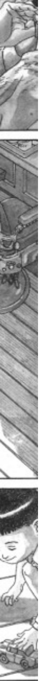


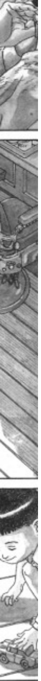
# Modellazione di Processi Amministrativi e Compliance Normativa

8c.BPMN - analytic modeling

Matteo Baldoni

A.Y. 2022/2023

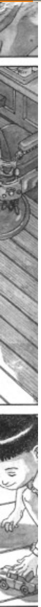
Università degli Studi di Torino - Dipartimento di Informatica



**Si noti che**

qesti lucidi sono basati su quelli di Marco Montali, KRDB Research Centre for Knowledge and Data Faculty of Computer Science, Free University of Bozen-Bolzano e su quelli di Andrea Marrella.

## BPMN Level 2: Analytic Modeling



- Level 1: each step of the process is triggered by the completion of the previous step.
- Level 2: expansion of level 1 with reaction to events.

**Event**

Something that happen in a process, at a specific point in time.

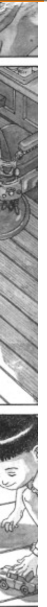
**Reaction to events**

Depends on whether the process is throwing or catching the event.

- Reaction to throwing an event: how the process generates a signal that something happened.
  - Type of signal represents the trigger.
- Reaction to catching an event: how the process responds to a signal that something happened.
  - Type of signal represents the result.

The type of signal is shown as an *icon* inside the event circle.

## About Events

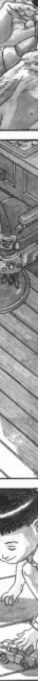


**Event Types**

- Start event (thin line): indicates where the process starts.
- End event (thick line): indicates where a path of the process ends.
- Intermediate event (double ring): indicates that something happens during the execution of the process.

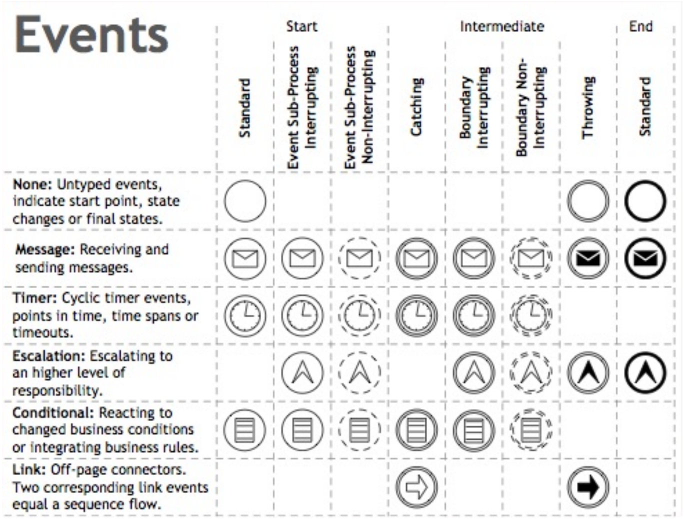
It is now possible to model:

- Reaction to external events.
- Exception handling.
- Parallel event handlers.
- Complex indirect interactions between different process parts, where one triggers an event and the other catches it.
- Cancelations and compensations.
- ...



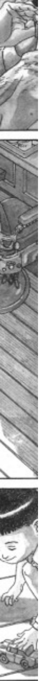
## Question

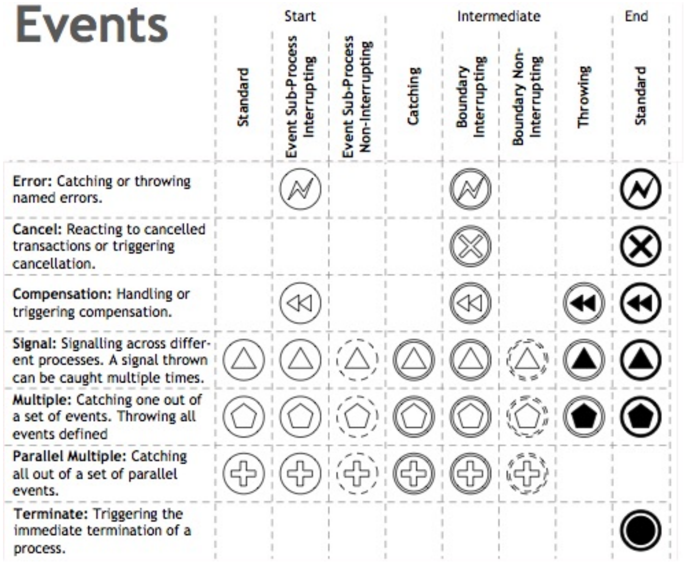
Why do people fear BPMN?



# Events

| | Start | | | Intermediate | | | | End |
|---|---|---|---|---|---|---|---|---|
| | Standard | Event Sub-Process Interrupting | Event Sub-Process Non-Interrupting | Catching | Boundary Interrupting | Boundary Non-Interrupting | Throwing | Standard |
| **None:** Untyped events, indicate start point, state changes or final states. | ○ | | | | | | ○ | ○ |
| **Message:** Receiving and sending messages. | ✉ | ✉ | ✉ | ✉ | ✉ | ✉ | ✉ | ✉ |
| **Timer:** Cyclic timer events, points in time, time spans or timeouts. | ◷ | ◷ | ◷ | ◷ | ◷ | ◷ | | |
| **Escalation:** Escalating to an higher level of responsibility. | | ▲ | ▲ | | ▲ | ▲ | ▲ | ▲ |
| **Conditional:** Reacting to changed business conditions or integrating business rules. | ▤ | ▤ | ▤ | ▤ | ▤ | ▤ | | |
| **Link:** Off-page connectors. Two corresponding link events equal a sequence flow. | | | | ⇨ | | | ➡ | |

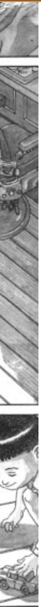




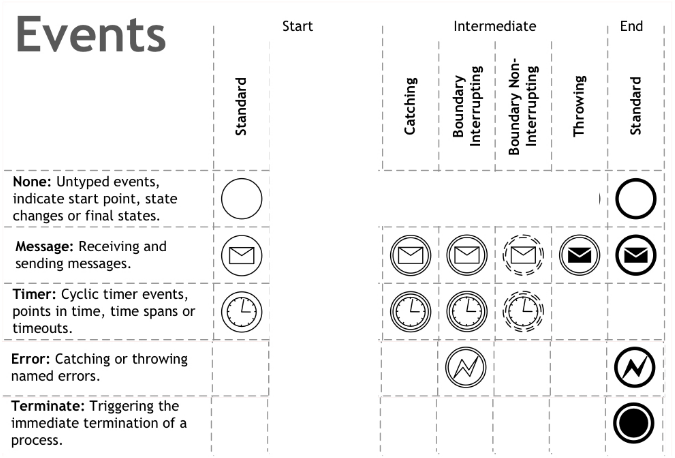
# Events

| | Start | | | Intermediate | | | | End |
|---|---|---|---|---|---|---|---|---|
| | Standard | Event Sub-Process Interrupting | Event Sub-Process Non-Interrupting | Catching | Boundary Interrupting | Boundary Non-Interrupting | Throwing | Standard |
| **Error:** Catching or throwing named errors. | | ϟ | | | ϟ | | | ϟ |
| **Cancel:** Reacting to cancelled transactions or triggering cancellation. | | | | | ✕ | | | ✖ |
| **Compensation:** Handling or triggering compensation. | | ⏪ | | | ⏪ | | ⏪ | ⏪ |
| **Signal:** Signalling across different processes. A signal thrown can be caught multiple times. | △ | △ | △ | △ | △ | △ | ▲ | ▲ |
| **Multiple:** Catching one out of a set of events. Throwing all events defined | ⬠ | ⬠ | ⬠ | ⬠ | ⬠ | ⬠ | ⬟ | ⬟ |
| **Parallel Multiple:** Catching all out of a set of parallel events. | ✚ | ✚ | ✚ | ✚ | ✚ | ✚ | | |
| **Terminate:** Triggering the immediate termination of a process. | | | | | | | | ● |

## First Step in the World of Events

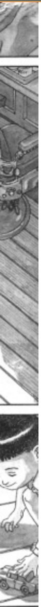


Out from the 104 possible combinations (only half of which are accepted by the standard), we first focus on:



| Events | Start | Intermediate | | | | End |
|---|---|---|---|---|---|---|
| | Standard | Catching | Boundary Interrupting | Boundary Non-Interrupting | Throwing | Standard |
| **None:** Untyped events, indicate start point, state changes or final states. | ○ | | | | | ○ |
| **Message:** Receiving and sending messages. | ✉ | ✉ | ✉ | ✉ | ✉ | ✉ |
| **Timer:** Cyclic timer events, points in time, time spans or timeouts. | 🕒 | 🕒 | 🕒 | 🕒 | | |
| **Error:** Catching or throwing named errors. | | | ⚡ | | | ⚡ |
| **Terminate:** Triggering the immediate termination of a process. | | | | | | ● |

## Intermediate Events



Events occurring after the start of a process level, but before its end. Four basic contexts of use:

1. Throw an intermediate event.
2. Catch an intermediate event.
3. Catch an intermediate event in a specific process level, by interrupting it.
4. Catch an intermediate event in a specific process level, without interrupting it.

## Throwing Intermediate Event

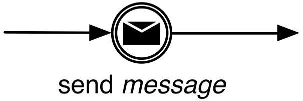
Intermediate event with a black icon inside.

**Semantics**

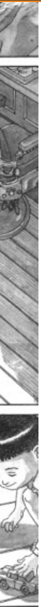
1. As soon as the sequence flow reaches the event, the corresponding signal is **thrown**.
2. The process continues immediately.

Supported only by few event types.

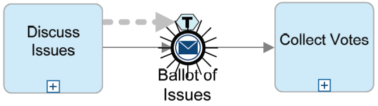
- Typical usage: throwing a message event.



## Token Game: Throw Intermediate Events[1]



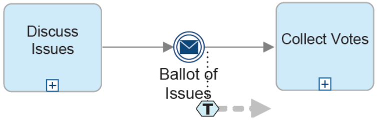
- A token arriving at a throw Intermediate Event would immediately fire the trigger. It would then leave immediately and travel down the outgoing Sequence Flow
- A Throwing Intermediate Event can not be attached to the boundary of an Activity



[1]Credits: Andrea Marrella “Modeling Business Processes with BPMN"

## Token Game: Throw Intermediate Events[1]

- A token arriving at a throw Intermediate Event would immediately fire the trigger. It would then leave immediately and travel down the outgoing Sequence Flow
- A Throwing Intermediate Event can not be attached to the boundary of an Activity



[1]Credits: Andrea Marrella "Modeling Business Processes with BPMN"

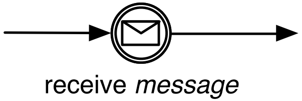
## Catching Intermediate Event

Intermediate event with a white icon inside.

**Semantics**

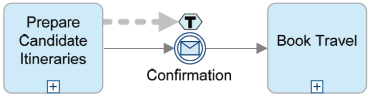
1. As soon as the sequence flow reaches the event, the process **waits** for the corresponding trigger signal.
2. When the trigger signal is caught, the process resumes immediately.

Supported by many event types, but not error events.



## Token Game: Catch Intermediate Events[2]

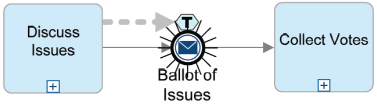
- When a token arrives at a catching Message Intermediate Event, the Process pauses until a message arrives



[2]Credits: Andrea Marrella “Modeling Business Processes with BPMN"

## Token Game: Catch Intermediate Events[2]

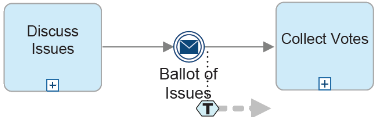
- When a token arrives at a catching Message Intermediate Event, the Process pauses until a message arrives
- When a token arrives at a throwing Message Intermediate Event, it immediately triggers the Event, which sends the message to a specific participant



[2]Credits: Andrea Marrella “Modeling Business Processes with BPMN"

## Token Game: Catch Intermediate Events[2]

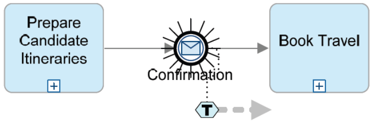
- When a token arrives at a catching Message Intermediate Event, the Process pauses until a message arrives
- When a token arrives at a throwing Message Intermediate Event, it immediately triggers the Event, which sends the message to a specific participant



[2]Credits: Andrea Marrella "Modeling Business Processes with BPMN"

## Token Game: Catch Intermediate Events[2]

- When a token arrives at a catching Message Intermediate Event, the Process pauses until a message arrives
- When a token arrives at a throwing Message Intermediate Event, it immediately triggers the Event, which sends the message to a specific participant
- If the token is waiting at the Intermediate Event and the message arrives, then the Event triggers



[2]Credits: Andrea Marrella "Modeling Business Processes with BPMN"

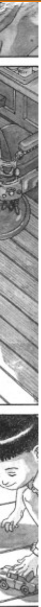
## Boundary Event

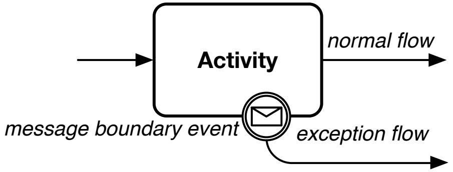
Catching intermediate event drawn on the boundary of an activity.

