BPMN 2.0 Symbol Reference

All BPMN 2.0 Symbols explained with examples.

Online Modeler

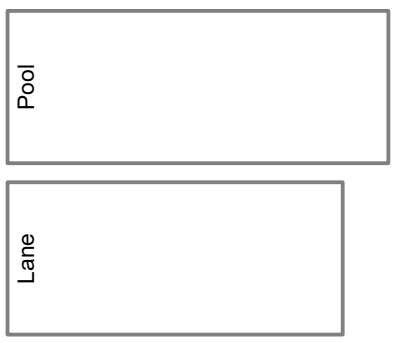
Our modeler is an online and desktop tool for designing, discussing and sharing BPMN diagrams with your team.

Try Modeler

Overview

BPMN Symbol Overview

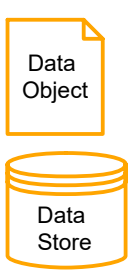
Participants



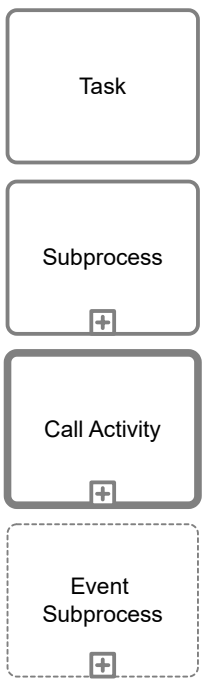
Artifacts

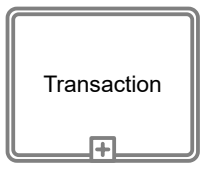
Gateways

Data



Activities





Events

Heads up!
For understanding the principle behavior of events in BPMN, check out **Events: Basic Concepts**.

	Start			Intermediate			
Type	**Normal**	**Event Sub process**	**Event Sub process non-interrupt**	**Catch**	**Boundary**	**Boundary non-interrupt**	**Throw**
None	◯						
Message	✉	✉	✉	✉	✉	✉	✉
Timer	🕒	🕒	🕒	🕒	🕒	🕒	
Conditional	☰	☰	☰	☰	☰	☰	
Link				⇨			➡
Signal	△	△	△	△	△	△	▲
Error		⚡			⚡		
Escalation		▲	▲		▲	▲	▲
Termination							
Compensation		⏪			⏪		⏪
Cancel					✕		
Multiple	⬠	⬠	⬠	⬠	⬠	⬠	⬟
Multiple Parallel	✚	✚	✚	✚	✚	✚	

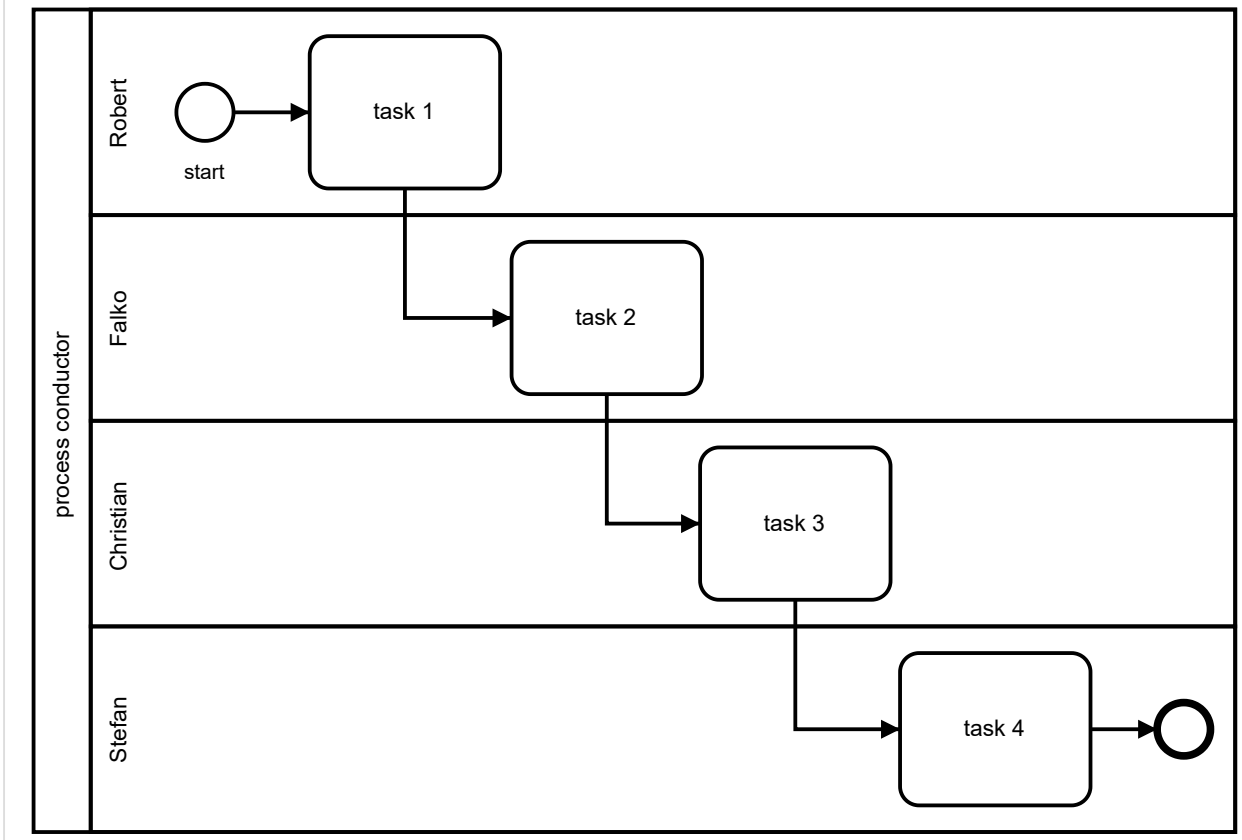
Participants

Introducing Pools, the conductor and the orchestra

We already described how to use lanes to assign responsibility for tasks or subprocesses to different task managers. Lanes always exist in a pool, and the lane boundaries represent process boundaries from start to end. To BPMN, the pool represents a higher-ranking instance compared to its lanes. The

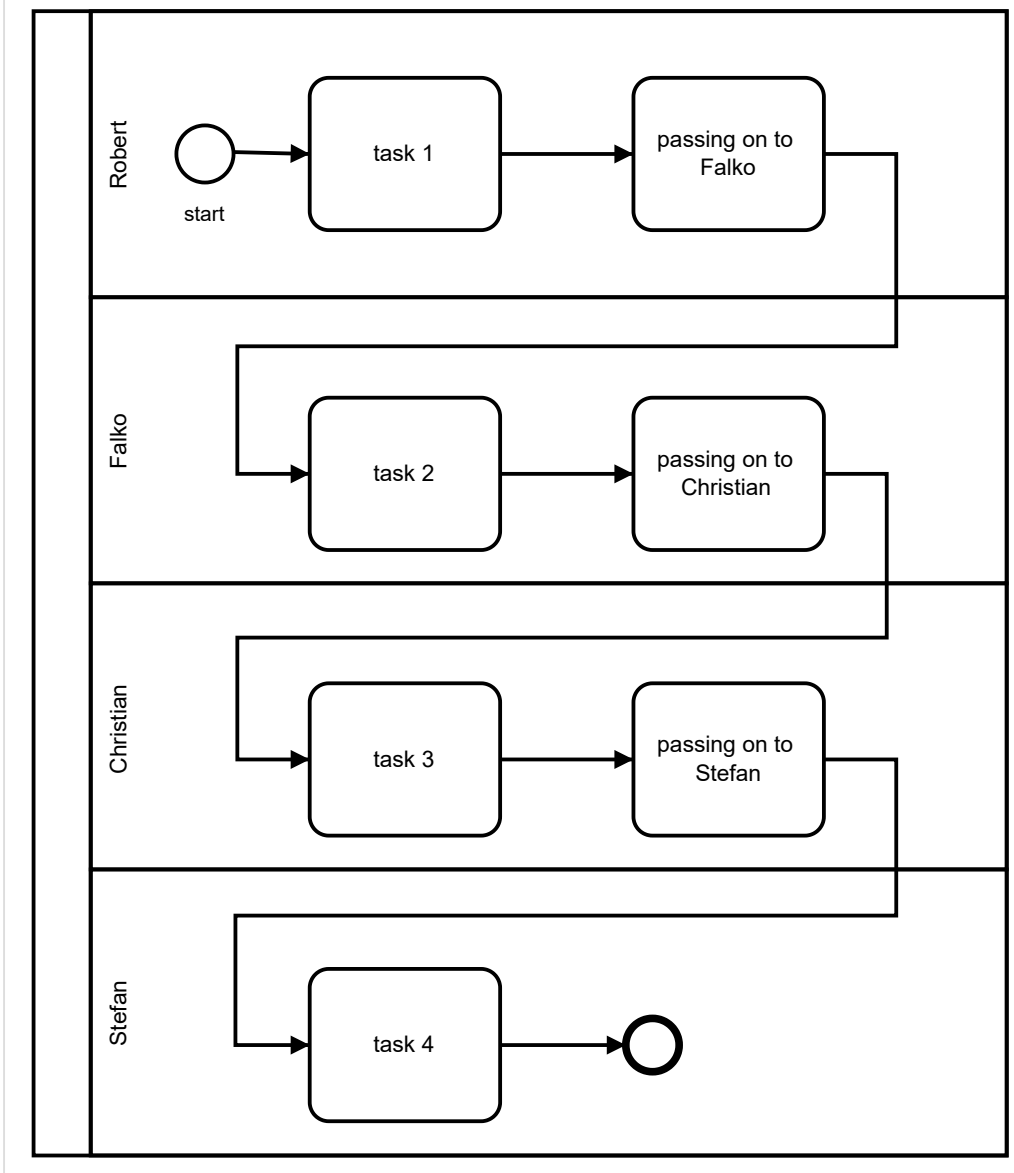
pool assumes process control – in other words, it assigns the tasks. It behaves like the conductor of an orchestra, and so this type of process is called “orchestration.”

In the diagram below, the “conductor” arranges for Falko to process task 2 as soon as Robert completes task 1. The conductor has the highest-level control of the process, and each instrument in the orchestra plays the tune the conductor decides upon:

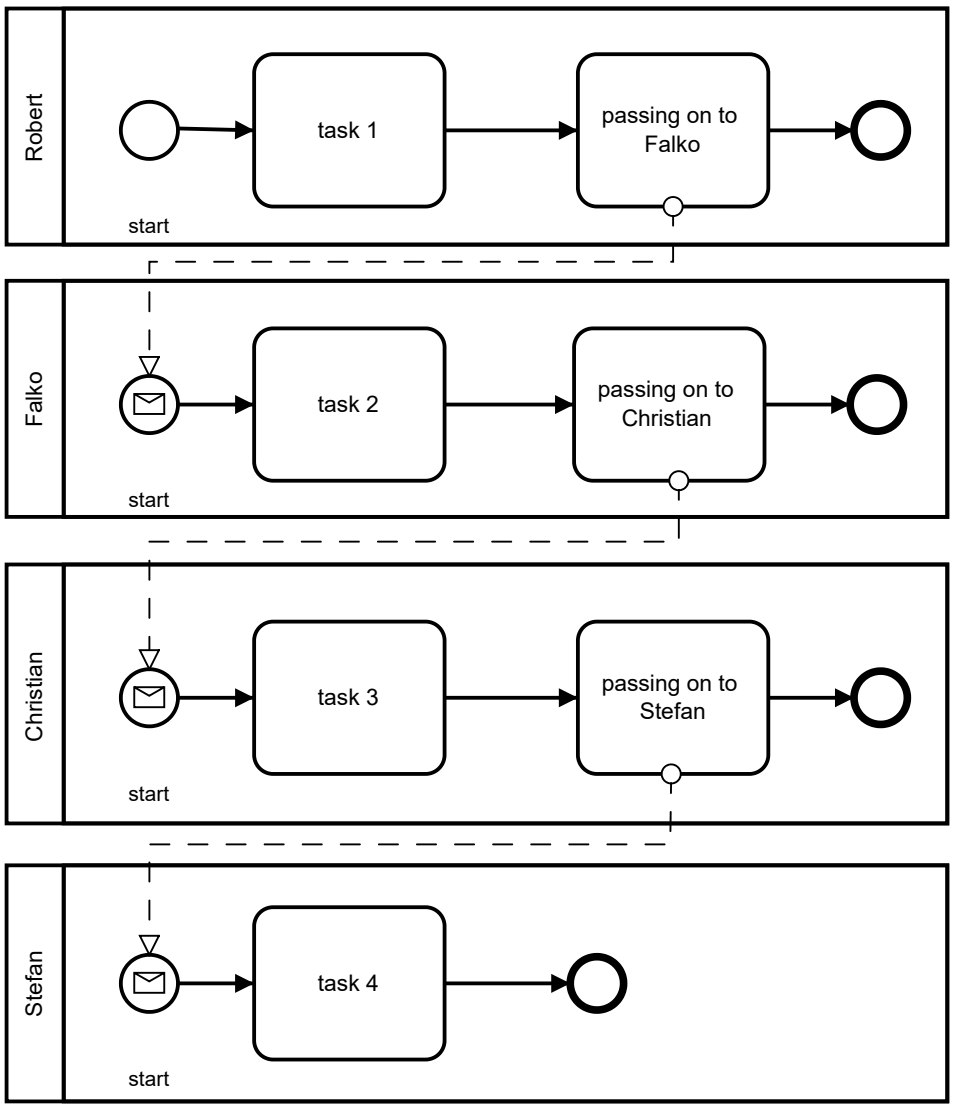


Do you think this is unrealistic? Many experienced process modelers have problems with this way of thinking. They would prefer to model a process sequence like that shown below on the assumption that no almighty conductor exists in their company, and that individual process participants have to coordinate and cooperate on their own:

In the diagram below, the “conductor” arranges for Falko to process task 2 as soon as Robert completes task 1. The conductor has the highest-level control of the process, and each instrument in the orchestra plays the tune the conductor decides upon:



But to coordinate cooperation with BPMN requires explicit modeling. You assign each task manager a separate pool, and the process passes from one to the next as a message flow as shown in below. In principle, this creates four independent conductors. These have control over their respective mini-processes, but they can't do anything other than to send messages that trigger their successor processes:



That seems complicated – and you don't have to choose this method for practical modeling. It reveals a basic principle, however, that you must understand. Even though BPMN lanes look very much like those of other process notations, they represent an entirely different way of thinking, which we attribute to BPMN's origin in the world of process automation. In that world, the process engine controls all tasks in the process, even though different task managers may execute them. So the process engine equates to the mysterious, almighty process conductor.

Have you heard of service orchestration in connection with Service Oriented Architecture (SOA)? That's almost exactly the task of a process engine, except that these services are not only fully automated web services; they also can be tasks executed by human process participants as directed by the process engine. What does that signify, however, for purely functional process modeling, in which you also describe processes **not** controlled by such a process engine? There's no general answer to that question.

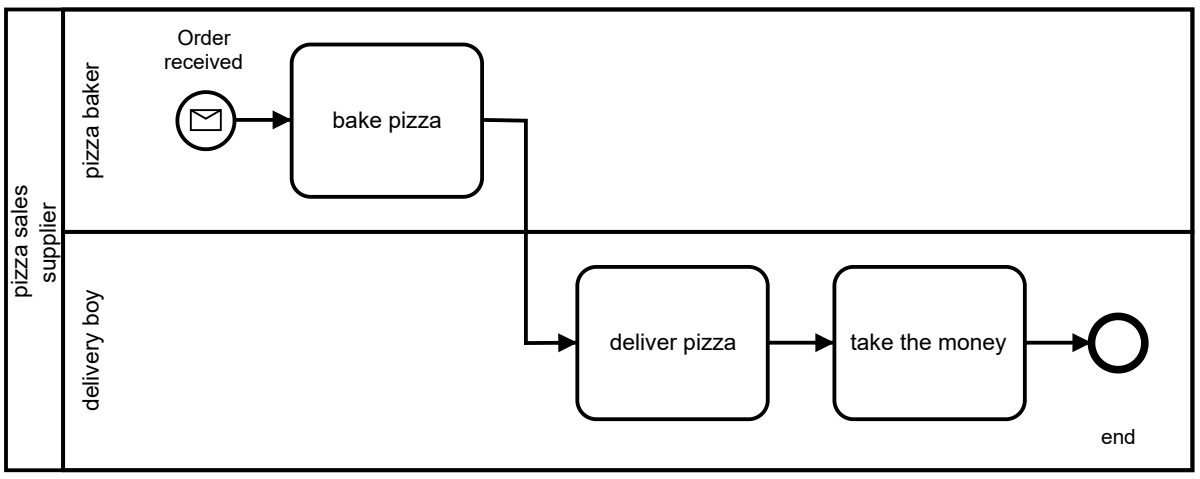
You can eliminate pools and work just with lanes, modeling the message exchange as normal tasks as shown before. That's traditional, and it's a pragmatic solution during, say, a transitional period that allows your co-workers to adapt. In the medium and long terms, however, avoiding pools denies you a powerful device for increasing the significance of process models.

We will show the usefulness of this new thinking by example. One thing to remember is that if you strive to harmonize your functional and technical process models to achieve a better alignment of business and IT, you inevitably face this type of process modeling whether you use BPMN or not.

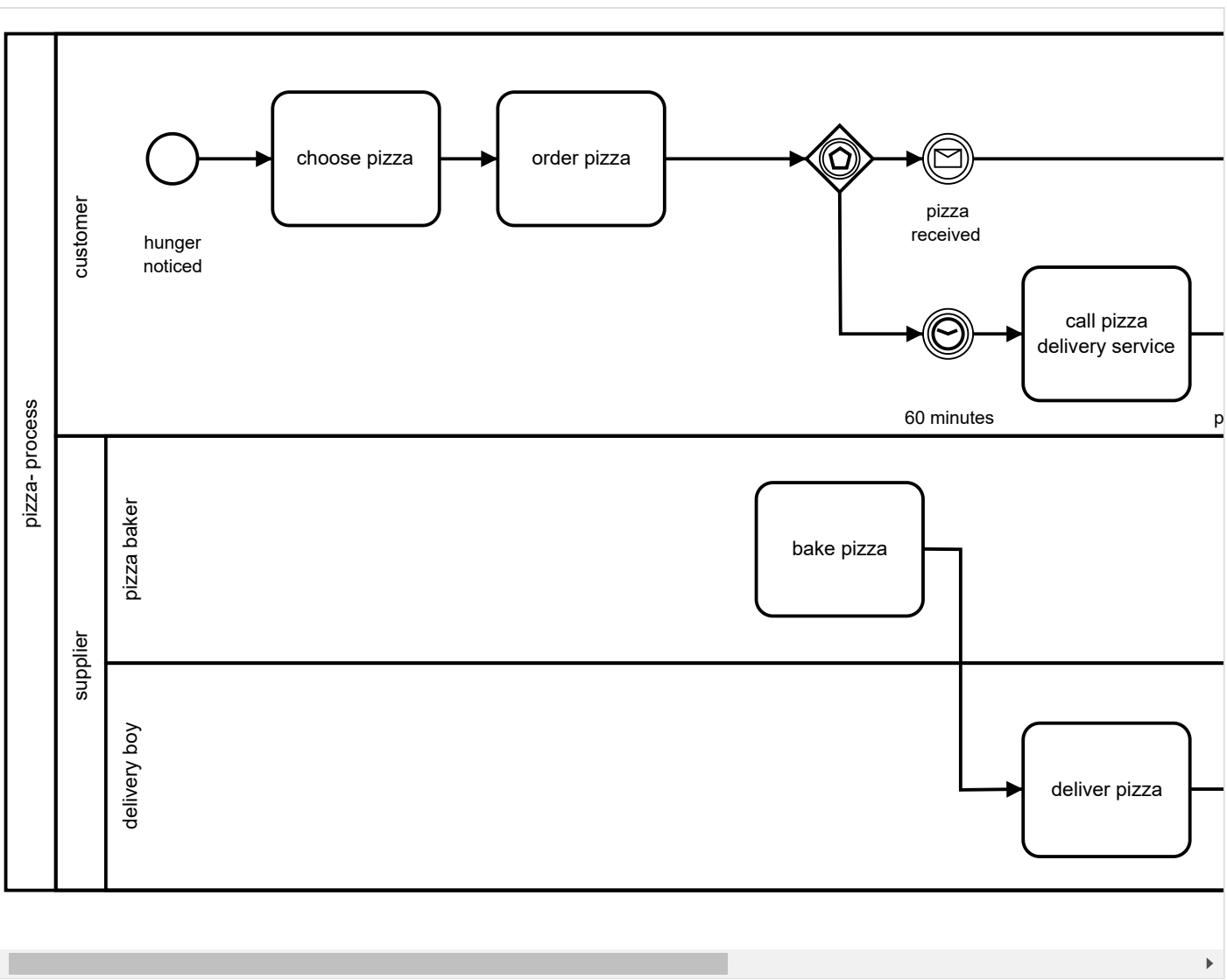
Pools: The art of collaboration

We already examined the process represented below in connection with the event-based gateway:

Now consider the broader picture, and think about how this process happens from the point of view of the pizza delivery service. Presumably, it looks like here: As soon as we receive an order, we bake the pizza. Our delivery person takes it to the customer and collects the money, whereby the process completes successfully.

