

## Chapter 7

# The Activity Diagram

The *activity diagram* focuses on modeling procedural processing aspects of a system. It specifies the control flow and data flow between various steps—the *actions*—required to implement an activity.

*Control flow and data flow*

In UML 2, activity diagrams use flow-oriented language concepts that find their origins in languages for defining business processes. Activity diagrams are also based on established concepts for describing concurrent communicating processes, such as the token concept of Petri nets [41]. One particular feature of activity diagrams is their support for modeling both object-oriented systems and non-object-oriented systems. They allow you to define activities independently of objects, which means, for example, that you can model function libraries as well as business processes and real-world organizations.

*Modeling of object-oriented and non-object-oriented systems*

The UML standard does not stipulate any specific form of notation for activities. In addition to the flow-based notation elements of the activity diagrams, the standard also allows other forms of notation, such as structural diagrams or even pseudocode. A number of recurring control flow and data flow patterns have emerged in addition to custom notation elements. They are used in particular for modeling business processes and have proven to be very useful for complex processes. These constructs are referred to as “workflow patterns”. For an overview of these types of patterns as well as guidance on how to model the patterns based on the concepts of UML 2 activity diagrams, see Wohed et al. [44].

*“Workflow patterns”*

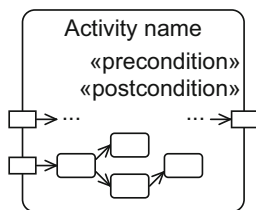
In this chapter, observant readers will note that not all examples model complete processes—some of the models are restricted to extracts of processes. Thus, for example, some of the diagrams do not contain initial and final nodes. In practice, however, a complete activity diagram must have clearly defined start and end points.

## 7.1 Activities

### Activity

An activity diagram allows you to specify user-defined behavior in the form of activities. An *activity* itself can describe the implementation of a use case. At a very detailed level, it can also define the behavior of an operation in the form of individual instructions, or at a less detailed level, model the functions of a business process. A business process defines how business partners have to interact with one another to achieve their goals. It can also describe the internal processes within a company. Behavior can thus be defined at different levels of granularity. An activity can be assigned to an operation of a class but it can also be autonomous.

### Activity

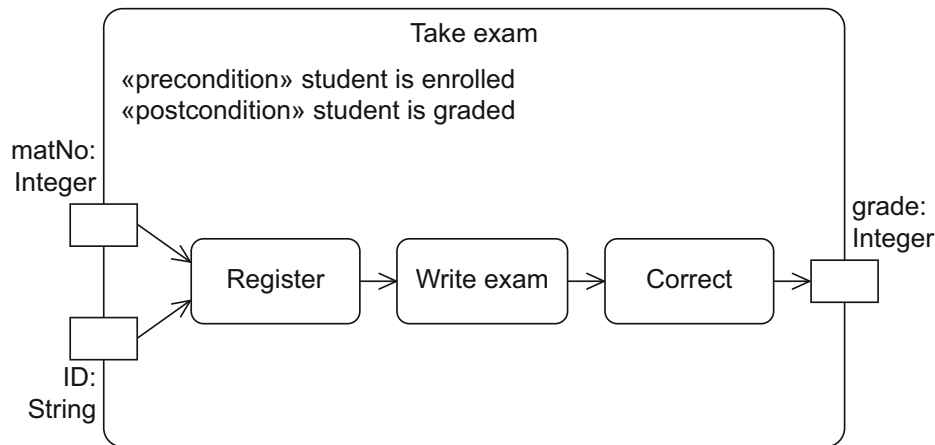


The content of an activity is—as with Petri nets—a directed graph whose nodes represent the components of the activity like actions, data stores, and control elements and whose edges represent the control flow or object flow, that is, the possible execution paths for the activity.

An activity is depicted as a rectangle with rounded corners and can, just like an operation, have *parameters*. These are shown as rectangles arranged overlapping at the boundary of the activity. To make the diagram easier to read, you should position *input parameters* at the left or upper boundary and *output parameters* at the right or lower boundary of the activity. This allows an activity to be read from left to right and/or from top to bottom. The values that are transferred to the activity via the input parameters are available to those actions that are connected to the input parameters by a directed edge (see the next section). In the same way, output parameters receive their values via directed edges from actions within the activity. The example in [Figure 7.1](#) shows the steps necessary to execute the activity Take exam. The input parameters are the matriculation number and the study program ID of a student. The actions Register, Write exam, and Correct are executed in this activity. The result of the activity is a grade. This example diagram does not show, however, who performs which action. To enable actions to be assigned to specific actors, the activity diagram offers the concept of partitions, which we will introduce in Section 7.5.

### Precondition and postcondition

You can specify preconditions and postconditions for an activity. These indicate which conditions have to be fulfilled before or after the activity is executed. The keywords «precondition» and «postcondition» are used to identify the respective conditions. In [Figure 7.1](#), a student who wants to take an exam must be enrolled. After the activity Take exam has been executed, the student must be graded.



**Figure 7.1**  
Example of an activity

## 7.2 Actions

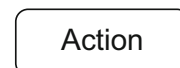
The basic elements of activities are *actions*. Just like an activity, an action is depicted as a rectangle with rounded corners, whereby the name of the action is positioned centrally within the rounded rectangle. You can use actions to specify any user-defined behavior. There are no specific language requirements for the description of an action. Therefore, you can define the actions in natural language or in any programming language. For example, if, as a result of the execution of an action, the value of the variable *i* is to be increased by one, you can express this by using `i++` or simply by writing `Increase i by one`.

Actions process input values to produce output values, which means that they are able to perform calculations. They can also load data from a memory and they can change the current state of a system. In the example in [Figure 7.1](#), `Register`, `Write exam`, and `Correct` are actions.

Within the context of an activity, actions are always atomic—that is, they cannot be broken down further within the modeled context. However, an action can refer to another activity that itself consists of actions (see call behavior action on page 145). Actions are considered atomic, even though they might consist of multiple individual steps. For example, registering for an exam usually requires multiple steps such as logging on to the system and selecting the appropriate course and the exam date. Despite this, in [Figure 7.1](#), we have intentionally modeled `Register` as an action rather than an activity. This is because in the present model, the execution of a registration is considered as a single step; the internal details of this step are of no interest to us, however, and therefore we do not break it down further.

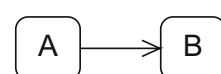
As we can see in [Figure 7.1](#), actions and parameters are connected to one another via directed edges. These edges express the order in which the actions are executed and thus define the execution steps of

*Action*



*Atomicity of actions*

*Edge between actions*



### Control flow edge versus object flow edge

an activity. Here we differentiate between *control flow edges* and *object flow edges*: control flow edges only define the order between actions, whereas object flow edges can also be used to exchange data or objects. This enables you to express a data dependency between a preceding action and a subsequent action. The completion of an action can initiate the execution of another action if these two actions are connected with one another via control flow edges or object flow edges. However, an action may only be executed if all previous actions have been completed successfully, all relevant guards evaluate to true and all input parameters have values. Guards are conditions that must be fulfilled to enable the transition from one activity or action to another activity or action. They usually occur in connection with alternative branches. We will look at the control flow, including the guards, and the object flow more closely in the following two sections after we have introduced two special types of actions.

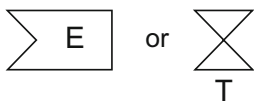
### Predefined actions

In UML, there are a number of predefined, non-language-specific actions that you can model easily in any target language due to their level of detail.

The predefined actions in UML can be classified into different categories based on their function and complexity. We will discuss the most important two of these categories in the following sections.

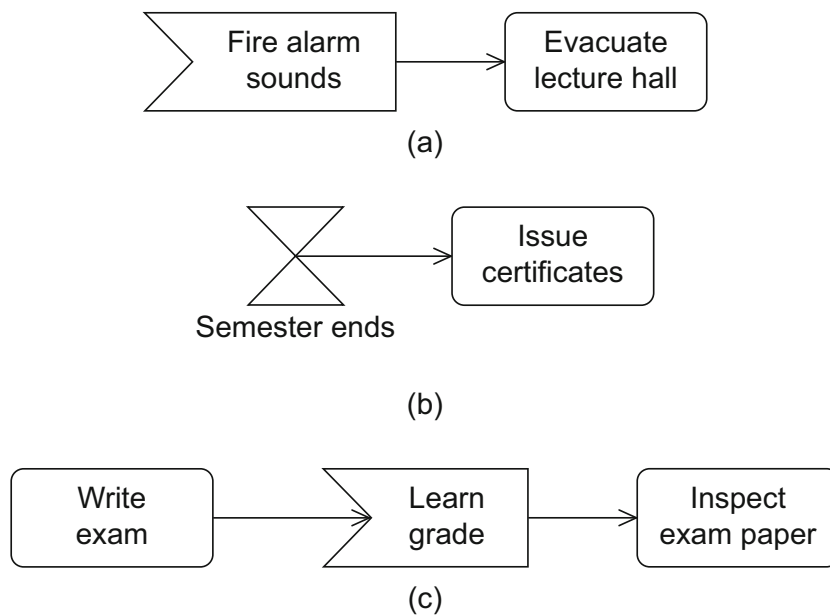
## 7.2.1 Event-Based Actions

### Accept (time) event action



Event-based actions enable objects and signals to be transmitted to receiver objects. They allow you to distinguish between different types of events. You can use an *accept event action* to model an action that waits for the occurrence of a specific event. The notation element for an accept event action is a “concave pentagon”—a rectangle with a tip that points inwards from the left. If the event is a time-based event, you can use an *accept time event action*, whereby in this case, the notation is an hourglass.