**Semantics**

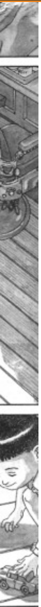
1. **While** the activity is running, it **listens** to the signal attached to the event.
2. If the activity completes without the occurrence of the boundary event signal, the process continues on the standard execution path (normal flow).
3. If the signal occurs before the activity completes, the sequence flow out of the event is triggered (exception flow).

How this is actually done depends on the *type* of the boundary event.

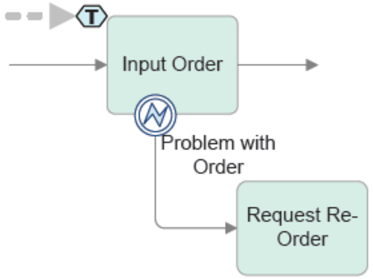




## Token Game: Boundary Events[3]



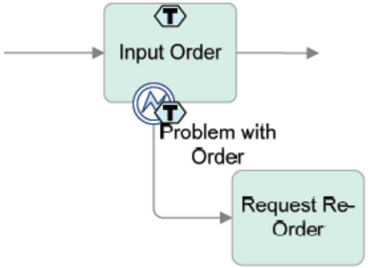
- The token leaves the previous flow object and arrives at the Activity with the attached Intermediate Event



[3]Credits: Andrea Marrella “Modeling Business Processes with BPMN"

## Token Game: Boundary Events[3]

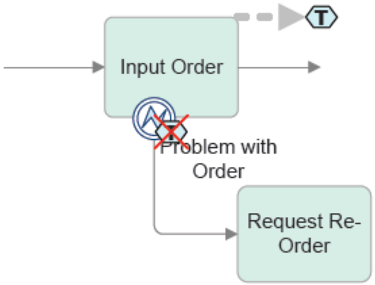
- The token leaves the previous flow object and arrives at the Activity with the attached Intermediate Event
- he token enters the Activity and starts the work of the Activity. At the same time, another token is created and resides in the Intermediate Event on its boundary



[3]Credits: Andrea Marrella "Modeling Business Processes with BPMN"

## Token Game: Boundary Events[3]

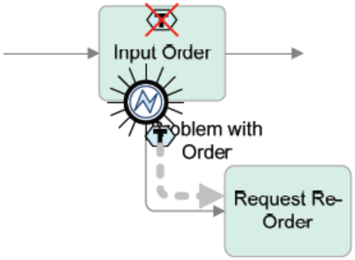
- The token leaves the previous flow object and arrives at the Activity with the attached Intermediate Event
- he token enters the Activity and starts the work of the Activity. At the same time, another token is created and resides in the Intermediate Event on its boundary
- If the Activity finishes before the trigger occurs, then the token from the Activity moves down the normal outgoing Sequence Flow of the Activity and the additional token is consumed



[3]Credits: Andrea Marrella “Modeling Business Processes with BPMN"

## Token Game: Boundary Events[3]

- The token leaves the previous flow object and arrives at the Activity with the attached Intermediate Event
- he token enters the Activity and starts the work of the Activity. At the same time, another token is created and resides in the Intermediate Event on its boundary
- If the Activity finishes before the trigger occurs, then the token from the Activity moves down the normal outgoing Sequence Flow of the Activity and the additional token is consumed
- However, if the attached Intermediate Event triggers before the Activity finishes, then the Activity is interrupted (all work stops). In this case, the token from the Event moves down its outgoing Sequence Flow. The token that was on the Activity is consumed



[3]Credits: Andrea Marrella “Modeling Business Processes with BPMN"

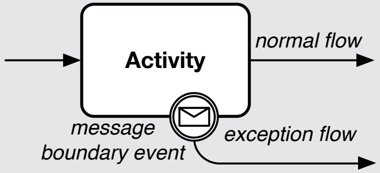
## Types of Boundary Events

### Interrupting boundary event

If the trigger signal occurs during the activity execution:

- The activity is **immediately terminated**.
- The exception flow is activated.

Drawn with a **solid** double ring.

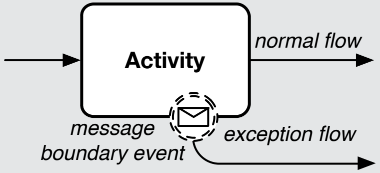


### Non-interrupting boundary event

If the trigger signal occurs during the activity execution:

- The activity is **normally continues**.
- The exception flow is activated (with a new parallel thread).

Drawn with a **dashed** double ring.



# Timer Event

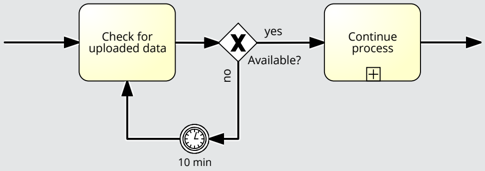
## Normal Catching timer

Indicates to:

- wait for a specific duration, or
- wait until a specific timepoint is reached.

## Example

The process checks whether data have been uploaded. If so, the process goes on. If not, the process waits for ten minutes and then repeats the check.



## Timer Event

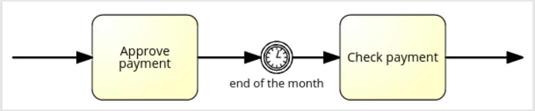
### Normal Catching timer

Indicates to:

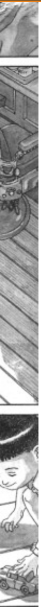
- wait for a specific duration, or
- wait until a specific timepoint is reached.

### Example

After the payment is approved, the process waits until the end of the month, and then checks whether the payment has been actually done.



## Timer Event



**Normal Catching timer**

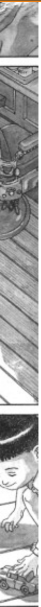
Indicates to:

- wait for a specific duration, or
- wait until a specific timepoint is reached.

**Warning**

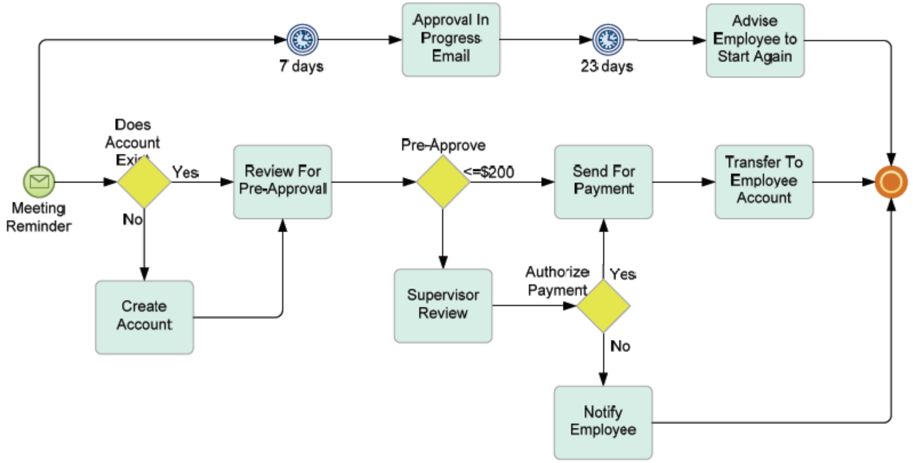
Don't use timer events to represent the duration of activities.

## Exercise



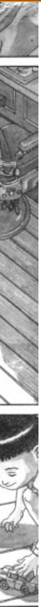
- Design a a sample expense reimbursement process. This process provides for reimbursement of expenses incurred by employees for the company. For example buying a technical book, office supplies or software. In a normal day there are several hundreds of instances of this process created. Concentrate on the basic flow of the Process...
- After the reception of a meeting remainder, a new account must be created if the employee does not already have one. The report is then reviewed for automatic approval. Amounts under \$200 are automatically approved, whereas amounts equal to or over \$200 require approval of the supervisor. In case of rejection, the employee must receive a rejection notice by email. The reimbursement goes to the employee's direct deposit bank account. If the request is not completed in 7 days, then the employee must receive an "approval in progress" email If the request is not finished within 30 days, then the process is stopped and the employee

## Exercise



7 days
Approval In Progress Email
23 days
Advise Employee to Start Again
Does Account Exist
Yes
No
Meeting Reminder
Review For Pre-Approval
Pre-Approve
<=$200
Send For Payment
Transfer To Employee Account
Create Account
Supervisor Review
Authorize Payment
Yes
No
Notify Employee

## Timer Event



**Timer boundary event**

Combination of stopwatch and alarm.

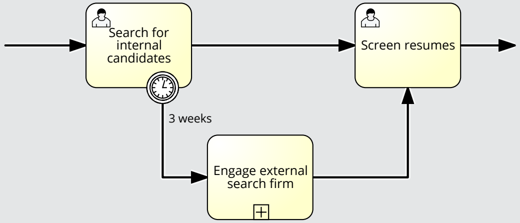
- When the activity is reached by an incoming flow (start event), the stopwatch starts.
- If the activity is still running when the stopwatch expires, an alarm is triggered and the exception flow is **immediately** activated.

The reaction then depends on whether the timer is interrupting on not.

## Timer Event: Examples

**Example**

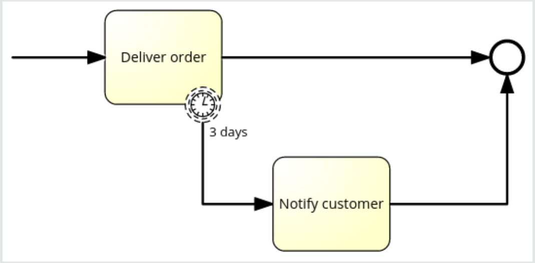
A hiring process starts by searching for internal candidates. If the search is still running after 3 weeks, the search is interrupted and an external search firm is engaged. In both cases, the collected CVs are then screened.



## Timer Event: Examples

**Example**

If the delivery of an order takes more than 3 days, the customer is informed of the delay.

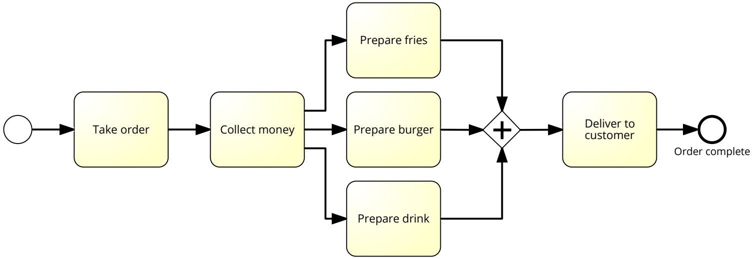


## Timed Interval

**Example**

The food delivery process starts when an order is received from the customer. The money is then collected from the customer. Next, drink and food are prepared. Once both food and drink are ready, the order is delivered to the customer.

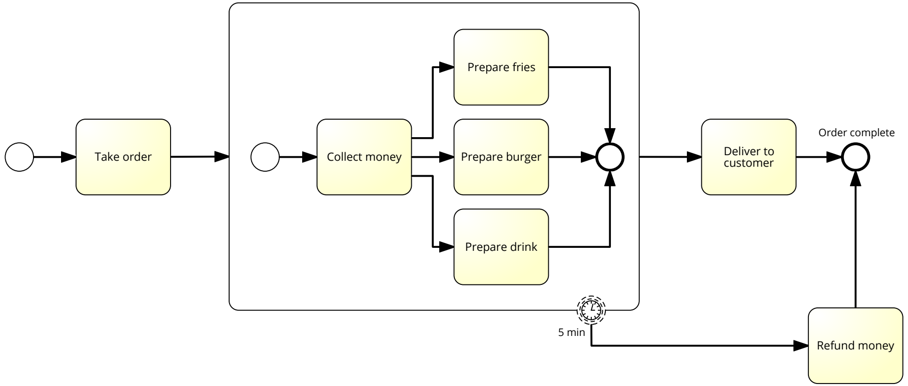
A policy of the food company is that if the time-to-delivery exceeds 5 minutes, then half of the money must be returned to the customer.



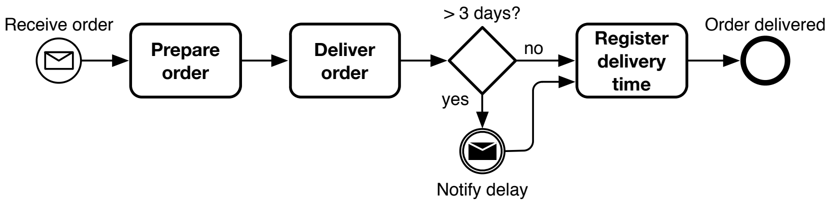
## Timed Interval

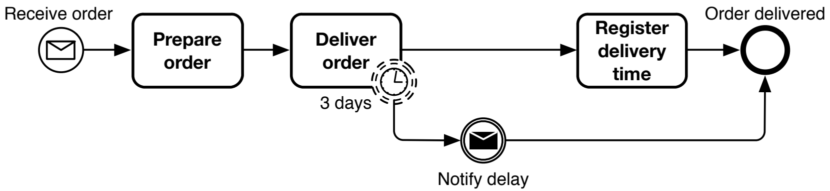
### Strategy

- Wrap the “time interval” of interest into a subprocess.
- Attach a timer boundary event to the subprocess.

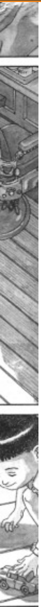


# Can You Tell the Difference?





## More on Messages



### BPMN definition for “message”

The content of a communication *between two participants*.

- Could represent even material flow, i.e., the delivery of a physical object.

A message has an **item definition** that specifies the message payload, i.e., an information or physical object.

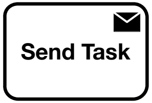
### Warning

Consider “send” and “receive” as BPMN keywords that denote the exchange of a message from the point of view of the emitter or destination.

### Location of a message

A message could correspond to many distinct message flows, representing the message in different situations of the process.