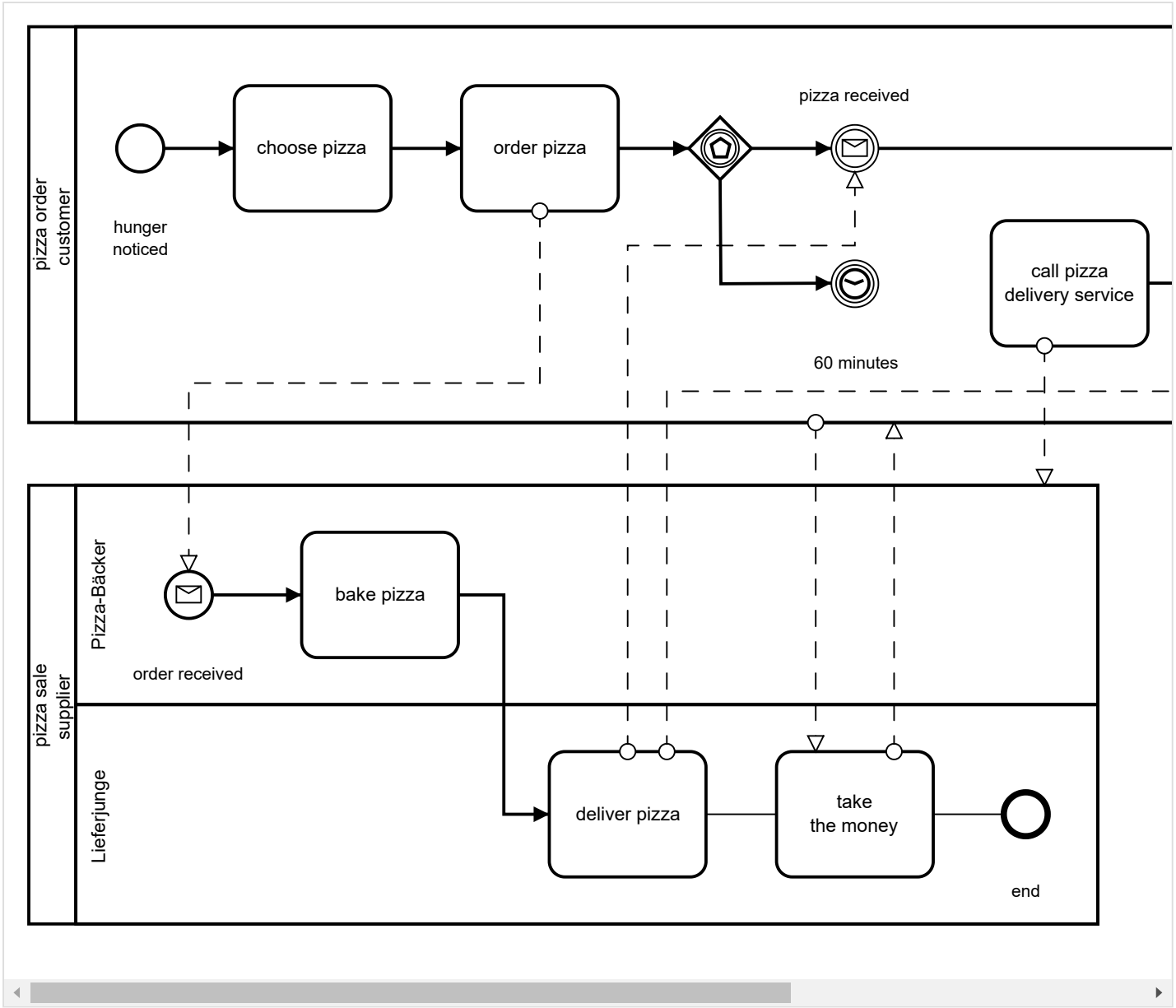


We want to link the two processes, that is, to examine the interaction of customer and delivery service from a neutral perspective. We can try to model this interaction by means of a pool and lanes as in here:



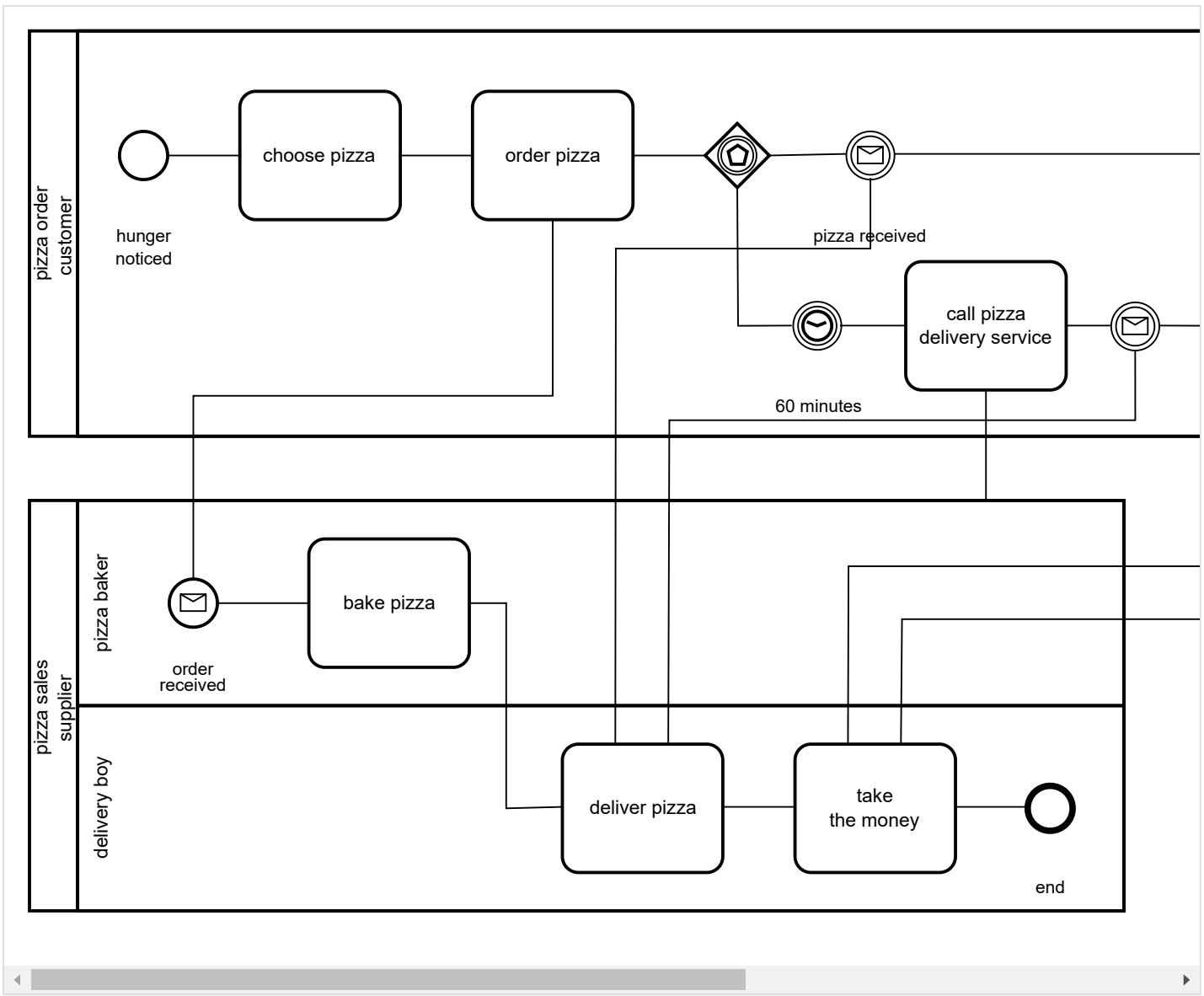
But this doesn't work well: There are tasks and events that reference interaction within the pool – waiting for the delivery, for instance, or collecting the money. Other tasks are carried out by roles oblivious to their partners, such as baking the pizza and eating the pizza. It is impossible to differentiate the two visually. Strictly speaking, the diagram is not semantically correct because message events always refer to messages received by the process from outside, and that's not the case here.

If we go with pools, the whole process looks like below. Both processes in the combined representation would look just as they did before, but now they connect through message flows. BPMN calls this form of visualization a collaboration diagram. It shows two independent processes collaborating.



But this doesn't work well: There are tasks and events that reference interaction within the pool – waiting for the delivery, for instance, or collecting the money. Other tasks are carried out by roles oblivious to their partners, such as baking the pizza and eating the pizza. It is impossible to differentiate the two visually. Strictly speaking, the diagram is not semantically correct because message events always refer to messages received by the process from outside, and that's not the case here.

If we go with pools, the whole process looks like below. Both processes in the combined representation would look just as they did before, but now they connect through message flows. BPMN calls this form of visualization a collaboration diagram. It shows two independent processes collaborating.



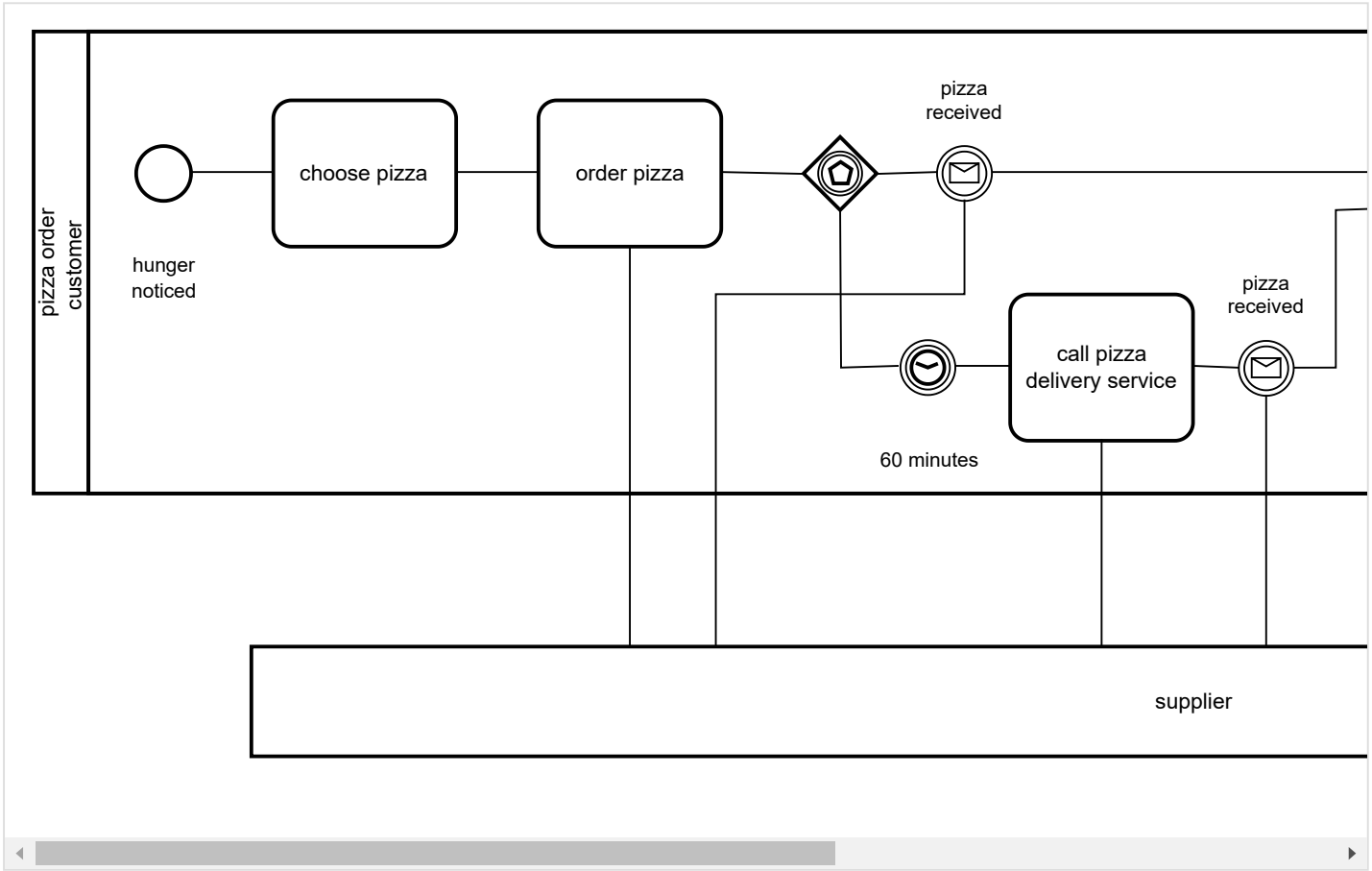
Create your own model or check out the “Order and Deliver Pizza” example diagram with Camunda’s free **online modeler**.

Collapsing Pools

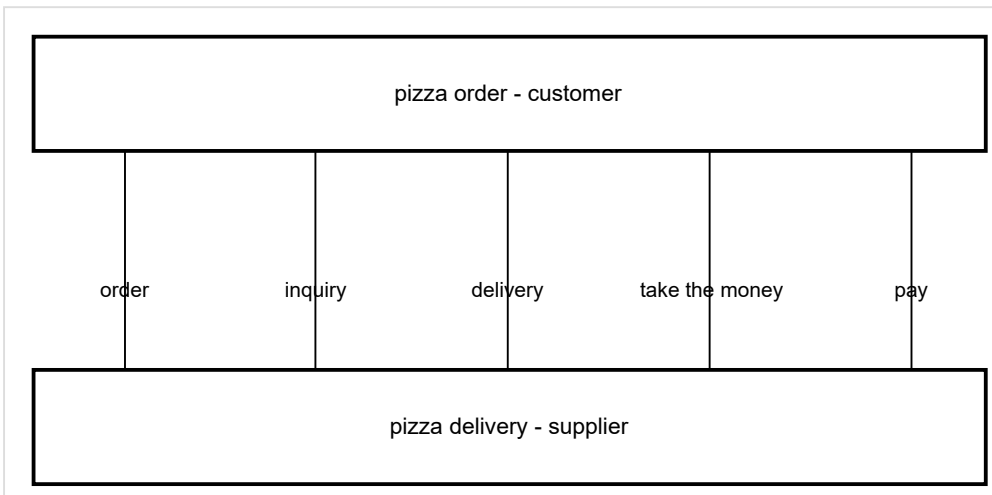
It often happens that we don’t know the processes of all parties in detail. We may know the processes of our own company, for example, but not those of a partner company. As long as our partner and we adhere to agreed-upon interfaces, such as receiving or sending certain messages, things can still operate smoothly. As customers of the pizza delivery service, we expect the deliverer to:

- Accept pizza orders,
- Deliver ordered pizzas and collect the money, and
- Be available for inquiries.

As customers, we have little interest in the deliverer’s internal process. Maybe he bakes and then delivers the pizza; maybe when he’s out of supplies, he gets another pizza service to bake the pizza and deliver it. That’s his problem – we simply expect to receive our pizza. In modeling such cases, we can hide the deliverer’s process and collapse the pool:



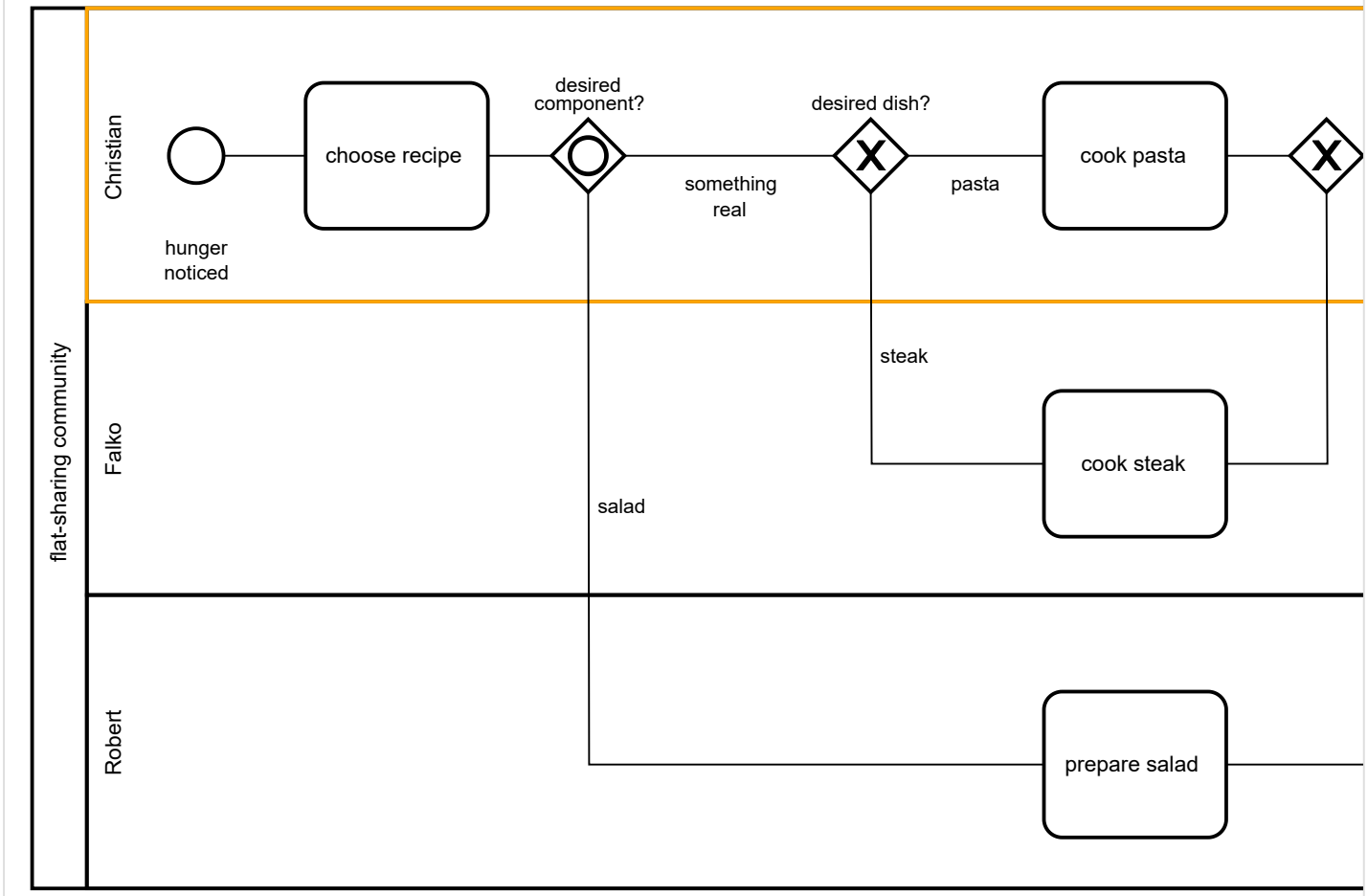
We could go a step further and collapse the customer’s pool too. Now we see only the messages to be exchanged, assuming that we label the arrows to give us the general idea. The downside is that we can’t recognize interdependencies any more. We can’t see if the inquiry always goes out, or only takes place under certain conditions – the actual case:



Lanes

We have talked about **what** to do in our processes, but we have not yet explained **who** is responsible for executing which tasks. In BPMN, you can answer this question with lanes.

Hover over orange symbols for explanation



The diagram shows that the tasks in our sample process were assigned to particular people. We can derive the following process description from the assignments: If Christian is hungry, he chooses a certain recipe. Depending on what Christian chooses, he can either take care of it himself (cook pasta), or he can get his roommates on board. If the latter, Falko cooks steak and Robert prepares salad. In the end, Christian eats. The three lanes (Christian, Falko, Robert) are united in one pool designated “flat-sharing community.”

FAQ: Do I have to label a lane with a concrete person?

In the example, lanes equate to people, but this meaning is not specified by BPMN. You can designate the lanes as you like. In practice, lanes are often used to assign:

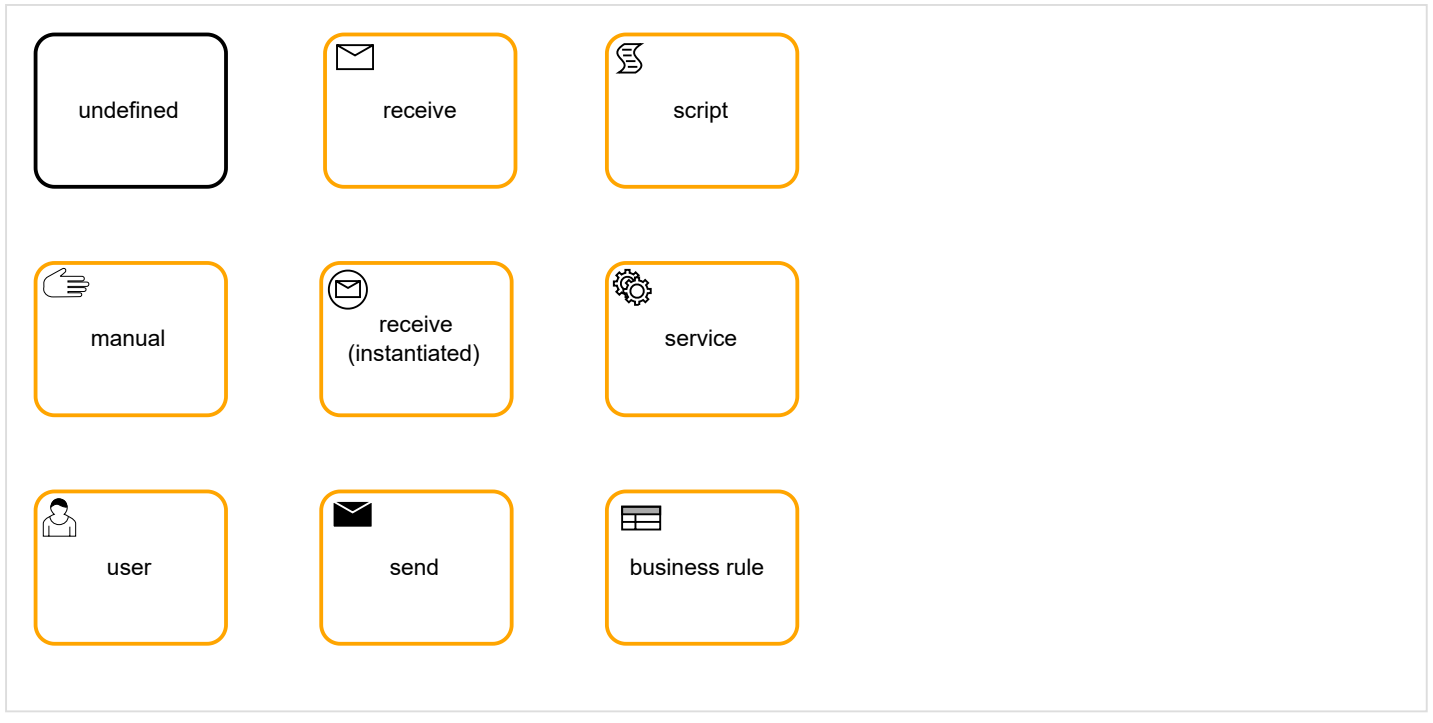
- Positions in the primary organization, for example, accounting clerk.
- Roles in the secondary organization, for example, data protection officer.
- General roles, for example, customer.
- Departments, for example, sales.
- IT applications, for example, CRM system.