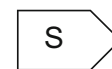
Accept (time) event actions do not necessarily have incoming edges. If they do not have incoming edges, they start when the corresponding event occurs. They remain active, that is, they can receive signals until the activity that contains them is ended. [Figure 7.2](#) shows three examples of accept (time) event actions: whenever a fire alarm is triggered, the lecture hall must be evacuated ([Fig. 7.2\(a\)](#)); at the end of a semester, certificates are issued ([Fig. 7.2\(b\)](#)); when a student has taken an exam, the student waits for the grade and inspects the exam paper when receiving the grade ([Fig. 7.2\(c\)](#)).

**Figure 7.2**

Examples of accept event actions (a+c) and accept time event actions (b)

To send signals, you can use *send signal actions*. Send signal actions are denoted with a “convex pentagon”—a rectangle with a tip that protrudes to the right. The action Send grade in Figure 7.3(b) is an example of a send signal action.

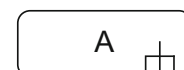
*Send signal action*



### 7.2.2 Call Behavior Actions

Actions can call activities themselves. These actions are referred to as *call behavior actions* and are marked with an inverted fork symbol. This fork symbol indicates a hierarchy. It symbolizes that the execution of this action starts another activity, thus dividing the system into various parts. Figure 7.3(a) shows an example of a call behavior action. In this diagram, the action Issue certificate in the activity Organize exam refers to an activity that specifies Issue certificate in more detail. Within the context of the activity Organize exam, the internal steps that lead to the issue of a certificate are not relevant. Therefore, Issue certificate is seen as an atomic unit here, even though it involves a process with multiple actions.

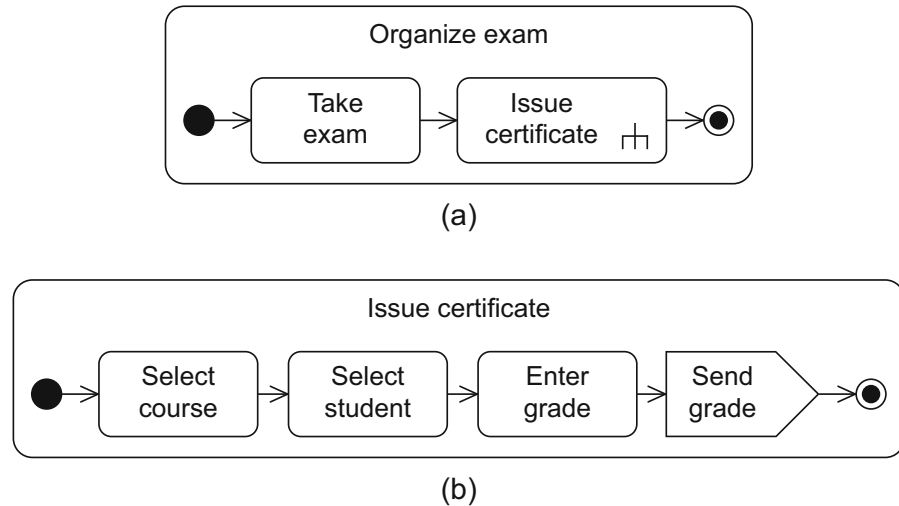
*Call behavior action*



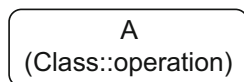
The content of the called activity can be depicted elsewhere in this or even another activity diagram in the form of an activity with the usual notation for activities that we have already seen. Figure 7.3(b) shows the details of the called activity Issue certificate with the input parameter grade.

**Figure 7.3**

Example of a call behavior action and a send signal action



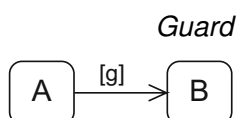
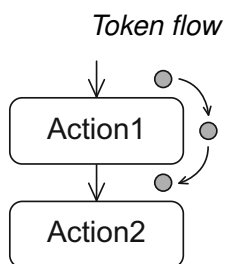
#### Call operation action



An action can also trigger the call of an operation. This type of action is referred to as a *call operation action*. It is represented in a rectangle with rounded edges. If the name of the operation does not match the name of the action, the name of the operation can be specified beneath the name of the action in the form (ClassName::operationName).

## 7.3 Control Flows

#### Token for describing flows

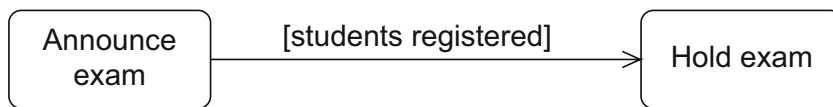


Activities consist of other activities and actions that are connected to one another by edges. If we look at the graph of an activity, the static structure does not show clearly how the execution works. To integrate the dynamic behavior aspects into the diagram, we need execution semantics, meaning that we have to specify exactly how an activity diagram is executed.

The token concept, as introduced in Petri nets [41], is the basis for the execution semantics of the activity diagram. A token is a virtual coordination mechanism that describes the execution exactly. In this context, virtual means that the tokens are not physical components of the diagram. They are mechanisms that grant actions an execution permission.

If an action receives a token, the action is active and can be executed. Once the action has ended, it passes the token to a subsequent node via an edge and thus triggers the execution of this action. Once this action has ended, it passes the token to the outgoing edges or retains it until a certain condition is fulfilled.

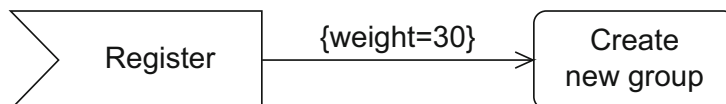
The passing of a token can be prevented by a *guard* evaluating to false. A guard is specified in square brackets. In the example in [Figure 7.4](#), an exam is only held if students have registered for it.



**Figure 7.4**  
Example of a guard

An action can only be executed if tokens are present at all of its incoming edges. If an action has multiple outgoing edges, a token is offered to all target nodes of these edges, thereby causing a split into multiple independent execution paths. Special nodes are also available as an alternative for modeling this concurrency. We will look at these in more detail later on.

When an action is executed, usually one token of each of its incoming edges is consumed. Alternatively, a *weight* may be placed on an edge to allow a certain number of tokens to be consumed at that edge by a single execution. The weight of an edge is specified in curly brackets with the keyword *weight*. It is always a whole number greater than or equal to zero. If the weight is zero, this means that all tokens present are consumed. Alternatively, all or \* can also be written instead of zero. If no weight is specified, 1 is assumed as the default value. Figure 7.5 gives an example of the use of weights. If the signal Register is received 30 times, meaning that at least 30 students have registered and thus 30 tokens are offered to the subsequent action, then this subsequent action is executed, consumes 30 tokens, and a new group is created.



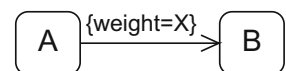
**Figure 7.5**  
Example of the weight of an edge

If two actions that are to be connected to one another via an edge are far apart in a diagram, you can use *connectors* to make the diagram clearer. In this case, you do not have to draw the edge as a continuous line from one action to the other; instead, the connector acts as a continuation marker comparable to the continuation markers in sequence diagrams (see page 129). A connector is depicted as a small circle containing the name of the connector. Each connector must appear twice in an activity: once with an incoming edge and once with an outgoing edge. Figure 7.6 models a relationship between two actions, once without a connector (Fig. 7.6(a)) and once with a connector (Fig. 7.6(b)).

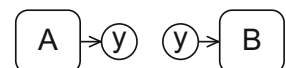
The activity diagram offers special nodes for controlling the control flow. These nodes are called *control nodes*.

