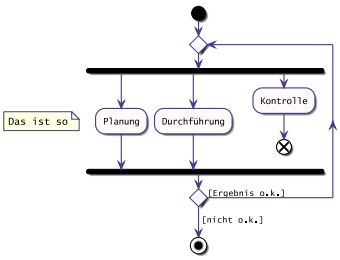
UML Aktivitätsdiagramm

und die Modellierung mit plantUML



1 Diagramm zur Verhaltensmodellierung

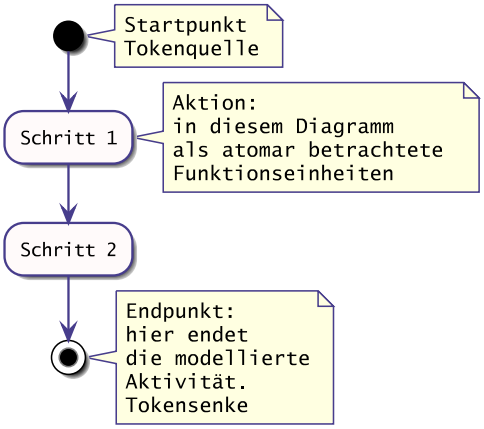
Ein UML-Aktivitätsdiagramm (*Activitydiagram*) beschreibt, wie das Verhalten eines (Software)-System *realisiert* ist. Es erweitert die Ausdrucksmöglichkeiten gegenüber einfacheren Flussdiagrammen wie dem Programmablaufplan. Im Aktivitätsdiagramm können Daten- und Kontrollflüsse modelliert werden, z.B. von

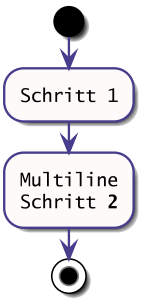
- Algorithmen,
- Geschäftsprozessen oder
- Workflows.

Im Fokus steht die Modellierung der Abfolge (sequenziell oder parallel), Bedingungen, Verzweigungen, Wiederholungen sowie Anfang und Ende von Aktivitäten. Alle Schritte können zudem in Verantwortungsbereichen konkreten Akteurinnen oder Systemen zugewiesen werden.

2 Das Tokenmodell im Flussdiagramm: mit Knoten und Kanten

Aktivitätsdiagramme bestehen aus **Knoten** und **Kanten**. Sie nutzen das Tokenmodell: ein *Token* ist eine gedachte Markierung (*Token*) kennzeichnet die momentan ausgeführten Aktionen - vergleichbar mit dem springenden Punkt beim Karaokesingen. Ein Token wird im Startpunkt (*initial node*) erzeugt und durchläuft das Diagramm entlang der Kanten (Pfeile, *edges*). Im Endpunkt (*activity final node*) wird der Token konsumiert und die Aktivität so beendet.
Ein minimales mit PlantUML modelliertes Aktivitätsdiagramm, das einen unverzweigte Aktivität darstellt, sieht etwa so aus:



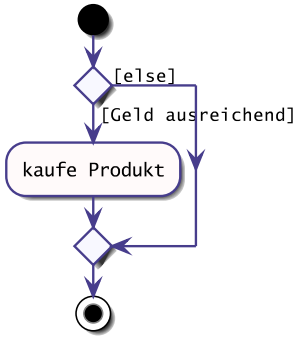



```
@startuml 'Muss immer am Anfang stehen
start
:Schritt 1;
:Multiline
   Schritt **2**;
stop
@enduml
```

3 Kontrollstrukturen: Verzweigung und Vereinigung

Neben dem rein sequenziellen unverzweigten Ablauf können auch bedingte Anweisungen modelliert werden. Verzweigungen werden als Entscheidungsknoten (*decision node*) mit dem Raute-Symbol notiert. Die Bedingungen selbst werden als *guard* bezeichnet und -abweichend vom Programmablaufplan- in eckigen Klammern an der jeweiligen Kante notiert. Die Bedingungen müssen als Prädikate formuliert sein - also mit *true* oder *false* auswertbar sein und sollten *disjunkt* sein (sich gegenseitig ausschließen), um ein vorhersagbares Verhalten zu modellieren.

