Animiertes UML als Medium für die Didaktik der objektorientierten Programmierung

Friedrich Steimann, Uwe Thaden, Wolf Siberski, Wolfgang Nejdl

Institut für Technische Informatik
Rechnergestützte Wissensverarbeitung
Universität Hannover
Appelstraße 4
D-30176 Hannover

Learning Lab Lower Saxony (L3S)
Expo Plaza 1
D-30539 Hannover
{steimann, thaden, siberski, nejdl}
@learninglab.de

Abstract: Das objektorientierte Paradigma hat die Konzepte des Programmierens näher an die der Realität gerückt. Zugleich ist damit aber der Abstand von Programmen zu der Maschine, die die Programme ausführen soll, größer geworden. Die alten anschaulichen Modelle vom Programmablauf auf einer Registermaschine sind somit zu didaktischen Zwecken kaum noch einsetzbar. Statt dessen schlagen wir eine auf der grafischen Modellierungssprache UML basierende animierte Visualisierung vor, die die Metaphern der objektorientierten Programmierung nutzt und dadurch eine intuitive Vorstellung von den Abläufen eines Programms ermöglichen sollte. Animiertes UML kann im Unterricht in Form von vorproduzierten Lehrfilmen oder interaktiv in Programmierübungen eingesetzt werden.

1 Einleitung

Mit dem immer größer werdenden Anteil von Software an der Wertschöpfung nimmt auch der Bedarf an gut ausgebildeten Programmierern weiter zu. Um wirklich produktiv zu sein, müssen Programmierer in der Lage sein, in den Konzepten ihrer Programmiersprache zu denken. Da Programme neben den zur Problemlösung notwendigen statischen Strukturen vor allem Abläufe spezifizieren, setzt dies voraus, daß intuitive Modelle von den Geschehnissen innerhalb eines Programms existieren und daß diese auch im Unterricht vermittelt werden können.

In diesem Beitrag wird eine erste Übersicht darüber gegeben, worin das Problem der Veranschaulichung objektorientierter Programmabläufe besteht und wie eine nach unserer Auffassung geeignete Lösung vor didaktischen Hintergrund auszusehen hat. Daß und warum sich unsere Aufgabenstellung und Lösung von den in anderen, verwandten Arbeiten vertretenen unterscheidet, wird nach einer kurzen Darstellung des gegenwärtigen Stands unserer Arbeit am Schluß diskutiert.

2 Das Problem

Der prozedurale oder imperative Programmierstil hat durch seine enge Verwandtschaft mit dem verbalen Algorithmus eine Folge von Anweisungen an eine abstrakte Maschine als intuitives Ausführungsmodell. Jeder kennt Prosa der Form

1. tue dies
2. tue das
3. wenn xyz, dann gehe zu 2
4. und so weiter

aus seinem frühen Informatikunterricht. Die Strukturierte Programmierung hat zwar Sätze wie den dritten in der obigen Formulierung, die – mit einer Von-Neumann-Architektur als abstrakte Maschine im Hinterkopf – besonders anschaulich sind, eliminiert, aber die Vorstellung von einem Prozessor, der die Sequenz der Anweisungen abarbeitet, bleibt trotzdem tragfähig.

Die objektorientierte Programmierung setzt demgegenüber einen anderen Schwerpunkt: Obwohl natürlich die (abstrakte) Zielmaschine dieselbe ist, steht hier nicht die Abarbeitung von Anweisungen im Vordergrund, sondern die dynamische Entstehung und das Vergehen von Objekten sowie die Kommunikation zwischen diesen. Eine solche Sichtweise ist in der Regel näher an dem Ausschnitt der Realität, der durch das jeweilige objektorientierte Programm abgedeckt werden soll. Es ist aber, vor allem wegen der für die objektorientierte Programmierung charakteristischen Verteilung der Anweisungen auf Objekte, nicht unmittelbar einsichtig, wie die Abbildung auf die abstrakte Maschine funktioniert oder wie so eine Maschine auch nur auszusehen hat (Abbildung 1). Das klassische Bild vom einfachen Prozessor mit seiner beschränkten Registerzahl und dem Programmzähler taugt jedenfalls nicht dazu. Dies ist in gewisser Weise paradox, denn man hält der objektorientierten Programmierung für gewöhnlich zugute, daß sie aufgrund ihrer Nähe zu unserer Auffassung von der Realität als einer Menge miteinander verknüpfter und interagierender Objekte um einiges intuitiver ist als die prozedurale. Es fehlt jedoch weitgehend eine anschauliche Vorstellung davon, wie ein objektorientiertes Programm auf einem angemessenen Abstraktionsniveau maschinell abläuft.