The *initial node* indicates where the execution of an activity begins. It does not have any incoming edges but has at least one outgoing edge and is noted as a solid black circle. As soon as an activity becomes

*Weight of an edge*



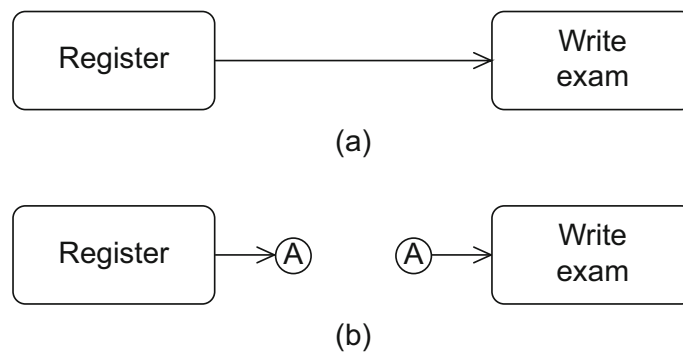
*Connector*



*Initial node*

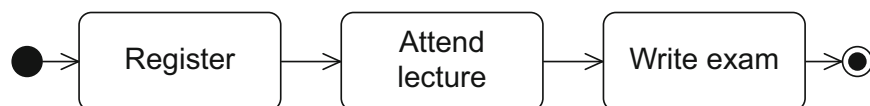


**Figure 7.6**  
Example of a connector



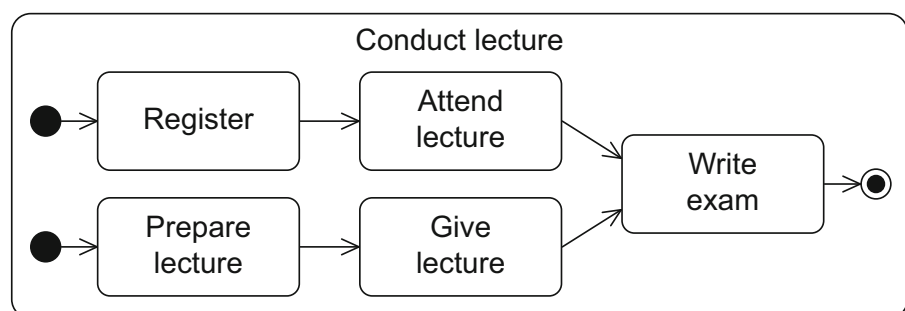
active, tokens are provided at all outgoing edges of an initial node and thus the activity is started. Figure 7.7 shows an example of an initial node. The diagram also contains an activity final node that represents the end of an activity. We will look at the activity final node, noted by a solid black circle within another circle, in more detail later in this chapter.

**Figure 7.7**  
Example of an initial node



Multiple initial nodes are also permitted for each activity. This allows you to express concurrency, meaning that multiple execution paths can be active simultaneously. If an activity with multiple initial nodes is called, the outgoing edges of all initial nodes are supplied with tokens simultaneously. The example in Figure 7.8 shows two concurrent subpaths of the activity Conduct lecture. If the activity Conduct lecture is activated, a token is placed at each of the two initial nodes and thus both subpaths are activated. One subpath relates to the actions of students and the other subpath refers to the actions performed by a lecturer. In the action Write exam, both paths are merged. A token must be present at both incoming edges for the action Write exam to be executed.

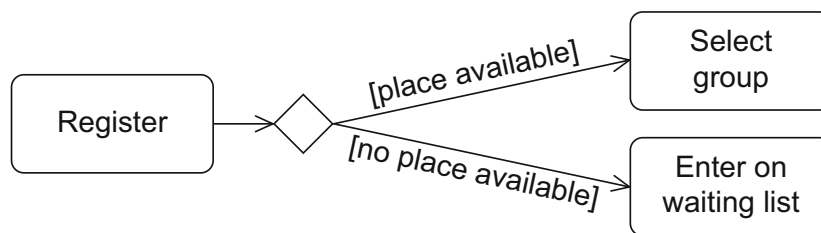
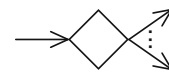
**Figure 7.8**  
Example with multiple initial nodes





In an activity diagram, you can model alternative branches using *decision nodes*. These act as a switch point for tokens and correspond to the `if` statement in a conventional programming language. A decision node is depicted as a diamond with one incoming edge and multiple outgoing edges. The outgoing edges have guards (also referred to as conditions). Just like in the other UML diagrams, guards are boolean expressions enclosed within square brackets. These conditions must not overlap, meaning that the system must be able to clearly decide which outgoing edge a token should take in a specific situation. For example, having one outgoing edge with the condition  $[x > 1]$  and another outgoing edge with the condition  $[x < 3]$  is not allowed, as there would be no unique choice of the edge a token should take if  $x = 2$  applies. If there is a token at a decision node, the system must be able to clearly decide, based on the current context (for example, dependent on the value of a variable), which path the token takes to exit the decision node. [Figure 7.9](#) shows an example of a decision node. If the action Register is executed, it is followed by the action Select group if there are still places available. If this is not the case, the action Enter on waiting list is executed.

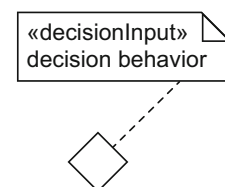
Decision node



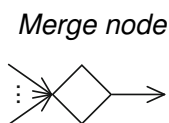
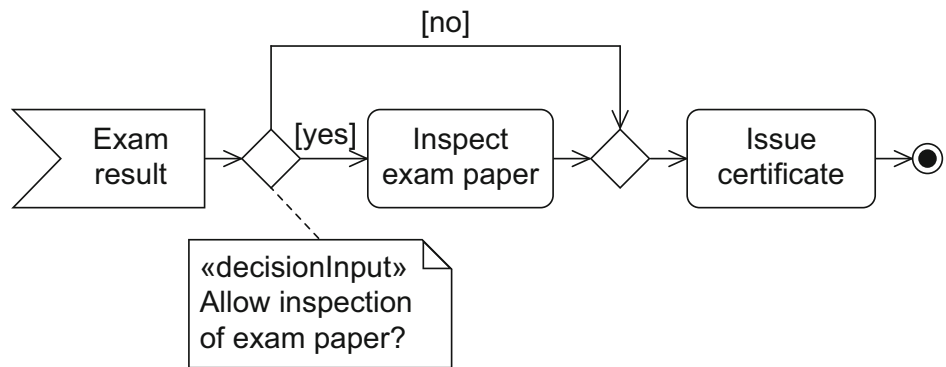
**Figure 7.9**  
Example of a decision node

You can specify *decision behavior* for a decision node. This means that you can specify behavior that is necessary for the evaluation of the guards. It allows you to avoid situations in which the same calculations have to be performed multiple times for different guards as the result of the calculation can be accessed in every guard. However, this behavior must not result in any side effects, meaning that the execution of the behavior defined in a decision node must never change the values of objects and variables. The decision behavior is attached to the decision node as a comment with the label `«decisionInput»`. [Figure 7.10](#) shows an example of this. As soon as the exam results are known, a decision is taken in a central department regarding whether to offer students the opportunity to inspect their corrected exam papers. If the decision is positive, the students are allowed to inspect their exam papers. Their certificates are issued afterwards. If the decision is negative, the certificates are issued immediately.

Decision behavior

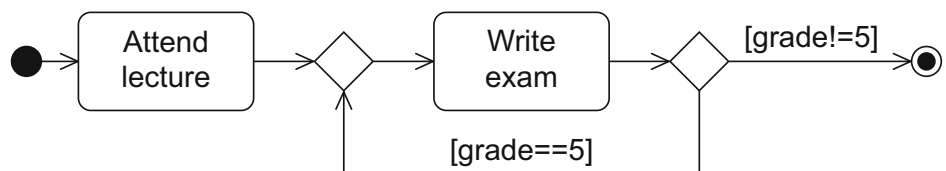


**Figure 7.10**  
Example of decision  
behavior

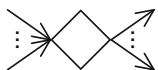


If you want to bring alternative subpaths back together again, you can do this using the *merge node*. This node is also depicted as a diamond but with multiple incoming edges and only one outgoing edge. In particular, a token may only be present at one incoming edge at most. Using decision and merge nodes, we can now model execution steps that are repeated, that is, loops (see [Fig. 7.11](#)).

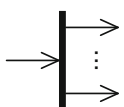
**Figure 7.11**  
Example of a loop



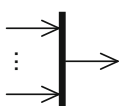
*Combined decision and  
merge node*



*Parallelization node*



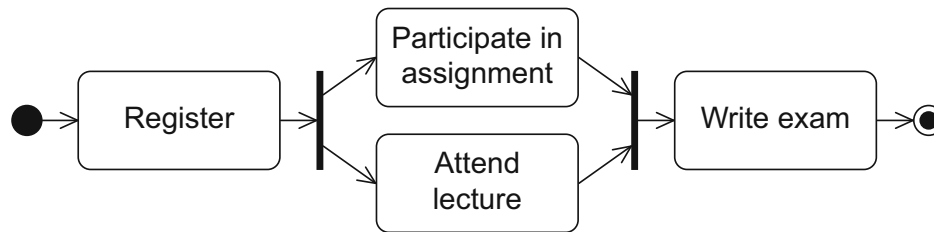
*Synchronization node*



You can also combine decision and merge nodes. This combined node then has multiple incoming edges and multiple outgoing edges.

For decision and merge nodes, only one of the possible paths is active. As already mentioned, you can use multiple initial nodes to model concurrency at the beginning of an activity. If an execution path splits into multiple simultaneously active execution paths later on, you can realize this using a *parallelization node*. A parallelization node is depicted as a black bar with one incoming edge and multiple outgoing edges. [Figure 7.12](#) shows an example of this type of node. Once a student has registered for a course, the student attends the lecture and participates in the assignment simultaneously. The student can only write the exam when both the lecture and the assignment have been completed (see next paragraph).

You can merge concurrent subpaths using a *synchronization node*. This node is the counterpart to the parallelization node. It is depicted as a black bar with multiple incoming edges but only one outgoing edge. As soon as tokens are present at all incoming edges, that is, as soon as all preceding actions have been executed, all incoming tokens are merged into one token that is passed on at the outgoing edge.



