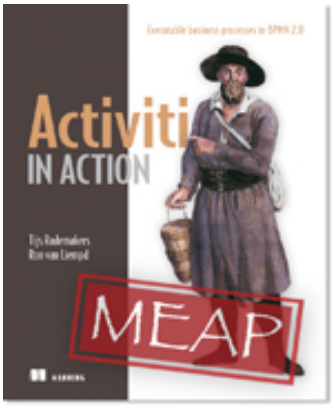
MANNING PUBLICATIONS



Activiti in Action
By Tijs Rademakers and Ron van Liempd

Workflow is an important part of developing process applications. In a lot of cases, not all tasks in a process can be executed automatically by invoking web services or other external resources. In this article, based on chapter 10 of Activiti in Action, the authors introduce you to subtasks and task delegation that's supported via the Activiti API and partially via the Activiti Explorer.

Going Beyond a Simple User Task

The workflow functionality surrounding a user task doesn't stop with claiming and completing tasks in the Activiti Engine. In this article, we'll be looking at advanced workflow patterns to support use cases where we, for example, need hierarchical tasks or want to delegate a specific task to another person. First, we discuss the concept of creating subtasks. Then, we'll introduce a common management skill called delegation. Finally, we'll show an implementation of the famous four-eyes principle workflow pattern using a task listener.

Working with subtasks

In a process definition, we're able to define a flat user task. BPMN 2.0 doesn't talk about grouping user tasks into a parent task with multiple subtasks. But, there are a lot of cases where there's a need of hierarchy in tasks. For example, when you are planning a wedding, there are a lot of subtasks related to this large parent task, such as inviting the guests, hiring a wedding location, choosing the dinner menu, and so on.

The Activiti Engine provides functionality to create a subtask for a specific user task. Let's look at a unit test showing the use of the `TaskService` interface to create subtasks in code listing 1.

Listing 1 A unit test showing the functionality to create subtasks

```
public class SubTaskTest extends AbstractTest {

  @Rule
  public ActivitiRule activitiRule = new ActivitiRule(
      "activiti.cfg-mem.xml");

  @Test
  public void completeSubTasks() {
    TaskService taskService = activitiRule.getTaskService();
    Task parentTask = taskService.newTask();                          #1
    parentTask.setAssignee("kermit");
    taskService.saveTask(parentTask);
    createSubTask("fozzie", parentTask.getId());                      #2
    createSubTask("gonzo", parentTask.getId());
    List<Task> taskList = taskService.getSubTasks(                    #3
        parentTask.getId());
    assertEquals(2, taskList.size());
    taskService.complete(taskList.get(0).getId());
    taskService.complete(taskList.get(1).getId());                    #4
```

```
    taskList = taskService.getSubTasks(parentTask.getId());
    assertEquals(0, taskList.size());

    List<HistoricTaskInstance> historicTaskList =
        activitiRule.getHistoryService()
            .createHistoricTaskInstanceQuery()
            .finished()
            .list();
    assertEquals(2, historicTaskList.size());

    taskService.complete(parentTask.getId());                          #5

    historicTaskList = activitiRule.getHistoryService()
            .createHistoricTaskInstanceQuery()
            .finished()
            .list();
    assertEquals(3, historicTaskList.size());                          #6

    List<String> taskIds = new ArrayList<String>();
    for (HistoricTaskInstance historicTaskInstance :
        historicTaskList) {

      assertNotNull(historicTaskInstance.getEndTime());
      taskIds.add(historicTaskInstance.getId());
    }

    for(String taskId : taskIds) {
      activitiRule.getHistoryService()
          .deleteHistoricTaskInstance(taskId);                         #A
    }
  }

  private void createSubTask(String assignee,
      String parentTaskId) {

    TaskService taskService = activitiRule.getTaskService();
    Task subTask = taskService.newTask();
    subTask.setAssignee(assignee);
    subTask.setParentTaskId(parentTaskId);                             #B
    taskService.saveTask(subTask);
  }
}
```

#A Cleans up the created user tasks
#B Sets the parent task identifier
#1 Creates a parent user task
#2 Creates a subtask
#3 Gets all subtasks
#4 Completes the second subtask
#5 Completes the parent task
#6 All tasks are completed now

In this example, we don't use a process definition. This shows that we can easily create user tasks without the need for a process definition, and it's easier to show the use of subtasks. The first step in the unit test is to create a parent user task (#1). The Activiti Engine has created a user task assigned to Kermit.

