



Vienna University of Technology

Objektorientierte Modellierung

Aktivitätsdiagramm

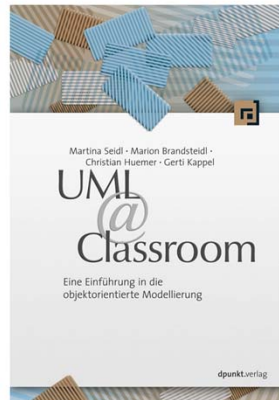


Business Informatics Group
Institute of Software Technology and Interactive Systems
Vienna University of Technology
Favoritenstraße 9-11/188-3, 1040 Vienna, Austria
phone: +43 (1) 58801-18804 (secretary), fax: +43 (1) 58801-18896
office@big.tuwien.ac.at, www.big.tuwien.ac.at

Ich begrüße Sie zur vorletzten Vorlesungseinheit aus Objektorientierter Modellierung mit dem Inhalt Aktivitätsdiagramm.

Literatur

- Die Vorlesung basiert auf folgendem Buch:



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

dpunkt.verlag

Juli 2012

ISBN 3898647765

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

Inhalt

- Einführung
- Aktivitäten
- Aktionen
- Kanten
- Initialknoten, Aktivitätsendknoten, Ablaufendknoten
- Token
- Alternative Abläufe
- Parallele Abläufe
- Objektknoten und Objektfluss
- Signale und Ereignisse
- Ausnahmebehandlung und Unterbrechungsbereich



© BIG / TU Wien



2

Zum Inhalt:

Wir beginnen mit einer **Einführung**.

Dann beschäftigen wir uns mit der Frage: Was sind **Aktivitäten**?

Was sind **Aktionen**? Was ist der Unterschied zwischen Aktivitäten und Aktionen?

Was bedeuten in diesem Diagramm die **Kanten**? Wir kennen Kanten schon aus dem Zustandsdiagramm. Dort haben die Kanten eine ähnliche Bedeutung, nämlich die eines Übergangs, aber trotzdem sind diese Kanten semantisch unterschiedlich.

Es gibt auch **spezielle Knoten**. In diesem Sinne sind das lauter Pseudo-Zustände, d.h. wir haben einen Initialknoten, einen Aktivitätsendknoten und einen Ablaufendknoten.

Dann lernen wir das Konzept der **Token** intensiv kennen.

Dieses Token-Konzept sieht man dann schön bei den **alternativen Abläufen** bzw. vor allem auch bei den **parallelen Abläufen**.

Alles was ich bisher inhaltlich gesagt habe bezieht sich auf die Modellierung eines Kontrollflusses. Man kann aber auch einen **Objektfluss** mittels Aktivitätsdiagrammen modellieren. Beim Objektfluss muss ich dann nicht nur Aktivitätsknoten, sondern auch Objektknoten berücksichtigen.

Ein weiterer wichtiger Punkt sind **Partitionen**. Partitionen sind Unterteilungen, sodass ich Aktivitäten gewissen Classifizieren, seien dies Akteure oder Objekte, zuordnen kann.

Als Spezialformen von Aktionen lernen wir letztendlich noch **Signale und Ereignisse** kennen.

Zum Abschluss widmen wir uns den Themen **Ausnahmebehandlung** und **Unterbrechungsbereich**, um Exceptions zu modellieren.

Einführung

- Fokus des Aktivitätsdiagramms: **prozedurale Verarbeitungsaspekte**
- Spezifikation von **Kontroll-** und/oder **Datenfluss** zwischen Arbeitsschritten (Aktionen) zur Realisierung einer Aktivität
- **Aktivitätsdiagramm in UML2:**
 - ablaforientierte Sprachkonzepte
 - basierend u.a. auf Petri-Netzen und BPEL4WS
- Sprachkonzepte und Notationsvarianten decken ein **breites Anwendungsgebiet** ab
 - Modellierung objektorientierter und nichtobjektorientierter Systeme wird gleichermaßen unterstützt
 - Neben vorgeschlagener grafischer Notation sind auch beliebige andere Notationen (z.B. Pseudocode) erlaubt



© BIG / TU Wien



3

Der Fokus des Aktivitätsdiagramms liegt auf der **prozeduralen Verarbeitung**. D.h. auf der Festlegung: Was passiert zuerst? Was passiert dann als nächstes?

Wir können einen **Kontrollfluss** und einen **Objektfluss** spezifizieren. Auch die Spezifikation von beidem ist möglich.

Ein **Kontrollfluss** legt die Reihenfolge zwischen den Aktionen bzw. Aktivitäten fest. D.h. hier geht es vor allem um die Ausführungsreihenfolge.

Beim **Objektfluss** kann es sein, dass eine Aktion Objekte erzeugt und diese an eine andere Aktion übergibt. Es kann aber so sein, dass ich eine Aktion 1 habe, die erzeugt das Output-Objekt A und gibt das dann an die – von der Ablaufreihenfolge her - dritte Aktion weiter. Die zweite Aktion in unserer Sequenz hat damit gar nichts zu tun. Das ist der Unterschied zwischen einem Kontrollfluss und einem Objektfluss. Der Kontrollfluss legt fest, in welcher Reihenfolge die Aktivitäten abgearbeitet werden, der Objektfluss legt fest, welche Input- und Output-Beziehungen zwischen den Aktivitäten bzw. Aktionen herrschen.

Die Modellierung des Kontrollflusses bzw. des Objektflusses wurde nicht erst durch die objektorientierte Modellierung eingeführt. Hier gibt es schon seit Jahrzehnten Flowchart-Techniken, mit deren Hilfe ich Abläufe beschreiben kann. Bzw. spiegeln Datenflussdiagramme auch die Grundidee des Objektflusses wieder.

In **UML 2** ist das Aktivitätsdiagramm jenes Diagramm das sich am signifikantesten geändert hat. D.h. wenn Sie ein altes Buch lesen basierend auf **UML 1.4** als Grundlage, so ist das das Diagramm wo sie wirklich signifikante Unterschiede merken sollten und daher sollten Sie auch kein altes Buch zum Lernen verwenden. Bei allen anderen Diagrammen, die wir im Wesentlichen bis jetzt durchgemacht haben, gibt es auf dem Detaillierungsgrad, auf dem wir uns befunden haben, kaum einen Unterschied zwischen 1.4 und 2.0 und Folgeversionen. Das Aktivitätsdiagramm stellt hier eine Ausnahme dar. Seien Sie also sicher, dass wenn sie ein Buch lesen, es die Version 2.0 oder höher behandelt. Es wurden nämlich neue Konzepte aufgenommen, vor allem aus den Petri-Netzen, von denen das Token-Konzept kommt. Auch aus der Business Process Execution Language (BPEL) – das ist eine XML-basierte Ablaufbeschreibungssprache – stammen Elemente wie die Signale und Ereignisse, die wir am Schluss dieser Vorlesungseinheit betrachten werden.

Was mache ich mit den Aktivitätsdiagrammen? – Damit kann ich relativ viel modellieren und zwar alles was eben ablaforientierte Aspekte beinhaltet. Ich kann Aktivitätsdiagramme ganz am Beginn des Softwareengineering-Prozesses einsetzen, aber genauso gut ganz am Ende des Softwareengineering-Prozesses.

Ganz am Beginn des Softwareengineering-Prozesses bedeutet, dass ich modelliere, was in einem Anwendungsfall oder Use Case als Standard-Ablauf beschrieben wird, wie ein Akteur mit dem System interagiert. Näheres zu Anwendungsfällen bzw. Use Cases folgt in der übernächsten Vorlesungseinheit. Ich kann mit dem Aktivitätsdiagramm unabhängig davon einen Geschäftsprozess modellieren, als autonome Einheit. Ich kann auch andere prozedurale Abläufe modellieren, die nicht unbedingt ein klassischer Geschäftsprozess sind.

Man kann aber das Aktivitätsdiagramm auch am Ende des Softwareentwicklungsprozess einsetzen, um zu modellieren, was innerhalb einer Operation, d.h. in der Implementierung der Methode, abläuft. Während ich im Sequenzdiagramm modelliert habe wie sich die einzelnen Objekte gegenseitig aufrufen, kann man mit dem Aktivitätsdiagramm spezifizieren was in den Methoden selbst abläuft. Auf dieser Ebene sollte man das Aktivitätsdiagramm nur einsetzen, um die wichtigsten Dinge zu berücksichtigen. Denn kein Mensch wird ein Aktivitätsdiagramm zeichnen für jeden einzelnen Befehlsschritt den Sie in der Programmiersprache haben, weil es einfacher ist Code-Zeilen zu schreiben, als 100.000 Kästchen zu malen. Man sollte es also nicht übertreiben und versuchen alles graphisch zu modellieren.

Ein weiterer Punkt, den ich aber schon angesprochen habe: Aktivitätsdiagramme gibt es schon eine halbe Ewigkeit. Sie sind nicht speziell auf objektorientierte Systeme ausgelegt. Ich kann sie aber auch für die Spezifikation objektorientierter Systeme verwenden, indem ich z.B. angebe, wie ich eine Methode implementiere, d.h. welche Schritte eine Methode ausführt.

Neben den vorgeschlagenen graphischen Notationen sind eigentlich auch andere **Notationsvarianten** wie ein Pseudocode erlaubt. Wir werden das aber nicht nutzen, sondern wir einigen uns auf eine Notationsvariante und zwar auf die im Standard vorgegeben, die auch eben Gegenstand dieser Vorlesung ist.

Aktivität

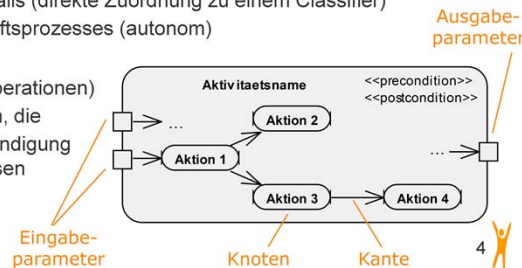
- Eine Aktivität ist ein **gerichteter Graph**
 - Knoten: Aktionen
 - Kanten: Kontroll- und Datenflüsse
- Kontroll- und Datenflüsse legen potentielle »Abläufe« fest
- Spezifikation von **benutzerdefiniertem Verhalten** auf unterschiedlichen Granularitätsebenen

Beispiele:

- Definition einer Operation in Form von einzelnen Anweisungen (indirekte Zuordnung zu einem Classifier)
- Ablauf eines Anwendungsfalls (direkte Zuordnung zu einem Classifier)
- Spezifikation eines Geschäftsprozesses (autonom)

optional:

- **Parameter** (z.B. wie bei Operationen)
- Vor- und Nachbedingungen, die
 - bei Beginn bzw. bei Beendigung der Aktivität gelten müssen



In UML 1.4 gab es noch das Modellierungselement eines Aktivitätsgraphen, d.h. der Graph selbst war ein Modellierungselement. Das gibt es jetzt in UML 2.0 nicht mehr. Hier ist die äußere Hülle eine **Aktivität** und die Aktivität entspricht selbst dem Graphen. Diese Aktivität muss ich verfeinern. Damit ist die Aktivität selbst ein **gerichteter Graph**. Sie beginnt irgendwo wohldefiniert und endet irgendwo wohldefiniert. Es können aber auch mehrere Einstiegs- und mehrere Endpunkte angegeben werden. Wichtig ist, dass der Graph gerichtet ist und Zyklen sind natürlich erlaubt.

Die **Knoten** in diesen Graphen sind Aktionen oder selbst wieder Aktivitäten. Die **Kanten** entsprechen den Kontrollflüssen, sie legen praktisch den potentiellen Ablauf fest. Zusätzlich gibt es noch Kanten, die Objektflüsse kennzeichnen. Diese werden wir später kennenlernen.

Optional habe Aktivitäten **Parameter**. Im dargestellten Beispiel hat die Aktivität zwei Input-Parameter und einen Output-Parameter.

Im Beispiel sehen wir also eine Aktivität als umschließenden Graphen, die durch einzelne Aktionen verfeinert wird. Die Aktionen werden durch Kontrollflusskanten miteinander verbunden und es gibt Parameter. Von den Parametern gehen natürlich keine Kontrollflusskanten weg, sondern Objektflusskanten.

Aktionen

- **Elementare Bausteine** für beliebiges benutzerdefiniertes Verhalten
- **Atomar**, können aber abgebrochen werden
- **Sprachunabhängig**, allerdings Definition in beliebiger Programmiersprache möglich
- Aktionen können Eingabewerte zu Ausgabewerten verarbeiten
- Spezielle **Notation** für bestimmte Aktionsarten
- **Kategorisierung** der 44 in UML vordefinierten Aktionen:
 - Kommunikationsbezogene Aktionen (z.B. Signale und Ereignisse)
 - Objektbezogene Aktionen (z.B. Erzeugen und Löschen von Objekten)
 - Strukturmerkmals- und variablenbezogene Aktionen (z.B. Setzen und Löschen einzelner Werte von Variablen)
 - Linkbezogene Aktionen (z.B. Erzeugen und Löschen von Links zwischen Objekten sowie Navigation)

Aktion A



© BIG / TU Wien



5

Aktionen sind die **elementaren Bausteine** einer Aktivität. Sie beschreiben einen Schritt im Prozess.

Sie sind **atomar**, können aber abgebrochen werden. Was heißt das? – Sie sind schon zeitverbrauchend. Eine Aktivität läuft eine gewisse Zeit, während der sie auch abgebrochen werden kann. Nur eine Aktion selbst wird nicht mehr durch weitere Aktionen verfeinert, denn dann wäre sie ja selbst eine Aktivität.

Aktivität

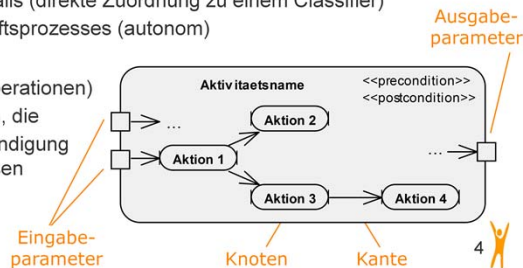
- Eine Aktivität ist ein **gerichteter Graph**
 - Knoten: Aktionen
 - Kanten: Kontroll- und Datenflüsse
- Kontroll- und Datenflüsse legen potentielle »Abläufe« fest
- Spezifikation von **benutzerdefiniertem Verhalten** auf unterschiedlichen Granularitätsebenen

Beispiele:

- Definition einer Operation in Form von einzelnen Anweisungen (indirekte Zuordnung zu einem Classifier)
- Ablauf eines Anwendungsfalls (direkte Zuordnung zu einem Classifier)
- Spezifikation eines Geschäftsprozesses (autonom)

■ optional:

- **Parameter** (z.B. wie bei Operationen)
- Vor- und Nachbedingungen, die
 - bei Beginn bzw. bei Beendigung der Aktivität gelten müssen



Gehen wir noch einmal auf die vorherige Folie zurück. Hier sehen wir, dass die dargestellte Aktivität durch die Aktionen 1, 2, 3 und 4 verfeinert wird. Diese Aktionen sind atomar, weil sie selbst nicht wieder aus Schritten bestehen. Wäre z.B. Aktion 4 eine Aktivität, Aktivität 4, dann würde hinter dieser Aktivität 4 ein weiterer Graph als Verfeinerung liegen.