Activities

Task

So far, we have used only tasks of undefined types, though BPMN provides the opportunity to work with task types just as it does for event types. Primarily, task types are intended to model processes that are technically executable. Task types are applied infrequently in practice. We know from experience, however, that task types can be particularly useful when modeling engineering requirements.

Hover over orange symbols for explanation

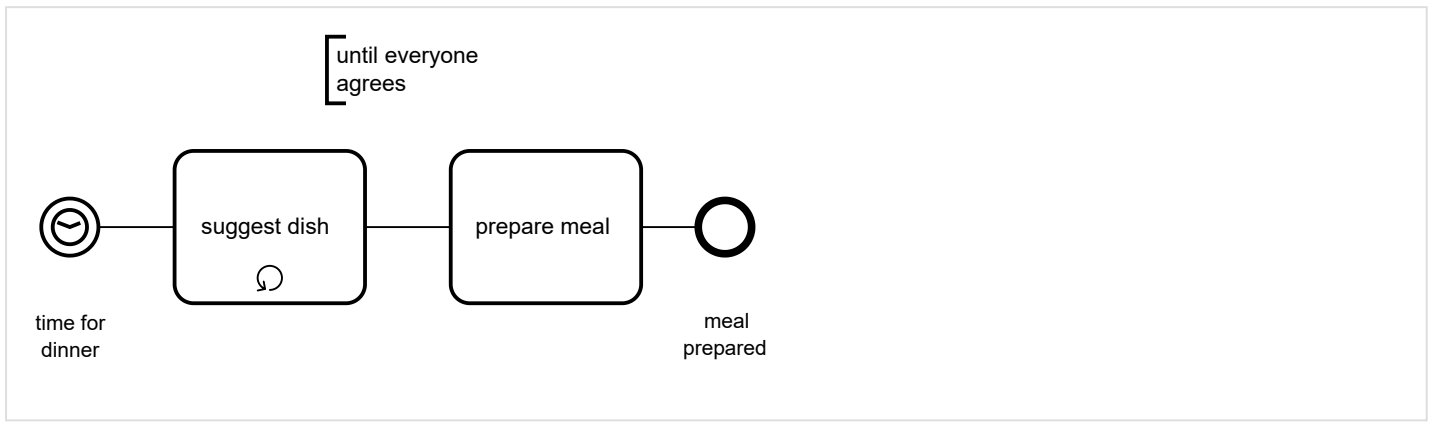


Task Markers

In addition to those various types of tasks, we can mark tasks as loops, multiple instances, or compensations. Markers can be combined with the assigned types.

Loop

A loop task repeats until a defined condition either applies or ceases to apply. Perhaps we suggest various dishes to our dinner guests until everyone agrees. Then, we can prepare the meal:

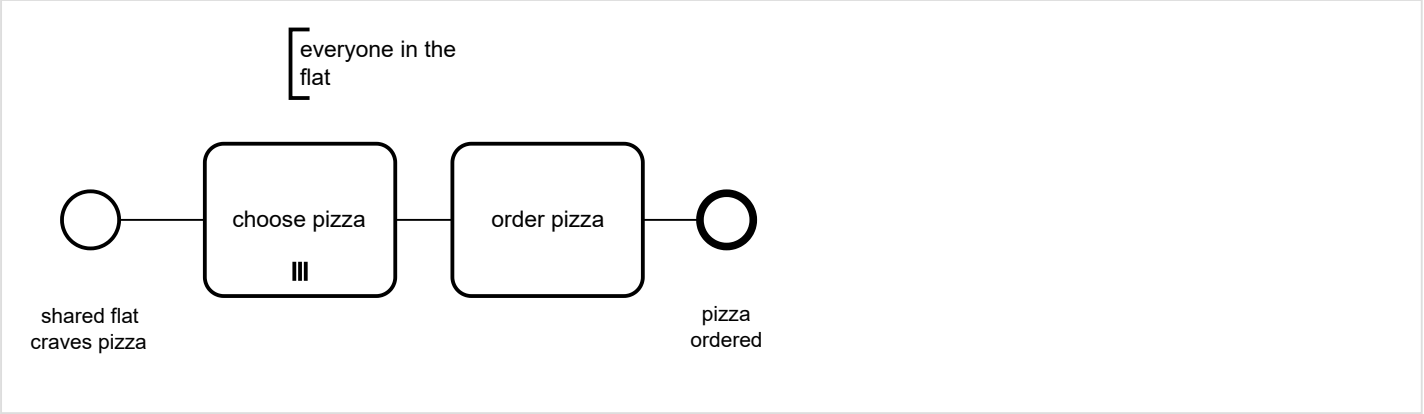


In the example, we executed the task first and checked afterwards to see if we needed it to execute again. Programmers know the principle as the “do-while” construct. We can also apply a “while-do” construct, however, and so check for a condition before the task instead of afterward. This occurs rarely, but it makes sense if the task may not execute at all.

You can attach the condition on which a loop task executes for the first time or, as shown in the example, apply the condition on repeated executions as an annotation to the task. You can store this condition as an attribute in a formal language of your BPMN tool as well. That makes sense if the process is to be executed by a process engine.

Multiple Instance

The individual cycles of a loop task must follow each other. If for example we live in a flat-sharing community and the roommates feel like eating pizza, the “choose pizza” task must be repeated for each roommate before we can order. You’d sit together and pass a menu around until finally everyone has made a decision. There are student apartments where they **do** handle it like that – more evidence that students have too much time on their hands! It is much more efficient for all roommates to look at the menu at once, and they choose a pizza together. You can model this process using the “multiple task” (see below). A multiple task instantiates repeatedly and can be executed in sequence or in parallel, with the latter being the more interesting case.

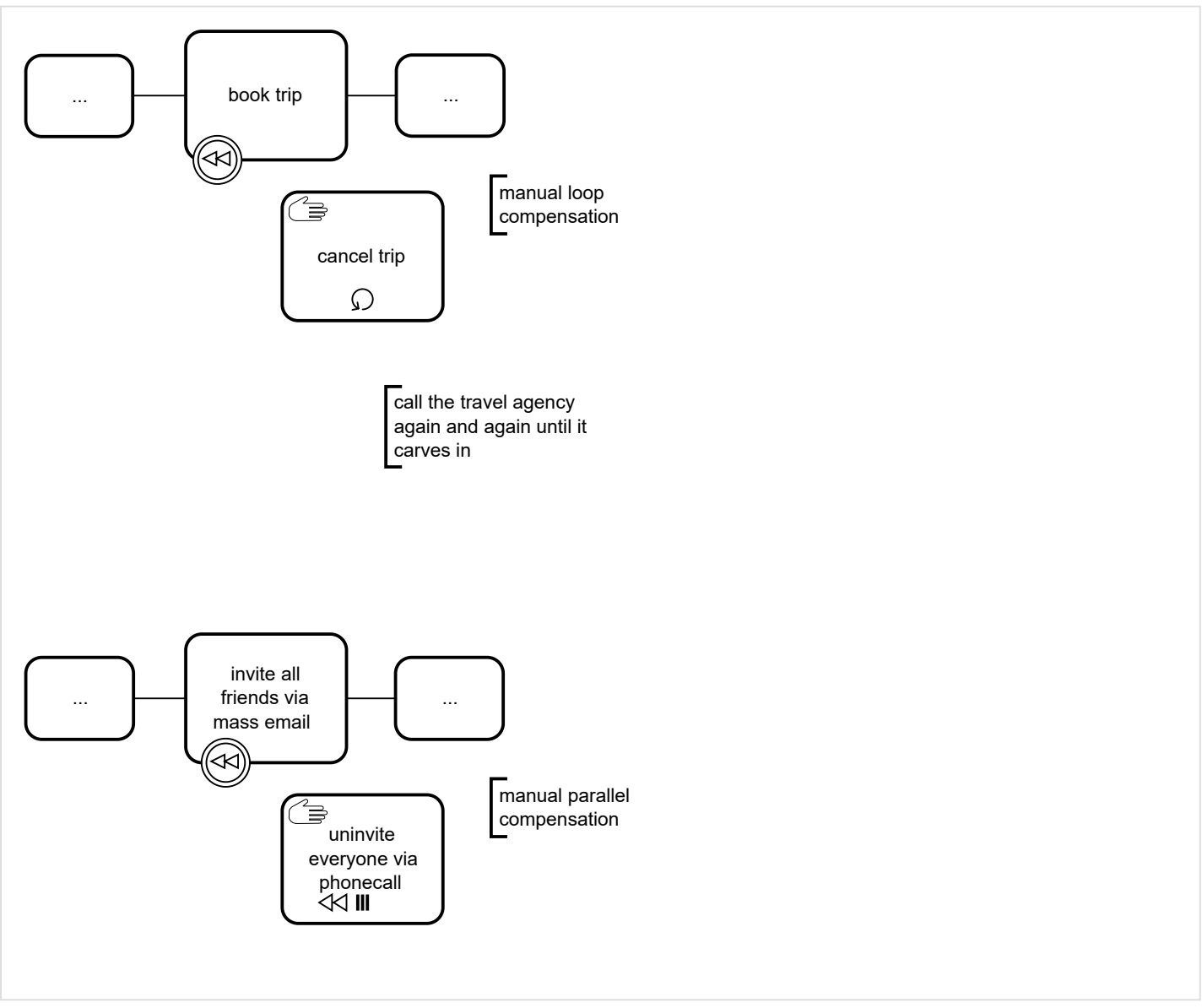


Do you think the example is absurd? How does your company check invoices for group orders, such as for office supplies? Do you forward the invoice from one employee to the next, so that each person can sign off on the items he or she ordered, before you pay the invoice? If so, you live in a flat-sharing community, and you urgently should consider optimizing your process. Automating invoices is still one of the top BPM projects, and the top goal of such projects often is one of parallelization.

Compensation

We explain the benefit of the compensation event by means of an example. The compensation task type is applied exclusively in the context of a compensation event. Accordingly, it is integrated in the process diagram only by associations, never by sequence flows.

The possible combination of the compensation with a loop or multiple instance as shown below is worth mentioning. In this case, both markers are placed in parallel. As with the other markers, the compensation can be combined with the task types already introduced. A manual compensation task that repeats until it succeeds or that executes repeatedly and in parallel as far as possible, is therefore eminently practical.



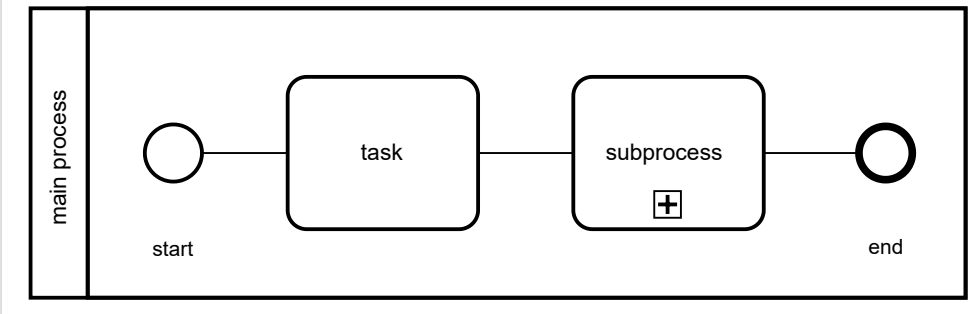
Subprocess

Encapsulate complexity

The examples in this tutorial either deal with simple processes, or they diagram complex processes superficially so that the models fit on one page. When modeling your process landscape, you don’t have this luxury. You have to rough out your processes so that you can get the general ideas in place and recognize correlations. Then you have to develop a detailed description, so that you can analyze exactly

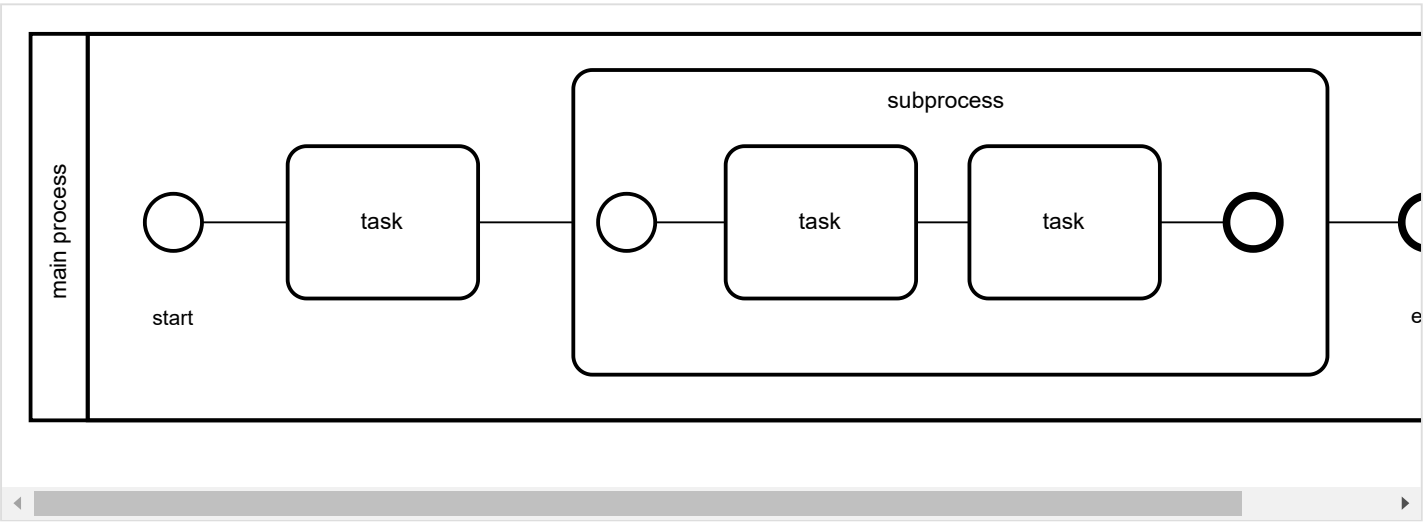
where the weak points are or how you'll have to execute the process in practice. The possible top-down refinements or bottom-up aggregations mark the difference between true process models and banal flow charts, between sophisticated BPM software products and mere drawing programs.

BPMN provides us with the subprocess to help with the expanding/collapsing view. A subprocess describes a detailed sequence, but it takes no more space in the diagram of the parent process than does a task. Both tasks and subprocesses are part of the activities class and are therefore represented as rectangles with rounded corners. The only difference is the plus sign, indicating a stored detailed sequence for the subprocess:



What good is that to us? That depends most on how your BPMN tool supports the following options for connecting subprocesses with their parent processes:

- **Representation in a separate process diagram**: The subprocess symbol links to a separate diagram. If your BPMN tool displays the process model in a web browser, for instance, clicking on the symbol would open a new page to display the detail diagram.
- **Expanding in the process diagram of the parent process**: The activity with the plus sign is called a collapsed subprocess. The plus sign suggests that you could click on it and make the subprocess expand. The BPMN specification provides for this option, though not all tool suppliers implement it. The diagram below shows how the subprocess was directly expanded in the diagram of the parent process. A tool supporting this function enables you to expand and collapse the subprocess directly in the diagram, respectively, to show or hide details.



Direct expansion may seem appealing, but often it is not useful in practice. Expanding the subprocess requires that all the adjacent symbols in the diagram shift to make room. This can result in sluggish performance with a complex diagram, and it can be visually nasty. The most important thing is that your tool provides for linking and that you can usefully navigate through the diagrams. In other words, it supports the first option above. Yes, it can be helpful to have your subprocess modeled and expandable directly from the parent process. That means process segments remain localized, and you can attach events too. This is, however, the less important option.

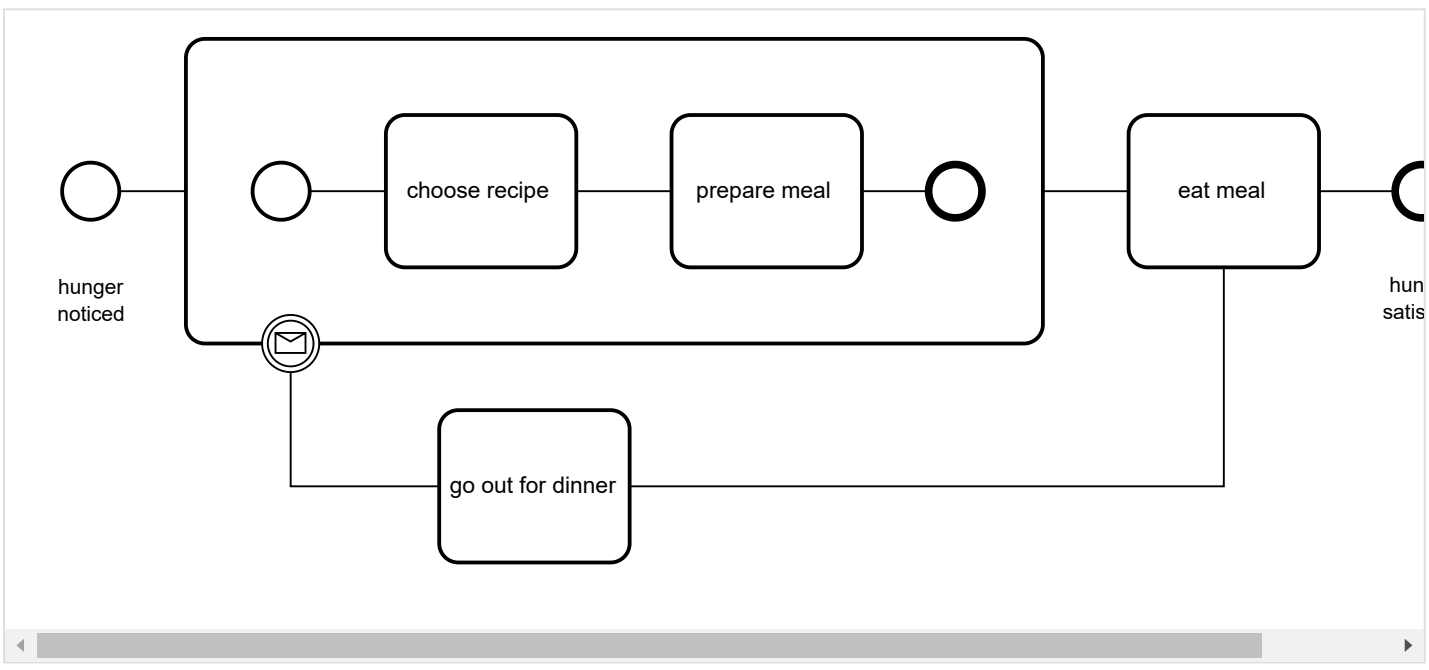
The sequence flow of the parent process ends in both cases at the left edge of the subprocess. The next sequence flow starts at the right edge. This means that sequence flows are not allowed to exceed the boundaries of the subprocess, which not every beginner knows, and which becomes a problem when a subprocess expands. Visualize a token that behaves as follows:

- The parent process starts, and a token is born.
- The token runs through the task and arrives at the subprocess, which causes the parent process to create an instance of the subprocess.
- Within the subprocess, a separate token is born which runs through the subprocess from the start to the end event, but the token of the parent process waits until the subprocess completes.
- When the subprocess token arrives at the end event, it is consumed, which completes the subprocess. Now the token of the parent process moves to its own end event.