Then, we can use the identifier of this parent task to create a subtask (#2). We create two subtasks and assign them to Fozzie and Gonzo. To retrieve the subtasks of a specific user task, we can use the `getSubTasks` method on the `TaskService` with the parent task identifier as input parameter (#3).

The subtasks are created just like normal user tasks, but the difference is that they have a parent task identifier pointing to the parent user task. But, we can complete the individual subtasks just like we can do with a standard user task (#4).

> **NOTE** When the subtasks of a parent user task are all completed, the parent user task will not be automatically completed. So, you'll have to explicitly complete the parent user task as well.

With the subtasks completed, we'll now also complete the parent user task (#5). When we query the history tables of the Activiti Engine for finished user tasks, we should now see that all three user tasks (the parent task and the two subtasks) are completed (#6).

The unit test of listing 1 shows how to deal with subtasks in a default way. First the subtasks are completed and finally the parent user task. In the next example (see listing 2), we show that you can also complete a parent user task without completing the subtasks.

Listing 2 Completing a parent user task before the subtasks have been completed

```
@Test
public void completeSubTasksViaParentTask() {
  TaskService taskService = activitiRule.getTaskService();
  Task parentTask = taskService.newTask();
  parentTask.setAssignee("kermit");
  taskService.saveTask(parentTask);
  createSubTask("fozzie", parentTask.getId());                          #1
  createSubTask("gonzo", parentTask.getId());
  List<Task> taskList = taskService.getSubTasks(
      parentTask.getId());
  assertEquals(2, taskList.size());
  taskService.complete(parentTask.getId());                             #2
  taskList = taskService.getSubTasks(parentTask.getId());
  assertEquals(0, taskList.size());                                     #3
  List<HistoricTaskInstance> historicTaskList =
      activitiRule.getHistoryService()
          .createHistoricTaskInstanceQuery()
          .finished()
          .list();
  assertEquals(3, historicTaskList.size());                             #4

  List<String> taskIds = new ArrayList<String>();
  for (HistoricTaskInstance historicTaskInstance :
      historicTaskList) {

    assertNotNull(historicTaskInstance.getEndTime());
    taskIds.add(historicTaskInstance.getId());
  }

  for(String taskId : taskIds) {
    activitiRule.getHistoryService()
        .deleteHistoricTaskInstance(taskId);
  }
}
```

#1 Creates a subtask
#2 Completes the parent task
#3 Retrieves all subtasks
#4 All tasks have finished

This unit test method is implemented in the same `SubTaskTest` unit test class we discussed in code listing 1. The first part of the unit test is also very similar. We start with creating a parent user task and two subtasks (#1). But, now, we complete the parent user task (#2) before we complete the subtasks.

When we now query the Activiti Engine for subtasks of the completed parent user task, we get zero tasks back (#3). This is the result of completing the parent user task. When a parent user task is completed, the subtasks are automatically completed as well. So, when we query the Activiti Engine for finished task instances, we get three results—the parent user task and the two subtasks (#4).

As you could see, it's very easy to create subtasks on a parent user task. For a user task in a process definition, you could automate creating subtasks by implementing a task listener on the user task. This task listener can create a specified number of subtasks when the user task is created. The following code snippet shows how you can implement this.

```
public class SubTaskListener implements TaskListener {

  private Expression subTaskList;

  @Override
```

```
  public void notify(DelegateTask delegateTask) {
    ProcessEngine processEngine = ProcessEngines.getProcessEngines()
        .get(ProcessEngines.NAME_DEFAULT);
    @SuppressWarnings("unchecked")
    List<String> subTaskNames = (List<String>)
        subTaskList.getValue(delegateTask.getExecution());
    for(String subTaskName : subTaskNames) {
      TaskService taskService = processEngine.getTaskService();
      Task subTask = taskService.newTask();
      subTask.setName(subTaskName);
      subTask.setAssignee("kermit");
      subTask.setParentTaskId(delegateTask.getId());
      taskService.saveTask(subTask);
    }
  }

  public void setSubTaskList(Expression subTaskList) {
    this.subTaskList = subTaskList;
  }
}
```

When a user task is created in a process definition, this `TaskListener` will be invoked. Then, we retrieve a process variable containing a list of subtask names from the process context and create a subtask for every name in that list. Of course, this example needs some polishing before you can use this in an enterprise context, but it shows that you can also create subtasks in a process instance without lots of coding.