## Who Can Send a Message?

- A black-box pool.
- A throwing message event.
- Any type of activity (optional send).
- A send task. 

**Send task**

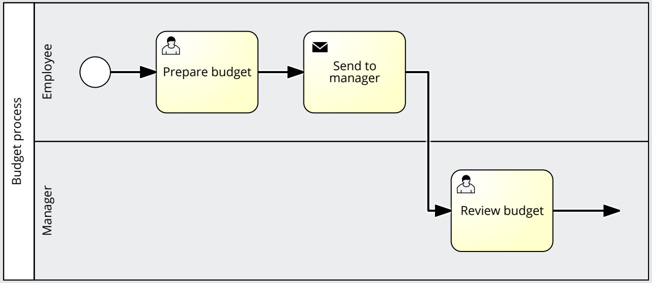
A task consisting in the **immediate emission** of a message (certain send). Difference with an intermediate throwing message event:

- The task has a performer.
  - Conventionally, the performer of a throwing event can be understood as the *lane* containing that event.
- The task can have boundary events attached to it.
- The task can have special decorators indicating forms of repetitions (see later).

## “Sending” within a Process

**Error**

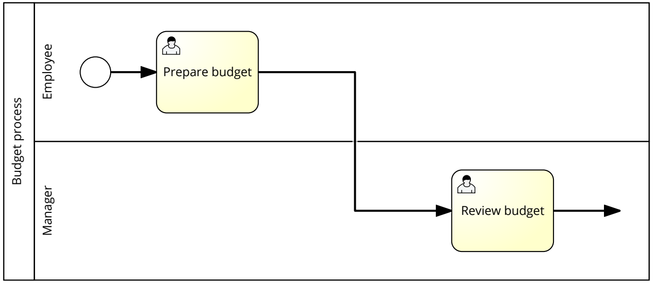
Message flow cannot be used to forward work to a downstream task within the same process.



**NO!!**

## “Sending” within a Process

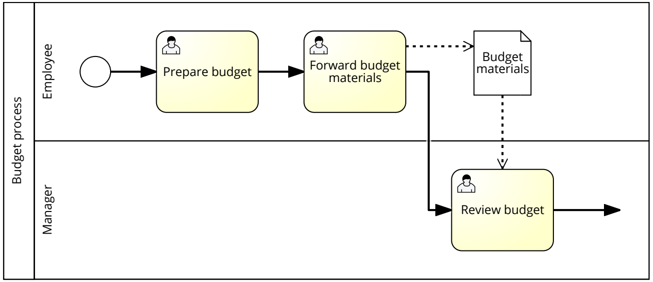
Solution: Work forwarded implicitly via control-flow.



## “Sending” within a Process

Solution: Forwarding of work encapsulated in a specific activity (without using the word “send”).

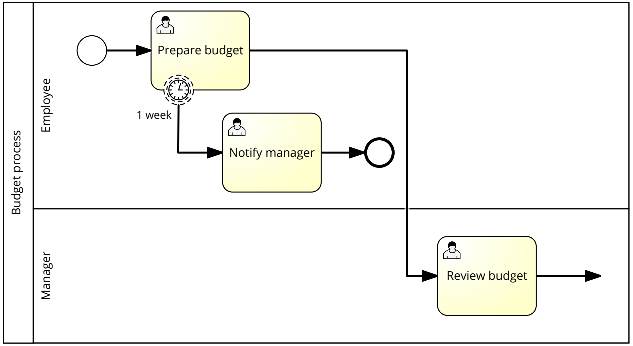
In case: Forwarded work explicitly shown as a data object.



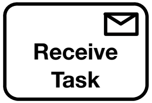
## “Sending” within a Process

Solution: *Notification* typically handled through a user task in the lane responsible for the notification.

- Not the one “receiving” the notification.



## Who Can Receive a Message?

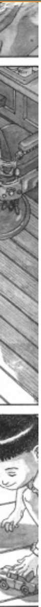
- A black-box pool.
- A catch message event.
- Any type of activity (optional receive).
- A receive task. 

**Receive task**

A task only responsible for **waiting** for a message (blocking receive). Difference with an intermediate catching message event:

- The task has a performer.
  - Conventionally, the performer of a catch event can be understood as the *lane* containing that event.
- The task can have boundary events attached to it.
  - Timers are particularly interesting here.

## Synchronous vs Asynchronous Communication



**Synhronous Communication**

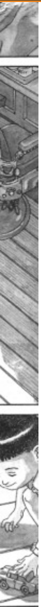
Whenever a message is sent by a process, the process waits for a response message before continuing.

Natively, BPMN message events account for **asynchronous communication**: the emission of a message does not interrupt the emitter process.

How to support synchronous communication?

- Short-running synchronous communication: service task.
  - Implicitly sends a message and then waits for a response to complete.
- Long-running synchronous communication: sequence of send and receive.
  - Possibly inserting other activities inbetween.

## Message Boundary Event



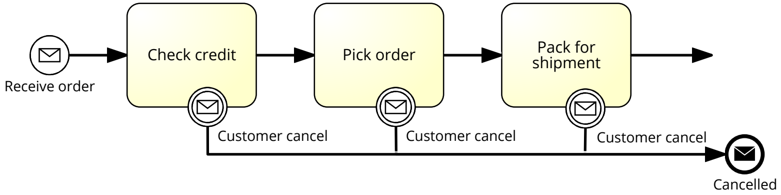
Management of **unsolicited messages**.

**Example**

An order management process starts when a customer places an order. The order is fulfilled, then shipped, and finally invoiced. The customer may cancel the placed order while it is being fulfilled or shipped. In the first case, the process immediately terminates by notifying to the customer that the cancelation has been correctly handled. In the second case, the process does not interrupt its normal course of execution (it is too late). However, an additional task is required to be executed, whose purpose to to authorize the return for credit to the customer.

## Message Boundary Event

Management of **unsolicited messages**.



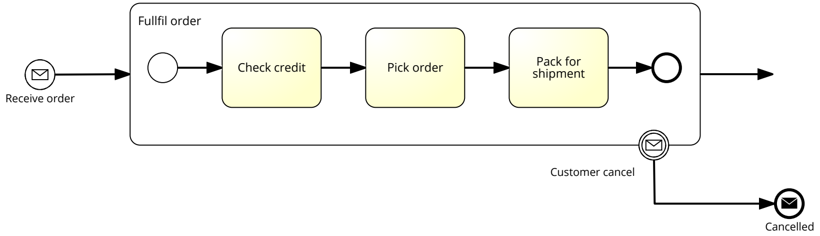
**NO!!**

## Message Boundary Event

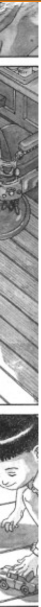
Management of **unsolicited messages**.

**Strategy**

- Identify the maximal fragments of a process level for which the management of an event is the same.
- Surround these fragments with subprocesses, using boundary events and exception flow to capture how the event is managed when it occurs in that specific process part.



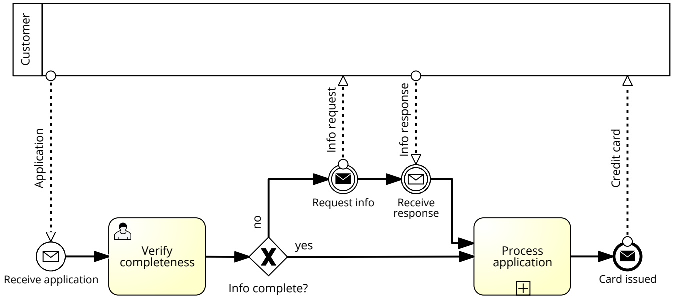
## Event-Driven Choices



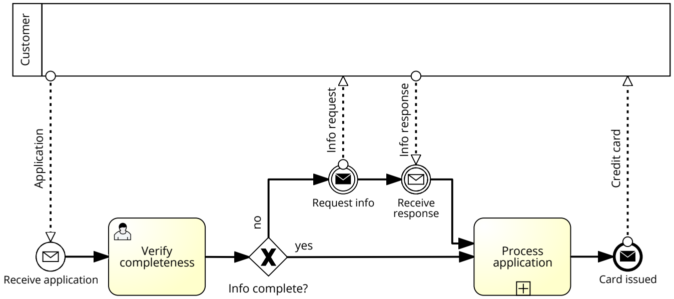
**Example**

An credit card generation process starts when a customer sends an application to the bank. An employee of the bank then verifies the completeness of the application. If the application is complete, it is processed (complex activity), and then the card is issued. If the application is not complete, before processing it an additional interaction with the customer is required. In particular, the bank requests the additional required info to the customer, and then waits for an answer. If the customer refused to provide the requested information, then the process ends by sending a rejection to the customer. If instead the customer positively replies to the request, the application is fixed considering the newly requested infromation, then processing the application.

# Event-Driven Choices



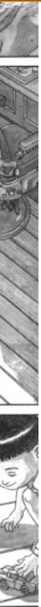
## Event-Driven Choices



**Need for regaining control**

What happens if the customer decides not to answer at all to the bank's request? **Timeout** needed!

## Event-driven Choices



Sometimes, a choice in a process depends on which event triggers first among a set of possible alternatives:

- Alternative messages (possibly from different participants).
- Timeouts.

**Warning**

This cannot be captured, in general, by using a normal choice gateway: the decision cannot be taken **before** the event triggers!

**Case of alternative messages from same participant**

- Alternative messages compacted into a unique message with an extended payload.
- Subsequent gateway checks the extended payload to internally decide how to continue.

**General case**

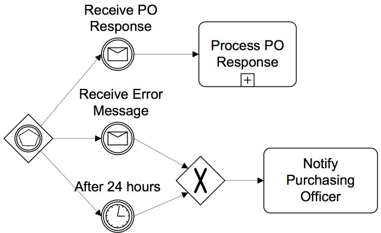
**Event gateway**.

## Event Gateway

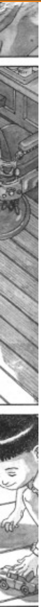
Choice point whose outcome depends on the first received event, not on an internal decision of the orchestration.

**Semantics**

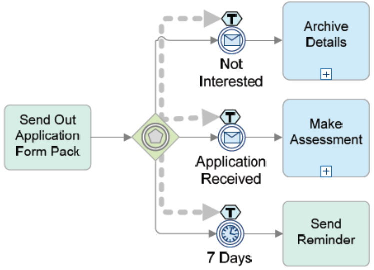
1. When the process reaches the event gateway, it waits.
2. The first triggering event attached to the gateway determines which path is taken (race conditions).
3. The process istantanteously reacts by moving accordingly.
4. If other signals are triggered later on, they are ignored.



## Token Game: Event gateway[4]



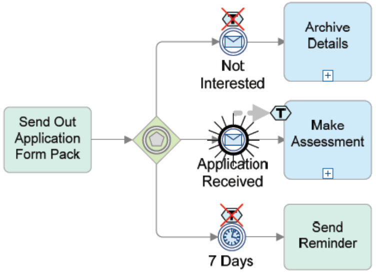
- The tokens will wait there until one of the Events is triggered
- The Intermediate Events that are part of the Gateway configuration become involved in a race condition. Whichever one finishes first (fires) will win the race and take control of the Process with its token
- Then the token will immediately continue down its outgoing Sequence Flow, by disabling the other paths



[4]Credits: Andrea Marrella “Modeling Business Processes with BPMN"

## Token Game: Event gateway[4]

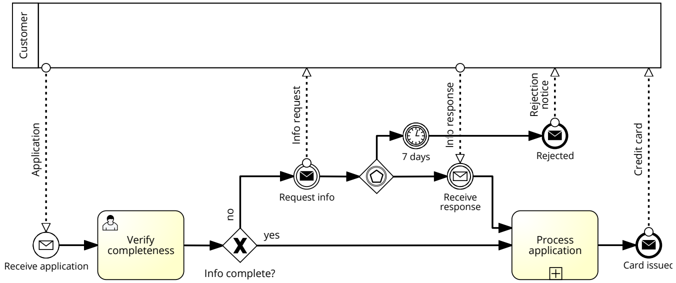
- The tokens will wait there until one of the Events is triggered
- The Intermediate Events that are part of the Gateway configuration become involved in a race condition. Whichever one finishes first (fires) will win the race and take control of the Process with its token
- Then the token will immediately continue down its outgoing Sequence Flow, by disabling the other paths



[4]Credits: Andrea Marrella "Modeling Business Processes with BPMN"

## Event-driven Choices

Event gataway waits for response or timeout, whichever occurs first



## Error event

**Error**

An **exception end state** of an activity (business or technical error).

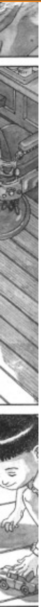
**Interrupting error boundary event**

Indicates an **error code** and how that error is handled if the activity execution experiences that error.

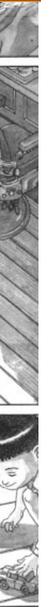
- On a task: represents the exception flow to be followed when the task completes unsuccessfully.
- On a subprocess: same meaning, but requires that the inner specification of a descendant subprocess explicitly mentions the corresponding **error end state**.

**Error end state**

End state marking an error termination of the process level (with a certain error code).
We assume that the label corresponds to its error code.