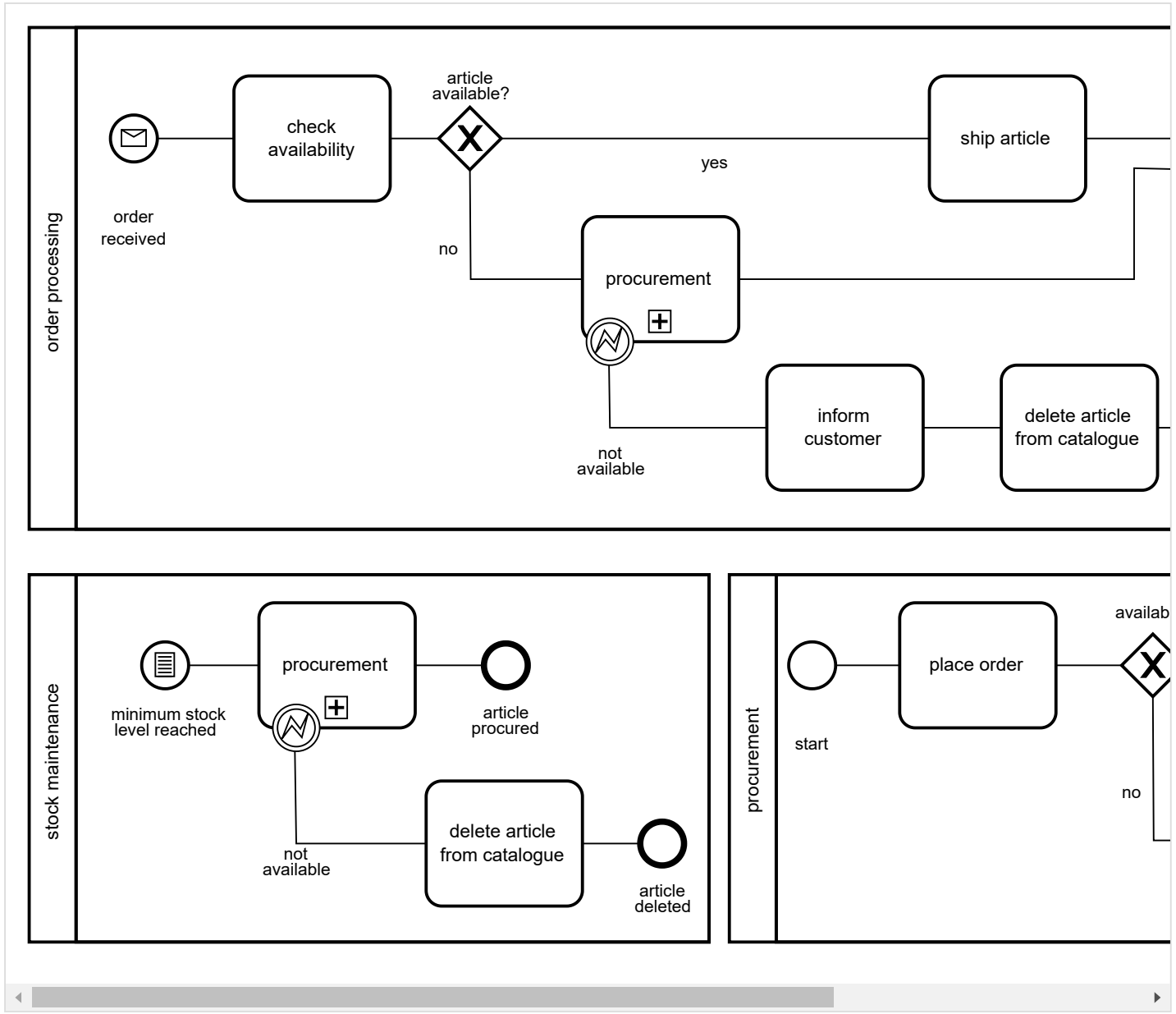
The encapsulation in subprocesses that we're describing isn't restricted to two levels. You could just as easily have a parent process as a subprocess, or you could model further subprocesses on the level of a defined subprocess. How many levels you use and the level of detail you apply to model them is up to you. BPMN doesn't specify this, and there can be no cross-company or cross-scenario cookbook to define levels. Participants in our BPMN workshops don't like this, but there's no point in hiding the fact nor attempting to explain it away. In the following chapters, we work often with subprocesses in explaining our best practices, but the truth is the number of refinement levels and their respective levels of detail is always situational. It depends on the organization, the roles of the project participants, and the goals for the process you're modeling.

Attaching Events

We already learned about intermediate events that can be attached to tasks. The same events can be attached to subprocesses as well, which opens up a wide range of opportunity in process modeling. As shown in the diagram below, we can represent how a spontaneous dinner invitation leads to canceling our cooking process. In the process shown, however, we could ignore the invitation if our meal had already been prepared and we are already eating it:



Where message, timer, and conditional events are involved, the parent process always aborts the subprocess when reacting to external circumstances. With error, cancellation, and escalation events, however, the subprocess reports these events to the parent process. This isn't as abstract as it may sound.

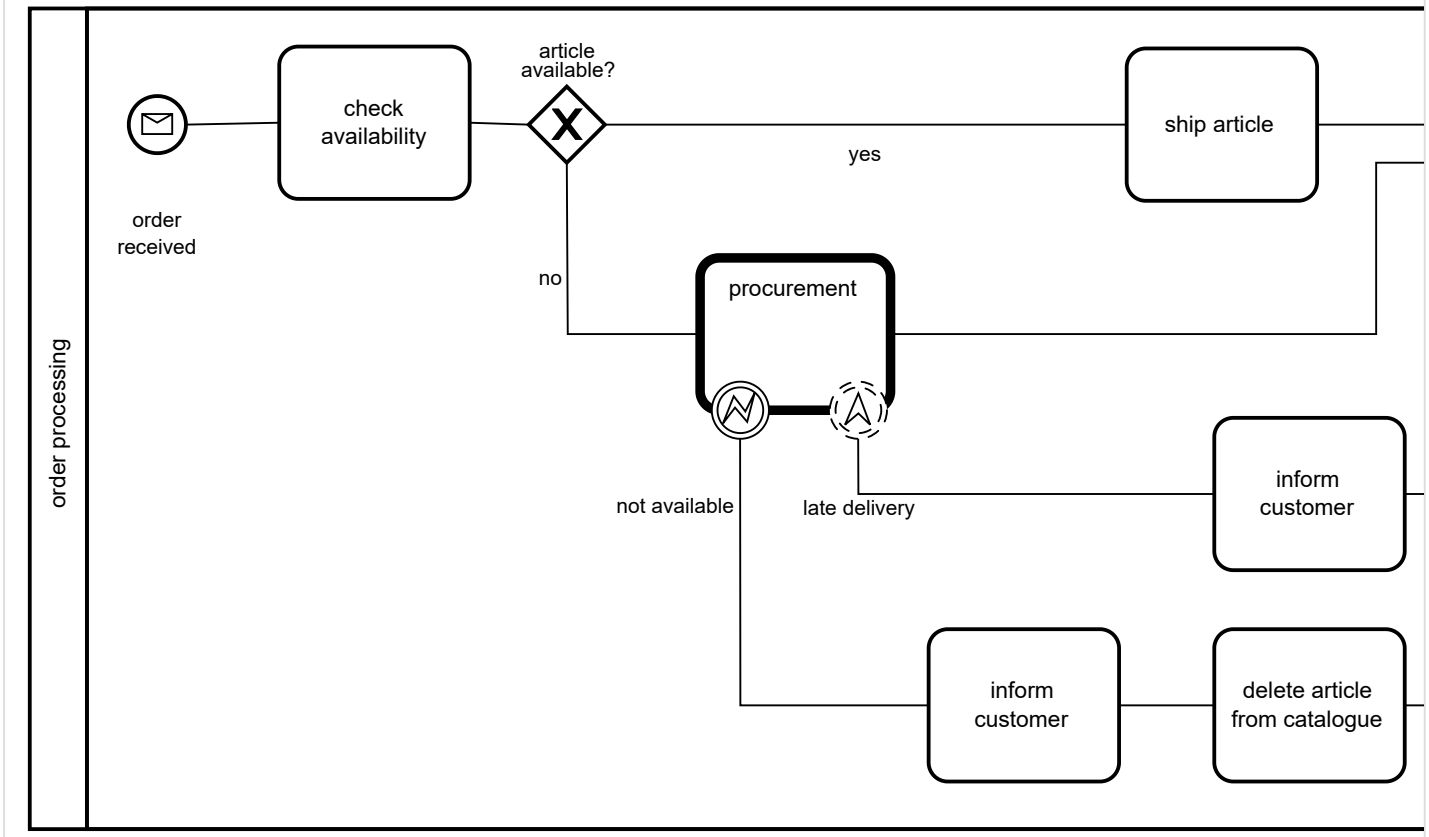


In the bottom right of the diagram above, the item procurement task can fail because the item is no longer available. Because item procurement is a global subprocess, it triggers an error event to tell the parent process that something went wrong. In business terms, this may mean that the customer who wanted to buy the item tells a salesperson that his or her order failed because the item is out of stock.

It is interesting that parent processes can handle the error message differently. While the disappointed customer must be informed within the scope of the order process, it is sufficient for the stock maintenance process to delete the item from the catalog. The respective parent processes decide what circumstances require canceling the subprocess and what happens next. That’s a principle that you can use to build flexible and modular process landscapes.

The signal event serves two functions. A parent process can react to a signal received from the outside while it executes a subprocess – this is much like a message event. But we also use the signal event to let the subprocess communicate things other than errors to the parent process. Primarily, this is because we can’t model this type of communication with message events. BPMN assumes that we always send messages to other participants who are outside of our pool boundaries; the communication between parent and subprocess doesn’t fit that mold. We don’t use signal events for directed communication, but rather to broadcast information akin to advertisements on the radio.

A better alternative provided in BPMN 2.0 is the escalation event. The subprocess can use an escalation event to report directly to the parent process, and the message won’t be regarded as an error message. Also, the parent process can receive and process messages from escalation events without canceling the subprocess because non-interrupting intermediate events can be attached:



Call Activity

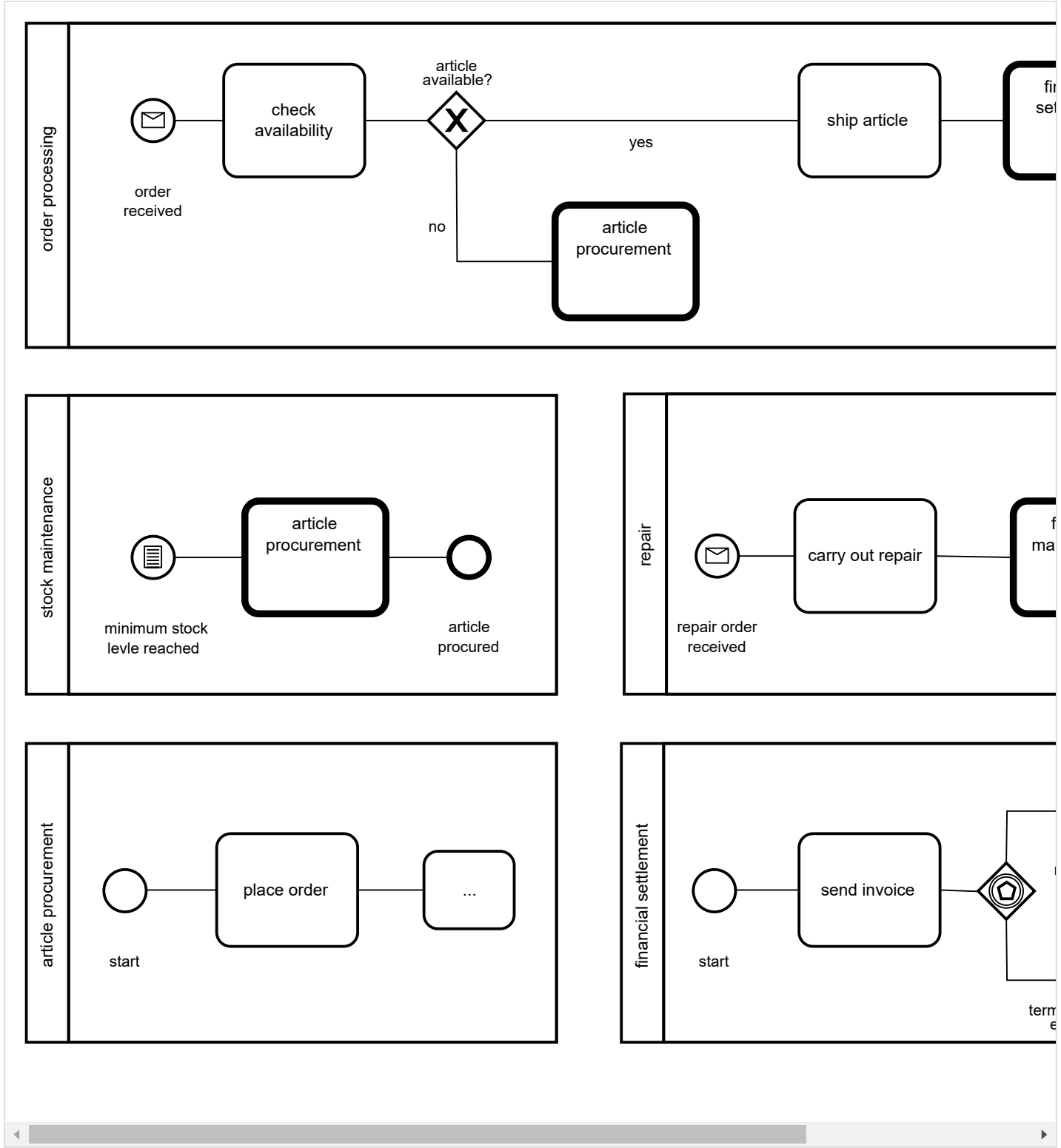
Modularization and reuse

In version 1.2, BPMN differentiated between embedded and reusable subprocesses by assigning an attribute to a subprocess. In version 2.0, BPMN maintains this differentiation in principle, but it is defined differently. A subprocess now is embedded intrinsically, and it can be reused only by defining it as a global subprocess, and then referencing it by means of a call activity. We therefore refer to embedded subprocesses and global subprocesses in the following.

An embedded subprocess can occur only within a parent process to which it belongs. An embedded subprocess cannot contain pools and lanes, but it can be placed within the pool or the lane of the parent process. Furthermore, an embedded subprocess may have only a none start event; start events such as messages or timers are not permitted. An embedded subprocess has essentially nothing more than a kind of delimited scope within the parent process, which may serve two goals:

- To encapsulate complexity (as already described)
- To formulate a “collective statement” on a part of the parent process by attaching events or placing markers. We deal with this option later.

On the other hand, global subprocesses may occur in completely different parent processes. There are a great many subprocesses that, in practice, are used over and over. A good example is the procurement of an item because a customer ordered it or you need to re-stock supply. Another example is invoicing because you’ve delivered or repaired an item as shown in the diagram below. In the example, notice that call activities differ from regular activities by their considerably thicker borders:



The connection a global subprocesses has to its parent is considerably less close, and they can have their own pools and lanes. You can think of the participant responsible for a subprocess as a service provider for various parent processes. It is a like a shared service center.

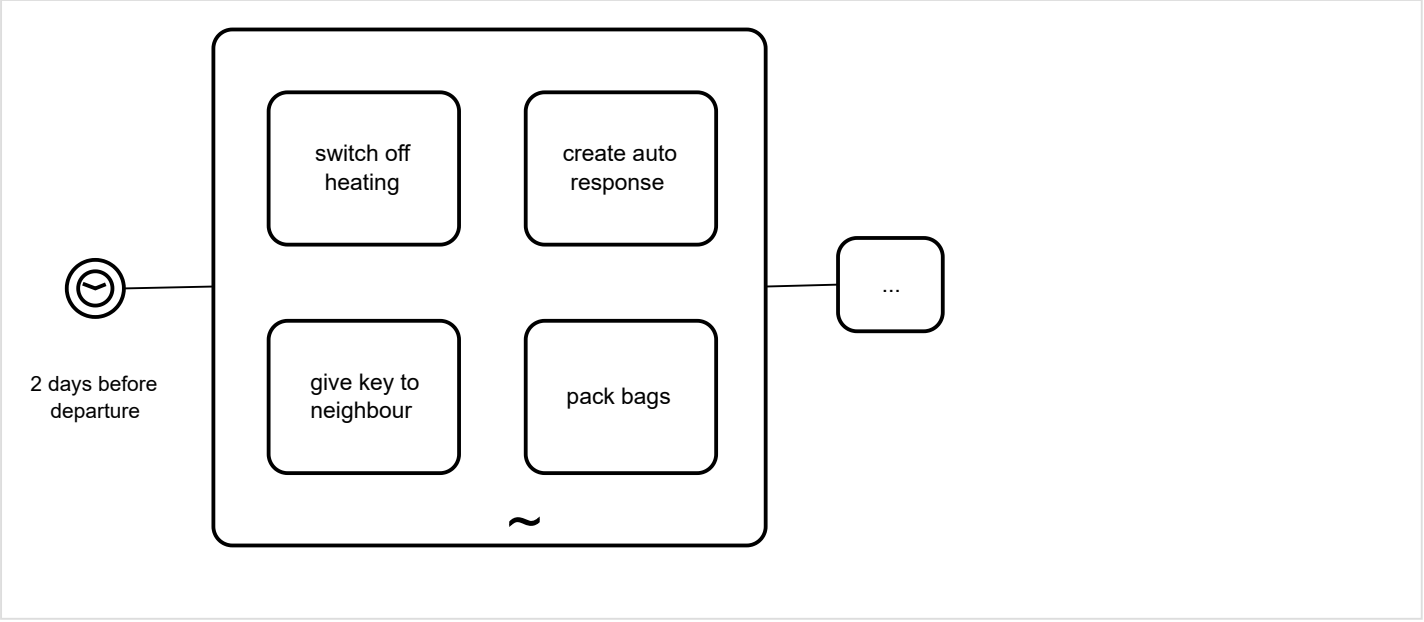
The loose connection also affects data transfer between the parent and the subprocess. BPMN assumes that embedded subprocesses can read all the data of the parent process directly, but an explicit assignment is required for global subprocesses to be able to read it. That may seem like merely a technical aspect at first, one that modelers and the consumers of their models care to know about but won't wish to bother with. After some consideration, however, you may see the impact this difference makes on the organization. Consider this: When your accounting department wants to issue an invoice for a repair, it always needs:

- A billing address
- The date of performance delivery
- A description of performance
- An amount to invoice
- An expected date of payment

The owners of order processing, not just the repair department, must provide this data. Accounting will want the data in a standard format, won't it? This corresponds well to what BPMN calls required data mapping between parent processes and global subprocesses. (Do you notice how often these weird techie issues correspond to the organizational needs and expectations of a process?) BPMN simply forces us to formalize many matters that seem self-evident, or that remained unconscious or forgotten in the process design. Formalization is our best chance of keeping up in a fast-changing environment with ever more complex processes.

Adhoc

One marker available only for subprocesses is called ad-hoc. Recognize it by the tilde character as shown in the diagram below:



Use the ad-hoc subprocess to mark a segment in which the contained activities (tasks or subprocesses) can be:

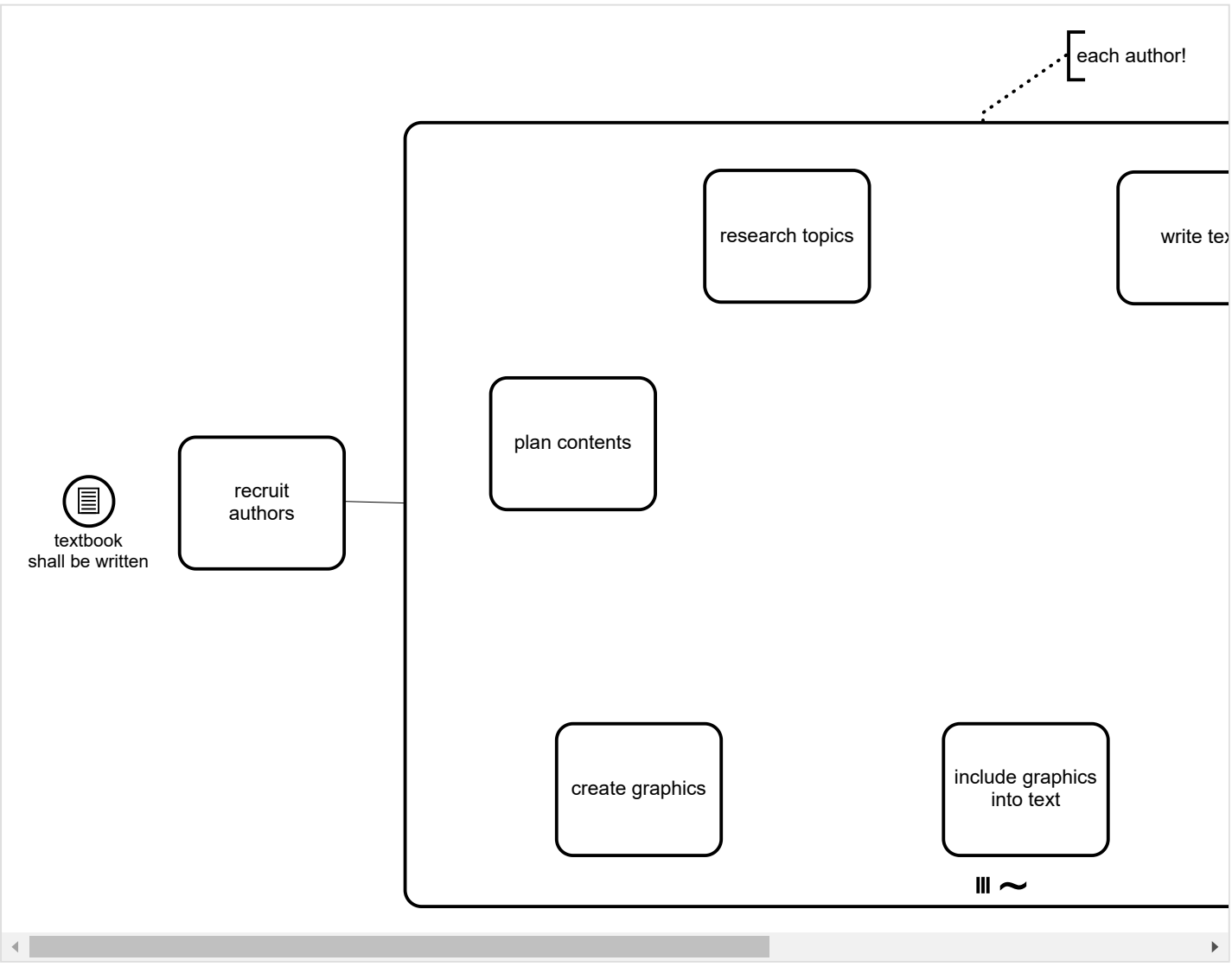
- Executed in any order,
- Executed several times, or
- Skipped.

Any party who executes this subprocess decides what to do and when to do it. You could say that the “barely structured” nature of what happens inside this subprocess reduces the whole idea of process modeling to an absurdity because what happens and when are the things we most want to control. On the other hand, this is the reality of many processes, and you can’t model them without representing their free-form character. Frequent examples are when a process relies largely on implicit knowledge or creativity, or when different employees carry out a process differently. You can use the ad-hoc subprocess to flag what may be an undesirable actual state. Doing so could be a step on the path to a more standardized procedure.