Aktionen

- **Elementare Bausteine** für beliebiges benutzerdefiniertes Verhalten
- **Atomar**, können aber abgebrochen werden
- **Sprachunabhängig**, allerdings Definition in beliebiger Programmiersprache möglich
- Aktionen können Eingabewerte zu Ausgabewerten verarbeiten
- Spezielle **Notation** für bestimmte Aktionsarten
- **Kategorisierung** der 44 in UML vordefinierten Aktionen:
 - Kommunikationsbezogene Aktionen (z.B. Signale und Ereignisse)
 - Objektbezogene Aktionen (z.B. Erzeugen und Löschen von Objekten)
 - Strukturmerkmals- und variablenbezogene Aktionen (z.B. Setzen und Löschen einzelner Werte von Variablen)
 - Linkbezogene Aktionen (z.B. Erzeugen und Löschen von Links zwischen Objekten sowie Navigation)

Aktion A



© BIG / TU Wien



5

Wir halten fest, eine Aktion ist atomar und besteht damit nicht weiter aus Subgraphen.

Prinzipiell ist die Aktion **sprachunabhängig**, ich kann aber eine Definition in einer bestimmten Programmiersprache angeben.

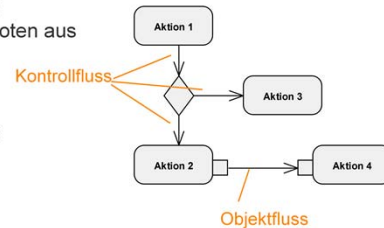
Aktionen sind wie kleine Systeme. Sie wandeln Input in Output um oder **Eingabewerte zu Ausgabewerte**.

UML selbst kennt laut Standard **44 verschiedene Arten von Aktionen**, die sich alle semantisch ein bisschen unterscheiden. Das werden wir hier nicht genauer durchmachen. Wir werden am Schluss nur ganz wenige kennenlernen, die wir von normalen Aktionen unterscheiden. D.h. für diese Beginner-Lehrveranstaltung sind für uns die meisten Aktionen gleich zu behandeln, mit Ausnahme derer, die wir am Schluss der Einheit durchnehmen werden. Für spezielle Aktionsarten gibt es dann spezielle Notationen wie ein "send" und ein "receive", die wir eben später sehen werden.

Es gibt aber auch **Unterkategorisierungen**: D.h. wir haben kommunikationsbezogenen Aktionen, dazu gehören eben Signale und Ereignisse, die besprechen wir heute noch; objektbezogene Aktionen zum Erzeugen und Löschen von Objekten; strukturmerkmals- und variablenbezogene Aktionen z.B. zum Setzen, Verändern und Löschen von Variablen; und linkbezogene Aktionen z.B. zum Erzeugen und Löschen von Links zwischen Objekten, sowie zur Navigation.

Kanten

- Kanten verbinden Knoten und legen **mögliche Abläufe** einer Aktivität fest
 - Kontrollflusskanten
 - Drücken eine **reine Kontrollabhängigkeit** zwischen Vorgänger- und Nachfolgerknoten aus
 - Objektflusskanten
 - Transportieren zusätzlich Daten und drücken dadurch auch eine **Datenabhängigkeit** zwischen Vorgänger- und Nachfolgerknoten aus
- Überwachungsbedingung (guard)
 - Bestimmt, ob Kontroll- und Datenfluss weiterläuft oder nicht



© BIG / TU Wien



6

Wir kommen zu den Kanten. Kanten verbinden Knoten, wie beim Zustandsdiagramm. Aber im Falle des Aktivitätsdiagramms legen sie nun mögliche Abläufe innerhalb einer Aktivität oder in anderen Worten, zwischen Aktionen oder zwischen Aktivitäten fest.

Die wichtigsten Kanten sind die **Kontrollflusskanten**. Sie drücken eine reine Kontrollabhängigkeit aus, d.h. was passiert zuerst, was passiert als nächstes. In unserem Beispiel gibt es die Aktion 1. Wenn die Aktion 1 fertig ist, verlässt man diese über die ausgehende Kontrollflusskante. In unserem Fall kommen wir dann zu einem Split, den lernen wir noch kennen. D.h. ich gehe einen dieser beiden Wege die vom Split hinaus führen. Und wir sagen, wir gehen jetzt zur Aktion 2 aufgrund eines hier nicht angegebenen Entscheidungsmerkmals. Es wird der Kontrollfluss, dass ich zuerst Aktion 1 und dann entweder Aktion 2 oder Aktion 3 durchführe, festgelegt. In einer Instanz entscheide ich mich nach dem Durchführen von Aktion 1, dass ich dann aufgrund eines Merkmals mit Aktion 2 weitermache.

Was ist der wesentlichste Unterschied zwischen den Kanten im Aktivitätsdiagramm, den Kontrollflusskanten, und den Kanten im Zustandsdiagramm, den Zustandsübergängen? Im Falle des Aktivitätsdiagramms ist auf den Kanten nichts angegeben. Eine Angabe, wann ein Übergang ausgelöst wird, ist hier nämlich nicht notwendig, weil der Übergang erfolgt, sobald die vorhergehende Aktion beendet ist. Ich muss nichts angeben, weil wenn die Vorgänger-, die Quellaktion, fertig ist, erfolgt automatisch ein Verfolgen der Kante, ein Kontrollfluss, und ich gehe zur nächsten Aktion über. Im Zustandsdiagramm war auf den Übergängen immer ein Event angegeben. D.h. im Zustandsdiagramm löst ein Event den Übergang zwischen Zuständen aus. Im Aktivitätsdiagramm wird der Übergang ausgelöst durch das Beenden der Vorgänger-Aktion.

Wenn Sie sich ans Zustandsdiagramm erinnern, habe ich damals gesagt, dass es auch eine Sonderform des Zustandsübergangs gibt: Wenn kein Event angegeben ist, dann nimmt man an, dass die Aktivitäten, die innerhalb eines Zustands ausgeführt werden, fertig sind. Darum wurde in UML 1.4 und früheren Versionen das Aktivitätsdiagramm auch als eine Spezialform des Zustandsdiagramms modelliert, eben aufgrund dieser Interpretation. Aber das vergessen wir jetzt wieder. Wichtig für uns ist nur, wann der Übergang ausgelöst wird, eben sobald die Quellaktion fertig ist.

Objektflusskanten transportieren Daten zwischen Aktionen und drücken dadurch eine Datenabhängigkeit zwischen Vorgänger- und Nachfolgerknoten aus. Das ist auch was man aus Datenflussdiagrammen kennt. Hier im Beispiel übergibt die Aktion 2 an die Aktion 4 einen Parameter. Im Fall von Aktion 2 ist das ein Output-Parameter und auf der anderen Seite bei Aktion 4 ein Input-Parameter. Es gibt nachher einen Sonderfall: Wenn Kontrollflusskanten und Objektflusskanten sich decken, kann ich auf die Angabe des Kontrollflusses verzichten. Das sehen wir auch in unserem Beispiel. Von Aktion 1 geht ein Kontrollfluss hinunter. Ich entscheide mich zu Aktion 2 und nicht zu Aktion 3 zu gehen und hier übergebe ich an die Aktion 4 auch einen Parameter, d.h. ich habe hier einen Objektfluss. Ich erspare mir zusätzlich noch den Kontrollfluss zwischen Aktion 2 und Aktion 4 einzuzichnen.

Start und Ende von Aktivitäten und Abläufen

Initialknoten



- Beginn eines Aktivitätsablaufs
- Versorgt alle ausgehenden Kanten mit Kontrolltoken
- Aufbewahrung von Token erlaubt, da Überwachungsbedingungen die Weitergabe blockieren können
- Pro Aktivität keine oder mehrere Initialknoten erlaubt – letzteres ermöglicht Nebenläufigkeit

Aktivitätsendknoten



- Beendet alle Abläufe einer Aktivität sowie den Lebenszyklus eines Objekts
- Der erste Token, der zu einem Endknoten gelangt, beendet die Aktivität (egal, wie viele Kanten in den Knoten führen)
- Keine Ausführung weiterer Aktionen
- Kontrolltoken werden gelöscht, Datentoken an Ausgabepins der Aktivität dagegen nicht
- Pro Aktivität mehrere Aktivitätsendknoten erlaubt

Ablaufendknoten



- Beendet einen Ablauf einer Aktivität



© BIG / TU Wien



Wie startet man und wie beendet man ein Aktivitätsdiagramm? – Die ersten beiden Knoten kennen wir noch vom Zustandsdiagramm.

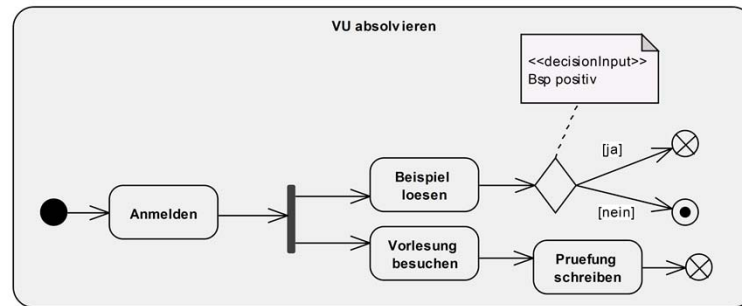
Der **Initialknoten** ist der Beginn eines Aktivitätsablaufes. Er versorgt alle ausgehenden Kanten mit Kontrolltoken. Innerhalb einer Aktivität gibt es einen, oder möglicherweise auch keinen, oder sogar mehrere Initialknoten. Und jeder dieser Initialknoten hat mindestens eine ausgehende Kante, möglicherweise aber auch mehrere ausgehende Kanten. Diese Kanten werden alle mit Token belegt. Genauere Hinweise zum Token-Konzept folgen noch später. Das Aufbewahren von Token im Initialknoten selbst ist erlaubt, falls die ausgehende Kante eine Überwachungsbedingung hat. Die Überwachungsbedingung kennen wir schon aus den Zustandsdiagrammen. Ich habe eine Überwachungsbedingung angegeben und wenn diese nicht erfüllt ist, dann bleibt der entsprechende Token einstweilen einmal im Initialknoten hängen. Pro Aktivität kann es jetzt keinen, einen oder mehrere Initialknoten geben. Das „kein“ sinnvoll ist, sehen wir erst später, wenn wir zu den Ereignissen kommen.

Wir haben den **Aktivitätsendknoten**, der wohl zu unterscheiden ist vom Ablaufendknoten. Er beendet alle Abläufe einer Aktivität, sowie den Lebenszyklus eines Objekts. D.h. sobald irgendein Token – irgendeiner – diesen Aktivitätsendknoten erreicht, ist alles vorbei. Alle anderen Token werden auch von der Aktivität entfernt und es gibt keine weitere Ausführung von Aktionen. Die Kontrolltoken werden gelöscht, die Datentoken an den Ausgabepins der Aktivitäten dagegen nicht. Pro Aktivität kann es auch mehrere Aktivitätsendknoten geben. Egal wo der erste Token einen Aktivitätsendknoten erreicht, ist alles vorbei.

Wir kommen zum **Ablaufendknoten**, der sich wohl vom Aktivitätsendknoten unterscheidet. Ein Ablaufendknoten beendet nämlich nicht den gesamten Ablauf aller Token in einer Aktivität, sondern entfernt nur den einen Token, der jetzt genau am Ablaufendknoten angekommen ist.

Bsp.: Absolvieren einer VU (für Student)

- Die Aktivität "VU absolvieren" besteht aus den Aktionen „anmelden“, „Beispiele lösen“ und „Prüfung schreiben“.



© BIG / TU Wien



8

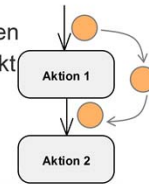
Der Token wird zu Beginn an den Initialknoten gesetzt und dann gleich weitergereicht. An alle ausgehenden Kanten wird jetzt ein Token gelegt. Der Token befindet sich also hier bei der Aktion „Anmelden“. Ich kann immer dann starten, wenn alle eingehenden Kanten belegt sind. Hier haben wir nur eine eingehende Kante. D.h. es wird jetzt diese Aktion „Anmelden“ durchgeführt. Sobald ich mit dem „Anmelden“ fertig bin, wird an alle ausgehenden Kanten ein Token weitergereicht. Wir haben im Beispiel nur eine ausgehende Kante. Der Token wird jetzt weitergereicht zu diesem Symbol. Wenn so ein Symbol vorkommt wird gesplittet, d.h. es wird ein zweiter Token erzeugt. Näheres dazu folgt später. Auf jede der ausgehenden Kanten von diesem Symbol wandert nun ein Token zu den jeweiligen Nachfolgeraktionen. Ich habe im Beispiel die Aktionen „Beispiel lösen“ und „Vorlesung besuchen“. Bin ich mit dem Beispiel lösen fertig, so verlasse ich die Aktion und gehe zu dem Entscheidungsknoten. Waren die Beispiele positiv, d.h. ist die Bedingung [ja] erfüllt, gehe ich den oberen Weg und komme zu dem Ablaufendknoten und der Token wird entfernt. Sobald man mit „Vorlesung besuchen“ fertig ist, wandert der Token zu „Prüfung schreiben“. Wenn man auch „Prüfung schreiben“ beendet, erreicht der Token einen Ablaufendknoten. Der Token wird entfernt. Und wenn kein Token mehr im Aktivitätsdiagramm enthalten ist, bin ich auch mit der Aktivität selbst fertig.

Wir kommen zu einem alternativen Szenario. Wir setzen die Token zurück in den Stand „Beispiel lösen“ und „Vorlesung besuchen“. Sie haben dann die Vorlesung besucht und der Token wandert zu „Prüfung schreiben“. Sie absolvieren die Prüfung, der Token wandert in den Ablaufendknoten und wird weggenommen. Es befindet sich der andere Token jedoch immer noch in „Beispiel lösen“. Nun werden die Beispiele gelöst, der Token wandert in den Entscheidungsknoten und nach dem die Beispiele positiv waren, wandert der Token in den Ablaufendknoten, wird ebenfalls entfernt. Es sind keine Token mehr vorhanden und wir sind am Ende angelangt.

