

Objektorientierte Modellierung

Sequenzdiagramm



Business Informatics Group
Institute of Software Technology and Interactive Systems
Vienna University of Technology
Favoritenstraße 9-11/188-3, 1040 Vienna, Austria
phone: +43 (1) 58801-18804 (secretary), fax: +43 (1) 58801-18896
office@big.tuwien.ac.at, www.big.tuwien.ac.at

Herzlich willkommen zur heutigen Vorlesungseinheit OOM. Heute schauen wir uns das Sequenzdiagramm, eines der vier Interaktionsdiagramme von UML an.

Literatur

- Die Vorlesung basiert auf folgendem Buch:



UML @ Classroom:
Eine Einführung in die objektorientierte Modellierung
*Martina Seidl, Marion Brandsteidl,
Christian Huemer und Gerti Kappel*

dpunkt.verlag

Juli 2012

ISBN 3898647765

- Anwendungsfalldiagramm
- Strukturmodellierung
- Zustandsdiagramm
- **Sequenzdiagramm**
- Aktivitätsdiagramm

Inhalt

- Interaktionen und Nachrichten
- Überblick Interaktionsdiagramme
- Basiselemente des Sequenzdiagramms
 - Diagrammrahmen
 - Lebenslinie
 - Nachrichten
 - Parameter, Lokale Attribute
- Zeiteinschränkungen und Zustandsinvarianten
- Kombinierte Fragmente
 - Verzweigungen und Schleifen
 - Nebenläufigkeit und Ordnung
 - Filterungen und Zusicherungen

Ich werde zunächst kurz motivieren, warum man Interaktionsdiagramme benötigt und was man damit machen kann. UML bietet vier Interaktionsdiagramme, von denen wir drei nur überblicksmäßig kennen lernen werden. Im Detail betrachten wir den prominentesten Vertreter, das Sequenzdiagramm.

Interaktionen und Nachrichten

- **Interaktion**
 - Zusammenspiel mehrerer Kommunikationspartner
 - Nachrichten- und Datenaustausch
- **Interaktionen durch**
 - Signale
 - Operationsaufrufe
 - Aufruf einer Operation einer Klasse
 - Antwort: Ergebnis der aufgerufenen Operation
- **Steuerung der Interaktionen durch**
 - Bedingungen
 - Zeitereignisse

Was bedeutet der Begriff der **Interaktion**? Bei der Interaktion geht es darum das Zusammenspiel zwischen unterschiedlichen Kommunikationspartnern darzustellen. Letztendlich wollen wir Nachrichtenaustausch und Datenaustausch modellieren.

Interaktionen stellen wir einerseits durch **Signale** und andererseits durch **Nachrichten** dar. Ein Signal wird versendet, es wird aber nicht unbedingt ein Feedback darauf erwartet, d.h. wir haben asynchrone Kommunikation. Oder aber, wenn wir an die Programmierung denken, haben wir **Operationsaufrufe**, d.h. in Java-Terminologie: Wir wollen Methoden aufrufen, irgendetwas berechnen und der Rückgabewert ist letztendlich die Antwort, die wir als Ergebnis der aufgerufenen Operation erhalten. In diesem Fall sprechen wir von Nachrichten, die synchrone Kommunikation realisieren.

Wir wollen unsere **Interaktionen auch steuern**, d.h. wir brauchen Kontrollelemente, ähnlich wie in Programmiersprachen. Dort haben wir ja z.B. Schleifen oder Verzweigungen, und auch das wollen wir in unseren Interaktionsdiagrammen haben. D.h. wir wollen beliebige **Bedingungen** angeben, die gelten, damit eine Interaktion stattfindet bzw. wir wollen **Zeitereignisse** darstellen, die notwendig sind, damit eine Kommunikation stattfindet. z.B. wollen wir sagen können, dass eine bestimmte Interaktion um 5:00 Uhr stattfindet.

Interaktionsdiagramme

- Zeigen wie Nachrichten zwischen verschiedenen Interaktionspartnern in einem bestimmten Kontext ausgetauscht werden
- Beschreibung von Kommunikationssituationen durch:
 - Kommunikationspartner und deren Lebenslinien
 - Interaktionen
 - Nachrichten
 - Mittel zur Flusskontrolle
- Unterschiedliche Anforderungen und Betonung unterschiedlicher Aspekte
 - ⇒ 4 verschiedene Typen von Interaktionsdiagrammen

UML bietet vier verschiedene Arten von Interaktionsdiagrammen, die wir uns jetzt überblicksmäßig anschauen werden. Wir werden uns in dieser Vorlesung wie schon gesagt vor allem auf das Sequenzdiagramm beschränken, das auch in der Praxis sehr häufig eingesetzt wird. Wenn Sie zum Beispiel ein Buch über Design-Patterns lesen, in dem beschrieben ist, wie man ein System idealer Weise vereinfacht oder effizient implementiert, werden Sie sehr oft Sequenzdiagramme finden, weil sie eine sehr einfache Weise darstellen, Kommunikationsabläufe zu modellieren.

Wir wollen Interaktionsdiagramme für die Abbildung von Kommunikationssituationen verwenden. Was haben wir dafür zur Verfügung bzw. welche Elemente brauchen wir dafür? Zunächst brauchen wir auf jeden Fall einmal den Kommunikationspartner. D.h. wir müssen zunächst einmal darstellen, wer überhaupt an der Interaktion beteiligt ist und wie lange die Kommunikationspartner existieren. Dann haben wir Interaktionen, die ich schon kurz beschrieben habe, z.B. in Form von Nachrichten. Und dann haben wir noch Mittel zur Flusskontrolle um die Abläufe zu steuern.

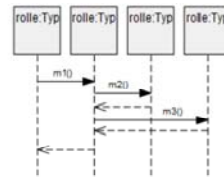
Interaktionsdiagramme – Arten (1/2)

- Die 4 Arten von Interaktionsdiagrammen sind für einfache Interaktionen semantisch äquivalent

- Betonung unterschiedlicher Aspekte

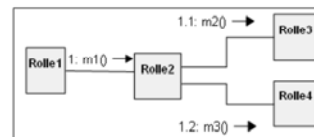
- Sequenzdiagramm** zeigt den zeitlichen und logischen Nachrichtenfluss

- Zeit ist eigene Dimension



- Kommunikationsdiagramm** ist »strukturell« orientiert

- Zeigt die Beziehungen zwischen Interaktionspartnern – Kontextaspekt
 - Reihenfolge von Nachrichten nur über Dezimalklassifikation ausgedrückt
 - Zeit ist keine eigene Dimension



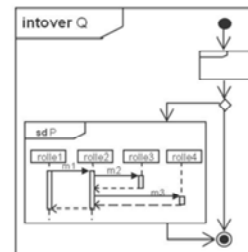
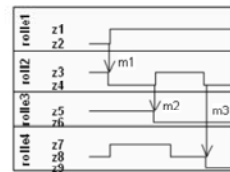
Die vier Diagramme, die wir in UML zur Darstellung von Interaktionen haben, sind einander sehr ähnlich. Sie bieten sehr ähnliche Konzepte, haben allerdings einen unterschiedlichen Fokus. D.h. ermöglichen es uns, unterschiedliche Aspekte der Kommunikation hervorzuheben.

Zunächst haben wir einmal das **Sequenzdiagramm**, das ich in dieser Vorlesung im Detail erklären werde. Hier gib es Interaktionspartner, die horizontal angeordnet werden und vertikal verläuft die Zeit. Den Nachrichtenaustausch zwischen den Interaktionspartnern drücken wir hier einfach durch Pfeile aus.

Weiters gibt es das **Kommunikationsdiagramm**. Dieses Diagramm ist strukturell orientiert. im Gegensatz zum Sequenzdiagramm, in dem wie gesagt die Zeit quasi von oben nach unten vergeht, haben wir hier die Zeit nicht als eigene Dimension dargestellt. Vielmehr geht es uns darum die Beziehung zwischen den Interaktionspartnern aufzuzeigen. In diesem Diagramm werden die Interaktionspartner als Rechtecke dargestellt. Diese werden durch Linien miteinander verbunden, wenn zwischen diesen ein Nachrichtenaustausch stattfindet. Die Nachrichten sind durch einen Nachrichtennamen gekennzeichnet, und ein kleiner Pfeil daneben stellt dar, in welche Richtung der Nachrichtenaustausch stattfindet. Und um eine Ordnung in diesen Kommunikationsfluss darzustellen, haben wir hier die Möglichkeit, dies durch Dezimalzahlen auszudrücken.

Interaktionsdiagramme – Arten (2/2)

- **Zeitdiagramm** zeigt Zustandsänderungen der Interaktionspartner aufgrund von Zeitereignissen
 - Vertikale Dimension repräsentiert Interaktionspartner und ihre möglichen Zustände
 - Horizontale Dimension repräsentiert die Zeitachse
- **Interaktionsübersichtsdiagramm** zeigt das Zusammenspiel von verschiedenen Interaktionen
 - Visualisiert in welcher Reihenfolge und unter welchen Bedingungen Interaktionsabläufe stattfinden



Beim **Zeitdiagramm** sind die Interaktionspartner vertikal, d.h. von oben nach unten, angeordnet, und die Zeit verläuft hier horizontal, d.h. von links nach rechts. Mittels Zeitdiagramm haben wir die Möglichkeit Zustandsänderungen auszudrücken. Diese Zustände können mittels Zustandsdiagramm detailliert spezifiziert werden. Ein Objekt befindet sich ja immer in einem bestimmten Zustand. Wenn Sie z.B. an einen Thread denken, dann kann der in einem Zustand „sleep“ sein, er kann „idle“ sein, er kann gerade etwas machen, usw. Im Zustandsdiagramm ist es möglich, Zustände explizit einzuzichnen, die ein Interaktionspartner annehmen kann. Und was wir hier im Zeitdiagramm machen können ist, die Zustandsübergänge, die aufgrund von Zeitereignissen oder eines Nachrichtenaustausches durchgeführt werden, explizit darzustellen. Hier haben wir z.B. einen Interaktionspartner in der Rolle „rolle1“. Das Rollenkonzept werde ich dann nachher noch einmal wiederholen, das sollten Sie schon rudimentär kennengelernt haben. Dieser Interaktionspartner befindet sich zunächst im Zustand „z2“, dann sendet er die Nachricht „m1“ und geht damit in den Zustand „z1“ über. Gleichzeitig bewirkt das Versenden der Nachricht „m1“ einen Zustandsübergang im Interaktionspartner mit der Rolle „rolle2“ vom Zustand „z3“ in den Zustand „z4“ usw. Das ist letztendlich das, was das Zeitdiagramm zu bieten hat. Leute von Ihnen, die einen Hintergrund aus der technischen Informatik oder aus der Elektrotechnik haben, werden diese Art von Diagrammen wahrscheinlich schon häufiger gesehen haben.

Das letzte Diagramm ist das **Interaktionsübersichtsdiagramm**. Es bietet die Möglichkeit, verschiedene Interaktionsdiagramme zusammenzubauen bzw. zusammenzustecken. D.h. wir haben hier Elemente die uns erlauben, verschiedene Interaktionsdiagramme miteinander zu verknüpfen. Diese Elemente finden Sie in anderen Diagrammen wie dem Zustandsdiagramm bzw. im Aktivitätsdiagramm wieder. Hier haben wir einen Startknoten, bei dem wir beginnen dieses Diagramm zu lesen. Dann wird eine Kommunikation durchgeführt, die wir hier schematisch dargestellt haben. Dann kommen wir zu so einer unausgefüllten Raute. Diese repräsentiert eine Verzweigung, d.h. normalerweise in einem konkreten Diagramm hätten wir hier jetzt noch eine Bedingung dabeistehen und wenn jetzt die Bedingung erfüllt ist, dann gehen wir in den einen Ast, wenn sie nicht erfüllt ist, gehen wir in den anderen Ast. Wenn wir in diesen linken Ast gehen, dann wird eine Interaktion ausgeführt, die wiederum durch ein Sequenzdiagramm repräsentiert ist und in beiden Fällen kommen wir dann zu einem Endknoten und sind fertig mit der Ausführung von diesem Ablauf. D.h. im Prinzip erlaubt uns das Interaktionsübersichtsdiagramm zu visualisieren, in welcher Reihenfolge und unter welchen Bedingungen komplette Interaktionen ausgeführt werden.

