# CourseCreator:
# Konzept zur Erweiterung auf neue Kurstypen

## Problembeschreibung

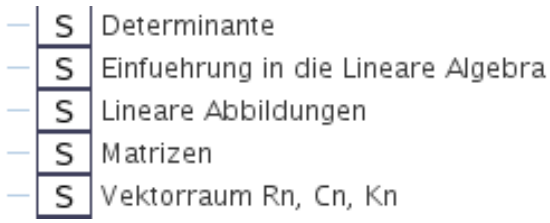
Der CourseCreator ist bisher darauf ausgelegt, Kurse zu erzeugen, die als Wissensnetzwerk angezeigt und gespeichert werden. Dabei beruht dieses Wissensnetzwerk auf:

- Knoten welchen Dokumente zugeordnet werden (im folgenden Dokumentknoten)
- Verzweigungsknoten
- Kanten, die die Knoten in eine logische Abhängigkeitsstruktur bringen
- dem Roten Faden, der die Dokumentknoten in eine Lineare Struktur bringt

Des weiteren kann ein solcher Graph auch Subdokumentknoten enthalten.

Jeder Knoten hat x- und y-Koordinaten, jede Kante hat mindestens zwei Stützknoten mit x- und y-Koordinaten, jeder Subdokumentknoten hat eine Ausrichtung (oben-links, oben-rechts, unten-rechts, unten-links) am Dokumentknoten... Im Folgenden wird solch ein Kurs als „Netzwerkkurs“ bezeichnet.

Mit dieser Struktur ist es möglich, beliebige Graphen zu erzeugen, allerdings kann man sich fragen, ob dies ausreicht, oder vielleicht auch „zu viel des Guten“ ist.




Der CourseCreator ist hervorragend geeignet, einen Kurs als Netzwerkgraphen darzustellen. In dieser Form ist die Ansicht und die Bearbeitung des Graphen sehr intuitiv und leicht verständlich. Möchte man aber zum Beispiel Graphen erzeugen, die viel einfacher sind, muss man sich fragen, ob Koordinaten, Stützknoten, Ausrichtungen, Rotem Faden usw. nicht zu viele Informationen in den XML-Dateien sind, die das System ungünstig belasten.

Neben der Speicherung des Graphen als XML-Dokument ist die Visualisierung des Graphen interessant. Kommen wir dafür zu einen fiktiven Beispiel. Der Benutzer möchte nur eine einfache Liste von Dokumenten erzeugen (z.B. eine Liste der Definitionen eines Kapitels), so erwartet er -unter Umständen- ein Bild welches vielleicht so aussieht:

Diese Darstellung kennt der Benutzer aus dem Elementzuweisungsfenster. Mit dem aktuellen CourseCreator erhält er aber nur ein Bild, welches für das gleiche Problem folgenden Bild erzeugen würde:

Um zum oben erwähnten überflüssig erzeugten Code zurückzukommen; der benötigte Code, der aus Bild 1 zu erzeugen ist, wäre (nur die Content-Datei, da die Metadatei gleich ist):

```
<crs:course xmlns:crs="http://www.mumie.net/xml-
namespace/document/content/course">
  <crs:list>
    <crs:course_section lid="1"/>
    <crs:course_section lid="2">
    <crs:course_section lid="3">
    <crs:course_section lid="4">
    <crs:course_section lid="5">
  </crs:list>
</crs:course>
```

der CourseCreator würde folgenden Code erzeugen:

```
<?xml version="1.0" encoding="US-ASCII"?>
<crs:course xmlns:crs="http://www.mumie.net/xml-
namespace/document/content/course">
   <crs:net>
     <crs:nodes>
       <crs:course_section lid="1" nid="1" posx="82" posy="32"/>
       <crs:course_section lid="2" nid="2" posx="82" posy="72"/>
       <crs:course_section lid="3" nid="3" posx="82" posy="112"/>
       <crs:course_section lid="4" nid="4" posx="82" posy="152"/>
       <crs:course_section lid="5" nid="5" posx="82" posy="192"/>
     </crs:nodes>
     <crs:arcs>
       <crs:arc from="1" to="2">
         <crs:point posx="82" posy="52"/>
         <crs:point posx="82" posy="52"/>
       </crs:arc>
       <crs:arc from="2" to="3">
         <crs:point posx="82" posy="92"/>
         <crs:point posx="82" posy="92"/>
       </crs:arc>
       <crs:arc from="3" to="4">
         <crs:point posx="82" posy="132"/>
         <crs:point posx="82" posy="132"/>
       </crs:arc>
       <crs:arc from="4" to="5">
         <crs:point posx="82" posy="172"/>
         <crs:point posx="82" posy="172"/>
       </crs:arc>
     </crs:arcs>
   </crs:net>
   <crs:thread>
     <crs:arcs/>
   </crs:thread>
</crs:course>
```

Zu bedenken ist, dass überflüssige Daten auch gern fehleranfällig sein können. Somit ist der CourseCreator so zu gestalten, dass er nur die Daten speichert, die er benötigt.