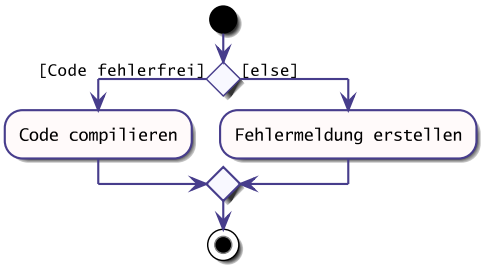
Daher sollte in den meisten Fällen eine `[else]`-Kante modelliert werden.



Bei PlantUML ist die Notation etwas gewöhnungsbedürftig: die erste Bedingung wird hinter `then()` in Klammern notiert, den `else`-Block sollte man in jedem Fall notieren:

```
8  if() then ([Geld ausreichend])
9     :kaufe Produkt;
10 else ([else])
11 endif
```

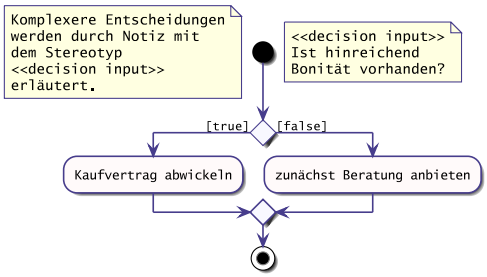
Im Gegensatz zum Programmablaufplan werden die verschiedenen Kanten einer Verzweigung auch wieder an einer Raute (*merge node*) zusammengeführt. Weder ein *decision node* noch ein *merge node* ändern die Anzahl der vorhandenen Token.




```
12 start
13 if() then ([Code fehlerfrei])
14    :Code compilieren;
15 else ([else])
16    :Fehlermeldung erstellen;
17 endif
18 stop
```

Nicht alle Bedingungen lassen sich unmittelbar als Prädikat formulieren. Sofern nähere Eräuterungen nötig sind sieht die UML vor, dass diese über eine Notiz mit dem Stereotyp `<<decision input>>` erfolgt, die mit der *decision node* verbunden wird.

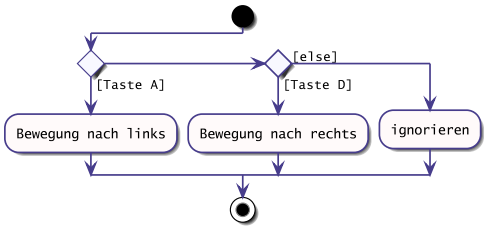
In PlantUML gibt es diese Möglichkeit nicht, daher muss man sich mit einer Notiz, die neben der vorigen Aktion steht, behelfen. Gemäß UML wird innerhalb der *decision node* nichts notiert - hier weicht die PlantUML-Dokumentation vom UML-Standard ab.




```
19 note right
20    <<decision input>>
21    Ist hinreichend
22    Bonität vorhanden?
23 end note
24 if() then ([true])
25    :Kaufvertrag abwickeln;
26 else ([false])
27    :zunächst Beratung anbieten;
28 endif
```

Mehrfache Verzweigungen können durch mehr Kanten realisiert werden, die den *decision node* verlassen. Wichtig ist auch hier, dass die *guards* disjunkt sind.

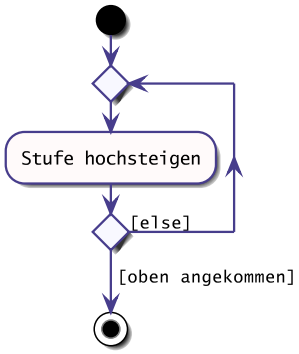
Leider bietet PlantUML nur die Möglichkeit, zwei ausgehende Kanten an einer *decision node* zu nutzen. Daher müssen multiple Bedingungen als verschachtelte If-Statements modelliert werden:




```
29 if() then ([Taste A])
30    :Bewegung nach links;
31 elseif() then ([Taste D])
32    :Bewegung nach rechts;
33 else ([else])
34    :ignorieren;
35 endif
```