Einsatzbereiche

- Modellierung der **Interaktionen eines Systems mit seiner Umwelt** (Systemgrenzen festlegen, System als Black-Box)
- Modellierung der **Realisierung eines Anwendungsfalls**
- Modellierung des **Zusammenspiels der internen Struktur** einer Klasse, Komponente oder Kollaboration
- Modellierung der Spezifikation von **Schnittstellen zwischen Systemteilen** (Zusammenspiel angebotene/benutzte Schnittstelle)
- Modellierung der **Operationen einer Klasse**

Warum brauchen wir Interaktionsdiagramme überhaupt? Es gibt verschiedene Einsatzbereiche. Letztendlich hängt das wieder damit zusammen, mit welcher Intention Sie Ihr Modell erstellen, es hängt damit zusammen in welcher Phase des Softwareentwicklungsprozesses Sie sich befinden und mit welchem Detailgrad Sie das ganze realisieren wollen. Sie können Interaktionsdiagramme auf einem sehr abstrakten Level einsetzen. Sie können aber auch bis hin zur konkreten Realisierung in einer Programmiersprache gehen und z.B. Methodenaufrufe darstellen.

Wenn wir irgendein Softwaresystem entwickeln beschreiben wollen, dann können wir mit dem Sequenzdiagramm z.B. sehr gut ausdrücken: **Wie soll das System mit seiner Umwelt interagieren?** Wer benutzt das System? Und wie wird es von den Benutzern benutzt? Wie ist hier der Ablauf? Wie kommuniziert der Benutzer mit diesem System? Hier ist es wichtig, die Systemgrenzen festzulegen. Es ist wichtig festzulegen, was kann das System, was kann es nicht. Wie wird es durch den Benutzern verwendet? Welche Funktionen stellt es dem Benutzer zur Verfügung? Was hier allerdings nicht gemacht wird ist, auf die Interna des Systems zu achten. Wir schauen in diesem Fall nicht in das System hinein, sondern wir sehen das System als Black Box mit wohldefinierten Schnittstellen, über die kommuniziert wird.

Man kann das Ganze dann verfeinern indem man sich einen **konkreten Anwendungsfall** anschaut und diesen näher spezifiziert. D.h. wir betrachten nicht mehr das System als Ganzes, sondern wir schauen uns ein konkretes Ziel an, das wir mit dem System erreichen wollen. Z.B. wenn Sie an ein System zur Verwaltung Ihres Kontos denken, dann können Sie z.B. „Geld überweisen“ als Anwendungsfall sehen und das könnten Sie mittels Sequenzdiagramm darstellen, und konkret visualisieren, wie hier die Interaktion zwischen Benutzer und System ausschauen muss, damit das Geld von einem Konto auf ein anderes Konto transferiert wird.

Dann kann man aber auch **in das System hineinschauen**, d.h. wir schauen uns wirklich schon an, wie hier die technische Realisierung bzw. die Implementierung aussieht – wir befinden uns dann auf Codeebene. Hier sehen wir das **Zusammenspiel zwischen den Klassen**. Wenn ich hier z.B. eine Methode habe, dann wird diese ja wahrscheinlich andere Methoden aufrufen. Und genau dieses Zusammenspiel kann z.B. durch das Sequenzdiagramm dargestellt werden.

Sequenzdiagramm

- **Darstellung von Interaktionen in 2 Dimensionen:**
 - **"horizontal"**: Interaktionspartner in Form von Rollen
Reihenfolge der Partner wird für eine möglichst übersichtliche Darstellung gewählt
 - **"vertikal"**: Zeitachse
Darstellung des zeitlichen Ablaufs der Kommunikation
- **wichtigste Notationselemente:**
 - **Lebenslinien**: Kommunikationspartner
 - **Nachrichten**: Pfeile



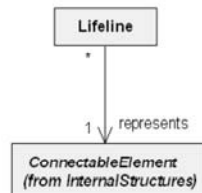
Dann kommen wir auch schon zum Sequenzdiagramm. Wie ist das Sequenzdiagramm aufgebaut? Zunächst haben wir wie schon einmal gesagt, **Interaktionspartner** und eine **Zeitachse**. Die Interaktionspartner sind auf der **horizontalen Achse** angeordnet. Die Zeitachse verläuft wie schon gesagt, auf der **vertikalen Achse** von oben nach unten. D.h. weiter oben ist – informell gesprochen - ein früherer Zeitpunkt als weiter unten und dadurch ist es möglich, Kommunikation darzustellen. Die Anordnung der Interaktionspartner auf der horizontalen Achse bleibt im Prinzip Ihnen überlassen. Wie die Interaktionspartner hier angeordnet sind spielt keine Rolle. Was allerdings zu beachten ist, ist, dass die Interaktionspartner möglichst in einer Art arrangiert sind, dass die Nachrichten möglichst übersichtlich dargestellt werden können. Also wenn Sie wissen, dass zwei Interaktionspartner viel miteinander kommunizieren, werden Sie versuchen diese möglichst nahe aneinander anzuordnen. Wenn Sie aber wissen, es findet nur ein geringer Nachrichtenaustausch statt, dann ist es natürlich kein Problem wenn diese weiter auseinander im Diagramm sind.

Im Prinzip haben wir zwei wichtigste Notationselemente. Einerseits die **Lebenslinien**, die die Kommunikationspartner darstellen und die **Nachrichten**, die durch Pfeile repräsentiert werden. Wenn Sie diese zwei Konzepte einmal verstanden haben, dann haben Sie im Prinzip schon die wichtigsten Elemente des Sequenzdiagramms verstanden. In der Praxis werden diese sehr oft informell eingesetzt.

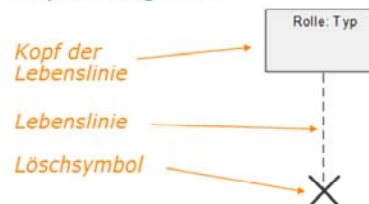
Lebenslinie

- Eine **Lebenslinie** beschreibt **genau einen Interaktionspartner**
- Als Interaktionspartner können alle Rollen des Kontext-Classifiers auftreten
 - Rollen sind vom Typ ConnectableElement (z.B. Klassen, Attribute oder Ports)

Metamodell



Notation im Sequenzdiagramm



Ein Interaktionspartner wird im Sequenzdiagramm durch eine **Lebenslinie** notiert, die aus einem Kopf und einer strichlierten Linie besteht.

Die **strichlierte Line** drückt im Prinzip aus, wie lange ein Objekt existiert, d.h. wir haben hier die Lebenszeit eines Objektes und eines Interaktionspartners ausgedrückt.

Im **Kopf** der Lebenslinie haben wir zwei Informationen stehen. Einerseits haben wir die Vollbezeichnung stehen und andererseits haben wir den Typ des entsprechenden Interaktionspartners stehen. Der **Typ** wird in unserem Fall meistens eine Klasse sein. Die **Rolle** bezeichnet auf welche Art und Weise, oder in welcher Form der Interaktionspartner an der ganzen Situation beteiligt ist – durch das Sequenzdiagramm drücken wir letztendlich Kommunikationssituationen aus. Und warum haben wir nicht hier einfach den Objektnamen angegeben? Die Verwendung des Rollenbegriffs ist eine viel allgemeinere Herangehensweise. Ein Objekt kann im Laufe seiner Lebenszeit viele unterschiedliche Rollen einnehmen. Wenn Sie jetzt hier nur eine konkretes Objekt darstellen würden, kann das vielleicht nicht unbedingt das auszudrücken, was Sie darstellen wollen. Denken wir an ein Beispiel: Sie haben eine Klasse vom Typ „Person“ und Sie haben eine konkrete Person, sagen wir „Hans Mayer“ und diese Person durchläuft eine Universitätskarriere. Zunächst studiert sie, dann arbeitet sie als Tutor, dann, wenn diese Person mit dem Studium fertig ist, wird sie Assistent, dann wird sie Professor usw. D.h. wir haben hier immer dieselbe Person, die aber immer unterschiedliche Rollen während ihrer Lebenszeit einnimmt. Und wenn wir hier jetzt eine konkrete Person stehen hätten, dann wären wir doch sehr eingeschränkt. Was wir mit der Rolle ausdrücken können ist ja z.B., dass es ein Professor sein muss, der ein Zeugnis ausstellt oder dass es ein Student sein muss, der eine Prüfung absolviert und vielleicht haben wir aber in unserem System aus irgendeinem Grund Studenten und Professoren nicht als eigenen Typ dargestellt und vielleicht wäre das viel zu sehr einschränkend, wenn wir hier einen Typ verwenden würden. Der Typ ist normalerweise an die Lebenszeit eines Objekts gebunden, oder anders gesagt: ein Objekt ändert während seiner Lebenszeit den Typ eher selten. Eine Rolle hingegen kann es aber ändern. Und dadurch sind wir, wenn wir hier nur eine Rolle angeben, viel flexibler als wenn wir uns durch einen Typ einschränken bzw. wenn wir uns nur auf eine konkrete Person beziehen.

Typ- vs. Instanzebene

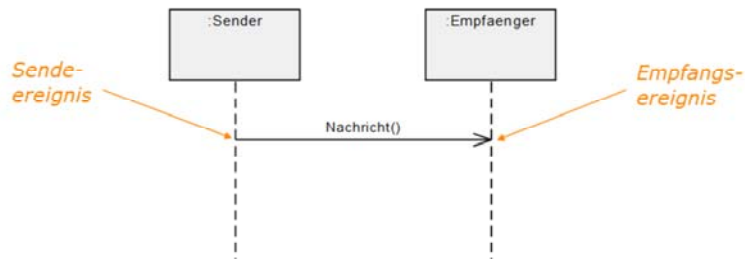
- Modellierung des Nachrichtenaustauschs zwischen Rollen und damit prinzipiell auf **Rollenebene**
 - Kontext der Interaktion durch strukturierte Classifier festgelegt = Kontext-Classifier
 - Die Rollen der Classifier stellen die Interaktionspartner dar
 - Tatsächliche Interaktion findet selbstverständlich auf Instanzebene zwischen Objekten statt
- Modellierung auf **Instanzebene** möglich, um eine Abfolge von Nachrichten zwischen konkreten Objekten darzustellen = Trace

Die Interaktion findet natürlich immer auf Instanz-Ebene statt. Auf Instanz-Ebene können Sie dann auch Abfolgen von Nachrichten zwischen konkreten Objekten darstellen. Diese Abfolgen werden als Traces bezeichnet.

Noch ein Hinweis zur Notation: Sie können entweder die Rollenbezeichnung oder die Typbezeichnung weglassen. Wenn Sie nur die Rollenbezeichnung hinschreiben, dann entfällt auch der Doppelpunkt ":". Wenn Sie die Rollenbezeichnung entfernen und nur den Typ hinschreiben wollen, dann müssen Sie auch den Doppelpunkt ":" hinschreiben. Also dann schreiben Sie z.B. ":Person".

Lebenslinie: Ereignisspezifikation (1/2)

- Interaktionen werden als **Folge von Ereignisspezifikationen** auf Lebenslinien betrachtet
- Beispiel für Ereignisspezifikationen
 - **Senden** und **Empfangen** von Nachrichten auf verschiedenen Lebenslinien oder der gleichen Lebenslinie

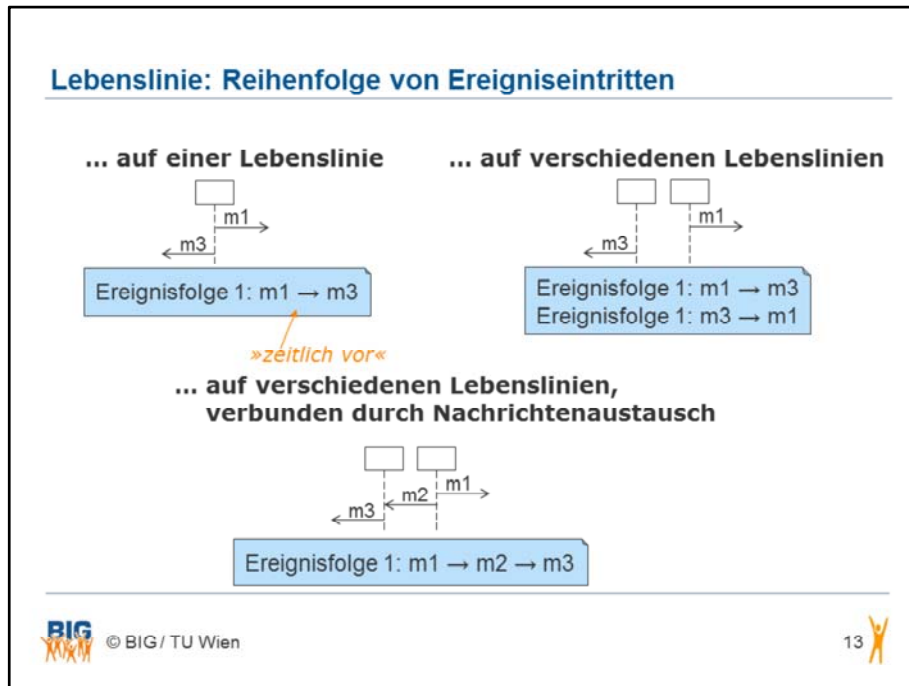


D.h. wir haben einmal ausgedrückt, wer die Interaktionspartner sind. Diese wollen jetzt miteinander etwas machen, d.h. sie wollen miteinander kommunizieren. Hier haben wir einen Sender und hier haben wir einen Empfänger und die tauschen miteinander Nachrichten aus. Das wird notiert, indem man zwischen dem Sender und dem Empfänger einem Pfeil vom Sender zum Empfänger einzeichnet. Auf der einen Seite, wo die Nachricht weggeht, ergibt sich dadurch ein Sendeereignis und hier wo der Pfeil hinkommt ein Empfangsereignis. Es muss aber nicht unbedingt sein, dass Sender und Empfänger unterschiedlich sind, d.h. es kann auch sein, dass Sender und Empfänger die gleiche Person sind. Der Nachrichtenaustausch wird also durch einen Pfeil zwischen Sender und dem Empfänger dargestellt.