Another way to create subtasks in a parent user task is by using the Activiti Explorer. For example, when a process instance has created a user task, you can use the Activiti Explorer to create a number of subtasks on the fly. Figure 1 shows an example of a user task where a subtask has been created.

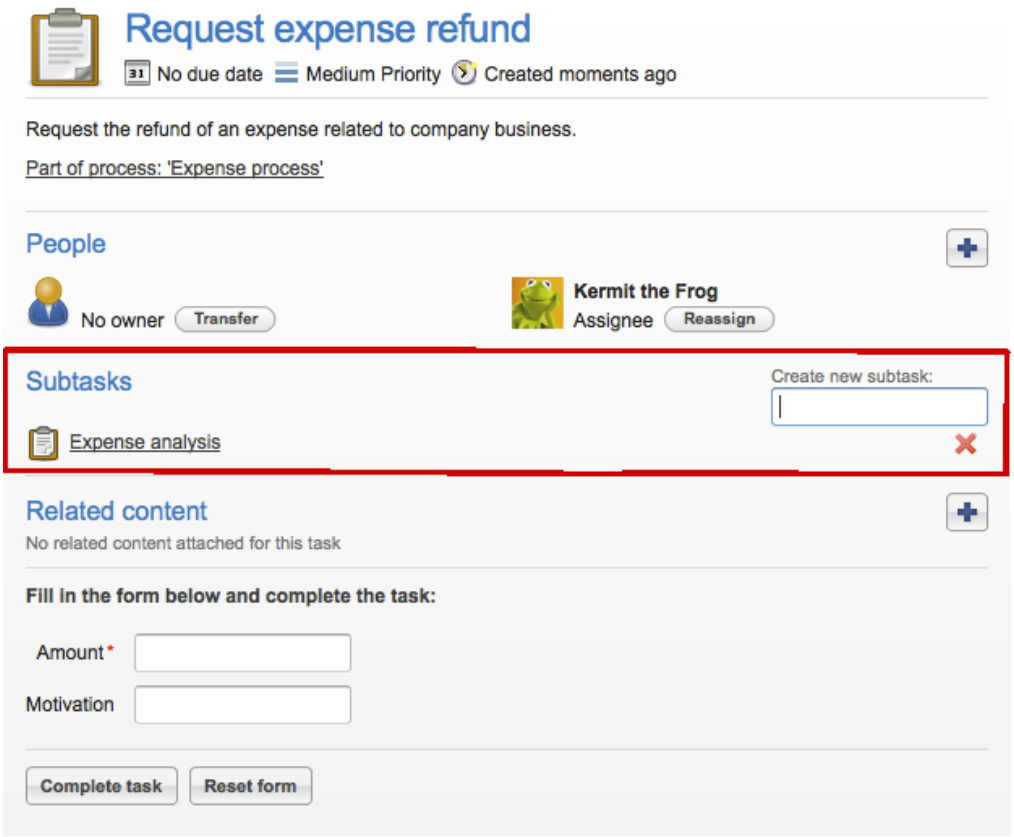


Figure 1 A screenshot of the Activiti Explorer highlighting the functionality to create subtasks

By only defining a name for the subtask, a new one is created. You can click through to the subtask to assign it to a user or group.

Now, let's move on to the next workflow feature we want to discuss—delegation. This reflects the managerial style of passing tasks from a manager to an employee.

Delegating tasks

Delegating a task means transferring the task to another person to complete the work and then the user tasks will be assigned back to the person who delegated to task, so that it can be reviewed and completed. Let's look at the default sequence of steps that are involved when delegating a user task in figure 2.