## Throw-Catch Pattern

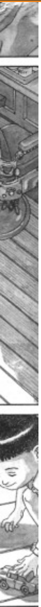


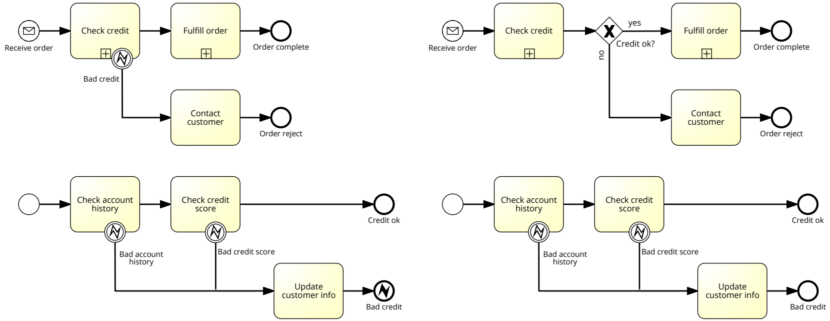
**Semantics of Error Handling**

When the current process level reaches an error end state:

1. All active parallel threads within the same process level are immediately terminated ($\sim$ termination end state).
2. The corresponding error signal is generated, and propagated to the parent level.
3. If the parent level defines an error boundary event with matching error code, the signal is caught and the exception flow is activated.
4. Otherwise, the signal is recursively propagated to the ancestors, until the *nearest* ancestor able to handle the error finally catches it.

# Throw-Catch Pattern

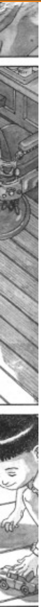




**Observation**

The boundary error event is not really “interrupting”, since it is triggered when the inner activity terminates in an error end state.

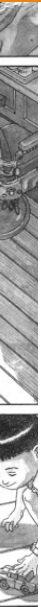
## Correspondence with Level 1 Exceptions



Business exceptions handled by the immediate parent process level can be equivalently modeled:

- Using the methodology seen in level 1.
  - Different end termination states.
  - Gateway used in the parent process to test which termination state has been reached.
- Using throw-catch patterns: more emphasis on the fact that we are handling errors.

## Escalation Event



Represented with an upward arrow inside.

**Escalation event**

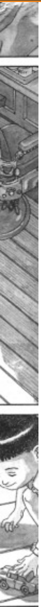
Used to handle a “mild” exception. Differently from errors:

- An escalation can be triggered in the middle of a process level, not just in an end state.
- An escalation is typically caught without interrupting the current process level.
  - But can also be interrupting when needed.

**Semantics**

Identical to the throw-catch pattern for error events, considering that in this case the inner process does not necessarily terminate upon escalation.

Typically used to handle ad-hoc exceptions in user tasks.



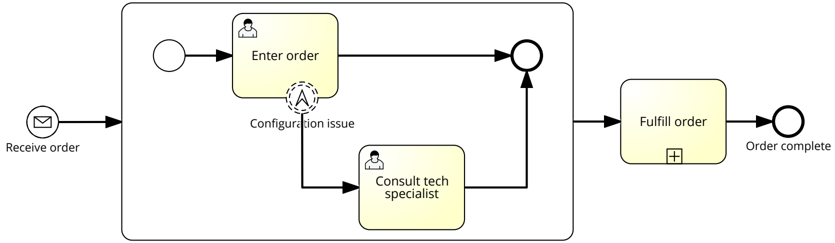
## Managing Exception Synchronization

**Example**

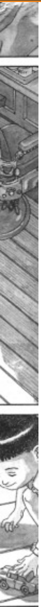
Whenever a request is received, the process handles it by entering its details. If a configuration issue arises, a specialist from the IT department is consulted. Once this phase is properly completed, the request is fulfilled.

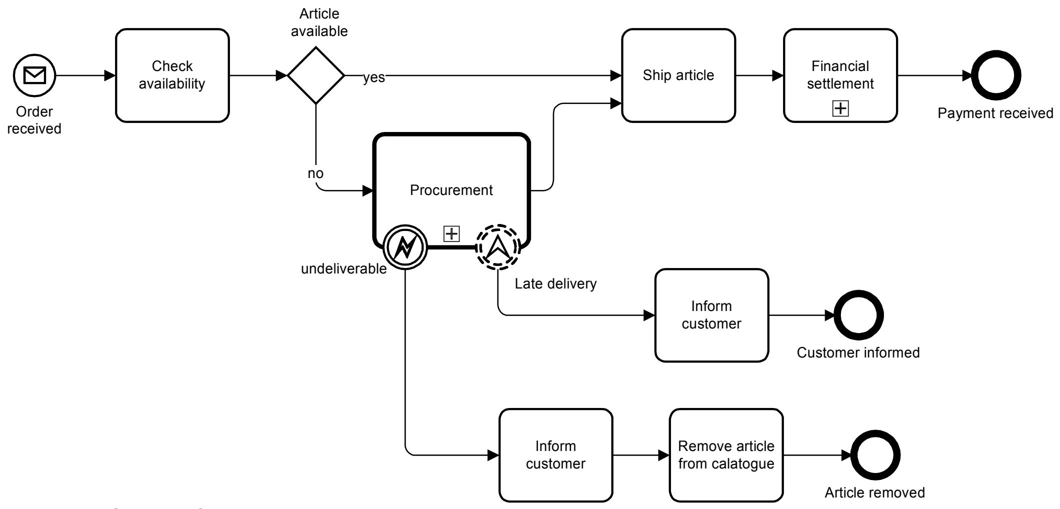
Usage of subprocess boundaries to introduce a “dynamic” synchronization point.

- We will see another construct to handle this issue later on.



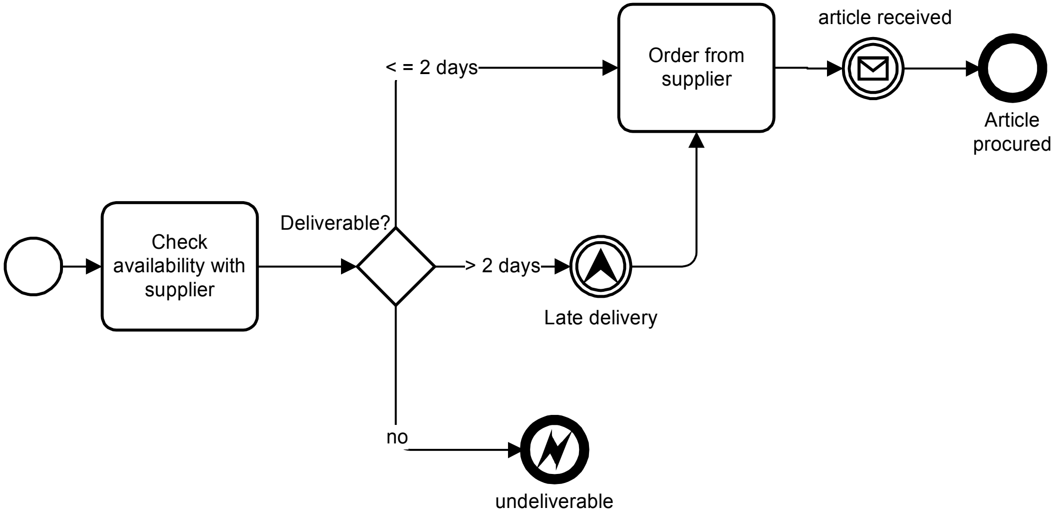
# BPMN - Order Example (Main Process)





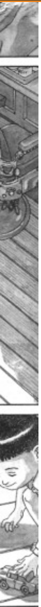
From *BPMN 2.0 by Example* - `http://www.bpmn.org/`

## BPMN - Order Example (Procurement Sub-Process)



From *BPMN 2.0 by Example* - `http://www.bpmn.org/`

## Signal Event



Represented with a triangle inside.

**Signal event**

Event broadcasted to all active threads in a process, and all (external) active participants.

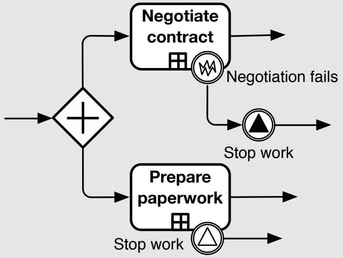
- Intra-process: loose coupling between thrower and catchers, with the possibility of reaching parallel threads.
  - A correlation mechanism is assumed to target the right process instance.
- Inter-process: Publish-subscribe paradigm (signal broadcasted to all interested participants).
  - Any process interested in the signal could trigger a new instance using a signal start event.

## Signal Event and Flexibility

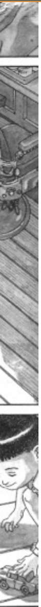
More flexible mechanism than for other events.

- Flexible catchers.
  - Errors/escalations: only target ancestor process levels.
  - Messages: only target other pools.
- Flexbile reaction.
  - Terminate/error are only able to terminate and entire process level.
  - Signal can selectively induce termination of some activities within the current process level.

### Selective termination



## Conditional Event



Represented using a list inside.

**Conditional event**

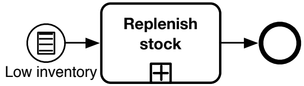
Continuous monitoring of a data condition/business rule.

- Event triggered every time the condition becomes true.
- Used as a starting, catching intermediate and boundary event.

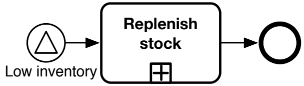
## Differences and Similarities

Replenishing stock when inventory is low. . .

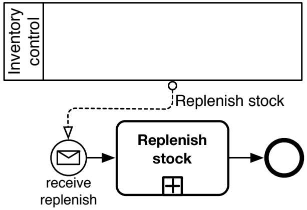
- Internal condition evaluation.



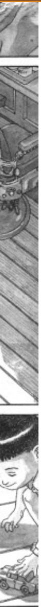
- External condition evaluation (by anybody).



- External condition evaluation by the mentioned participant.



## Event Subprocess



Exception flow can be thought as the **handler** of an event.

**Event subprocess**

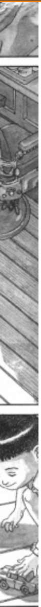
An event handler whose trigger is active during the whole execution of a process level.

Features:

- Subprocess with a *triggered start event*: message, timer, or error.
- Interrupting vs non-interrupting.
- Graphically: dashed subprocess, plus:
  - Collapsed: has a start event icon in the corner to show the type of trigger it listens to
  - Expanded: uses the start event icon to start.
- Start event: solid/dashed for interrupting/non-interrupting subprocess.

The event subprocess may trigger errors/escalations: they will be managed by the parent process in the usual way.

## Notes for Architects

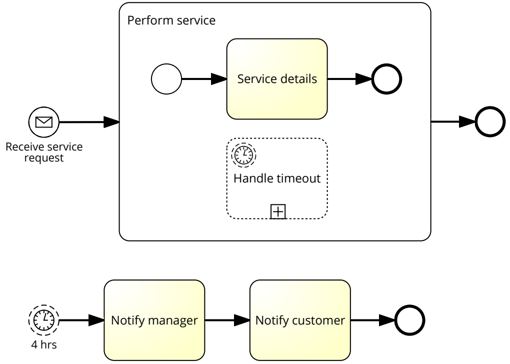


- Event subprocesses more easily map to BPEL.
- Event subprocess can access the **context** of its process level (data and state values).
  - This is not possible when a boundary exception flow is used, because in that case the handler **belongs to the parent level**.

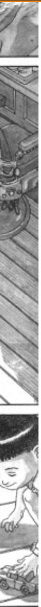
## Handling Timeout

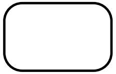
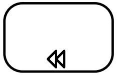
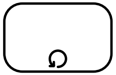
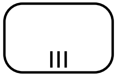
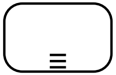
**Example**

Upon a service request, the process enters into a “perform service” subprocess. This subprocess is meant to run within 4 hours at most. If this time is exceeded, then the subprocess continues, but two notifications have to be sent: one to the manager, and one to the customer.



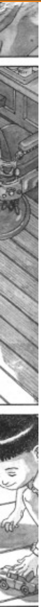
# Activity Decorators



| | |
|---|---|
|  | Basic task. |
|  | Compensation task. |
|  | Loop task (looping information attached to the activity). |
|  | Multi-instance task with parallel composition (expression attached to the activity to calculate the number of instances). |
|  | Multi-instance task with sequential composition. |

For sub-processes only: ad-hoc (tilde marker) - flexible execution of the inner activities, without a complete specification of the process.

## Loop Activity



An activity with a loop marker.

**Loop activity**

An activity possibly repeated multiple times. A Loop condition indicates whether the activity must be repeated or whether the process must continue on the outgoing flow.

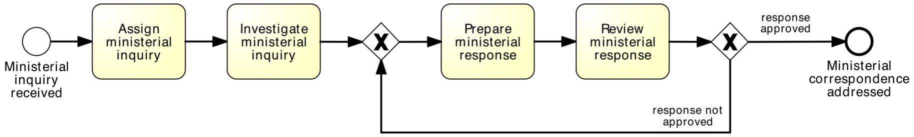
- Loop condition “Until X” is captured by gate “If not X”.
- A text annotation used to graphically show the loop condition.
- Iterations are performed sequentially.
- Number of iterations dynamically established.

**Warning**

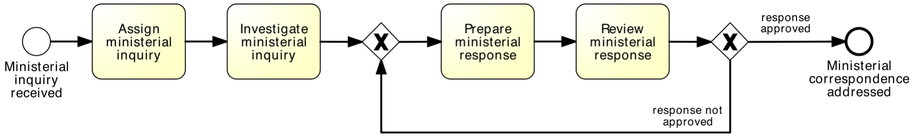
Corresponds to a normal activity followed by a gateway that connects back to the activity if the loop condition is true.

- Do not mix the two notations!