Wir kommen zum dritten alternativen Szenario. Wir setzen die Token wieder in den Stand als sie sich in „Beispiel lösen“ und „Vorlesung besuchen“ befunden haben. Die Aktion des Beispiellösens wird nun beendet, wir wandern mit dem Token wieder in den Entscheidungsknoten. Doch dieses Mal nehmen wir an, dass die Beispiele negativ waren und wir wandern mit unserem Token in den Aktivitätsendknoten. Da es sich um einen Aktivitätsendknoten handelt, wird nicht nur dieser Token entfernt, sondern es werden alle Token der Aktivität entfernt. D.h. es wird auch der Token der sich in „Vorlesung besuchen“ befindet, entfernt. Somit sind wir am Ende der Aktivität angelangt.

Token

- »**Virtueller Koordinationsmechanismus**« zur Beschreibung von Aktivitätsabläufen
- Vorgabe für die Implementierung einer Aktivität
- Token beschreibt möglichen Ablauf einer Aktivität zur Laufzeit
- Token fließen entlang der Kanten von Vorgänger- zu Nachfolgerknoten
 - Werden angeboten und aufbewahrt
 - Lösen Verarbeitung aus
- Unterscheidung in Kontroll- und Datentoken
 - **Kontrolltoken**: "Ausführungserlaubnis" für den Nachfolgeknoten
 - **Datentoken**: Transport von Datenwert oder Referenz auf Objekt
- Überwachungsbedingung kann Weitergabe von Token verhindern (Ansammlung mehrerer Token im Vorgängerknoten)



© BIG / TU Wien



9

Die Kontrolle des Ablaufs innerhalb einer Aktivität wird durch das aus den Petri-Netzen stammende Token-Konzept gesteuert.

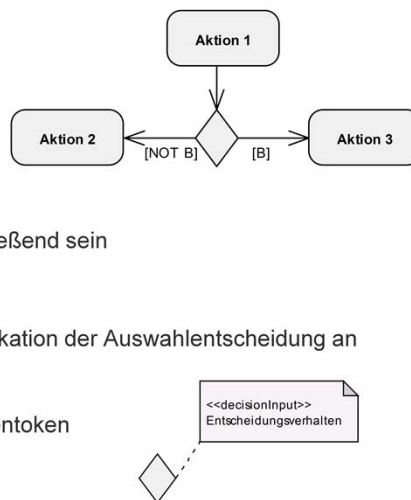
Das Token-Konzept ist ein **virtueller Koordinationsmechanismus** zur Beschreibung von Aktivitätsabläufen. Dabei stellt der Token eine Marke dar, welche durch die Aktivität fließt. In einer Aktivität befindet sich nicht immer nur ein Token, sondern es können auch mehrere Token durch die Aktivität fließen. Die Token fließen immer entlang der Kanten vom Vorgänger- zum Nachfolgeknoten. Die Token lösen die Verarbeitung aus. Eine Aktion beginnt, sobald an allen eingehenden Kanten ein Token vorhanden ist. Sobald eine Aktion abgearbeitet ist, werden an alle ausgehenden Kanten dieser Aktion die Token angeboten und sie fließen weiter. Sollte jedoch eine Überwachungsbedingung gegeben sein, die nicht erfüllt ist, so können die Token auch in der Aktion aufbewahrt werden.

Wir unterscheiden **Kontrolltoken** und **Datentoken**. Das Konzept der unterschiedlichen Token stammt ebenfalls aus den Petri-Netzen. Dort gibt es Colored Petri-Netze, wo unterschiedlich Farben für unterschiedliche Semantiken verwendet werden. Kontroll-Token stehen sozusagen für die Ausführungserlaubnis für den Nachfolgeknoten, während die Datentoken den Transport von Datenwerten oder eine Referenz auf ein Objekt ermöglichen.

Wie bereits erwähnt sind auch **Überwachungsbedingungen** im Konzept erlaubt. Überwachungsbedingungen verhindern die Weitergabe von Token und führen damit zu einer Ansammlung mehrerer Token im Vorgängerknoten. Und wenn ich einen Token nicht weitergeben kann, weil die Überwachungsbedingung nicht erfüllt ist, dann bleibt der Token im Vorgängerknoten hängen.

Alternative Abläufe - Entscheidungsknoten

- Definiert alternative Zweige und repräsentiert eine »Weiche« für den Tokenfluss
 - Verwendung auch zur Modellierung von Schleifen
- Überwachungsbedingungen
 - Wählen den Zweig aus
 - Müssen wechselseitig ausschließend sein
 - [else] ist vordefiniert
- Entscheidungsverhalten
 - Ermöglicht detailliertere Spezifikation der Auswahlentscheidung an zentraler Stelle
 - Ankunft von Token startet das Entscheidungsverhalten – Datentoken fungieren als Parameter



10 

Wir kommen zu den alternativen Abläufen. Für diese verwenden wir den **Entscheidungsknoten**. Der Entscheidungsknoten definiert alternative Zweige und repräsentiert eine Weiche für den Token-Fluss.

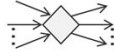
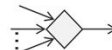
Im dargestellten Beispiel ist es folgendermaßen: Sobald ich mit Aktion 1 fertig bin, verlasse ich die Aktion 1 über die Kontrollflusskante und komme hier mit dem Token zu einem Entscheidungsknoten. Bei einem Entscheidungsknoten kann ich nur in die eine oder in die andere Richtung weitergehen. Also entweder tritt [B] ein, oder es gilt [NOT B]. Im ersten Fall gehe ich nach rechts zur Aktion 3. Im zweiten Fall würde der Token nach links zu Aktion 2 wandern. Man kann auf einer der ausgehenden Kanten vom Entscheidungsknoten auch **[else]** verwenden, denn [else] ist vordefiniert, d.h. ich hätte auch rechts [B] als Bedingung schreiben können und links dann [else] angegeben.

Wichtig ist, wie im Zustandsdiagramm auch, dass sich die Bedingungen **wechselseitig ausschließen** müssen. Beispielsweise könnte man auf einer Kante angeben [x<2] und auf der anderen [x>5]. Dies wäre gültig, denn die Bedingungen schließen sich wechselseitig aus. Es darf also nicht sein, dass die Bedingungen einen überlappenden Bereich haben. Doch bei den Bedingungen [x<2] und [x>5] gibt es auch einen Bereich dazwischen. Was ist mit diesem Bereich? – Wenn ein Wert zwischen 2 und 5 eintreten kann, dann bleibt der Token am Entscheidungsknoten ewig hängen und wird nie wieder wegkommen, es sei denn, ich habe einen Aktivitätsendknoten erreicht bei dem alle Token entfernt werden. D.h. eine Bedingung, wo Sie keine vollständige Aufteilung haben, ist zwar nicht notwendigerweise falsch, Sie sollten es aber hinterfragen. Wichtig ist, dass die Überwachungsbedingungen wechselseitig ausschließend sind.

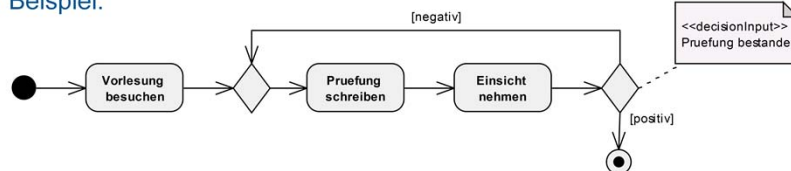
Sie können zusätzlich bei dem Entscheidungsknoten das **Entscheidungsverhalten** mit <<decisionInput>> in einer beliebigen Sprache angeben. Damit wird eine detaillierte Spezifikation der Auswahlentscheidung am Knoten ermöglicht.

Alternative Abläufe - Vereinigungsknoten

- Ein Vereinigungsknoten führt alternative (keine nebenläufigen!) Abläufe wieder zusammen
- Token werden, sobald möglich, an den Nachfolgerknoten weitergereicht
- Kombiniertes Entscheidungs- und Vereinigungsknoten



- Beispiel:



Das Gegenstück zum Entscheidungsknoten ist der **Vereinigungsknoten**. Ein Vereinigungsknoten dient dazu, mehrere alternative Abläufe wieder zusammenzuführen. Hier ist die Semantik, dass nur eine der eingehenden Kanten mit einem Token belegt sein muss, damit der Token automatisch weitergereicht wird. D.h. vorher habe ich immer gesagt, es müssen alle Kanten belegt sein. Beim Vereinigungsknoten muss nur eine Kante belegt sein. D.h. hier im Beispiel ist z.B. die obere Kante mit einem Token belegt und dieser wird beim Vereinigungsknoten automatisch an die ausgehende Kante weitergegeben.

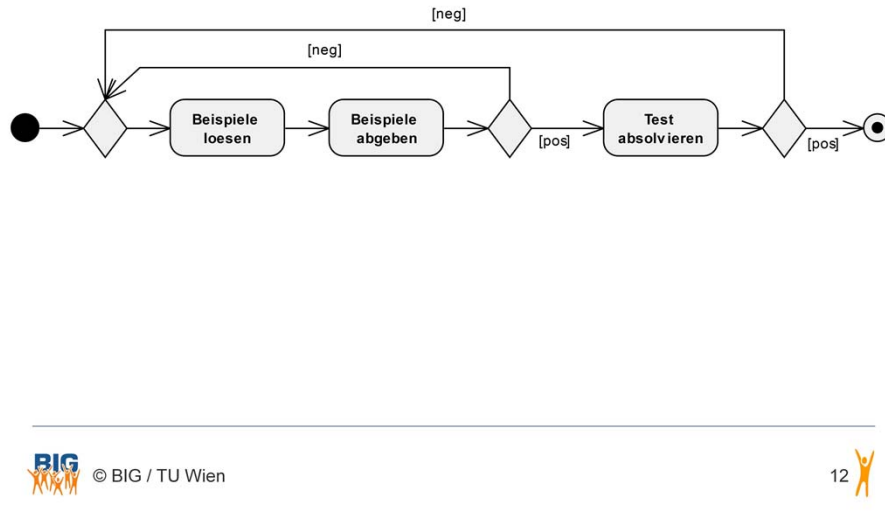
Sie können den Entscheidungsknoten und den Vereinigungsknoten auch in einen Knoten verschmelzen. Dementsprechend haben Sie dann einen **kombinierten Entscheidungs- und Vereinigungsknoten**. Der besitzt mehrere eingehende Kanten und mehrere ausgehende Kanten. Sobald eine der eingehenden Kanten dann belegt ist, kommt der Token in den kombinierten Knoten, dann muss natürlich eine Entscheidung mit Überwachungsbedingungen getroffen werden und der Token wird entsprechend an eine der drei ausgehenden Kanten weitergegeben.

Der Entscheidungsknoten hat im Normalfall eine eingehende und mehrere ausgehende Kanten. Ein Vereinigungsknoten hat mehrere eingehende Kanten, aber nur eine ausgehende Kante. Und wenn Sie beide kombinieren, dann haben Sie sowohl mehrere eingehende, als auch mehrere ausgehende Kanten.

Hier haben wir auch noch Beispiel. Der Token startet am Initialknoten. Danach wird „Vorlesung besuchen“ ausgeführt. Dann haben wir hier einen Vereinigungsknoten. Wir sehen, dass es vor einem Vereinigungsknoten nicht immer einen Entscheidungsknoten geben muss. Eine der beiden eingehenden Kanten des Vereinigungsknotens muss belegt sein, um die Ausführung fortsetzen zu können. Und das ist ja nun der Fall. Wir gehen also hier zu „Prüfung schreiben“. Dann wird „Einsicht nehmen“ ausgeführt. Wir kommen nun zu einem Entscheidungsknoten. Waren Sie negativ, dann gehen Sie hier zurück zum Vereinigungsknoten, damit ist wieder eine der beiden eingehenden Kanten belegt und es beginnt wieder mit „Prüfung schreiben“, dann „Einsicht nehmen“. Wir kommen wieder zum Entscheidungsknoten. Waren Sie positiv, kommen Sie zum Aktivitätsendknoten und die Aktivität ist damit beendet.

An diesem Beispiel sieht man auch einen der größten Fehler der häufig passiert. Sie zeichnen so ein Diagramm und beginnen mit „Vorlesung besuchen“, „Prüfung schreiben“, „Einsicht nehmen“ und kommen jetzt drauf, dass Sie eine Schleife brauchen. Wenn man negativ war, dann muss die Prüfung noch einmal geschrieben werden. Der typischste Fehler der hier passiert ist, dass Sie die Kante direkt zu der Aktion „Prüfung schreiben“ einzeichnen. Was haben wir jetzt falsch gemacht? – Es fehlt ein Token um die Aktion „Prüfung schreiben“ starten zu können. Denn alle eingehenden Kanten müssen mit Token belegt sein, um eine Aktion auszuführen und jetzt ergibt sich hier eine Deadlock-Situation, weil die obere Kante mit keinem Token belegt ist und auch nie belegt werden kann. Dazu müssten „Prüfung schreiben“ und „Einsicht nehmen“ schon einmal durchlaufen werden und man dann negativ sein, damit die obere Kante überhaupt belegt werden kann. Das kann aber beim ersten Durchlauf niemals der Fall sein. Dies ist ein ganz typischer Fehler, der immer wieder bei der Modellierung von Aktivitätsdiagrammen passiert.

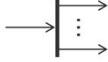
Alternative Abläufe – Bsp.: Absolvieren einer LU



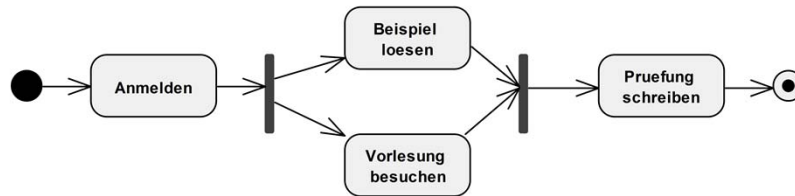
Hier haben wir noch einmal ein ähnliches Beispiel.

Wir starten am Initialknoten, kommen als nächstes zum Vereinigungsknoten. Damit ist eine der drei eingehenden Kanten des Vereinigungsknotens belegt. Dementsprechend starten wir mit „Beispiele lösen“. Als nächstes kommt „Beispiele abgeben“. Wir kommen zum Entscheidungsknoten. Waren Sie negativ, müssen wir wieder zurück zum Vereinigungsknoten. Es ist somit eine eingehende Kante des Vereinigungsknotens belegt. Als nächstes gehen wir zu „Beispiele lösen“ und dann zu „Beispiele abgeben“. Waren Sie nun positiv, gehen wir als nächstes zu „Test absolvieren“. Waren Sie negativ auf den Test, so müssen wir wieder zurückgehen zum Vereinigungsknoten. Es ist wieder eine der eingehenden Kanten belegt. Wir gehen im nächsten Schritt wieder zu „Beispiele lösen“, dann zu „Beispiele abgeben“, kommen zum Entscheidungsknoten. Waren Sie positiv, kommen wir zu „Test absolvieren“. Sind Sie dieses Mal auch positiv, so erreichen wir den Aktivitätsendknoten und damit sind wir am Ende angelangt.