Lebenslinie: Ereignisspezifikation (2/2)

- Reihenfolge von Ereignisspezifikationen
 - Vertikale Zeitachse bestimmt nur die Ordnung der Ereigniseintritte pro Lebenslinie
 - Jedoch nicht die Reihenfolge von Ereigniseintritten auf verschiedenen Lebenslinien
 - Erst durch **Nachrichten zwischen Lebenslinien** wird eine Ordnung über Lebenslinien hinweg erzwungen

Wie schon gesagt, verläuft im Sequenzdiagramm die Zeit von oben nach unten, d.h. Ereigniseintritte, die sich weiter oben befinden, finden im Allgemeinen vor Ereigniseintritten statt, die weiter unten stattfinden. Wie dann die konkrete Reihenfolge zwischen Ereignissen auf unterschiedlichen Lebenslinien aussieht, werden wir uns noch nachher an einem Beispiel genauer anschauen, aber prinzipiell ist es so: Wenn ich keinen Nachrichtenaustausch zwischen zwei Interaktionspartnern haben, dann ist die Reihenfolge der Ereignisse nicht festgelegt.



Hier haben wir Beispiele für die Reihenfolge von Ereigniseintritten.

Wir können unseren Nachrichtenaustausch mittels sogenannter „**Traces**“ darstellen, d.h. wir können quasi sequenzialisiert unsere Nachrichten darstellen.

Im ersten Beispiel haben wir eine Lebenslinie, eine Nachricht „m1“ und eine Nachricht „m3“. Wenn man die Reihenfolge jetzt als Trace darstellt können wir sagen, dass „m1“ vor „m3“ stattfindet, d.h. wir haben „m1“ → „m3“. Dieser Pfeil drückt aus, dass die linke Seite zeitlich vor der rechten Seite stattfindet.

Wenn wir uns jetzt das zweite Beispiel anschauen dann haben wir hier zwei Interaktionspartner. Beide senden Nachrichten. Prinzipiell ist hier nicht festgelegt in welcher Reihenfolge diese Nachrichten stattfinden, weil zwischen diesen beiden Interaktionspartnern kein Nachrichtenaustausch stattfindet. D.h. wir erhalten hier zwei mögliche Traces. Einerseits wäre es möglich, dass zuerst „m1“ und dann „m3“ stattfindet, oder es findet zuerst „m3“ und dann „m1“ statt.

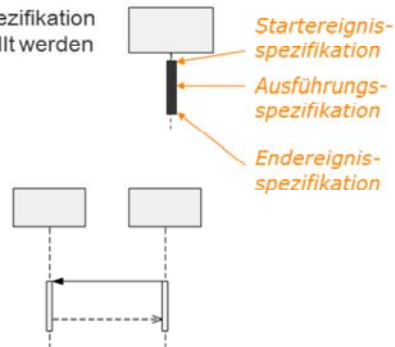
Und wenn wir jetzt noch eine Nachricht „m2“ einführen, die zwischen den zwei Interaktionspartnern ausgetauscht wird, dann haben wir jetzt damit aber eine Reihenfolge zwischen „m1“ und „m3“ festgelegt. Zuerst erfolgt die Nachricht „m1“, auch wenn die in irgendeine andere Richtung versendet wird und nicht zu dem dargestellten Interaktionspartner. Danach kommt die Nachricht „m2“ und zu guter Letzt die Nachricht „m3“.

Das sind die Regeln, die Sie sich merken müssen, wenn kein Sonderfall auftritt, der dann explizit im Diagramm dargestellt wird. Das ist die Default-Ordnung von Nachrichten im Sequenzdiagramm.

Noch einmal zusammengefasst: Auf einer Lebenslinie sind die Nachrichten von oben nach unten in ihrer Reihenfolge festgelegt. Zwischen unterschiedlichen Interaktionspartnern ergibt sich erst eine Reihenfolge, wenn zwischen diesen explizit eine Kommunikation stattfindet.

Lebenslinie: Ausführungsspezifikation

- Die **Ausführung** einer Aktivität/Operation wird durch zwei Ereignisspezifikationen (Start und Ende) auf der gleichen Lebenslinie definiert
 - Diese sogenannte Ausführungsspezifikation kann durch einen Balken dargestellt werden
- Ausführungsarten**
 - Direkt**
 - Interaktionspartner führt Verhalten selbst aus
 - Indirekt**
 - Ausführung wird an andere Interaktionspartner delegiert



Wir können im Sequenzdiagramm darstellen, wann Verhalten wirklich ausgeführt wird. Dieses Verhalten wird im Aktivitätsdiagramm in Form von Aktionen näher spezifiziert. Im Sequenzdiagramm visualisieren sogenannte **Ausführungsspezifikationen Verhalten in Form von Balken, die entlang der Lebenslinie verlaufen**. Diese Balken sind optional und dienen einfach zur Veranschaulichung wann Verhalten ausgeführt wird.

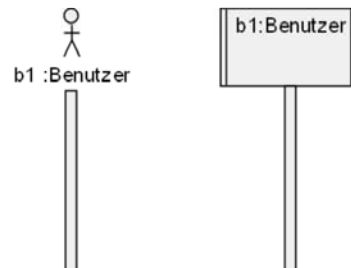
Es gibt zwei Arten von Balken. Ein ausgefüllter Balken bringt zum Ausdruck, dass **direkt** das Verhalten ausgeführt wird, d.h. der Interaktionspartner führt das Verhalten selbst aus, während bei der **indirekten** Ausführung die Darstellung durch einen nicht ausgefüllten Balken erfolgt. Hier wird die Ausführung an andere Interaktionspartner delegiert.

Lebenslinie: Aktives Objekt

- **Aktive Objekte** verfügen über **eigenen Kontrollfluss** (Prozess oder Thread)
- Können **unabhängig** von anderen Objekten operieren

- **Notation**

- Kopf der Lebenslinie wird links und rechts mit doppeltem Rand versehen
- durchgehender Balken über gesamte Lebenslinie



In einem System können aktive und passive Objekte auftreten.

Aktive Objekte sind Objekte, die einen eigenen Kontrollfluss haben, d.h. sie können unabhängig von anderen Objekten agieren. Menschen sind z.B. immer aktive Objekte, Threads sind aktive Objekte. Wenn Sie z.B. an ein Betriebssystem denken, dann können Sie ja viele Programme gleichzeitig starten und es ist jetzt mittlerweile nicht mehr so, dass ein Programm wartet bis das andere Programm fertig ist und erst dann fortfahren kann, sondern sie können quasi gleichzeitig ihre Tasks erledigen. Genau das kann ich durch aktive Objekte darstellen, d.h. ich habe meine Objekte, meine Interaktionspartner, die nicht davon abhängig sind, dass sie durch andere Objekte angestoßen werden. Wenn Sie an ganz normale Java-Objekte denken, die jetzt nicht speziell die Thread-Klasse erweitern, ist es ja auch so, dass diese nichts machen können, bevor ihre Methoden explizit aufgerufen werden, dass diese einfach keinen eigenen Kontrollfluss haben. Um darzustellen, dass Interaktionspartner einen eigenen Kontrollfluss haben, bietet das Sequenzdiagramm die Möglichkeit aktive Objekte explizit zu kennzeichnen. Das kann entweder durch die Angabe des Actor-Symbols geschehen, das Sie dann noch einmal im Usecasediagramm kennenlernen werden bzw. der Kopf der Lebenslinie kann durch die Angabe von doppelten Linien auf der vertikalen Seite explizit gekennzeichnet werden. Dadurch drücken wir aus, dass wir hier aktive Objekte haben. Oft wird zusätzlich noch ein durchgängiger Balken anstelle der strichlierten Linie bei einem aktiven Objekt eingezeichnet. Wann ist es notwendig das einzuzeichnen? Natürlich wenn Sie Personen haben, die haben wie gesagt immer einen eigenen Kontrollfluss. Aber Sie werden trotzdem sehr oft Sequenzdiagramme sehen, in denen auch menschliche Akteure nicht als aktive Objekte eingezeichnet sind. Die Frage ist: Ist das falsch? Nein, das ist in vielen Situationen nicht unbedingt falsch, weil Sie auch hier wieder wie vorhin bei der Verhaltensspezifikationen ausdrücken können oder explizit machen können, dass es in dem Kontext wichtig ist, dass Sie aktive Objekte haben. In manchen Situationen ist es nicht wichtig darzustellen, dass ein Mensch einen eigenen Kontrollfluss hat. Ein anderes Beispiel: Stellen Sie sich vor, Sie wollen einen Webserver implementieren. Sie realisieren diesen Webserver so, dass jedes Mal, wenn an den Webserver ein Request kommt ein Worker gestartet wird, der sich um den Request kümmert. Der Worker läuft als eigener Thread, damit unterschiedliche Anfragen quasi gleichzeitig bearbeitet werden können. Die Worker haben einen eigenen Kontrollfluss. Dies können Sie durch die Notation der aktiven Objekte ausdrücken. Wenn Sie aber diesen Aspekt nicht hervorheben wollen, dann werden Sie diese Notation nicht unbedingt verwenden.

Nachricht

Arten der Kommunikation

Synchrone Kommunikation

- Der Sender wartet bis zur Beendigung der Interaktion, die durch die Nachricht ausgelöst wurde

$m1(p1,p2)$ →

Asynchrone Kommunikation

- Die Nachricht wird als Signal betrachtet
- Der Sender wartet nicht auf das Ende der Interaktion

$m2$ →

Antwortnachricht (optional)

att: Name eines Attributs, dem der Rückgabewert zugewiesen werden soll

m1: Name der Nachricht, auf die geantwortet wird

wert: Rückgabewert

$att=m1:wert$
←-----

Jetzt kommen wir zu einem ganz wichtigen Punkt, nämlich zu den unterschiedlichen Nachrichtenarten, die durch unterschiedliche Pfeile ausgedrückt werden. Beim Sequenzdiagramm ist es extrem wichtig, dass wir die richtigen Arten von Pfeilen verwenden, weil Pfeile durch ihre Notation bzw. durch ihre verschiedenen Abwandlungen, unterschiedliche Aussagen tätigen.

Beim Sequenzdiagramm gibt es einen Pfeil, der eine durchgängige Linie hat, mit einer geschlossenen Pfeilspitze, einen durchgängigen Pfeil mit einer offenen Pfeilspitze und einen strichlierten Pfeil, wiederum mit einer offenen Pfeilspitze. Diese drei Pfeile haben unterschiedliche Bedeutungen und ich möchte Sie darauf hinweisen, dass Sie unbedingt darauf achten sollten, dass Sie den richtigen Pfeil verwenden, weil Sie ganz einfach durch die Art des Pfeiles eine bestimmte Aussage treffen.

Der erste Pfeil stellt **synchrone Nachrichten** dar. Hiermit können wir synchrone Kommunikation ausdrücken. Diese besagt, dass der Sender der Nachricht so lange wartet und nichts anderes macht, bis er vom Empfänger der Nachricht eine Antwortnachricht erhalten hat. Denken Sie z.B. an einen ganz normalen Methodenaufwurf in einem Programm. Ihr Programm wird normaler Weise nicht fortfahren mit der Ausführung, solange die Methode, die aufgerufen wurde, nicht retourniert ist.

Auch wenn Sie sich z.B. mit jemandem unterhalten, dann werden Sie nicht Ihre Aussage treffen und dann einmal etwas anderes machen und dann irgendwann nachschauen ob der Gesprächspartner vielleicht jetzt schon eine Antwort geliefert hat. Ein natürliches Gespräch ist üblicherweise auch ein typischer Fall von synchroner Kommunikation.

