Einführung in PlantUML

Version 1.0

Vanessa Petrausch

Projektpartner:

Karlsruher Institut für Technologie

FZI

Gefördert durch:

Bundesministerium für Arbeit und Soziales

aus Mitteln des Ausgleichsfonds
Förderkennzeichen: 01KM141108

INHALTSVERZEICHNIS

1 EINLEITUNG

PlantUML ist ein Open Source Werkzeug zur Erstellung von UML-Diagrammen. PlantUML ist frei verfügbar unter der GNU Public License[1]. Im Gegensatz zu der üblichen Darstellung von UML-Diagrammen als Grafiken werden die Diagramme mithilfe von PlantUML als Text dargestellt. Die Sprache ist dabei einfach gehalten und lehnt sich teilweise an allgemeine Programmiersyntaxen an. Beispielsweise werden die Begriffe „if“, „then“ und „else“ für bedingte Verzweigungen verwendet oder „join“ und „fork“ für parallele Abläufe.

Es sollte beachtet werden, dass PlantUML sich nicht für die Modellierung von UML-Diagrammen eignet, sondern ein reines Zeichenwerkzeug ist, mit welchem man semantisch falsche UML-Diagramme im Sinne der UML-Definition erzeugen kann.

Zusätzlich zur textuellen Darstellung können Grafiken aus der Syntax von PlantUML automatisch erstellt werden. Einerseits kann dies durch einen frei verfügbaren PlantUML-Server[2] realisiert werden, in welchen man seinen PlantUML-Text kopieren kann. Beim Aufrufen des Servers wird immer ein Beispiel angezeigt. PlantUML kann ebenso in weitere lokale Programme integriert werden. Beispiele dafür sind:

- Eclipse IDE
- GoogleDocs
- Microsoft Word
- NetBeans
- LaTex
- Jekyll
- Gedit, Notepad++, Vim, SublimeText
- Trac, Redmine, MediaWiki
- und einige mehr

Einige der Programme benötigen Graphviz[3] für die Darstellung der grafischen Darstellung. Falls keine Grafiken automatisch erzeugt werden sollen, genügt ein einfacher Texteditor um UML-Diagramme zu erzeugen. Für Diskussionen mit Personen, welche kein PlantUML beherrschen, kann es jedoch einfacher sein, eine grafische Darstellung erzeugen zu lassen.

Es ist jedoch zu beachten, dass keine der Tools für PlantUML einen inhaltlichen Abgleich machen. PlantUML ist vor allem ein „Zeichentool“ für UML-Diagramme, keine Modellierumgebung. Einige syntaktische Fehler in der Notation werden erkannt und ausgegeben, jedoch ist es immer noch möglich Diagramme zu erstellen, welche weder UML-Standards noch Programmierparadigmen entsprechen. Beispielsweise können Vererbungsbeziehungen in beide Richtungen erstellt werden, d.h. das eine Klasse „Kind“ von der Klasse „Eltern“ erben kann, die Klasse „Eltern“ jedoch gleichzeitig von der Klasse „Kind“ erbt.

Zurzeit werden acht Diagrammtypen unterstützt. Diese sind in alphabetischer Reihenfolge der deutschen Begriffe: Aktivitätsdiagramm, Anwendungsfalldiagramm (Use Case Diagram), Klassendiagramm, Komponentendiagramm, Objektdiagramm, Sequenzdiagramm, Verteilungsdiagramm (Deployment Diagram) und das Zustandsdiagramm (Statechart). Innerhalb

[1] http://www.gnu.org/licenses/gpl-3.0.de.html
[2] http://www.plantuml.com/plantuml/uml/
[3] http://www.graphviz.org/

dieses Dokumentes wird jedoch nur die Syntax für vier Diagramme angegeben: Das Klassendiagramm, Anwendungsfalldiagramm, Aktivitätsdiagramm und das Zustandsdiagramm. Eine Übersicht über alle verfügbaren Elemente von PlantUML innerhalb der jeweiligen Diagramme kann über die Webseite angesehen werden.[4] Weiterhin befindet sich eine Sammlung von Beispielen in PlantUML-Syntax in dem Dokument 1_3_PlantUML Code der Grafiken.docx, welches Teil der Schulungsunterlagen ist. In diesem werden alle Grafiken, welche Teile der Schulung sind in PlantUML ausgedrückt.