Aktuell gibt es zwei neue Kurstypen, die erzeugt werden sollen. Diese sind im folgenden Kapitel beschrieben. Da es sich im Folgenden nicht mehr um Graphen im eigentlichen Sinne handelt, wird ab jetzt der Begriff „Struktur“ verwandt.

## *neue Kurstypen*

### Hierarchische Struktur

Hierarchische Strukturen sind Bäume, deren Blätter die Dokumente zugeordnet werden. Eine

ähnliche Darstellung der Inhalte existiert im Prinzip schon im Elementzuweisungsfenster, da dort die Dokumente nach Themen sortiert in nummerierte Kapitel untergebracht sind. Außerdem soll es möglich sein, den Dokumentknoten Subdokumentknoten zuzuordnen. Diese Hierarchische Struktur entspricht der Darstellung von Inhalten in einem Buch.

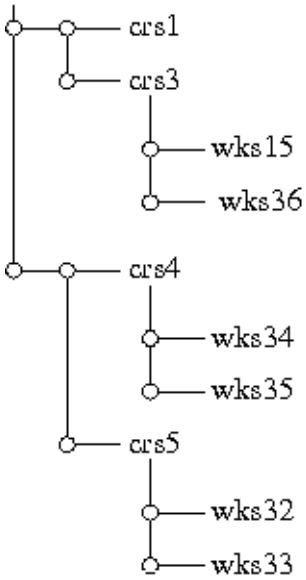
### *XML-Code*

Die Metadatei des Kurses unterscheidet sich nicht von der bisherigen Metadatei. Die Contentdatei muss nur einen Block enthalten, der die Hierarchie beschreibt. Dies kann exemplarisch in folgender Art geschehen:

```
<?xml version="1.0" encoding="US-ASCII"?>

<crs:course xmlns:crs="http://www.mumie.net/xml-
namespace/document/content/course">
  <crs:tree>
    <crs:headline>The headline</crs:headline>
   <crs:branch>
      <crs:headline>The headline</crs:headline>
      <crs:course_section lid="1"/>
      <crs:course_section lid="3">
        <crs:worksheet lid="15"/>
        <crs:worksheet lid="36"/>
      </crs:course_section>
    </crs:branch>
     <crs:branch>
        <crs:headline>The headline</crs:headline>
        <crs:course_section>
          <crs:worksheet lid="34"/>
          <crs:worksheet lid="35"/>
        </crs:course_section>
        <crs:course_section lid="5">
          <crs:worksheet lid="32"/>
          <crs:worksheet lid="33"/>
        </crs:course_section>
      </crs:branch>
   </crs:branch>
 </crs:tree>
</crs:course>
```

Dieses Beispiel würde folgenden Struktur erzeugen:




*Funktionalitäten*

Die Funktionalitäten, die diese Struktur benötigt, sind:

- (i) das Einfügen von neuen Blättern (später „leere“ Verzweigungen sind beim Anlegen auch nur Blätter), bzw. alternativ dazu
  (ii) das Einfügen von Blättern und Verzweigungen
- das Verschieben von einzelnen Teilbäumen (an eine beliebige Stelle, bzw. in Baum hoch- oder runter rücken, bzw. auf gleichem Level verschieben)
- Anhängen von Subdokumentknoten
- Versehen der Blätter bzw des Kurses mit Metainformationen (z.B. Labels bzw Summarys)
- Löschen von Blättern und Teilbäumen
- Laden/Speichern des spezifischen Formats
- Selektieren eines Knotens (zum Löschen des Selben, aber auch zum Anhängen von Subdokumentknoten und Zuweisen von Dokumenten)
- das Einklappen und Ausklappen von Teilbäumen

*welche Kursarten soll es geben?*

Bei Kursen: Blätter sind
- Kursabschnitte (course sections) mit angehängten Aufgabenblättern,
- Blöcke von nicht an Kursabschnitten angehängten Aufgabenblättern,
- einzelne Aufgabenblätter.

