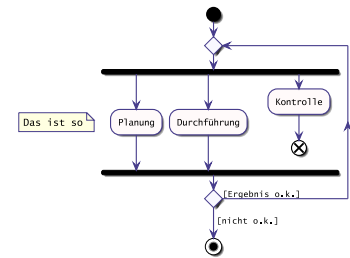


# UML Aktivitätsdiagramm

und die Modellierung mit plantUML



## 1 Diagramm zur Verhaltensmodellierung

Ein UML-Aktivitätsdiagramm (*Activitydiagramm*) 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

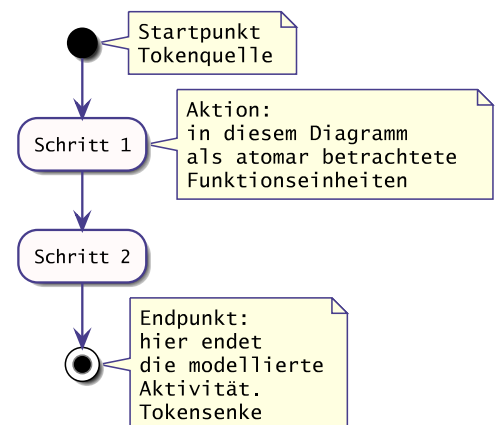
- Algorithmen,
- Geschäftsprozessen oder
- Workflows.

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

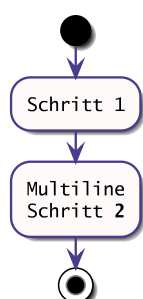
Aktivitätsdiagramme bestehen aus **Knoten** und **Kanten**. Sie nutzen das Tokenmodell: ein *Token* ist eine gedachte Markierung (*Token*) kennzeichnet die momentan ausgeführten Aktionen - vergleichbar mit dem springenden Punkt beim Karaokesingen. Ein Token wird im Startpunkt (*initial node*) erzeugt 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, ist rechts dargestellt.



Hinweis: Den meisten Diagrammen in diesem Dokument sind Links zu den zugrundeliegenden Vektorgrafiken und Quelltexten hinterlegt. Der die komplette URL öffnet das Diagramm, die zweite Hälfte (nach src=) den Quelltext.

<https://www.plantuml.com/plantuml/proxy?fmt=svg&src=https://www.quelltext.url/source.plantuml>



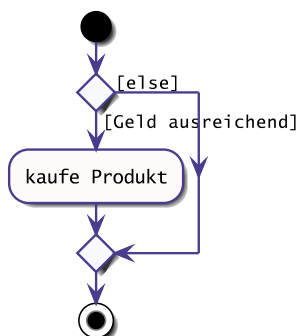
```
1 @startuml 'Muss immer am Anfang stehen
2 start
3 :Schritt 1;
4 :Multiline
5     Schritt **2**;
6 stop
7 @enduml
```



### 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 Prädikat formuliert sein - also mit *true* oder *false* auswertbar sein. Sie sollten *disjunkt* sein (sich gegenseitig ausschließen), um ein vorhersagbares Verhalten zu modellieren. Bei einfachen Verzweigungen erreicht man das, in dem zu jeder Bedingung ein *[else]*-Zweig modelliert wird.

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:

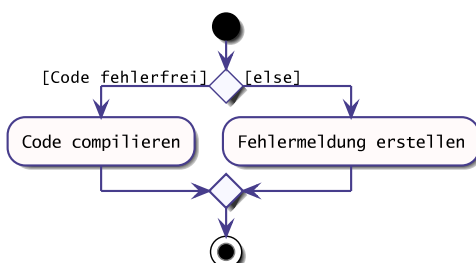


```

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.



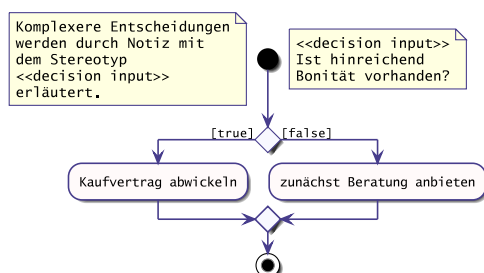
```

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 Erörterungen nötig sind sieht die UML vor, dass diese über eine Notiz mit dem Stereotyp *<<decision input>>* erfolgt, die mit der *decision node* verbunden wird.