Nebenläufige Abläufe - Parallelisierungsknoten

- Zur Modellierung der Aufspaltung von Abläufen 
- Eingehende Token werden für alle ausgehenden Kanten dupliziert, sobald zumindest eine Überwachungsbedingung diese akzeptiert
- Nichtakzeptierte Token werden aufbewahrt

- Beispiel:

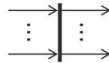
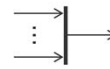


Wir kommen als nächstes zum Konzept der parallelen Abläufe. Diese werden mit dem **Parallelisierungsknoten** modelliert. Ich nenne ihn auch oft Synchronisationsbalken, doch das ist eine alte Bezeichnung. Das wesentliche ist: Eingehende Token werden für alle ausgehenden Kanten dupliziert. Man kann auf einer ausgehenden Kante auch eine Überwachungsbedingung angeben. Sollte die Überwachungsbedingung nicht erfüllt sein, dann bleibt der Token im Parallelisierungsknoten.

Sehen wir uns das hier rechts oben auf der Folie an: Wenn ein Token zum Parallelisierungsknoten kommt, bedeutet das, dass für alle ausgehenden Kanten der Token vervielfacht wird und diese Token an die dahinter liegenden Aktionen weitergegeben werden.

Nebenläufige Abläufe – Synchronisierungsknoten (1/2)

- Führt nebenläufige Abläufe zusammen
- Tokenverarbeitung
 - Vereinigung der Token, sobald an allen Kanten vorhanden
 - An einer Kante anliegende Kontrolltoken werden vereinigt
 - Kontrolltoken verschiedener Kanten werden vereinigt und nur ein einzelnes Token weitergereicht
 - Datentoken werden alle weitergereicht
 - Bei Kontroll- und Datentoken - nur Datentoken werden weitergereicht
 - Nichtakzeptierte Token werden nicht wieder angeboten
- Kombiniertes Parallelisierungs- und Synchronisierungsknoten:



© BIG / TU Wien



15

Das Gegenstück zum Parallelisierungsknoten ist der **Synchronisierungsknoten**. Hier werden nebenläufige Abläufe wieder zusammengeführt. Wenn an allen eingehenden Kanten ein Token liegt, werden diese zu einem Token vereinigt, der anschließend weitergegeben wird. D.h. es gibt mehrere eingehende Kanten, wenn diese alle belegt sind, wird ein Token an die ausgehende Kante weitergegeben.

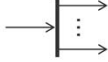
Das Ganze gilt für die Kontrollflusstoken. Bei den **Datentoken** werden alle weitergereicht und nicht verschmolzen, was logisch ist, weil wenn zwei verschiedene Objekte wie eine Adresse und ein Personen-Objekt eingegeben werden, kann ich diese nicht verschmelzen. D.h. das Verschmelzen bezieht sich auf den Kontrollfluss, nicht auf den Objektfluss.

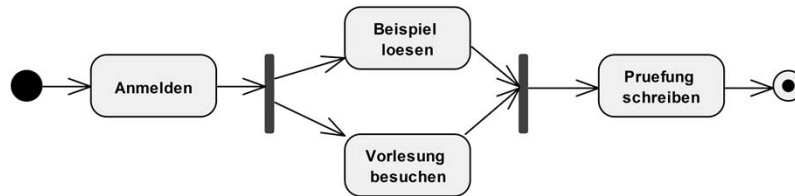
Sollten **Kontroll- und Datentoken** eingehen, so werden nur die Datentoken weitergegeben.

Nichtakzeptierte Token werden nicht wieder angeboten.

Eine wesentliche Eigenschaft, die wir vorher bei den Entscheidungs- und Vereinigungsknoten gelernt haben, gilt auch für die Parallelisierungs- und Synchronisierungsknoten. Nämlich man kann sie auch **kombinieren**. Damit hat man mehrere eingehende und mehrere ausgehende Kanten. Man muss am kombinierten Knoten also zuerst warten, bis alle eingehenden Kanten mit Token belegt sind, diese werden dann zu einem Token vereint und dieser eine Token wird vervielfacht für alle ausgehenden Kanten.

Nebenläufige Abläufe - Parallelisierungsknoten

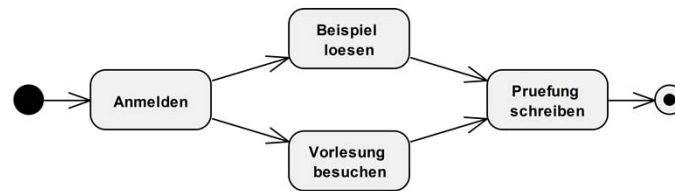
- Zur Modellierung der Aufspaltung von Abläufen 
- Eingehende Token werden für alle ausgehenden Kanten dupliziert, sobald zumindest eine Überwachungsbedingung diese akzeptiert
- Nichtakzeptierte Token werden aufbewahrt
- Beispiel:



Gehen wir noch einmal zur vorigen Folie zurück und sehen uns das Beispiel an.

Wir starten mit einem Token am Initialknoten. Danach kommen wir zur Aktion „Anmelden“. Sobald wir angemeldet sind, kommen wir zum Parallelisierungsknoten. Der anliegende Token wird aufgesplittet in zwei Token, diese werden weitergereicht an „Beispiel lösen“ bzw. an „Vorlesung besuchen“. Wenn „Beispiel lösen“ fertig ausgeführt wurde, wird der Token an den Synchronisierungsknoten weitergegeben. Aber erst wenn alle eingehenden Kanten des Synchronisierungsknotens belegt sind, geht es weiter. Nachdem auch „Vorlesung besuchen“ ausgeführt wurde, sind nun beide Kanten belegt. Wir verschmelzen diese beiden Token zu einem Token. Es geht dann weiter mit „Prüfung schreiben“. Nachdem „Prüfung schreiben“ ausgeführt wurde, erreichen wir den Aktivitätsendknoten und damit ist die Aktivität zu Ende.

Alternative Darstellung



© BIG / TU Wien



14

Dieses Beispiel hätte man alternativ auch folgendermaßen darstellen können.

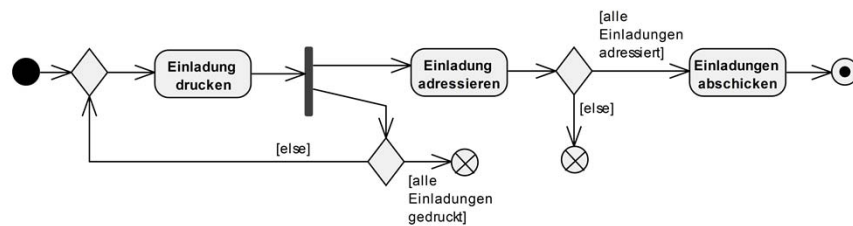
Die Aktivität beginnt nach dem Initialknoten wieder mit „Anmelden“. Sobald ich mit „Anmelden“ fertig bin, lautet die Regel: Alle ausgehenden Kanten werden mit einem Token belegt. Damit werden zwei Token weitergereicht und es werden „Beispiel lösen“ und „Vorlesung besuchen“ ausgeführt. Wurden die Beispiele gelöst, wird der Token an die ausgehende Kante weitergegeben. Die Aktion „Prüfung schreiben“ kann aber noch nicht ausgeführt werden, weil dafür alle eingehenden Kanten belegt sein müssen. D.h. erst wenn „Vorlesung besuchen“ durchgeführt ist, sind beide Kanten belegt.

D.h. die beiden Darstellungen auf der vorigen Folie und auf dieser Folie sind äquivalent. Die Angabe von Parallelisierungsknoten und Synchronisierungsknoten ist daher in UML 2 letztendlich reiner syntactic sugar. Es macht das Diagramm jedoch besser lesbar wenn ich Parallelisierungs- und Synchronisierungsknoten verwende, weil ich explizit sehe, dass es sich um parallele Abläufe handelt, aber diese erreiche ich auch ohne die Angabe der jeweiligen Knoten. Dementsprechend kann ich auf Parallelisierungs- und Synchronisierungsknoten verzichten, indem ich einfach mit mehreren aus einer Aktion hinaus bzw. mit mehreren Kanten in eine Aktion hinein gehe.

Entscheidungsknoten und der Vereinigungsknoten haben natürlich eine andere Semantik weil sie eben dazu führt, dass immer nur ein Token weitergegeben wird bzw. nur ein Token anliegen muss, damit die Ausführung weitergeht. Dementsprechend kann ich auf Entscheidungsknoten und Vereinigungsknoten zur Modellierung alternativer Abläufe nicht verzichten.

Bsp.: Erstellen und Versenden von Einladungen zu einem Termin im CALENDARIUM

- Einladungen werden einzeln gedruckt und parallel adressiert



© BIG / TU Wien



16

Wir sehen uns jetzt noch ein weiteres Beispiel an.

Wir starten wieder am Initialknoten. Für den Vereinigungsknoten reicht es, wenn eine Kante anliegt. Der Token wird also weitergereicht zu „Einladung drucken“. Sie drucken eine Einladung und danach wird der Token zum Parallelisierungsknoten weitergereicht. Hier wird er für alle ausgehenden Kanten dupliziert. Der eine Token geht weiter zu „Einladung adressieren“. Der zweite geht hinunter zu einem Entscheidungsknoten und hier stellt man sich die Frage: Sind alle Einladungen gedruckt? Die Antwort lautet: Nein, es sind noch nicht alle Einladungen gedruckt. Damit wird der Token zum Vereinigungsknoten am Beginn des Diagramms weitergereicht.

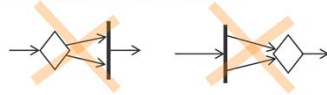
An dieser Bedingung sieht man schon, dass es sich um die Modellierung eines Geschäftsprozesses handelt, wie ihn ein Betriebswirt beschreiben würde, denn der Entscheidungsknoten weiß natürlich nicht, wann alle Einladungen gedruckt sind. Für ein Workflow-System ist so ein Modell unbrauchbar.

Der Token wird also weitergegeben an „Einladung drucken“ und danach vervielfältigt. Wiederum wird einer der Token an „Einladung adressieren“ weitergereicht und der andere gelangt zum Entscheidungsknoten. Es sind noch nicht alle Einladungen gedruckt, daher wird der Token also wieder weitergegeben zu „Einladung drucken“ und dann vervielfacht. Jetzt gibt es keine weiteren Einladungen mehr, somit kommt der untere Token zum Ablaufendknoten und wird gelöscht. Wir machen weiter bei „Einladung adressieren“. Haben wir alle Einladungen adressiert? – Nein, es kommen noch welche, daher verfolgt der Token den [else] Zweig und wird am Ablaufendknoten gelöscht. Dann geht der nächste Token zum Entscheidungsknoten, wieder wird der [else] Zweig gewählt, da noch Einladungen zu adressieren sind und der Token wird am Ablaufendknoten gelöscht. Beim letzten Token sind nun alle Einladungen adressiert. Damit wird der Token weitergereicht zu „Einladungen abschicken“. Nach der Ausführung dieser Aktion erreichen wir den Aktivitätsendknoten und damit ist die Aktivität beendet.

Als Geschäftsprozess ist dieses Modell richtig, als Workflow ist es extrem schlecht, da eine Maschine die angegebenen Überwachungsbedingungen nicht interpretieren kann.

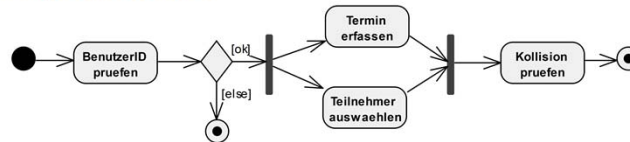
Nebenläufige Abläufe – Synchronisierungsknoten (2/2)

- Falsche Modellierungen



- Beispiel: Einladung im CALENDARIUM

- Wurde die BenutzerID erfolgreich geprüft, ist parallel die Terminerfassung und die Teilnehmerauswahl möglich. Erst wenn diese beiden Aktionen abgeschlossen sind, kann die Kollisionsprüfung durchgeführt werden.



Hier sehen wir Beispiele für eine schlechte Modellierung.

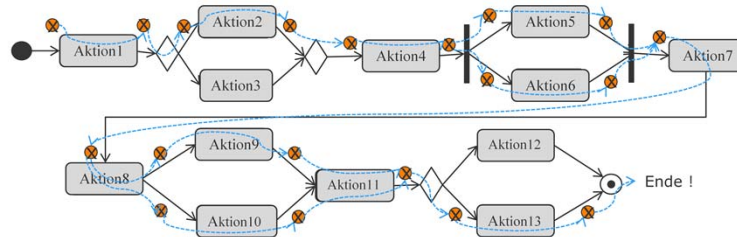
Was ist beim ersten Modell falsch? – Der Token, der am Entscheidungsknoten anliegt wird nur an eine der beiden ausgehenden Kanten weitergegeben. Der Synchronisierungsknoten benötigt aber zwei eingehende Token um diese zu vereinigen und weiterzugeben. In seltenen Fällen kommt unter Umständen noch von irgendwo ein zweiter Token. Im Allgemeinen sollten Sie aber so ein Konstrukt in Ihrem Modell noch einmal überdenken.

Im zweiten Fall kommt ein Token zum Parallelisierungsknoten, er wird aufgeteilt und damit werden zwei Token an den Vereinigungsknoten weitergegeben und jeder der beiden wird von diesem weitergegeben. Ist das falsch? – Natürlich kann das durchaus gewollt sein, jedoch kommt das nur sehr selten vor.

In den meisten Fällen gilt: Wenn ein Pfad mit einem Entscheidungsknoten aufgespalten wird, so wird er mit einem Vereinigungsknoten wieder zusammengeführt. Parallele Abläufe, die durch einen Parallelisierungsknoten starten, werden später durch einen Synchronisierungsknoten wieder zusammengeführt.

Wir beginnen im dargestellten Beispiel wieder mit dem Token am Initialknoten. Danach gelangen wir zu „BenutzerID prüfen“. Wir kommen zu einem Entscheidungsknoten. Wenn die BenutzerID nicht ok ist, wird die Aktivität beenden und wir sind bereits fertig. Ist sie jedoch ok, wird er Token am Parallelisierungsknoten vervielfacht und die beiden Aktionen „Termin erfassen“ und „Teilnehmer auswählen“ werden mit Token belegt. Danach werden die beiden Token am Synchronisierungsknoten vereinigt und ein Token wird an „Kollision prüfen“ weitergegeben. Danach ist die Aktivität zu Ende.