[4] http://plantuml.com

2 PlantUML Syntax

Alle Diagramme beginnen und enden mit einem vordefinierten Schlüsselwort. „@startuml“ leitet den PlantUML-Code ein, „@enduml“ beendet diesen wieder. Diese Schlüsselwörter müssen bei jedem Diagramm vorhanden sein, wenn es mittels Automatismus in eine grafische Darstellung überführt werden soll. Es erleichtert zudem die Abgrenzung verschiedene Diagramme innerhalb eines Dokuments. Im Klassen-, Anwendungsfall- und Zustandsdiagramm können Aliase für Elementnamen verwendet werden, sodass längere Bezeichnung mittels einer abkürzenden Bezeichnung innerhalb des Diagramms verwendet werden können. Diese werden immer mit dem Schlüsselwort „as“ definiert. Dabei steht der lange Name links vom Schlüsselwort und der Alias rechts davon, z.B. „Dies ist ein langer Name mit Leerzeichen as lang“. Die korrekte Syntax innerhalb der einzelnen Diagramme wird in den jeweiligen Abschnitten gegeben.

Alle Einrückungen, welche in den verschiedenen Beispielen vorgenommen wurden, dienen nur der besseren Übersicht und sind nicht notwendig. Es ist möglich alle Elemente an den Zeilenanfang zu setzen. Da in PlantUML jedoch nicht immer abschließende Zeichen einer Zeile gesetzt werden müssen, wie z.B. das Semikolon in vielen Programmiersprachen, ist es notwendig jede Definition eines Elementes oder einer Verbindung zwischen Elementen in eine neue Zeile zu setzen. Dies ist auch notwendig, wenn es einen eindeutigen Abschluss eines Elements durch eine Klammer oder eine anderes Zeichen gibt. Schließende Klammern von Blöcken müssen immer in einer eigenen Zeile stehen, öffnende Klammern immer in der jeweiligen Zeile der Definition.

2.1 Klassendiagramm

2.1.1 Titel von Klassendiagrammen

Der Titel eines Klassendiagramms kann mit Leerzeichen getrennt hinter das Schlüsselwort „title“ geschrieben werden. Der Titel muss nicht in Klammern oder Hochkommas gesetzt werden.

```
@startuml
title Einfacher Titel mit Leerzeichen
@enduml
```

2.1.2 Klassendefinition

Eine Klasse mit dem Namen „MeineKlasse“ ohne Attribute oder Methoden:

```
@startuml
class MeineKlasse
@enduml
```

Eine Klasse mit einem langen Klassennamen und einem Alias:

```
@startuml
class „Dies ist ein langer Klassenname“ as long
@enduml
```

Wurde eine Klasse definiert kann anschließend auf das Schlüsselwort „class“ verzichtet werden und nur der definierte Name der Klasse oder ein definierter Alias verwendet werden.

Eine Klasse kann sowohl Attribute als auch Methoden enthalten. Diese werden dabei ähnlich zu Programmiersyntaxen innerhalb von geschweiften Klammern gestellt. Die öffnende Klammer muss dabei zwingend in der Zeile der Definition der Klasse stehen und die schließende Klammer in einer eigenen Zeile. Die Reihenfolge des Typs und Name ist frei wählbar, d.h. es kann sowohl „attributtyp attributname“ als auch „attributname attributtyp“ oder wahlweise mit Doppelpunkt dazwischen „attributname: attributtyp“ geschrieben werden. Das gleiche gilt für Rückgabewerte der Methoden oder Parameterübergaben innerhalb von Methoden. Attribute und Methoden werden gleich definiert. Die Klammern der Parameter bei Methoden ist ausreichen, um diese als Methode zu definieren. Sollten Klammern vorhanden sind, wird diese von den Übersetzern in Grafiken automatisch in einen eigenen Abschnitt der Klasse gesetzt. Attribute und Methoden müssen nicht mit einem Zeichen, z.B. Semikolon abgeschlossen werden. Jedes Attribut und jede Methode muss jedoch in einer eigenen Zeile stehen.

```
@startuml
class MeineKlasse{
      String name
      Integer alter
      void methode(String parameter1, Int parameter2)
}
@enduml
```