BPMN 2.0 specifies which symbols must, which may, and which are forbidden to occur within an ad-hoc subprocess. They are:

- **Must**: Activities
- **May**: Data objects, sequence flows, associations, groups, message flows, gateways, and intermediate events
- **Forbidden**: Start and end events, symbols for conversations and choreographies

By means of the specification, mixed forms – so-called weakly structured processes – can be modeled as shown in here:

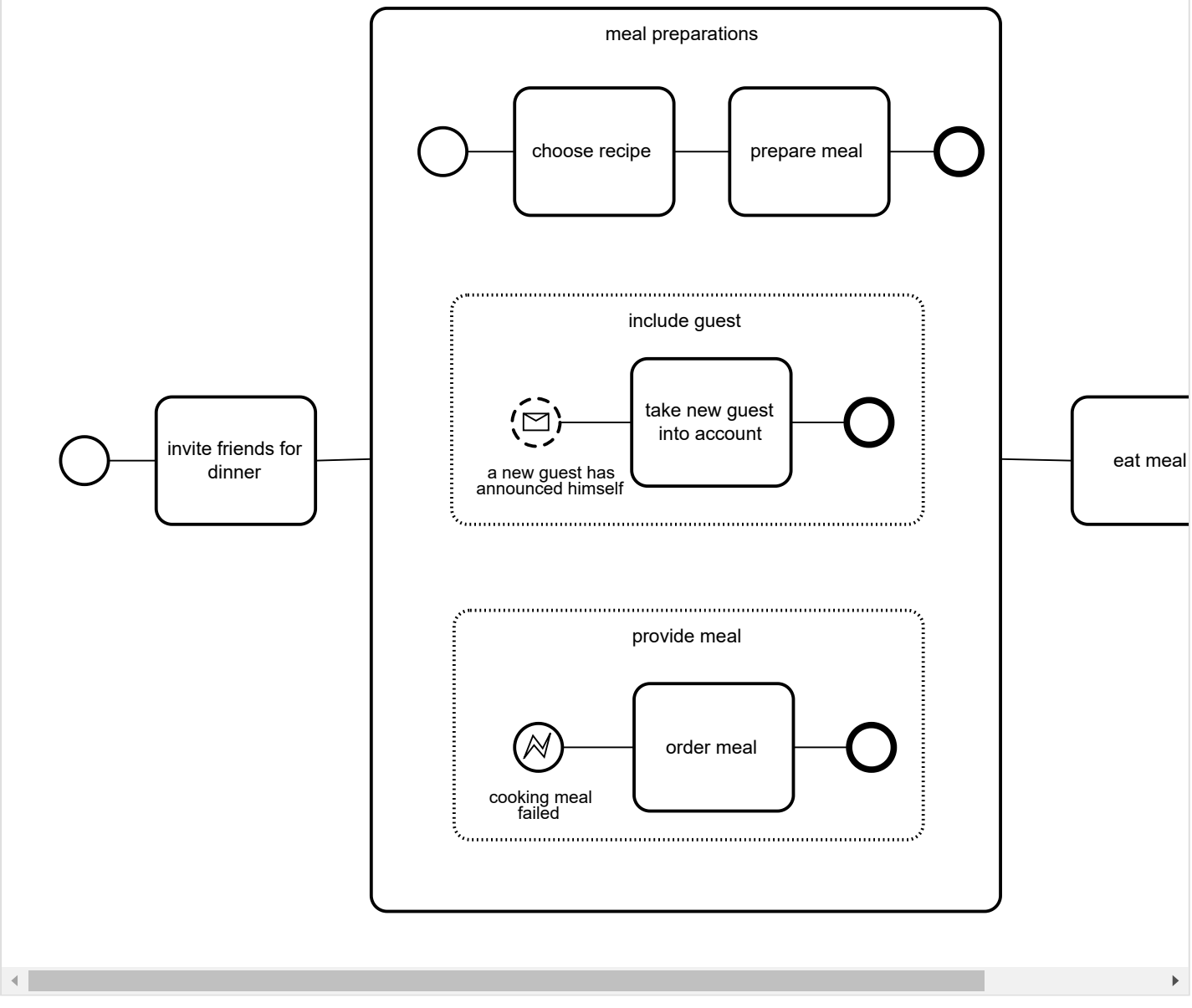


Event Subprocess

BPMN 2.0 introduced a completely new construct, the event subprocess. We locate an event subprocess within another process or subprocess. Recognize them by their dotted-line frames.

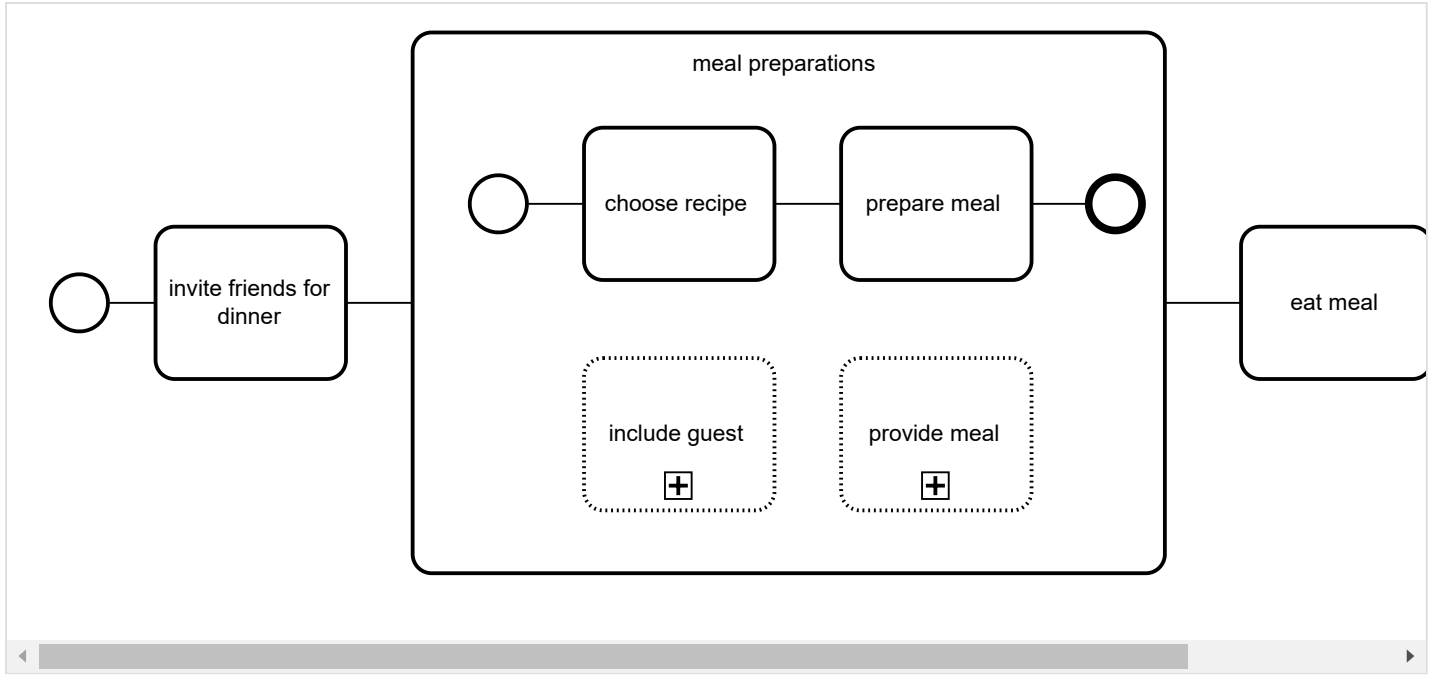
A single start event always triggers an event subprocess, and this can only happen while the enclosing process or subprocess remains active. For event subprocesses, there can be interrupting (continuous line) and non-interrupting (dashed line) events. This is the same differentiation made as for attached intermediate events. Depending on the type of start event, the event subprocess will cancel the enclosing subprocess, or it will execute simultaneously. You can trigger non-interrupting event subprocesses as often as you wish, as long as the enclosing subprocess remains active.

Okay, that’s pretty abstract, but we can demonstrate how an event subprocess works with an example:



We invited a couple of friends for dinner. This starts the “dinner preparation” subprocess of choosing a recipe and then preparing the meal. While we are doing that, the telephone rings. Another guest invites himself to dinner. Spontaneous as we are, we just increase the amount of food or set another place at the table without interrupting the meal preparation. If an accident happens during preparation, however, the error immediately triggers the interrupting event subprocess for remedial action. We order food for delivery. When this event subprocess completes, we exit the enclosing subprocess through the regular exit and attend to eating the meal.

You can see below how event subprocesses are represented in collapsed state: The frame is a dotted line, and we have again used the plus sign to represent collapsed subprocesses. In the top left corner, we also have the start event triggering the subprocess.



The event types that can trigger both interrupting and non-interrupting event subprocesses are:

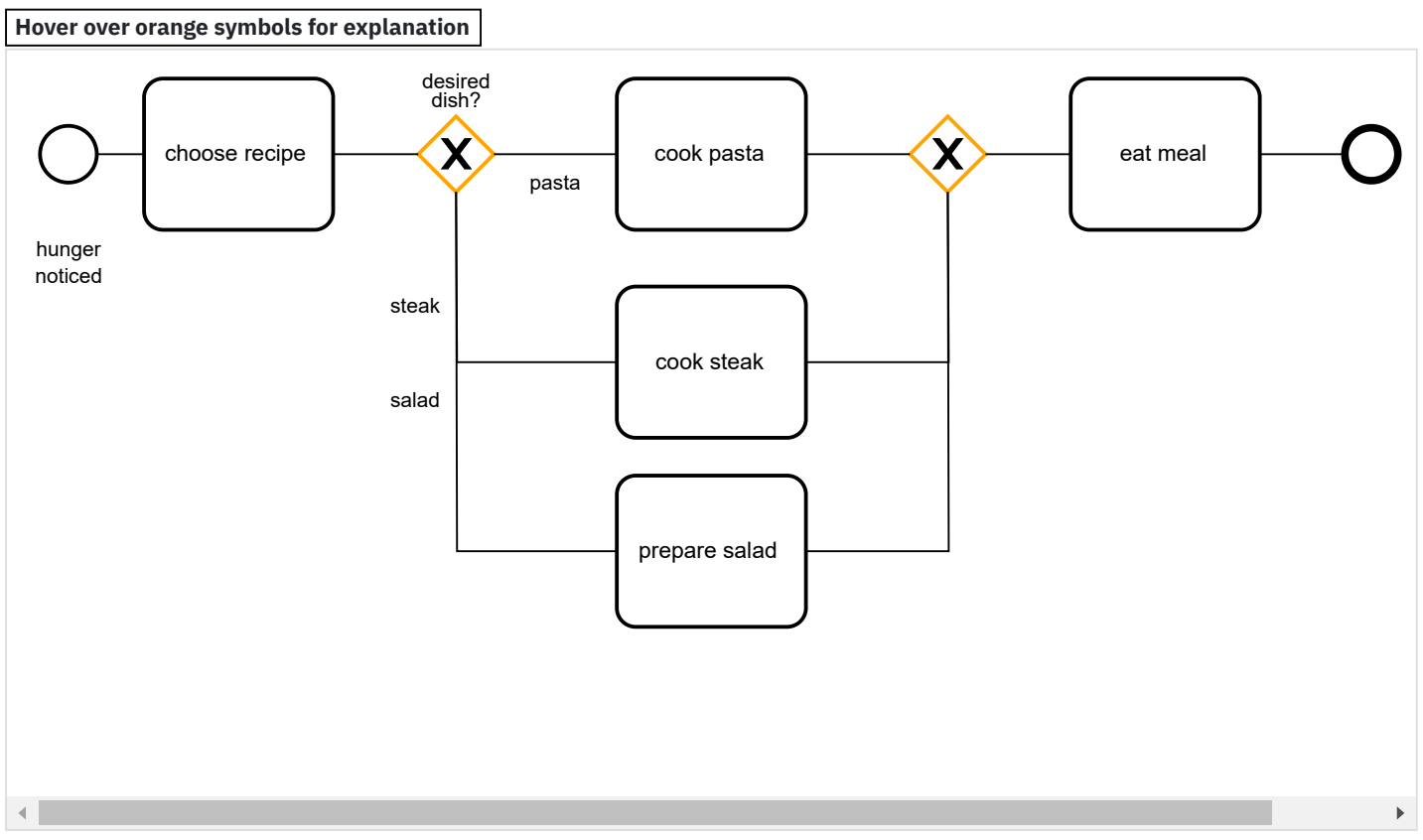
- **Message**
- **Timer**
- **Escalation**
- **Conditional**
- **Signal**
- **Multiple**
- **Multiple parallel**

There are two more types for the interrupting event subprocesses:

- **Error**
- **Compensation**

Gateways

Data-based Exclusive Gateways



Certain things can only be done under certain circumstances, so few processes always take the same course. In our simple example, we want to go into the details of cookery. Driven by hunger, we think about what we are going to cook today. We only know three recipes, so we choose one. We can either cook pasta **or** cook a steak **or** prepare a salad. Let’s say that these options are exclusive – we will never prepare more than one at a time. The point of decision on what to do next is called gateway. We decide based on available data (the chosen recipe) and we follow only one of the paths, which is a data-based exclusive gateway. We abbreviate “exclusive gateway” as **XOR**.

> **Heads up!**
> Bear in mind that a gateway is not a task! You have to determine facts and needs before reaching a gateway.

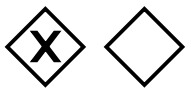
Best Practice: Naming Conventions

As in the diagram above, we place the crucial question before the gateway. This is our convention, which has proved its value in our projects. Possible answers go on parallel paths after the gateway, which is how the BPMN specification shows them. We always work with XOR gateways as follows:

- Model the task that requires a decision for the XOR gateway.
- Model the XOR gateway after that.
- Create a question with mutually exclusive answers.
- Model one outgoing path (or sequence flow) for each possible answer, and label the path with the answer.

FAQ: Do I have to draw the “X”-Marker inside the rhombus? I have already seen that symbol without any marker...

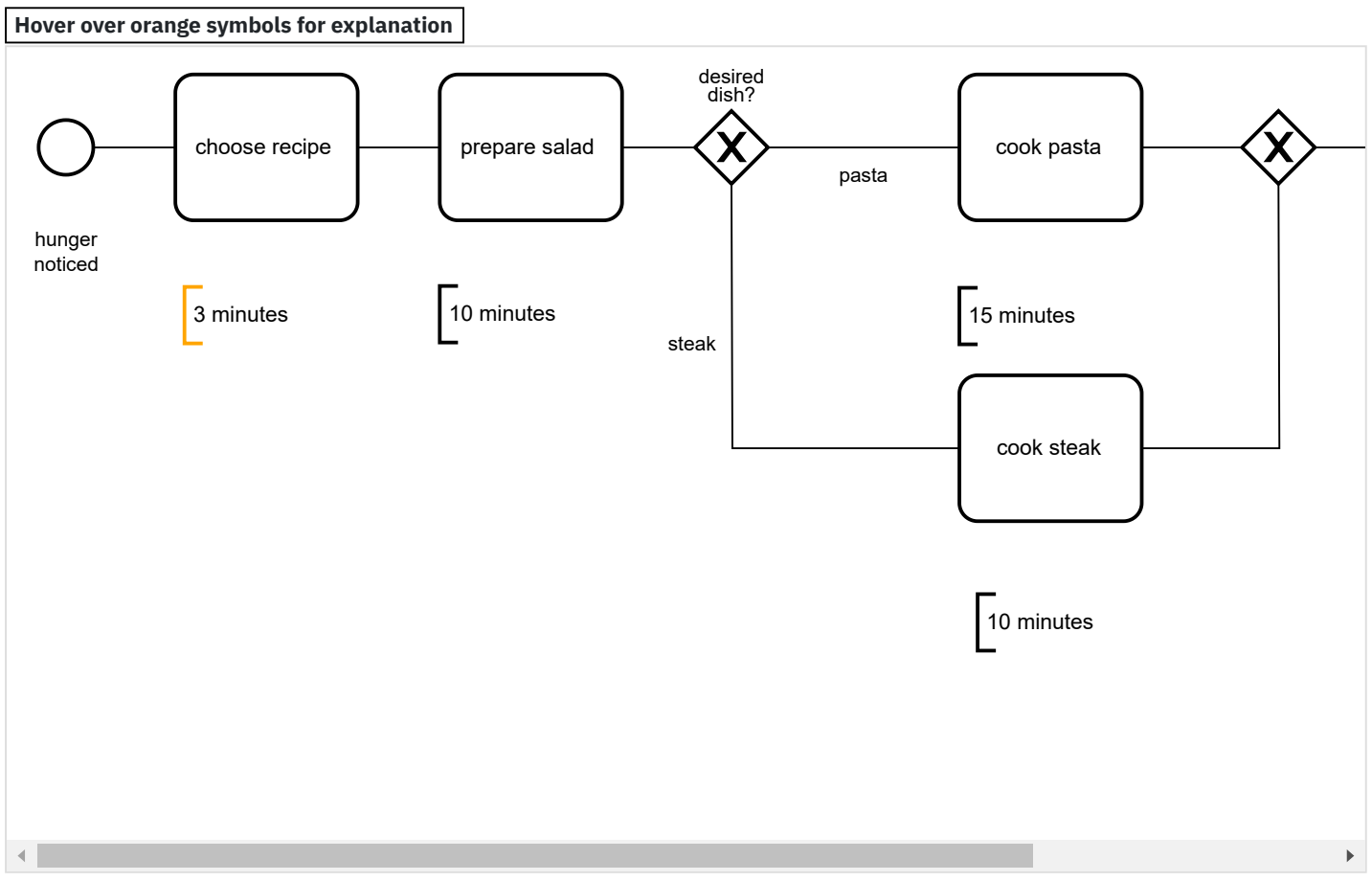
BPMN uses two symbols for XOR gateways:



They are identical in meaning. We always use the version with the X because it seems less ambiguous.

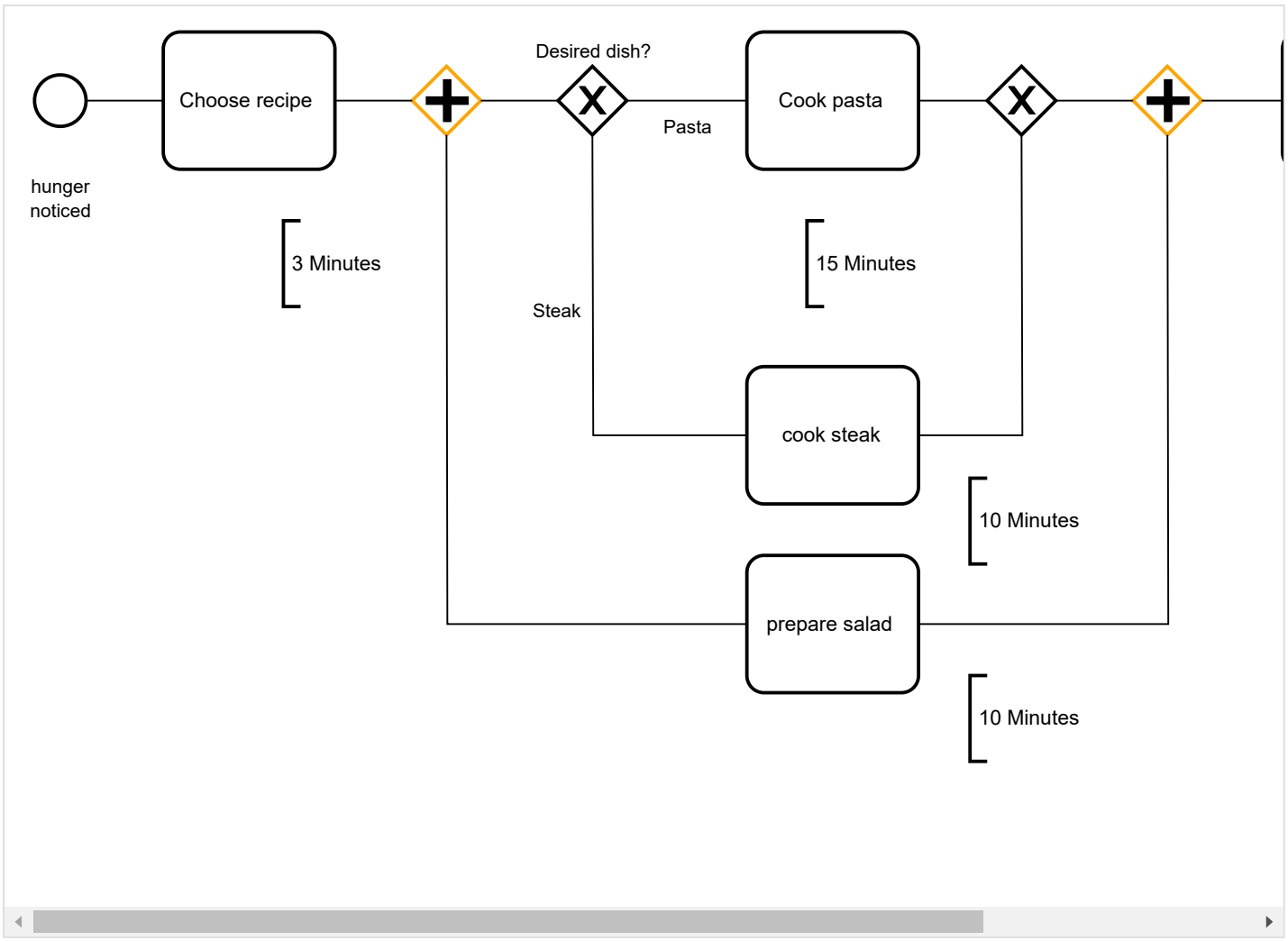
Parallel Gateways

Suppose that now we want a salad on the side. If you want salad no matter what, you could model it as we have done in this diagram:



The total of the task times equals the running time of the process, which was a total of 48 minutes for pasta and 43 minutes for steak. Congratulations: you’ve just analyzed your first process based on key data!

