Bonita You'll never work alone

BONITA – Workflow patterns support

Abstract

This document describes how the Bonita workflow implements the 20 workflow patterns defined by X.Van der Halst. Each pattern is described with an example showing how to implement the pattern using Bonita workflow.

Christophe Loridan
Jordi Anguela Rosell

BULL R&D

ObjectWeb Open Source Middleware

CHANGES RECORD		
RÉFÉRENCES	DATE	CHANGE
1.0	August, 2004	First version
1.1	April, 2006	Revision

INDEX

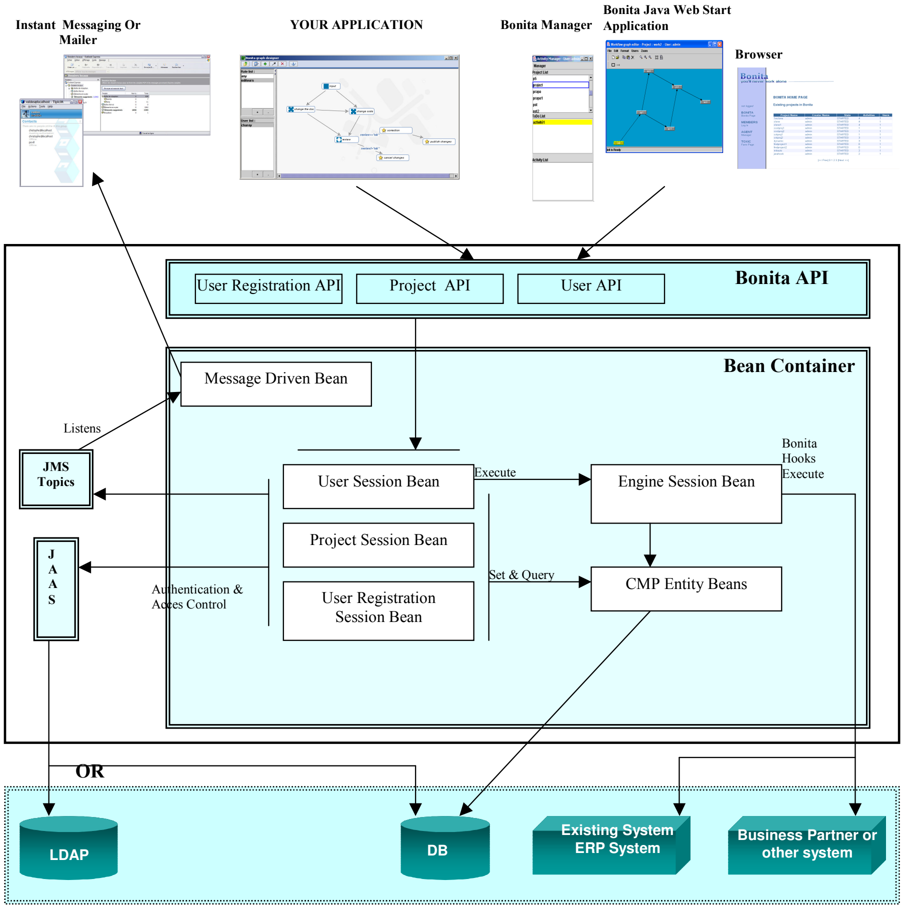
1 INTRODUCTION 4
2 WORKFLOW PATTERNS 6
2.1 Basic Control Flow Patterns 6
2.1.1 Pattern 1, Sequence 6
2.1.2 Pattern 2, Parallel Split 7
2.1.3 Pattern 3, Synchronization 8
2.1.4 Pattern 4, Exclusive choice 9
2.1.5 Pattern 5, Simple Merge 10
2.2 Advanced Branching and Synchronization Patterns 11
2.2.1 Pattern 6, Multi Choice 11
2.2.2 Pattern 7, Synchronizing Merge 12
2.2.3 Pattern 8, Multi-merge 13
2.2.4 Pattern 9, Discriminator 14
2.2.5 Pattern 9a, N-out-of-M Join 15
2.3 Structural Patterns 16
2.3.1 Pattern 10, Arbitrary cycles 16
2.3.2 Pattern 11, Implicit Termination 19
2.4 Patterns involving Multiple Instances 20
2.4.1 Pattern 12, Multiple instances without synchronization 20
2.4.2 Pattern 13, Multiple instances with a Priori Design Time knowledge 21
2.4.3 Pattern 14, Multiple instances with a Priori Run Time knowledge 22
2.4.4 Pattern 15, Multiple instances without a Priori Run Time knowledge 23
2.5 State-based Patterns 24
2.5.1 Pattern 16, Deferred Choice 24
2.5.2 Pattern 17, Interleaved Parallel Routing 26
2.5.3 Pattern 18, Milestone 28
2.6 Cancellation Patterns 30
2.6.1 Pattern 19, Cancel Activity 30
2.6.2 Pattern 20, Cancel Case 31
3 NEW WORKFLOW PATTERNS 32
3.1.1 Pattern Extra 1, Explicit Termination 32
3.1.2 Pattern Extra 2, Cancel Path 33
4 HOW TO RUN THE TESTS 34
5 APENDIX: LEGEND 35

1 INTRODUCTION

BONITA is a workflow system featuring a lot of innovative features like activities that can start in anticipation, awareness infrastructure allowing users to be notified of any events occurring during the execution in a given process , or automatic activation of user's code according to a defined activity life cycle. Traditional workflow features like dynamic user/roles resolution, activity performer and sequential execution are also included in Bonita to support both cooperative and administrative workflow processes.

BONITA is a fully conformant J2EE application, taking advantage of the power and robustness of the J2EE platform. The BONITA API is accessible either thru EJB's.

Processes are created using a graphical definition tool or by using the Project interface API. A process is defined as a set of activities and an associated execution model. The enactment engine takes care of scheduling the activities according to the defined execution model. The User API provides full control over the execution of the process, for example allowing starting or stopping an activity. BONITA supports also dynamic modification of an existing process, that is the Project interface can be used against a running process.



- The **User Registration Session bean** provide the interface for :
    - User creation and management
    - Group creation
- The **Project Session Bean** provides the interface for :
    - Creation of the process
    - Definition of nodes and edges
    - Modifications of properties
- The **User Session Bean** implements commands and queries related to
    - Projects of a user
    - Todo List
    - Executing activities
    - Start/terminate/Cancel commands
- The **Engine Bean** is a special session bean that implements the state machine and controls Process execution. It is not part of the API.
- Each method call in the Bonita API involving a state modification of the workflow system is registered into a JMS Topic. Depending on user preferences (defined while user creation), the **Message Driven Bean** notifies the user either using Instant Messaging services, either Traditional Mailer.

Bonita Hooks can access existing systems in the SI (Erp or whatever else), or Business partner systems using JCA or Web services.

Both User and project APIs are available either as Session Bean, or as web services.