2.1.3 Zusätzliche Merkmale für Attribute und Methoden

Zusätzlich können Sichtbarkeiten der Attribute und Methoden gegeben werden. Da die Definition der Attribute und Methoden frei wählbar ist, können diese einfach als Schlüsselwort davor geschrieben werden oder mittels der von UML definierten Zeichen abgekürzt werden. Das nachfolgende Beispiel zeigt eine Klasse mit vier Attributen. Die Attributnamen entsprechen in diesem Fall den Sichtbarkeiten der vorgestellten Zeichen.

```
@startuml
class MeineKlasse{
      - minusFürPrivate
      # rauteFürProtected
      ~ tildeFürPackagePrivate
      + plusFürPublic
}
@enduml
```

Ebenfalls können Attribute oder Methoden die Merkmale „Abstract“ oder „Static“ zugewiesen bekommen. Die entsprechenden Schlüsselwörter werden dabei in geschweiften Klammern vor die jeweilige Definition gestellt. Das nachfolgende Beispiel enthält ein static Attribut und eine abstract Methode.

```
@startuml
class MeineKlasse{
      {static} String name
      {abstract} void methods()
      }
@enduml
```

2.1.4 Abstrakte Klassen und Interfaces

Klassen können ebenfalls als “Abstract” definiert werden. Dabei wird das Schlüsselwort „abstract“ vor das Klassenschlüsselwort gestellt, jedoch ohne Klammern, wie bei den Attributen und Methoden.

```
@startuml
```

```
abstract class MeineAbstrakteKlasse
@enduml
```

Ein Interface wird ebenfalls mit dem vorangestellten Schlüsselwort „interface“ definiert.

```
@startuml
interface MeinInterface
@enduml
```

2.1.5 Verbindungen zwischen Klassen

Zwischen Klassen können verschiedene Verbindungen bestehen, welche nacheinander vorgestellt werden. Die einfachsten Verbindungen sind Assoziationen. Dabei kann die Assoziation gerichtet oder ungerichtet sein. Einfache Assoziationen werden durch ein einfaches oder doppeltes Minuszeichen dargestellt. Der Unterschied, ob ein einfaches oder doppeltes Zeichen verwendet wird liegt nur in der grafischen Interpretation, welche die Klassen jeweils unterschiedlich ausrichtet. Zur deutlicheren Darstellung der Assoziationen werden innerhalb dieses Dokuments doppelte Minuszeichen verwendet. Generell werden in PlantUML die grafischen Darstellungen der Assoziation in ASCII-Art nachgemacht, sodass die Zeichen nicht immer intuitiv verständlich sind und die einzelnen Zeichenkombinationen erlernt werden müssen. Alle Assoziationen können ebenfalls in die andere Richtung modelliert werden, an welchem Ende der Pfeil, das Aggregat, etc. sich befindet ist nicht festgelegt.

Einfache Assoziation:

```
@startuml
class A -- class B
@enduml
```

Gerichtete Assoziation in Richtung B:

```
@startuml
class A --> class B
@enduml
```

Bidirektionale Assoziation:

```
@startuml
class A <--> class B
@enduml
```

Aggregation, mit dem Aggregationsende bei der Klasse A:

```
@startuml
class A o-- class B
@enduml
```

Komposition, mit dem Kompositionsende bei der Klasse A:

```
@startuml
class A *-- class B
@enduml
```

Bei der Aggregation und Komposition kann optional noch ein Pfeil an das Ende bei B hinzugefügt werden, d.h. bei der Aggregation beispielsweise:

```
@startuml
class A o--> class B
@enduml
```

Vererbung zwischen der Unterklasse und der Oberklasse:

```
@startuml
class Unterklasse --|> class Oberklasse
@enduml
```

2.1.6 Zusatzelemente für Assoziationen

Assoziationen können Beschriftungen erhalten und eine Leserichtung festlegen. Ebenfalls können Multiplizitäten an den Enden definiert werden.