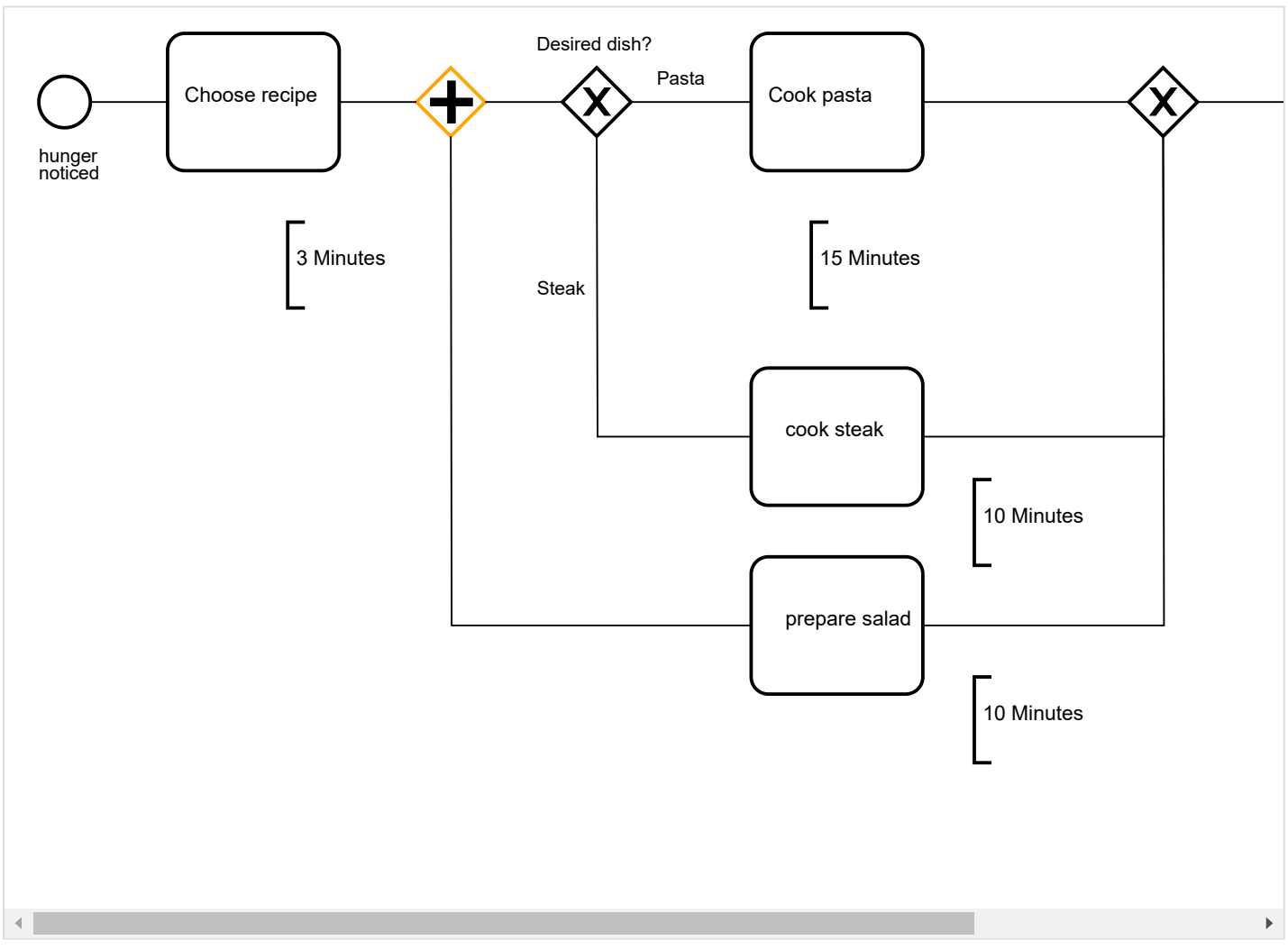
Still, this means waiting 23 or even 28 minutes until you can start to eat. Insufferable! You’re really hungry, but what can you do? Maybe you don’t prepare the salad first and then cook the pasta or the steak, but you work on both at the same time – in parallel. The appropriate symbol is the parallel gateway, or the “AND gateway” for short, as shown in here:



Diagramming tasks as parallel does not make simultaneous processing compulsory. In contrast to the example shown before, it is also not imperative that you prepare the salad before starting other tasks. Parallel preparation does, however, reduce our total time by 10 minutes. It is classic process optimization to make tasks parallel as much as possible.

Check yourself: What if we draw the same process, but leave the AND merge out for lack of space, and the path from the “prepare salad” task leads directly to the XOR merge. What happens if we instantiate the process, and we decide in favor of pasta?

The token is generated and then cloned as always at the AND split. As soon as we finish preparing the salad, the token passes through the XOR merge and “eat meal” executes. Five minutes later,”cook pasta” also completes. Its token passes through the XOR merge and “eat meal” executes again! That’s not the behavior we wanted.



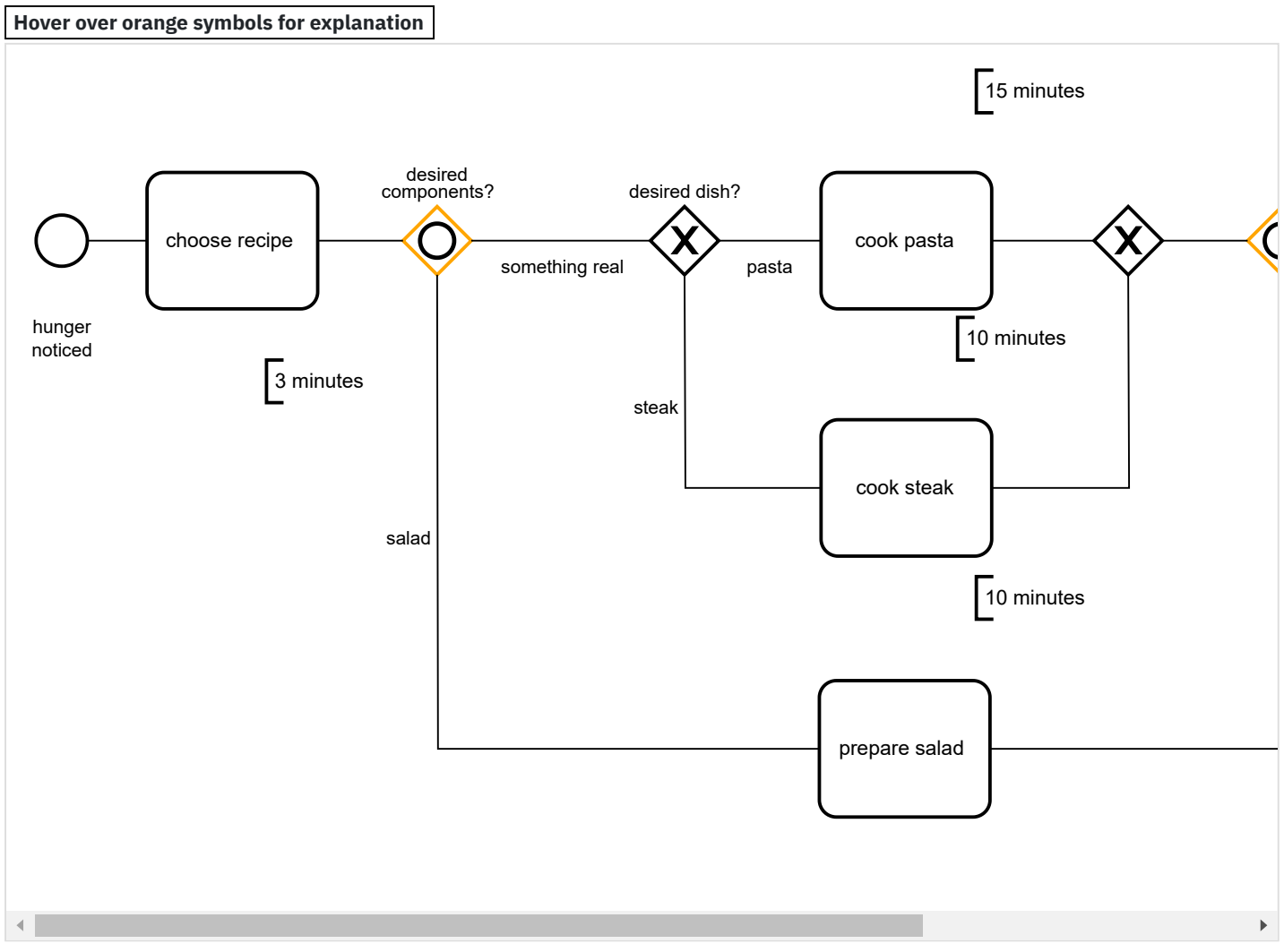
Data-based inclusive gateways

We want to make our process even more flexible: When we are hungry, we want to eat

- Only a salad,

- A salad and “something real,” like pasta or steak, or
- Only something real.

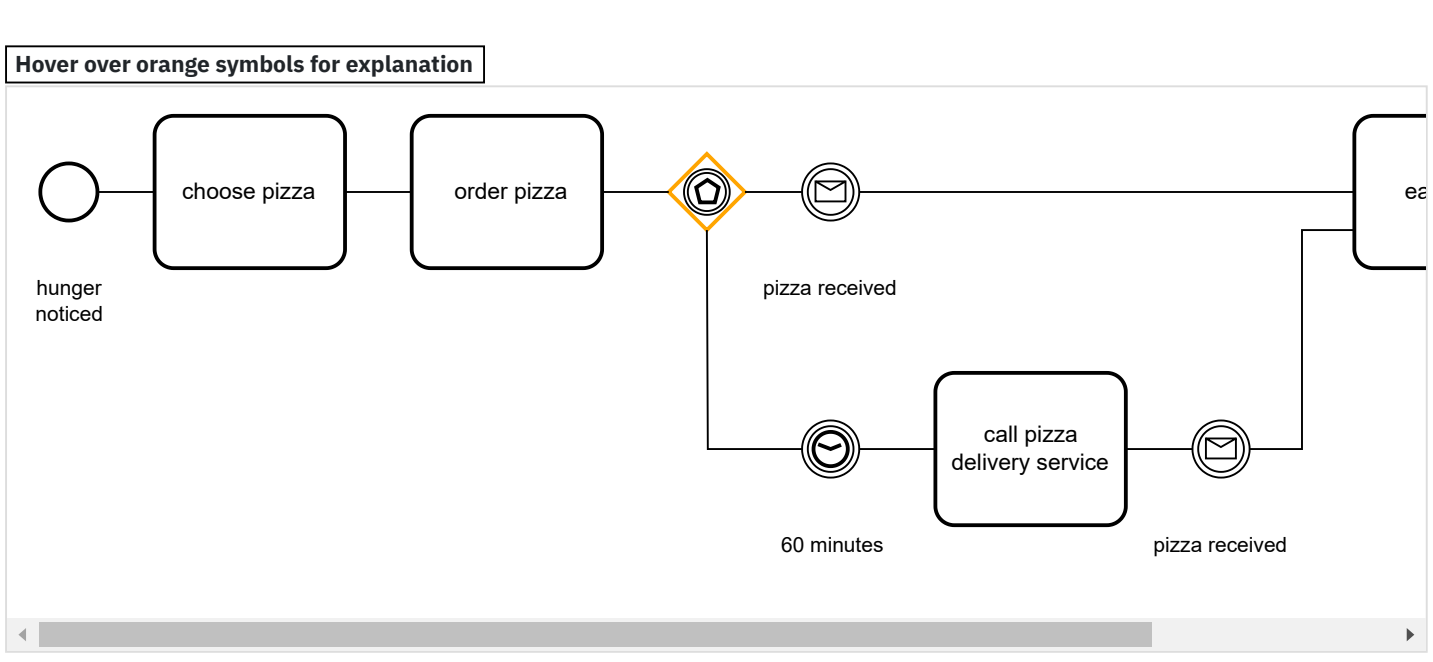
If you want a more compact representation, you can use the data-based inclusive gateway – the OR gateway for short:



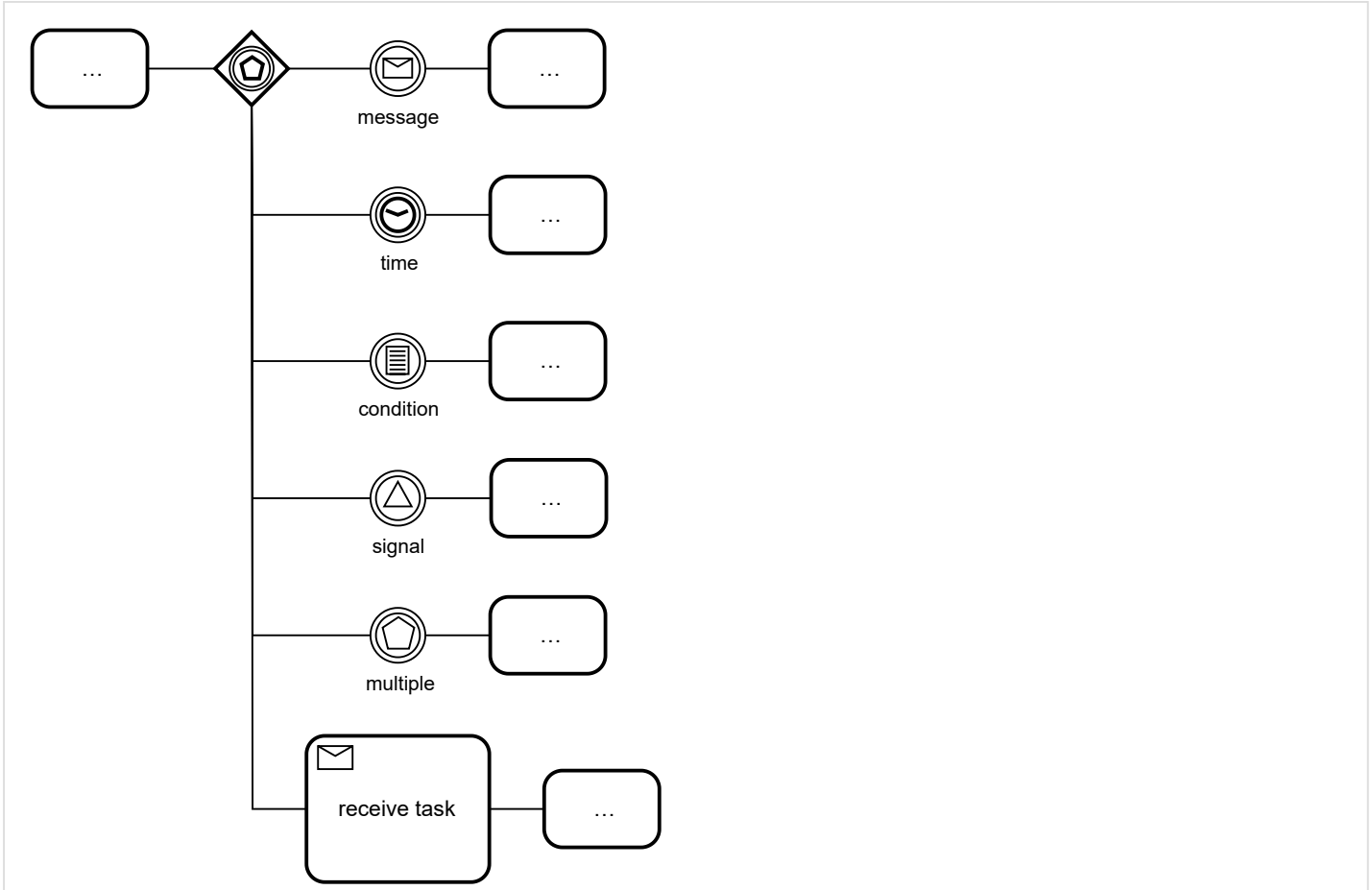
Heads up! In practice, handling OR gateways is not as simple as these examples imply. It’s easy to understand that progress depends on waiting for another token to reach an OR merge. It can be harder to trace the synchronization rules with complex diagrams that sprawl across several pages.

Event-based Gateways

We learned about the exclusive data-based (XOR) gateway option as a way to use different paths with regard to the data being processed. Users of other process notations recognize this type of branching, but BPMN gives us another way to design process paths: the event-based gateway – event gateway, for short. This gateway does not route based on data, but rather by which event takes place next. To understand the benefit, consider the process shown below: We order pizza and wait for it to be delivered. We can eat only after we receive the pizza, but what if the pizza doesn’t arrive after 60 minutes? We’ll make an anxious phone call, that’s what! We can model this with the event gateway:



As you can see here, not all intermediate events combine with the event gateway. You can, however, combine it with the receive task.



Events

Basic Concepts

Tasks and gateways are two of three flow elements we've come to know so far: Things (tasks) have to be done under certain circumstances (gateways). What flow element is still missing? The things (events) that are supposed to happen. Events are no less important for BPMN process models than tasks or gateways. We should start with some basic principles for applying them. We already saw Start events, intermediate events, and end events. Those three event types are also catching and/or throwing events:

Catching events are events with a defined trigger. We consider that they take place once the trigger has activated or fired. As an intellectual construct, that is relatively intricate, so we simplify by calling them catching events. The point is that these events influence the course of the process and therefore must be modeled. Catching events may result in:

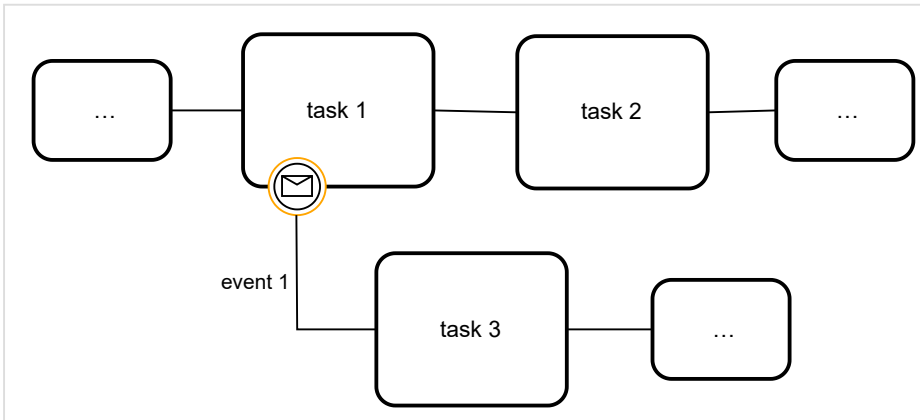
- The process starting
- The process or a process path continuing
- The task currently processed or the sub-process being canceled
- Another process path being used while a task or a sub-process executes

Throwing events are assumed by BPMN to trigger themselves instead of reacting to a trigger. You could say that they are active compared to passive catching events. We call them throwing events for short, because the process triggers them. Throwing events can be:

- Triggered during the process
- Triggered at the end of the process

We can also model attached intermediate events with BPMN. These do not explicitly require waiting, but they do interrupt our activities, both tasks and sub-processes. Such intermediate events are attached because we position them at the boundary of the activity we want to interrupt.

Hover over orange symbols for explanation

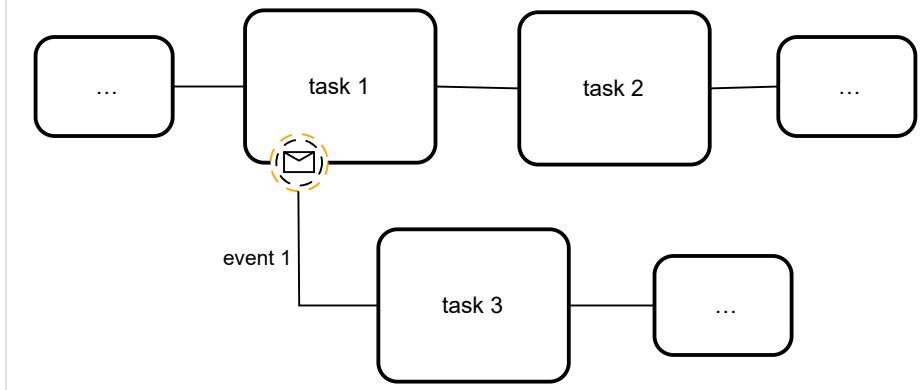


A token running through the process would behave this way:

- The token moves to task 1, which starts accordingly.
- If event 1 occurs while task 1 is being processed, task 1 is immediately canceled, and the token moves through the exception flow to task 3.
- On the other hand, if event 1 does not occur, task 1 will be processed, and the token moves through the regular sequence flow to task 2.
- If event 1 occurs only after task 1 completes, it will be ignored.

Through BPMN version 1.2, with the exception of compensation events, attached intermediate events inevitably resulted in canceled activities. BPMN 2.0 defines a new symbol: the non-interrupting intermediate event. It sounds awkward, but it is useful:

Hover over orange symbols for explanation



The token moves through the process as follows:

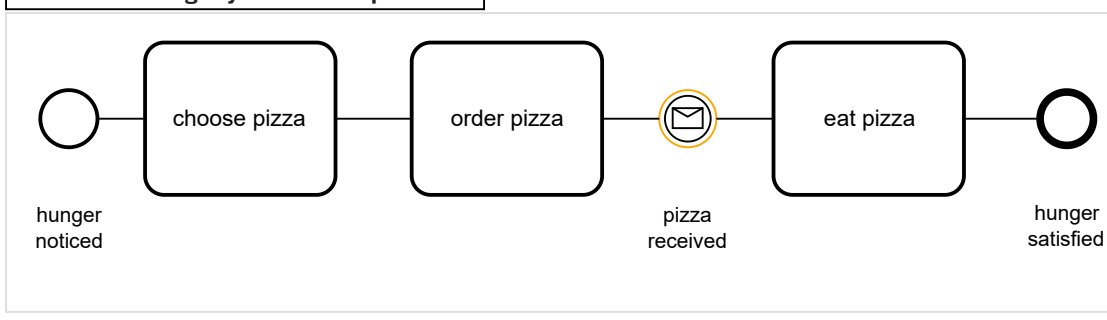
- The token moves to task 1, which starts accordingly.
- If event 1 occurs while task 1 is being processed, the token is cloned. Task 1 continues to be processed while the second token moves to task 3, which is now also processed. This procedure may even take place repeatedly, that is, the event can occur many times. Each occurrence results in another cloned token.
- If event 1 does **not** occur, task 1 will be completed, and the token moves through the regular sequence flow to task 2.
- If event 1 occurs only after task 1 completes, it ceases to matter.