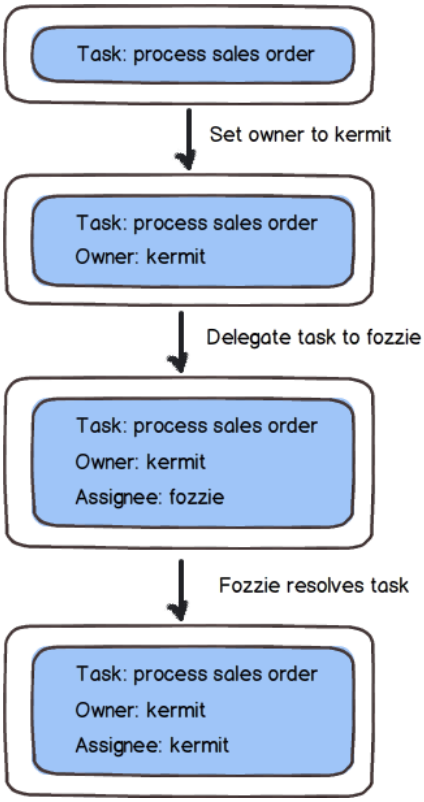


Figure 2 The default flow of steps when delegating a user task. In this example, Kermit delegates a task to Fozzie.

When a user task is created in a process instance or as an adhoc user task, at first it only has a name and a unique identifier. In the example shown in figure 2, we have a user task named `Process sales order`. When you want to delegate a user task in Activiti, it's important that the user task has an owner. In this example, we set the owner to Kermit. Then Kermit can delegate the task to another user, such as Fozzie. At that point, the user task has an assignee named Fozzie and an owner named Kermit. This means that, when you query the Activiti Engine for user tasks assigned to Kermit, you won't retrieve this task anymore. Only Fozzie can work on it.

Then, the most interesting functionality of delegating a user task is executed. Fozzie is ready with the user task and resolves the task. Note that resolving a user task is something else than completing a user task and it's also implemented via another `TaskService` method, as we'll see in listing 3 in a minute. When the user task is resolved, the assignee of the user task will be set to Kermit because he's the one who delegate the task and is the owner. So, the user task will not be completed, only the assignee value will be changed to the value of the user task owner.

Enough talking! Let's look at a unit test example in listing 3, which shows how you can use the task delegation features of the Activiti Engine using the `TaskService` interface.

Listing 3 Unit test example showing the task delegation functionality

```
public class DelegateTaskTest extends AbstractTest {

  @Rule
  public ActivitiRule activitiRule = new ActivitiRule(
      "activiti.cfg-mem.xml");

  @Test
  public void delegateTask() {
    TaskService taskService = activitiRule.getTaskService();
    Task delegateTask = taskService.newTask();
    delegateTask.setOwner("kermit");                                    #1
    taskService.saveTask(delegateTask);
    Task queryTask = taskService.createTaskQuery()
        .singleResult();
    assertEquals("kermit", queryTask.getOwner());
    assertNull(queryTask.getAssignee());
    taskService.delegateTask(                                           #2
        delegateTask.getId(), "fonzie");
    queryTask = taskService.createTaskQuery().singleResult();
    assertEquals("fonzie", queryTask.getAssignee());
    assertEquals(DelegationState.PENDING,
        queryTask.getDelegationState());
    taskService.resolveTask(delegateTask.getId());                      #3
    queryTask = taskService.createTaskQuery().singleResult();
    assertEquals("kermit", queryTask.getAssignee());                    #4
    assertEquals(DelegationState.RESOLVED,
        queryTask.getDelegationState());
    taskService.complete(delegateTask.getId());
    List<HistoricTaskInstance> historicTaskList =
        activitiRule.getHistoryService()
            .createHistoricTaskInstanceQuery()
            .list();
    assertEquals(1, historicTaskList.size());
    for (HistoricTaskInstance historicTaskInstance :
          historicTaskList) {

      assertNotNull(historicTaskInstance.getEndTime());
    }
  }
}
```

#1 Sets the task owner
#2 Delegates the task to Fozzie
#3 Fozzie resolves the task
#4 Kermit becomes the assignee

In this example, we first create a user task and set the task owner to Kermit (#1). Then, we delegate the user task using the `delegateTask` method to Fozzie (#2). Now, the user task has an assignee value of Fozzie and the Activiti Engine also maintains a so-called `DelegationState` value, which is `PENDING` at first. `PENDING` means that the person to whom the task is delegated still has to complete the work.

When Fozzie has finished his work, the user task is resolved using the `resolveTask` method (#3). When this method is invoked, the Activiti Engine processes the delegation logic and sets the assignee to the task owner, which is Kermit in this example (#4). The DelegationState is also changed to `RESOLVED` at that point.

Kermit can now complete the user task when he chooses to. Task delegation is an interesting workflow feature that you can use for adhoc tasks but also for user tasks in process definitions. We already saw that we can use a task listener to implement additional workflow features in a process definition. And that would also work fine for task delegation.

