Ausgewählte Fragmente im UML-Sequenzdiagramm

Quelle: Kecher, Christoph & Salvanos, Alexander: UML 2.5 – Umfassende Handbuch, 5. Auflage, Rheinwerk Computing (2015)

Alternative

In einem mit **alt** gekennzeichneten kombinierten Fragment darf nur einer der Interaktions-Operanden ausgeführt werden. Er entspricht einem `if-then-else`-Konstrukt.

In eckigen Klammern notierte **Guards** überwachen die Ausführung eines jeden Operanden, die nur bei einer Auswertung des Guards zu `true` erfolgt.

Abbildung 11.24 zeigt ein Beispiel für ein `alt`-Fragment. Nachdem ein `:Lehrling` seine Arbeit erledigt hat, wird diese vom `:Meister` geprüft. Hat der `:Lehrling` gute Arbeit geleistet, wird er gelobt, andernfalls (`else`) getadelt.

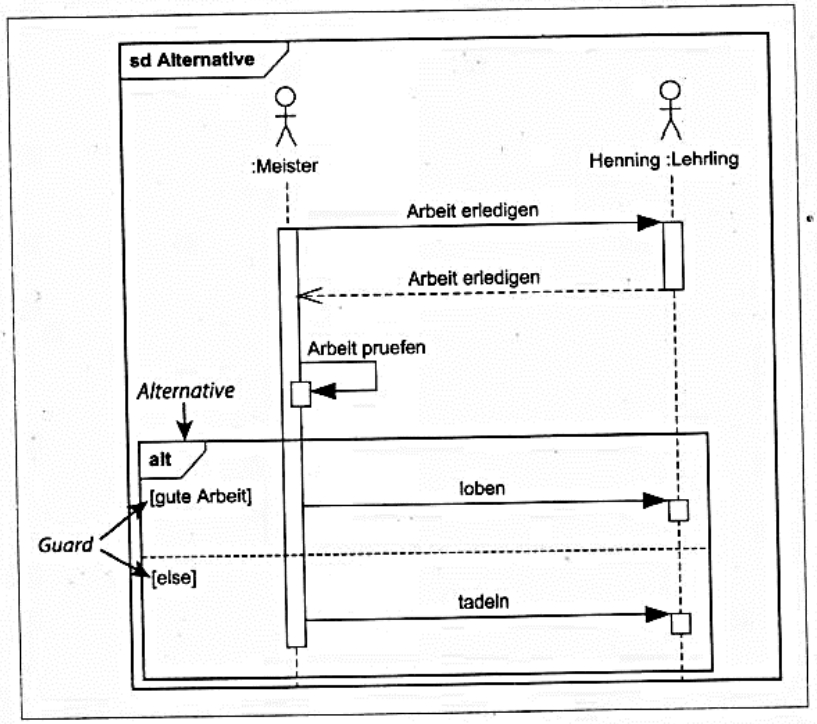

Abbildung 11.24 Alternative

Die UML schreibt vor, dass höchstens einer der Interaktions-Operanden eines `alt`-Fragments ausgeführt werden darf. Achten Sie daher darauf, dass die Guards disjunkt sind, sich also gegenseitig vollständig ausschließen.

Die Guards müssen erschöpfend alle möglichen Werte der spezifizierten Bedingungen erfassen. Fügen Sie gegebenenfalls den Guard `else` hinzu, der immer dann zu

376

`true` ausgewertet wird, wenn alle anderen nicht zutreffen. Der Verzicht auf die Definition eines Guards impliziert einen Guard, der immer `true` ist und damit die Ausführung des Interaktions-Operanden immer gestattet.

Zwischen unterschiedlichen `alt`-Fragmenten können zusätzlich **Fortsetzungen** (engl. *Continuations*) definiert werden.

Wie bereits erläutert wurde, läuft die Zeit in Sequenzdiagrammen von oben nach unten ab. Liest man ein Sequenzdiagramm damit von oben nach unten, stellt die erste gefundene Fortsetzung die *Sprungmarke*, die nächste Fortsetzung mit demselben Namen die *Aufsetzmarke* dar.