Bei Kursabschnitten: Blätter sind Elemente mit angehängten Subelementen.
Bei Aufgabenblättern: Blätter sind Aufgaben.

## Listenartige Struktur

Listenartige Kurse sind „einfache“ hierarchische Kurse. Genauer heißt dies, dass listenartige Kurse, anders als hierarchische Kurse, keine Verzweigungen haben. Der XML-Code sähe also wie folgt aus:

```
<crs:course xmlns:crs="http://www.mumie.net/xml-
namespace/document/content/course">
  <crs:list>
```

```
        <crs:course_section lid="1"/>
        <crs:course_section lid="3">
          <crs:worksheet lid="15"/>
          <crs:worksheet lid="36"/>
        </crs:course_section>
        <crs:course_section>
          <crs:worksheet lid="34"/>
          <crs:worksheet lid="35"/>
        </crs:course_section>
      </crs:list>
    </crs:course>
```

Diese Darstellung wäre zum Beispiel bei Aufgabensammlung sinnvoll, da diese häufig keinen Bezug zueinander haben und auch keine Reihenfolge benötigen.

### *Funktionalitäten*

Die Funktionalitäten die diese Struktur benötigt sind:

- das Einfügen von neuen Blättern
- das Verschieben von einzelnen Blättern
- Versehen der Blätter bzw des Kurses mit Metainformationen (z.B. Labels bzw Summarys)
- Anhängen von Subdokumentknoten
- (wiederholtes) Zuweisen von Dokumenten zu den (Sub-)Dokumentknoten
- Löschen von Blättern
- Laden/Speichern des spezifischen Format
- Selektieren eines Knotens (zum Löschen des Selben, aber auch zum Anhängen von Subdokumentknoten und Zuweisen von Dokumenten)

### *welche Kursarten soll es geben?*

Bei Kursen: Blätter sind
- Kursabschnitte (course sections) mit angehängten Aufgabenblättern,
- Blöcke von nicht an Kursabschnitten angehängten Aufgabenblättern,
- einzelne Aufgabenblätter.

Bei Kursabschnitten: Blätter sind Elemente mit angehängten Subelementen.
Bei Aufgabenblättern: Blätter sind Aufgaben.

Durch die andere XML-Struktur ist dieser Kurstyp als vom Hierarchiekurs verschieden anzusehen.

## Eine einfache Lösung

Als einfachste Lösung wäre die Anpassung des Netzwerkkurses an die neuen Voraussetzungen zu sehen. Das bedeutet, dass der bisherige Kurs um zwei graphische Oberflächen erweitert wird. Es werden also zwei graphische Darstellung implementiert, wobei eines die listenartigen, die andere die hierarchischen Kursen darstellt.

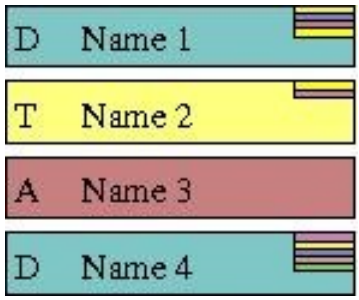


*Abbildung 1: Listenartiger Kurs*

### *Darstellung*

Als Grundlage der listenartigen Darstellung wird die Java-Klasse JList verwendet. Diese Klasse kann Daten als Liste darstellen, die einzelnen Elemente der Liste sind dann die bisherigen Dokumentknoten. Diesen können dann noch Subdokumentknoten angefügt werden. Die graphische Darstellung ist schwierig so zu gestalten, dass sie übersichtlich und detailliert ist. Daher wird die Liste aus farbigen

Rechtecken bestehen, an denen die Subdokumentknoten als ebenfalls farbige, gleichgroße Rechtecke an gehangen werden.

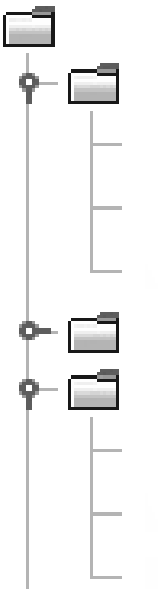


*Abbildung 2: Baum*

Die Darstellung des Kurses ist in Abbildung  gezeigt. Dieser Kurs hat 4 Dokumentknoten, wobei der erste 4, der zweite 2 und der letzte 5 Subdokumentknoten besitzen. Der dritte Dokumentknoten besitzt keine Subdokumentknoten.