Note: for a detailed insight into the Bonita workflow features please refer to the Bonita API document: http://bonita.objectweb.org/html/Documentation/docs/bonitaAPI.pdf

2 WORKFLOW PATTERNS

All these patterns can be found at: **www.workflowpatterns.com**

2.1 Basic Control Flow Patterns

2.1.1 Pattern 1, Sequence

Description

An activity in a workflow process is enabled after the completion of another activity in the same process.

Synonyms

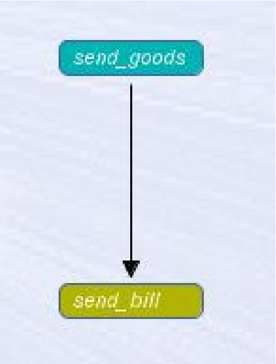
Sequential routing or serial routing.

Examples

- Activity send_bill is executed after the execution of activity send_goods.
- An insurance claim is evaluated after the client's file is retrieved.
- Activity add_air_miles is executed after the execution of activity book_flight.

Support 🟢

This pattern is supported by Bonita.



In this example, the activities *send_goods* and *send_bill* are linked by an unconditional arrow expressing an outgoing transition from *send_goods* to *send_bill*. In this example, we have execute the activity *send_goods* so its state is *terminated* (represented in cyan), meaning its execution has finished. The activity *send_bill* is in *ready* state (represented in yellow) meaning it can be started immediately because the previous activity has already finished.

2.1.2 Pattern 2, Parallel Split

Description

A point in the workflow process where a single thread of control splits into multiple threads of control which can be executed in parallel, thus allowing activities to be executed simultaneously or in any order.

Synonyms

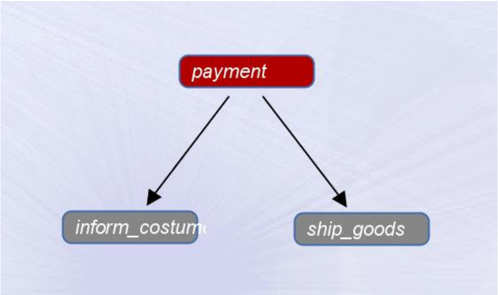
AND-split, parallel routing or fork.

Examples

- The execution of the activity payment enables the execution of the activities ship_goods and inform_customer.
- After registering an insurance claim two parallel subprocesses are triggered: one for checking the policy of the customer and one for assessing the actual damage.

Support ●

This pattern is supported by Bonita.



In this example, the activities *inform_costumer* and *ship_goods* can be executed in parallel. They are both in *initial* state (represented in grey). Note that there is no need to define a specific routing node to use this pattern. The activity *payment* is a regular activity that has two outgoing transitions without conditions.

2.1.3 Pattern 3, Synchronization

Description

A point in the workflow process where multiple parallel sub processes/activities converge into one single thread of control, thus synchronizing multiple threads. It is an assumption of this pattern that each incoming branch of a synchronizer is executed only once (if this is not the case, then see Patterns 13-15 (Multiple Instances Requiring Synchronization)).

Synonyms

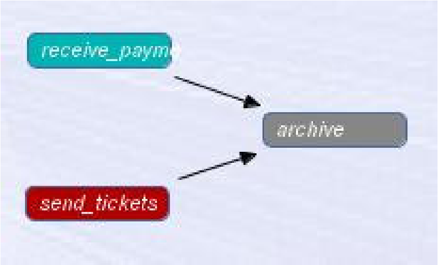
AND-join, rendezvous or synchronizer.

Examples

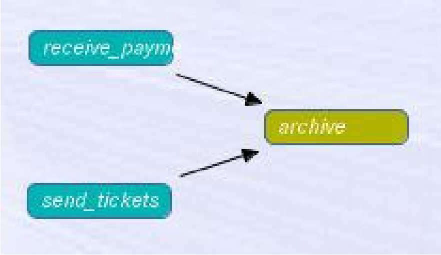
- Activity archive is enabled after the completion of both activity send_tickets and activity receive_payment.
- Insurance claims are evaluated after the policy has been checked and the actual damage has been assessed.

Support ●

This pattern is supported by Bonita.



In this example, the activity *receive_payment* is in *terminated* state, while activity *sent_tickets* is in *executing* state. The activity *archive* is in state *initial* (represented in grey). This activity will not be allowed to start until *receive_payment* and *sent_tickets* are both terminated. Again, note that *archive* is not a specific routing node but a regular activity that also acts as an implicit AND-join.



When both activities are terminated the state of *archive* activity changes to *ready* (represented in yellow).

2.1.4 Pattern 4, Exclusive choice

Description

A point in the workflow process where, based on a decision or workflow control data, one of several branches is chosen.

Synonyms

XOR-split, conditional routing, switch or decision.

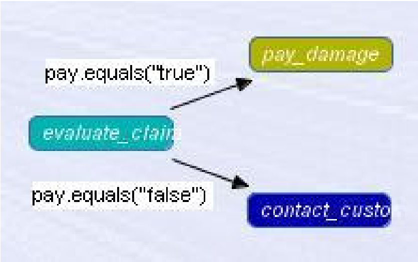
Examples

- Activity evaluate_claim is followed by either pay_damage or contact_customer.

- Based on the workload, a processed tax declaration is either checked using a simple administrative procedure or is thoroughly evaluated by a senior employee.

Support ●

This pattern is supported by Bonita.



The activity *evaluate_claim* has two outgoing transitions. They carry mutually exclusive conditions, therefore allowing the exclusiveness of choice. In the example, the exclusive choice has been made: *evaluate_claim* is in state *terminated*, *contact_costumer* is in state *dead* (because the corresponding condition evaluated to false), while *pay_damage* is *ready* to be started.

Let’s get some basic insight how conditions work in Bonita. A condition is related to the process control data. Control data are associated to activities and are called properties. Conditions are expressed in java. Two scopes for the properties are defined in Bonita workflow: project properties that are accessible from all activities inside the project and node properties, where the data stored in the property is only accessible from a concrete node.

In the example above, there is a node property named *pay*. It is associated to the activity *evaluate_claim* and has the value *“true”*. The transition between *evaluate_claim* and *pay_damage* has a condition expressed as *pay.equals(“true*”*)*, while the transition between *evaluate_claim* and *contact_costumer* has a condition expressed as *pay.equals(“false”).* This configuration allows the exclusiveness of choice between those activities.

2.1.5 Pattern 5, Simple Merge

Description

A point in the workflow process where two or more alternative branches come together without synchronization. It is an assumption of this pattern that none of the alternative branches is ever executed in parallel (if this is not the case, then see Pattern 8 (Multi-merge) or Pattern 9 (Discriminator)).

Synonyms

XOR-join, asynchronous join or merge.