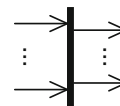
**Figure 7.12**  
Example of the use of parallelization and synchronization nodes

In the same way that you can combine decision and merge nodes, you can also combine synchronization and parallelization nodes using a bar with multiple incoming edges and multiple outgoing edges.

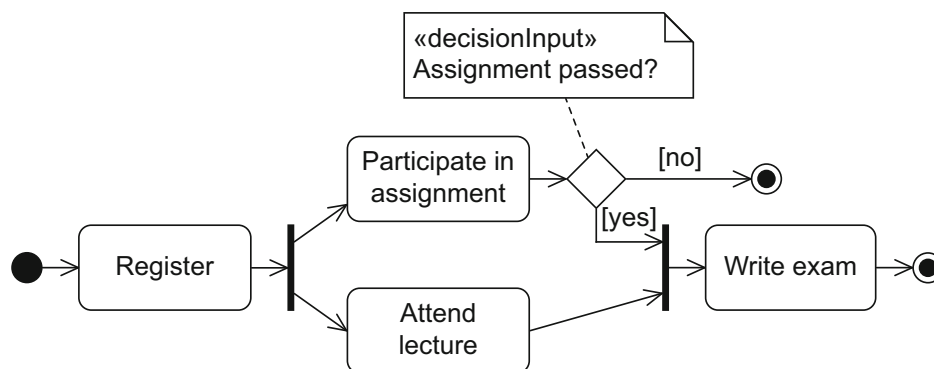
To express the end of an activity, the activity diagram offers a special node for this purpose: the *activity final node*. This node is depicted as a small circle containing a solid circle and is often referred to as a “bull’s eye”. If a token is present at an incoming edge of an activity final node, the entire activity is terminated—that is, all active actions of this activity are terminated. This also includes active concurrent subpaths and thus all tokens in the activity are deleted. An exception to this rule is data tokens that are already present at the output parameters of the activity (see the next section). If a diagram contains multiple activity final nodes, the first one reached during the execution ends the entire activity. For example, participation in a course is ended if either the assignment has not been passed or the exam has been taken (see Fig. 7.13).

You can merge multiple activity final nodes into one activity final node with multiple incoming edges. As soon as one token reaches the activity final node via an incoming edge, the entire activity is ended.

*Combined parallelization and synchronization node*



*Activity final node*



**Figure 7.13**  
Example of multiple activity final nodes in one activity diagram

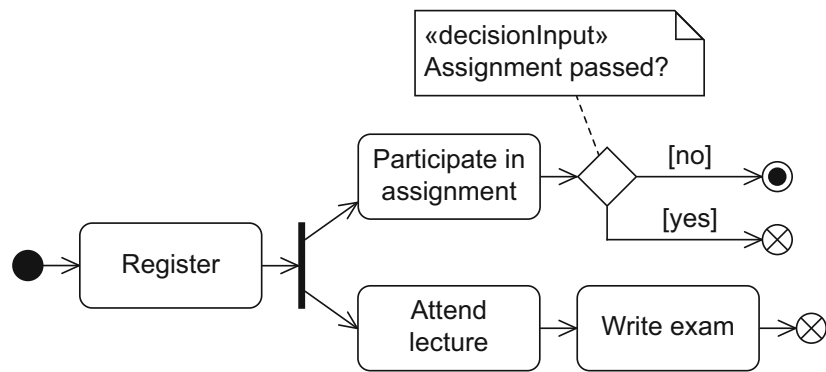
If you only want to end one execution path, leaving the other concurrently active execution paths unaffected, you have to use the *flow final node*. This node only deletes the tokens that flow into it directly, thus ending just the respective path. All other tokens of the activity remain unaffected and may continue to exist. The flow final node is represented by a small circle containing an X and has only incoming edges.

*Flow final node*



**Figure 7.14**

Example of a flow final node and an activity final node



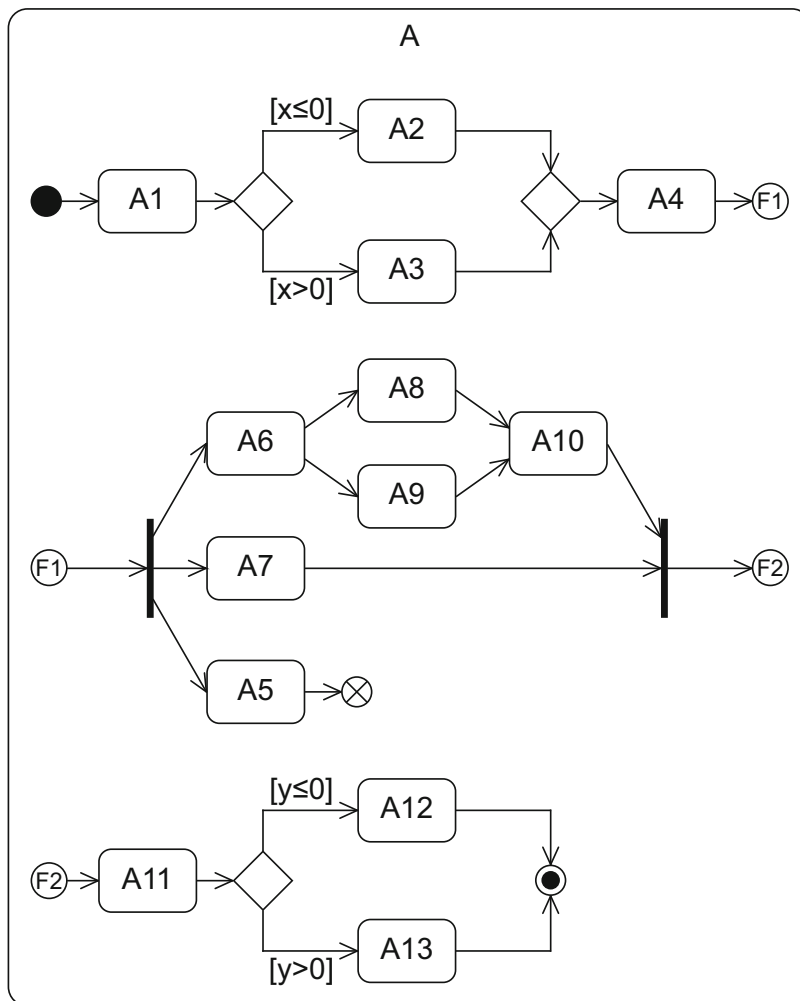
In [Figure 7.14](#), the example from [Figure 7.13](#) has been modified. The successful participation in an assignment and the simultaneous attendance of the related lecture combined with taking the exam are two independent execution paths that each end with a flow final node. However, failing the assignment ends the entire activity, meaning that if a student has a negative assignment grade, the student can no longer take the final exam if the student has not done so already.

Using [Figure 7.15](#), we will now demonstrate the execution semantics of the individual control nodes applying the token concept.

If activity A is activated, all outgoing edges of all initial nodes are assigned a token. Thus, in our example, action A1 receives a token and starts the execution. Once A1 has been successfully completed, it passes the token on to the decision node. Depending on the value of the variable  $x$ , the decision node passes the token to A2 if  $[x \leq 0]$  is true or to A3 if  $[x > 0]$  is true. The subsequent merge node passes on every token it receives to the subsequent node. Thus, after the execution of A2 or A3, the action A4 is activated. The subsequent parallelization node duplicates the token for all outgoing edges, thus creating three tokens and activating A5, A6, and A7. The following three execution paths are taken concurrently:

- One token activates A5. As soon as the execution of A5 has ended, the token passes to the flow final node which then ends this execution path. No other execution path is affected.
- A further token activates A6. After the execution of A6, all outgoing edges from A6 are assigned tokens and thus A8 and A9 are executed concurrently. A10 can only be executed when tokens are present at all incoming edges, that is, when A8 and A9 have been completed. If we look at the token flow, we can see that multiple outgoing edges of an action are equivalent to a parallelization node. If multiple edges lead into an action node, all incoming execution paths have to be completed before the execution of this action. This behavior could also be modeled using synchronization nodes as an alternative.
- The third token activates A7.

When A7 and A10 have both been successfully executed and there is thus a token at both incoming edges of the synchronization node, these tokens are merged into one token and A11 is activated. Depending on the value of the variable  $y$ , the decision node passes the token to A12 if  $[y \leq 0]$  is true or to A13 if  $[y > 0]$  is true. In both cases, after execution of the respective activity, the token enters the activity final node. When the token reaches the activity final node, the entire activity is ended. Any remaining tokens are withdrawn from all actions. Therefore, for example, the execution of A5 is terminated if this action has not yet ended at this point in time.