Sehr ungünstig bei dieser Darstellung ist, das wird deutlich, dass bei den Subelementen die Kategorie der Knoten nicht mehr dazu geschrieben wird. Dies muss mit deutlich unterschiedlichen Farben der Subdokumentknoten geregelt werden, gegebenenfalls einer Legende und den lange fälligen Tooltips.

Für die Darstellung von hierarchischen Kursen werden ebenfalls die oben dargestellten Icons für die Dokumentknoten und die Subdokumentknoten verwendet, allerdings wird dem ein strukturierender Baum vorgesetzt. Siehe Abbildung 2. Als Grundlage hierfür wird die Klasse JTree verwendet.

### *XML*

Neben der Darstellung muss der Graph sich in verschiedene XML-Formate schreiben lassen. Dazu muss es für jeden Kurstypen eine Funktion geben, die angelehnt an die bisherige Funktionalität `toXML()` heißt und den notwendigen XML-Code erzeugt.

## *Kontrollstrukturen und Überführung der Formate*

Des weiteren benötigen die unterschiedlichen Kurstypen unterschiedliche Kontrollstrukturen, um zu prüfen, ob die Struktur „gültig“ ist. bei dem bekannten Netzgraphen ist dies zum Beispiel die Überprüfung ob der Rote Faden korrekt ist. Beim Tauschen vom Baumteilen der Hierarchiekurse muss geprüft werden, ob ein Tausch möglich ist, z. B. kann ein Knoten nicht gegen einen darüber liegenden Knoten getauscht werden.

Des Weiteren ist es möglich (und vermutlich auch sinnvoll) Kurstypen in andere zu überführen. Allgemein ist die Richtung Listenkurs --> Hierarchiekurs --> Netzwerkkurs immer möglich, die Rückrichtung Netzwerkkurs --> Hierarchiekurs nur, wenn der Netzwerkkurs keine Kreise enthält, sowie Hierarchiekurs --> Listenkurs wenn der Hierarchiekurs keine Verzweigungen enthält.

Dafür benötigt es zum einen der oben genannten Kontrollfunktionen, zum anderen Übergangsfunktionen, die einen Kurs als den anderen darstellen.

## *Warum nicht einfach den Netzwerkkurs erweitern?*

Der Netzwerkkurs arbeitet auf der Klasse JGraph, die genau die Anforderungen erfüllt, die für den Netzwerkkurs bedeutsam sind. Die Klasse ist darauf ausgerichtet, dass die zugrunde liegenden Daten untereinander abhängig sind (Kanten). Listenkurse und Hierarchiekurse haben diese Eigenschaften nicht. Auch die erwarteten Darstellungen kann die Klasse JGraph nicht bieten.

Die Funktionen, die für die Listenkurse und die Hierarchiekurse erwartet werden, sind „vom Gefühl her“ sehr ähnlich.

Das Verschieben von Teilgraphen im Hierarchiekurs ist im Prinzip das Gleiche wie das Vertauschen von Knoten im Netzwerkkurs, jedoch ungleich komplizierter, da ja noch der darunter liegende Teilbaum verschoben werden muss. Dies ist beim Netzwerkkurs nicht nötig.

Auch das Einfügen von neuen Komponenten ist sehr unterschiedlich. Beim Netzwerkgraphen wird nur eine leere Komponente an eine feste Stelle (oben links) eingefügt und diese Komponente dann an die gewünschte Stelle verschoben. Beim Hierarchiekurs ist es aber sinnvoll (und mit dem JTree machbar) die Komponente gleich an der richtigen Stelle einzufügen.

Die Klasse JTree und JList sind daher viel besser für den Listenkurs, bzw. den Hierarchiekurs geeignet.

Trotz alledem ist eine ordentliche Programmierung im bestehenden CourseCreator möglich; diese Erweiterung ist im Folgenden Beschrieben.

## neue Strukturen – die Erweiterung des CourseCreators

Um diese beiden neuen Strukturen zu ermöglichen, sollte man sich zuerst fragen, auf welcher Grundlage die Strukturen arbeiten sollen. Das einzige, was wir als Grundlage annehmen können, sind die Dokumente, die wir aus der Datenbank kennen[1]. Diese wollen wir (das ist der Sinn des CourseCreators) in eine Struktur einordnen und wir wollen den Kurs „sehen“, das heißt, es muss eine graphische Darstellung existieren. Des weiteren wissen wir, dass wir der Struktur Metainformationen zuordnen wollen, und wir wollen auf der Struktur „arbeiten“. Außerdem lässt sich eine Struktur als Kurs, Kursabschnitt oder Aufgabenblatt formulieren.