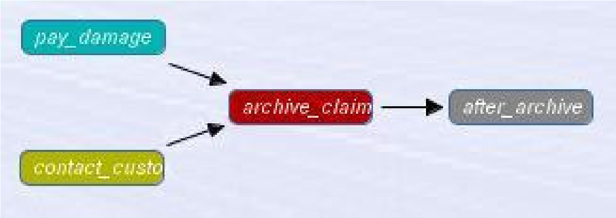
Examples

- Activity archive_claim is enabled after either pay_damage or contact_customer is executed.

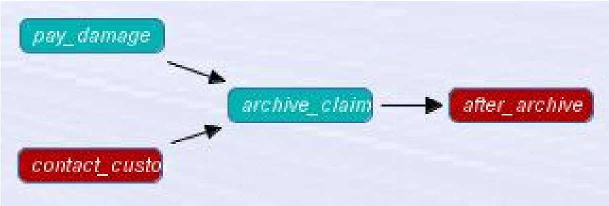
- After the payment is received or the credit is granted the car is delivered to the customer.

Support 🟢

This pattern is supported by Bonita.



To select a path between different options Bonita uses OR-JOIN activities. A new activity of this type called *archive_claim* is added to do this merge. We can start *archive_claim* activity because *pay_damage* has already finished.



If *contact_costumer* activity is executed at this point, its termination will not trigger the execution of *archive_claim* activity again.

2.2 Advanced Branching and Synchronization Patterns

2.2.1 Pattern 6, Multi Choice

Description

A point in the workflow process where, based on a decision or workflow control data, a number of branches are chosen.

Synonyms

Conditional routing, selection, OR-split.

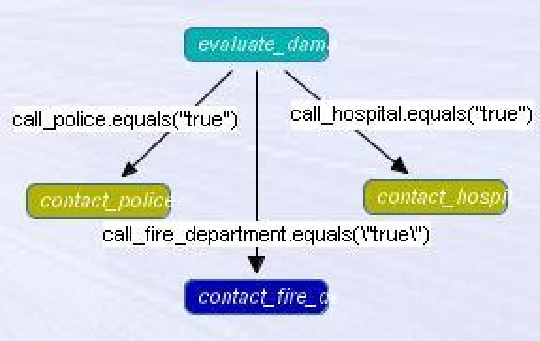
Examples

- After executing the activity evaluate_damage the activity contact_fire_department or the activity contact_insurance_company is executed. At least one of these activities is executed. However, it is also possible that both need to be executed.

Support ●

This pattern is supported by BONITA.

The way to implement this pattern is specifying conditions on the transitions. If more than one condition is true when they are evaluated then more than path is taken.


evaluate_dam
call_police.equals("true")
call_hospital.equals("true")
contact_police
contact_hospi
call_fire_department.equals(\"true\")
contact_fire_c

The *evaluate_damage* activity has three properties named *call_fire_department*, *call_police* and *call_hospital.*

Transitions from *evaluate_damage* to *contact_police, contact_fire_department*, and *contact_hospital* have a condition expression related these properties.

Only *call_police* and *call_hospital* node properties are set to "true". Because *evaluate_damage* is in *terminated* state, both conditions related to *contact_police* and *contact_hospital* have succeeded: these activities are ready to be executed. The condition related to *contact_fire_departement* has failed. This activity is in *dead* state.

2.2.2 Pattern 7, Synchronizing Merge

Description

A point in the process where multiple paths converge into one single thread.

- If more than one path is taken, synchronization should occur.
- If only one path is taken, the alternative branches should converge without synchronization. Also known as Synchronizing join.

It is an assumption of this pattern that a branch that has already been activated, cannot be activated again while the merge is still waiting for other branches to complete.

Synonyms

Synchronizing join.

Examples

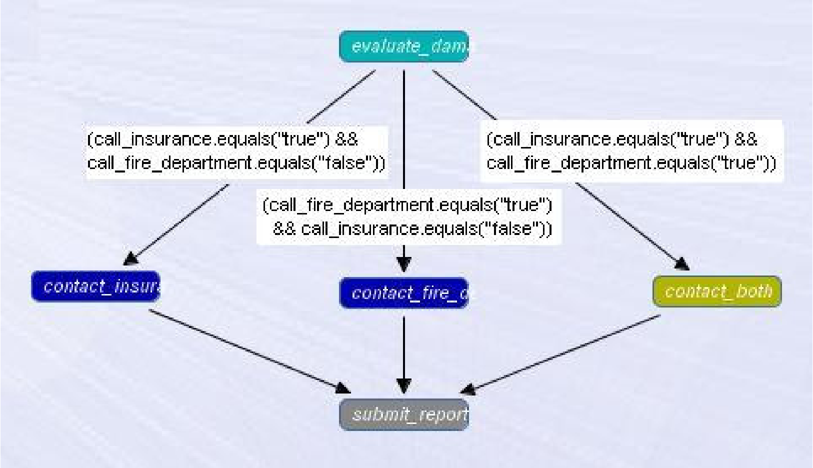
- Extending the example of Pattern 6 (Multi-choice), after either or both of the activities contact_fire_department and contact_insurance_company have been completed (depending on whether they were executed at all), the activity submit_report needs to be performed (exactly once).

Support

This pattern is partially supported by Bonita.

AND-JOIN is not suitable here because some branches could not be taken. OR-JOIN is not suitable either, the first branch that finishes the execution is considered and there is not synchronization with other executing branches.

An implementation of this pattern with Bonita workflow is possible only if we have a simple case. Let's see an example with where we have two possible paths:


evaluate_dam
(call_insurance.equals("true") && call_fire_department.equals("false"))
(call_insurance.equals("true") && call_fire_department.equals("true"))
(call_fire_department.equals("true") && call_insurance.equals("false"))
contact_insur
contact_fire_d
contact_both
submit_report

The idea is to add an extra activity (*contact_both*) that executes both *contact_insurance* and *contact_fire_department* activities. *evaluate_damage* is implemented as an exclusive choice (see pattern 4) selecting one of the paths.

This solution is not suitable when we have lots of possible paths.

2.2.3 Pattern 8, Multi-merge

Description

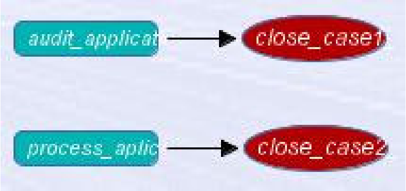
A point in a workflow process where two or more branches reconverge without synchronization. If more than one branch gets activated, possibly concurrently, the activity following the merge is started for every activation of every incoming branch.

Examples

- Sometimes two or more parallel branches share the same ending. Instead of replicating this (potentially complicated) process for every branch, a multi-merge can be used. A simple example of this would be two activities audit_application and process_application running in parallel which should both be followed by an activity close_case.

Support ●

This pattern is supported by Bonita. The purpose “sharing of activity definition” of this pattern can however be achieved thanks to the concept of Bonita sub process.



In this example, *close_case1* and *close_case2* are two activities that are mapped on the sub process *close_case* (not shown here). The definition of *close_case* sub process can be as complex as needed.