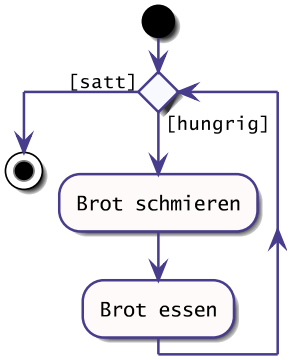
4 Wiederholungsstrukturen: kopf- und fussgesteuerte Schleifen

Eine nachgelagerte Bedingung, die eine Wiederholung bestimmter Aktionen erzwingt wird über die Rückführung der ausgehenden Kante mit *guard* einer *decision node* modelliert. Bei diesen fussgesteuerten Schleifen werden die Aktionen innerhalb der Schleife in jedem Fall einmal ausgeführt. Mit PlantUML erfolgt die Modellierung mit einem `repeat / repeat while () is ([` *guard* `])`-Block:




```
36 start
37 repeat
38    :Stufe hochsteigen;
39 repeat while () is ([else])
40 ->[oben angekommen];
41 stop
```

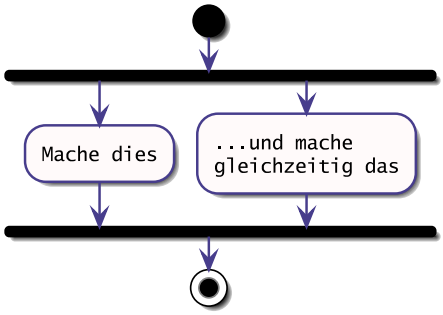
Im Fall einer kopfgesteuerten Schleife wird die Bedingung zunächst geprüft, die zu wiederholenden Aktionen also ggf. nie ausgeführt.




```
44 start
45 while() is ([hungrig])
46    :Brot schmieren;
47    :Brot essen;
48 endwhile ([satt])
49 stop
```

5 Concurrency: Parallelisierung von Aktionen (Splitting und Synchronisation)

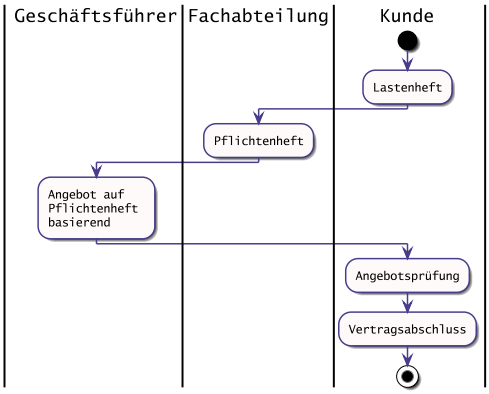
Im Gegensatz zu einem *decision node* oder *merge node* müssen bei Gabelungen (*fork node*) oder Synchronisierung (*join node*) an allen eingehenden Kanten ein Token anliegen, damit sie wiederum Token weiterreichen. Entsprechend werden an allen ausgehenden Kanten dann Token weitergereicht. Auf diese Art werden parallele Prozesse und Synchronisierungen modelliert.




```
start
fork
   :Mache dies;
fork again
   :...und mache
   gleichzeitig das;
end fork
stop
```

6 Partitionen / Swimlanes zur Unterscheidung von Verantwortungsbereichen

Die UML sieht vor, dass modelliert werden kann, welche Akteurin oder welches System für bestimmte Aktionen verantwortlich oder organisatorisch Zuständig ist. Aufgrund des Aussehens werden diese Verantwortungsbereiche, die die UML *partition* nennt, oft *swimlanes* genannt.