Der CourseCreator muss also die Funktionen eines Editors erfüllen. Das heißt es soll

- eine neue Struktur erzeugt werden
- die Struktur wird graphisch dargestellt
- es können Komponenten eingefügt werden
- den Komponenten können (wiederholt) Dokumente zugeordnet werden
- es können Komponenten wieder gelöscht werden
- eine Komponente ist auswählbar, zum Beispiel um sie zu löschen oder ein Dokument zuzuordnen
- der Struktur werden Metainformationen zugeordnet
- die Struktur kann gespeichert und geladen werden

Für die Darstellung des Kurses wird vom CourseCreator ein leeres Scrollfenster zur Verfügung gestellt (bisher wurde das Feld automatisch mit einem -wenn auch leeren- Netzwerkkurs gefüllt.

Möchte eine Struktur weitere beschreibenden Eigenschaften nutzen, so müssen diese extra implementiert werden.

Programmiertechnisch bedeutet die Umsetzung der oben genannten Eigenschaften, dass der CourseCreator nur auf einer abstrakten Klasse (Struct) arbeitet. Natürlich weiß er auch, welcher Kurstyp gerade angezeigt wird und reagiert entsprechend, d.h. durch einfache `if`−Abfragen werden die Nutzerangaben an die richtigen Strukturen weitergeleitet.

1 nun ja, es wäre auch möglich, die Dokumente des Datenbank mit dem CourseCreator zu erzeugen. Man kann diese Dokumente als hierarchische Struktur ansehen; es wäre möglich, die strukturierenden Elemente der Dokumente (defequivalence, suppositions, implication,remarks ..) als „Knoten“ anzusehen, die dann mittels intuitiver Eingabefelder zu füllen wären.

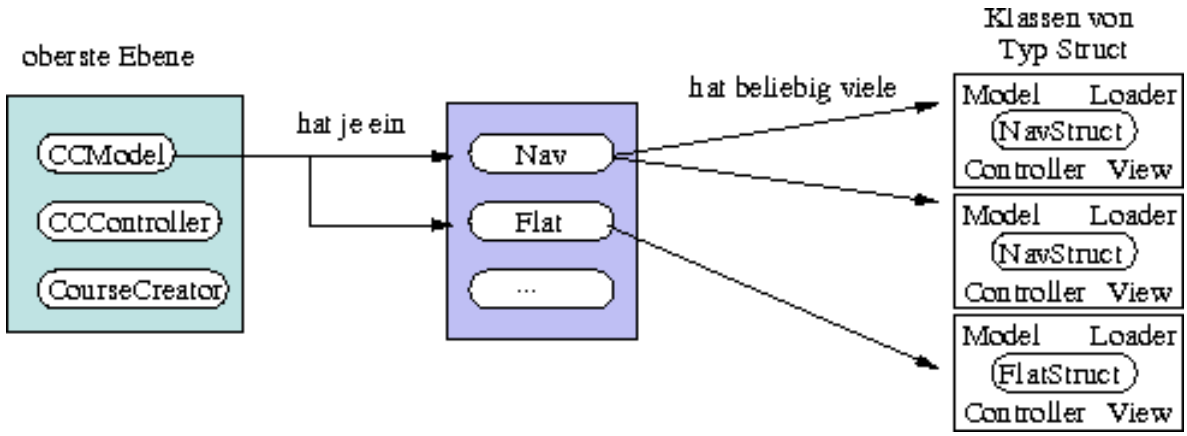



*Abbildung 3: Schematische Klassenstruktur*

Die Klassen CCModel, CourseCreator und CCController sind die „wesentlichen“ Klassen des CourseCreators. Sie arbeiten nach dem Model-View-Controller Prinzip, also der Trennung von Daten (CCModel), Darstellung (CourseCreator -die Klasse müsste konsequenterweise CCView heißen..) und Controller (CCController). Diese drei Klassen werden als Hauptebene bezeichnet.