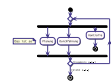
In PlantUML gibt es diese Möglichkeit nicht, daher muss man sich mit einer Notiz, die neben der vorigen Aktion steht, behelfen. Gemäß UML wird innerhalb der *decision node* nichts notiert - hier weicht die PlantUML-Dokumentation vom UML-Standard ab.



```

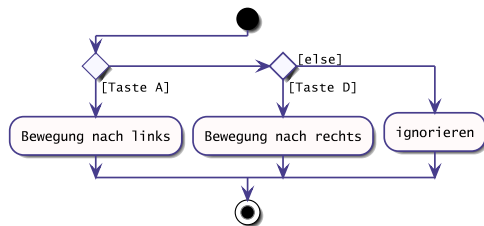
19 note right
20     <<decision input>>
21     Ist hinreichend
22     Bonität vorhanden?
23 end note
24 if() then ([true])
25     :Kaufvertrag abwickeln;
26 else ([false])
27     :zunächst Beratung anbieten;
28 endif

```



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:

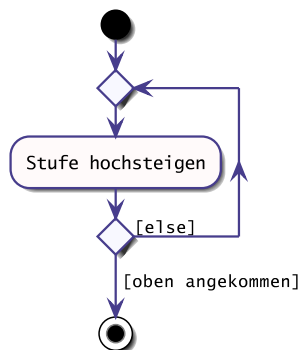


```

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

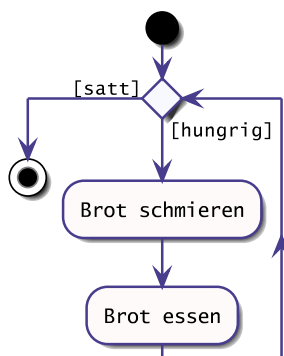
Eine nachgelagerte Bedingung, die eine Wiederholung bestimmter Aktionen erzwingt wird über die Rückführung der ausgehenden Kante mit *guard* einer *decision node* modelliert. Bei diesen fussgesteuerten Schleifen werden die Aktionen innerhalb der Schleife in jedem Fall einmal ausgeführt. Mit PlantUML 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.



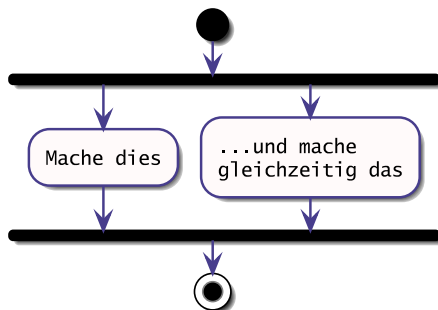
```

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.

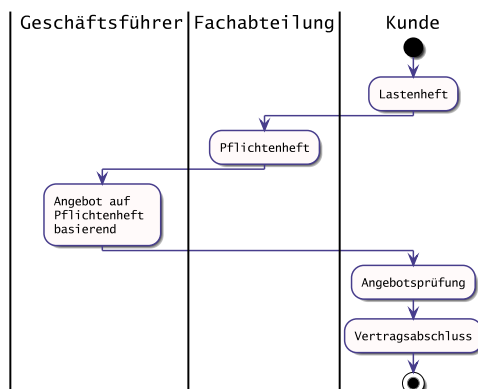


```

51 start
52 fork
53   :Mache dies;
54 fork again
55   :...und mache
56   gleichzeitig das;
57 end fork
58 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.



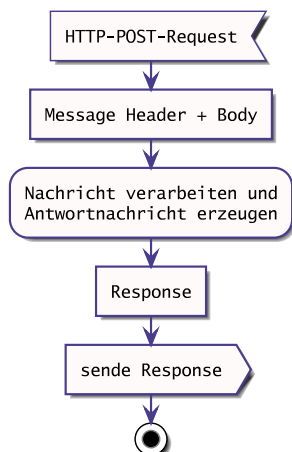
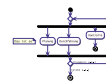
```