Beschriftungen einer Assoziation werden in der Zeile der Definition dieser mittels eines Doppelpunkts angehängt. Dabei kann die Beschriftung Leerzeichen enthalten und muss nicht, kann jedoch, in Hochkommas gesetzt werden.

```
@startuml
class A -- class B: Beschriftung mit Leerzeichen
@enduml
```

Die Leserichtung kann zusätzlich mittels eines Kleiner- oder Größer-Zeichens dargestellt werden. Dabei zeigt die „Spitze“ des Zeichens in die Leserichtung. Das Zeichen kann vor oder nach der Beschriftung stehen und alleine, falls keine Beschriftung angegeben ist. Im Beispiel ist die Leserichtung von der Klasse A zur Klasse B zusätzlich zur Beschriftung der Assoziation mit dem Label „Leserichtung“.

```
@startuml
class A -- class B: Leserichtung >
@enduml
```

Multiplizitäten werden an den jeweiligen Enden der Klasse beschrieben, auf welche diese zutreffen sollen. Die Multiplizitäten werden dabei in Anführungsstriche “ “ und zwischen Klassenname und Assoziationszeichen gesetzt. Als Beispiel nehmen wir eine einfache Assoziation zwischen den Klassen A und B an. Klasse A hat dabei die Multiplizität unendlich, d.h. * und die Klasse B 0 oder 1: “0..1“. Intervalle werden wir in der UML-Schreibweise mit zwei Punkten getrennt.

```
@startuml
class A “*” – “0..1”class B
@enduml
```

Es können auch individuelle Multiplizitäten angegeben werden, wie z.B. “many“, solange diese an der richtigen Position und in Anführungsstriche gesetzt sind. Diese Notation eignet

sich gut, um Rollennamen von Klassen zu simulieren, da es bisher in PlantUML keine Notation dafür gibt. Da beliebige Zeichen und Wörter innerhalb der Anführungsstriche geschrieben werden können, ist auch die Modellierung von Sichtbarkeiten von Rollen möglich, z.B. “+ Rollenname“.

2.2 Allgemeines Element: Notizen

Notizen sind in allen Diagrammen möglich und werden in allen Diagrammen mit der gleichen Notation dargestellt. Eine Notiz wird mit dem Schlüsselwort „note“ eingeleitet und dem nachfolgenden Text der Notiz in Anführungsstrichen. Mithilfe des Schlüsselwortes „as“ muss ein Alias, eine Abkürzende Notation, für die Notiz definiert werden, sodass spätere Aufrufe durch diesen Alias stattfinden können.

```
@startuml
note “Dies ist eine lange Notiz ohne sinnvollen Inhalt“ as notiz1
@enduml
```

Notizen werden in UML mithilfe von gestrichelten Linien an Elemente angehängt oder daneben platziert. Um eine Notiz beispielsweise an eine Klasse zu notieren wird dies folgendermaßen notiert:

```
@startuml
note “Dies ist eine lange Notiz ohne sinnvollen Inhalt“ as notiz1
class A .. notiz1
@enduml
```

Es ist ebenfalls möglich Notizen an Assoziationen zu definieren, indem direkt nach der Definition der Assoziation das Schlüsselwort „note on link“ verwendet wird. Dabei wird keine Verbindungslinie zur Assoziation hergestellt, wie bei einer Klasse oder anderen Elemente, wie States oder Anwendungsfälle, sondern die Notiz wird räumlich nahe an den Link platziert.

```
@startuml
class A -- class B
note on link: Notiz an der Verbindung zwischen A und B
@enduml
```

2.3 Anwendungsfalldiagramm

2.3.1 Anwendungsfälle

Anwendungsfälle werden innerhalb von runden Klammern dargestellt. Diese Notation kopiert wiederum die grafische Notation von Anwendungsfällen, welche als Ellipse dargestellt werden. Alternativ kann auch das Schlüsselwort „usecase“ genutzt werden. Beide Varianten werden im nachfolgenden Beispiel verwendet. Im weiteren Dokument wird wegen der kürzeren Notation jedoch die Klammernotation verwendet.

```
@startuml
(Einfacher Anwendungsfall) as usecase1
usecase (Ein anderer Anwendungsfall) as usecase2
@enduml
```

