSource("pvm.cfg.xml");
```

See javadocs package org.jbpm.stream for more types of stream sources.

There is a default parser in the environment factory that will create `DefaultEnvironmentFactory`s. The idea is that we'll also support spring as an IoC container. But that is still TODO. Feel free to help us out :-). The parser can be configured with the static setter method `EnvironmentFactory.setParser(Parser)`

8.3. Environment block

An environment exists for the duration of a try-finally block. This is how an environment block looks like:

```
Environment environment = environmentFactory.openEnvironment();
try {
```

```
  ...

} finally {
  environment.close();
}
```

The environment block defines another lifespan: the `block` context. A transaction would be a typical example of an object that is defined in the block context.

Inside such a block, objects can be looked up from the environment by name or by type. If objects can looked up from the environment with method `environment.get(String name)` or `<T> T environment.get(Class<T>)`.

when an environment is created, it has a `application` context and a `block` context.

In the default implementation, the `application` context and the `block` context are `WireContexts`. A `WireContext` contains a description of how its objects are created and wired together to form object graphs.

8.4. Example

To start with a simple example, we'll need a `Book`:

```
public class Book {
  ...
  public Book() {}
  ...
}
```

Then let's create an environment factory that knows how to create book

```
static EnvironmentFactory environmentFactory = EnvironmentFactory.parse(new StringStreamSource(
    "<environment>" +
    "  <application>" +
    "    <object name='book' class='org.jbpm.examples.ch09.Book' />" +
    "  </application>" +
    "</environment>"
));
```

Now we'll create an environment block with this environment factory and we'll look up the book in the environment. First the lookup is done by type and secondly by name.

```
Environment environment = environmentFactory.openEnvironment();
try {

  Book book = environment.get(Book.class);
  assertNotNull(book);

  assertSame(book, environment.get("book"));

} finally {
  environment.close();
}
```

To prevent that you have to pass the environment as a parameter in all methods, the current environment is maintained in a threadlocal stack:

```
Environment environment = Environment.getCurrent();
```

8.5. Context

Contexts can be added and removed dynamically. Anything can be exposed as a `Context`.

```
public interface Context {

  Object get(String key);
  <T> T get(Class<T> type);
  Set<String> keys();

  ...
}
```

When doing a lookup on the environment, there is a default search order in which the contexts will be scanned for the requested object. The default order is the inverse of the sequence in which the contexts were added. E.g. if an object is defined in both the application context and in the block context, the block context is considered more applicable and that will be scanned first. Alternatively, an explicit search order can be passed in with the `get` lookups as an optional parameter.

9

Persistence

9.1. Standard environment configuration

This section describes how the environment can be configured to use hibernate in a standard Java environment.

```
01 | <environment>
02 |
03 |   <application>
04 |     <hibernate-session-factory />
05 |     <hibernate-configuration>
06 |       <properties resource="hibernate.properties" />
07 |       <mappings resources="org/jbpm/pvm.hibernate.mappings.xml" />
08 |       <cache-configuration
09 |            resource="org/jbpm/pvm.definition.cache.xml"
10 |            usage="nonstrict-read-write" />
11 |     </hibernate-configuration>
12 |   </application>
13 |
14 |   <block>
15 |     <standard-transaction />
16 |     <hibernate-session />
17 |     <pvm-db-session />
18 |   </block>
19 |
20 | </environment>
```

`line 04` specifies a hibernate session factory in the application context. This means that a hibernate session factory is lazy created when it is first needed and cached in the `EnvironmentFactory`.

A hibernate session factory is build calling the method `buildSessionFactory()` on a hibernate configuration. By default, the hibernate configuration will be looked up by type.

`line 05` specifies a hibernate configuration.

`line 06` specifies the that the resource file `hibernate.properties` should be loaded into the configuration.

`line 07` (note the plural form of mapping**s**) specifies that resources `org/jbpm/pvm.hibernate.mappings.xml` contain references to hibernate mapping files or resources that should be included into the configuration. Also note the plural form of `resources`. This means that not one, but all the resource files on the whole classpath will be found. This way new library components containing a `org/jbpm/pvm.hibernate.mappings.xml` resource can plug automatically into the same hibernate session by just being added to the classpath.

Alternatively, individual hibernate mapping files can be referenced with the singular `mapping` element.

`line 08 - 10` provide a single place to specify the hibernate caching strategy for all the PVM classes and collections.

`line 15` specifies a standard transaction. This is a very simple global transaction strategy without recovery that can be used in standard environments to get all-or-nothing semantics over multiple transactional resources.

`line 16` specifies the hibernate session that will automatically register itself with the standard transaction.

`line 17` specifies a `PvmDbSession`. That is a class that adds methods that bind to specific queries to be executed on the hibernate session.

9.2. Standard hibernate configuration

Here is a set of default properties to configure hibernate with hsqldb in a standard Java environment.