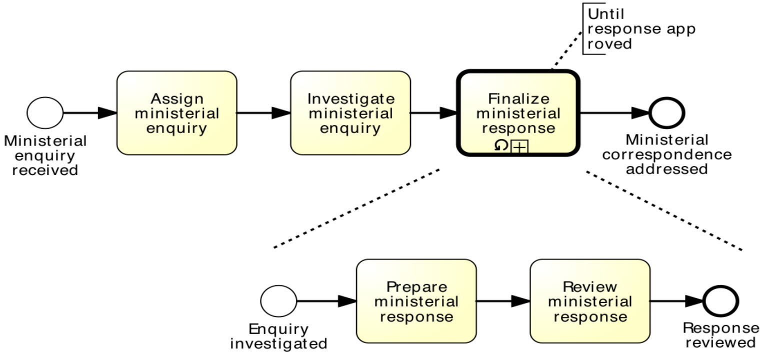
## Example: Loops


Ministerial inquiry received
Assign ministerial inquiry
Investigate ministerial inquiry
X
Prepare ministerial response
Review ministerial response
X
response approved
Ministerial correspondence addressed
response not approved

## Example: Loops

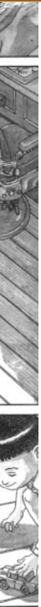


Ministerial inquiry received
Assign ministerial inquiry
Investigate ministerial inquiry
Prepare ministerial response
Review ministerial response
response approved
Ministerial correspondence addressed
response not approved



Until response app roved
Ministerial enquiry received
Assign ministerial enquiry
Investigate ministerial enquiry
Finalize ministerial response
Ministerial correspondence addressed
Enquiry investigated
Prepare ministerial response
Review ministerial response
Response reviewed

## Multi-Instance Activity



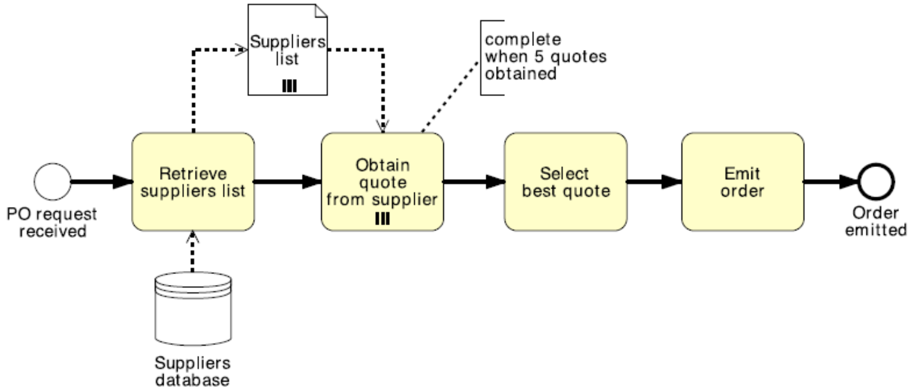
An activity marked by three parallel bars.

**MI activity**

Activity instantiated **multiple times**, one per element in a *collection*.

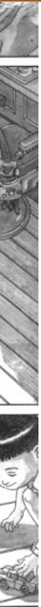
- Number of iterations depends on the case and is **known** when the activity is reached.
    - Design time or dynamically, but before the activity starts.
- Text annotation of the form “for each X” is useful if the activity label does not clarify the MI nature.
- Execution can be:
    - Sequential (horizontal bars).
    - Parallel (vertical bars).
- Default termination of the activity: **all-complete**.
- A terminate/interrupting boundary event can be used to stop all running instances.

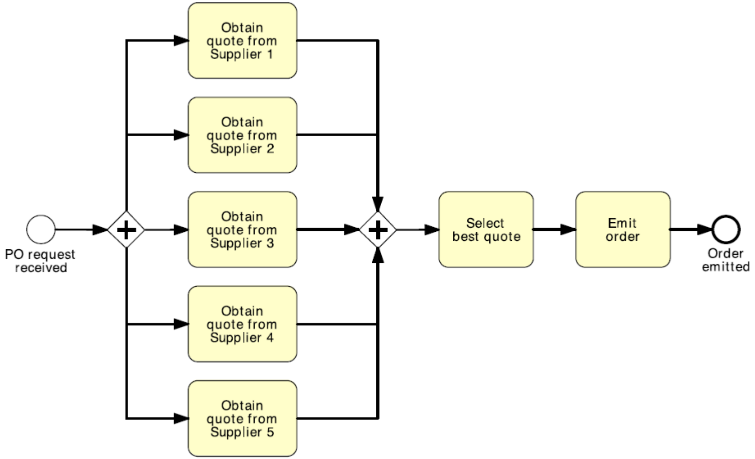
## Example: Multi-Instances



Note that: the number of items in the collection determines the number of activity instances to be created. After all quotes have been received, they are evaluated and the best quote is selected.

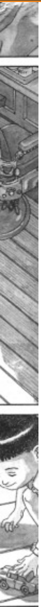
## Example: Multi-Instances




Obtain quote from Supplier 1
Obtain quote from Supplier 2
Obtain quote from Supplier 3
Obtain quote from Supplier 4
Obtain quote from Supplier 5
PO request received
Select best quote
Emit order
Order emitted

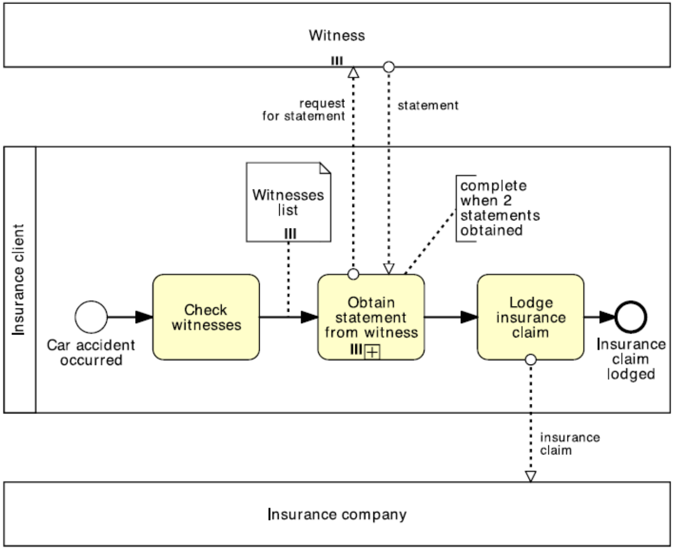
Note that: Basically, a multi-instance activity can be modeled through AND gateways. There are two problems with this model: (1) Readability (2) Updating

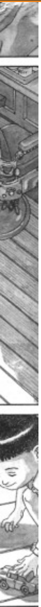
## Exercise



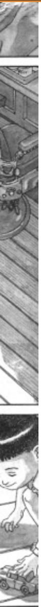
- After a car accident, a statement is required from two witnesses out of the five that were present, in order to lodge the insurance claim
- As soon as the first two statements are received, the claim can be lodged with the insurance company without waiting for the other statements

## Exercise


Witness
request for statement
statement
Insurance client
Witnesses list
complete when 2 statements obtained
Check witnesses
Obtain statement from witness
Lodge insurance claim
Car accident occurred
Insurance claim lodged
insurance claim
Insurance company



## One-To-Many

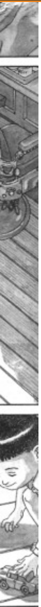


**Example**

Consider a hiring process that starts autonomously by posting information about a new job. The process continues by accepting applications (from an “Applicant” pool), interviewing candidates, ultimately hiring one of them.

Refine the process considering that only 5 selected candidates are interviewed.

## One-To-Many



**Example**

Consider a hiring process that starts autonomously by posting information about a new job. The process continues by accepting applications (from an "Applicant" pool), interviewing candidates, ultimately hiring one of them.

Refine the process considering that only 5 selected candidates are interviewed.

**Warning**

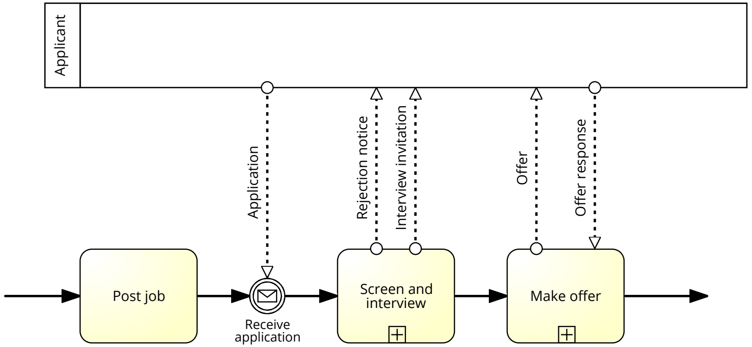
Process at two different granularities:

- Case: a job.
- Portion of the process: applicant (one job, many applicants).

The "many" part requires forms of synchronization.

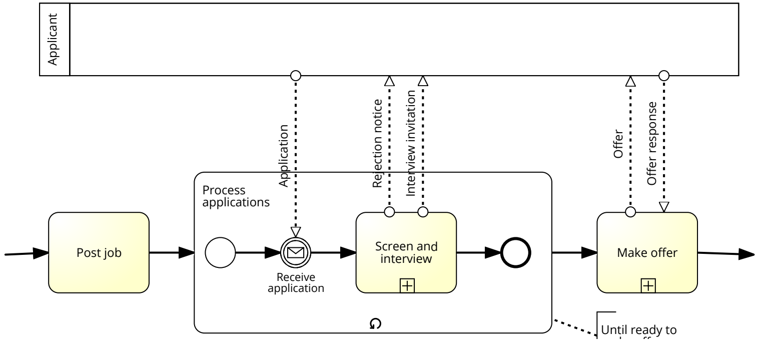
# One-To-Many

A common beginner mistake



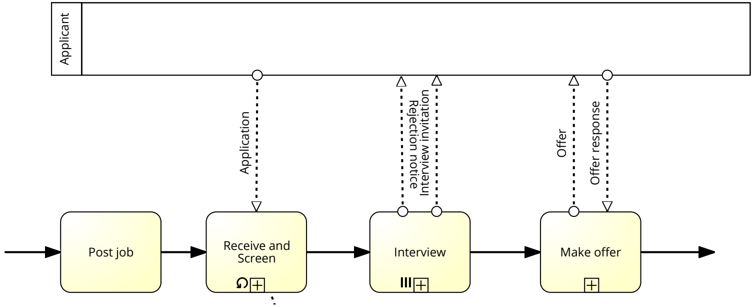
# One-To-Many

A valid but impratical solution

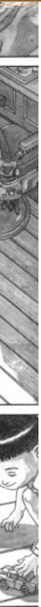


## One-To-Many

A more practical process model



## One-To-Many within a Pool



**Guideline**

1. Identify all N:1 activities.
2. Surround them in a loop or MI activity or a combination thereof.

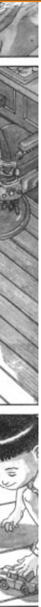
Main issues:

- If in a process phase we do not know the number of instances in advance, we can only use loop.
  - Pipelining impossible.
- If in a process phase we know the number of instances in advance, we can use MI.
  - Pipelining still impossible across phases.

**Example**

- Do we know the number of applicants in advance?
- Shall we keep “screen” and “interview” within the same loop?
- What if we want to interview only 5 candidates?
- Is it possible to pipeline “screen” and “interview”?

## Multipool Process



### Observation

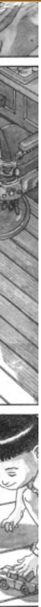
BPMN recommends to associate a process to a pool.
However, this cannot be done in general if activity instances within a process are not **aligned**, i.e., there is no 1:1 correspondence between them.

### Multipool process

In general a process consist of many **coordinated** pools, each grouping activities that are aligned with each other.

- N:1 situation: avoids looping + MIs with the corresponding pipelining issues.
- N:M situation: the only viable solution.

## Multipool Hiring Process



**Example**

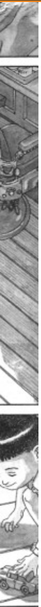
Consider again the hiring process.

- “Post a job” is 1:1 with the job (case).
- “Evaluate a candidate” is 1:1 with an applicant.

What is the relationship between applicant and job?

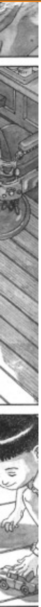
- N:1 (isolation): each applicant sends CV for a specific job.
  - Single pool solution requires looping + MIs, with pipelining issues.
  - Multipool solution avoids this.
- N:M (sharing): each applicant sends CV, and the company looks whether there are matching jobs for that CV.
  - Multipool separation necessary.

## Multipool Process: Needs and Issues



- Harder to understand.
- The process is actually split into independent processes: this requires **coordination** to synchronize the states of such processes and suitably reconstruct the overall process.
  - Data store, using data objects and their states for indirect synchronization/decision making.
  - Signals, to broadcast a control-flow state and implicitly trigger multi-pool synchronization.
  - Messages, to exchange data, hand-over work, and handle “selective” synchronization between two pools.
- Particularly useful to model **batch processes**.

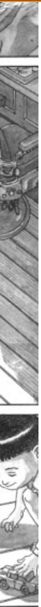
## Hiring Process and Multipool Coordination



**Example**

We revisit the hiring process using one external pool for the applicant, and two process pools: **Evaluate Candidate** and **Hiring Process**.
The Hiring process starts autonomously by posting a job. If the job is successfully assigned to someone, it is marked as “filled”. If this is not the case within 3 months, the job is marked as “abandoned”.
The Evaluate candidate process starts when a resume is received by the company from an applicant. We assume that the resume is for a specific job. If the job is not currently offered, the process terminates by notifying this to the applicant. If it is open, a subprocess “screen and interview” is invoked. Then the company decides whether to make an offer to the candidate or not. If not, this is notified to her. If so, then the offer is made to the candidate, who can then accept or reject the offer. As soon as the job is filled, the evaluation of candidates must be interrupted.