Token – Beispiel (Kontrollfluss)



- ... zu Beginn werden alle vom Initialknoten ausgehenden Kanten mit einem Token belegt....
- ... eine Aktion, bei der alle eingehenden Kanten mit einem Token belegt sind, ist aktiviert und kann durchgeführt werden
- ... vor der Durchführung „nimmt“ sich die Aktion von jeder eingehenden Kante einen Token; nach der Durchführung belegt die Aktion jede ausgehende Kante mit einem Token
- ... ein Entscheidungsknoten gibt den Token an **eine** ausgehende Kante weiter
- ... ein Vereinigungsknoten reicht jeden Token, den er bekommt, einzeln weiter
- ... ein Parallelisierungsknoten dupliziert den Token für **jede** ausgehende Kante
- ... ein Synchronisierungsknoten wartet, bis an allen eingehenden Kanten Token anliegen und gibt dann einen Token weiter
- ... der Endknoten ist eine Ausnahme vom Tokenkonzept. Er beendet mit dem ersten Token, den er (egal über welche Kante) bekommt, den gesamten Ablauf

18

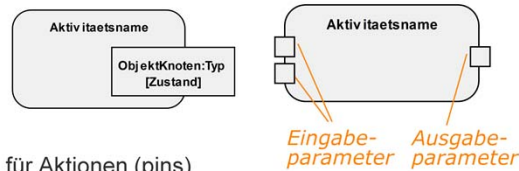
Auf dieser Folie wird das Token-Konzept noch einmal dargestellt.

Die Aktivität beginnt mit der Aktion 1. Sie wird durchgeführt. Danach wird der Token am Entscheidungsknoten an eine der beiden Kanten weitergegeben. Im Beispiel geben wir ihn weiter an Aktion 2. Wenn Aktion 2 ausgeführt wurde wird der Token weitergegeben. Damit ist eine der beiden eingehenden Kanten des Vereinigungsknotens belegt und der Token wird an Aktion 4 weitergegeben. Aktion 4 wird durchgeführt. Der Token wird am Parallelisierungsknoten vervielfacht. Aktion 5 und Aktion 6 werden durchgeführt. Beide Token kommen zum Synchronisierungsknoten. Sie werden verschmolzen zu einem Token und Aktion 7 wird ausgeführt. Der Token wird weitergegeben an die Aktion 8. Hier sehen wir die alternative Darstellung zum Parallelisierungsknoten. Aktion 8 entspricht also Aktion 4. Aufgrund der beiden ausgehenden Kanten werden auch zwei Token weitergegeben an Aktion 9 und an Aktion 10. Wenn beide durchgeführt sind, kann Aktion 11 starten, da dann an allen eingehenden Kanten von Aktion 11 ein Token anliegt. Beim Entscheidungsknoten wird der Token in unserem Beispiel an Aktion 13 weitergegeben und hier haben wir einen Spezialfall. Beim Aktivitätsendknoten müssen nicht so wie bei Aktion 11 beide Kanten belegt sein, sondern es genügt, dass eine der beiden Kanten am Aktivitätsendknoten belegt ist. Weil der Token von Aktion 13 kommt, wird die Aktivität beendet. D.h. beim Aktivitätsendknoten müssen nicht alle eingehenden Kanten belegt sein, schon ein Token an einer Kante führt zum Beenden der Aktivität.

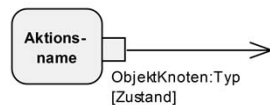
Damit haben wir bereits das wichtigste zum Kontrollfluss kennengelernt.

Objektknoten (1/7)

- Inhalt: **Datentoken**
- Objektknoten stehen durch Objektflüsse miteinander in Beziehung
- Inhalt ist **Ergebnis einer Aktion** und Eingabe für eine weitere Aktion
- Typangabe und Zustandseinschränkung sind optional
- Objektknoten als **Ein-/Ausgabeparameter**
 - für Aktivitäten (activity parameter node)



- für Aktionen (pins)



19

Wir kommen als nächstes zum **Objektfluss** in dem Datentoken fließen. Diesen modellieren wir mit Objektknoten. Dafür gibt es verschiedene Möglichkeiten der Notation.

Objektknoten stehen durch Objektflüsse miteinander in Beziehung. Der Inhalt eines **Datentokens** ist das Ergebnis einer Aktion und Eingabe für eine weitere Aktion bzw. wird ein Output-Parameter als Input-Parameter an die nächste Aktion weitergegeben.

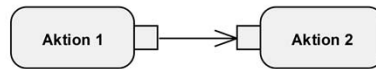
Im Beispiel oben gibt es die Aktion A. Sie erzeugt ein Objekt als Output-Parameter, das Input für die Aktion B ist. Ich kann im Objektknoten ein Objekt angeben. Wie man ein Objekt angibt, kennen Sie schon vom Klassendiagramm, nämlich geben wir zuerst den Namen des Objekts an, gefolgt von einem Doppelpunkt (:), dann kommt die Klasse bzw. der Typ des Objekts und darunter kann man noch den Zustand angeben in dem sich das Objekt befindet. Dieser Zustand ist eine Referenz zum Zustandsdiagramm der angegebenen Klasse.

Ich kann Objektknoten auch als **Ein- oder Ausgabeparameter** notieren, indem ich die Pin-Notation verwendet. Pins werden direkt an die Kanten der Aktivität bzw. Aktion gesetzt.

Objektknoten (2/7)

- Kennzeichnung als Ein- und Ausgabepin

- Notationskonvention: Eingabepins links bzw. oberhalb einer Aktion, Ausgabepins rechts bzw. unterhalb
- Richtung der Objektflusskante



- Weitere Notationsvariante z.B.:



Hier sehen wir oben ein Beispiel für **Eingabe- und Ausgabepins**. Auf der Seite von Aktion 1 haben wir einen Ausgabepin, auf der Seite von Aktion 2 einen Eingabepin. Der Objektfluss wird mittels eines Pfeils zwischen den Pins notiert. Der Objektfluss fließt entsprechend der Richtung der Objektflusskante.

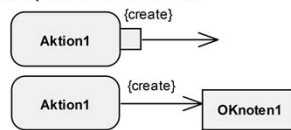
Dasselbe könnte man auch mit **Objektknoten** einzeichnen. Die beiden Notationsvarianten sind äquivalent.

D.h. die unten angeführte Darstellung ist äquivalent zur oben angeführten Darstellung.

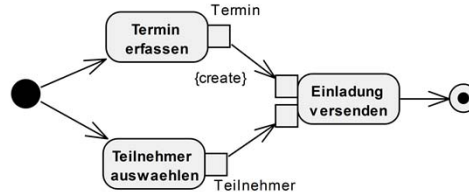
Objektknoten (3/7)

- Effekte einer Aktion auf Daten und Objekte

- create, read, update und delete



- Beispiel für Ein- und Ausgabepins und Effekte



Sie können bei den Pins oder auch bei den Objektknoten angeben, welchen **Effekt** die Aktionen auf die Daten bzw. Objekte haben, wie beispielsweise das Erzeugen, Lesen, Updaten oder Löschen eines Objekts. Den Effekt können Sie in geschweiften Klammern angeben, das wird aber in der Praxis kaum verwendet.

Hier haben wir ein Beispiel für Ein- und Ausgabepins und deren Effekte. Wir starten am Initialknoten. Alle ausgehenden Kanten werden mit einem Token versorgt. Beide Aktionen „Termin erfassen“ und „Teilnehmer auswählen“ werden ausgeführt. Anstatt Kontrollflusskanten haben wir jetzt nur mehr Objektflusskanten, da Kontroll- und Objektfluss zusammenfallen. Die Kontrollfluss-Token können also entfallen. Den Objektfluss stelle ich rot dar. „Termin“ ist also ein Output-Parameter von „Termin erfassen“ und „Teilnehmer“ ist ein Output-Parameter von „Teilnehmer auswählen“. Diese Output-Parameter werden nun an „Einladung versenden“ als Input-Parameter weitergegeben. Und weil nun beide Kanten belegt sind, kann „Einladung versenden“ ausgeführt werden und wenn wir damit fertig sind, wird ein Kontrolltoken an den Aktivitätsendknoten weitergegeben. Damit ist die Aktivität beendet.

Objektknoten (4/7)

- **Konstanter Eingabewert – Wertepin**
 - Zur Übergabe konstanter Werte
 - **Startet nicht die Verarbeitung eines Knotens**
- **Pufferknoten**
 - Zentrale Pufferung von Datentoken
 - Transienter Pufferknoten (central buffer node)
 - Löscht Datentoken, sobald er sie weitergegeben hat
 - Persistenter Pufferknoten (data store node)
 - Bewahrt Datentoken auf und gibt Duplikate weiter
 - Keine Mehrfachspeicherung identer Objekte
 - Explizites »Abholen« der Datentoken möglich

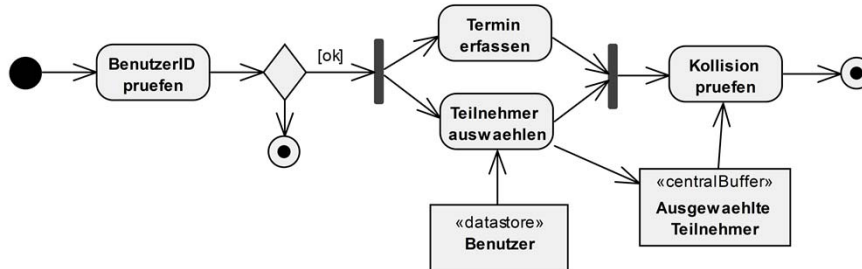


Es können auch **Konstanten** als Eingabeparameter angegeben werden. In unserem Beispiel wird er Wert 5 als Konstante an die Aktion 1 als Eingabeparameter übergeben. Solch eine Übergabe startet jedoch nicht die Verarbeitung eines Knotens.

Entscheidend sind noch die **Pufferknoten**. Eine Aktion kann Datentoken an einen Pufferknoten weitergeben und dort ablegen. Der Pufferknoten puffert diese und gibt sie nicht gleich an die nächste Aktion weiter, sondern sie werden erst später verwendet. Dazu gibt es den central buffer und den data store. Sie dienen dazu, mehrere Token zu speichern. Der Unterschied ist, dass beim **central buffer**, wenn die Aktion 2 einen Token fordert, dieser Token aus dem Puffer entnommen und weitergegeben wird, d.h. der Token wird aus dem central buffer gelöscht. Beim **data store** wird vom Token eine lokale Kopie angelegt, die an die Aktion weitergegeben wird, aber das Objekt selbst bleibt im data store persistent erhalten. Dementsprechend ist der central buffer ein transiente Pufferknoten, der Datentoken aus dem Puffer löscht, sobald sie weitergegeben werden. Der data store ist ein persistenter Pufferknoten, hier wird der Token nicht gelöscht, wenn er weitergegeben wird, sondern er bleibt im Puffer erhalten.

Objektknoten (5/7)

- Beispiel: Erweiterung - Einladung im CALENDARIUM
- alle Benutzer werden im Puffer "Benutzer" gespeichert (Datenbank)
- Ausgewählte Benutzer werden in einem transienten Puffer zwischengespeichert und erst für die Versendung von Einladungen wieder entnommen.



Wir kommen zu einem weiteren Beispiel.

Wie immer starten wir am Initialknoten. Dann wird „BenutzerID prüfen“ ausgeführt. Ist diese BenutzerID ok, wird der Kontrollflusstoken gesplittet und je einer weitergereicht an „Termin erfassen“ und „Teilnehmer auswählen“. Bei „Teilnehmer auswählen“ wird irgendein SELECT ausgeführt. Z.B. werden zwei Token ausgewählt. Es handelt sich um einen data store, d.h. von den ausgewählten Token werden lokale Kopien erzeugt, die an „Teilnehmer auswählen“ weitergegeben werden. Die beiden Teilnehmer werden über die Objektflusskante an den central buffer übergeben. Wurden „Termin erfassen“ und „Teilnehmer auswählen“ ausgeführt, werden die Kontrollflusstoken weitergereicht. Beim Synchronisierungsknoten werden die beiden Token vereinigt und dieser eine Token wird zu „Kollision prüfen“ weitergereicht. Die beiden ausgewählten Teilnehmer werden aus dem central buffer weitergegeben an „Kollision prüfen“ und aus dem central buffer gelöscht. Nach dem Ausführen von „Kollision prüfen“ wird der Token an den Aktivitätsendknoten weitergegeben und damit sind wir am Ende.

Das war also ein Beispiel für data store und central buffer.

Objektknoten (6/7)

■ Ausgabeparameter für Ausnahmen

- Werden nur im Falle des Auftretens einer Ausnahme weitergegeben (gewöhnliche Parameter werden in diesem Fall nicht weitergegeben)



■ Gruppierung von Parametern – Parametersatz

- Nur ein ausgewählter Parametersatz ist jeweils für die Ausführung der Aktion relevant - Spezifikation von alternativen, einander ausschließenden Gruppen von Ein- bzw. Ausgabewerten



24 

Wir kommen zu einer speziellen Art von Parametern, nämlich den **Parametern für Ausnahmen**. Hier wird im Normalfall kein Parameter übergeben, sondern nur unter einer Ausnahmebedingung. Das wird gekennzeichnet mit dem dargestellten roten Dreieck. D.h. der Objektfluss tritt nur im Ausnahmefall auf. Des weiteren können auch **Streams** übergeben werden. Während normale Parameter als weiße Pins dargestellt werden, werden Streams als schwarze Pins dargestellt. Dies sieht man auch an dem untern Beispiel, wo wir links oben einen Stream als Input haben und rechts unten einen Stream als Output.

An diesem Beispiel werden aber auch **Parametersätze** erläutert. Im Normalfall ist es so, dass wenn eine Aktion oder Aktivität z.B. vier Eingangspins besitzt, die Parameter miteinander AND-verknüpft sind. Es gibt also vier Parameter und diese müssen alle belegt sein, damit weitergearbeitet werden kann. Jetzt möchte man aber auch modellieren können, dass z.B. entweder die oberen beiden Parameter belegt sein müssen oder die unteren beiden. Wenn man das mit Operationen vergleicht, so handelt es sich also um eine überladene Operation die einmal die beiden oberen Parameter verlangt und einmal die beiden unteren. Und um das darzustellen, kann man sogenannte Parametersets verwenden. Dazu werden je die Parameter innerhalb eines Sets mit einem Rahmen umgeben. Dementsprechend werden die ersten beiden Parameter mit einem Rahmen umgeben und der dritte und der vierte Parameter mit einem Rahmen umgeben. Die Semantik ist dann folgendermaßen: Die Parameter innerhalb eines Parametersets sind AND-verknüpft und müssen allesamt vorhanden sein. Und die Parametersets selbst sind mit einander ENTWEDER/ODER-verknüpft. D.h. entweder die ersten beiden Parameter müssen vorhanden sein, oder die beiden anderen Parameter müssen vorhanden sein.