**Figure 7.15**  
Example of the token  
concept

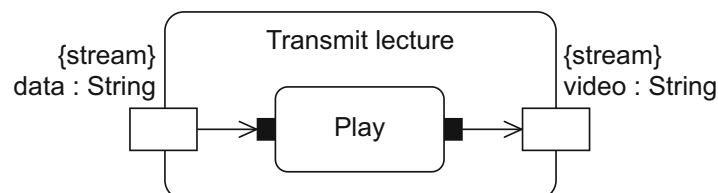
## 7.4 Object Flows

### Control tokens vs data tokens

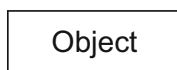
Up to this point, we have looked mainly at the control flow, concentrating on the logical sequence of actions. The described token concept used exclusively *control tokens*. For the purposes of simplification, we have referred to these simply as tokens so far. However, it may be the case, or is even very probable, that actions exchange data using *data tokens*. Just like control tokens, these are never drawn in the diagram and are also used only to describe the execution semantics of activity diagrams. Data tokens are implicitly also control tokens, as the exchange of these tokens influences the flow of the activity. Data can be the result of an action and can also serve as input for a subsequent action. However, data can also be received via the input parameters of the activity and passed on to output parameters, as already described above (see [Fig. 7.1](#) on page 143).

Input parameters are usually only read once at the beginning of the activity and output parameters are written once at the end of the activity. If you want to allow the parameters to be read and written continually during the execution of the activity, you can label the input or output parameter with the keyword `{stream}`. [Figure 7.16](#) shows examples of streams for input and output parameters of activities or actions (parameters for actions are described below). Streaming parameters for actions can be noted by a filled rectangle.

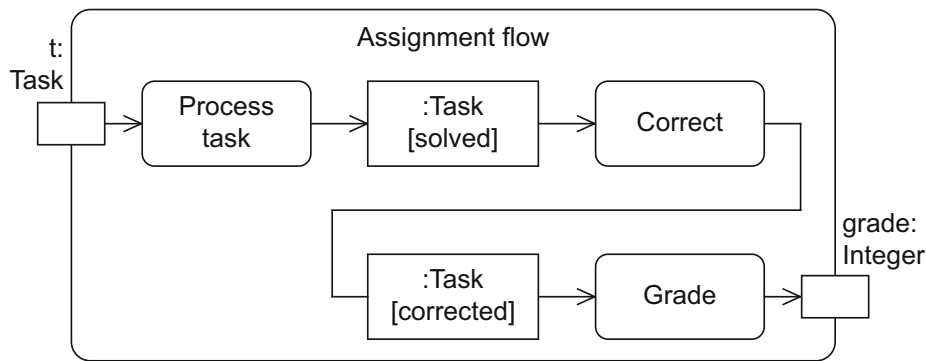
**Figure 7.16**  
Example of streams



### Object nodes



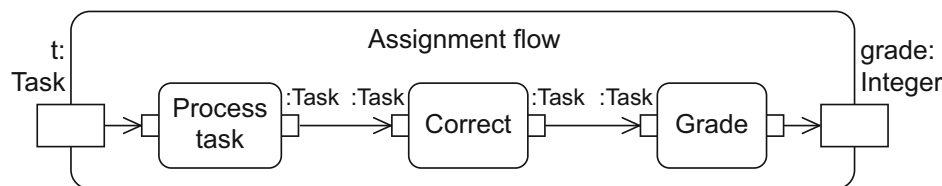
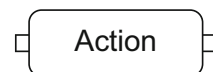
When the activity is ended, any output parameters that have no token are assigned a null token. Within an activity, you can use *object nodes* to explicitly represent the exchange of data. Object nodes can be depicted in various ways. They are shown either as a separate node as in the object diagram (see page 49) or they are attached to an action directly as input or output pins. [Figure 7.17](#) gives an example of an object node as an independent node. It is added as a rectangle between the action that delivers the data and the action that consumes the data. The rectangle contains the name of the object that it represents. You can optionally specify an object type as well. Within square brackets, you can also stipulate the state that the object must be in.



**Figure 7.17**  
Example of an object node

The *pin notation* for actions corresponds to the notation of parameters of an activity. It is used in the same way to represent objects that serve as input and output for actions. A small rectangle is specified at the beginning or end of the edge at the boundary of the corresponding action. The pin can be annotated with the same information that we use when explicitly representing object nodes as rectangles. Figure 7.18 shows an example.

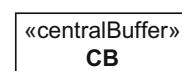
*Pins*



**Figure 7.18**  
Example of an object node in pin notation

The activity diagram offers special object nodes for saving and passing on data tokens: the central buffer and the data store. The *central buffer* is a special object node that manages the data flow between multiple sources and multiple receivers. It accepts incoming data tokens from object nodes and passes these on to other object nodes. In contrast to pins and activity parameters, a central buffer is not bound to actions or activities. When a data token is read from the central buffer, it is deleted there and cannot be consumed again. Figure 7.19 shows an example of the use of a central buffer. To execute the action Grant access authorization, a key must be withdrawn from the KeyCabinet. The key is then no longer in the KeyCabinet until it is returned in the action Withdraw access authorization.

*Central buffer*



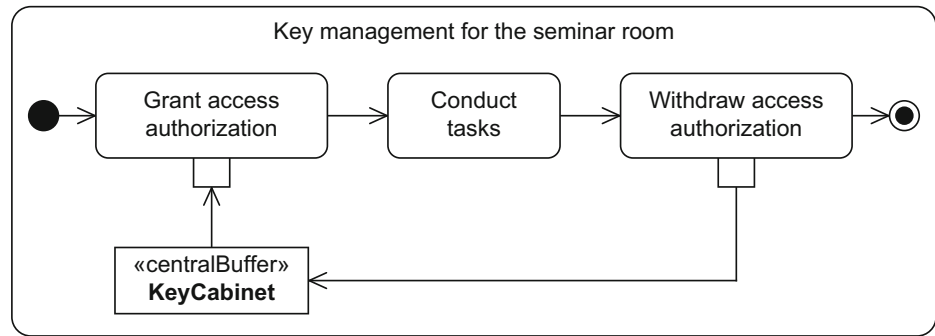
In a *data store*, all data tokens that flow into the data store are saved permanently, meaning that they are copied before they leave the data store again. You can define queries regarding the content of the data store at the outgoing edges leading from the data store. These queries are attached to the outgoing edge using the note symbol. A data store can therefore model the functionality of a database. Figure 7.20 shows

*Data store*



**Figure 7.19**

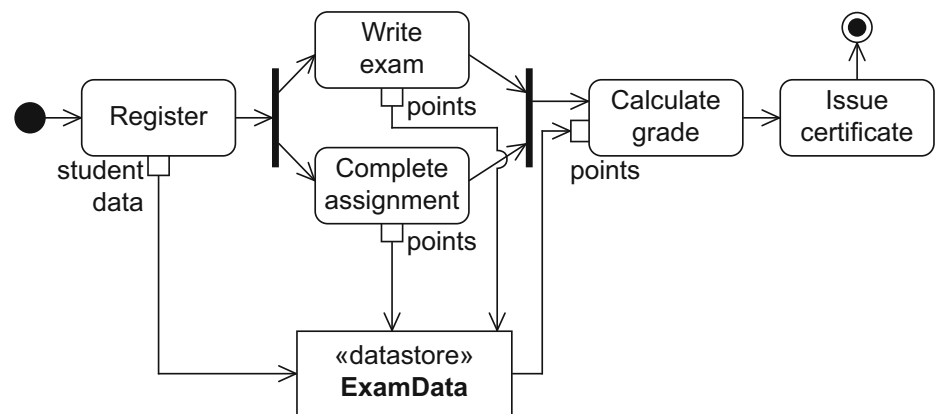
Example of a central buffer



an example of the use of a data store. The performance of the participants of a course is managed in the data store ExamData. This data includes the assessment of assignment tasks and exam results necessary for calculating the overall grade.

**Figure 7.20**

Example of a data store



A central buffer represents transient memory, whereas a data store represents permanent memory. With the former, the information can only be used once, meaning that once it has been read from the central buffer and forwarded it is lost. With a data store, the information can be used as often as required, provided it has been saved once in the data store.

## 7.5 Partitions

### Partition

A	B

A *partition* allows you to group nodes and edges of an activity based on common properties. If we consider a business process, for example, we could use a partition to group all actions that a specific entity is responsible for executing. UML has no strict rules regarding the grouping criteria that can be used. Generally, partitions reflect organizational units