## Hiring Process and Multipool Coordination



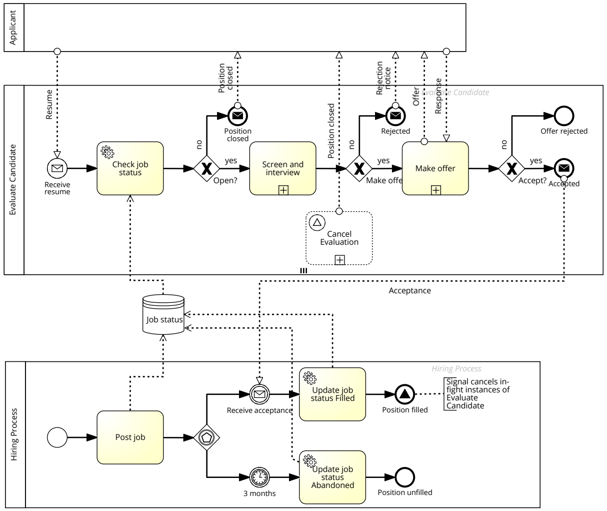
**Guideline**

Hiring process could be organized as follows:

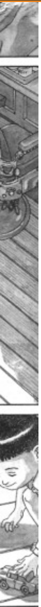
- 1 external pool for the Applicant, two process pools: Hiring Process and Evaluate Candidate.
- Job Status data store to synch on the job lifecycle.
- Message exchange to send data around and take decisions in agreement with the other pools.
- Signal event + event subprocess to interrupt sibling cases running in the other pools (e.g., to stop evaluating people when the job has been filled).

# One-To-Many

## Multi-pool solution to the hiring process problem



## Batch Process



Common situation where the process must be captured via multiple pools.

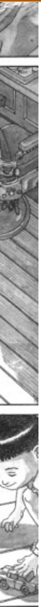
**Batch process**

A process operating over a **batch** of items, which are handled separately in another process (i.e., the other process treats them as cases).

- Typically, batch processing is triggered by a time event.
- There are cases in which the result of batch processing is needed also in the “item-level” process.

Whenever the batch-level process needs to interact with the item-level process, the same forms of coordination seen before can be used.

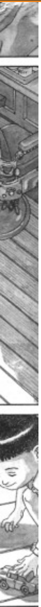
## Batch Posting of Orders

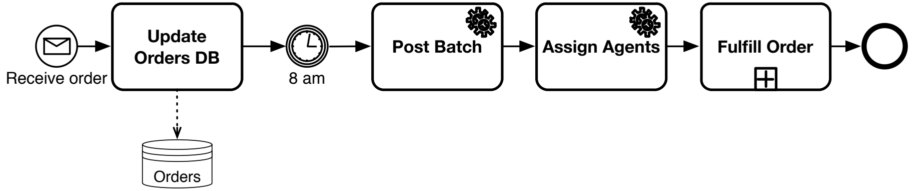


**Example**

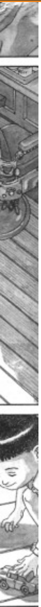
Once an order is received from the customer, it is registered in the “Orders” DB. Orders are posted on a daily basis: at 8am in the morning, all the orders issued during the previous 24 hours are posted. As a consequence of the posting, each order is automatically assigned to an agent, storing this information into the DB. Then the agent takes care of fulfilling the order.

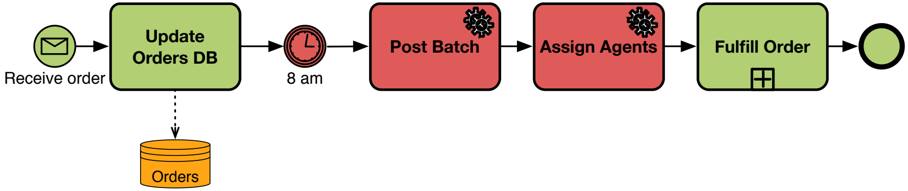
## Solution?




Receive order
Update Orders DB
Orders
8 am
Post Batch
Assign Agents
Fulfill Order

## Solution?

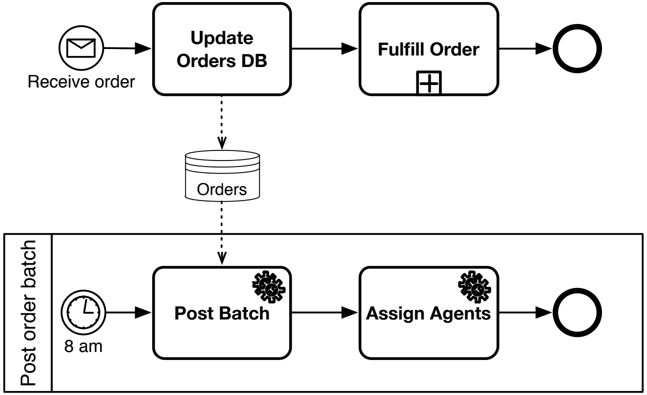




- Green: one order.
- Red: many orders.
- Orange: all known orders.

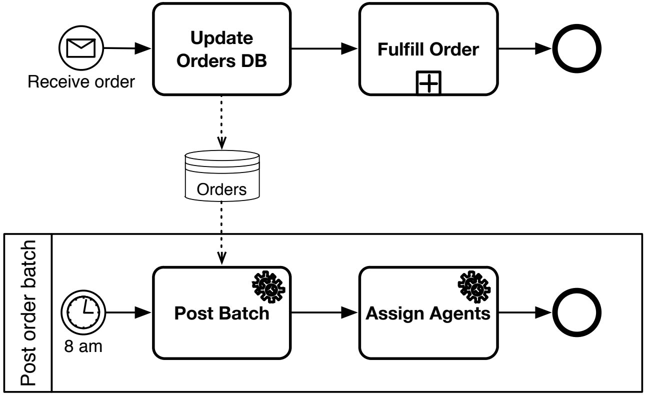
## Solution?!?

Data-store provides a suitable **loose coupling** mechanism.



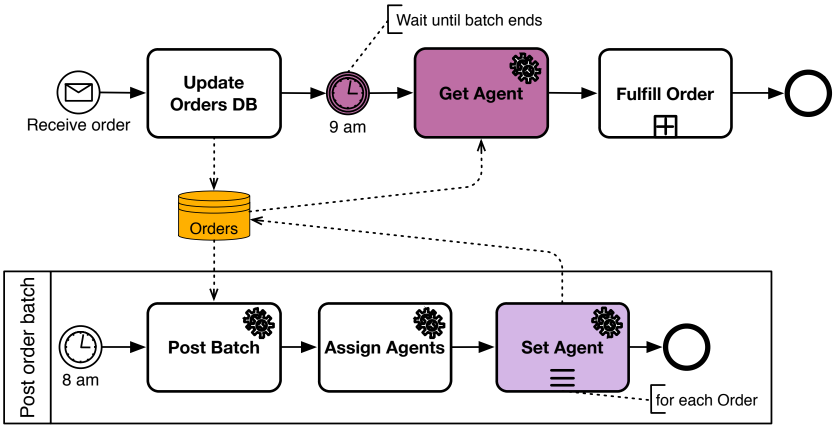
## Solution?!?

Data-store provides a suitable **loose coupling** mechanism.

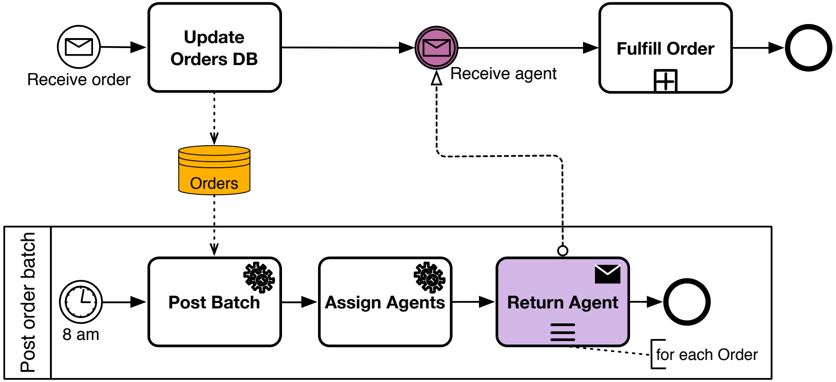


Who fulfills the order? The order is fulfilled without waiting for the agent assignment: race conditions.

# Solution 1: Stopwatch + Data-Store Synchronization

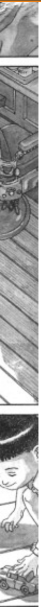


## Solution 2: Message-Based Synchronization



If the fact that an agent is set must be broadcasted: throw signal event!

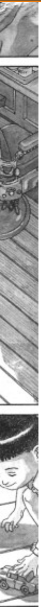
## Advanced Splitting and Merging



BPMN provides advanced splitting/merging behavior.

- Conditional flows.
- OR split/join.
- Complex gateway: discriminator, n-out of-m, . . .

# An Interesting Example



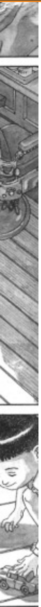
**Claim handling**

When a claim is received, it is registered. After registration, the claim is classified leading to two possible outcomes: simple or complex. If the claim is simple, the policy is checked. For complex claims, both the policy and the damage are checked independently.

**Questions**

- Can we model the example using the constructs seen so far?
- Can we model it in a compact way?
- Is this a decision? Or an AND-split?

## Another Interesting Example



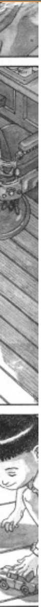
**Contract Check**

After a contract is drafted, it is subject to different checks. If it is a technical contract, a technical review is done. If it involves a financial transaction of more than 10K Euros, then a financial check is carried out. If the contract involves a premium customer, then a quality check is executed. If none of these special conditions applies, then a quick check suffices.

**Questions**

- Can we model the example using the constructs seen so far?
- Can we model it in a compact way?
- Is this a decision? Or an AND-split?

## Conditional Flow



Sequence flow taken only if a certain condition is true. Multiple conditional flows departing from the same activity can be used to model **inclusive choice**. BPMN supports conditional flows in two ways.

**OR-split (gateway decorated with an "O")**

Inclusive gateway whose gates are **independent** choices.

- At least one condition must be true.
- If more than one is true, corresponding gates are followed in parallel.
- Default flow (barred) can be used to model "otherwise".
- Can be used to model optional work.
  - Use label "always" for the path that must be always taken.

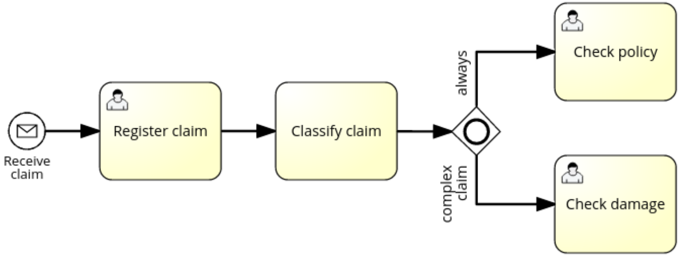
**Conditional flow (Sequence flow + diamond-on-the-tail + condition)**

Reflects the general intuition above. (1) Can only be used if the source is an activity. (2) Same properties mentioned for the OR-split.

## Claim Handling with OR-Split

**Example**

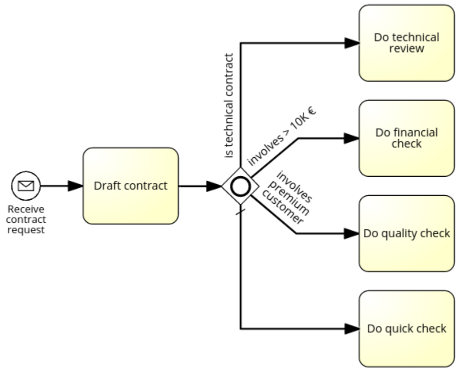
When a claim is received, it is registered. After registration, the claim is classified leading to two possible outcomes: simple or complex. If the claim is simple, the policy is checked. For complex claims, both the policy and the damage are checked independently.



## Contract Check with Conditional Flow

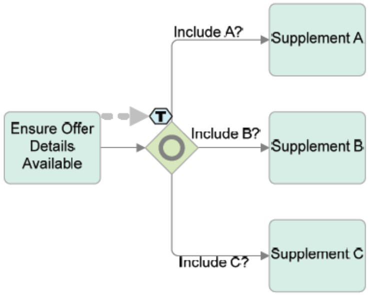
**Example**

After a contract is drafted, it is subject to different checks. If it is a technical contract, a technical review is done. If it involves a financial transaction of more than 10K Euros, then a financial check is carried out. If the contract involves a premium customer, then a quality check is executed. If none of these special conditions applies, then a quick check suffices.



# Token Game: OR split[5]

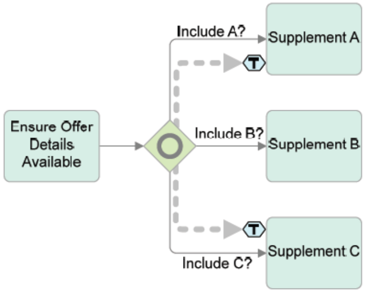
- Inclusive gateways support decisions where more than one outcome is possible at the decision point
- Inclusive gateway with multiple outgoing sequence flows creates one or more paths based on the conditions on those sequence flow



[5]Credits: Andrea Marrella "Modeling Business Processes with BPMN"

## Token Game: OR split[5]

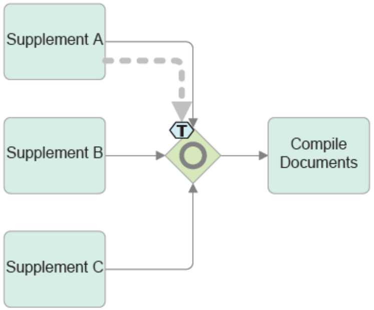
- In terms of token semantics, this means that the OR-split takes the input token and generates a number of tokens equivalent to the number of output conditions that are true
- Every condition that evaluates to true will result in a token moving down that sequence flow
- At least one of those conditions must evaluate to true



[5]Credits: Andrea Marrella “Modeling Business Processes with BPMN"

## Token Game: OR join[6]

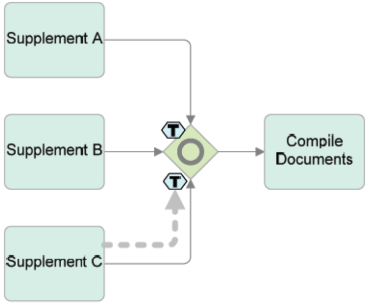
- When the first token arrives at the gateway, the gateway will “look" upstream for each of the other incoming sequence flow to see if there is a token that might arrive at a later time
- Thus, the gateway will hold the first token that arrived in the upper path until the other token from the lower path arrives.