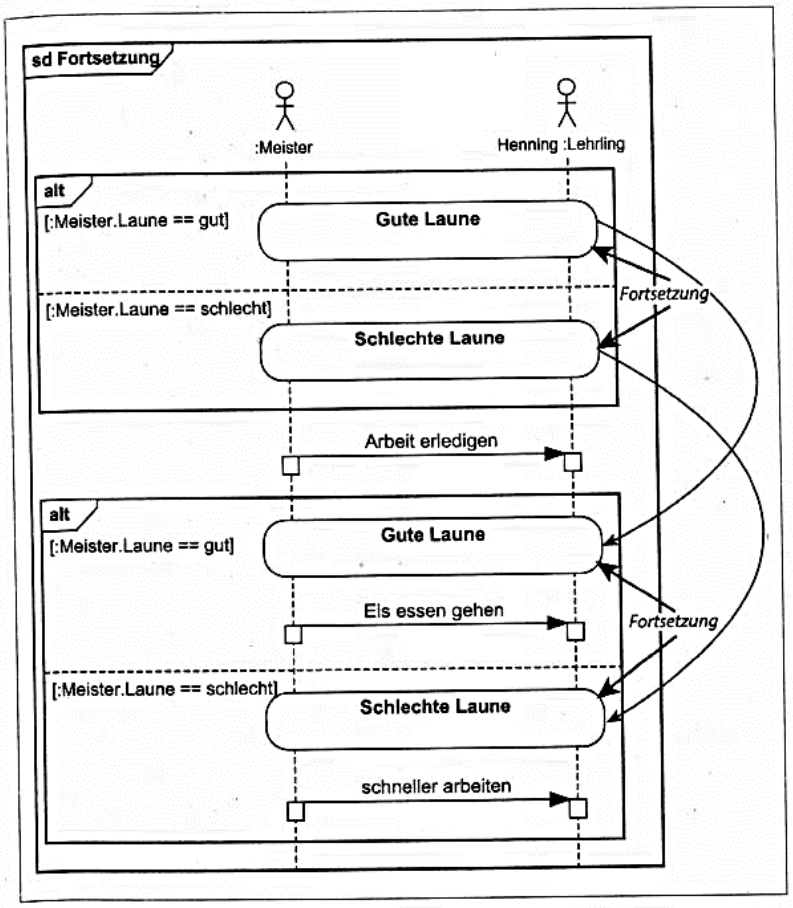

Abbildung 11.25 Fortsetzung

Hat der `:Meister` aus Abbildung 11.25 beispielweise gute Laune (erste `Gute Laune`-Fortsetzung von oben), erfolgt ein Sprung zur nächsten Fortsetzung `Gute Laune` und der `:Meister` sendet dem `:Lehrling` die Nachricht, Eis essen zu gehen. Hat er schlechte Laune (erste `Schlechte Laune`-Fortsetzung von oben), wird die nächste Fortsetzung `Schlechte Laune` angesprungen und der `:Lehrling` angewiesen, `schneller zu arbeiten`.

Treffen beide Guards nicht zu, erhält der :Lehrling lediglich die Nachricht, die Arbeit zu erledigen.

Fortsetzungen stellen in Sequenzdiagrammen eine Art »goto« dar, das beispielsweise in der Programmiersprache C# bekannt ist. Die meisten Programmierer vermeiden die Verwendung dieses Konstrukts, das durch eine andere Art der Programmierung ersetzt werden kann.

Ebenso verhält es sich auch mit Fortsetzungen in Sequenzdiagrammen, sodass von deren Verwendung an dieser Stelle eher abgeraten wird (Java unterstützt z. B. kein goto). Das Diagramm aus Abbildung 11.25 lässt sich beispielsweise auch wie folgt ohne Fortsetzungen modellieren:

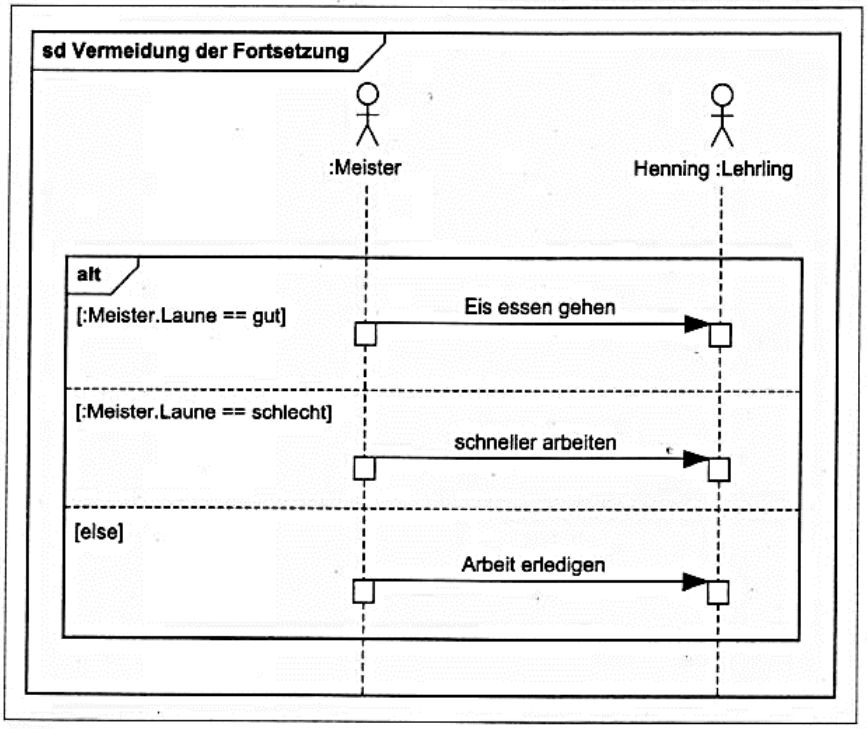

Abbildung 11.26 Diagramm aus Abbildung 11.25 ohne Fortsetzungen

Ist die Laune des Meisters gut, sendet er dem Lehrling eine Nachricht, Eis essen zu gehen. Ist die Laune des Meisters dagegen schlecht, bekommt der Lehrling die Nachricht, schneller zu arbeiten. Treffen beide Guards nicht zu, erhält der Lehrling die Nachricht, seine Arbeit zu erledigen.

Option

Durch den Interaktions-Operator **opt** wird ein kombiniertes Fragment als optional gekennzeichnet.

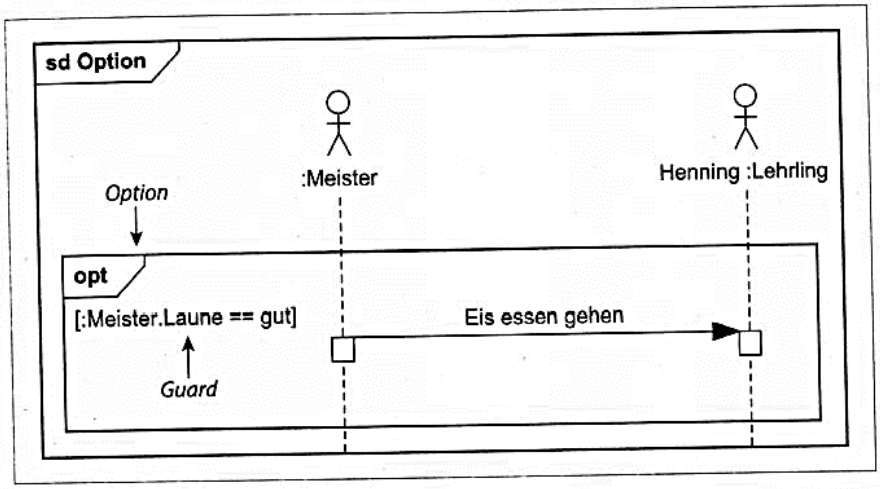

Abbildung 11.27 Option

Alle Interaktionen innerhalb des Fragments werden nur unter der definierten Bedingung (Guard) ausgeführt. Ein `opt`-Fragment entspricht semantisch einem `alt`-Fragment, das nur einen einzigen Interaktions-Operanden beinhaltet.

Nur wenn er gute Laune hat, erlaubt der `:Meister` dem `:Lehrling` aus Abbildung 11.27, Eis essen zu gehen.

Programmiertechnisch kann ein `opt`-Fragment sowohl in Java als auch in C# mit einem `if` ohne `else` realisiert werden.