Because task delegation is not implemented in the Activiti Explorer, this completes our delegation discussion. We now move on to implementing the four-eyes principle, which is not supported by the Activiti Engine out of the box.

Implementing the four-eyes principle

A commonly used workflow pattern in process definitions is the four-eyes principle. Imagine you have two user tasks—develop solution and review solution—in a process and both tasks are assigned to a candidate group of users, like for example development. The first user task is claimed and completed by the infamous Kermit, and now Kermit also wants to claim the review user task. The four-eyes principle prohibits Kermit from claiming the review user task because he also has claimed and completed the first user task. The review solution user task must be performed by a second pair of eyes, which explains the name of the four-eyes principle.

The Activiti Engine does not support the four-eyes principle pattern by default, but we can implement this functionality without a lot of coding. What we need is a piece of logic that checks that the person who claims the second user task is not the same person who claimed and completed the first user task. We can implement a task listener that is executed when someone claims the user task. Let's look at the task listener implementation in code listing 4.

Listing 4 Task listener that implements the 4-eyes principle

```
public class FourEyesListener implements TaskListener {

  private FixedValue otherTaskId;
  private FixedValue processEngineName;

  @Override
  public void notify(DelegateTask delegateTask) {
    String name = null;
    if(processEngineName != null && StringUtils.isEmpty(
        processEngineName.getExpressionText()) == false) {

      name = processEngineName.getExpressionText();
    } else {
      name = ProcessEngines.NAME_DEFAULT;                              #1
    }
    ProcessEngine processEngine =
        ProcessEngines.getProcessEngines().get(name);                  #2
    HistoryService historyService =
        processEngine.getHistoryService();
    HistoricTaskInstance historicTask = historyService
        .createHistoricTaskInstanceQuery()
        .processInstanceId(delegateTask.getProcessInstanceId())
        .taskDefinitionKey(otherTaskId.getExpressionText())            #3
        .singleResult();

    if(historicTask == null) {
      throw new ActivitiException("The previous task " +
          otherTaskId.getExpressionText() +
              " could not be found");
    }

    String claimer = delegateTask.getAssignee();
    String previousAssigneee = historicTask.getAssignee();

    if(claimer.equalsIgnoreCase(previousAssigneee)) {                  #4
      throw new ActivitiException("Assignee of task " +
          otherTaskId.getExpressionText() +
              " is not allowed to claim this task");
    }
  }

  public void setOtherTaskId(FixedValue otherTaskId) {
    this.otherTaskId = otherTaskId;
  }

  public void setProcessEngineName(
      FixedValue processEngineName) {
    this.processEngineName = processEngineName;
  }
}
```

#1 Default process engine name
#2 Retrieves the process engine
#3 Retrieves the other task instance
#4 Checks if the assignee is not the same

The task listener implementation is developed to be directly reusable in your process definitions. Therefore, the process engine name can be overridden in the task listener configuration in the process definition. Otherwise, the default value (#1) is used to retrieve the process engine from the cache (#2). When a process engine is created in the Activiti Explorer or by using the Java API, it's registered in the cache of the `ProcessEngines` singleton.

With the process engine instance available we can implement the four-eyes principle logic. First, we have to retrieve the previous user task for which we have to validate the four eyes. The task identifier of the previous user task needs to be set as a field property in the Activiti listener configuration we'll see in code listing 5. Then, this task identifier is used to retrieve the user task via the `HistoryService` (#3).

Then, we can validate if the person who tries to claim the second user task is not the same as the assignee of the first user task (#4). If the values are the same, an `ActivitiException` is thrown and the user task will not be claimed.

To show how we can use this newly created task listener, listing 5 shows a small process definition example that has configured this task listener on the second user task.