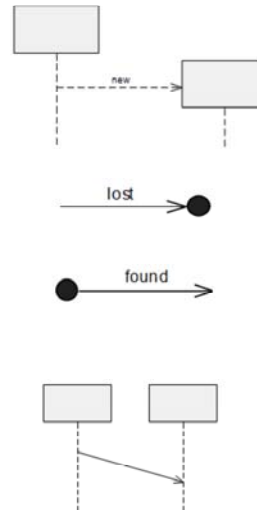
Im Gegensatz dazu wird die **asynchrone Kommunikation** ausgedrückt durch einen durchgängigen Pfeil mit einer offenen Pfeilspitze. Hier wartet der Sender der Nachricht nicht auf eine Antwort. Was wären hier Beispiele? Denken Sie z.B. an das Versenden von E-Mails: Sie senden Ihre E-Mail-Nachricht ab und warten aber nicht, ob sofort eine Antwort zurückkommt. Sie machen zunächst andere Dinge und schauen dann gelegentlich nach, ob schon eine Antwort gekommen ist, aber Sie warten nicht explizit auf die Antwort. Hier verwenden Sie asynchrone Kommunikation. Wenn Sie sofort eine Antwort haben wollen, würden Sie wahrscheinlich ein anderes Kommunikationsmittel verwenden, dann würden Sie wahrscheinlich ein Telefonat vorziehen. Hier haben Sie wiederum synchrone Kommunikation.

Bei der synchronen Kommunikation werden wie gesagt **Antwortnachrichten** erwartet und diese Antwortnachrichten werden durch einen strichlierten Pfeil mit offener Pfeilspitze dargestellt. D.h. wenn ich meine synchrone Nachricht absende, dann erhalte ich als Antwortnachricht eine Nachricht, die durch einen strichlierten Pfeil mit offener Pfeilspitze dargestellt ist. Wichtig ist, dass die Angabe dieser Antwortnachricht optional ist. Auch hier gebe ich diese wiederum nur dann an, wenn ich diese explizit darstellen will. Durch die Angabe des Pfeils für die synchrone Kommunikation drücke ich ja schon explizit aus, dass eine Antwortnachricht kommen muss. Wenn ich diese aber in meinem Diagramm noch explizit einzeichnen will, dann verwende ich die Notation der Antwortnachricht. Warum mache ich das? Ich kann meine Nachrichten beschriften und bei der Antwortnachricht kann ich angeben, was der Rückgabewert ist. Dazu gibt man zunächst an wie meine Nachricht heißen hat. In diesem Beispiel antworte ich auf die Nachricht „m1“. Ich gebe auch den Wert an, den ich zurückbekomme und dann kann ich noch eine Zuweisung tätigen. Ich kann in diesem Fall ausdrücken, dass auf die Nachricht „m1“ der Wert „wert“ zurückgegeben wird, und dieser Wert wird dem Attribut „att“ zugewiesen.

Überhaupt kann ich Nachrichten beschriften, das ist bei synchronen und asynchronen Nachrichten gleich. Ich gebe meinen Nachrichten einen Namen. In dem einen Fall ist es „m1“ und in dem anderen Fall ist es „m2“. Und wiederum ist es wie beim Methodenaufwurf möglich, Argumente mitzugeben und im Falle der hier dargestellten synchronen Kommunikation sind meine zwei Argumente „p1“ und „p2“.

Nachricht: Spezielle Nachrichtenarten

- **Objekterzeugung**
 - Ermöglicht, einen Interaktionspartner erst im Laufe der Interaktion zu erzeugen
- **Verlorene Nachricht**
 - Senden einer Nachricht an unbekannten oder nicht relevanten Interaktionspartner
- **Gefundene Nachricht**
 - Empfang einer Nachricht von einem unbekannten oder nicht relevanten Interaktionspartner
- **Zeitkonsumierende Übertragung**



17

Dann gibt es noch spezielle Arten von Nachrichten.

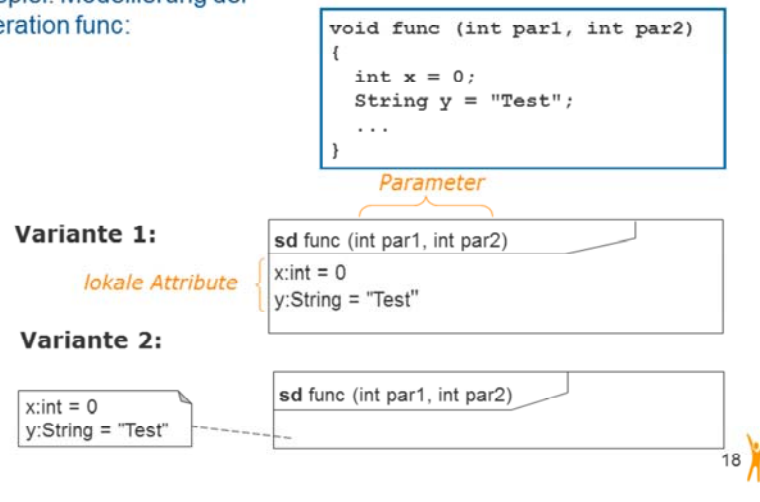
Ich habe Nachrichten mit denen ich in der Lage bin, **neue Objekte** zu erzeugen, d.h. es müssen in einem Sequenzdiagramm nicht alle Interaktionspartner von Anfang an eingezeichnet werden. Es kann sein, dass zur Laufzeit bzw. während der Ausführung neue Interaktionspartner erzeugt werden. Dies kann dadurch dargestellt werden, dass ein strichlierter Pfeil mit offener Pfeilspitze und dem Schlüsselwort „new“ direkt in den Kopf einer Lebenslinie hineingeht. Dann wird dieser Interaktionspartner im Zuge der Laufzeit erzeugt.

Dann gibt es noch die zwei Fälle, dass man den Sender bzw. den Empfänger der Nachricht nicht kennt. Wenn man den Empfänger nicht kennt, dann redet man von einer sogenannten **„lost“-Nachricht**. Das wird durch einen Pfeil mit einer durchgehenden Linie und offener Pfeilspitze dargestellt, die in einen Punkt hineingeht, d.h. ich kenne den Empfänger entweder nicht, oder es ist mir einfach nicht wichtig. Und analog dazu haben wir noch eine **„found“-Nachricht**. Hier kenne ich den Sender nicht, oder er ist mir egal, und drücke das wiederum durch so einen Punkt aus und aus diesem Punkt geht ein Pfeil hinaus mit einer durchgängigen Linie und einer offenen Pfeilspitze.

Was wir bis jetzt immer hatten war, dass die Nachrichtenübertragung keine Zeit kostet. Wenn Sie ausdrücken wollen, dass die **Übertragung der Nachricht auch Zeit benötigt** – es kann ja unter Umständen sein, dass z.B. ein Brief nicht sofort ankommt – dann drücken Sie das durch eine schräge Linie aus. D.h. das Sendeereignis ist weiter oben als das Empfangsereignis, d.h. die haben einen schrägen Pfeil für die Nachricht im Diagramm eingezeichnet.

Basiskonzepte – Parameter, lokale Attribute

- Darstellung von Parametern und lokalen Attributen
- Beispiel: Modellierung der Operation func:



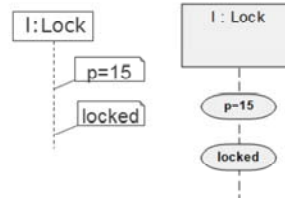
Weiters können wir in unserem Diagramm **Parameter** verwenden, die an unsere Interaktion an sich übergeben werden. Das ist im Prinzip wiederum sehr ähnlich, wie bei der Funktion in einer Programmiersprache. Wie bei allen UML-Diagrammen können wir das Sequenzdiagramm in so ein eckiges Kästchen schreiben, das oben ein Fünfeck aufweist mit dem Inhalt „sd“, was für Sequenzdiagramm steht. Dadurch kennzeichnen wir, dass es sich um ein Sequenzdiagramm handelt. Dann folgt der Name des Sequenzdiagrammes und dann können wir Parameter eingeben, die in der Interaktion zur Verfügung gestellt werden. Diese können dann innerhalb des Sequenzdiagramms ganz normal verwendet werden.

Wir können auch an beliebiger Stelle **lokale Attribute** definieren. Das wäre z.B. eine Möglichkeit um Ergebnisse von Methodenaufrufen zu speichern. Wir können dies aber nicht nur im Diagramm, sondern auch außerhalb des Diagrammes machen und zwar in Form einer Notiz, die dann durch eine strichlierte Linie mit dem Diagramm verbunden wird.

Zustandsinvariante

- **Zusicherung**, dass eine bestimmte **Bedingung** zu einem bestimmten Zeitpunkt **erfüllt** ist
- Bezieht sich immer auf eine bestimmte Lebenslinie
- Wird vor Eintritt des darauf folgenden Ereignisses ausgewertet
- Falls Zustandsinvariante nicht erfüllt ist - Fehler
- Notationsvarianten – Beispiel: Schloss (Lock)

Zustände, die ein Schloss annehmen kann:



Im Sequenzdiagramm gibt es **Zustandsinvarianten**. Vorhin haben wir die Zeiteinschränkungen kennengelernt, die erfüllt sein müssen, damit der Nachrichtenaustausch korrekt abläuft. Wir können aber auch Einschränkungen angeben, die sich nicht unbedingt auf die Zeit beziehen, sondern auf den Zustand der Interaktionspartner. Wenn diese Bedingungen nicht erfüllt sind, dann tritt ein Fehler in der Ausführung auf, d.h. es wird dann nicht mit der Programmausführung fortgefahren.

Hier haben wir das Beispiel eines Schlosses. Für das Schloss gibt es ein Zustandsdiagramm. Zustandsdiagramme werden Sie in der nächsten Einheit kennenlernen. Das Schloss kann sich in den Zuständen „geschlossen“ und „offen“ befinden. Abhängig von Ereigniseintritten und dem aktuellen Zustand, kann es vom Zustand „geschlossen“ in „offen“ übergehen, wenn es aufgesperrt wird und von „offen“ in „verschlossen“, wenn es wieder zugesperrt wird.

Im Interaktionsdiagramm ist das Schloss ein Interaktionspartner und kann sich eben in einem der beiden Zustände befinden. Hier haben wir die Zustandsinvariante abgebildet, dass sich zu diesem Zeitpunkt, den ich hier angegeben habe, das Schloss im Zustand „verschlossen“ befinden muss. Das macht z.B. dann Sinn wenn ich eine Nachricht „öffnen“ erwarte. Dann möchte ich vielleicht zusichern, dass das Schloss „geschlossen“ ist, weil sonst die Nachricht „öffnen“ keinen Sinn machen würde. Außerdem haben wir angegeben, dass die Variable „p“ – die vorher definiert sein muss – den Wert „15“ haben muss. Hat diese Variable „p“ nicht den Wert „15“, dann liegt hier wiederum eine Fehlersituation vor und es ist nicht definiert, wie sich das System dann verhalten soll, d.h. die Kommunikation ist in diesem Fall nicht möglich.

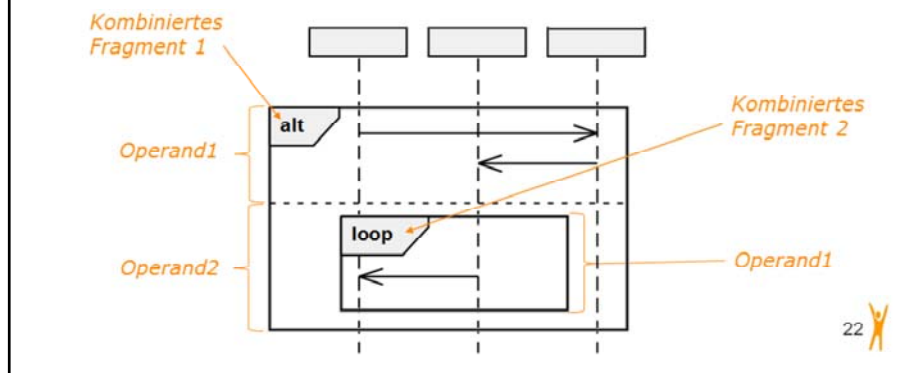
Kombinierte Fragmente

- Modellierung von **Kontrollstrukturen**
- Bestandteile: Operator und Operanden
- **Operator**
 - Definiert Art des kombinierten Fragments
 - 12 vordefinierte Operatoren
- **Operand**
 - Ein Operator enthält 1 oder mehrere Operanden, je nach Operatorart
 - Kann Interaktionen, kombinierte Fragmente (Schachtelung!) und Referenzen auf Sequenzdiagramme umfassen

Ihnen wird aufgefallen sein, dass wir bisher kaum Kontrollstrukturen hatten. D.h. wir haben den Nachrichtenaustausch bisher nur sehr eingeschränkt kontrollieren können. Um eine bessere Kontrolle über den Nachrichtenaustausch zu erhalten, wurden in UML 2 die sogenannten kombinierten Fragmente eingeführt. Insgesamt gibt es davon 12. Wir haben hier 12 Operatoren, die eben auf den Ablauf des Sequenzdiagramms Einfluss nehmen können und diese Operatoren setzen sich wiederum aus mindestens einem oder auch aus mehreren Operanden zusammen. Wichtig ist auch, dass kombinierte Fragmente wiederum kombinierte Fragmente enthalten können, d.h. diese können beliebig geschachtelt werden.