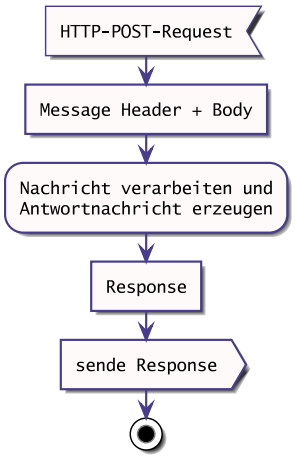



```
'Durch Nennung Reihenfolge festlegen
|Geschäftsführer|
|Fachabteilung|
|Kunde|
   start
|Kunde|
   :Lastenheft;
|Fachabteilung|
   :Pflichtenheft;
|Geschäftsführer|
   :Angebot auf
   Pflichtenheft
   basierend;
|Kunde|
   :Angebotsprüfung;
   :Vertragsabschluss;
   stop
```

7 Signale/ Ereignisse senden und empfangen; Objektflüsse darstellen

Token können nicht nur aus einem Startknoten entspringen, sondern auch über Signale erzeugt werden, die in einer *accept event action* empfangen werden. Sofern dieser Signalempfängerknoten auch eingehende Kanten hat, wird der ausgehende Token erst gefeuert, wenn ein Token anliegt und ein Signal eingeht. Der Token wandert dann wie gewohnt entlang der Kanten von Knoten zu Knoten. Signale können auch gesendet werden. Beim erreichen einer *send signal action* wird asynchron das Signal gesendet und die nächste Aktion bearbeitet. Es wird nicht auf Antwort oder Empfangsbestätigung gewartet.

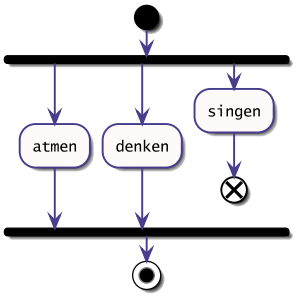
Sofern nicht der Kontrollfluss symbolisiert werden soll, sondern konkrete Daten, so wird ein Objektknoten als Rechteck notiert. An den ein- und ausgehenden Kanten wird anstelle eines abstrakten Tokens dann ein Objekt transportiert.




```
78
79 :HTTP-POST-Request<
80 :Message Header + Body]
81 :Nachricht verarbeiten und
82    Antwortnachricht erzeugen;
83 :Response]
84 :sende Response>
85 stop
```

8 Ablaufende

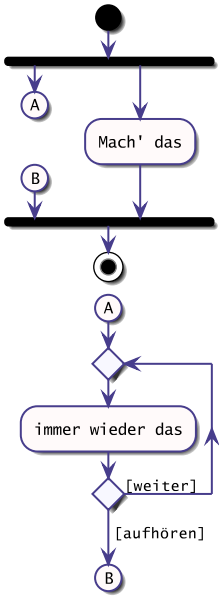
Wenn das Erreichen eines Endes zwar den aktuellen Ausführungsstrang beendet - also den eingetroffenen Token konsumiert - aber in der Gesamtaktivität noch weitere Token vorhanden sein können, muss ein Ablaufende (*activity final node*) modelliert werden. Im Gegensatz zu einem *activity final node* werden nicht alle vorhandenen Token konsumiert, sondern nur der am *activity final node* eintreffende Token.




```
86 start
87 fork
88    :atmen;
89 fork again
90    :denken;
91 fork again
92    :singen;
93    end
94 end fork
95 stop
```

9 Konnektoren (Sprungmarken) zur übersichtlicheren Darstellung

Um Kreuzungen zu vermeiden oder um komplexere Zusammenhänge übersichtlicher darstellen zu können, können Sprungmarken (A) verwendet werden. Abläufe können dadurch unterbrochen werden (bei PlantUML mit `detach`) und in gesonderten Aktivitätsdiagrammen ausgeführt werden.