```
hibernate.dialect                      org.hibernate.dialect.HSQLDialect
hibernate.connection.driver_class      org.hsqldb.jdbcDriver
hibernate.connection.url               jdbc:hsqldb:mem:.
hibernate.connection.username          sa
hibernate.connection.password
hibernate.cache.use_second_level_cache true
hibernate.cache.provider_class         org.hibernate.cache.HashtableCacheProvider
```

Optionally in development the schema export can be used to create the schema when the session factory is created and drop the schema when the session factory is closed.

```
hibernate.hbm2ddl.auto                 create-drop
```

For more information about hibernate configurations, see the hibernate reference manual.

9.3. Standard transaction

By default, the `<hibernate-session />` will start a hibernate transaction with `session.beginTransaction()`. Then the hibernate transaction is wrapped in a `org.jbpm.hibernate.HibernateTransactionResource` and that resource is enlisted with the `<standard-transaction />` (`org.jbpm.tx.StandardTransaction`)

Inside of the environment block, the transaction is available through `environment.getTransaction()`. So inside an environment block, the transaction can be rolled back with `environment.getTransaction().setRollbackOnly()`

When created, the standard transaction will register itself to be notified on the close of the environment. So in side the close, the standard transaction will commit or rollback depending on whether `setRollbackOnly()` was called.

So in the configuration shown above, each environment block will be a separate transaction. At least, if the hibernate session is used.

9.4. Basics of process persistence

In the next example, we'll show how this hibernate persistence is used with a concrete example. The 'persistent process' is a simple three-step process:

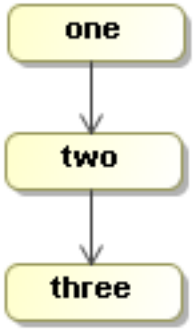

Figure 9.1. The persistent process

The activities in the three nodes will be wait states just like in Section 2.4

To make sure we can persist this class, we create the hibernate mapping for it and add it to the configuration like this:

```
<hibernate-configuration>
  <properties resource="hibernate.properties" />
  <mappings resources="org/jbpm/pvm.hibernate.mappings.xml" />
  <mapping resource="org/jbpm/examples/ch09/state.hbm.xml" />
  <cache-configuration
        resource="org/jbpm/pvm.definition.cache.xml"
        usage="nonstrict-read-write" />
```

The next code pieces show the contents of one unit test method. The method will first create the environment factory. Then, in a first transaction, a process definition will be created and saved into the database. Then the next transaction will create a new execution of that process. And the following two transactions will provide external triggers to the execution.

```
EnvironmentFactory environmentFactory = EnvironmentFactory.parse(new ResourceStreamSource(
    "org/jbpm/examples/ch09/environment.cfg.xml"
));
```

Then in a first transaction, a process is created and saved in the database. This is typically referred to as deploying a process and it only needs to be done once.

```
Environment environment = environmentFactory.openEnvironment();
try {
  PvmDbSession pvmDbSession = environment.get(PvmDbSession.class);

  ProcessDefinition processDefinition = ProcessFactory.build("persisted process")
    .node("one").initial().behaviour(new State())
      .transition().to("two")
    .node("two").behaviour(new State())
      .transition().to("three")
    .node("three").behaviour(new State())
  .done();

  pvmDbSession.save(processDefinition);
} finally {
  environment.close();
}
```

In the previous transaction, the process definition, the nodes and transitions will be inserted into the database tables.

Next we'll show how a new process execution can be started for this process definition. Note that in this case, we provide a business key called 'first'. This will make it easy for us to retrieve the same execution from the database in subsequent transactions. After starting the new process execution, it will wait in node 'one' cause the behaviour is a wait state.