61 'Durch Nennung Reihenfolge festlegen
62 |Geschäftsführer|
63 |Fachabteilung|
64 |Kunde|
65 start
66 |Kunde|
67   :Lastenheft;
68 |Fachabteilung|
69   :Pflichtenheft;
70 |Geschäftsführer|
71   :Angebot auf
72   Pflichtenheft
73   basierend;
74 |Kunde|
75   :Angebotsprüfung;
76   :Vertragsabschluss;
77 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.



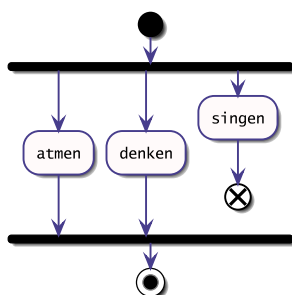
```

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.



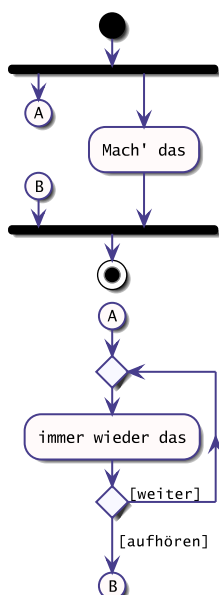
```

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

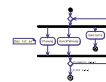
Um Kreuzungen zu vermeiden oder um komplexere Zusammenhänge übersichtlicher darstellen zu können, können Sprungmarken (A) verwendet werden. Abläufe können dadurch unterbrochen werden (bei PlantUML mit **detach**) und in gesonderten Aktivitätsdiagrammen ausgeführt werden.



```

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

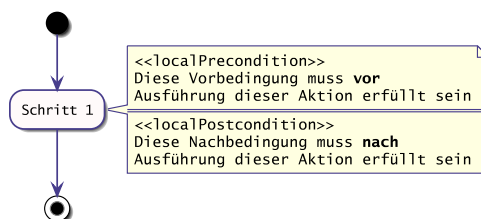
```



## 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



```

114 start
115 :Schritt 1;
116 note right
117     <<localPrecondition>>
118     Diese Vorbedingung muss <b>vor</b>
119     Ausführung dieser Aktion erfüllt sein
120     ===
121     <<localPostcondition>>
122     Diese Nachbedingung muss <b>nach</b>
123     Ausführung dieser Aktion erfüllt sein
124 end note
125 stop
    
```

## 11 plantUML-Webservice

PlantUML bietet einen Webservice, der Diagramme unmittelbar online rendert.

Variante 1: der PlantUML-Quelltext ist online als Resource verfügbar und soll gerendert ausgegeben werden. Hierzu muss eine URL nach dem Muster:

[http://www.plantuml.com/plantuml/proxy?fmt=AUSGABEFORMAT&src=https://URL\\_PLANTUMLSOURCE](http://www.plantuml.com/plantuml/proxy?fmt=AUSGABEFORMAT&src=https://URL_PLANTUMLSOURCE)  
erstellt werden, wobei als *AUSGABEFORMAT* **png**, **svg**, **eps**, **epstext** und **txt** genutzt werden kann.

Variante 2: PlantUML kodiert den Quelltext, um ihn so über den Webservice zugänglich (und bearbeitbar) zu machen. Die URL ist in diesem Fall wie folgt aufgebaut:

<http://www.plantuml.com/plantuml/AUSGABEFORMAT/CODIERTERQUELLTEXT>

wobei als *AUSGABEFORMAT* **png**, **svg**, **eps**, **epstext** und **txt** genutzt werden kann. Wird als *AUSGABEFORMAT* **uml** gewählt erhält man den bearbeitbaren Quelltext.