Die Klasse CCModel verwaltet die Strukturen. Jede Struktur besteht aus mindestens 5 Klassen:

StructA[2]

StructAModel

StructAView

StructAController

StructALoader

Nur die Klasse StructA ist dem CCModel bekannt. Die Klasse StructA kann beliebig viele StructAModel besitzen und dient hauptsächlich als „Tunnel“, um die Kommandos von der Hauptebene an die einzelnen Strukturen weiterzuleiten. Die einzelnen StructAModel sind gereiht, so dass die erste Struktur angezeigt wird, falls die angezeigte Struktur vom Kurstyp StructA sein soll. Die StruktAModel-Klassen sind jeweils von der Klasse Struct abgeleitet. Dies macht es möglich, das die Klasse CCModel quasi „trotzdem“ auf der eigentlichen Struktur arbeiten kann, obwohl er sie nicht genau kennt.

## *Am Beispiel*

Veranschaulichen wir das prinzipiell an zwei Beispielen:

Die Funktion delete soll eine ausgewählte Komponente der Struktur entfernen. Diese Funktionalität muss jede Struktur kennen. Die folgende Funktion stammt aus der Klasse CCModel, die ja die einzelnen Strukturgruppen verwaltet:

```
public void delete() {
  this.getFirstStruct().delete();
 }
```

Diese Funktion könnte zum Beispiel von der Klasse CCController aufgerufen werden, die das Klicken des Delete-Buttons umsetzt. getFirstStruct liefert die erste Struktur, also, in Abhängigkeit vom Kurstyp das erste Model der Struktur. Diese liefert aber eine Instanz vom Typ Struct, so dass jede Struktur darauf arbeiten kann.

Eine andere Beispielfunktion ist die Funktion `toggleEdgeDirection`, die nur vom Netzwerkgraphen benötigt wird. Diese Funktion muss sofort auf die entsprechende Struktur geleitet werden.

2 StructA steht dabei für Nav (Netzwerkgraph), Flat (hierarchische Struktur) oder List (listenartige Strukturen)

```
protected void toggleEdgeDirection(){
  if (this.kursTyp== KURS_NAVStruct) {
    this.getNavModel().toggleEdgeDirection();
  }
 }
```

In der Klasse NavModel muss es nun eine Funktion geben, die den Funktionsaufruf an die richtige Struktur weiterleitet:

```
public void toggleEdgeDirection(){
    this.getMainGraph().toggleEdgeDirection();
}
```

## *Erweiterbarkeit, dieses CourseCreators*

Möchte man in diesen CourseCreator eine neue Struktur einfügen, muss man, wie oben kurz erwähnt, ein Paket aus den fünf Klassen erzeugen.

Die Klasse StructA muss die Kommunikation mit der Klasse CCModel herstellen, also die durch die Klasse Struct vorgegebenen Funktionen implementieren. Das sind Funktionen, die oben bereits erwähnt wurden wie:

```
delete
setSummary
insertCell
setMetaInfos, getMetaInfos
getSelectionCells
isProblemGraph
isSectionGraph
isElementGraph
```

Aber auch Funktionen die die Kommunikation mit der Oberfläche ermöglichen. So müssen z.B. die Buttons der Toolbar ja nach ausgewählten Komponenten gesetzt werden (der Button für das verbinden darf nur klickbar sein, wenn zwei Graphkomponenten ausgewählt sind. Ist ein Listenartige Struktur angezeigt, so wird der verbinden-Button sogar versteckt). Daher gibt es Funktionen wie

```
boolean getButtonEnable
boolean getButtonVisible
setChanged
```

Sowie zum Speichern des Graphen noch:

```
public String toXMLContent();
public String toXMLMeta();
```

Diese Funktionen müssen dann an die eigentliche Struktur (die Klasse StructAModel) weitergeleitet werden. Diese Klasse implementiert dann, was zB beim insertCell genau passieren soll. Die Klasse StructALoader muss nur die Funktion createGraph und einen Konstruktor implementieren. Der Loader des CourseCreators muss leider um die neue Struktur erweitert werden, da erst nach dem Laden klar ist, um welches Kursformat es sich handelt.
Desweiteren benötigt die neue Datei noch eine Darstellungsstruktur, die zu implementieren ist. Diese Funktionalitäten können einfach implementiert werden. Hat die Struktur keine weiteren Funktionalitäten, so reichen diese Funktionen fast schon aus (mit der Implementation eines Konstruktors in der Klasse StructAModel natürlich), um auf dem Graphen schon arbeiten zu können.
Soll der Graph noch „spezielle“ Funktionen besitzen, so müssen die in den CourseCreator eingefügt werden. Dies geschieht entweder durch die Implementierung ganz neuer Funktionen, oder die Erweiterung einer `if`-Funktion nach dem zweiten Beispiel im Kapitel Am Beispiel.