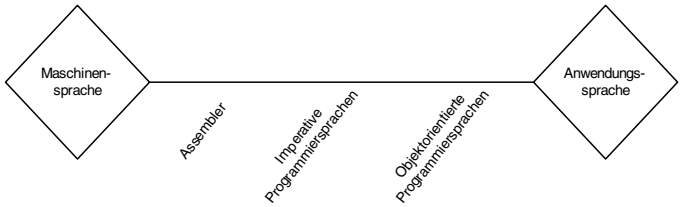

Abbildung 1: Bei der Entwicklung eines Softwaresystems sind die Konzepte der Anwendungsdomäne (Anwendungssprache) in die des (virtuellen oder realen) Laufzeitsystems (Maschinensprache) zu übersetzen. Während die Strukturähnlichkeit von Software und Anwendungswelt [Zü01] durch die Einführung der objektorientierten Programmiersprachen ein vorläufiges Maximum erreicht hat, wird es immer schwerer, die Funktionsweise des Laufzeitsystems dieser Sprachen zu erklären, da naturgemäß mit der Verringerung des Abstandes zur Anwendungssprache der Abstand zur Maschinensprache immer größer wird.

Nun ist es sicher möglich, die Abbildung objektorientierter Programme auf eine abstrakte Maschine bekannten Typs (mit imperativem Charakter) vorzunehmen und sich damit den Programmablauf zu veranschaulichen, doch wird damit der Abstand zwischen Realität und Programm, der durch die objektorientierte Denkweise ja gerade verringert werden sollte, wiederhergestellt. Wenn man es erst ins Prozedurale übersetzen muß, um sich den

Ablauf eines objektorientierten Programms zu erklären, wie soll man dann objektorientiert Denken lernen? Viel günstiger wäre es, wenn der Abbildungs- oder Modellcharakter, den ein objektorientiertes Programm in Bezug auf die Realität von Natur aus hat, erhalten bliebe und dadurch zugleich (und nicht trotzdem) der Programmablauf erklärt würde.

Es erscheint also lohnenswert, zu überprüfen, ob der intuitive Charakter der objektorientierten Programmierung nicht nur für die direkte Nachbildung (unserer Auffassung von) der Realität, sondern auch für die Erklärung des Programmablaufs herangezogen werden kann. Einen ersten Ansatz dazu mag die UNIFIED MODELING LANGUAGE (UML) mit ihren dynamischen Diagrammtypen wie den Sequenz- oder den Kollaborationsdiagrammen bieten. Deren Visualisierungen des Programmablaufs sind aufgrund der Beschränkung des Mediums Papier jedoch lediglich statische Projektionen, die die Anschaulichkeit nicht eben fördern.

Die Visualisierung in anderen Disziplinen mit vergleichbarem Bedarf (nicht nur für die Lehre) an Veranschaulichung abstrakter oder zumindest nicht sichtbarer Prozesse ist demgegenüber weiter fortgeschritten, weil sie auch bewegte Modelle (Animationen) umfaßt, in denen die zeitliche Dimension in ihrer natürlichen Form vorkommt. Die Molekular- und Mikrobiologie z. B. visualisieren die Vorgänge des Stoffwechsels oder der Zellteilung durch kleine Filme, die die Abläufe auf molekularer Ebene plastisch veranschaulichen. Was liegt also näher, als UML zu animieren, um die Abläufe in objektorientierten Programmen begreiflich zu machen?

Animiertes UML verspricht im Unterricht objektorientierten Programmierens prinzipiell auf zwei Arten einsetzbar zu sein:

1. in Form von Lehrfilmen, in denen in separaten, kommentierten Sequenzen die grundsätzlichen Abläufe in objektorientierten Programmen wie Instanziierung, Methodenaufruf etc. veranschaulicht werden, und
2. zur Visualisierung des Ablaufs zumindest theoretisch beliebiger objektorientierter Programm(teil)e.