[6]Credits: Andrea Marrella “Modeling Business Processes with BPMN"

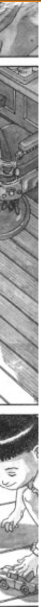
## Token Game: OR join[6]

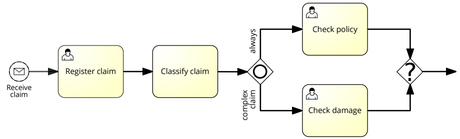
- When all the **expected** tokens have arrived at the gateway, the process flow is synchronized (the incoming tokens are merged) and then a token moves down the gateway's outgoing sequence flow

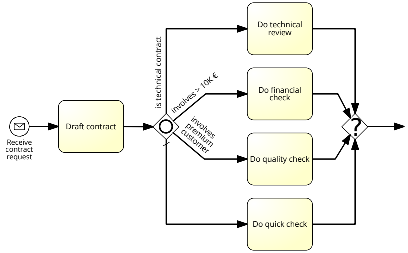


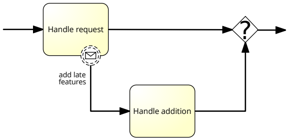
[6]Credits: Andrea Marrella "Modeling Business Processes with BPMN"

## How to Join?

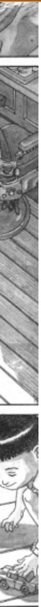


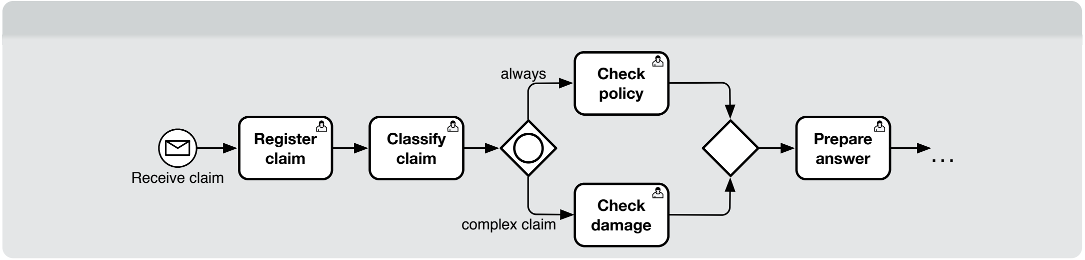

Receive claim
Register claim
Classify claim
always
complex claim
Check policy
Check damage
?

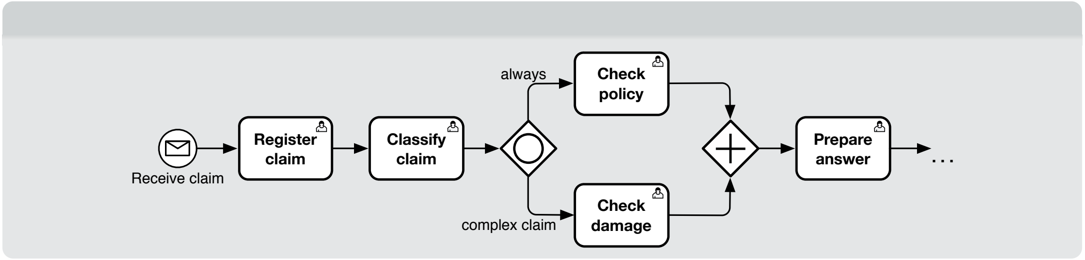

Receive contract request
Draft contract
is technical contract
involves > 10K €
involves premium customer
Do technical review
Do financial check
Do quality check
Do quick check
?


Handle request
add late features
Handle addition
?

## Useless Attempts

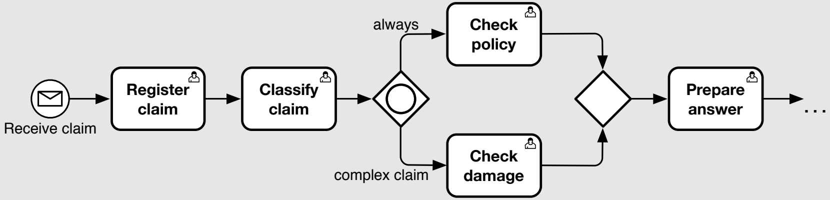




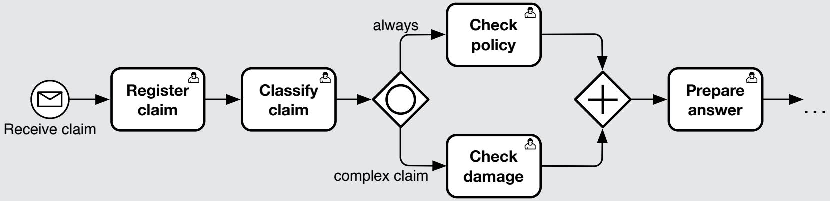


# Useless Attempts

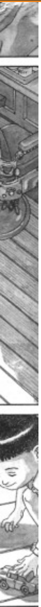
## Two answers



## No answer



## OR-Join



### OR-Join

Selectively synchronizes conditionally enabled parallel threads.

- **Non-local semantics**: needs to know which parallel threads actually exist and will need to be synchronized in the future.
- Decides to wait or forward work depending on the presence or absence of remaining threads to be synchronized.

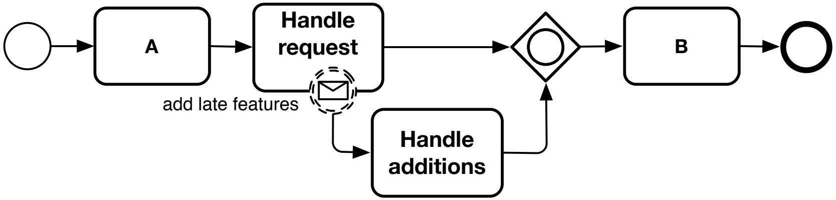
Typical usages:

- Selective synch corresponding to an inclusive choice.
- Synch in case of interrupting exceptional flow.
- Synch in case of non-interrupting exceptional flow.

### Big Question

How to determine such threads?

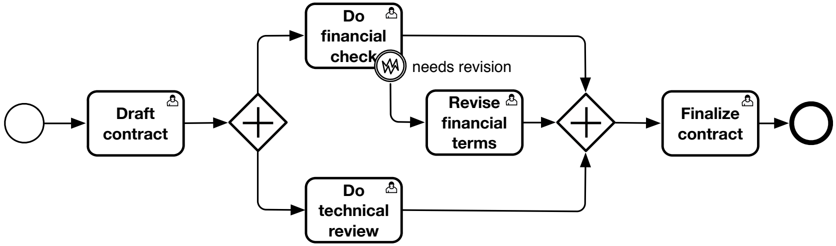
## OR-Join for Non-Interrupting Exceptional Flow



**B** executed only if the request is completed, possibly including additions.

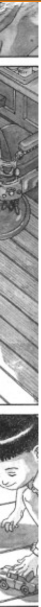
- Remember: additions are only handled if the message is received in the proper process phase. We cannot know in advance whether this will be indeed the case.

## OR-Join for Interrupting Exceptional Flow



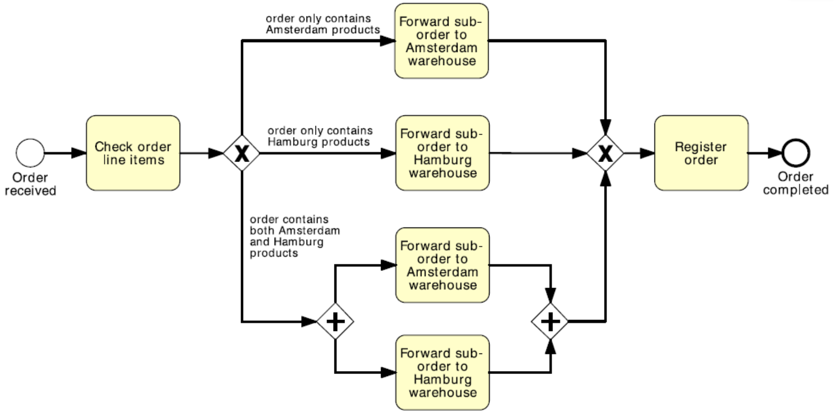
We do not know in advance whether the technical review will need to synch with the financial review or the revision of financial terms.

## Exercise



- A company has two warehouses that store different products: Amsterdam and Hamburg
- When an order is received, it is distributed across these warehouses: if some of the relevant products are maintained in Amsterdam, a sub-order is sent there; likewise, if some relevant products are maintained in Hamburg, a sub-order is sent there.
- Afterwards, the order is registered and the process completes

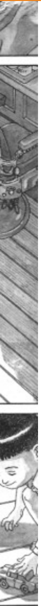
# Exercise

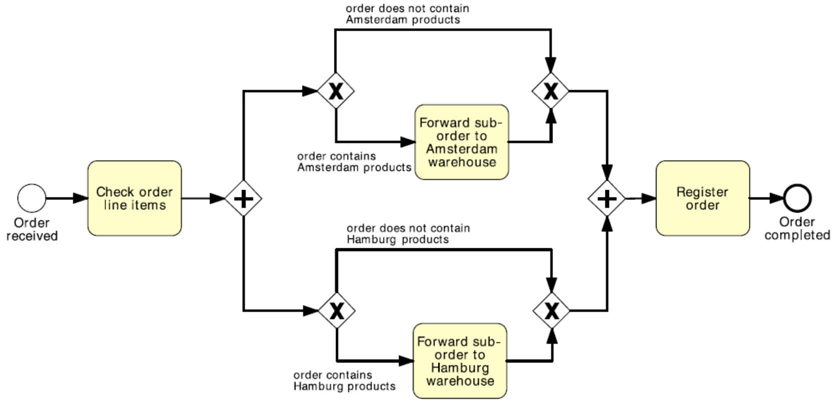

order only contains Amsterdam products
Forward sub-order to Amsterdam warehouse
Order received
Check order line items
order only contains Hamburg products
Forward sub-order to Hamburg warehouse
Register order
Order completed
order contains both Amsterdam and Hamburg products
Forward sub-order to Amsterdam warehouse
Forward sub-order to Hamburg warehouse

**Note:**

Some activities represented in the process model have to be duplicated!

## Exercise

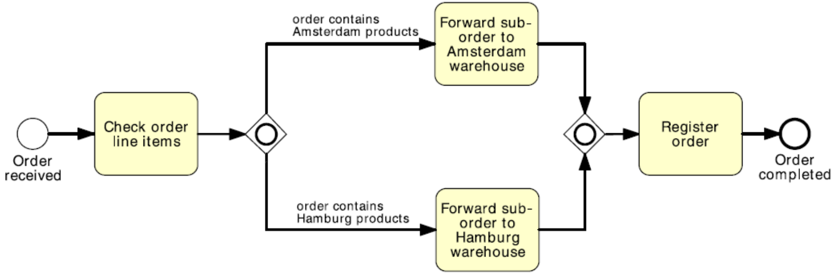



order does not contain Amsterdam products
Forward sub-order to Amsterdam warehouse
order contains Amsterdam products
Order received
Check order line items
Register order
Order completed
order does not contain Hamburg products
Forward sub-order to Hamburg warehouse
order contains Hamburg products

**Note:**

This process works also for empty orders (i.e., for orders that do not contain neither Amsterdam nor Hamburg products).

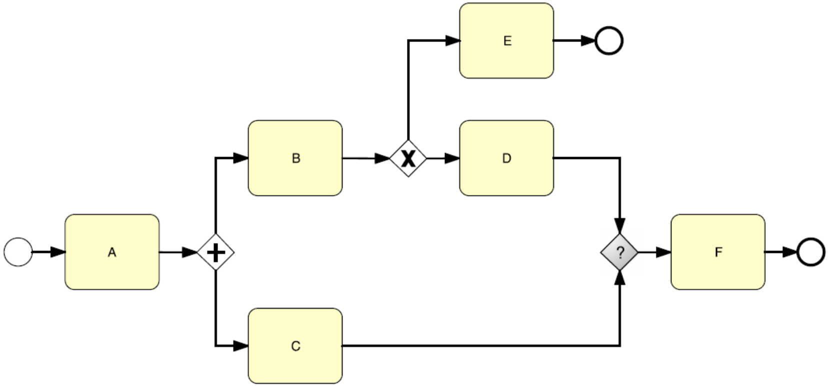
# Exercise


order contains Amsterdam products
Forward sub-order to Amsterdam warehouse
Order received
Check order line items
Register order
Order completed
order contains Hamburg products
Forward sub-order to Hamburg warehouse

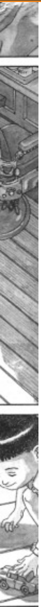
A third solution with OR gateways.

## Exercise

What type should we assign to this join?

