7 Forms of Business Process Management with JBoss jBPM

By Tom Baeyens

This article will explain Business Process Management (BPM) in terms of 7 distinct use cases for JBoss jBPM. By giving more insight in those use cases, you'll get a better understanding of the different forms of BPM and workflow and when a BPM engine like jBPM makes sense in your project. We'll also highlight the specific jPDL process language features related to those use cases.

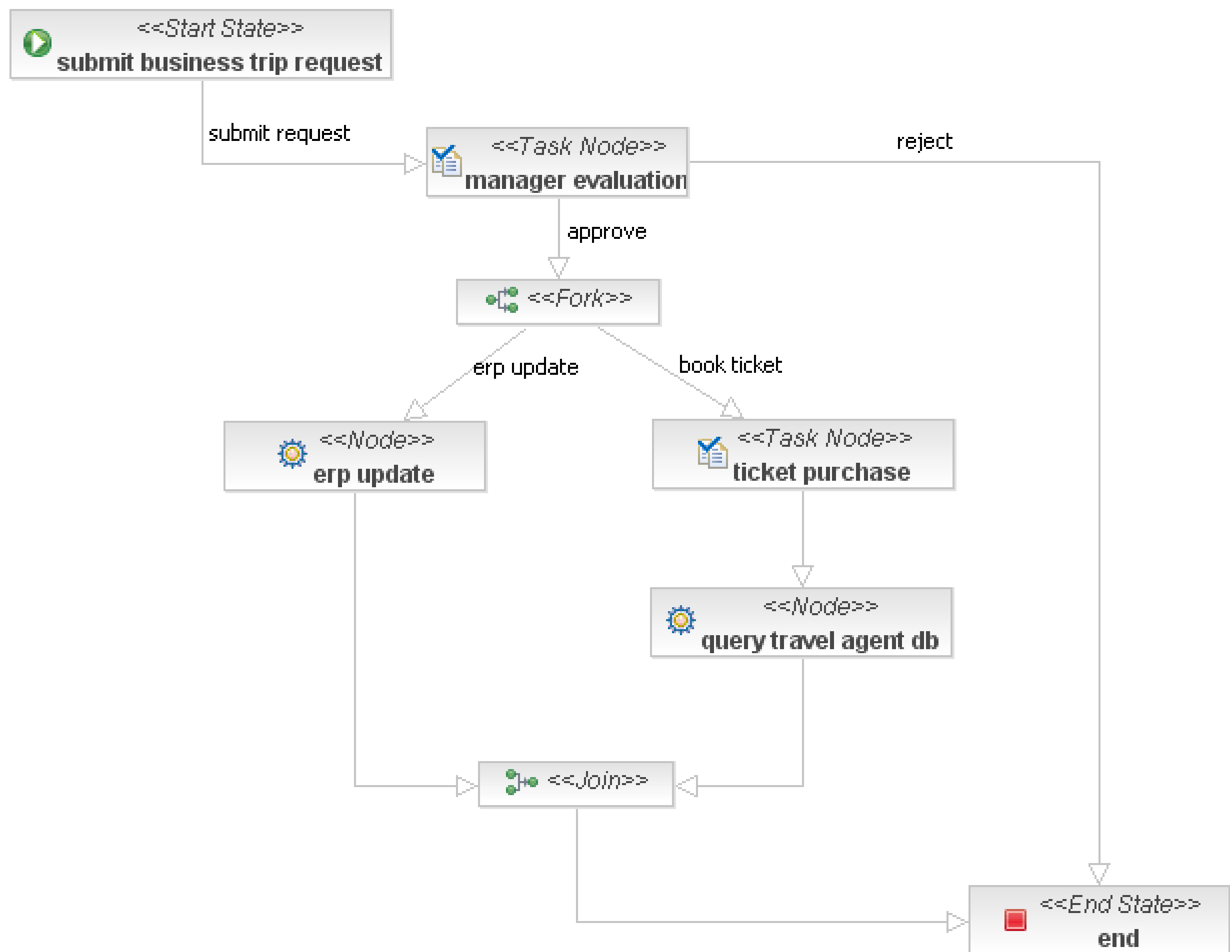
The term BPM is highly overloaded and used for many different things resulting in a lot of confusion. These use cases give concrete descriptions for the different interpretations of the term BPM.