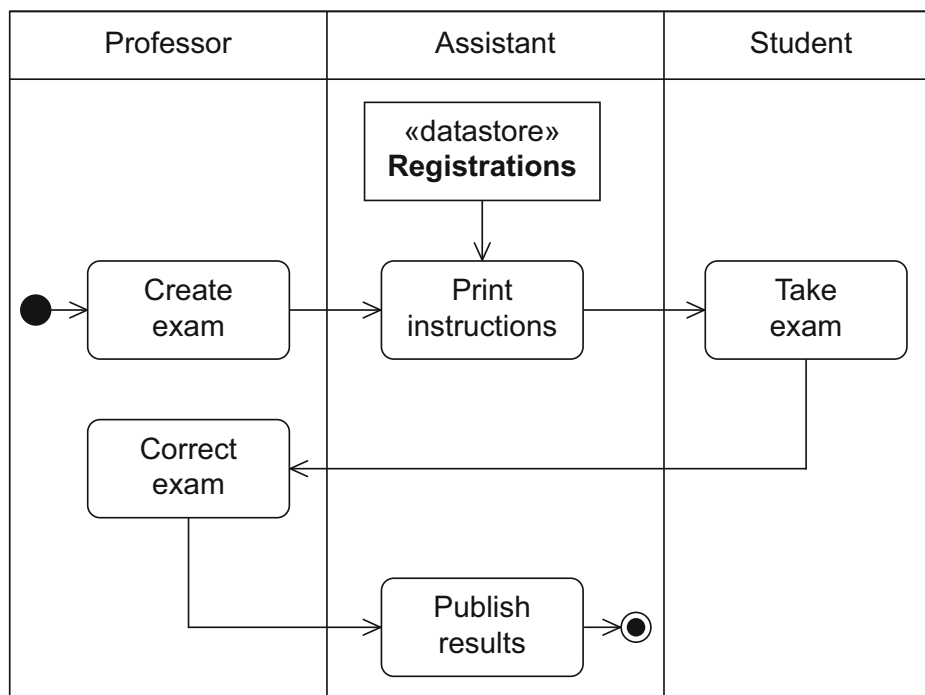


or roles that are responsible for the execution of the actions within the partitions. You can specify partitions at different levels of detail down to the level of individual classes. Partitions may overlap and be nested in any way required. They do not change the meaning of the execution semantics of an activity diagram, which is defined by tokens. This means that partitions do not influence the token flow but merely represent a logical view of its components. Partitions make the diagram clearer, enabling you to see the areas of responsibility quickly, thus introducing more detailed information into the model.

Partitions can be depicted either graphically or in textual form. When depicted in graphic form, they are placed on top of the activity diagram as “open” rectangles. All elements that lie within an “open” rectangle belong to a common group. The name of the partition is specified at one end of the rectangle. Due to their appearance, partitions are also referred to as *swimlanes*. Figure 7.21 shows an example of the use of partitions in an activity diagram that models the execution of an exam. The parties involved are a student, an assistant, and a professor. The use of partitions allows each of these actors to be assigned the actions that they have to perform.

Synonyms:

- Partition
- Swimlane

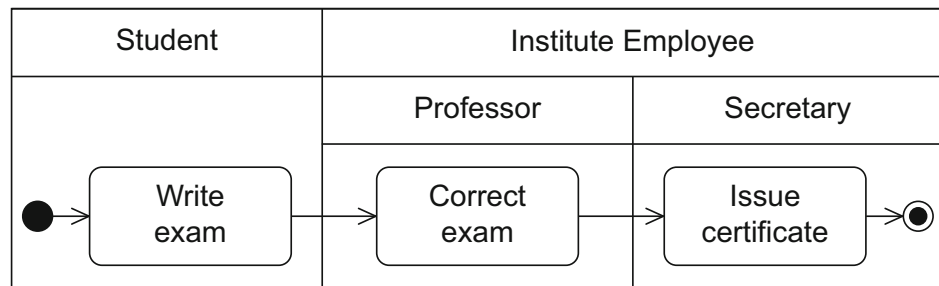


**Figure 7.21**  
Example of one-dimensional partitions

A partition can itself be subdivided into multiple *subpartitions*. Figure 7.22 shows an example of this. In this example, the institute employees Professor and Secretary are involved in the execution of an exam.

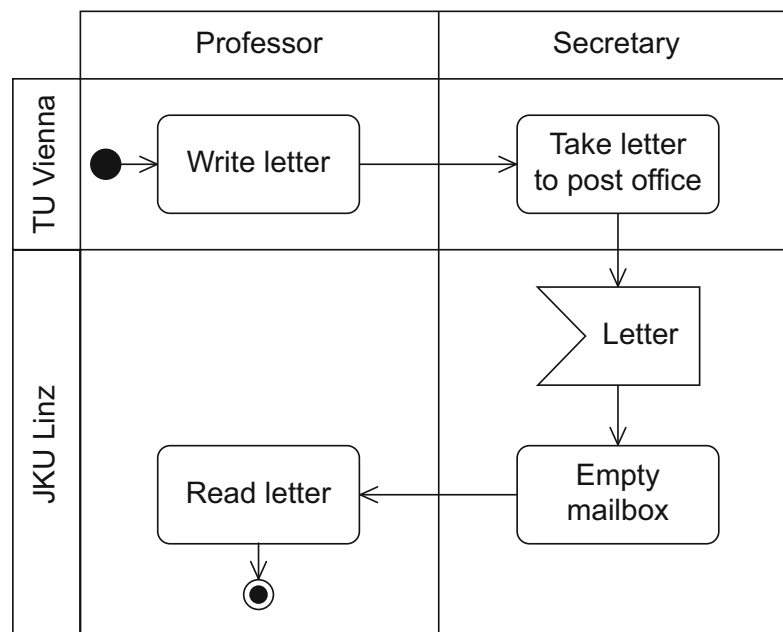
*Subpartition*

**Figure 7.22**  
Example of subpartitions

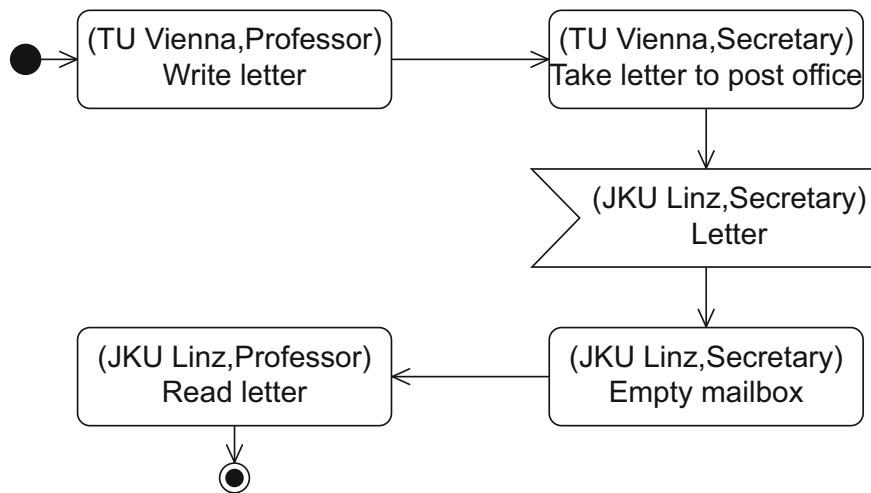


As we can see in the example in [Figure 7.23](#), partitions can also be multi-dimensional. In this example, we model correspondence between a professor of the Johannes Kepler University Linz (JKU Linz) and the Vienna University of Technology (TU Vienna). The Professor at TU Vienna writes a letter to the Professor at JKU Linz. The professor gives the letter to the Secretary, who takes the letter to the post office. The Secretary at JKU Linz fetches the letter from the mailbox as soon as the letter arrives and has it delivered to the Professor at JKU Linz, who then reads it. This shows how we need multi-dimensional partitions when various groups of actors can appear in various forms.

**Figure 7.23**  
Example of multi-dimensional partitions



You can also assign an action to a partition or—in the case of multi-dimensional partitions—to a set of partitions in text form. In this situation, you specify the partitions in parentheses above the action name. If the action belongs to multiple partitions, these partitions are listed separated by a comma, for example (Partition 1, Partition 2). When specifying



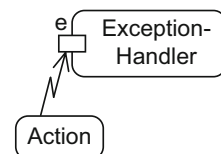
**Figure 7.24**  
Example of multi-dimensional partitions with textual notation

subpartitions, you use a double colon—(Partition::Subpartition)—to express a hierarchical partitioning. The activity diagram from [Figure 7.24](#) shows the example from [Figure 7.23](#) in this notation.