In the following sections, we introduce the event types to be used when working with BPMN. We also explain how you can react to different events using the event-based gateway.

Message

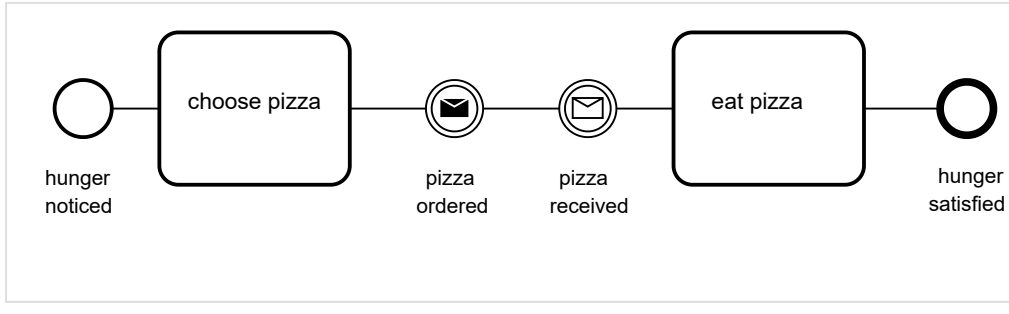
Sooner or later, most processes require communication, which can be represented in BPMN by means of the message event. You'll recognize it as the small envelope. The meaning of "message" in BPMN is not restricted to letters, e-mails, or calls. Any action that refers to a specific addressee and represents or contains information for the addressee is a message.

Hover over orange symbols for explanation



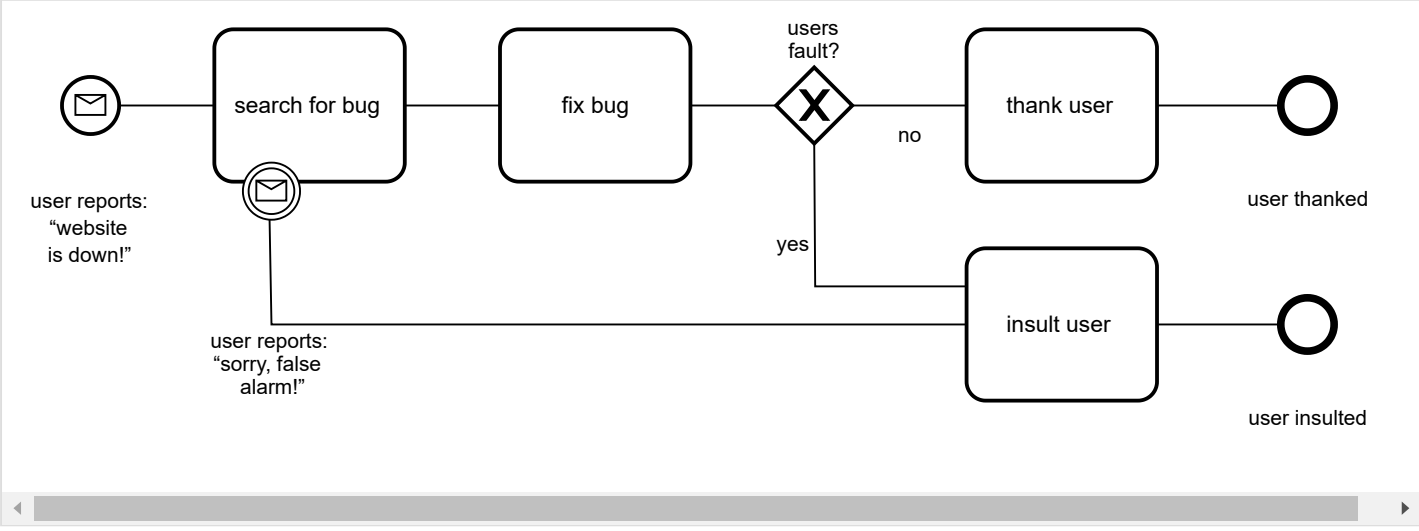
FAQ: Could I replace the task "order pizza" with a throwing message event?

Yes, you could:



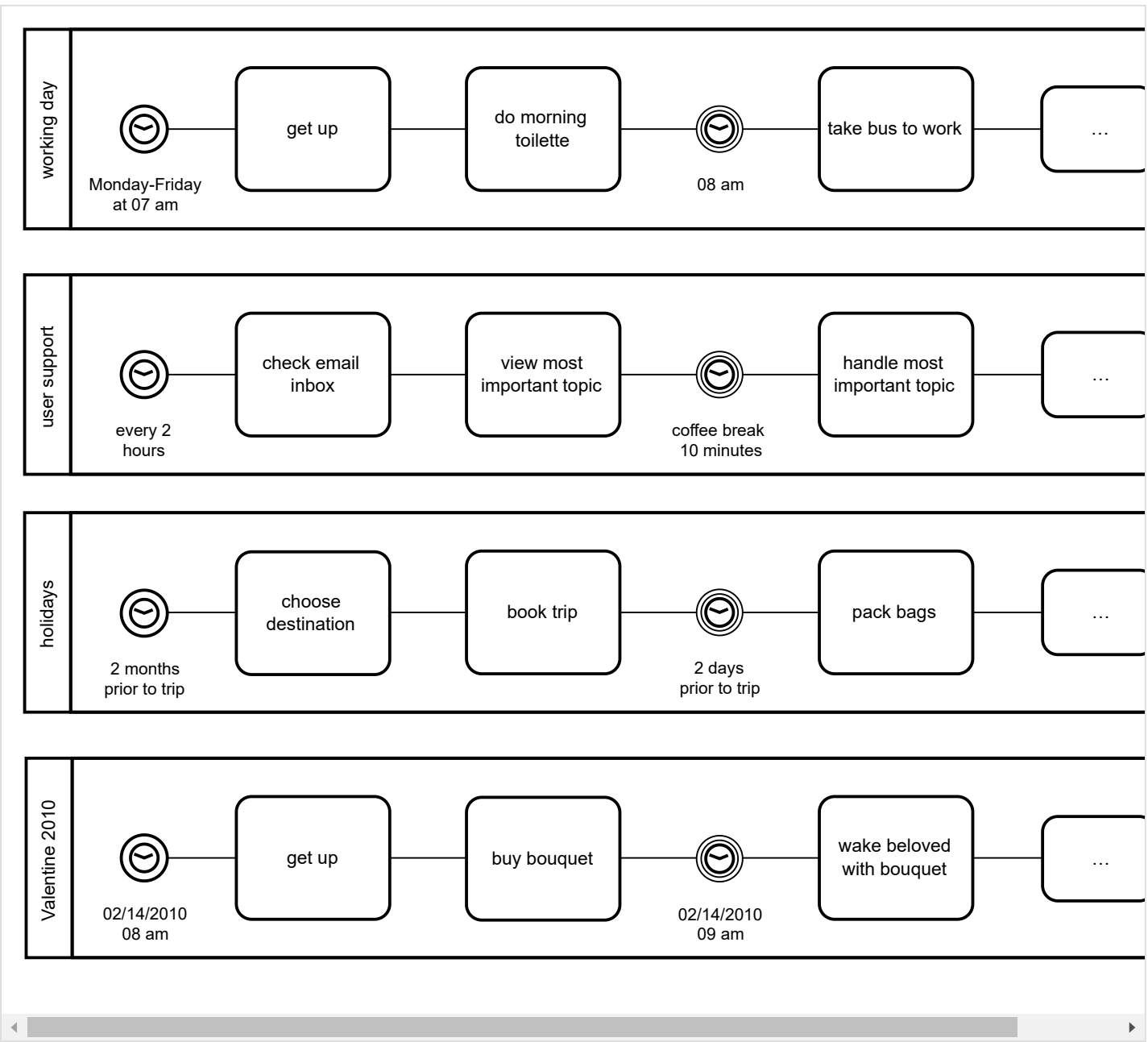
Heads up! We are not always happy with the throwing intermediate event. Implying a “send message” task without modeling it explicitly can easily confuse inexperienced consumers of our models. We choose not to use throwing intermediate events for messages and instead use a task. There are also special BPMN task types for sending and receiving messages.

In the example below, we show a message leading to cancellation. In this scenario, we administer a web application. When a user notifies us that the website does not work, we immediately search for the error. But maybe the user is mistaken, and the website is fine. Maybe the user’s Internet connection is defective. If the user tells us about the false alarm, we cancel the search and swear at the user for wasting our time. If the error is actually found, however, we eliminate it and simultaneously figure out who caused the error. If the user caused the error, we can swear at the user for a different reason. If the user is not at fault, however, we thank him or her graciously for letting us know about the problem.



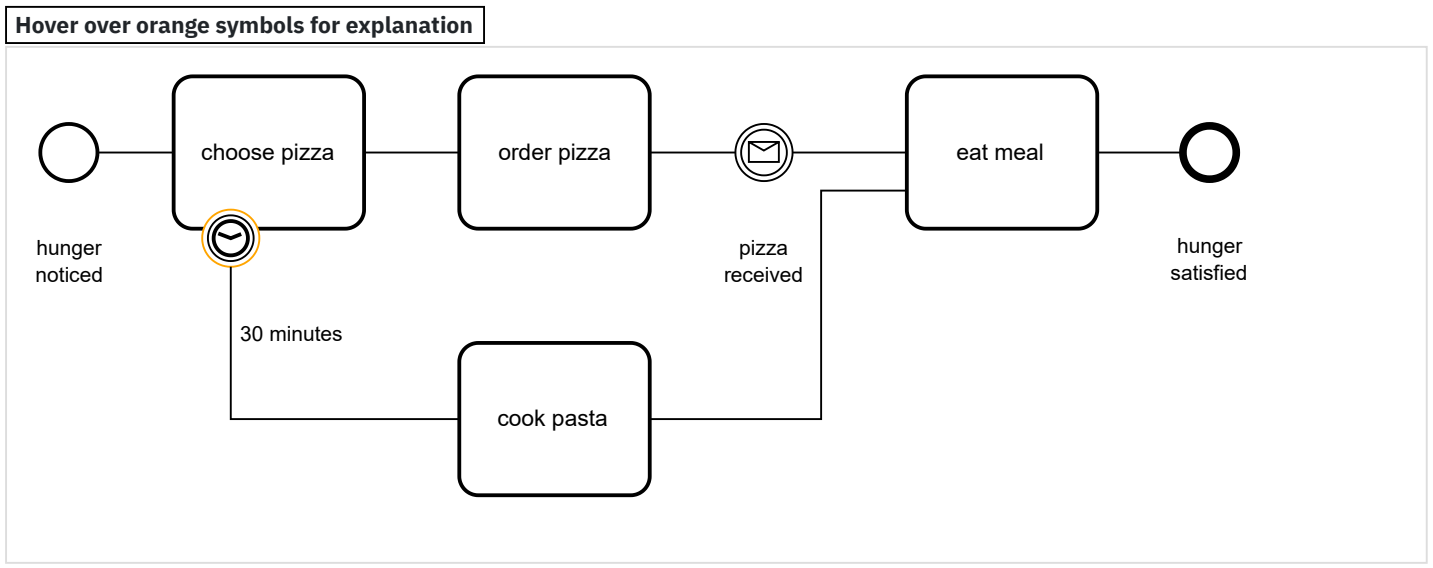
Timer

The timer event is often used when working with BPMN because it is so flexible to apply. A clock icon represents the timer event. The diagram below shows a few examples of applications:

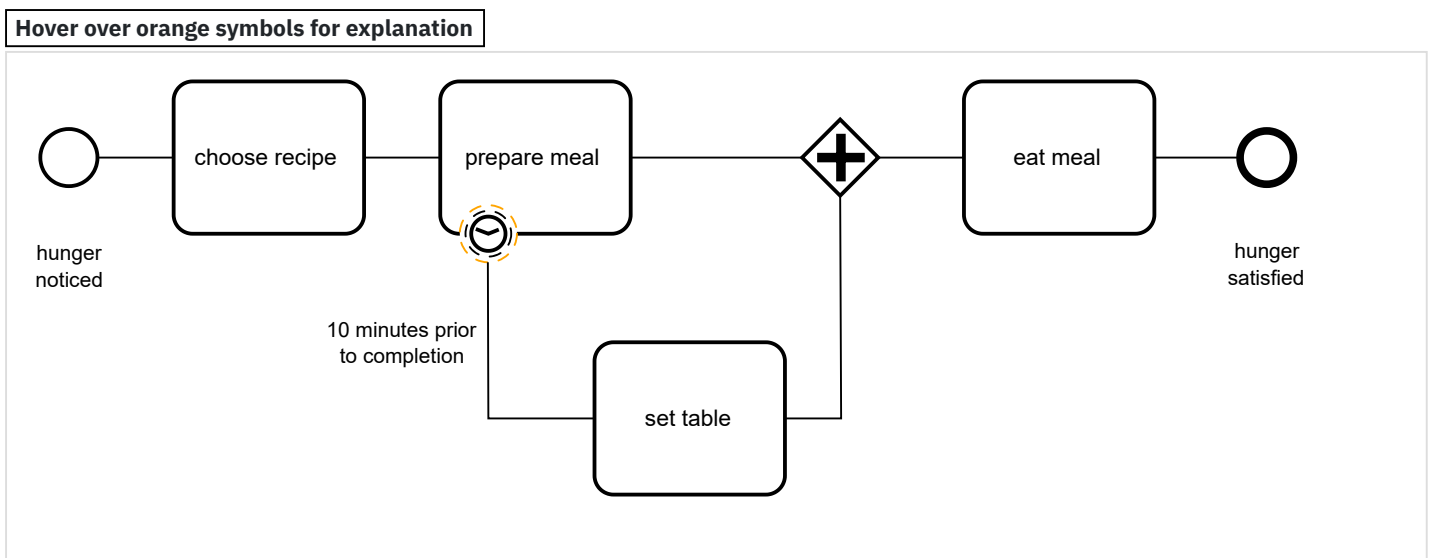


Time moves on no matter what we or our processes do, so timer events can exist only as catching starts or intermediate events.

You can model countdown times with an attached timer event. They are used this way frequently. You can specify upper time limits – the maximum time allowed for a processing task – for instance:



Non-interrupting timer events became possible with BPMN 2.0:



Error

Do your processes run completely error-free? If not, you can identify potential errors in your models as a step toward eliminating them, or as part of modeling escalation processes. In BPMN, error events are represented by a flash symbol.

The BPMN specification does not specify what an error may be. As the modeler, you have to decide that.

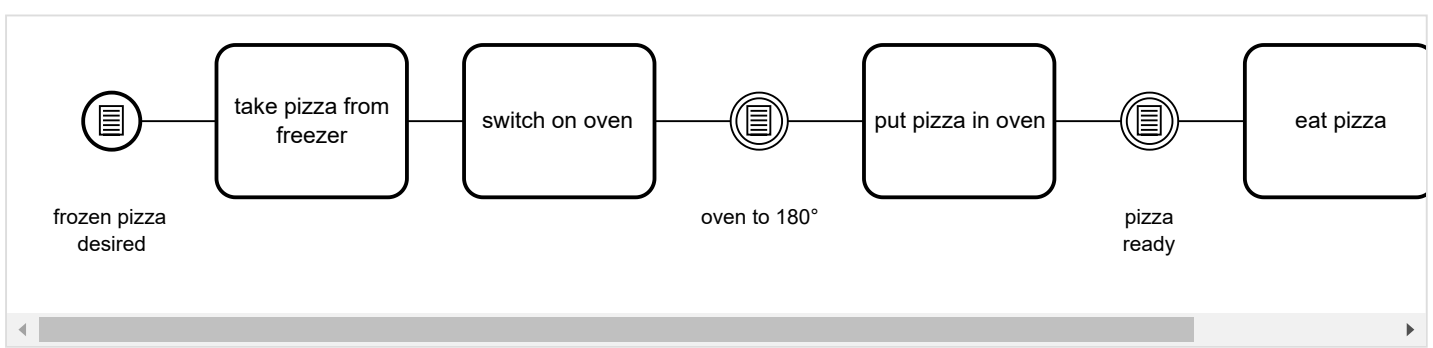
An error is a serious event in BPMN, so if catching, it can be modeled only as an attached intermediate event. This means that an error during task execution must be handled in a specific way: As a throwing event, it can be modeled only at the end of a process path so that the participant knows the process has failed. The parent process should likewise recognize the failure.

You can find example of error events e.g. in the **Implementation section for "Event Subprocesses"**.

Conditional

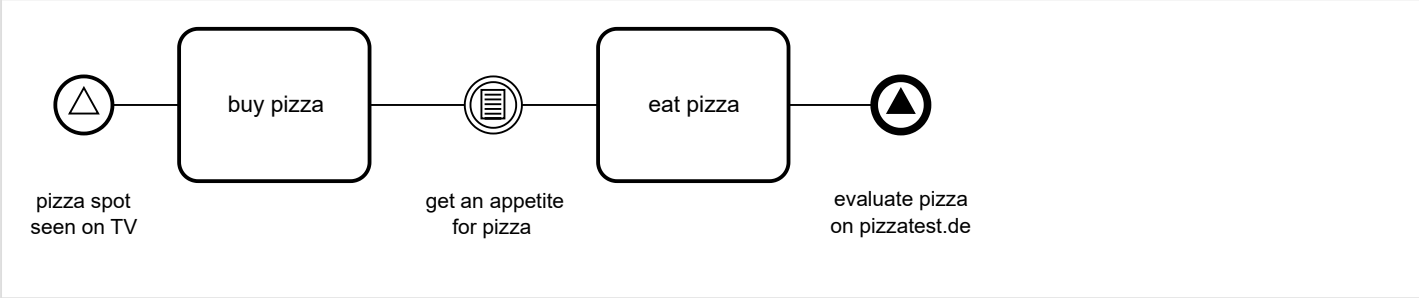
Sometimes we only want a process to start or to continue if a certain condition is true. Anything can be a condition, and conditions are independent of processes, which is why the condition (like the timer event) can only exist as a catching event. A process cannot therefore conditional event trigger a conditional event.

We can enhance our pizza process with conditions. If we want to have frozen pizza, the process starts as shown in the diagram below. We take the pizza from the freezer and turn on the oven. But only after the temperature in the oven reaches 180 C do we put the pizza in, and only after it is done do we take it out to eat.



Signal

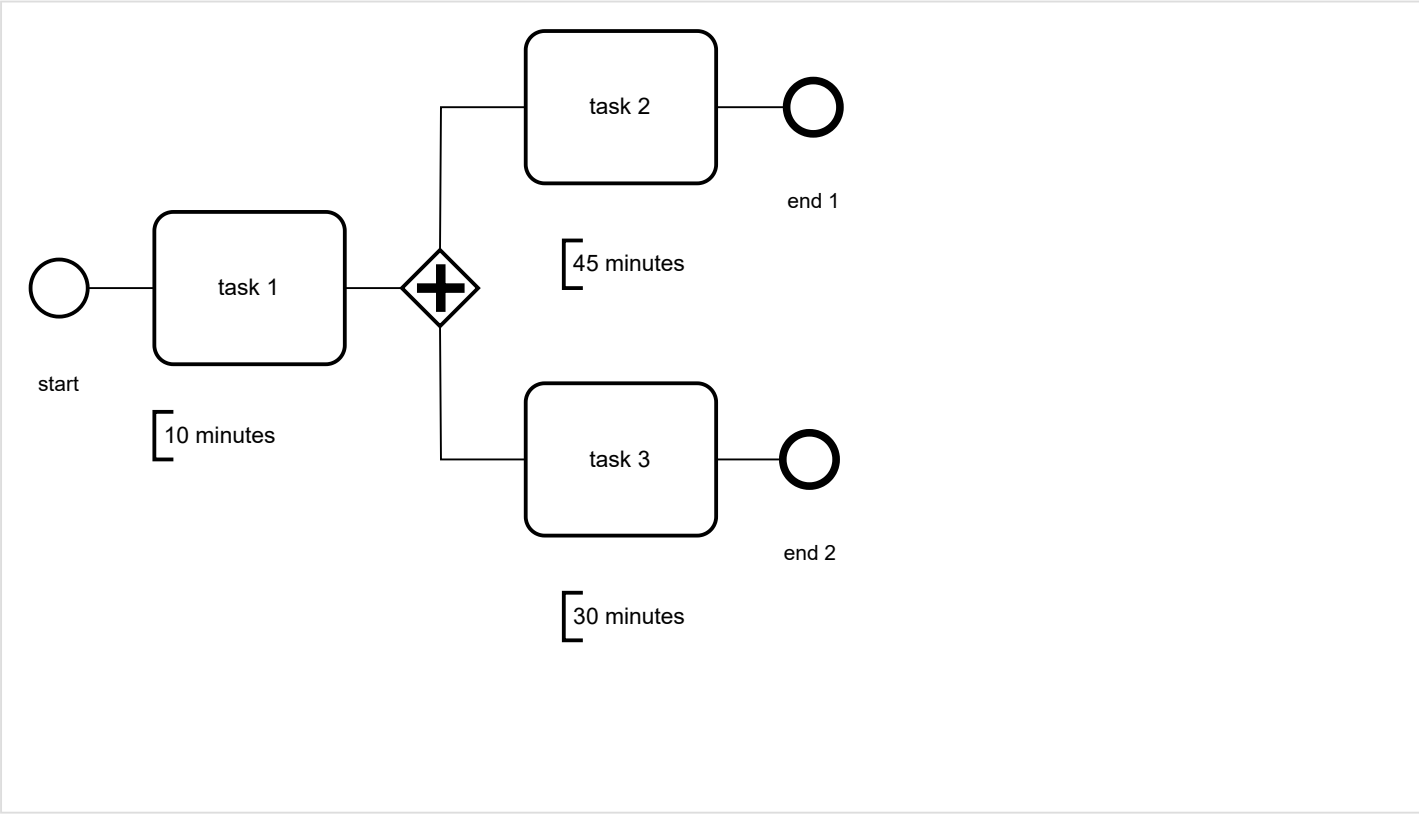
Signals are similar to messages, which is why you can model them in BPMN as events just as you can with messages. The symbol for a signal is a triangle. The essential difference between a signal and a message is that that latter is always addressed to a specific recipient. (An e-mail contains the e-mail address of the recipient, a call starts with dialing the telephone number, and so on.) In contrast, a signal is more like a newspaper advertisement or a television commercial. It is relatively undirected. Anyone who receives the signal and wants to react may do so.