The individual nature of these use cases is important. BPM software vendors often take a mix of different aspects and concerns into account when developing their products. That often results into BPM products that are suitable only for a specific purpose in a specific environment. This is in my opinion the reason why there are so many different BPM products which only serve a small niche market. It also explains why new products and standards in this space keep appearing, don't get enough momentum and then get pushed aside by yet another new product or standard.

When evaluating a BPM products, it should be done with specific use cases in mind.

What is JBoss jBPM

JBoss jBPM is a flexible and powerful BPM engine. In essence, BPM systems allow for execution flows to be specified graphically. As an example, here is a process diagram for a business trip:



A key capability of BPM systems is that processes steps can be wait states. For example in the business trip process above, nodes 'manager evaluation' and 'ticket purchase' are human tasks. When the execution of the process arrives in those nodes, the system executing the process should wait till the assigned user completes the task.

From a software technical point of view that capability is a big deal. As the alternative is a bunch of methods that are linked by HTTP requests, Message Driven Beans (MDB), database triggers, task forms, etc. Even when using the most applicable architectural components available in Java today, it is still very easy to end up in a bunch of unmaintainable hooks and eyes. Using an overall business process makes it a lot easier to see and maintain the overall execution flow, even from a software technical perspective.

JBoss jBPM does exactly that and it differentiates itself from other BPM projects in the following topics:

- Easily embeddable into a Java project. Traditional BPM Systems typically require a separate server to be installed which makes it hard to integrate into the Java software development cycle. One of the deployments that JBoss jBPM supports is just adding the jBPM library to the classpath. The jBPM tables can be hosted in any database next to the application's tables. Using JBoss jBPM really fits with the normal way of developing Java software.
- Support for multiple process languages. The view on what BPM actually is has not yet been stabilized. There are currently many different interpretations of what BPM is, resulting in big fragmentation in the market. In fact, the body of this article tries to identify 6 distinct concepts that all are associated to BPM.

- Very flexible transaction management. If your application just uses a JDBC connection in a standard Java environment, then jBPM can use that very JDBC connection to perform its work. If your application uses hibernate in a standard environment, then jBPM can use the same hibernate session factory. If your application runs in an enterprise environment, then jBPM can bind to the surrounding JTA transaction.
- Readable jPDL process language. For developers it is very convenient if the process language is compact and readable. Not all developers want to keep using the graphical editor. Most hard core programmers start to hand code the processes after a while. Then jBPM's jPDL process language is the most readable, compact and complete.

Standards BPEL and BPMN

Todays most known standards in the space of BPM are BPEL and BPMN. Those two standards have a completely different background. That is a clear indication of the diverse type of problems that is currently associated to BPM.

BPEL is defined with an Enterprise Service Bus (ESB) in mind. It's all based on WSDL, which binds naturally to web services. The result of deploying a BPEL process is that a new service is being published. A BPEL process can hence be seen as a scripting language for services on the bus. A more elaborate conceptual explanation of BPEL can be found in *Process Component Models: The Next Generation In Workflow ?* So BPEL is an executable process language, which implies that a it is human to computer communication, just like a programming language.

On the other hand, the first target of **BPMN** is modeling process diagrams by non-technical business analysts. It defines the shapes, types and meaning of the boxes and arrows. This type of modeling should be seen in the context of 'BPM as a discipline' (see below). By definition, non technical business analyst do not think in terms of web services or other technology aspects. BPMN processes are primarily intended for communication between humans.