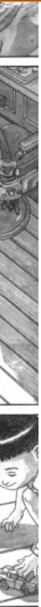


## Exercise



- **AND-join**: If activity “D” is not executed, the AND-join will wait indefinitely for that token, with the consequence that the process instance will not be able to progress any further. This behavioral anomaly is called **deadlock** and should be avoided
- **XOR-join**: we may execute activity “F” once or twice, depending whether the preceding XOR-split routes the token to “E” (in this case “F” is executed once) or to “D” (“F” is executed twice). While this solution may work, we have the problem that we do not know whether activity “F” will be executed once or twice, and we may actually not want to execute it twice. Moreover, if this is the case, we would signal that the process has completed twice, since the end event following “F” will receive two tokens. And this, again, is something we want to avoid

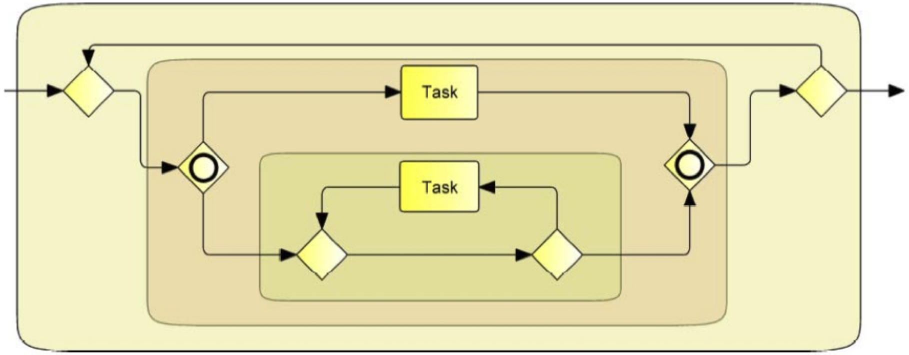
## Exercise



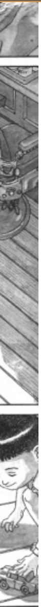
- **OR-join**: An OR-join will wait for all incoming active branches to complete. If the XOR-split routes control to “E”, the OR-join will not wait for a token from the branch bearing activity “D”, since this will never arrive. Thus, it will proceed once the token from activity “C” arrives. On the other hand, if the XOR-split routes control to “D”, the OR-join will wait for a token to also arrive from this branch, and once both tokens have arrived, it will merge them into one and send this token out, so that “F” can be executed once and the process can complete normally

## Block-Structured Process Model - Intuition

The graph structure can be understood as composition of blocks.



## Block-Structured Process Model



A process model obtained via recursive composition of **SESE blocks**.

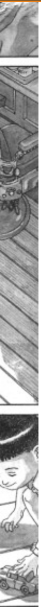
**SESE block**

A process fragment that has a *single input* and a *single output*, and can be abstracted as a subprocess. Block types:

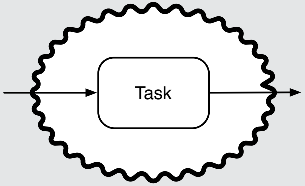
- end-to-end process.
- Single task.
- Sequence block.
- Choice block (X).
- Parallel block (+).
- Inclusive block (O).
- Loop block.

This is just a guideline, we typically need at least some exceptions for end points and exception handling.

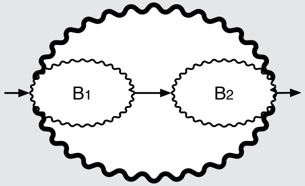
## SESE Blocks



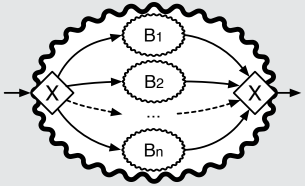
### Task



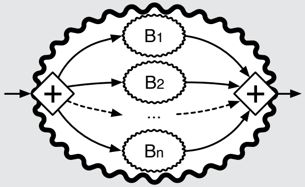
### Sequence



### Choice

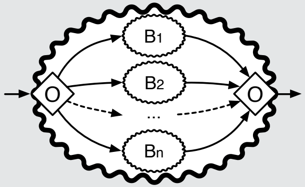


### Parallel

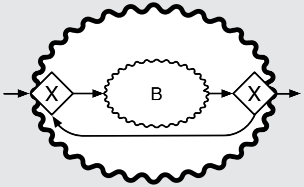


## SESE Blocks

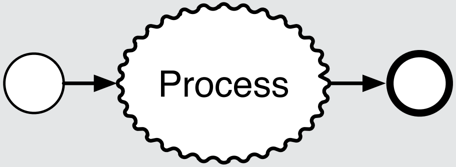
**Inclusive**



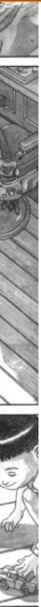
**Loop**



**End-to-end process**



## OR-Join Semantics for SESE Block



Quite simple:

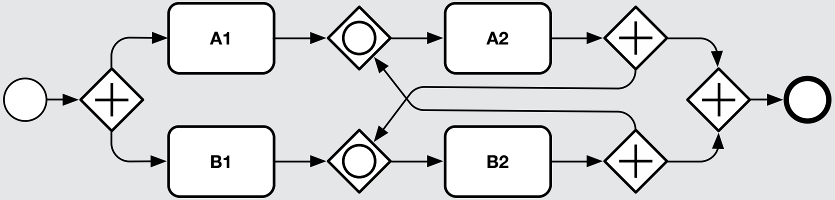
1. When the OR-split of the block is reached, the enabled gates are determined.
2. Each enabled gate has a 1:1 correspondence with a sequence flow entering into the OR-join of the block.
3. The OR-join therefore knows which execution threads need to be synchronized, and **where** they are located.
4. It can be then realize as a simple variant of an AND-join.

This can be generalized to non-block-structured processes, provided that they do not contain loops indirectly producing work that goes back into the OR-join.

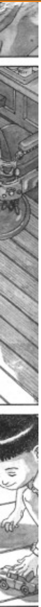
## OR-Join in the General Case

Do you like paradoxes?

**Vicious circle (adapted from van der Aalst et. al)**



## General Execution Semantics for OR-Join



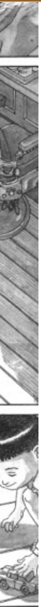
From the BPMN official documentation. . .

An Inclusive Gateway is activated if:

- **At least one** incoming Sequence Flow has *at least one* token **and**
- **For every** directed path formed by sequence flow that
  - starts with a Sequence Flow $f$ of the diagram that has a token,
  - ends with an incoming Sequence Flow of the inclusive gateway that has *no token*,
  - does not visit the Inclusive Gateway

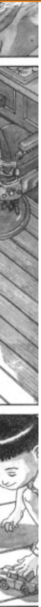
  **then there is also a** directed path formed by Sequence Flow that
  - starts with $f$,
  - ends with an incoming Sequence Flow of the inclusive gateway that has a token, and
  - does not visit the Inclusive Gateway.

## When should we use an OR-join?



- Since the OR-join semantics is not simple, the presence of this element in a model may confuse the reader
- Thus, we suggest to use it only when it is strictly required

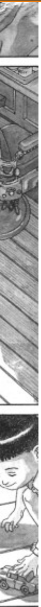
## When should we use an OR-join?



- Since the OR-join semantics is not simple, the presence of this element in a model may confuse the reader
- Thus, we suggest to use it only when it is strictly required

Clearly, it is easy to see that an OR-join must be used whenever we need to synchronize control from a preceding OR-split. Similarly, we should use an AND-jointo synchronize control from a preceding AND-split and an XOR-join to merge a set of branches that are mutually exclusive.

## Other Forms of Synchronization



Consider another variant of the contract example seen before.

**Example**

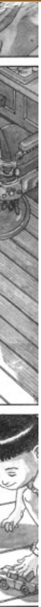
Once a contract is drafted, a financial and technical reviews are conducted in parallel. As soon as one of the two ends, an executive review must be conducted as well. When all reviews are finished, the contract is finalized.

**Question**

How to enable an activity as soon as the first among two other activities completes?

- A simple merge does not work, because it would be triggered multiple times!

## Discriminator



**Discriminator**

Passes the **first** incoming sequence flows and blocks all those that arrive later.

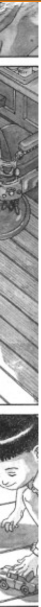
- It resets when all incoming flows arrive.

It does not have a specific symbol in BPMN. There is however the notion of “complex gateway” ($*$ decoration) to represent advanced forms of synchronization like the discriminator one.

**Generalization**

N-out of-M join.

# Contract Drafting with Discriminator



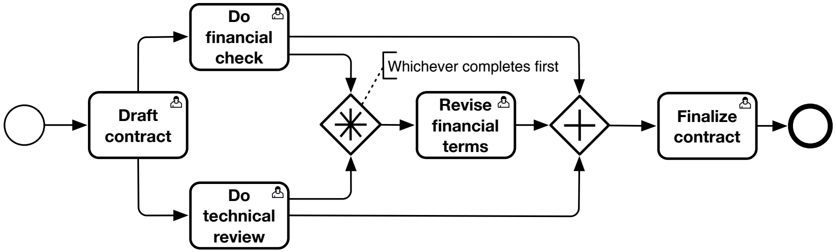
**Example**

Once a contract is drafted, a financial and technical reviews are conducted in parallel. As soon as one of the two ends, an executive review must be conducted as well. When all reviews are finished, the contract is finalized.

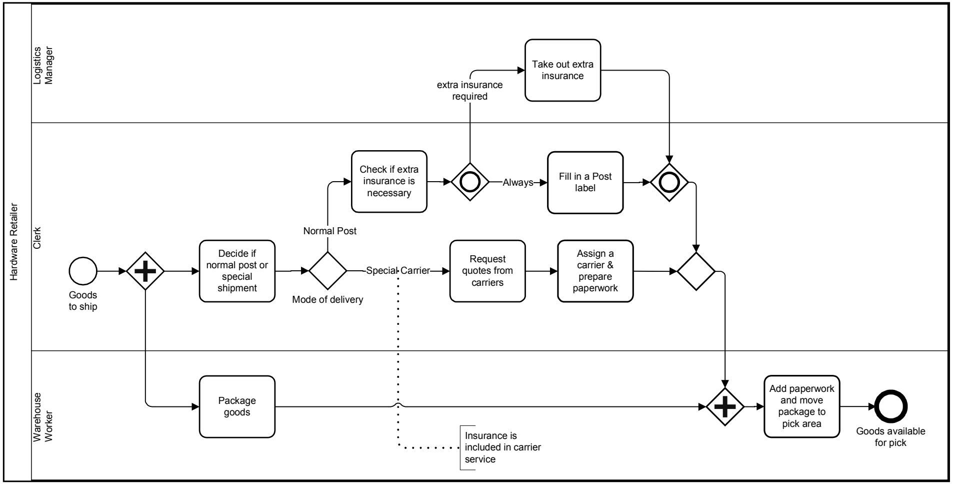
## Contract Drafting with Discriminator

**Example**

Once a contract is drafted, a financial and technical reviews are conducted in parallel. As soon as one of the two ends, an executive review must be conducted as well. When all reviews are finished, the contract is finalized.

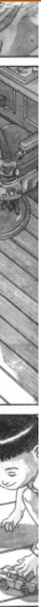


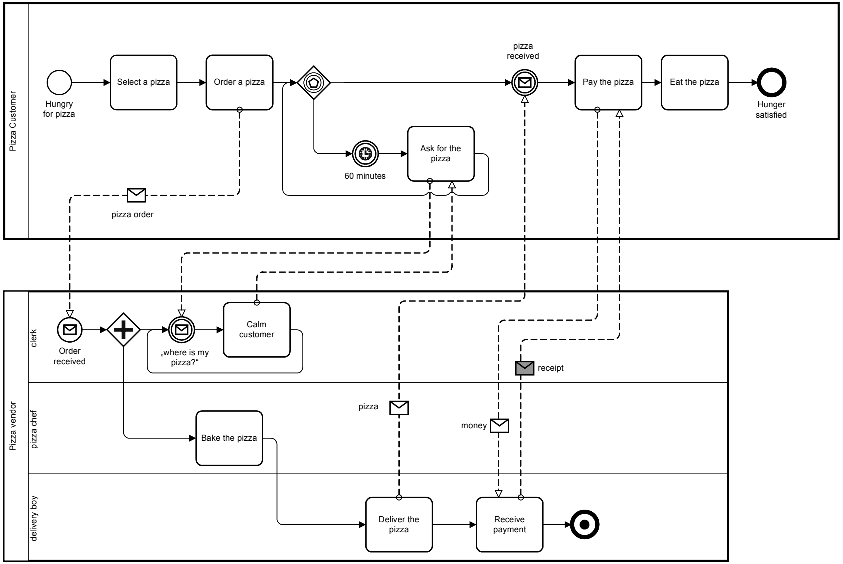
## BPMN - Hardware Retailer Example


Hardware Retailer
Logistics Manager
Clerk
Warehouse Worker
Goods to ship
Decide if normal post or special shipment
Mode of delivery
Normal Post
Special Carrier
Check if extra insurance is necessary
extra insurance required
Take out extra insurance
Always
Fill in a Post label
Request quotes from carriers
Assign a carrier & prepare paperwork
Package goods
Insurance is included in carrier service
Add paperwork and move package to pick area
Goods available for pick

From *BPMN 2.0 by Example* - `http://www.bpmn.org/`

# BPMN - Pizza Example




Pizza Customer
Hungry for pizza
Select a pizza
Order a pizza
pizza received
Pay the pizza
Eat the pizza
Hunger satisfied
60 minutes
Ask for the pizza
pizza order
Pizza vendor
clerk
Order received
„where is my pizza?“
Calm customer
receipt
pizza chef
pizza
money
Bake the pizza
delivery boy
Deliver the pizza
Receive payment