Die technischen Erfordernisse für die Realisierung des zweiten Punkts sind natürlich wesentlich höher: Während für den ersten lediglich eine genaue Vorstellung davon, was animiertes UML ist, und ein handelsübliches 3D-Animationsprogramm erforderlich sind, müssen für den zweiten die Zusammenhänge und Abläufe aus (beliebigen) objektorientierten Programmen extrahiert und automatisch in eine Animation umgesetzt werden. Aus didaktischer Sicht ist das aber auf jeden Fall erstrebenswert, da so das experimentelle Erlernen der objektorientierten Programmierung unterstützt wird. Überhaupt würde wohl jeder, der schon einmal versucht hat, den Ablauf eines Programms anhand der (aus der prozeduralen Programmierung) hergebrachten Traces zu verfolgen, eine solche Hilfe zu schätzen wissen.

3 Was ist animiertes UML?

Es liegt in der Natur der Animation, daß sie dynamische Aspekte veranschaulicht. Das schließt jedoch das Vorkommen statischer Aspekte nicht aus – nur wird z. B. ein Klassendiagramm wenig Anlaß zur Animation geben, es sei denn, man wollte den Vorgang der Programmierung selbst auch visualisieren. Ein kombiniertes Klassen- und Objektdia-

gramm ist jedoch bereits animierbar, wenn man den Vorgang der Erzeugung der Objekte und die Einrichtung der Verknüpfungen (Links) zwischen diesen zeigt. Interpretiert man die entstandenen Verknüpfungen zwischen Objekten in der Folge als Kanäle des (potentiellen) Nachrichtenaustauschs, dann ist man schon beim Kollaborationsdiagramm, das sich als dynamischer Diagrammtyp zur Animation geradezu aufdrängt.

Tatsächlich ist *die* Metapher des objektorientierten Programmierens, der Nachrichtenaustausch zwischen Objekten, die sich kennen, zugleich eine dankbare Grundlage für die Erstellung von Animationen zur Visualisierung von Abläufen: Objekte schicken einander Nachrichten über eine Art Rohrpost, der (Verweise auf) andere Objekte als Parameter beigefügt sind. Der Empfänger einer solchen Post reagiert darauf mit einem Zustandswechsel seinerseits und/oder dem Versenden weiterer Nachrichten an andere Objekte. Selbst die Instanziierung fällt unter diese Metapher, wenn man sie als durch eine Nachricht an eine Klasse ausgelöst auffaßt – sie führt jedoch zusätzlich zur Erzeugung eines neuen Objekts, für die aber ebenfalls eine geeignete Metapher vorhanden ist; mehr dazu unten.

Auch wenn sich mit diesem einen Animationstyp (Animationstyp als Pendant zum UML-Diagrammtyp) bereits ein Großteil der objektorientierten Programmierung anschaulich machen läßt, ist es sinnvoll, aus didaktischen Gründen differenzierte Animationstypen anzubieten, die jeweils einen bestimmten Aspekt der Abläufe hervorheben. Zunächst betrachtet werden:

1. Struktur- oder besser Instanziierungsanimationen, die die Typhierarchie und die Instanziierung von Klassen zu Objekten und Beziehungen zwischen diesen visualisieren, und
2. Interaktionsanimationen, die den Austausch der Nachrichten, ggf. auch mit Parametern, visualisieren.

Bereits hier sie bemerkt, daß hier, anders als in anderen Arbeiten, nicht die Möglichkeiten der Softwaremodellierung, für die UML ja eigentlich gedacht ist, vorangetrieben werden sollen – vielmehr soll eine intuitive Vorstellung von den konkreten Abläufen in objektorientierten Programmen geliefert werden. Auf Diagrammtypen wie die Use-case- oder Komponentendiagramme, so wichtig sie für die Softwaremodellierung sein mögen, kann daher hier verzichtet werden.

3.1 Instanziierungsanimationen

Bei der Instanziierungsanmination wird die Typhierarchie dargestellt und der Vorgang der Instanziierung, also der Erzeugung von Objekten und den Verbindungen zwischen diesen. Da Links zwischen Objekten nur da entstehen dürfen, wo auf Typebene Assoziationen vorgesehen sind, sind auch die Assoziationen eingezeichnet.