Parallelität

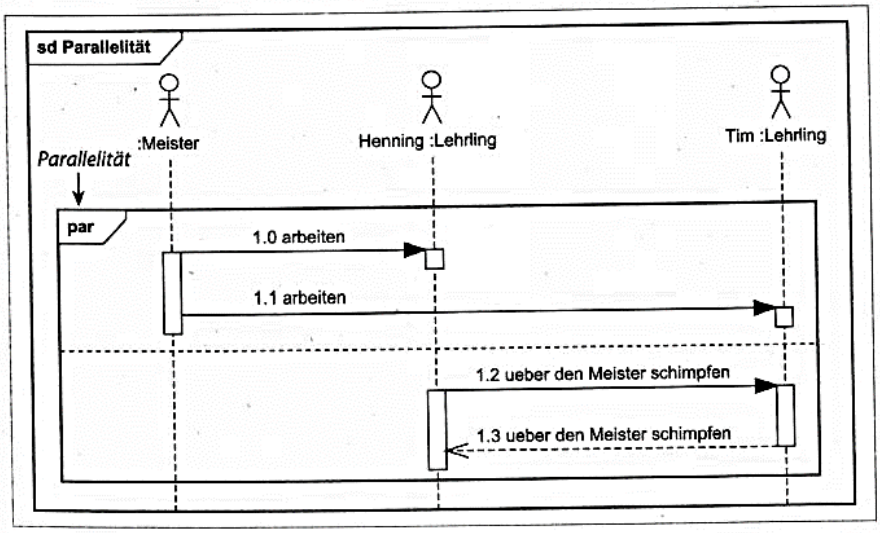

Abbildung 11.28 Parallelität

Der Interaktions-Operator **par** kennzeichnet die Ausführung der Interaktions-Operanden als parallel. Der Austausch von Nachrichten aus *unterschiedlichen* Interaktions-Operanden kann in *beliebiger* Reihenfolge durchgeführt werden, solange die Folge der Nachrichten innerhalb der *einzelnen* Operanden *eingehalten* wird.

Abbildung 11.28 beinhaltet ein Beispiel eines par-Fragments. Da die Nachrichtenfolge innerhalb eines Interaktions-Operanden nicht veränderbar ist, muss die Nachricht `1.0` vor der Nachricht `1.1` gesendet werden. Ebenfalls muss die Nachricht `1.2` vor `1.3` erfolgen.

Die Reihenfolgen der Nachrichten unterschiedlicher Interaktions-Operanden ist jedoch frei. Damit ist es auch erlaubt, Nachricht `1.2` vor `1.0` abzusenden, wodurch beispielsweise die folgenden Nachrichtenreihen zulässig werden:

`1.2 => 1.0 => 1.3 => 1.1`

`1.0 => 1.2 => 1.3 => 1.1`

`1.2 => 1.0 => 1.1 => 1.3`

Die Realisierung von parallelen Abläufen in Java und C# wurde bereits in Abschnitt 9.3.9 vorgestellt.

Schleife

Der **Schleifen-Operator** (engl. *loop*) spezifiziert eine Anzahl von **wiederholten Ausführungen des Interaktions-Operanden.**

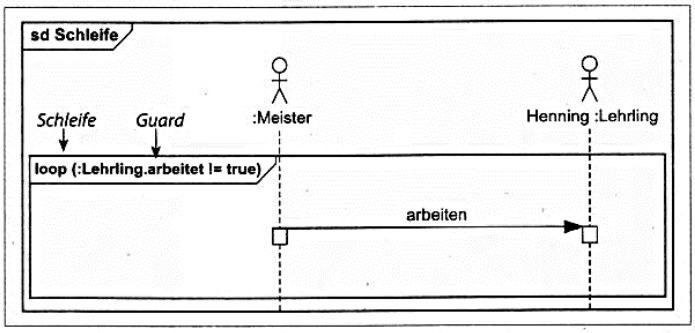

Abbildung 11.38 Schleife

Die Anzahl der Wiederholungen wird dabei vom **Guard** bewacht, sodass sich die Syntax des Schleifen-Operators wie folgt darstellt (eckige Klammern bedeuten »optional«):

loop ['(' <minint> [, <maxint>] ')']

oder

388

loop ['('<boolescher Ausdruck>')']

Die Elemente **<minint>** und **<maxint>** stellen Platzhalter für natürliche Zahlen dar, wobei **<minint>** kleiner oder gleich **<maxint>** sein muss. Weiterhin erlaubt die UML die Verwendung eines Sterns (*), der beliebig viele Wiederholungen erlaubt.

Der Ausführung des Interaktions-Operanden wird mindestens **<minint>** und maximal **<maxint>** oder so lange wiederholt, bis der **<boolesche Ausdruck>** zu `false` ausgewertet wird.

Die nachfolgenden Beispiele machen die Definitionsmöglichkeiten der Iterationsanzahl deutlicher:

- `loop(5)`
  feste Anzahl von Wiederholungen (5)
- `loop(3,*)`
  minimal 3 bis beliebig viele Wiederholungen
- `loop(0,5)`
  maximal 5 Wiederholungen
- `loop`
  beliebige Anzahl von Wiederholungen, äquivalent zu: `loop(*)`
- `loop(x < 5)`
  die Schleife wird so lange wiederholt, bis der Ausdruck `x < 5` zu `false` ausgewertet wird.

Die Schleife aus Abbildung 11.38 modelliert, dass der `:Meister` dem `:Lehrling` so lange die Nachricht `arbeiten` senden soll, bis das Attribut des `:Lehrlings` `arbeite` auf `true` gewechselt ist – bis er also endlich zu arbeiten begonnen hat.

Wie Schleifen in Java oder C# realisiert werden können, beschreibt Abschnitt 9.3.10.