In this example, assuming that the transitions do not hold any condition, one instance of *close_case* will be created after each completion of *audit_application* and *process_aplication* activities.

The use of activity properties can be very useful to configure the parameters of each sub process.

After the sub processes execution, we could add another AND-JOIN Bonita activity to synchronize both.

An asynchronous behavior can also be achieved, thanks to the use of Bonita Hooks

2.2.4 Pattern 9, Discriminator

Description

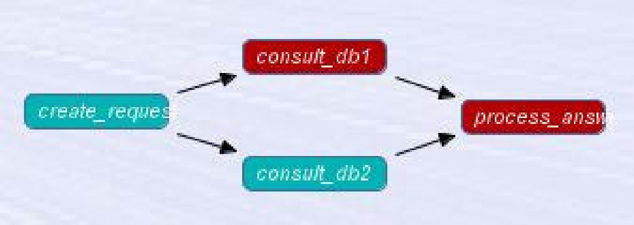
The discriminator is a point in a workflow process that waits for one of the incoming branches to complete before activating the subsequent activity. From that moment on it waits for all remaining branches to complete and "ignores" them. Once all incoming branches have been triggered, it resets itself so that it can be triggered again (which is important otherwise it could not really be used in the context of a loop).

Examples

- To improve query response time, a complex search is sent to two different databases over the Internet. The first one that comes up with the result should proceed the flow. The second result is ignored.

Support ●

This pattern is supported by Bonita.



The behavior of this pattern is very similar to Simple Merge pattern. As seen before, an activity with type OR-JOIN will trigger the execution of downwards activities only once. Nevertheless, Bonita engine will not wait for other branches to complete: it will just process the termination Hooks of the other branches as they occur.

2.2.5 Pattern 9a, N-out-of-M Join

Description

N-out-of-M Join is a point in a workflow process where *M* parallel paths converge into one. The subsequent activity should be activated once *N* paths have completed. Completion of all remaining paths should be ignored. Similarly to the discriminator, once all incoming branches have "fired", the join resets itself so that it can fire again.

Examples

- A paper needs to be sent to three external reviewers. Upon receiving two reviews the paper can be processed. The third review can be ignored.

Support ●

This pattern is not supported in Bonita workflow.

2.3 Structural Patterns

2.3.1 Pattern 10, Arbitrary cycles

Description

A point in a workflow process where one or more activities can be done repeatedly.

There are two types of cycles: structured and non structured.

- A *structured cycle* has at maximum one entry and/or one exit point.

- A *non structured cycle* may have several entry points, and/or several exit points from the cycle, in another words, it doesn’t have a predefined entry and exit points.

Synonyms

Loop, iteration or cycle.

Examples

- A classical example to see the use of iterations is an approval process where an user request a demand that has be approved for another person. If the demand is approved then it is processed, if not, then the flow is redirect to the first activity again to able to change the request.

Support 🟢

Bonita supports both structured and non structured cycles.

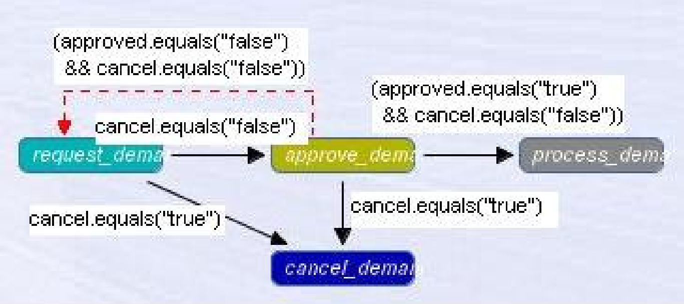
Bonita has some rules when a process has iterations to guarantee its correct behavior:

Premise: it is not possible to continue the execution inside an iteration and leave it at the same time.

1. Only one iteration is allowed between two connected nodes
2. It’s possible to have more than one iteration starting in the same node
3. All transitions leaving from a node that starts an iteration must have a condition.
   If there is more than one transition leaving from that node, all of them must have a condition.
4. If we have an extra exit point from iteration it is strictly necessary to have conditions in all transitions leaving from that node.
   These conditions have to be mutually exclusive in order to take a path to continue iterating or to exit form the iteration.

Extra exit point example

Let's see how to implement these constraints in our example. In this case we have an approval process with and extra exit point from the iteration: *cancel_demand* activity.



In the figure beside, there is an iteration from *approve_demand* to *request_demand* (represented with a red arrow added to the original picture). This cycle enables the possibility to repeat the request with different parameters.

Three project properties are used in this example:

- *approved* property: indicates if we have to leave the iteration through *process_demain* activity or if we have to return to *request_demand.*
- *cancel* property: used to indicate that the request has been canceled and *cancel_demand* activity will be executed.
- *iterations* property: indicates the number of iterations that we will do.

Transitions conditions are set to accomplish the constraints:

- Transition from *request_demand* to *approve_demand* is: *cancel.equals("false")*
- Transition from *request_demand* to *cancel_demand* is: *cancel.equals("true")*
- Transition from approve*_demand* to *process_demand* is:
  approved.equals("true") && cancel.equals("false")
- Transition from approve*_demand* to *process_demand* is: *cancel.equals("true")*

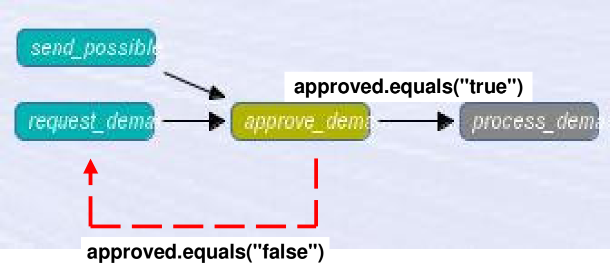
These conditions are necessary to fulfill constrains 3 and 4. Also, all iterations in Bonita must have a condition in order to indicate when it has to iterate. This condition, like the properties, is a java expression. In this example iteration's condition is:

approved.equals("false") && cancel.equals("false")

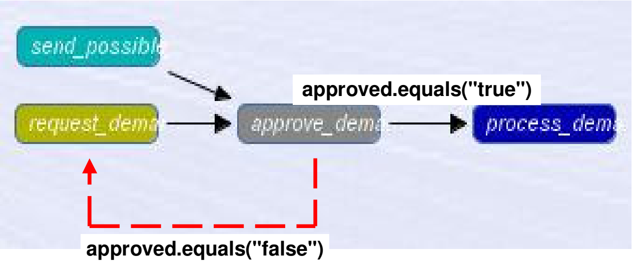
This basic example is configured to iterate one time and after that *approved* property is set to leave the iteration and enabling the execution of *process_demand* activity.