We saw a new frozen pizza on TV, and we are keen to try it. The diagram above illustrates this new situation. We buy the pizza, but we keep it in the freezer until we're really hungry for pizza. That's a conditional event. After trying the new pizza, we go to Pizzatest.de to rate the new product. That's a signal. It is a signal for the general public too. (Pizzatest.de actually exists, by the way, which proves again that you can find simply **everything** on the Internet!)

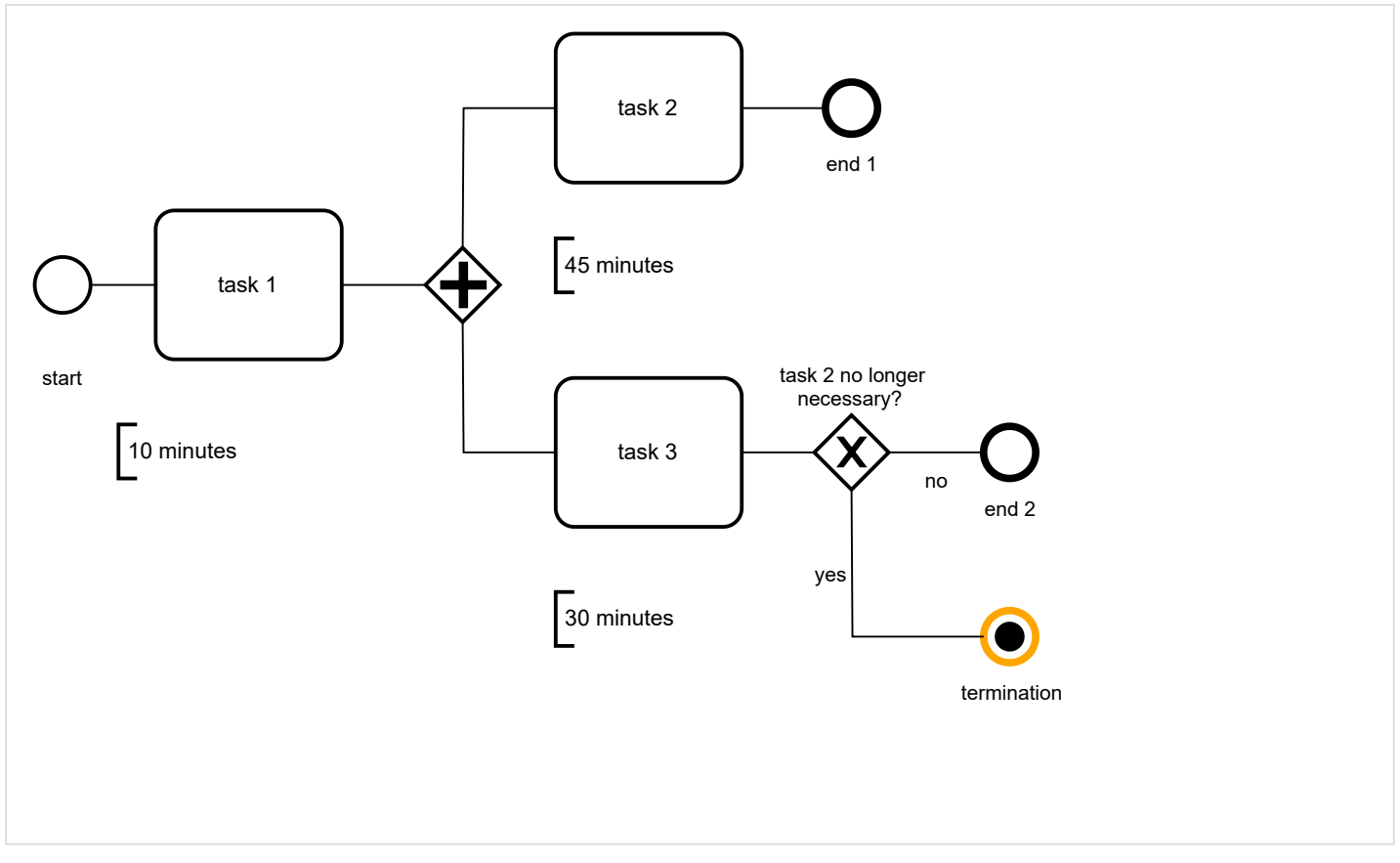
Termination

Let's look at the abstract example below. We already discussed (simple) Key Performance Indicator (KPI) analysis, and we therefore know that this process always takes 55 minutes. After task 1, tasks 2 and 3 can be processed simultaneously. Processing task 2 takes more time than does processing task 3, which is why it determines the runtime of the process. A token that runs through the process is cloned in the AND split. The first token stays in task 2 for 45 minutes; the second token stays in task 3 for 30 minutes. The second token arrives at the none event first, where it is consumed. After 15 more minutes, the first token arrives at the upper none event, where it is consumed too. Since no more tokens are available, the process instance finishes after 55 minutes.



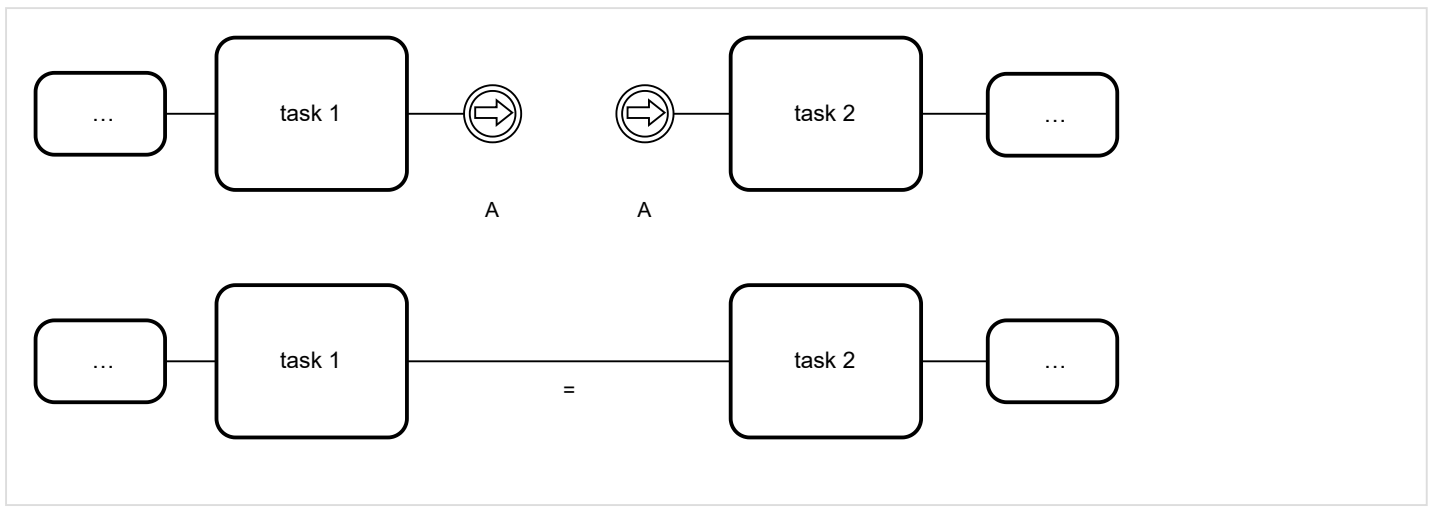
So far, so good, but what happens if we already know that, after having completed task 3, task 2 has become redundant? This is a frequent situation with parallel task executions related to content. In such cases, we can apply the pattern shown in here:

Hover over orange symbols for explanation



Link

The link event is a special case. It has no significance related to content, but it facilitates the diagram-creation process. As shown below, you can draw two associated links as an alternative to a sequence flow. Here, “associated” means there is a throwing link event as the “exit point,” and a catching link event as the “entrance point,” and the two events are marked as a pair – in our example by the designation “A.”



Link events can be very useful if:

- You have to distribute a process diagram across several pages. Links orient the reader from one page to the next.
- You draw comprehensive process diagrams with many sequence flows. Links help avoid what otherwise might look like a “spaghetti” diagram.

Link events can be used as intermediate events only.

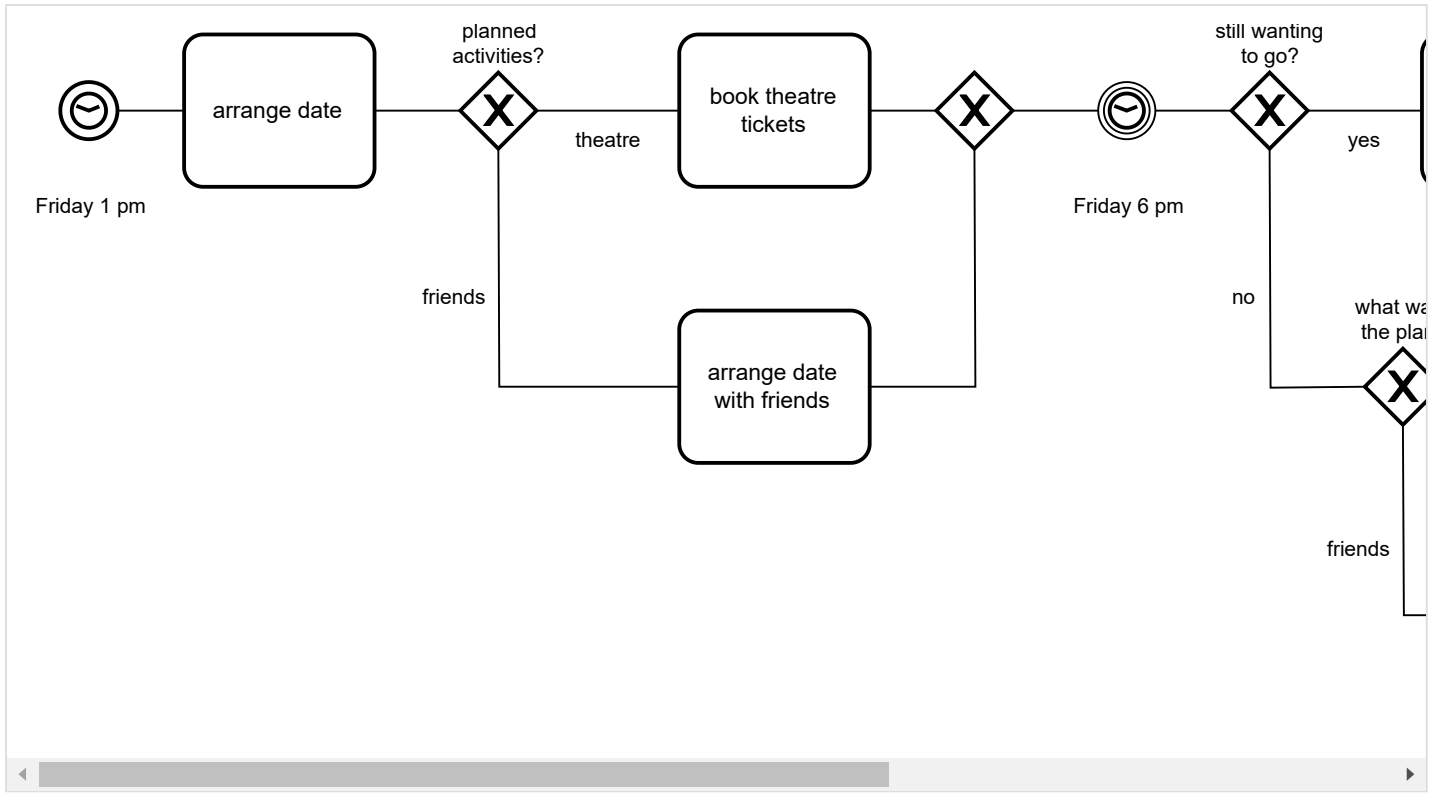
Compensation

We execute tasks in our processes that sometimes have to be canceled later under certain circumstances.

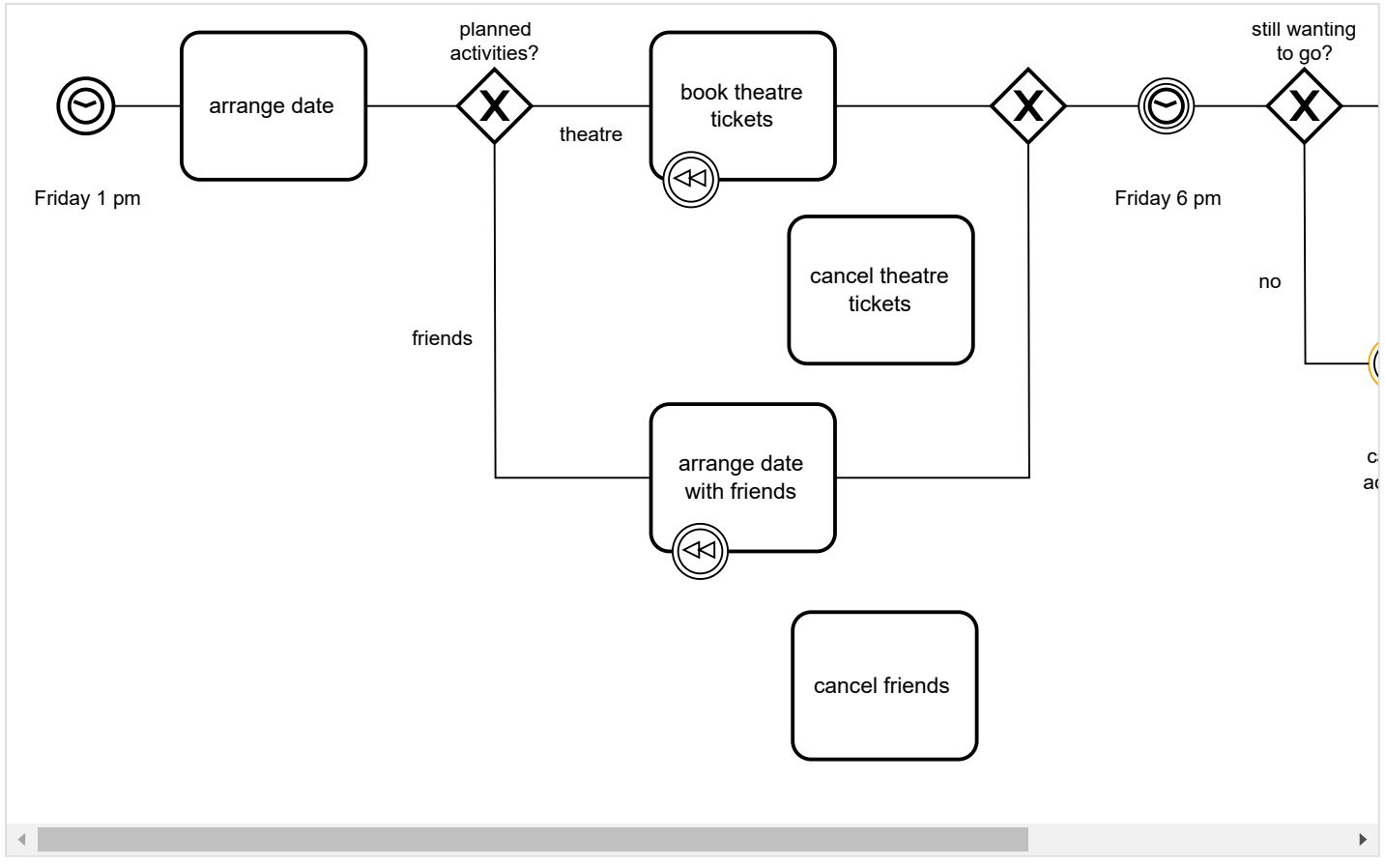
Typical examples are:

- Booking a train or airline ticket
- Reserving a rental car
- Charging a credit card
- Commissioning a service provider

Below, we see this process: On Friday at 1 p.m. we agree with our partner either to go to the theater or to spend the evening with friends. In both cases, we have to do something binding, either to reserve the theater tickets or make the arrangements with our friends. When evening arrives, perhaps we no longer feel like going out at all. We then have to cancel the arrangements we made with the theater or our friends before we can collapse in front of the TV in peace:



We can represent the latter part of the model more compactly with a compensation event, as shown in here:



There are special rules for handling compensations:

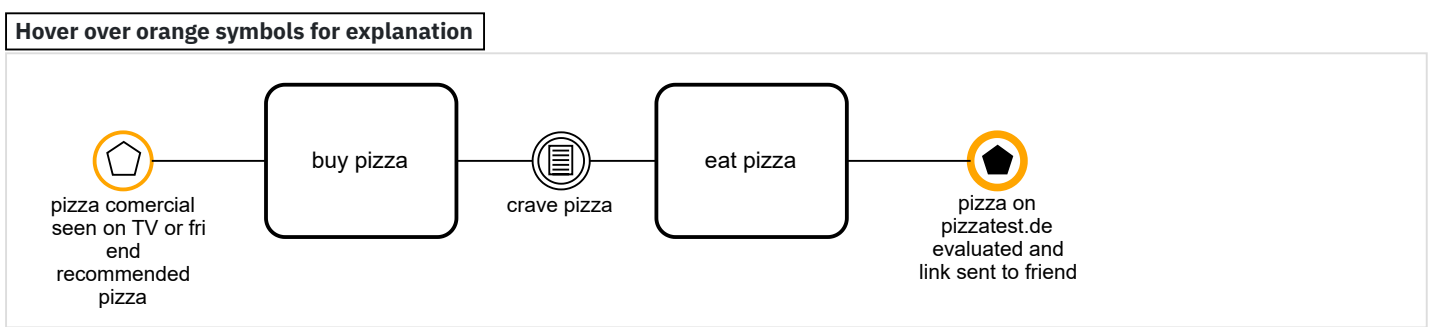
- Throwing compensations refer to their own processes, so the event is effective within the pool. This shows how this event type differs from a throwing message event.
- Other attached events can take effect only while the activities to which they are attached remain active. In contrast, an attached compensation takes effect only if the process triggers a compensation **and** the activity to which the compensation is attached successfully completed.
- Attached compensation events connect to compensation tasks through associations, and **not** through sequence flows, which would otherwise be common usage. BPMN thus emphasizes that compensations are beyond the regular process sequence; executing one is an exception.

Best Practice: Using Compensation Events

This example may be too simple to illustrate how much work this construct can save you. If you think of the complex business processes that frequently require compensations, however, you'll see how much leaner your models can be. You'll also be quick to spot the circumstances that demand compensations. We use compensation events only occasionally to describe complex processes.

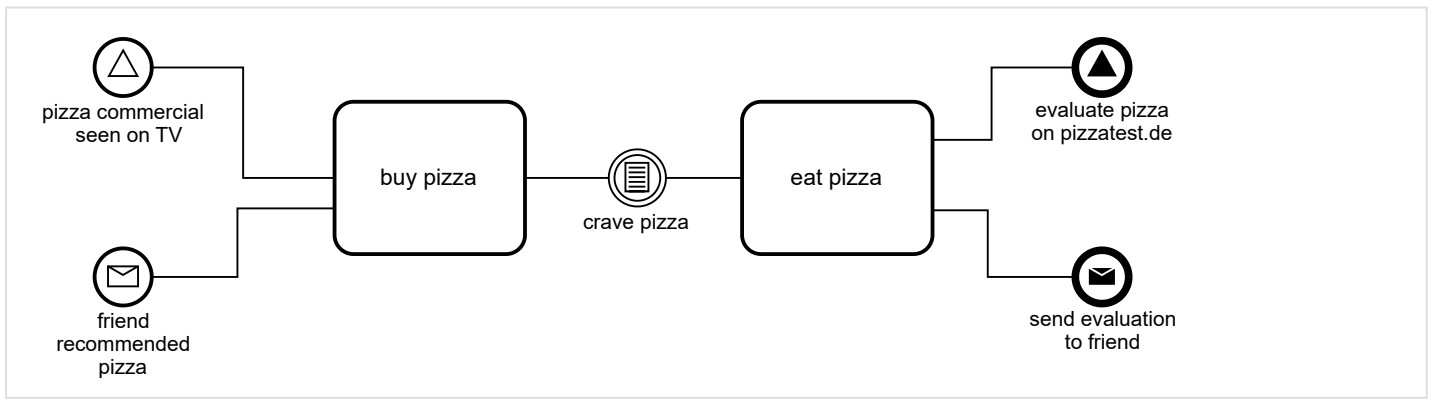
Multiple

We can use the multiple event to summarize several events with a single symbol. The diagram below applies the multiple event to our pizza scenario. In the example, we try a new pizza after having seen it on TV or after a friend recommended it. After eating it, we will rate the pizza on Pizzatest.de and in turn inform our friend if we also recommend this pizza.



Best Practice: Avoiding Multiple Events

You have to decide if multiple events serve your purposes. We concede their benefit in rough functional process descriptions, but they cease to be as useful in the more advanced technical-implementation phase. You can't afford to leave relevant details hiding in the descriptive text. We don't find the multiple event to be intuitive, nor is it helpful on a functional level. It may make your diagrams larger to model all events separately, but the resulting diagrams will be both more comprehensive and more comprehensible. The bottom line is that we have never used this symbol in practice, nor have we seen anybody else doing so.The model in here describes the same process, but the events are fully modeled:



Parallel

The parallel event was added in BPMN 2.0 to supplement the multiple event. While a catching multiple event has XOR semantics – it occurs as soon as **one** of its contained events occurs – the parallel event uses AND semantics. It doesn't occur until **all** of its contained events occur. Because the throwing multiple event already implies AND semantics, the specification defines parallel events as catching events only.

Escalation

The BPMN 2.0 specification added the escalation event. Mainly, it shows communication between parent and sub-processes.

Cancel

You can use the cancel event only in the context of the transactions.