```
environment = environmentFactory.openEnvironment();
try {
  PvmDbSession pvmDbSession = environment.get(PvmDbSession.class);

  ProcessDefinition processDefinition = pvmDbSession.findProcessDefinition("persisted process");
  assertNotNull(processDefinition);

  Execution execution = processDefinition.startExecution("first");
  assertEquals("one", execution.getNode().getName());
  pvmDbSession.save(execution);

} finally {
  environment.close();
}
```

In the previous transaction, a new execution record will be inserted into the database.

Next we feed in an external trigger into this existing process execution. We load the execution, provide a signal and just save it back into the database.

```
environment = environmentFactory.openEnvironment();
try {
  PvmDbSession pvmDbSession = environment.get(PvmDbSession.class);

  Execution execution = pvmDbSession.findExecution("persisted process", "first");
  assertNotNull(execution);
  assertEquals("one", execution.getNode().getName());

  // external trigger that will cause the execution to execute until
  // it reaches the next wait state
  execution.signal();

  assertEquals("two", execution.getNode().getName());

  pvmDbSession.save(execution);

} finally {
  environment.close();
}
```

The previous transaction will result in an update of the existing execution, reassigning the foreign key to reference another record in the node table.

```
UPDATE JBPM_EXECUTION
SET
  NODE_=?,
  DBVERSION_=?,
  ...
WHERE DBID_=?
  AND DBVERSION_=?
```

The version in this SQL shows the automatic optimistic locking that is baked into the PVM persistence so that process persistence can easily scale to multiple JVM's or multiple machines.

In the example code, there is one more transaction that is completely similar to the previous which takes the execution from node 'two' to node 'three'.

All of this shows that the PVM can move from one wait state to another wait state transactionally. Each transaction correcponds to a state transition.

Note that in case of automatic activities, multiple activities will be executed before the execution reaches a wait state. Typically that is desired behaviour. In case the automatic activities take too long or you don't want to block the original transaction to wait for the completion of those automatic activities, check out Chapter 11 to learn about how it's possible to demarcate transactions in the process definition, which can also be seen as safe-points during process execution.

9.5. Business key

TODO

TODO: General persistence architecture

TODO: Object references

TODO: Threads, concurrency with respect to forks and joins

TODO: Caching

TODO: Process instance migration

10

Services

10.1. Introduction

All session facades are called services in the PVM and it's related projects. A service is the front door of the API. It has a number of methods that expose the functionality of the component. The service takes care of getting or setting up an environment for each operation that is invoked.

10.2. PvmService

The class org.jbpm.PvmService is the main way to access functionality from the PVM.

10.3. Architecture

Service methods are implemented through command classes. Each method creates a command object and the command is executed with the `execute` method of the `CommandService`. The `CommandService` is responsible for setting up the environment.

There are three command executors:

- `standard-command-service` will just execute the command and pass in the current environment.
- (UNTESTED) `async-command-service` will send an asynchronous message. So right after that in a separate transaction, the message is consumed and the command is executed.
- (TODO) `cmt-command-service` will delegate execution of the command to a local SLSB that has transaction attribute `required`.
- (TODO) `remote-command-service` will delegate execution of the command to a remote SLSB.

Each of the command services can be configured with a list of interceptors that span around the command execution. Following interceptors are available:

- `environment-interceptor`: Will execute the command within an environment block.
- (UNTESTED) `authorization-interceptor`: Will perform an authrorization check before the command is executed. The authorization interceptor will look up the AuthorizationSession from the environment to delegate the actual authorization check to.

- `retry-interceptor`: Will catch hibernate's optmistic locking exceptions (StaleStateException) and retries to execute the command for a configurable number of times
- `transaction-interceptor`: Will get the transaction from the current context and invoke setRollbackOnly() on it in case an exception comes out of the command execution.

Following configuration can be used in default standard persistence situations:

```
<environment>
  <application>

    <pvm-service />

    <standard-command-service>
      <retry-interceptor />
      <environment-interceptor />
      <transaction-interceptor />
    </standard-command-service>

    ...
  </application>
  ...

</environment>
```

11

Asynchronous continuations

12

Timers

13

Process languages

TODO: xml parser infrastructure

TODO: inherit from ProcessDefinitionImpl, ExecutionImpl

TODO: overriding the default proceed()

TODO: node type implementations

TODO: persistence

TODO: compensation: languages like bpel and bpnm define that as a normal contination that fits within the process structures available in the pvm (taking a transition and executing a nested node).