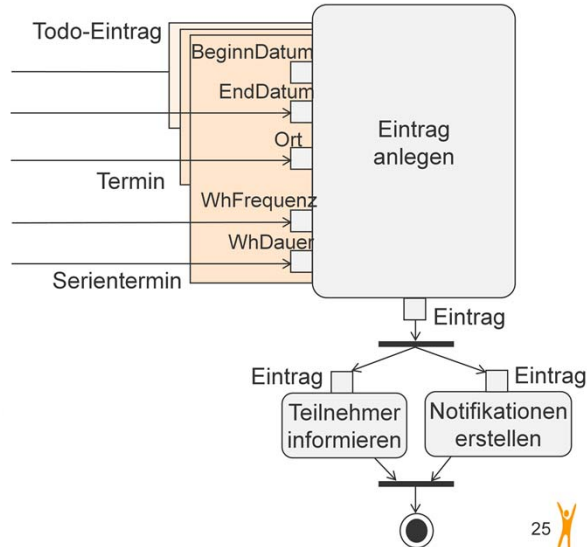
Die Parametersets können sich auch überschneiden, das sehen wir auch in dem Beispiel auf der Folie. D.h. man könnte auch Pin 2 und Pin 3 zu einem Parameterset zusammenfügen. Dann müssten entweder Parameter 1 und Parameter 2 oder Parameter 2 und Parameter 3 oder Parameter 3 und Parameter 4 übergeben werden. Hier habe ich zwei Eingangsparameter. Was müsste ich machen, wenn ich möchte, dass entweder der eine Parameter oder der andere Parameter übergeben werden müssen? – Dann muss ich beide einzeln umranden. Ich erzeuge gewissermaßen zwei Parametersets, wobei jedes dieser Parametersets nur einen Parameter hat.

In diesem Beispiel haben wir also ein Parameterset mit dem zweiten und dritten Pin (von oben gezählt) und ein Parameterset mit dem dritten, dem vierten und dem fünften Pin. Und nachdem der Pin für den Stream in keinem Parameterset enthalten ist, muss dieser immer übergeben werden. D.h. es müssen entweder der Stream und der zweite und der dritte Pin oder der Stream und der dritte und der vierte und der fünfte Pin als Input übergeben werden.

Objektknoten (7/7) – Bsp.: Anlegen eines Kalendereintrags

3 Terminarten:

- Todo-Eintrag
 - + BeginDatum
 - + EndDatum
- Termin
 - + BeginDatum
 - + EndDatum
 - + Ort
- Serientermin
 - + BeginDatum
 - + EndDatum
 - + Ort
 - + WhFrequenz
 - + WhDauer



Hier sehen wir ebenfalls das Überlagern der Parametersets. In diesem Beispiel werden entweder „BeginnDatum“ und „EndDatum“ übergeben, das ist der äußerste Rahmen. Oder es werden „BeginnDatum“, „EndDatum“ und „Ort“ übergeben, oder „BeginnDatum“, „EndDatum“, „Ort“, „WhFrequenz“ und „WhDauer“. Je nachdem wie die Pins belegt sind, startet „Eintrag anlegen“, erzeugt einen Eintrag und gibt diesen Token weiter, der am Parallelisierungsknoten vervielfältigt wird und die beiden Token werden dann an die Aktionen „Teilnehmer informieren“ und „Notifikation erstellen“ weitergereicht. Diese werden durchgeführt. Danach wird je ein Kontrolltoken an den Synchronisierungsbalken weitergereicht. Die beiden Kontrolltoken werden vereinigt und an den Aktivitätsendknoten weitergegeben. Damit ist diese Aktivität zu Ende.

Objektfluss (1/4)

- Hat eine **Transport-** und eine **Kontrollfunktion**
- **Verknüpft** Aktionen nicht direkt, sondern **über Objektknoten**
- Objektknoten bestimmen den Typ der zu transportierenden Objekte
- **Steuerungsmöglichkeiten** der Weitergabe von Datentoken:
 - Reihenfolge
 - Kapazitätsobergrenze und Gewicht
 - Selektionsverhalten
 - Transformationsverhalten

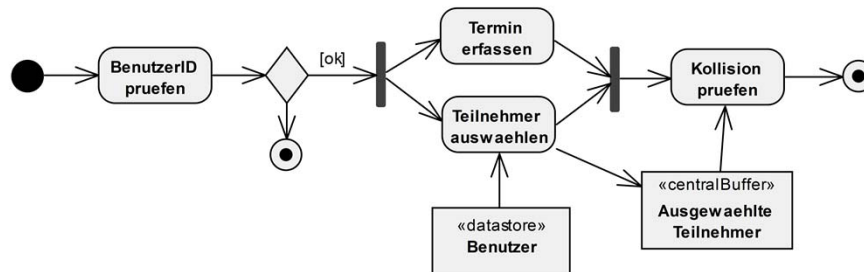


Der Objektfluss hat eine **Transport-** und eine **Kontrollfunktion**. Die Objekte werden transportiert und die Pfeilrichtung kontrolliert, wie der Fluss weitergeht. Aktionen werden nicht direkt, sondern über Objektknoten verknüpft. Objektknoten bestimmen den Typ der zu transportierenden Objekte.

Wir haben **Steuerungsmöglichkeiten** für die Weitergabe der Datentoken, die auf den nächsten Seiten dargestellt sind. Wir können die Reihenfolge der Weitergabe festlegen. Wir haben Obergrenzen und Gewicht. Wir können ein Selektionsverhalten angeben und ebenso ein Transformationsverhalten.

Objektknoten (5/7)

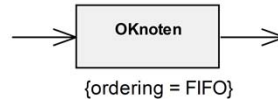
- Beispiel: Erweiterung - Einladung im CALENDARIUM
- alle Benutzer werden im Puffer "Benutzer" gespeichert (Datenbank)
- Ausgewählte Benutzer werden in einem transienten Puffer zwischengespeichert und erst für die Versendung von Einladungen wieder entnommen.



Gehen wir noch einmal zurück zu unserem Beispiel mit central buffer und data store. In diesem Beispiel hatten wir die Teilnehmer aus dem data store einfach zufällig ausgewählt. Jetzt könnten wir dafür eine Reihenfolge angeben, z.B. könnten wir im Falle einer Seminaranmeldung jene Teilnehmer auswählen, die schon am längsten warten. Es kommt als erstes der dran, der am längsten schon in der Queue ist. Das entspricht also einer First In First Out FIFO Reihenfolge. Oder wie in der Stapelverarbeitung könnte der letzte eingegangene als erstes dran kommen, das wäre also LIFO Last In First Out Reihenfolge. Oder man könnte ein Selektionskriterium angeben wie: Alle mit einer Matrikelnummer, die am Ende eine 8 hat, werden ausgewählt.

Objektfluss (2/4) – Reihenfolge der Tokenweitergabe

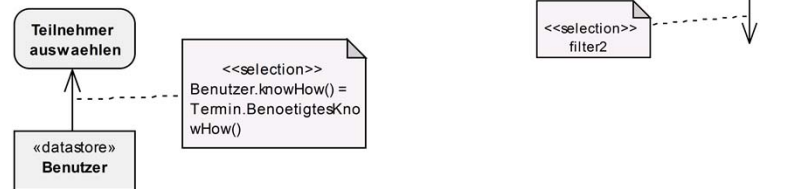
- Explizites Festlegen der Reihenfolge, in der ein **Datentoken** an eine ausgehende Objektflusskante weitergegeben werden kann
 - **FIFO** (first in, first out) - {ordering = FIFO}
 - Token werden in jener Reihenfolge weitergegeben, in der sie den Objektknoten erreichen (default)
 - **LIFO** (last in, first out) - {ordering = LIFO}
 - Token, die zuletzt eingegangen sind, werden als erste weitergegeben
 - **Geordnet** - {ordering = ordered}
 - benutzerdefinierte Reihenfolge (Angabe von Selektionsverhalten)
 - **Ungeordnet** - {ordering = unordered}
 - Reihenfolge in der die Token eingehen, hat keinen Einfluss auf die Reihenfolge, in der sie weitergereicht werden



Auf dieser Folie sind also die verschiedenen Möglichkeiten der Reihenfolge der Token-Weitergabe angeführt. Ich könnte explizit eine Reihenfolge festlegen. Beispielsweise mit First In First Out FIFO, Last In First Out LIFO, geordnet nach einem Selektionsverhalten oder eben absolut ungeordnet.

Objektfluss (3/4) - Selektionsverhalten

- Wählt bestimmte Token zur Weitergabe aus
- Objektknoten und Objektflusskanten können Selektionsverhalten aufweisen
- Selektionsverhalten – z.B. in Form einer Aktivität – muss einen Eingabe- und einen Ausgabeparameter aufweisen
- Beispiel:



Auf dieser Folie wird die Angabe eines Selektionskriteriums illustriert. Im illustrativen Beispiel werden jene Benutzer ausgewählt, die über das benötigte Know How verfügen.

Objektfluss (4/4)

- **Kapazitätsobergrenze** eines Objektknotens

- max. Anzahl von Token, die sich zu einem Zeitpunkt in diesem Knoten befinden dürfen

OKnoten1 {upperBound = Wert}

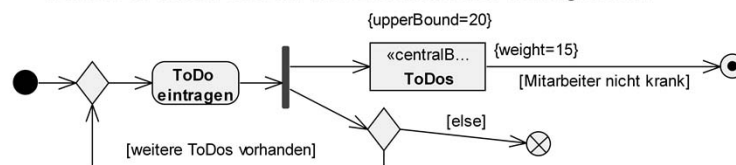
- **Gewicht** einer Objektflusskante:

- Anzahl der Token die anliegen müssen, bevor sie an Nachfolgeknoten weitergegeben werden

OKnoten2 {weight = Wert}

- **Beispiel: Pufferknoten kann max. 20 ToDos aufnehmen.**

- Wenn mind. 15 ToDos vorhanden sind und der Mitarbeiter nicht krank ist, werden 15 davon an den Aktivitätseinknoten weitergereicht

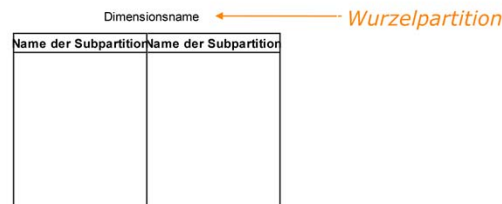


Hier haben wir noch das Konzept der **Kapazitätsobergrenze**. Wir können für jeden Objektknoten eine Obergrenze angeben, d.h. die maximale Anzahl an Token, die ein Knoten speichern kann. Zusätzlich kann man auch ein **Gewicht** angeben. Das Gewicht drückt aus, wie viele Token am Objektknoten anliegen müssen, bevor sie an den Nachfolgeknoten weitergegeben werden. Dieses Gewicht ist per Default immer 1, denn wenn eine Aktion ausgeführt ist, wird der Token im Normalfall gleich weitergegeben. Wenn das Gewicht 5 angegeben ist, müssen erst 5 Token erzeugt werden und dann werden alle 5 auf einmal weitergegeben.

Sehen wir uns das Beispiel an. Zunächst wird „ToDo eintragen“ ausgeführt, dann wird der Token gesplittet. Einer der Token wandert hinauf in den central buffer, einer hinunter und erzeugt in einer Schleife weitere ToDos. Wenn diese Schleife 15mal durchlaufen wurde, liegen 15 Token im central buffer. Wenn 15 Token anliegen, werden diese an die nächste Aktion weitergegeben, jedoch nur, wenn die Überwachungsbedingung, dass der Mitarbeiter nicht krank ist, erfüllt ist. Sollte die Überwachungsbedingung nicht erfüllt sein, d.h. der Mitarbeiter ist krank, so bleiben die 15 Token weiterhin im central buffer. Im central buffer können dann noch bis zu weitere 5 Token aufgenommen werden, weil ja die Kapazitätsobergrenze des central buffers mit 20 beschränkt ist.

Partitionen

- Erlauben die **Gruppierung von Knoten und Kanten** einer Aktivität nach bestimmten Kriterien
- **Logische Sicht auf eine Aktivität** zur Erhöhung der Übersichtlichkeit und Semantik des Modells
- »Schwimmbahnen«-Notation (partitions, swimlanes)
- Hierarchische Partitionen
 - Zur Schachtelung auf verschiedenen Hierarchieebenen

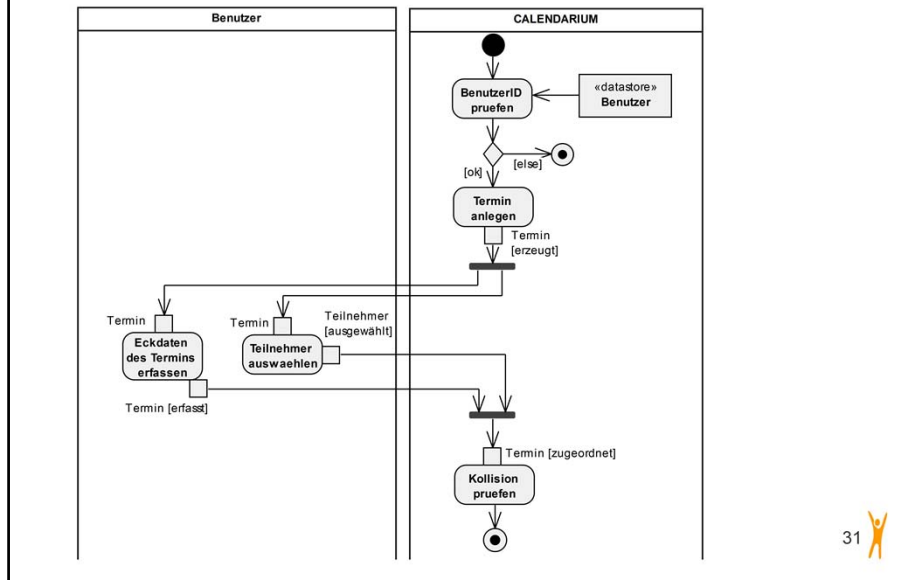


30



Ein weiteres wesentliches Konzept sind die sogenannten **Partitions**. Diese haben früher auch Schwimmbahnen oder swim lanes geheißen. Mit Hilfe von Partitions kann ich Aktionen zusammenfassen und diese Classifiern zuordnen. Partitionen können horizontal oder vertikal angegeben werden. Aktionen werden gewissen Partitions zugewiesen. In jeder Partition wird dann ein Classifier angegeben, der verantwortlich für die Ausführung der zugeordneten Aktionen ist.