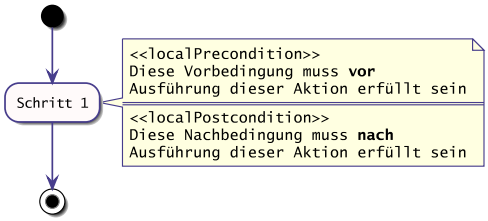



```
96  start
97  fork
98     (A)
99     detach
100    (B)
101 fork again
102    :Mach' das;
103 end fork
104 stop
105
106
107 (A)
108 repeat
109    :immer wieder das;
110 repeat while () is ([weiter])
111 ->[aufhören];
112 (B)
```

10 Vor- und Nachbedingungen

Aktionen können mit Vor- und Nachbedingungen versehen werden. Diese werden jeweils als Notiz mit dem jeweiligen Stereotyp notiert:

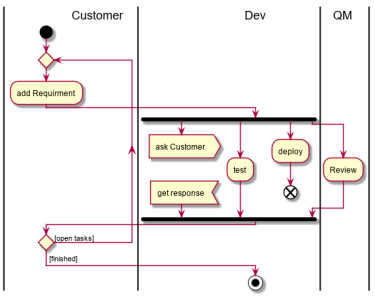
- «localPrecondition»: Diese Bedingungen müssen vor Eintritt in die Aktion erfüllt sein
- «localPostcondition» : Diese Bedingungen müssen vor Beendigung der Aktion erfüllt sein



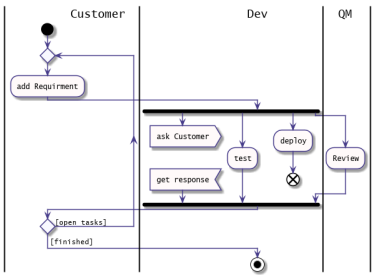

```
start
:Schritt 1;
note right
   <<localPrecondition>>
   Diese Vorbedingung muss <b>vor</b>
   Ausführung dieser Aktion erfüllt sein
   ====
   <<localPostcondition>>
   Diese Nachbedingung muss <b>nach</b>
   Ausführung dieser Aktion erfüllt sein
end note
stop
```

11 plantUML-Formatierung: Aufhübschen von Aktivitätsdiagrammen

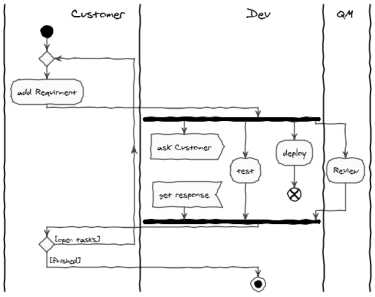
Wenn die Diagramme erstmal stehen will man sie aufhübschen. Dafür stehen allerlei möglichkeiten zur Verfügung, die v.a. auf der plantUML-Seite dargestellt werden. Einige Beispiele sind hier abgebildet:



Minimalbeispiel



Anpassung von Schriftart und Farben



Sieht nach Entwurf aus: um die Vorläufigkeit und Änderbarkeit zu unterstreichen kann man es nach einer Skizze aussehen lassen.

```
@startuml
|Customer|
|Dev|
|QM|
|Customer|
start
repeat
:add Requirment;
|Dev|
fork
:ask Customer>
Detach
:get response<
fork again
:test;
fork again
:deploy;
end
fork again
|QM|
:Review;
end fork
|Customer|
repeat while () is ([open
    tasks])
->[finished];
|Dev|
stop
@enduml
```

```
@startuml
skinparam DefaultFontName
    "Lucida Sans Typewriter"

skinparam Activity{
BackgroundColor snow
BorderColor DarkSlateBlue
DiamondBackgroundColor
    ghostwhite
DiamondBorderColor
    DarkSlateBlue

}
skinparam Note{
BorderColor DarkSlateBlue
BackgroundColor LightYellow
}

skinparam ArrowColor
    DarkSlateBlue
|Customer|
|Dev|
|QM|
|Customer|
start
...
stop
@enduml
```

```
@startuml
' Welche Schriften gibt es
    auf dem System?
' listfonts als
    plantUML-Kommando gibt's
    aus.
skinparam DefaultFontName
    "FG Virgil"
skinparam handwritten true
skinparam monochrome true
skinparam packageStyle rect
skinparam shadowing false


|Customer|
|Dev|
|QM|
|Customer|
start
...
stop
@enduml
```

References

[plantUML] Projektwebsite., Dokumentation
https://www.plantuml.com/

[plantText] Projektwebsite., Website, auf der direkt plantUML-QUelltexte geparst werden können:
https://www.planttext.com/