Extra entry point example

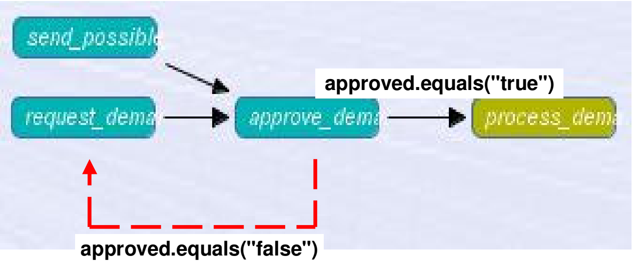
Now let's see another example to see an iteration with an extra entry point. In this case we have the same approval process but now there is an extra activity that sends the possible options to *approve_demand* activity.



send_possible_options activity acts as extra entry point into the iteration. *approve_demand* is an AND-JOIN activity so it will not start until both *request_demand* and *send_possible_options* are terminated.



After iterating, *request_demand* becomes ready again, *approve_demand* is set initial state and *process_demand* is momentarily canceled because the transition condition fails (it iterates instead of leaving the iteration).



At the end, when *approved* property is set to true it leaves the iteration setting the state of *process_demand* to *ready.*

For more information related with iterations read Bonita documentation

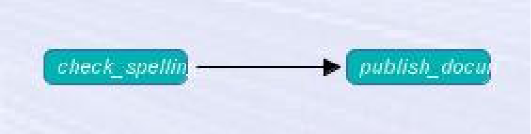
2.3.2 Pattern 11, Implicit Termination

Description

A given process should be terminated when there is nothing left to be done. In the contrary, some workflow engines use an explicit *Final* node: the process terminates only once this node is reached.

Support ●

Bonita supports the implicit termination pattern.



This is true for both processes and sub-processes.

In this example both activities are terminated so the process is also terminated because there is nothing else to be done.

Bonita supports run-time dynamic modification of a given process instance: in that sense, there is always potentially something more to be done! Then, an explicit call to the API must be used to formally terminate the process once all activities are completed.

2.4 Patterns involving Multiple Instances

2.4.1 Pattern 12, Multiple instances without synchronization

Description

Within the context of a single case, multiple instances can be created: a spawn-off facility allows creating several threads of control which don't need to be synchronized.

Synonyms

Threading without synchronization, spawn off facility.

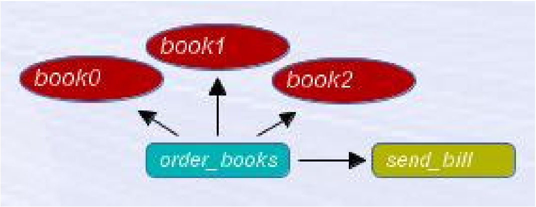
Examples

- A customer ordering a book from an electronic bookstore such as Amazon may order multiple books at the same time. Many of the activities (e.g., billing, updating customer records, etc.) occur at the level of the order. However, within the order multiple instances need to be created to handle the activities related to one individual book (e.g., update stock levels, shipment, etc.). If the activities at the book level do not need to be synchronized, this pattern can be used.

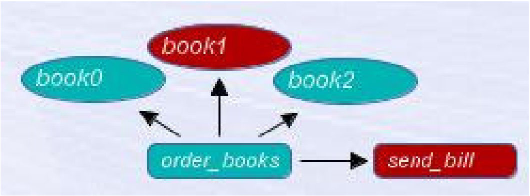
Support ●

Bonita supports multiple instances without synchronization pattern.

This is achieved thru the use of java Hooks. Hooks are java codelets executed according to a defined activity life cycle. By using the Bonita Project API, one can asynchronously launch a new process instance from the context of one activity.



In this example, *order_books* activity instantiates three times *book* sub process during the execution of its hook. These sub processes are independent of *send_bill* activity and between them.



Here we can see that *book0* and *book2* have been executed (colored in cyan). *book1* and *send_bill* activities are being executed at the same time.

2.4.2 Pattern 13, Multiple instances with a Priori Design Time knowledge

Description

For one process instance, an activity is enabled multiple times. In this usage scenario, the number of instances to be created is known at design time.

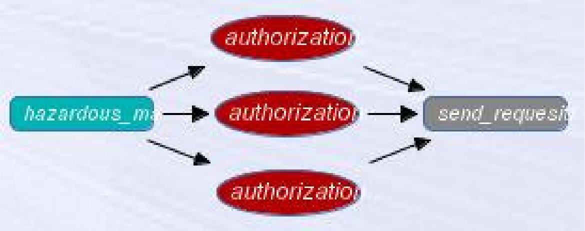
Examples

- The requisition of hazardous material requires three different authorizations.

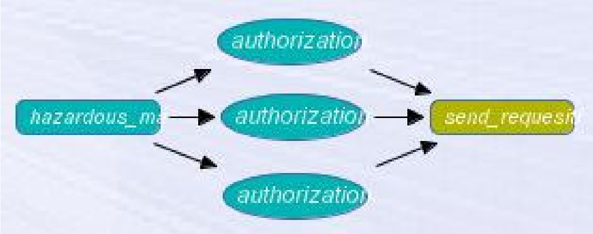
Support ●

Bonita supports multiple instances with a priori design time knowledge.

The idea is to follow the same process that in the previous pattern. The difference with the previous pattern is that know we have another activity (Bonita AND-JOIN activity) that synchronizes the sub processes defined in the model. Let's see and example:



hazardous_materials_requisition activity will instantiate as many times as defined *authorization* sub processes. After that, *send_requisition* AND-JOIN activity will synchronize the sub processes.



When all authorizations are accepted *send_requisition* activity becomes ready to be executed.

2.4.3 Pattern 14, Multiple instances with a Priori Run Time knowledge

Description

For one process instance, an activity is enabled multiple times. In this usage scenario, the number of instances to be created is known only at run time, before the instances of that activity need to be created. After all instances are completed, some other activity needs to be started.

Examples

- In the review process of a scientific paper submitted to a journal, the activity review_paper is instantiated several times depending on the content of the paper, the availability of referees, and the credentials of the authors. Only if all reviews have been returned, processing is continued.

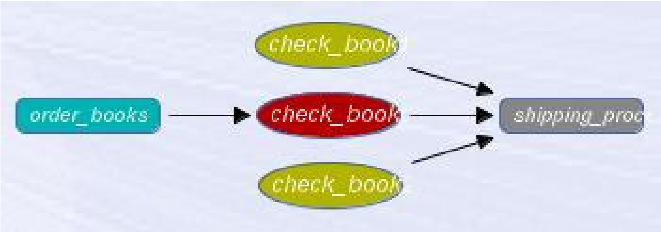
- For the processing of an order for multiple books, the activity check_availability is executed for each individual book. The shipping process starts if the availability of each book has been checked.

- When booking a trip, the activity book_flight is executed multiple times if the trip involves multiple flights. Once all bookings are made, the invoice is to be sent to the client.