Partitionen – Bsp.: Koordination von Terminen mit 2 Partitionen "Benutzer" und "CALENDARIUM"



In diesem Beispiel werden die Aktionen „BenutzerID prüfen“, „Termin anlegen“ und „Kollision prüfen“ von der Kalender-Anwendung CALENDARIUM ausgeführt. Daher befinden sich alle diese Aktionen in der Partition welche dem CALENDARIUM zugeordnet ist.

Der Benutzer führt die Aktionen „Eckdaten des Termins erfassen“ und „Teilnehmer auswählen“ aus. Daher sind diese beiden Aktionen der Partition zugeordnet, der der Benutzer zugewiesen wurde.

Partitions werden meistens genutzt, wenn nur wenige Classifier involviert sind. Sind viele beteiligt, wird die graphische Anordnung schwierig. Partitions haben noch einen weiteren Nachteil. Sie können Aktionen nur dann in einer Partition zusammenfassen, wenn diese eindeutig einem Classifier zugeordnet werden kann. Man kann also damit nicht darstellen, dass zwei Classifier gemeinsam für eine Aktion zuständig sind. Sie können eine Partition nämlich nicht mehreren Classifiern zuordnen. Somit können Sie nicht darstellen, dass gewisse Aktionen gemeinsam von zwei Akteuren als Classifier durchgeführt werden.

Signale und Ereignisse

- Sonderformen von Aktionen
- Senden von Signalen:



- Empfangen von Ereignissen:

Asynchrones Ereignis



Asynchrones Zeitereignis

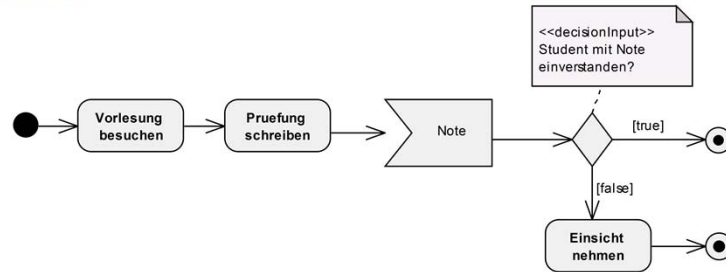


Hier lernen wir noch besondere Aktionen und zwar **Signale und Ereignisse** kennen. Das sind Sonderformen von Aktionen. Sie erinnern sich, am Anfang der Einheit hat es geheißen, dass es 44 verschiedene Arten von Aktionen gibt und diese teilweise ihre eigene Notationsform haben. Und das sehen Sie hier am Beispiel der Signale und Ereignisse. Wir haben hier das **Senden von Signalen**, das Empfangen eines **asynchronen Ereignisses** und das **asynchrone Zeitereignis** wie z.B. „Monatsende“.

Das bedeutet, es gibt das Senden eines Signales. Beim asynchronen Ereignis wird erwartet, dass ein Signal eingeht. Und beim asynchronen Zeitereignis tritt ein gewisser Umstand im zeitlichen Kontext ein.

Bsp.: Asynchrones Ereignis

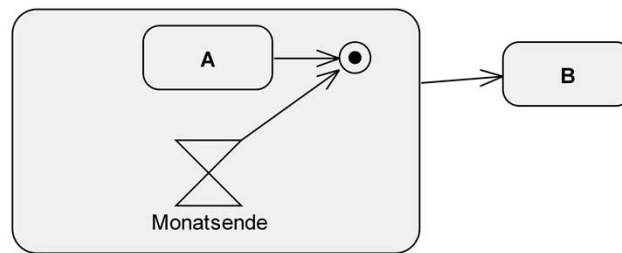
- Um eine Vorlesung zu absolvieren, besucht ein Student zuerst die Vorlesung, dann schreibt er eine Prüfung und wartet auf die Note. Der Student wird informiert, sobald die Note verfügbar ist. Nachdem der Student die Note erfahren hat, besteht die Möglichkeit, Einsicht zu nehmen.



In diesem Beispiel haben wir zuerst die Aktion „Vorlesung besuchen“, dann „Prüfung schreiben“ und aus Ihrer Sicht als Student warten Sie dann als nächstes, dass die Note kommt. Das ist ein Signal, das sie empfangen, deshalb wird das hier als asynchrones Ereignis dargestellt. Man hätte statt diesem asynchronen Ereignis auch einfach eine Aktion „Note empfangen“ einzeichnen können.

Das asynchrone Ereignis setzt man oft ein, wenn man einen Ablauf modelliert, der nur fortgesetzt werden kann, wenn von außen ein Ereignis eintritt. Das asynchrone Ereignis ist damit ähnlich wie ein Initialknoten, nur dass er nicht automatisch belegt wird, sondern dass gehorcht wird, bis ein Signal von außen eingeht.

Bsp.: Asynchrones Zeitereignis



Nun möchte ich noch ein Beispiel zum asynchronen Zeitereignis zeigen.

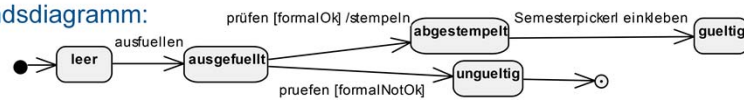
Angenommen wir haben Aktion A und Aktion B. Aktion A wird relativ lange ausgeführt aber Sie möchten, dass am Monatsende auf jeden Fall Aktion B ausgeführt wird, auch wenn Aktion A noch nicht fertig ist.

Aber wie stellen Sie an? – Dazu wird eine Aktivität eingeführt und innerhalb dieser Aktivität wird die Aktion A ausgeführt. Außerdem gibt es ein asynchrones Zeitereignis „Monatsende“. Wird der Token an diese Aktivität weitergegeben, wird die Aktion A gestartet. Wenn Sie mit Aktion A fertig sind, wird die Aktivität durch den Aktivitätsendknoten beendet und Sie können weiter zu Aktion B gehen. Ist aber zuerst das Monatsende erreicht, gehen Sie auch zum Aktivitätsendknoten, was die ganze Aktivität beendet und damit gehen Sie ebenfalls weiter zu Aktion B.

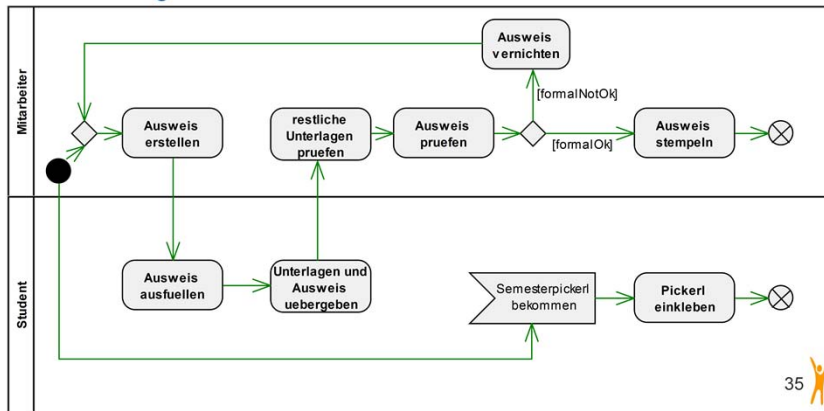
Dieses Konzept ist z.B. dann nützlich, wenn ein Kunde 5 Tage Zeit hat, um auf etwas zu antworten und Sie diesen Zeitablauf modellieren wollen.

Beispiel: Studentenausweis (1/3)

■ Zustandsdiagramm:



■ Aktivitätsdiagramm - Kontrollfluss:



Jetzt kommen wir noch zu einem umfassenden Beispiel.

Wir haben oben das Zustandsdiagramm für den Studentenausweis angegeben. Unser Studentenausweis ist ein Objekt, das sich in verschiedenen Zuständen befindet. Er hat hier den Zustand „leer“, den Zustand „ausgefüllt“, den Zustand „ungültig“, den Zustand „abgestempelt“ und den Zustand „gültig“. Die Zustandsfolge geht von „leer“ nach „ausgefüllt“ nach „abgestempelt“ und nach „gültig“ bzw. von „ausgefüllt“ nach „ungültig“.

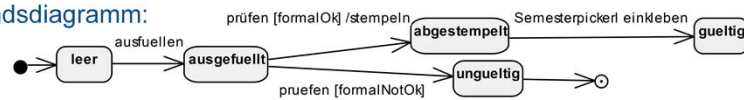
Darunter habe ich einen Kontrollfluss dargestellt mit zwei Partitions, je eine für den Studenten und eine für den Mitarbeiter.

Wir starten beim Initialknoten. Dieser hat zwei ausgehende Kanten. Auf jede dieser Kanten wird ein Token gelegt. Gemäß der unteren Kante wird das Warten auf ein Semesterpickerl aktiviert. Mit der oberen Kante gehen wir in einen Vereinigungsknoten, da eine Kante belegt ist gehen wir unmittelbar zu „Ausweis erstellen“. Diese Aktion führt der Mitarbeiter aus. Der Mitarbeiter gibt den Ausweis weiter an den Studenten, der ihn ausfüllt. Der Student übergibt dann die Unterlagen und den Ausweis an den Mitarbeiter. Der Mitarbeiter überprüft die restlichen Unterlagen. Dann prüft er den Ausweis. Wenn dieser Aufweis formal nicht in Ordnung ist, vernichtet der Mitarbeiter den Ausweis und der Prozess beginnt von vorne. Bzw. wenn der Ausweis formal ok ist, dann stempelt er den Ausweis und der Token kommt zum Ablaufknoten, wo er gelöscht wird. Wir haben jedoch immer noch das Horchen, ob das Semesterpickerl bereits angekommen ist, aktiviert. D.h. als Student warten Sie darauf, dass Sie das Semesterpickerl bekommen. Sobald Sie es bekommen, kleben Sie es in Ihren Studentenausweis ein. Damit gelangen Sie zum Ablaufknoten und damit wird der Token gelöscht und die Aktivität wird beendet, da es keine Token mehr gibt.

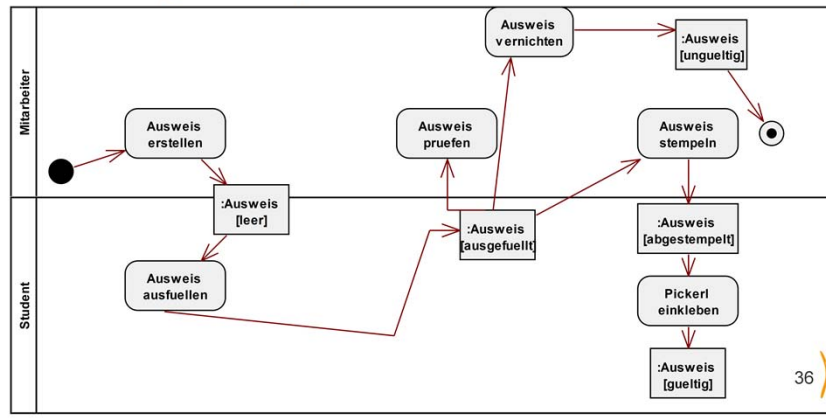
Hier ist es wichtig, dass Sie die Kante vom Initialknoten zu „Semesterpickerl bekommen“ angeben, denn sonst passiert es Ihnen unter Umständen, dass alle Token schon vorher aus dem Diagramm entfernt wurden und damit die gesamte Aktivität beendet ist, weil das „Semesterpickerl bekommen“ nur solange horcht, solange die Aktivität aktiv ist.

Beispiel: Studentenausweis (2/3)

■ Zustandsdiagramm:



■ Aktivitätsdiagramm - Objektfluss:

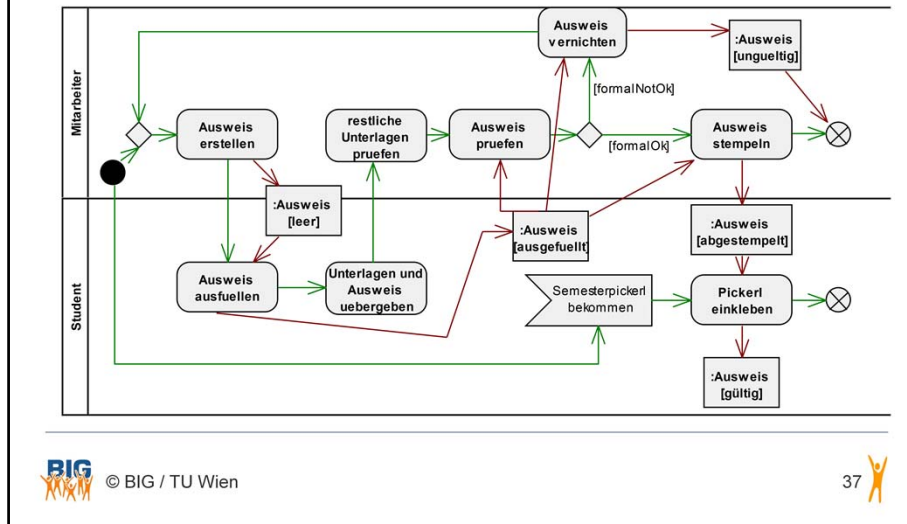


Und jetzt zeichnen wir auf Basis des Zustandsdiagramms und des Kontrollflusses den Objektfluss mit den Objekten und deren Zustände ein.

Wir starten bei „Ausweis erstellen“. „Ausweis erstellen“ hat als Output einen leeren Ausweis, der Input für „Ausweis ausfüllen“ ist. Der Zustand „leer“ wird als Zustand für das Objekt im Objektknoten angegeben. Wenn der Ausweis ausgefüllt wurde, befindet er sich im Zustand „ausgefüllt“. Er ist nun Input für „Ausweis prüfen“, und für „Ausweis vernichten“ oder „Ausweis stempeln“. Wenn der Ausweis vernichtet wird, kommt der Ausweis in den Zustand „ungültig“, wird er abgestempelt kommt er in den Zustand „abgestempelt“, wird das Pickerl eingeklebt, ist er „gültig“.