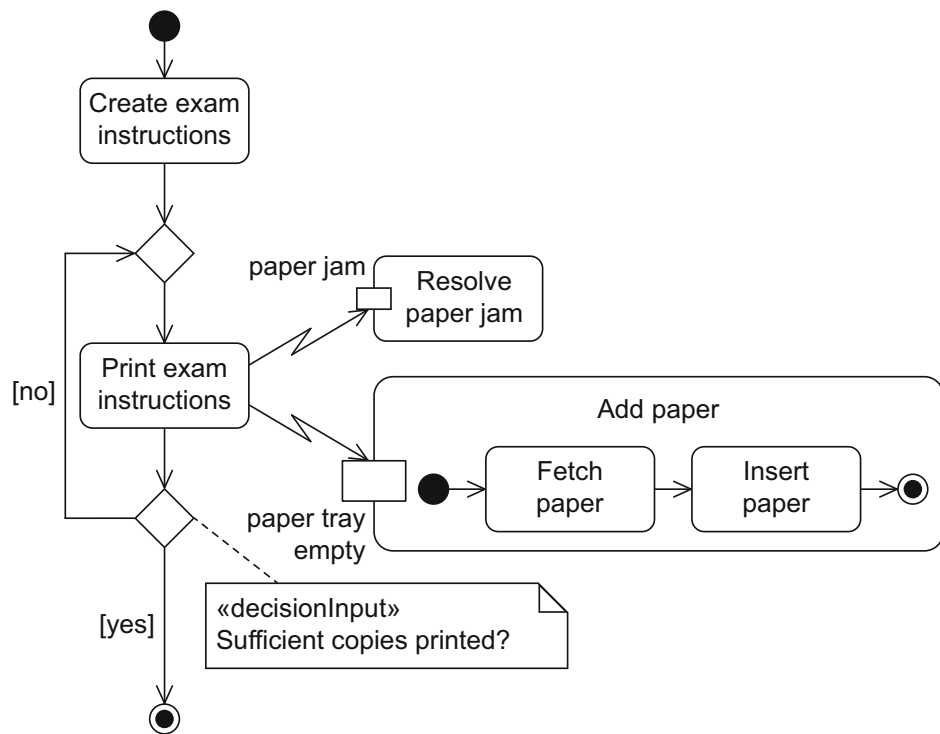
## 7.6 Exception Handling

If an error occurs during the execution of an action, the execution is terminated. In this situation, there is no guarantee that the action will deliver the expected output. If an action has an *exception handler* for a specific error situation, this exception handler is activated when an exception occurs. Using an exception handler, you can define how the system is to react in a specific error situation *e*. This allows you to minimize the effects of this error. You specify an exception handler for a specific type of error—that is, you can use different exception handlers for different errors. If an error situation occurs, all tokens in the action concerned are deleted immediately. If there is a matching exception handler, this replaces the content of the action concerned and instead, the content of the exception handler is executed. The sequence then continues as the regular path of the activity as if the defective action had ended normally. As an exception handler is an activity node, it is depicted as a rectangle with rounded corners. The action safeguarded by the exception handler points to the exception handler with a lightning bolt arrow. The tip of the arrow is labeled with the type of the error. [Figure 7.25](#) shows two examples for handling exceptions. If a paper jam occurs during printing, printing can continue once the paper jam has been removed. If there is no paper in the printer, paper must be inserted for printing to continue until sufficient copies of the exam instructions have been printed.

*Exception handler*



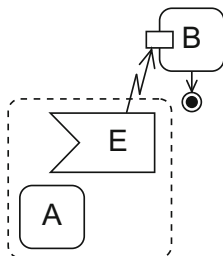
**Figure 7.25**  
Examples of exception  
handling



If there are multiple matching exception handlers, the handler to be executed is not specified. If there is no matching exception handler, the exception is forwarded to the surrounding structure. If the exception is passed to the outermost activity without a matching exception handler being found, the behavior of the system is undefined.

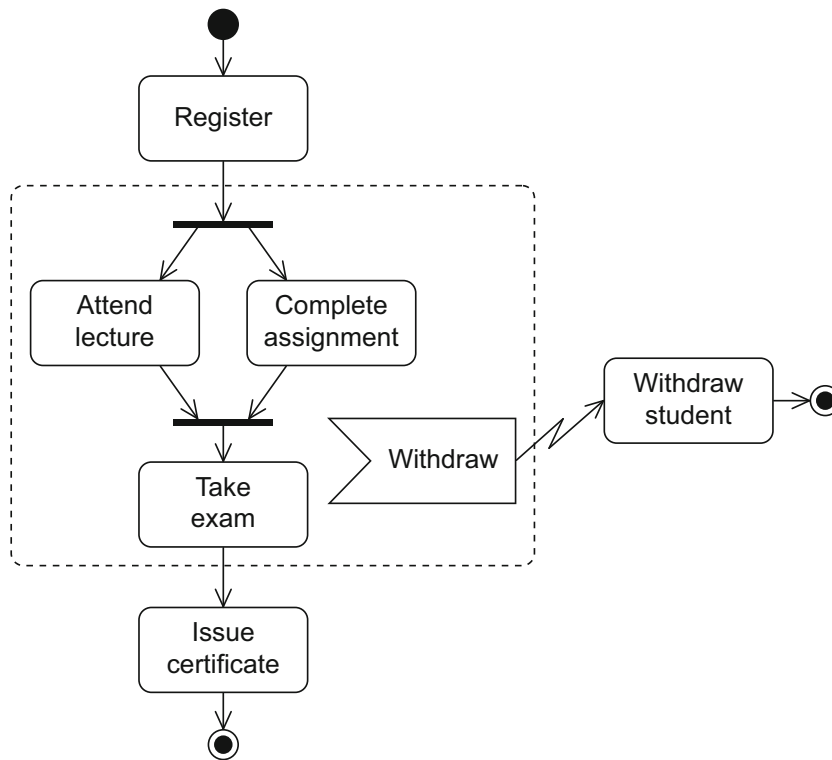
An exception handler has no explicit incoming or outgoing edges. It has the same access rights to variables and values as the nodes that it safeguards. The tokens that result from the execution of the content of the exception handler become result tokens of the safeguarded node. Therefore, the exception handler and the safeguarded node must have the same number of return values. This ensures that in the event of an error, every outgoing edge of the safeguarded node receives the required token.

*Interruptible activity  
region*



The *interruptible activity region* offers a further way to handle exceptions. Using this concept, you can define a group of actions whose execution is to be terminated immediately if a specific event occurs. The interruptible activity region is depicted as a dashed rectangle with rounded corners that encloses the relevant actions. The execution of these events is monitored for the occurrence of a specific event, for example an error. If the event does occur during this execution, then as a consequence certain behavior is executed. Within the interruptible activity region, you model an accept event action that represents the special event and leads out from the edge in lightning bolt form to an activity outside the inter-

interruptible activity region. If the modeled event occurs, all control tokens in the interruptible activity region are deleted and the action that the accept event action points to is activated. Figure 7.26 shows an example of an interruptible activity region. If a student withdraws from the university while attending a course, the action Withdraw student is executed. However, a withdrawal is only possible if the student has previously registered and if the action Take exam has not yet ended. In all other cases a certificate is issued.



**Figure 7.26**  
Example of an interruptible  
activity region

## 7.7 Concluding Example

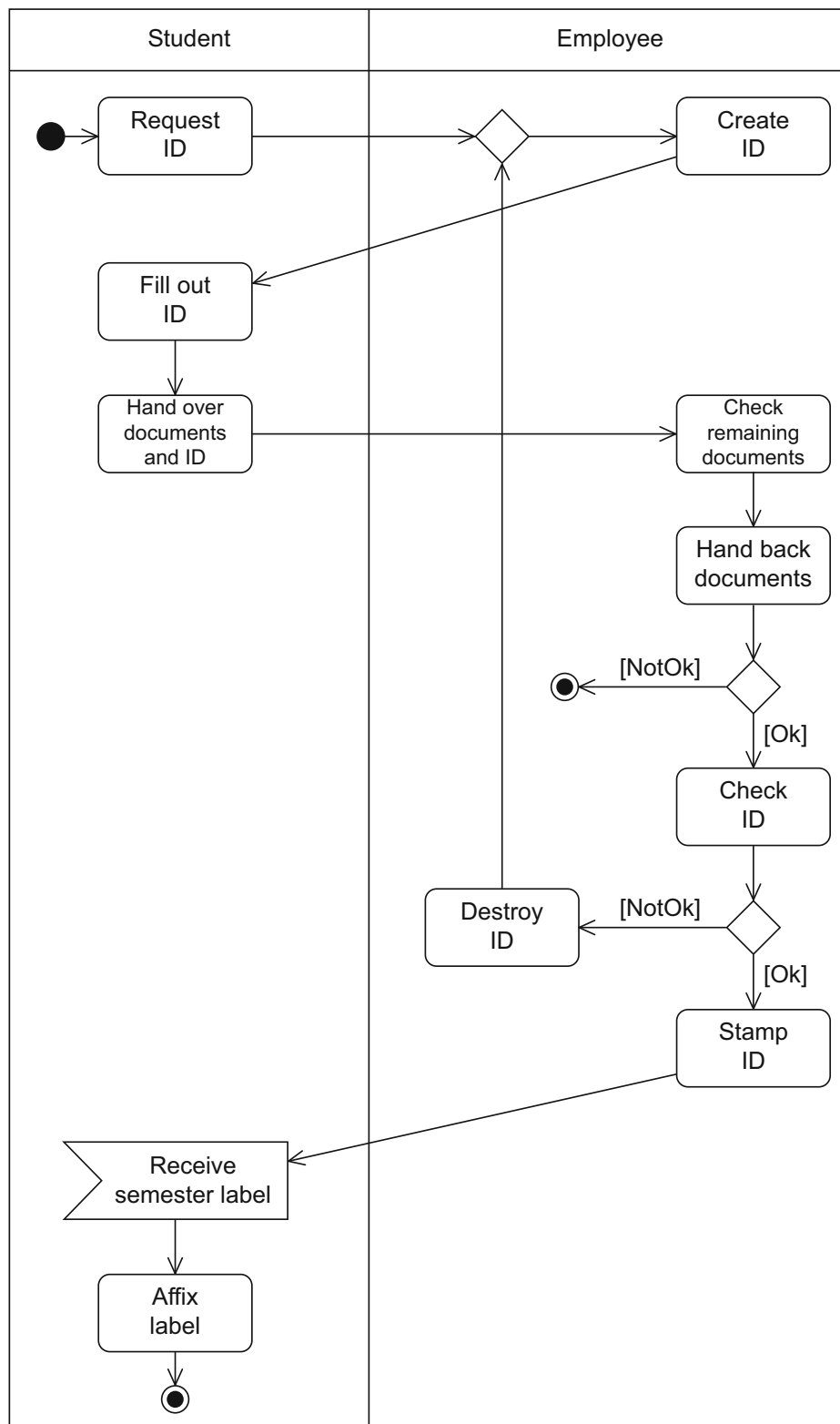
To sum up, we look at the process that has to be executed for a future student of a university to receive a student identification card (student ID). We use an activity diagram to model this process. In textual form, the process of issuing a student ID can be described as follows: To obtain a student ID, the student must request this ID from an employee of the student office. The employee hands the student the forms that the student has to fill out to register at the university. These forms include the student ID itself, which is a small, old-style cardboard card. The