Kombinierte Fragmente – Notation

- Kombiniertes Fragment wird wie Sequenzdiagramm mit Rahmen dargestellt
- Art des Fragments wird durch Operator im Pentagon festgelegt
 - default: seq
- Operanden werden durch gestrichelte Linien voneinander getrennt



Hier haben wir ein Beispiel für ein kombiniertes Fragment und zwar das „alt“-Fragment. Wir werden uns nachher noch genauer anschauen, was es bedeuten. „alt“ ist der **Operator** und das Fragment besitzt zwei **Operanden**. Operanden werden durch eine strichlierte Linie ausgedrückt, d.h. die strichlierte Linie drückt eine Operandengrenze aus. Der zweite Operand des „alt“-Fragments ist ein „loop“-Fragment. Mit dem „loop“-Operator kann eine Schleife realisiert werden und er besitzt nur einen Operanden.

Per **Default** ist der Operator eines kombinierten Fragments im Sequenzdiagramm „seq“. Wenn im Pentagon links oben im Rahmen also nichts angegeben ist, handelt es sich um den „seq“-Operator. Auch insgesamt gilt per Default die „seq“-Ordnung im Sequenzdiagramm, die wir uns dann gleich noch einmal anschauen werden.

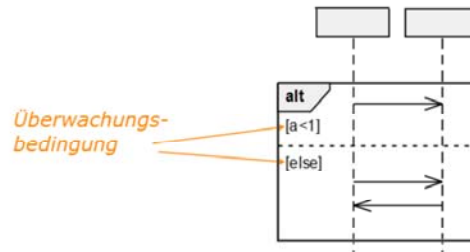
Kombinierte Fragmente – Operatorarten

| | Operator | Zweck |
|-------------------------------|----------|---|
| Verzweigungen und Schleifen | alt | Alternative Interaktionen |
| | opt | Optionale Interaktionen |
| | break | Ausnahme Interaktionen |
| | loop | Iterative Interaktionen |
| Nebenläufigkeit und Ordnung | seq | Sequentielle Interaktionen mit schwacher Ordnung (Default-Operator) |
| | strict | Sequentielle Interaktionen mit strenger Ordnung |
| | par | Nebenläufige Interaktionen |
| | critical | Atomare Interaktionen |
| Filterungen und Zusicherungen | ignore | Irrelevante Interaktionen |
| | consider | Relevante Interaktionen |
| | assert | Zugesicherte Interaktionen |
| | neg | Ungültige Interaktionen |

In dieser Tabelle sind die 12 Operatoren dargestellt. Diese können in drei Gruppen eingeteilt werden. Wir haben einerseits die „Verzweigung und Schleifen“. Sie sind sehr ähnlich wie die Konstrukte, die sie aus Programmiersprachen kennen. Wir haben „Parallelität und Ordnung“. Mit diesen kann die Reihenfolge von Nachrichten beeinflusst werden. Und dann gibt es noch die Gruppe „Filterungen und Zusicherungen“. Die kombinierten Fragmente dieser Gruppe geben Hinweise darauf, worauf bei der praktischen Umsetzung, bei der Realisierung und bei der Implementierung konkret zu achten ist.

Verzweigungen und Schleifen: alt-Operator

- Darstellung von zwei oder mehreren **alternativen Interaktionsabläufen** (mind. 2)
- Zur Laufzeit wird maximal ein Operand ausgeführt
- Auswahl eines Operanden anhand von Überwachungsbedingungen
 - Boolescher Ausdruck in eckigen Klammern
 - Vordefinierte else-Bedingung: Operand wird ausgeführt, falls die Bedingungen aller anderen Operanden nicht erfüllt sind
 - default: true



24

Wir beginnen mit den „Verzweigungen und Schleifen“.

In dieser Gruppe gibt es zunächst den „alt“-Operator. Der „alt“-Operator hat mindestens zwei **Operanden**. Im Prinzip entspricht der „alt“-Operator, wenn er zwei Operanden hat, einem „if then else“-Konstrukt aus einer Programmiersprache. Sie geben beim jedem Operanden eine **Bedingung** an. Eine Bedingung ist immer ein Boolescher Ausdruck, d.h. ein Ausdruck, der zu „wahr“ oder „falsch“ evaluiert. Wenn er zu „wahr“ evaluiert, dann wird der Inhalt des Operanden ausgeführt. Es gibt auch die spezielle Überwachungsbedingung „else“. Der Inhalt des Operanden mit der „else“-Bedingung wird genau dann ausgeführt, wenn keine andere Bedingung zutrifft.

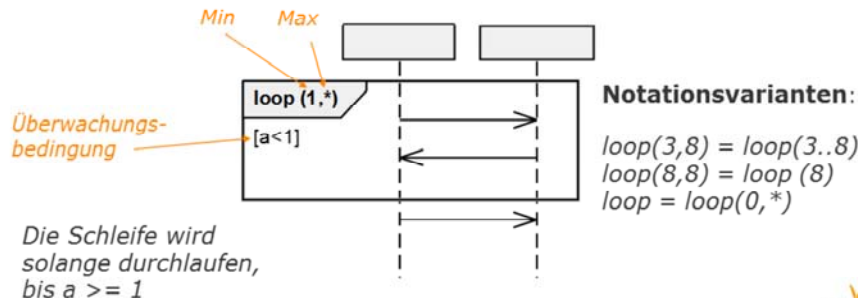
Achtung: Wenn Sie keine Überwachungsbedingung angeben, dann ist per **Default „true“** anzunehmen.

Wichtig ist noch, dass wenn Sie Überwachungsbedingungen haben, die sich nicht gegenseitig ausschließen, die Semantik nicht so ist, dass von oben nach unten die Ausführung erfolgt, sondern, dass Sie einen **Indeterminismus** haben, d.h. Sie haben nicht klar spezifiziert, welcher Operand wirklich ausgeführt wird.

Tafelzeichnung: Haben sie also z.B. einen „alt“-Operator mit einem Operanden mit der Bedingung „x=1“ und einem zweiten mit der Bedingung „x>0“, dann sind wenn „x=1“ beide Bedingung erfüllt. Wenn Sie die selbe Situation in einem Case-Statement hätten, dann würden Sie das Ganze von oben nach unten abarbeiten. In dem Fall des „alt“-Operators haben Sie aber wie gesagt einen Indeterminismus, d.h. Sie haben hier nicht klar spezifiziert, welcher dieser zwei Operanden ausgeführt werden sollen. Solche Überlappungen sollten also vermieden werden.

Verzweigungen u. Schleifen: loop-Operator

- Darstellung einer **Schleife** über einen bestimmten Interaktionsablauf
 - Fragment enthält nur einen Operanden
 - Ausführungshäufigkeit wird durch Zähler mit Unter- und Obergrenze dargestellt
 - Optional: Überwachungsbedingung; wird bei jedem Durchlauf überprüft, sobald die minimale Anzahl an Durchläufen stattgefunden hat



25

Der „loop“-Operator ermöglicht es, Iterationen oder Wiederholungen auszudrücken. Wiederum gibt es nur einen **Operanden**. Dieser eine Operand enthält genau die Interaktionen, die wiederholt werden sollen. Der „loop“-Operator kann **Argumente** haben. Diese Argumente drücken aus, wie oft die Schleife mindestens durchlaufen werden soll und wie oft sie maximal durchlaufen werden soll. Also Achtung: Wir haben hier nicht wie bei der for-Schleife in einer Programmiersprache eine Laufvariable, sondern wir haben hier eine Grenze für die Mindestanzahl der Durchläufe und eine Grenze für Maximalanzahl der Durchläufe.

Das Argument (1,*) gibt also dann, dass die Schleife mindestens einmal durchlaufen wird und der Stern sagt dann wieder wie beim Klassendiagramm bei den Multiplizitäten, dass hier keine Begrenzung nach oben hin gegeben ist. Was wir aber schon haben ist wiederum die Möglichkeit eine Überwachungsbedingung anzugeben. Die Schleife wird nur so lange ausgeführt, solange die Überwachungsbedingung erfüllt ist. Wenn die Überwachungsbedingung nicht mehr erfüllt ist, und ich die Schleife der Mindestanzahl entsprechend oft ausgeführt wurde dann wird die Schleife abgebrochen.

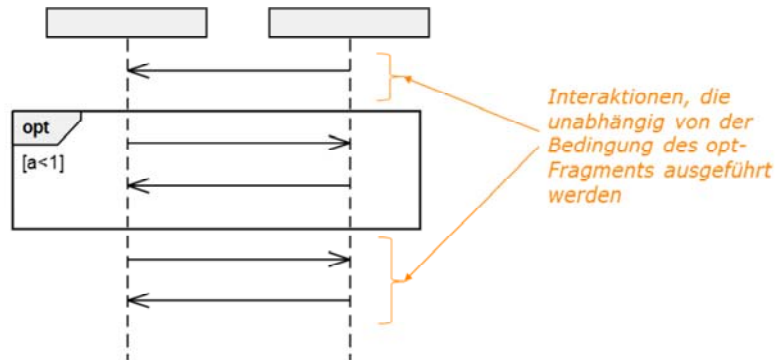
D.h. im Beispiel von loop(3,8), dass die Schleife mindestens 3mal ausgeführt wird, wenn ich sie das 4. Mal ausführen möchte, wird vorher die Überwachungsbedingung überprüft. Wenn die Bedingung „wahr“ ist, dann wird sie auch das 4. Mal ausgeführt. Genau so beim 5., 6. und 7. Mal. Ein 8. Mal kann ich sie auch noch ausführen. Wenn ich jetzt aber die Bedingung immer noch auf „wahr“ gesetzt habe, dann führe ich die Schleife dennoch nicht ein neuntes Mal aus, weil ja 8 die Obergrenze ist.

Ganz wichtig: Ich hab hier die Untergrenze und die Obergrenze, die die Anzahl der Schleifendurchläufe definieren. Hier wiederum kann es unterschiedliche Notationsvarianten geben. Man kann die Untergrenze und die Obergrenze durch einen Beistrich trennen oder durch zwei Punkte. Man kann auch für die Untergrenze und für die Obergrenze den selben Wert angeben. Das bedeutet dann, dass Mindest- und Maximalanzahl gleich sind, d.h. ich weiß genau wie oft meine Schleife ausgeführt werden soll. Im Fall von loop(8,8) führe ich die Schleife genau 8mal aus. Das kann ich entweder durch „8,8“ oder durch „8..8“ ausdrücken, oder ich gebe einfach nur 8 an. D.h. wenn ich die genaue Anzahl der Schleifendurchläufe weiß, dann gebe ich eine konkrete Zahl an. Wichtig ist, dass in dem Fall die Überwachungsbedingung natürlich obsolet ist, weil ich ja hier genau annehme, wie oft die Schleife durchlaufen wird. Die letzte Notationsvariante ist die, dass ich gar keine Parameter angegeben habe, das entspricht im Prinzip der Eingabe von „0..*“, d.h. ich führe die Schleife vielleicht gar nicht aus, abhängig von der Bedingung, oder ich kann sie beliebig oft ausführen, was wiederum abhängig von der Überwachungsbedingung ist. Ist keine Überwachungsbedingung gegeben, dann wird per Default „true“ angenommen. Das ist genauso, wie bei den „alt“- , „break“- oder „opt“-Operanden.

Was wir jetzt hier nicht haben, ist eine Laufvariable. Was ich allerdings machen kann ist, ich kann durch einen Kommentar angeben, dass es einen Zähler gibt und diesen pro Schleifendurchlauf erhöhen. Auch hier wiederum ist es wichtig anzumerken, dass es uns beim Sequenzdiagramm nicht darum geht, wie etwas konkret realisiert wird, sondern wie der Nachrichtenaustausch erfolgt.