So far, so good. But now comes the hard and confusing reality: BPEL has been presented as a solution for 'BPM as a discipline' and BPMN 2.0 plans to add concrete executable semantics. These days, most IT people already realized that BPEL is not a convenient solution for 'BPM as a discipline'. On the other hand adding concrete executable semantics to a business analyst modeling notation means that non-technical analysts will be implicitly writing a piece of executable software. Would you put software into production written by a non-technical person ?

Use Case 1 : BPM as a discipline

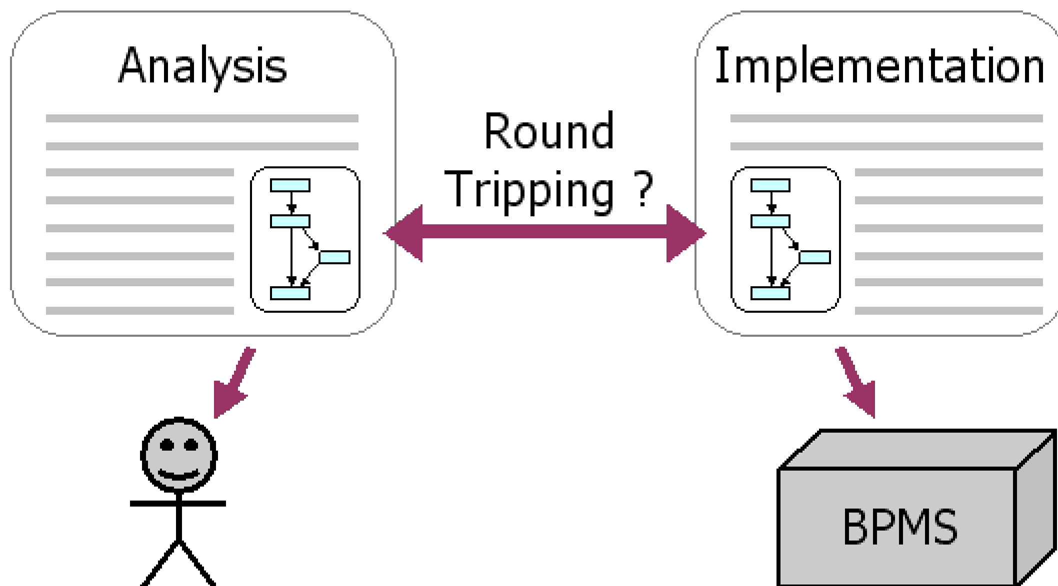
BPM as a discipline refers to the analysis, documentation and improvement of the procedures that describe how people and systems work together in an organization. Realize that most BPM is done in this way, not even resulting into any form of software or IT support. MS Visio is one the most used tools to document business processes.

Of course, once the business process is documented, it might make sense to develop software support for it. Suppose that a business analyst modeled and documented a 'business trip' process, then one usage might be to find optimizations in that process. And another usage might be the development of software support for business trips.

Please be aware that an automatic translation between pure analysis models and executable process models are not feasible in general. Initially BPM products tried to make this translation automagically transparent. In the context of BPM as a discipline, we believe that a process model from a non technical business analyst can never be translated into an executable process model by just adding technical details to it.

The next attempt was to acknowledge existence of an analysis model and an executable process

model and then try to keep them in sync. Especially because of the big differences between BPMN and BPEL this approach got a lot of attention. But then the problem of maintaining the links between analysis and executable process model appears. Who is going to spend the time to link the analysis blocks to the executable blocks and then keep that mapping up to date. This is illustrated in the following picture:



For mainstream software development resulting from 'BPM as a discipline', we believe that the original analysis diagram should serve as input with the rest of the software requirements. Developers should then construct an executable process that looks as close as possible to the original analysis diagram. After that, the analysis diagram is discarded and replaced by the executable process diagram. That way, the developer is in full control of the software technical aspects, while the analyst will still recognize the diagram.

jBPM's jPDL language is designed to facilitate this approach. First of all, it's based on free graph modeling. Secondly, it has so called actions, which can be seen as event listeners. These event listeners are pieces of Java code that are called when process execution produces the event, but they are hidden from the process diagram. This allows the developer to add programming logic without changing the diagram structure. Also super-states are often used in the context of creating better communication between business analyst and developer.