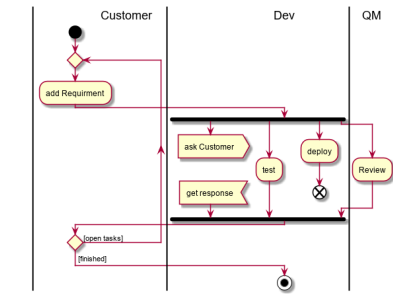
Editierbare Links lassen sich auch über den Service von planttext.com erstellen, sie nutzen die selbe Quelltextcodierung und URLs nach dem Muster: <https://www.planttext.com/?text=CODIERTERQUELLTEXT>

Auch in den erzeugten PNG-Dateien wird der codierte Quelltext als Titel hinterlegt, so dass sie sich relativ einfach später weiterverarbeiten lassen.

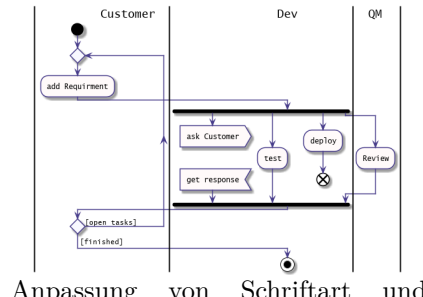


## 12 plantUML-Formatierung: Aufhübschen von Aktivitätsdiagrammen

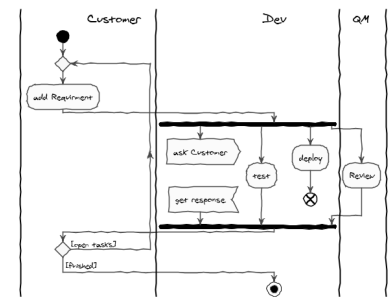
Wenn die Diagramme erstmal stehen will man sie aufhübschen. Dafür stehen allerlei möglichkeiten zur Verfügung, die v.a. auf der plantUML-Seite dargestellt werden. Einige Beispiele sind hier abgebildet:



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.

```

127 @startuml
128 |Customer|
129 |Dev|
130 |QM|
131 |Customer|
132 start
133 repeat
134 :add Requirment;
135 |Dev|
136 fork
137 :ask Customer>
138 Detach
139 :get response<
140 fork again
141 :test;
142 fork again
143 :deploy;
144 end
145 fork again
146 |QM|
147 :Review;
148 end fork
149 |Customer|
150 repeat while () is ([open
    tasks])
151 ->[finished];
152 |Dev|
153 stop
154 @enduml
    
```

```

156 @startuml
157 skinparam DefaultFontName
    "Lucida Sans Typewriter"
158
159 skinparam Activity{
160 BackgroundColor snow
161 BorderColor DarkSlateBlue
162 DiamondBackgroundColor
    ghostwhite
163 DiamondBorderColor
    DarkSlateBlue
164 }
165
166 skinparam Note{
167 BorderColor DarkSlateBlue
168 BackgroundColor LightYellow
169 }
170
171 skinparam ArrowColor
    DarkSlateBlue
172 |Customer|
173 |Dev|
174 |QM|
175 |Customer|
176 start
177 ...
178 stop
179 @enduml
    
```

```

180 @startuml
181 ' Welche Schriften gibt es
    auf dem System?
182 ' listfonts als
    plantUML-Kommando gibt's
    aus.
183 skinparam DefaultFontName
    "FG Virgil"
184 skinparam handwritten true
185 skinparam monochrome true
186 skinparam packageStyle rect
187 skinparam shadowing false
188
189 |Customer|
190 |Dev|
191 |QM|
192 |Customer|
193 start
194 ...
195 stop
196 @enduml
    
```

## References

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

[plantText] Projektwebsite., Website, auf der direkt plantUML-Quelltexte geparkt werden können:  
<https://www.planttext.com/>