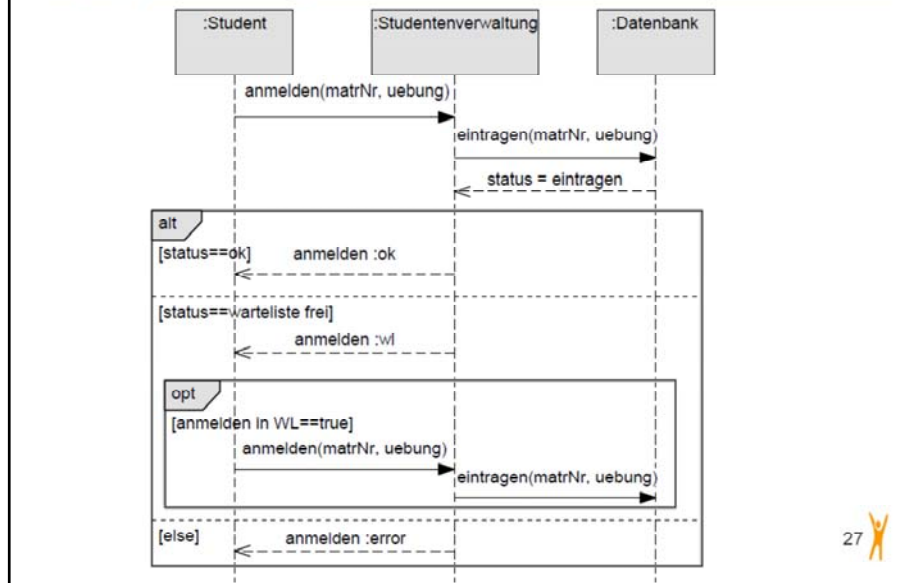
Verzweigungen u. Schleifen: opt-Operator

- **Optionale** Interaktionen
- Überwachungsbedingung steuert Durchlauf der Interaktionen
- Fragment wird nur aktiv, wenn Bedingung erfüllt ist
 - Modellierung von "wenn ..., dann ..."



Dann schauen wir uns als nächstes den „opt“-Operator an. Der „opt“-Operator ist im Prinzip ein „alt“-Operator mit nur einem **Operanden**. Hier haben Sie also nicht die Möglichkeit Alternativen auszudrücken, sondern Sie können hier nur angeben, dass wenn eine bestimmte **Bedingung** gilt, der Operand ausgeführt wird und wenn sie nicht gilt, dann soll er einfach ausgelassen werden. Im Prinzip entspricht diese Darstellung also einem „alt“-Operator mit zwei Operanden, wobei der zweite Operand ein „else“ enthält und der Inhalt leer ist. Das wäre eine alternative Darstellungsweise des Operators.

Opt-Operator - Beispiel



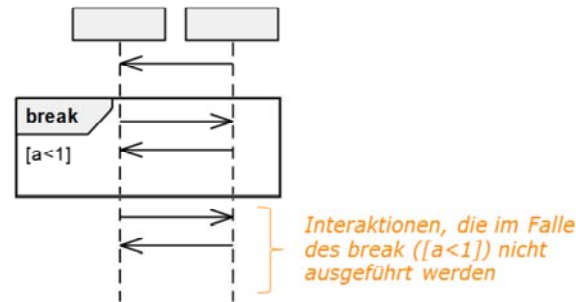
27 

Hier haben wir ein Beispiel: Ein Student möchte sich für eine Übung anmelden. Die Anmeldung erfolgt über das Studentenverwaltungssystem. Bei erfolgreicher Anmeldung wird der Student in der Datenbank eingetragen.

Wenn dies nicht möglich ist, aber noch Plätze in der Warteliste frei sind, so kann sich der Student in die Warteliste eintragen, sofern er dies will. Dies wird durch den `opt`-Operator ausgedrückt. Ansonsten erhält er eine Fehlermeldung.

Verzweigungen u. Schleifen: break-Operator

- Ausnahme-Interaktionen
- Überwachungsbedingung steuert Durchlauf der Interaktionen
- Behandlung von Sonderfällen und Ausnahmen

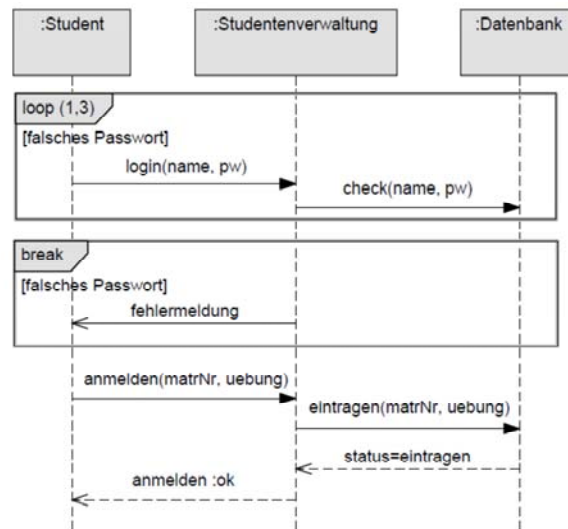


Nun zum „break“-Operator. Der „break“-Operator sieht so ähnlich aus, wie der „opt“-Operator, d.h. wir haben hier nur einen **Operanden** und nur eine **Bedingung**, er hat aber eine ganz andere Bedeutung. Wenn die angegebene Bedingung „wahr“ ist, dann wird – wie beim „opt“-Operator – der Inhalt des einzigen Operanden ausgeführt. Es wird aber nicht nur der Inhalt des einzigen Operanden ausgeführt, sondern es wird der folgende Teil der Interaktion dann nicht ausgeführt. D.h., wenn die angegebene Bedingung wahr ist, wird der Inhalt des Operanden zwar ausgeführt, aber die Interaktionen danach werden nicht mehr ausgeführt.

Wenn Sie jetzt an eine objektorientierte Sprache denken wie Java, dann wird Sie das an ein sehr einfaches **Exception-Handling-Konzept** erinnern. Wichtig ist, dass einfach die Interaktionen, die in dem zu schließenden kombinierten Fragment enthalten sind, nicht mehr ausgeführt werden, d.h. es wird eine Ebene hinaufgesprungen.

D.h. durch das „break“ ist es möglich, eine einfache Art des Exception-Handlings zu realisieren.

Break-Operator - Beispiel



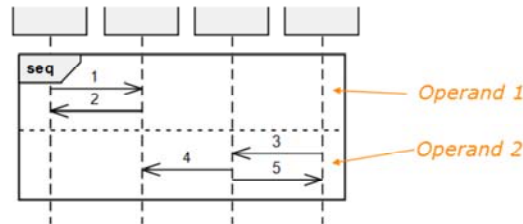
29



In diesem Sequenzdiagramm modellieren wir, dass ein Student sich zuerst einloggen muss, bevor er sich für eine Übung anmelden kann. Allerdings darf er seine Benutzerdaten nur 3 Mal falsch eingeben, dann kommt eine Fehlermeldung. Hat er sein Passwort nicht falsch angegeben, wird er für die Übung angemeldet.

Nebenläufigkeit u. Ordnung: seq-Operator

- Sequentielle Interaktion mit schwacher Ordnung (default!)
- mind. 1 Operand
- Reihenfolge der Ereigniseintritte:
 - Reihenfolge der Ereignisse pro Lebenslinie gilt über Operandengrenze hinaus (Reihenfolge der Operanden im Diagramm ist relevant)
 - Reihenfolge auf unterschiedlichen Lebenslinien von unterschiedlichen Operanden ist nicht signifikant
 - Reihenfolge auf unterschiedlichen Lebenslinien in einem Operanden ist nur signifikant, wenn hier ein Nachrichtenaustausch stattfindet



30

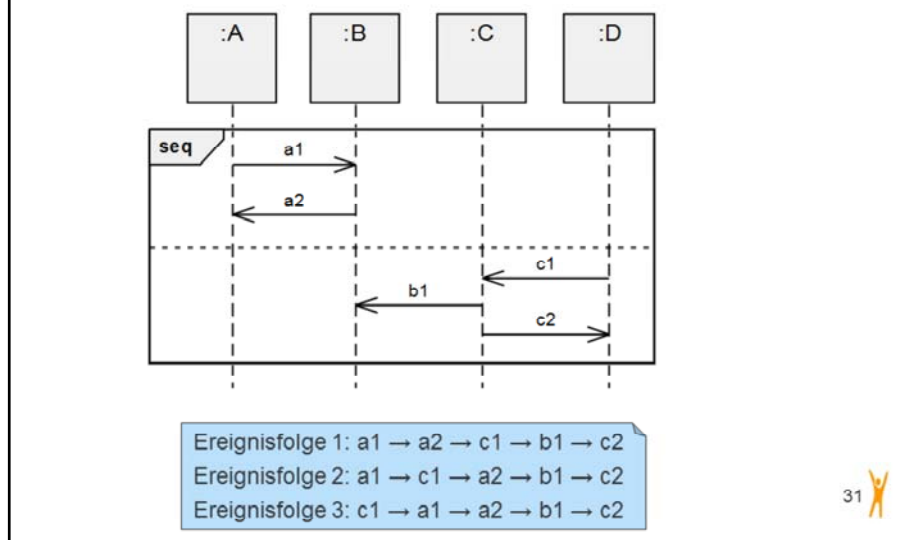
Dann kommen wir zu der zweiten Gruppe: Hier geht es um Nebenläufigkeit und Ordnung. Wir haben vorhin schon gesagt, dass wir im Sequenzdiagramm eine Ordnung zwischen den Nachrichten haben. Wenn Ereignisse auf derselben Lebenslinie stattfinden, dann ist die Ordnung von oben nach unten zu sehen. Wenn wir diese Ordnung kontrollieren wollen, dann bieten uns die Operatoren aus der Gruppe „Nebenläufigkeit und Ordnung“ die Möglichkeit dazu.

Der „seq“-Operator stellt die Default-Ordnung dar. Er besitzt mindestens einen Operanden. Die Reihenfolge der Ereignisse pro Lebenslinie gilt über die Operanden-Grenzen hinaus. D.h. wenn Sie in unterschiedlichen Operanden Ereignisse haben und diese befinden sich auf der selben Lebenslinie, dann ist auch hier wieder die Ordnung von oben nach unten zu sehen. Wenn Sie auf unterschiedlichen Lebenslinien Ereignisse von unterschiedlichen Operanden haben, dann ist die Reihenfolge nicht signifikant, wie wir vorher schon besprochen haben. Die Reihenfolge auf unterschiedlichen Lebenslinien in einem Operanden ist nur dann signifikant, wenn zwischen diesen ein Nachrichtenaustausch stattfindet. Das ist im Prinzip das, was im Beispiel durch die Nachricht 2 bzw. durch die Nachricht 4 erreicht wird.

Warum gibt es überhaupt unterschiedliche Operanden? Durch die Operanden hat man die Möglichkeit die Nachrichten besser zu gruppieren und besser zu ordnen.

Warum braucht man die „seq“-Ordnung, wenn das ohnehin schon die Default- Ordnung ist? Wir werden gleich sehen, dass es noch andere Operatoren gibt, mit denen man diese Ordnung aufheben kann. Will man dann aber in dem Bereich, für den die „seq“-Ordnung aufgehoben wurde, diese für einen Teilbereich wieder herstellen, so das kann mit dem „seq“-Operator gemacht werden.

Nebenläufigkeit u. Ordnung: seq-Operator - Beispiel



Hier haben wir ein Beispiel zur „seq“-Ordnung: Es gibt 4 Interaktionspartner und 5 Nachrichten, Nachricht „a1“, „a2“, Nachricht „c1“, „c2“ und die Nachricht „b1“. Es ergeben sich durch dieses Sequenzdiagramm folgenden Traces:

„a1“ muss immer vor „a2“ sein, weil wir ja hier Ereignisse auf derselben Lebenslinie haben.

Allerdings ist „c1“ unabhängig von „a1“ und „a2“, d.h. es kann natürlich nach „a1“ und „a2“ sein, aber ich kann es auch vorziehen. D.h. ein möglicher Trace wäre auch „a1“, „c1“, „a2“ oder „c1“ ganz vorne: „c1“, „a1“, „a2“. Eben weil hier keine Beziehungen zwischen den Interaktionspartnern bestehen, habe ich die Möglichkeit „c1“ beliebig mit „a1“ und „a2“ zu vertauschen.