2.3.2 Akteure

Akteure werden entweder mittels des Schlüsselwortes „actor“ gefolgt vom Namen oder zwischen zwei Doppelpunkten dargestellt. Beide Varianten sind im nachfolgenden Beispiel angegeben. Ist der Name des Akteures mit Leerzeichen getrennt und es wird ein Alias verwendet müssen beide Elemente verwendet werden, wie das dritte Beispiel zeigt.

```
@startuml
:Ein Akteur: as actor1
actor Mensch
actor :Akteur mit längerem Namen: as actor2
@enduml
```

Akteure können zusätzlich Stereotype besitzen. In PlantUML ist es momentan nicht möglich verschiedene Typen von Akteuren darzustellen, weshalb die Definition von Stereotypen bei verschieden verwendeten Typen sinnvoll ist. Stereotype werden innerhalb von doppelt spitzen Klammern hinter den Namen des Akteures gesetzt.

```
@startuml
:Ein Akteur: as actor1 <<Human>>
:Machineller Akteur: <<Server>>
@enduml
```

2.3.3 Das System

Anwendungsfälle sind meist innerhalb eines Systems realisiert, welches als Rechteck um die Anwendungsfälle grafisch dargestellt wird. Entsprechend wird ein System mit dem Schlüsselwort „rectangle“ definiert. Geschweifte Klammern grenzen den Inhalt des Systems ab. Der Name des Systems steht direkt nach dem Schlüsselwort und vor der öffnenden Klammer. Alle enthaltenen Anwendungsfälle und ihre Verbindungen müssen innerhalb der Definition des Systems beschrieben werden. Die schließende Klammer des Systems muss in einer eigenen Zeile stehen. Das Beispiel enthält ein System mit dem Namen „Testsystem“ und einem Anwendungsfall „Usecase“.

```
@startuml
rectangle Testsystem{
      (Usecase)
}
@enduml
```

2.3.4 Assoziationen zwischen Akteur und Anwendungsfall

Assoziationen zwischen Akteuren und Anwendungsfällen werden wie im Klassendiagramm mit zwei Minuszeichen dargestellt mit optionalen Pfeilspitzen. Das nachfolgende Beispiel stellt eine ungerichtete und eine gerichtete Assoziation zwischen einem Akteur und je einem Anwendungsfall dar. Die Anwendungsfälle sind nach den jeweiligen Assoziationen benannt: Ungerichtet und Gerichtet.

```
@startuml
actor Person
Person -- (Ungerichtet)
Person -> (Gerichtet)
@enduml
```

Wie im Klassendiagramm können Beschriftungen und Multiplizitäten angegeben werden. Die Syntax ist dabei identisch zu derjenigen im Klassendiagramm. Im nachfolgenden Beispiel ist wiederum ein Akteur „Person“ mit zwei anwendungsfällen verbunden. Die Verbindung zum Anwendungsfall „Label“ ist mit einer Beschriftung „Assoziationsname“ beschriftet. Bei der Verbindung zum Anwendungsfall „Multiplizität“ ist die Multiplizität „2“ am Akteurende angegeben.

```
@startuml
actor Person
Person -- (Label): Assoziationsname
Person “2“ -- (Multiplizität)
@enduml
```

2.3.5 Vererbung

Die Vererbung zwischen Akteuren und Anwendungsfällen wird mit der gleichen Symbolfolge wie im Klassendiagramm erzeugt. Das nachfolgende Beispiel zeigt je eine Vererbung zwischen zwei Akteuren und Anwendungsfällen.

```
@startuml
actor Enkel
actor Person
Enkel --|> Person
(erbender Anwendungsfall) --|> (vererbender Anwendungsfall)
@enduml
```

2.3.6 Assoziationen zwischen Anwendungsfällen

Die Include- und Extend-Assoziation zwischen Anwendungsfällen wird durch einen gestrichelten Pfeil (..>) und einer Beschriftung der Assoziation mit dem Schlüsselwort erreicht. Beide Varianten sind im folgenden Beispiel enthalten.

```
@startuml
(Kasse) ..> (Bezahlung): <<include>>
(Hilfe) ..> (Kasse): <<extend>>
@enduml
```

2.4 Aktivitätsdiagramm

In diesem Dokument wird die Syntax des Aktivitätsdiagramms Beta verwendet, da diese einfacher zu verstehen und zu erlernen ist. Bisher können keine Aktivitäten dargestellt werden, sondern nur Aktionen. Aktivitäten könnten mit Hilfe von Gruppierungen erstellt werden, dies entspricht jedoch nicht der korrekten Syntax und wird deshalb hier nicht verwendet.