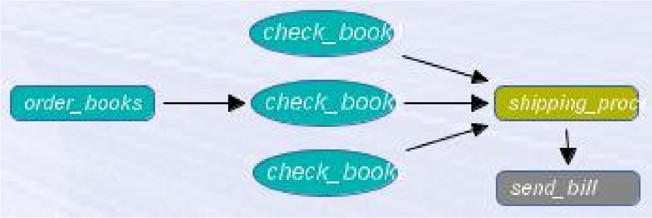
- When authorizing a requisition with multiple items, each item has to be authorized individually by different workflow users. Processing continues if all items have been handled.

Support ●

Bonita supports multiple instances with a priori run time knowledge.
This support is provided thanks to two Bonita features: java Hooks and ability to dynamically modify (e.g. at run time) the definition of a process instance. For a correct design, a sub process instance is defined in the model an the other replicas are created during the runtime.



When *check_book* activity defined in the model is executed a *beforeStart* Hook is lunched and creates the other sub processes as many times as needed.



When all sub process have finished, *shipping_process* (an AND-JOIN activity) synchronize them an continues the execution.

2.4.4 Pattern 15, Multiple instances without a Priori Run Time knowledge

Description

For one process instance, an activity is enabled multiple times. In this usage scenario, the number of instances to be created is known only at run time. The difference with the previous pattern is that new instances of that activity may need to be created even after some of them are already executing (or are terminated). After all instances are completed, some other activity needs to be started.

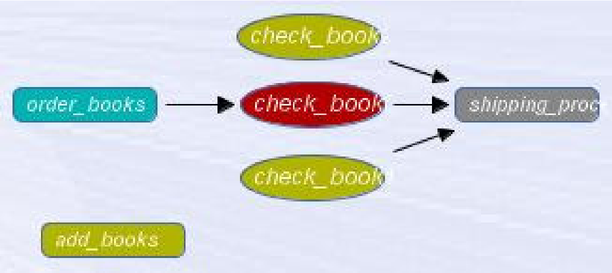
Examples

- For the processing of an insurance claim, zero or more eyewitness reports should be handled. The number of eyewitness reports may vary. Even when processing eyewitness reports for a given insurance claim, new eyewitnesses may surface and the number of instances may change.

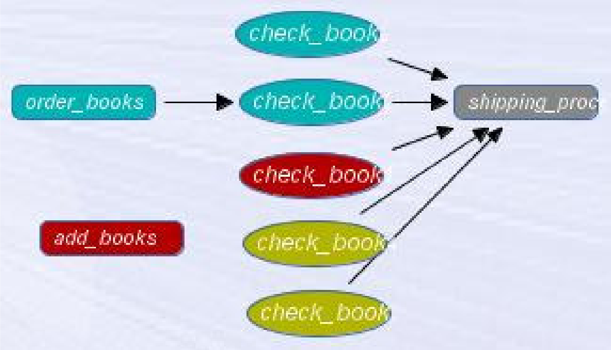
Support ●

Bonita supports multiple instances without a priori run time knowledge.
The implementation of this pattern is based on the previous one and with the Bonita feature that permits to create dynamically more sub process instances during the runtime.

To see and example will suppose the same scenario as in the previous example but now we will have an extra activity that will create more sub processes when it is executed.



This time we have an activity called *add_books* that creates two new sub processes when it is executed. The creation of the new sub processes can be done at any time between *order_books* termination and before starting *shipping_process*.



In this picture we can see that *add_books* activity is executing (coloured in red) and two more sub processes have been added while the other activities are being executed.

2.5 State-based Patterns

2.5.1 Pattern 16, Deferred Choice

Description

A point where one of several branches are chosen. The choice of the branch to execute must not be based on a pre defined criteria, but rather it should be deferred as lately as possible, that is delayed until the processing of one of the alternative branch is started

Synonyms

External choice, implicit choice or deferred XOR-split.

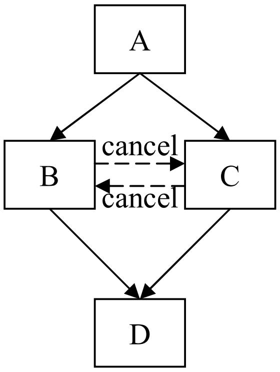
Examples

- At certain points during the processing of insurance claims, quality assurance audits are undertaken at random by a unit external to those processing the claim. The occurrence of an audit depends on the availability of resources to undertake the audit, and not on any knowledge related to the insurance claim. Deferred Choices can be used at points where an audit might be undertaken. The choice is then between the audit and the next activity in the processing chain. The audit activity triggers the next activity to preserve the processing chain.

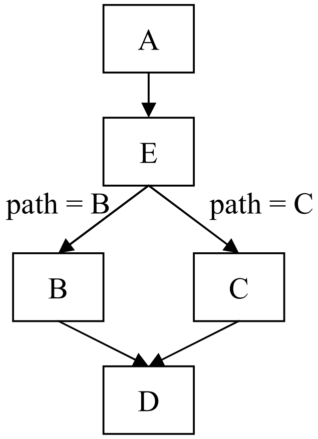
- Business trips require approval before being booked. There are two ways to approve a task. Either the department head approves the trip (activity A1) or both the project manager (activity A21) and the financial manager (activity A22) approve the trip. The latter two activities are executed sequentially and the choice between A1 on the one hand and A21 and A22 on the other hand is implicit, i.e., at the same time both activity A1 and activity A21 are offered to the department head and project manager respectively. The moment one of these activities is selected, the other one disappears.

Support ●

Bonita supports the deferred choice pattern.

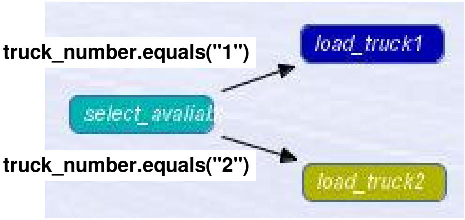


A possible strategy to implement this pattern is to split the flow in two activities B and C where they wait for a user action. When B or C is started then a hook is lunched to deactivate the other sister activity.This solution does not always work because B and C can be selected/executed at same time.



Another solution purposed is to create a new activity E receiving all triggers that active the alternative branches. Then, activity E takes the decision of which path has to be chosen using XOR-split.

Let's see a Bonita example:



In this case we have an activity called *select_avaliable_truck* that decides which truck is available. This decision is taken inside a Hook during the activity execution. Once the decision has been taken a property called *truck_number* is set to take a specific transition. Each transition has a condition related with this property that enables to take one path or another one.

2.5.2 Pattern 17, Interleaved Parallel Routing

Description

A set of activities is executed in an arbitrary order: Each activity in the set is executed, the order is decided at run-time, and no two activities are executed at the same moment (i.e. no two activities are active for the same workflow instance at the same time).

Synonyms

Unordered sequence.