student has to enter personal data on this card and the employee confirms it with a stamp after checking it against certain documents. The student ID is only valid if it has a current semester label. Once the student has filled out the forms, the student returns them to the employee in the student office and hands over documents such as photo identification, school-leaving certificate, and birth certificate. The employee checks the documents. If the documents are incomplete or the student is not authorized to receive a student ID for the university, the process is terminated immediately. If the documents are all in order, the employee checks whether the student has filled out the student ID correctly. If there are any errors, this ID is destroyed and the student has to fill out another one. Otherwise the ID is stamped. However, the student ID is not valid until it bears the semester label sent to the student by post.

Two actors are involved in the process to be modeled: the Student and the Employee. To assign the individual actions precisely, we use partitions. We can derive the actions and the control flow directly from the text above and these are shown in [Figure 7.27](#). To model the termination of the process in the event of invalid or incomplete documents, we use a decision node where one path leads to an activity final node. The requirement that part of the entire process has to be repeated if the forms are filled out incorrectly results in the use of a loop. We implement this with a decision node after the action Check ID and a merge node before the action Create ID. If we were to allow the edge to lead directly to the node of the action Create ID, we would need two tokens for the execution of this action. As this will never happen, it is important to use a merge node. If the student has handed over the documents completely and filled out the forms correctly, the student ID is stamped and the student receives the current semester label by post. We model this action as an accept event action. To validate the ID, the student must then affix the label. This action ends the process that has to be executed to obtain a new student ID.

In the activity diagram in [Figure 7.27](#), we have modeled only the control flow. However, in this example, an object is changed: the student ID. Initially it is blank, then filled out, then stamped. A student ID is not valid until the semester label has been affixed. The changes to the student ID are shown in [Figure 7.28](#), which expands [Figure 7.27](#) to include the object flow of the object Student ID. This highlights which actions need and process the object Student ID.

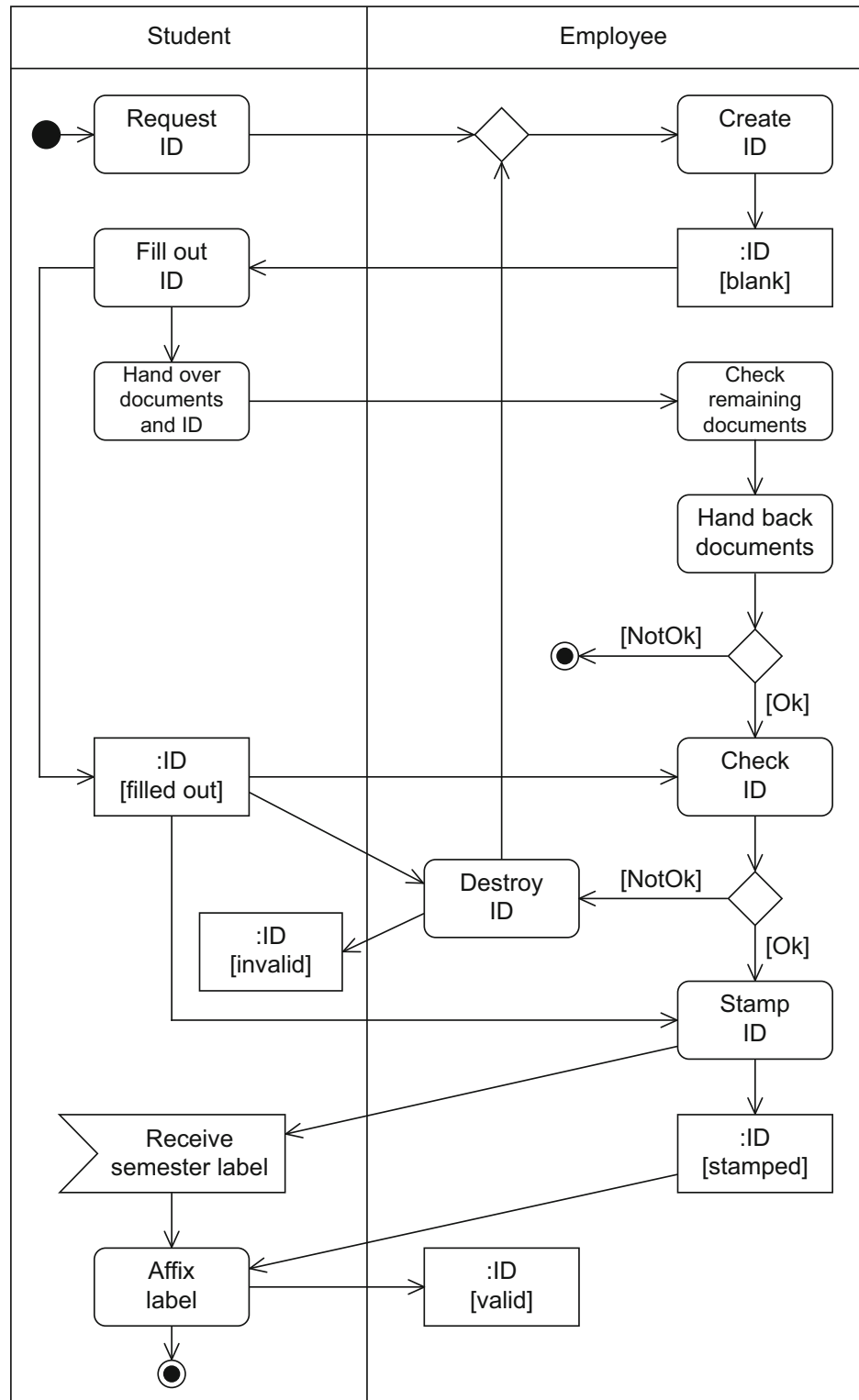
The most important elements of the activity diagram are summarized in [Tables 7.1](#) and [7.2](#).








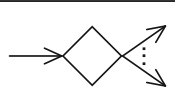
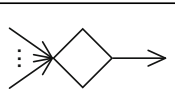
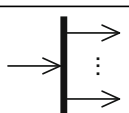
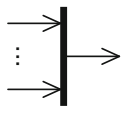
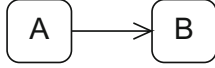
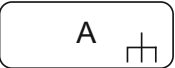
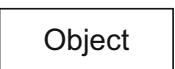
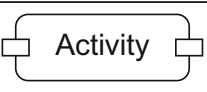
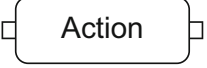
**Figure 7.27**  
Activity diagram “Issue student ID”

**Figure 7.28**

Activity diagram “Issue student ID” with control and object flow





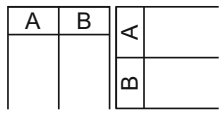

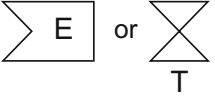
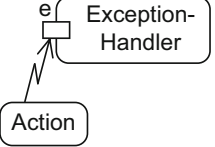
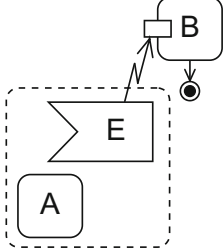
<i>Name</i>	<i>Notation</i>	<i>Description</i>
Action node		Actions are atomic, i.e., they cannot be broken down further
Activity node		Activities can be broken down further
Initial node		Start of the execution of an activity
Activity final node		End of ALL execution paths of an activity
Flow final node		End of ONE execution path of an activity
Decision node		Splitting of one execution path into two or more alternative execution paths
Merge node		Merging of two or more alternative execution paths into one execution path
Parallelization node		Splitting of one execution path into two or more concurrent execution paths
Synchronization node		Merging of two or more concurrent execution paths into one execution path
Edge		Connection between the nodes of an activity
Call behavior action		Action A refers to an activity of the same name
Object node		Contains data and objects that are created, changed, and read
Parameters for activities		Contain data and objects as input and output parameters
Parameters for actions (pins)		Contain data and objects as input and output parameters

**Table 7.1**

Notation elements for the activity diagram

**Table 7.2**

Notation elements for the activity diagram, part 2

<i>Name</i>	<i>Notation</i>	<i>Description</i>
Partition		Grouping of nodes and edges within an activity
Send signal action		Transmission of a signal to a receiver
Asynchronous accept (time) event action		Wait for an event E or a time event T
Exception handler		Exception handler is executed instead of the action in the event of an error e
Interruptible activity region		Flow continues on a different path if event E is detected