2.4.1 Einfache Aktionen

Einfache Aktionen werden mithilfe eines Doppelpunktes begonnen und mithilfe eines Semikolons beendet. Dabei kann der Name Leerzeichen enthalten und sogar über mehrere Zeilen gehen, da in diesem Fall die Syntax einen eindeutigen Abschluss mit dem Semikolon hat, im Vergleich zu anderen Diagrammarten. Der Zeilenumbruch der Definition wird dabei auch in der Aktion dargestellt, sodass der Name der zweiten Aktion auch im Diagramm in zwei Zeilen steht. Beide Varianten sind im nachfolgenden Beispiel aufgeführt.

```
@startuml
:Einfache Aktion;
```

```
:Einfache Aktion über
mehrere Zeilen definiert;
@enduml
```

2.4.2 Start- und Endzustände

Der Startzustand wird durch das einfache Wort „start“ dargestellt, ohne Sonderzeichen, in einer eigenen Zeile.

```
@startuml
start
@enduml
```

Mithilfe des Schlüsselwortes „stop“, ebenfalls ohne Sonderzeichen in einer eigenen Zeile, kann ein Endknoten dargestellt werden.

```
@startuml
stop
@enduml
```

Alternativ dazu kann ein Flussende mittels des Schlüsselwortes „end“ erzeugt werden.

```
@startuml
end
@enduml
```

2.4.3 Kontrollflüsse

Kontrollflüsse werden in Aktivitätsdiagrammen implizit dargestellt und nicht explizit modelliert wie in den vorhergehenden Diagrammen. Kontrollflüsse werden immer zwischen zwei Elementen hergestellt, welche direkt untereinander stehen. Dabei spielt es keine Rolle, ob es sich um Aktionen oder vordefinierte Knoten wie Start-End- oder Verzweigungsknoten handelt. Das nachfolgende Beispiel stellt einen Kontrollfluss von einem Startknoten zu einer Aktion und dann zu einem Endknoten dar.

```
@startuml
start
:Hello World;
stop
@enduml
```

2.4.4 Bedingte Verzweigungen

Bedingte Verzweigungen werden mithilfe der Schlüsselwörter „if (Bedingung) then (Option)…elseif (Bedingung2) then (Option)… else (Option) endif“ verwendet. Dabei können beliebig viele Bedingungen hintereinander geschaltet werden. Für syntaktisch korrekte UML-Diagramme können die Bedingungen hinter „if ()“ und „elseif ()“ nicht gesetzt werden, da diese damit falsch gesetzt werden. Die verschiedenen Bedingungen, welche die einzelnen Pfade schalten werden in die Klammern nach dem „then()“ gesetzt oder nach dem „else()“. Die Syntax von PlantUML orientiert sich hierbei eher an derjenigen von Prozessdiagrammen, welche bei Verzweigungen Fragen stellen, welche mit verschiedenen Optionen beantwortet werden können. Dies ist jedoch nicht der Fall in der UML. Zuerst wird ein Beispiel mit einer Verzweigung gezeigt, welche zwei Optionen enthalten und danach wieder zusammengeführt werden. Nach der Bedingung wird jeweils eine Aktion ausgeführt. Das Diagramm beginnt und endet mit einem Start- und Endzustand.

```
@startuml
start
if () then ([Schokoladeneis verfügbar])
      :Schokoeis essen;
else ([kein Schokoladeneis])
      :Vanilleeis essen;
endif
stop
@enduml
```