Listing 5 Example process definition using the four-eyes task listener

```
<definitions xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"
    xmlns:activiti="http://activiti.org/bpmn"
    targetNamespace="http://www.bpmnwithactiviti.org/foureyes">

  <process id="fourEyesProcess">
    <startEvent id="theStart" />
    <sequenceFlow sourceRef="theStart"
        targetRef="firstTask" />
    <userTask id="firstTask"
        activiti:candidateGroups="sales" />                          #1
    <sequenceFlow sourceRef="firstTask"
        targetRef="secondTask" />
    <userTask id="secondTask"
        activiti:candidateGroups="sales">
      <extensionElements>
        <activiti:taskListener
            class="org.bpmnwithactiviti.chapter10.                   #2
                [CA]foureyes.FourEyesListener"                       #2
            event="assignment">
          <activiti:field name="otherTaskId"
              stringValue="firstTask" />                             #3
        </activiti:taskListener>
      </extensionElements>
    </userTask>
    <sequenceFlow sourceRef="secondTask"
        targetRef="theEnd" />
    <endEvent id="theEnd" />
  </process>
</definitions>
```

#1 Assigned to candidate group sales
#2 Configures the four-eyes task listener
#3 Reference to the first user task

The process definition consists of two user tasks directly connected via a sequence flow. Both user tasks have the sales group as their candidate group definition (#1). The second user task where we want to validate the four-eyes principle has a task listener configured that corresponds to our implementation of code listing 4. The `otherTaskId` field property is set to reference the first user task (#3).

With the task listener implementation and process definition in place, we can now create a unit test to be able to test our solution easily. Because we want to use the default process engine configuration we need to initialize the process engine ourselves, without the help of the Activiti test classes. The `ActivitiRule` test class initializes the process engine without registering it in the `ProcessEngines` cache. Let's look at the unit test implementation in code listing 6.

Listing 6 Testing the four-eyes task listener implementation

```
public class FourEyesTest extends AbstractTest {

  @Test
  public void validateFourEyes() {
    ProcessEngine processEngine = ProcessEngineConfiguration
        .createProcessEngineConfigurationFromResource(
            "activiti.cfg-mem.xml")
        .setProcessEngineName(ProcessEngines.NAME_DEFAULT)              #1
        .buildProcessEngine();

    processEngine.getRepositoryService().createDeployment()
        .addClasspathResource(
            "chapter10/foureyes/fourEyes.bpmn20.xml")                   #2
        .name("fourEyes")
        .deploy();

    processEngine.getRuntimeService()
        .startProcessInstanceByKey("fourEyesProcess");

    TaskService taskService = processEngine.getTaskService();
    Task firstTask = taskService
        .createTaskQuery()
        .singleResult();
    taskService.claim(firstTask.getId(), "kermit");
    taskService.complete(firstTask.getId());                            #3

    Task secondTask = taskService
        .createTaskQuery()
        .singleResult();

    try {
      taskService.claim(secondTask.getId(), "kermit");
      fail("Expected claim error");                                     #4
    } catch(ActivitiException e) {
      // claim error expected
    }

    secondTask = taskService
        .createTaskQuery()
        .taskId(secondTask.getId())
        .singleResult();
    assertNull(secondTask.getAssignee());

    taskService.claim(secondTask.getId(), "fozzie");                    #5
    taskService.complete(secondTask.getId());
  }
}
```

#1 Creates a default process engine
#2 Deploys the 4-eyes process definition
#3 Completes the first user task
#4 Kermit is not allowed to claim
#5 Fozzie can claim the user task

Because we can't use the `ActivitiRule` test support class, we build the process engine with a default process name ourselves (#1). Then, we deploy the process definition we discussed in code listing 5 to the process engine (#2).

When the process instance is started, the first user task is created and claimed and completed by Kermit (#3). The process instance will now create the second user task. We first try to claim the user task with the Kermit user. But because Kermit also has claimed and completed the first user task, our four-eyes task listener will throw an `ActivitiException` (#4). Then, we claim and complete the user task using the user Fozzie because he's allowed to claim the user task (#5).

Task listeners provide nice integration points in the Activiti Engine and user tasks in particular, which helps to implement workflow patterns without a lot of additional coding. The four-eyes principle is just an example to shows how to do this.

Summary

Workflow is an important part of developing process applications. In a lot of cases, not all tasks in a process can be executed automatically by invoking web services or other external resources. For example, we need people to provide information needed in the process and to review and approve specific parts of the process execution.

We introduced you to subtasks and task delegation that's supported via the Activiti API and partially via the Activiti Explorer. Then, we saw that it's not hard to implement a workflow pattern like the four-eyes principle using a task listener.

Here are some other Manning titles you might be interested in:

Open-Source ESBs in Action
Tijs Rademakers and Jos Dirksen

Open Source SOA
Jeff Davis

Mule in Action
David Dossot and John D'Emic

Last updated: November 28, 2011