Use Case 2 : Combining template based and ad hoc task management

Orchestrating human tasks can be found in most process engines in one form or another. But what is often overlooked is that template based task orchestration only suits for a limited number of scenarios. First the process must be relatively stable. And secondly, enough executions of this process have to happen so that the gain that can be achieved with software support is worth the development effort.

A lot of work being done in organizations does not meet those two requirements. For example, organizing a one-time team building event, a restyling of the office space or cleaning up after the sprinkler system went off unexpectedly. For that type of work, people in charge will invent the process on the spot and it will only be executed once.

Human Interaction Management (HIM) focuses on this type of work. A story in HIM reflects a task that person creates for which an ad-hoc process will be created. People can get involved in different roles. There are big benefits of tracing such work with a task management system. First, people that get involved with such a task after a while will get instant visibility into the history of the

whole story. And secondly, an audit trail is logged automatically.

In jBPM 4, the task management component will support this ad hoc human tasks. The combination will be awsome. Human tasks in processes can be specified at a course grained level. When such a process task is created for a person, the assignee can involve other people by creating subtasks and assigning people in different roles to these tasks. Only when the owner decides that the overall task is finished, then the process will continue.

Use case 3 : Transactional asynchronous architectures

Gregor Hohpe's Entperprise Integration Patterns describes building blocks for asynchronous architectures. These patterns are based on the notion of Message Oriented Middleware (MOM) aka message brokers. MOM's are good at asynchronous transactional communication between two systems and JMS is the Java standard to work with it.

Of course point-to-point communication can be set up easily with existing Java infrastructure like Message Driven Beans. But in many cases many of these point-to-point communications are related. For example, to process one order, a message might come from a client into the order processing system. As part of that transaction, a notification might be sent to the warehouse and stock planning team. Eventually after many more steps, a message might be sent back to the client as a confirmation of the order. If all these transactional communications are build linearly, it will be very hard to manage the overall state of order processing.

BPEL also focuses on asynchronous architectures, but then in a (web) services environment, rather then a Java environment. In BPEL only (web) services can be invoked and XML based technologies like XPath are integrated. So for more complex calculations, a piece of programming logic needs to be included. To include programming logic in BPEL, it needs t be wrapped and exposed as a service. That can be quite cumbersome in certain situations.

In contrast, jPDL is embedded in a Java environment. Such a central dispatching functionality is really suited to be implemented by a process language like jPDL. An incoming message can start a new jPDL process execution or provide a trigger for an existing execution. jPDL allows to include programming logic straight into the process transaction. Cause jPDL is based on a Java architecture, that results into much more flexibility and convenience.

Using the central dispatching approach creates immediately more insight into the overall state of a process. In terms of our order processing example, it will be very easy to keep track of the state of each order. Further more, process engines like jPDL collect the history information. This is an extra feature that you get for free when using a process engine in this way. From the history information it is very easy to collect valuable statistical information like the average time in each step of the process.

The difference with the use case 'BPM as a discipline' is that in this case, the goal is software technical in nature. The process is looked at from an implementation perspective. There doesn't have to be a relation to a business level analysis process.

Use case 4 : Service orchestration

Service orchestration is actually a variant of the previous use case 'transactional asynchronous architectures'. Interactions between services on an Enterprise Service Bus (ESB) are usually asynchronous. This time, the communication is not done through sending messages over a MOM, but instead by calling services over an ESB.

BPEL is a process language specifically designed for this use case. Apart from a construct to invoke WSDL services, it has a whole infrastructure for supporting conversations. Conversations are so called long running processes. The clue here is that a BPEL process specifies a number of

service operations that are being published when a BPEL process is deployed. A BPEL process can then contain receive activities, which mean that the process execution will wait until a service operation invocation is received.