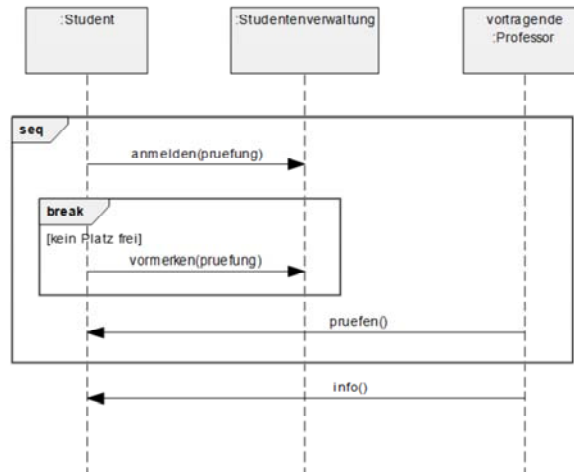
Anders hingegen verhält es sich mit „b1“. Mit „b1“ habe ich ja den gemeinsamen Interaktionspartner „B“ und deswegen darf ich „b1“ und „a1“ bzw. „b1“ und „a2“ nicht vertauschen.

Außerdem darf ich „b1“ und „c2“ nicht vertauschen, weil ich hier den gemeinsamen Interaktionspartner habe und daraus ergibt sich eben die Reihenfolge „c1“, „b1“, „c2“.

Das „b1“ muss aber hinter dem „a2“ sein. So haben wir die Reihenfolge „a2“, „b1“, „c2“ in allen Traces.

Das sind im Prinzip die Möglichkeiten, die ich hier mit dem „seq“-Operator habe.

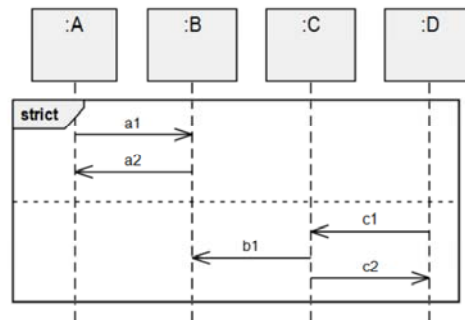
Nebenläufigkeit u. Ordnung: seq-Operator – Beispiel (2/2)



In diesem Beispiel wird der seq-Operator für die Gruppierung von Nachrichten verwendet. Wenn die Bedingung des break-Operators erfüllt ist, wird die Methode vormerken() ausgeführt, die Methode pruefen() aber nicht, info() aber schon wieder.

Nebenläufigkeit und Ordnung: strict-Operator

- Sequentielle Interaktion mit **strenger** Ordnung
- Reihenfolge auf unterschiedlichen Lebenslinien von unterschiedlichen Operanden ist signifikant

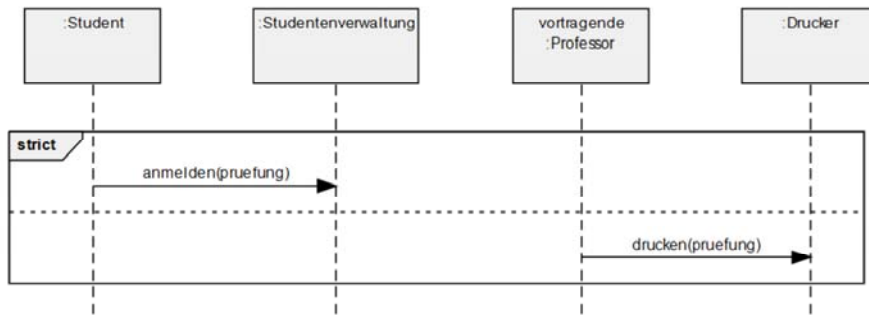


Ereignisfolge 1: $a1 \rightarrow a2 \rightarrow c1 \rightarrow b1 \rightarrow c2$

33 

Wenn ich jetzt aber sagen will, dass mir auch über die Lebenslinie hinaus die Ordnung wichtig ist, dann kann ich das durch die Verwendung des „strict“-Operators machen. Dann habe ich, wenn ich in unserem Beispiel von gerade „strict“ angebe, nur eine Möglichkeit einen Trace zu bilden und zwar „a1“, „a2“ dann kommt „c1“, dann kommt „b1“ und dann kommt „c2“. D.h. hier hab ich jetzt nicht mehr die Freiheit, Nachrichten beliebig anzuordnen, wenn zwischen den Interaktionspartner keine Kommunikationsbeziehung besteht.

Strict-Operator - Beispiel

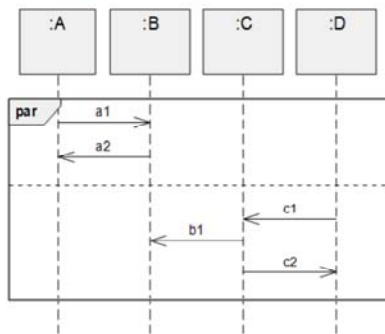


Hier haben wir ein Beispiel für strict. Eine Prüfung wird nur dann ausgedruckt, nachdem sich ein Student angemeldet hat. Es kann nicht sein, dass zuerst eine Prüfung gedruckt wird und sich dann der Student anmeldet.

Nebenläufigkeit und Ordnung: par-Operator

■ Nebenläufige Interaktionen

- Lokale Reihenfolge pro Operand muss erhalten bleiben
- Reihenfolge der Operanden im Diagramm ist irrelevant!
- mind. 2 Operanden



Ereignisf. 1: $a1 \rightarrow a2 \rightarrow c1 \rightarrow b1 \rightarrow c2$
Ereignisf. 2: $a1 \rightarrow c1 \rightarrow a2 \rightarrow b1 \rightarrow c2$
Ereignisf. 3: $a1 \rightarrow c1 \rightarrow b1 \rightarrow a2 \rightarrow c2$
Ereignisf. 4: $a1 \rightarrow c1 \rightarrow b1 \rightarrow c2 \rightarrow a2$
Ereignisf. 5: $c1 \rightarrow a1 \rightarrow a2 \rightarrow b1 \rightarrow c2$
Ereignisf. 6: $c1 \rightarrow a1 \rightarrow b1 \rightarrow a2 \rightarrow c2$
Ereignisf. 7: $c1 \rightarrow a1 \rightarrow b1 \rightarrow c2 \rightarrow a2$
Ereignisf. 8: $c1 \rightarrow b1 \rightarrow a1 \rightarrow a2 \rightarrow c2$
Ereignisf. 9: $c1 \rightarrow b1 \rightarrow a1 \rightarrow c2 \rightarrow a2$
Ereignisf. 10: $c1 \rightarrow b1 \rightarrow c2 \rightarrow a1 \rightarrow a2$



© BIG / TU Wien

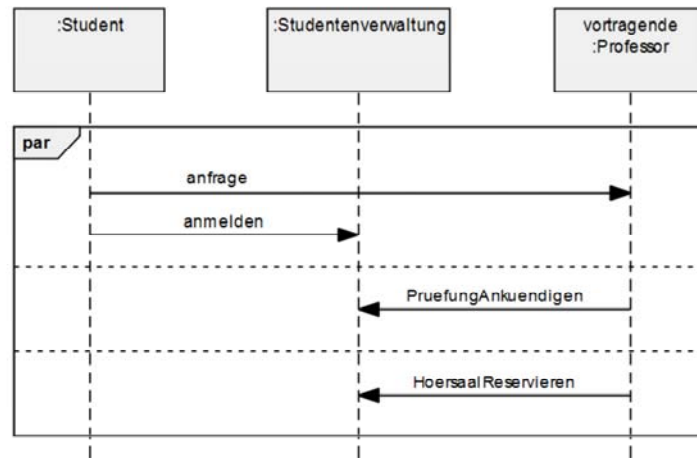
35



Dann haben wir die Möglichkeit, dass wir überhaupt die zeitliche Abfolge aufheben. Diese Möglichkeit haben wir durch den „par“-Operator. Hier haben wir mindestens zwei Operanden. Innerhalb der Operanden gibt es lokale Zeitachsen für die wieder die normale „seq“-Ordnung gilt, d.h. „a1“ kommt immer vor „a2“ im ersten Operanden und „c1“ kommt immer vor „b1“ und „b1“ immer vor „c2“ im zweiten Operanden. Allerdings besteht jetzt kein zeitlicher Zusammenhang mehr zwischen den beiden Operanden. D.h. es ist jetzt nicht mehr festgelegt, dass z.B. „b1“ nach „a1“ oder „a2“ kommen muss. „b1“ kann hier beliebig eingeordnet werden, genauso „c1“ und „c2“.

D.h. lokal bleiben Zeit und Ordnung erhalten, es gibt also eine lokale Zeitachse, aber die globale Zeitachse ist insgesamt aufgehoben.

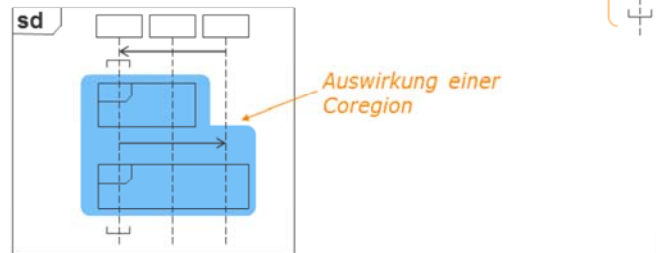
Par-Operator - Beispiel



Dieses Beispiel drückt aus, dass die Reihenfolge, in der ein Professor Studentenanfragen beantwortet, Prüfungen ankündigt und Hörsäle reserviert, nicht fix festgelegt ist. Ein Student kann sich allerdings nur nach einer Anfrage an den Professor anmelden.

Nebenläufigkeit und Ordnung: Coregion

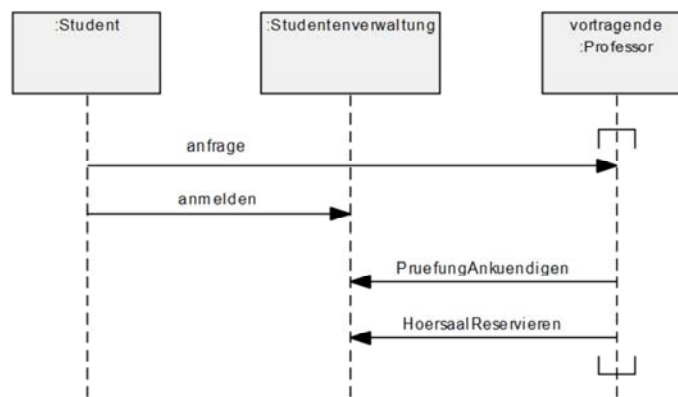
- **Coregion:** Darstellung von nebenläufigen Abläufen auf EINER Lebenslinie
- Reihenfolge der Ereigniseintritte innerhalb von Coregions ist auf keine Weise beschränkt ("Aufhebung der Zeitdimension")
- Coregion kann weitere kombinierte Fragmente beinhalten – kombinierte Fragmente können als Ganzes in bel. Reihenfolge ausgeführt werden



Wenn ich nicht für mehrere Interaktionspartner die Zeitdimension aufheben möchte, sondern nur für einen Interaktionspartner, dann kann man das durch sogenannte Co-Regionen machen. Co-Regionen werden einfach durch umgedrehte, eckige Klammern ausgedrückt, und der Teil, der von diesen eckigen Klammern umschlossen wird, ist keiner „seq“-Ordnung mehr unterworfen. D.h. alle Ereignisse, die in diesem Bereich auftreten, sind an keine Reihenfolge mehr gebunden und natürlich propagiert sich das dann an die entsprechenden Interaktionspartner fort, d.h. wenn ich hier zwei Sendeereignisse habe, dann sind diese natürlich auch nicht mehr an eine gewisse Zeitfolge gebunden. Co-Regionen können wiederum ganze kombinierte Fragmente enthalten und hier gilt wiederum, dass diese dann auch beliebig angeordnet werden können. Innerhalb der kombinierten Fragmente muss dann wiederum die Ordnung beachtet werden.

Durch eine Co-Region ist es also möglich, für einen Interaktionspartner die Zeitdimension aufzuheben.

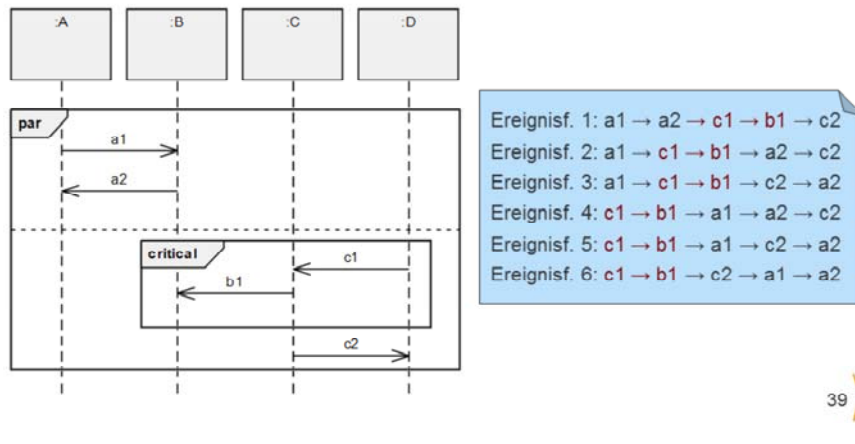
Coregion – Beispiel



Hier ist das vorige Beispiel mit einer Co-Region modelliert, was genau den selben Effekt erzielt.