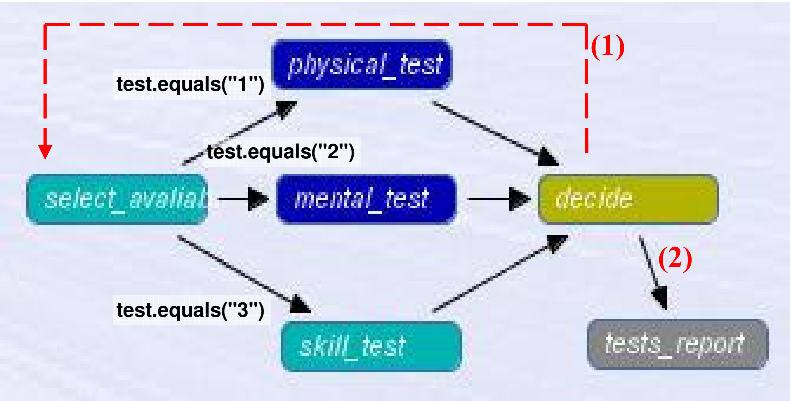
Examples

- The Navy requires every job applicant to take two tests: physical_test and mental_test. These tests can be conducted in any order but not at the same time.
- At the end of each year, a bank executes two activities for each account: add_interest and charge_credit_card_costs. These activities can be executed in any order. However, since they both update the account, they cannot be executed at the same time.

Support ●

Bonita supports the interleaved parallel routing pattern.

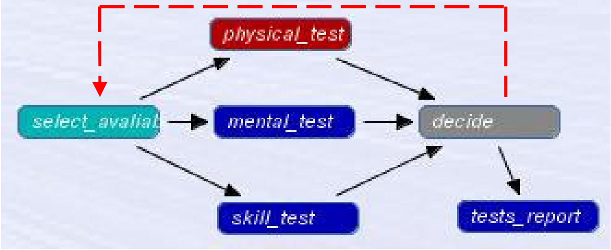
The main idea of this solution is to have an activity that implements a deferred choice (see pattern 16) between the activities that we want to execute and another activity that iterates until all activities have been done.



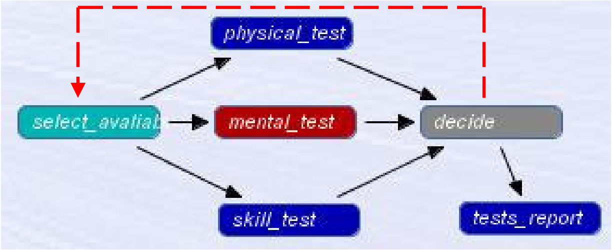
(1) (physical_test.equals("not_passed")
|| mental_test.equals("not_passed")
|| skill_test.equals("not_passed"))

(2) (physical_test.equals("passed")
&& mental_test.equals("passed")
&& skill_test.equals("passed"))

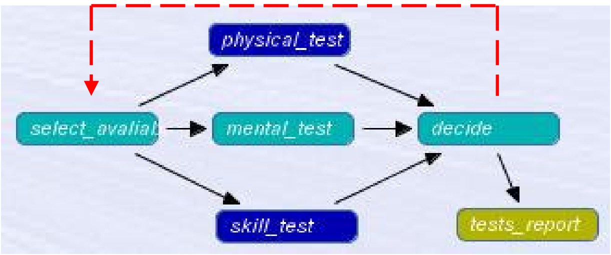
In this example, *select_avaliable_test* activity implements deferred the choice pattern selecting one of the three tests. In this case, *skill_test* has been executed and *decide* activity is ready to be executed. This activity will iterate until the three activities are done. Bonita properties are used to control which activities have been done.



After the first iteration a different activity is selected.



After the second iteration the last activity is chosen.



When all activities have been executed it will not iterate again and next activity (*tests_report*) becomes ready to executed.

2.5.3 Pattern 18, Milestone

Description

The enabling of an activity depends on the case being in a specified state, i.e. the activity is only enabled if a certain milestone has been reached which did not expire yet. Consider three activities named A, B, and C. Activity A is only enabled if activity B has been executed and C has not been executed yet, i.e. A is not enabled before the execution of B and A is not enabled after the execution of C.

Synonyms

Test arc, deadline, state condition or withdraw message.

Examples

- In a travel agency, flights, rental cars, and hotels may be booked as long as the invoice is not printed.

- A customer can withdraw purchase orders until two days before the planned delivery.

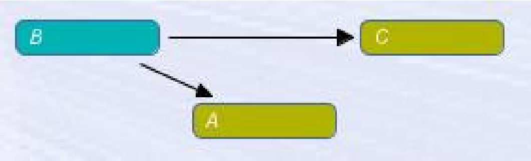
- A customer can claim air miles until six months after the flight.

Support ●

This pattern is support in Bonita.
To achieve it two different solutions are purposed:

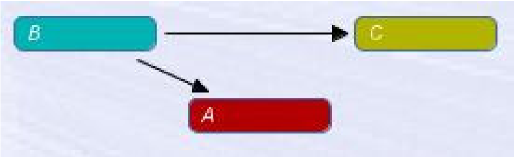
- Using a Hook (in C activity) that before starting C it cancels A if it is in ready state.
- Use Bonita Deadlines feature to set A availability.

Let's see an example of each case:



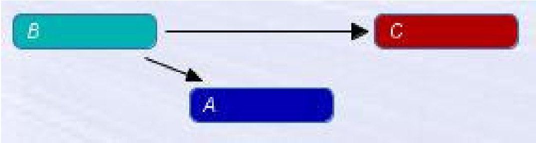
A is only enabled after the execution of B. This is achieved just setting a transition from B to A.

Then two possible situations can happen:



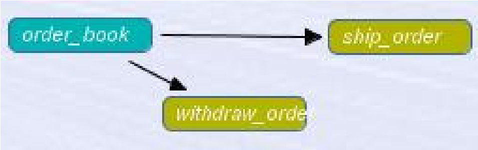
As is executed before C

OR

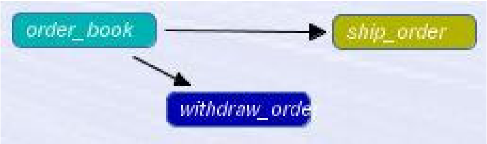


C is executed before A. Then a Hook is lunched and if A it is in *ready* state then A is cancelled.

The second example uses Bonita Deadline feature. For each activity in our process we can set a collection of deadlines to execute a hook each time that deadline is reached.



In this example we have process for booking books. We want let the user the possibility to withdraw the books order. This option will be only enabled during a certain period after the booking is accepted and before executing the shipping process.



In this sample, after termination of *order_book* activity we have 10sec to execute *withdraw_order.* If *withdraw_order* is not lunched then a onDeadLine Hook over this activity is raised cancelling the activity.