Beispiel: Studentenausweis (3/3)

- Kontrollfluss (grün) und Objektfluss (rot) in einem Diagramm



© BIG / TU Wien

37



Hier haben wir den Kontrollfluss in grün und den Objektfluss in rot gemeinsam in einem Diagramm dargestellt. Einige Kontrollflusskanten könnte ich mir im Diagramm sparen und zwar dann, wenn Kontroll- und Objektfluss gleich verlaufen. Z.B. die Kontrollflusskante von „Ausweis erstellen“ zu „Ausweis ausfüllen“ müsste ich nicht einzeichnen, aber zusätzlich Kanten einzuzichnen, ist nie ein Fehler, dadurch wird das Modell präziser und die Fehlerwahrscheinlichkeit wird minimiert, jedoch wird die Lesbarkeit beeinträchtigt. Wo Kontroll- und Datenfluss nicht übereinstimmen, muss ich beide auf alle Fälle einzeichnen. Das sieht man z.B. am Datenfluss der von „Ausweis ausfüllen“ nach „Ausweis prüfen“, „Ausweis vernichten“ und „Ausweis stempeln“ führt, während der Kontrollfluss von „Ausweis ausfüllen“ nach „Unterlagen und Ausweis übergeben“ verläuft. Hier stimmen Daten- und Kontrollfluss nicht überein, daher muss ich beide extra einzeichnen.

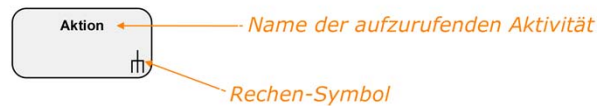
An diesem Beispiel sieht man, dass ein Aktivitätsdiagramm rasch komplex wird. Das gilt besonders dann, wenn Sie in einem Diagramm Objekt- und Kontrollfluss einzeichnen wollen. Einerseits sind die Aktivitätsdiagramme sehr einfach, man stellt einfach den Ablauf von Aktionen dar. Andererseits sind diese Diagramme oft sehr komplex und daher passieren auch hier sehr sehr viele Fehler.

Man muss immer unterscheiden, wozu man das Aktivitätsdiagramm verwenden möchte. Wenn man es als Kommunikationsmittel mit einem Betriebswirt verwendet, muss es meist nicht sehr präzise sein und man nimmt kleine Ungereimtheiten in Kauf.

Verwendet man es jedoch zur Spezifikation in einem Workflow-System, dann ist das Aktivitätsdiagramm eine graphische Notation für ein Diagramm, das von einer Workflow-Engine verarbeitet werden soll. Dann muss das Diagramm präzise und fehlerfrei sein. Dementsprechend ist das Beschreiben von Workflows mittels Aktivitätsdiagrammen eine große Herausforderung, weil viele kleine Details sehr wichtig sind.

Schachtelung von Aktivitäten

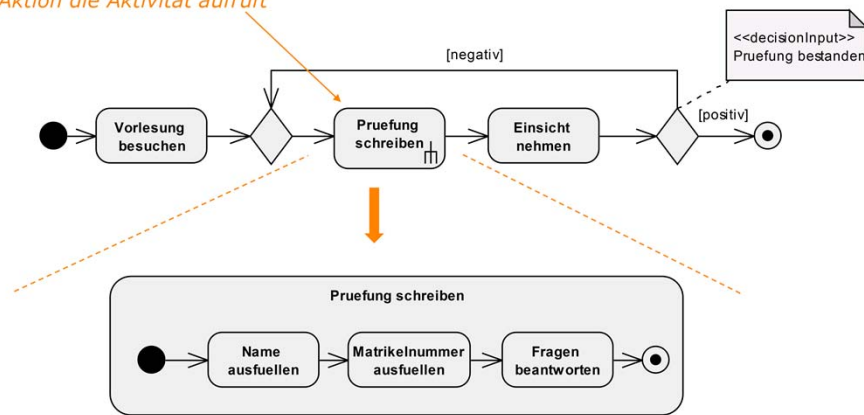
- Aktivitäten können wiederum Aktivitäten aufrufen
- So können Details in eine tiefere Ebene ausgelagert werden
- Vorteile:
 - Bessere Lesbarkeit
 - Wiederverwendung
- Notation:
 - In einer Aktion wird eine Aktivität aufgerufen



Innerhalb einer Aktivität kann eine Aktion wiederum eine andere untergeordnete Aktivität aufrufen. Somit können Aktivitäten auch **geschachtelt** werden. Damit können Details in eine tiefere Ebene ausgelagert werden. Dadurch erhöht sich natürlich die Lesbarkeit und wir haben ein Konzept der Wiederverwendung. Wir haben hier eine Aktion dargestellt, in der rechts unten ein **Rechen-Symbol** aufgeführt ist. Das dient als Aufruf einer untergeordneten Aktivität, sozusagen vergleichbar mit einem Unterprogramm-Aufruf.

Schachtelung von Aktivitäten - Beispiel

Aktion die Aktivität aufruft



© BIG / TU Wien

39



Wir sehen uns den folgenden Ausschnitt aus diesem Beispiel an.

In der oberen Aktivität ist eine Aktion „Prüfung schreiben“. „Prüfung schreiben“ ist selbst eine Aktion, weil das Aufrufen einer Aktivität ja selbst atomar ist. Die Aktion „Prüfung schreiben“ ruft dann die Aktivität „Prüfung schreiben“ auf. D.h. im oberen Diagramm handelt es sich bei „Prüfung schreiben“ um eine Aktion in der das Rechen-Symbol eingezeichnet ist. Diese Aktion ist atomar, es handelt sich um einen Aufruf der Aktivität und dadurch werden „Name ausfüllen“, „Matrikelnummer ausfüllen“ und „Fragen beantworten“ als Teil von „Prüfung schreiben“ ausgeführt. Was erreiche ich damit? – Modularität, d.h. ich kann Teile in verschiedenen Aktivitäten wiederverwenden. Dementsprechend könnte ich die Aktivität „Prüfung schreiben“ auch noch in weiteren Aktivitäten, außer in der oben dargestellten, noch wiederverwenden.

Ausnahmebehandlung – Exception Handler (1/2)

- **Vordefinierte Ausnahmen**, beispielsweise durch das Laufzeitsystem (z.B. Division durch 0)
- **Benutzerdefinierte Ausnahmen**
 - `RaiseExceptionAction`
- Behandlung einer Ausnahme durch **dezidierten Ausnahmebehandlungsknoten** – nach Abarbeitung der Ausnahme kann mit dem "normalen" Ablauf fortgefahren werden
- Der Ausnahmebehandlungsknoten **substituiert** den „geschützten“ Knoten und hat daher keine eigenständigen ausgehenden Kontroll- oder Objektflüsse
- Notation:

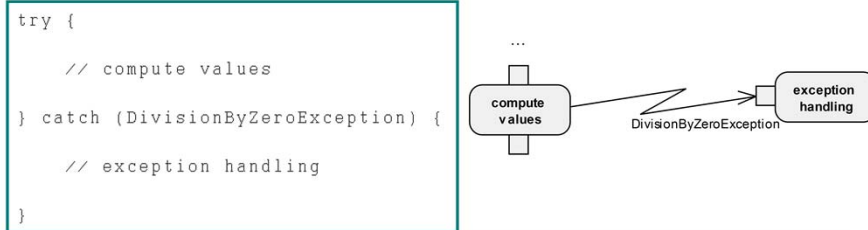


Als letzten Punkt behandeln wir jetzt noch das **Exception Handling**. Im wesentlichen gibt es zwei verschiedene Varianten um ein Exception Handling zu modellieren.

Die erste Variante sieht folgendermaßen aus: Sollte in einer Aktion eine Exception geworfen werden, dann gehe ich mit diesem Blitz-artigen Pfeil aus der Aktion hinaus, führe das Exception Handling durch und kehre dann aber wieder in den normalen Ablauf des Aktivitätsdiagramms zurück.

Ausnahmebehandlung – Exception Handler (2/2)

- Existiert für einen Ausnahmetyp keine Ausnahmebehandlung, wird die betroffene Aktion beendet und die Ausnahme nach außen propagiert (d.h. es wird in der umgebenden Aktivität nach passender Ausnahmebehandlung gesucht)
- Beispiel

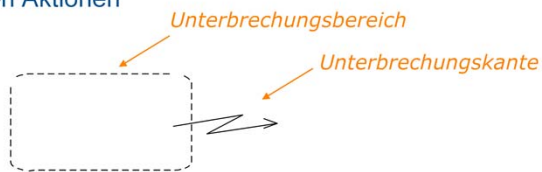


Dazu ein einfaches Beispiel.

Hier sehen wir, dass „compute values“ zunächst ausgeführt wird. Tritt eine „DivisionByZeroException“ auf, wird das „exception handling“ ausgeführt, danach wird der Token zurück an „compute values“ gegeben und dann wird im normalen Ablauf fortgesetzt.

Ausnahmebehandlung – Unterbrechungsbereich (1/2)

- Umschließt 1-n Aktionen
- Notation:



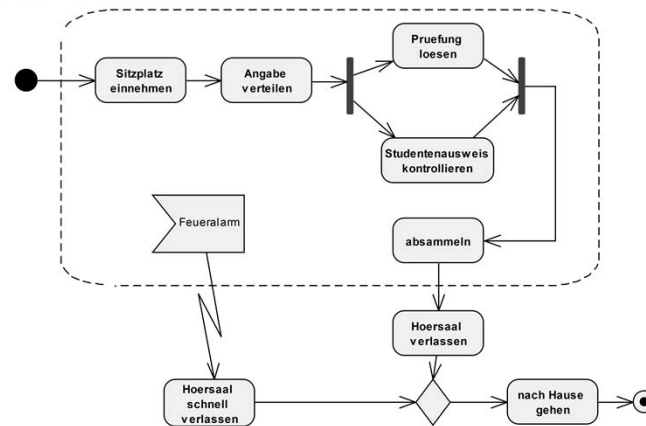
- Wird der Unterbrechungsbereich über die Unterbrechungskante verlassen, so werden alle in der Region vorhandenen Token gelöscht



Als zweite Variante der Ausnahmebehandlungs-Möglichkeiten gibt es den **Unterbrechungsbereich**. Mit einem Unterbrechungsbereich lege ich fest, falls in diesem selbstdefinierten Unterbrechungsbereich, der eine oder auch mehrere Aktionen bzw. Aktivitäten enthalten kann, eine Ausnahmesituation auftritt, so wird der Unterbrechungsbereich über die Blitz-artige Unterbrechungskante verlassen, aber man kehrt dann nicht mehr in den normalen Ablauf zurück. D.h. alle Token, die sich im Unterbrechungsbereich befunden haben, werden gelöscht.

Ausnahmebehandlung – Unterbrechungsbereich (2/2)

■ Bsp.: Prüfung schreiben






Es folgt nun noch ein Beispiel zum Unterbrechungsbereich.

Das Modell hier beschreibt was passiert, wenn beim Schreiben einer Prüfung ein Feueralarm ausbricht. Wir starten zunächst regulär mit unserer Aktivität. D.h. als erstes führen wir die Aktion „Sitzplatz einnehmen“ durch. Dann wird die Angaben verteilt. Anschließend kommen wir zu den Aktionen „Prüfung schreiben“ und „Studentenausweis kontrollieren“. Und angenommen, jetzt geht der Feueralarm los. Weil der Feueralarm im Unterbrechungsbereich los geht, wird der Token weitergereicht an „Hörsaal schnell verlassen“. Alle Token im Unterbrechungsbereich werden gelöscht. Als nächste Aktion wird „nach Hause gehen“ ausgeführt und dann ist die Aktivität zu Ende.

Signale werden häufig im Unterbrechungsbereich verwendet um darzustellen, dass wenn ein spezielles Ereignis eintritt, der Unterbrechungsbereich verlassen und alle Token gelöscht werden und sozusagen ein alternativer Ablauf ausgeführt wird. Beim Unterbrechungsbereich handelt es sich also um ein wichtiges Konzept um Sonderfälle zu behandeln.

Das UML-Konstrukt des Exception Handlings ist also äquivalent zum try-catch-Konstrukt in Java, während der Unterbrechungsbereich eher dazu dient, Sonderfälle darzustellen.

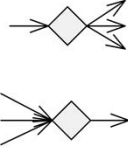
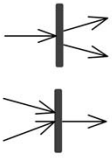
Basiselemente (1/5)

Name	Syntax	Beschreibung
Aktionsknoten		Repräsentation von Aktionen (Aktionen sind atomar!)
Initialknoten		Kennzeichnung des Beginns eines Ablaufs einer Aktivität
Aktivitätseend- knoten		Kennzeichnung des Endes ALLER Abläufe einer Aktivität



Damit sind wir am Ende des Aktivitätsdiagramms angelangt. Wie üblich folgt auf den folgenden Seiten noch einmal eine Übersicht über die wichtigsten Notationselemente des Aktivitätsdiagramms.




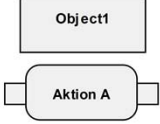
Basiselemente (2/5)

Name	Syntax	Beschreibung
Entscheidungs-/ Vereinigungs- knoten		Aufspaltung/Zusammenführung von alternativen Abläufen
Parallelisierungs-/ Synchronisations- knoten		Aufspaltung eines Ablaufs in nebenläufige Abläufe / Zusammen- führung von nebenläufigen Abläufen in einen Ablauf



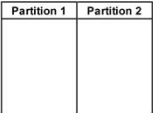

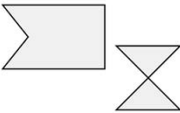
© BIG / TU Wien

Basiselemente (3/5)

Name	Syntax	Beschreibung
Ablaufend-knoten		Ende von EINEM Ablauf
Transition		Verbindung der Knoten einer Aktivität
Aktivitätsaufruf		Schachtelung von Aktivitäten
Objektknoten, Pins		Beinhalten Daten und Objekte


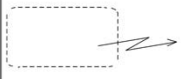


Basiselemente (4/5)

Name	Syntax	Beschreibung
Partition		Gruppierung von Knoten und Kanten innerhalb einer Aktivität
Signal		Übermittlung eines Signals an einen Empfänger
asynchrones Ereignis/ Zeitereignis		Warten auf ein Ereignis bzw. einen Zeitpunkt



Basiselemente (5/5)

Name	Syntax	Beschreibung
Exception Handler		Aktivität wird bei Ausnahme ausgeführt
Unterbrechungsbereich		Wird der Unterbrechungsbereich über die Unterbrechungskante verlassen, so werden alle in der Region vorhandenen Token gelöscht

Zusammenfassung

- Sie haben diese Lektion verstanden, wenn Sie wissen ...
- was mit dem Aktivitätsdiagramm modelliert wird.
- was Aktivitäten und Aktionen sind und wie Daten- und Kontrollfluss festgelegt werden.
- dass es unterschiedliche Knoten und Kanten im Aktivitätsdiagramm gibt.
- wie Nebenläufigkeit dargestellt wird.
- was Ein- und Ausgabepins sind.
- dass Aktivitätsdiagramme partitioniert werden können.