2.4.5 Parallele Verzweigung

Eine parallele Verzweigung wird mittels der Schlüsselwörter „fork“, „fork again“ und „end fork“ dargestellt. Mitels „fork“ wird eine parallele Ausführung gestartet. Mithilfe von „end fork“ wird diese wieder beendet. Alles was dazwischen definiert wird, wird zwischen den parallelen Ausführungen dargestellt. Es ist möglich eine „fork“-Anweisung mit einer Aktion direkt wieder mit einem „end fork“ zu schließen, dies stellt jedoch nur die Ausführung eines Pfades dar und nicht von parallelen. Mithilfe von „fork again“ wird ein paralleler Pfad hinzugefügt. Dies kann beliebig oft wiederholt werden, sodass nicht nur zwei Pfade parallel ausgeführt werden, sondern mehrere. Es ist jedoch stets nur ein „end fork“ notwendig, nicht alle „fork again“ müssen syntaktisch beendet werden. Das folgende Beispiel zeigt die parallele Ausführung von „Schokoeis essen“ und „Sahne essen“. Vor und nach der parallelen Ausführung befindet sich nur der Start- und Endknoten.

```
@startuml
start
fork
      :Schokoeis essen;
fork again
      :Sahne essen;
end fork
stop
@enduml
```

2.5 Zustandsdiagramm

Im Zustandsdiagramm gibt es bisher die wenigsten Elemente, welche modelliert werden können. Bedingte Verzweigungen und parallele Abläufe können nicht modelliert werden.

2.5.1 Einfache Zustände

Ein Start- und Endzustand wird mit dem gleichen Symbol modelliert: ein Stern in eckigen Klammern. Je nachdem ob Kanten davon ausgehend modelliert werden oder eingehende wird zwischen dem Start- und Endzustand unterschieden. Sobald ein Start- oder Endzustand modelliert ist können einfache Zustände ohne Schlüsselwörter verwendet werden, die die Syntax eindeutig ein Zustandsdiagramm modelliert. Das nachfolgende Beispiel zeigt ein Beispiel von einem Startzustand zu einem einfachen Zustand zu einem Endzustand.

```
@startuml
[*] --> Zustand
Zustand --> [*]
@enduml
```

2.5.2 Transitionen

Transitionen werden immer zwischen zwei Zuständen modelliert, wie bereits in den anderen Diagrammarten durch Pfeile. Diese können mittels der Doppelpunktnotation beschriftet werden. „Events [Bedingung]/Aktion“ können durch die jeweiligen Trennzeichen symbolisiert werden, da diese keine Bedeutung innerhalb einer Beschriftung einer Transition haben. Das nachfolgende Beispiel zeigt eine einfache Transition zwischen zwei Zuständen mit einer Beschriftung.

```
@startuml
[*] --> Zustand
Zustand --> Zustand2: event[bedingung]/aktion
Zustand2 --> [*]
@enduml
```

2.5.3 Zusammengesetzte Zustände

Zusammengesetzte Zustände werden mithilfe des Schlüsselwortes „state“ dem Namen, einen optionalen Alias und den inneren Zuständen in geschweiften Klammern definiert. Innerhalb zusammengesetzter Zustände können wiederum zusammengesetzte Zustände definiert werden. Das Beispiel zeigt einen Zusammengesetzten Zustand „Composite“ mit einem Kindzustand „Kindcomposite“, welcher wiederum mit einem neuen zusammengesetzten Zustand „Composite2“ verbunden wird. „Composite2“ hat zwei Kindzustände „Zustand1“ und „Zustandt2“

```
@startuml
state Composite{
      Kindcomposite --> Composite2
      state Composite2{
            Zustand1 --> Zustand2
      }
}
@enduml
```

2.5.4 Orthogonale Zustände

Orthogonale Zustände benötigen immer eine Definition eines zusammengesetzten Zustands. Innerhalb dieses Zustands können die Regionen angelegt werden. Für diese Regionen können jedoch keine Namen vergeben werden, es sei denn es werden Notizen dazu verwendet. Die Inhalte der verschiedenen Regionen werden mit der normalen Syntax von Zustandsdiagrammen definiert. Zur Abgrenzung zwischen den Regionen werden in eine Zeile zwei Minuszeichen definiert. Das nachfolgende Beispiel enthält 3 Regionen, welche jeweils vollständige Zustandsdiagramme mit je einem Start- und Endzustand enthalten und einem Zustand dazwischen, welcher mit RegionA, RegionB oder RegionC benannt ist.

```
@startuml
state BeispielRegionen{
      [*] -> RegionA
      RegionA -> [*]
--
      [*] -> RegionB
      RegionB -> [*]
--
      [*] -> RegionC
      RegionC -> [*]
}
@enduml
```