As in the previous example, if *ship_order* is executed before *withdraw_order* then *ship_order* cancels *withdraw_order.*

2.6 Cancellation Patterns

2.6.1 Pattern 19, Cancel Activity

Description

An enabled activity is disabled, i.e. a thread waiting for the execution of an activity is removed.

Synonyms

Withdraw activity.

Examples

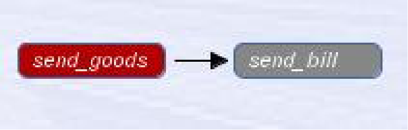
- Normally, a design is checked by two groups of engineers. However, to meet deadlines it is possible that one of these checks is withdrawn to be able to meet a deadline.

- If a customer cancels a request for information, the corresponding activity is disabled.

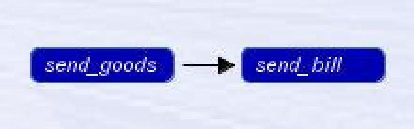
Support ●

Bonita supports the ability to cancel an activity.

This cancellation is done using *cancelActivity* method of the UserSessionBean API. This method tries to cancel an activity that is executing or in anticipating state. Cancellation is propagated through those activities that depends on the activity cancelled and will not be able to execute. Let’s see an example:



While *send_goods* activity is been execute, method *cancelActivity* is called.



Then *send_goods* activity is cancelled and also *send_bill* because it depends on the previous one.

2.6.2 Pattern 20, Cancel Case

Description

A case, i.e. workflow instance, is removed completely (i.e., even if parts of the process are instantiated multiple times, all descendants are removed).

Synonyms

Withdraw case.

Examples

- In the process for hiring new employees, an applicant withdraws his/her application.
- A customer withdraws an insurance claim before the final decision is made.

Support 🟢

Bonita supports the ability to cancel a whole process.

This cancellation is done using *removeProject* method of the UserSessionBean API. This method is used for remove either project instance or a project model. A project model cannot be deleted if there are instances of the model running.

3 NEW WORKFLOW PATTERNS

3.1.1 Pattern Extra 1, Explicit Termination

Description

A case where a process finishes its execution when a specific node is reached.

Synonyms

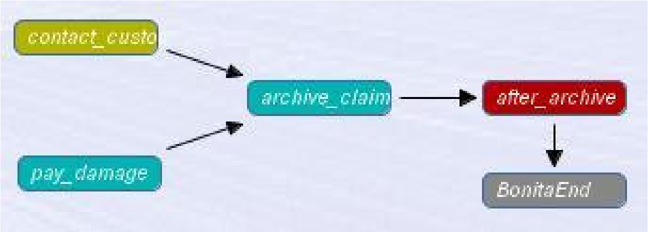
Terminal or end activity.

Examples

- Activity archive_claim is enabled after either pay_damage or contact_customer is executed.

Support ●

Bonita supports the ability to terminate a process explicitly.



To achieve that it is only necessary to reach a node called *BonitaEnd*.
This node is a special activity that finalizes the executing process when it is reached.

The difference with Simple Merge pattern is that in this case we don't need to wait the execution of *contact_costumer* activity to finish the process.
In Bonita workflow, implicit termination is done when all activities are either *terminated* or *cancelled.* So in this case is not possible to apply implicit termination pattern and it is necessary to finish thru an activity that explicitly terminates the process.

3.1.2 Pattern Extra 2, Cancel Path

Description

A point in a workflow process where we want to cancel a group of activities that are executed consecutively.

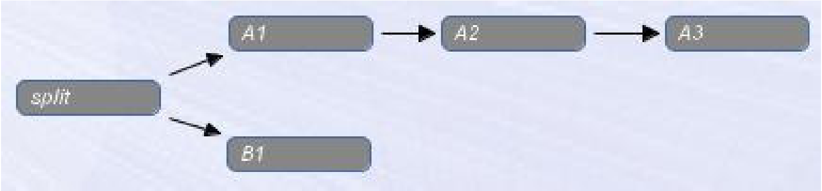
Examples

- A process splits in some branches. During the execution we want to cancel a whole branch.

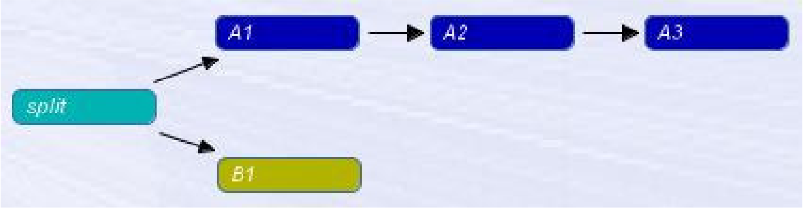
Support ●

Bonita supports the ability to cancel a path of activities inside a process.

Suppose that we have a diagram process like this:



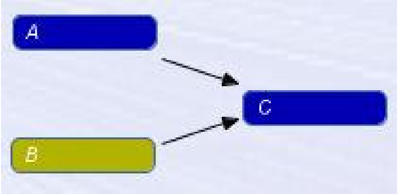
What happens with *A2* and *A3* activities if *A1* is cancelled? As they are not reachable they should be cancelled too. If we execute a hook that cancels *A1* in our example the process will look like this:



A1 has been cancelled after the hook execution and then, the Bonita engine, has cancelled *A2* and *A3*.

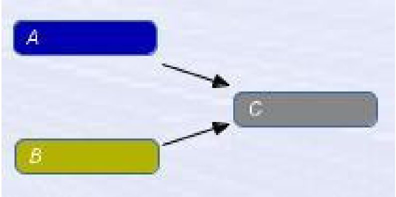
But what happens if we have a merging point?

C is an AND-JOIN activity



Cancel is propagated from A to C

C is an OR-JOIN activity



Cancel is not propagated because B can be executed. C is cancelled if B is cancelled too.

4 HOW TO RUN THE TESTS

First of all you have to download and install the latest version of Bonita from:
http://bonita.objectweb.org

To install the pattern examples:

- If you are using Windows:

```
cd %BONITA_HOME%\src\main\client\hero\client\samples\patterns
ant install_patterns
```

- If you are using Linux:

```
cd $BONITA_HOME/src/main/client/hero/client/samples/patterns/
ant install_patterns
```

To execute the pattern examples you can use the Bonita manager application:

- If you are using Windows:

```
cd %BONITA_HOME%
ant manager
```

- If you are using Linux:

```
cd $BONITA_HOME
ant manager
```

5 APENDIX: LEGEND

Shapes legend:

B	Bonita activity Can be an AND-JOIN or OR-JOIN
→	Transition from an activity to another
test.equals("1") →	Transition with condition
⇠	Iteration between two activities

Activities colors legend:

⬜	Activity in INITIAL state
🟨	Activity in READY state
🟥	Activity in EXECUTING state
🟦	Activity in TERMINATED state
🟦	Activity in DEAD state

Pattern support legend:

🟢	Pattern support with Bonita workflow
🟠	Patter partially supported with Bonita workflow
🔴	Pattern not supported with Bonita workflow