Nebenläufigkeit und Ordnung: critical-Operator

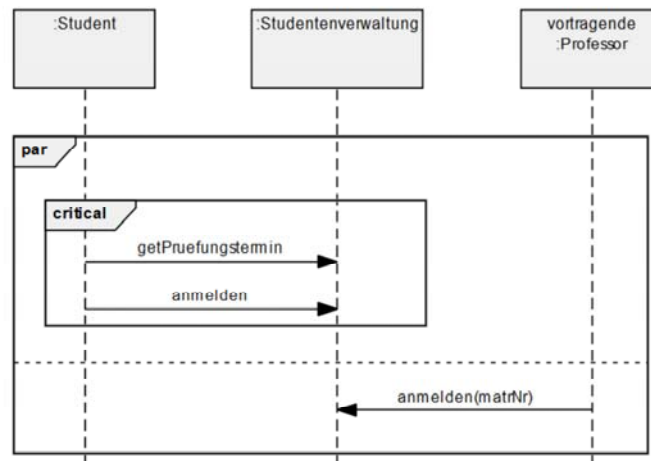
- Kritischer Bereich: atomarer (nicht unterbrechbarer) Interaktionsablauf
- Keine Beschränkung auf Interaktionen außerhalb des kritischen Bereichs



Dann gibt es noch den „critical“-Operator. Der „critical“-Operator ermöglicht eine Menge von Nachrichten als nicht unterbrechbar zu kennzeichnen. D.h. in diesem Beispiel ist es nicht möglich, dass „c1“ und „b1“ durch irgendwelche anderen Nachrichten unterbrochen werden. Das ist insbesondere dann sehr sinnvoll, wenn Nebenläufigkeit herrscht bzw. keine Kommunikation zwischen den einzelnen Interaktionspartnern stattfindet und man sicherstellen möchte, dass „c1“ und „b1“ unmittelbar aufeinanderfolgen, ohne dass sie z.B. durch „a1“ oder „a2“ unterbrochen werden.

Wichtig ist aber, dass innerhalb des „critical“-Operators wiederum die Default-Ordnung gilt, d.h. „critical“ ist nicht mit „strict“ gleichzusetzen.

Critical-Operator - Beispiel

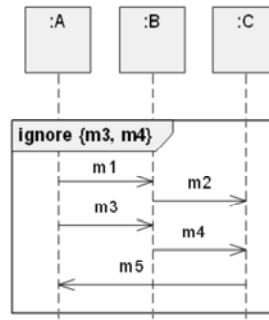


In diesem Beispiel können die zwei Nachrichten, die vom Studenten ausgehen, nicht durch die Nachricht unterbrochen werden, die vom Professor ausgeht.

Filterungen u. Zusicherungen: ignore-Operator

■ Darstellung von irrelevanten Nachrichten

- Modellierung von Nachrichten aus technischen Gründen oder wegen syntaktischer Vollständigkeit
- Nachrichten, die zur Laufzeit auftreten können (z.B. keep-alive Signale)

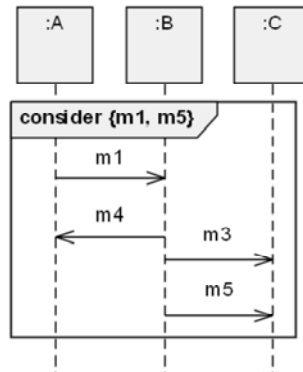


Dann kommen wir auch schon zur letzten Gruppe, den Filterungen und Zusicherungen. Wie vorhin schon angedeutet, geht es hier darum, dass durch die Realisierung des Systems Zusicherungen getroffen werden, die zu erfüllen sind.

Wir können mit dem „ignore“-Operator ausdrücken, dass es Nachrichten gibt, die prinzipiell auftreten können, die aber von keiner weiteren Bedeutung sind. In dem Beispiel definieren wir, dass es die Nachrichten „m3“ und „m4“ gibt, die in unserem System zu dem dargestellten Zeitpunkt auftreten können, sie aber voll und ganz ignoriert werden können. Wichtig ist aber, dass die Nachrichten, die nicht in der Menge enthalten sind, die als Argument von „ignore“ angegeben ist, wichtige Nachrichten sind. D.h. „m1“, „m2“ und „m5“ sind die Nachrichten, die letztendlich für die Realisierung der Funktionalität notwendig sind, sie werden hervorgehoben.

Filterungen u. Zusicherungen: consider-Operator

- Gegenstück von Ignore
- Spezifikation von besonders relevanten Nachrichten
- andere Nachrichten im Operanden werden automatisch als nicht relevant eingestuft

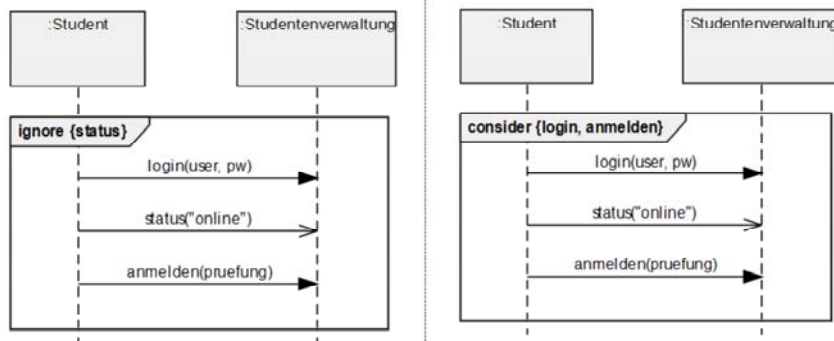


42



Hier haben wir das duale Konstrukt zu „ignore“. Wir können mit „consider“ hervorheben, welche Nachrichten wichtig sind – in diesem Fall „m1“ und „m5“. Und alle Nachrichten, die hier nicht als wichtig gekennzeichnet sind durch die Angabe als Argument von „consider“, sind die unwichtigen Nachrichten. D.h. wir könnten analog das Ganze auch mittels „ignore“ ausdrücken und „m3“ und „m4“ als unwichtige Nachrichten angeben, die ignoriert werden können.

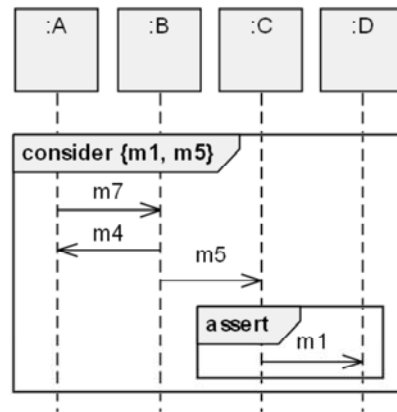
Consider-Operator - Beispiel



Hier haben wir ein Beispiel für consider und ignore. Die zweite Nachricht ist irrelevant, sie trägt nicht dazu bei, dass die eigentliche Funktionalität – das Anmelden – umgesetzt wird.

Filterungen u. Zusicherungen: assert-Operator

- Zugesicherte Interaktionen: Kennzeichnung der Interaktion als verpflichtend – Abweichungen, die im Diagramm nicht berücksichtigt sind, aber in der Realität auftreten, sind nicht zulässig
⇒ Forderung von getreuer Abbildung in der Implementierung

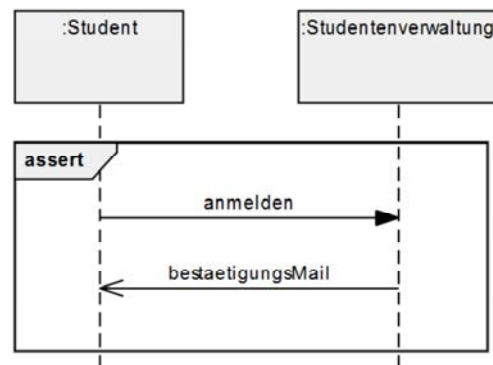


44



Dann haben wir noch den „assert“-Operator. Der „assert“-Operator sagt, dass in der Praxis etwas genauso ausgedrückt werden soll, wie wir es hier im Diagramm dargestellt haben und nicht anders. D.h. hier drücken wir aus, dass keine anderen Nachrichten auftreten dürfen, und dass der Ablauf genauso wie er hier modelliert ist, umgesetzt werden soll. Damit haben wir einfach die Möglichkeit auszudrücken, dass Abweichungen in der Realität nicht zulässig sind. Mit „assert“ wird das wiederum explizit modelliert, wird man aber auch eher selten verwendet.

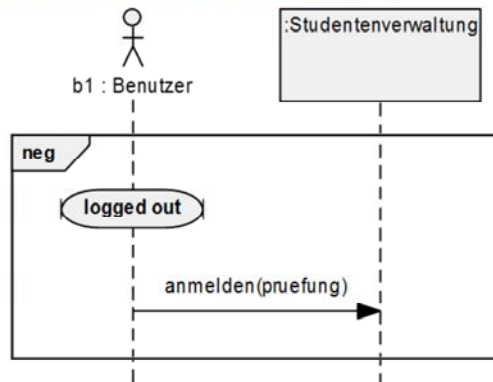
Assert-Operator – Beispiel Prüfungsanmeldung



Hier haben wir noch ein Beispiel für **assert**. Auf die Anmeldung muss immer ein zusätzliches Bestätigungsmail kommen. Andere Abläufe sind nicht zulässig.

Filterungen und Zusicherungen: neg-Operator



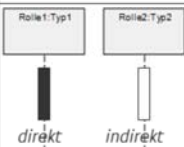
- Ungültige Interaktionen: Es darf nicht sein, dass sich ein Benutzer, der „logged out“ ist, zu einer Prüfung anmeldet



Der letzte Operator ist der neg-Operator. Mit dem neg-Operator können wir einen Sachverhalt ausdrücken, der nicht eintreten darf. In dem Beispiel drücken wir aus, dass es nicht sein darf, dass sich ein Student im Zustand logged out für eine Prüfung anmeldet.



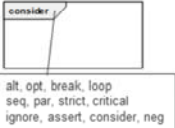
Der neg-Operator ist sparsam einzusetzen. Üblicherweise modellieren wir nicht, was nicht gilt, sondern, was gilt. In manchen, besonders kritischen Fällen ist es aber sinnvoll, verbotene Szenarien explizit zu modellieren und dafür haben wir den neg-Operator.

Basiselemente (1/3)



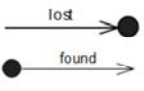
| Name | Syntax | Beschreibung |
|--|---|--|
| Diagramm- rahmen |  | Begrenzung des Diagramms, Angabe von Parametern |
| Lebenslinie |  | Interaktionspartner |
| Ausführungs- spezifikation (direkt/indirekt) |  | Periode, in der ein Interaktionspartner ein Verhalten (direkt/indirekt) ausführt |

Damit komm ich auch schon zum Ende dieser Vorlesungseinheit. In den letzten 3 Folien finden Sie die wichtigsten Sprachkonzepte noch einmal zusammengefasst. Vielen Dank für Ihre Aufmerksamkeit.

Basiselemente (2/3)

| Name | Syntax | Beschreibung |
|-----------------------|---|--|
| aktives Objekt |  | Objekt mit eigenem Kontrollfluss |
| Löschesymbol |  | Zeitpunkt zu dem ein Objekt aus seiner Rolle gelöscht wird |
| Kombiniertes Fragment |  | Steuerung des Kontrollflusses |

Basiselemente (3/3)

| Name | Syntax | Beschreibung |
|--------------------------------------|---|---|
| Synchrone Kommunikation |  | <p>Nachricht</p> <p>Antwort</p> |
| Asynchrone Kommunikation |  | <p>Nachricht</p> |
| Gefundene/ Verlorene Nachricht |  | <p>spezielle Nachrichten von oder an unbekannte Interaktionspartner (z.B. Modellierung von Fehlerfällen in der Kommunikation)</p> |

Zusammenfassung

- Sie haben diese Lektion verstanden, wenn Sie wissen ...
- wofür Interaktionsdiagramme verwendet werden.
- welche Arten von Interaktionsdiagrammen es gibt.
- aus welchen Komponenten ein Sequenzdiagramm besteht.
- was mit einer Lebenslinie gemeint ist.
- wie die Reihenfolge von Ereigniseintritten im Sequenzdiagramm definiert ist.
- was der Unterschied zwischen direkter und indirekter Ausführung, sowie aktiven und passiven Objekten ist.
- welche Operatoren im Sequenzdiagramm zur Verfügung stehen.
- wie Nebenläufigkeit ausgedrückt werden kann.