In jBPM we have implemented the BPEL process language as on of the languages that we support on top of our Process Virtual Machine.

Use Case 5 : Visual programming

With visual programming, we will target developers that do not yet have the full skillset to develop in Java. They can graphically specify the activities and draw transitions to indicate the control flow. Instead of typing Java code, just put blocks like 'perform hql query', 'generate pdf', 'send email', 'read file', 'parse xml', 'send msg to esb'. Instead of writing Java, they will be composing software programs by linking activity boxes in the diagram and filling in the configuration details.

This will, of course, not have the same kind of flexibility as Java programming itself. Visual programming as we describe it here will not replace programming as we know it today. Today's programmers can be considered the power users and that kind will always be needed. But visual programming can lower the treshold to build applications for developers that have no or limited Java knowledge.

The Process Virtual Machine (PVM) is the foundation of jBPM. In the PVM infrastructure it is really easy to add a new activity type. Think of it as a kind of command pattern, where the commands are configurable. So it will be very easy for us to develop new activity types. This way, we can expose the bulk part of the Java programming functionalities as activity types in the jPDL process language.

In jPDL, these visual programs will be executable transactionally or without persisting the state of the execution.

Use Case 6 : Thread Control Language

This is a specific aspect of visual programming that also can be used by experienced developers. Coding multi-threaded Java programs is not easy. In fact, it is at least tedious and mostly pretty hard to code. Starting new threads, passing data into them, joining and making them stop properly can be a challenge.

We'll develop a Thread Control Language which lets you specify a multithreaded Java concurrency by drawing forks, joins and method activities. A method activity invokes a methods on your Java object. And the concurrency control is handled by the forks and join in the process. These processes will be executed without any persistence.

Use Case 7 : Easy creation of DSLs

General purpose process languages are different from domain specific process languages. For example, think of a process language to specify approvals in an Enterprise Content Management (ECM) system. This gives a scoped functionality and a fixed environment. In such cases, a specific process language could be developed for this purpose.

One of the advantages of such dedicated languages is that they can be made simple enough so that non technical business users can actually create fully executable processes.

Another example of a domain specific process language is SEAM's Pageflow. It allows developers of a JSF based web site to specify the pages and navigations between the pages graphically.

There is even an easier way. Instead of creating a full process language for a specific purpose, it is also possible to leverage jPDL's capabilities and just add new node types to it. That is already

possible in jBPM 3 and will be peanuts in jBPM 4, even adding graphical support in the process designer for these new node types will be made simple.

Conclusion

The typical understanding of BPM is that non technical business people create diagrams that then automagically get executed on a BPM system. The first use case 'BPM as a discipline' describes that point of view. The value lies in the fact that non technical business people can communicate with the developers around a diagram. That diagram facilitates the communication between business analysts and developers.

But even in that use case, the traditional understanding needs fine tuning. An analysis diagram at some point has to be converted into an executable process. At that moment, the responsibility over the process is transferred from the analyst to the developer. Many vendors have created the tools and illusion that this can be done automagically. But in practice, this black box approach creates more problems then it solves.

The description of the 7 individual use cases shows distinct aspects of BPM. Knowing the difference between service orchestration and 'BPM as a discipline' will be key for organizations to select the most appropriate technology.

All of the use cases have at least a technical side. That is understandable as all the use cases here target some form of software automation at some point. And other use cases target solely technical aspects. That is why the embeddability of jBPM is so important. Monolithic BPM engines have a high treshold to be incorporated into a typical software development project. jBPM has a big focus on delivering solutions for these distinct use cases to developers in a way that is easy for them to consume and integrate into their own software development project.

About the author:

Tom Baeyens is the founder and lead of JBoss jBPM, the leading open source BPM system. Tom mission is to bring the power of BPM technology into the hands of the developers. He's a frequent speaker at international conferences and maintains a blog at
http://processdevelopments.blogspot.com