Zur Darstellung des Vorgangs der Instanziierung wird die im Deutschen durch das Wort selbst nahegelegte Metapher des Stanzens von Objekten herangezogen: Wie ein Stempel senkt sich der Boden eines Klassensymbols nieder und prägt ein Objekt mit seinen Attributen in die Ebene, auf der die Objekte angeordnet sind. Der Vorgang ist in Abbildung 2 dargestellt.

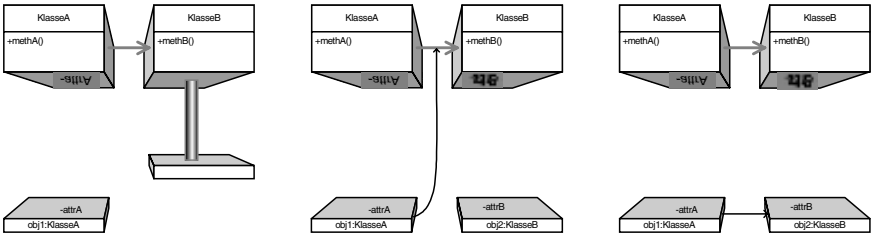

Abbildung 2: Schnappschüsse einer Instanziierungsanimation
Zuerst wird ein neues Objekt gestanzt, dann fällt ein Link herab. Während die Klassen stets an ihren Positionen bleiben, ist die Ebene, in die die Objekte gestanzt werden, verschiebbar.

Die durch die Stanzmetapher implizierte Dreidimensionalität der Darstellung bietet zugleich die Möglichkeit, die Typhierachie adäquat darzustellen. Im zweidimensionalen UML besteht ja das Problem, daß die Relationen erster Ordnung (wie die Assoziationen) und die zweiter Ordnung (wie der Generalisierung) nur unzureichend voneinander unterschieden werden, so daß die Generalisierung von Anfängern häufig für eine spezielle Assoziation (vergleichbar den Aggregationen) gehalten wird. Vorläufer der UML haben daher eine Venn-Diagrammen ähnliche Notation für die Typhierarchie verwendet. In der dreidimensionalen Darstellung wird es nun möglich, Supertypen räumlich über ihren Subtypen anzuordnen, so daß die Relationen der beiden Ordnungen in unterschiedlichen Dimensionen verlaufen (Abbildung 3). Daß sich eine Assoziation auch über mehrere Ebenen erstrecken, also eine Ober- mit einer Unterklasse verbinden kann, läßt sich dadurch richten, daß man per Konvention sicherstellt, daß Assoziationen stets an den Seiten der (kubischen) Typsymbole andocken, während die Generalisierungs- bzw. Implements-Relation stets eine Typenober- mit einer -unterseite verbindet.

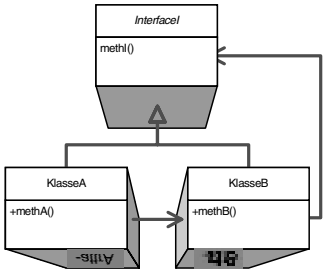

Abbildung 3: Typhierarchie, in der Supertypen räumlich über ihren Subtypen angeordnet sind.

Ein nützlicher Seiteneffekt der Darstellung der Klassenhierarchie über mehrere Ebenen ist die Möglichkeit, auf Verlangen den Effekt der Vererbung darzustellen, also z. B. die Wiederholung der geerbten Assoziationen für Subtypen, ohne dadurch das Diagramm zu überfrachten. Von einer solchen Option abhängig kann auch die Instanziierung von Objekten als ein Vorgang (mit allen Attributen zugleich) oder als Sequenz mehrerer Einzelinstanziierungen (nämlich durch die Klasse und alle ihre Superklassen) dargestellt werden, was z. B. dem rekursiven Aufruf der Konstruktoren aller Superklassen in JAVA entspräche.

3.2 Interaktionsanimationen

Als Vorlage für die Erstellung von Interaktionsanimationen dienen die Interaktionsdiagramme der UML. Da Animationen inhärent dynamisch sind, verlieren die Ausdrucksmittel des zeitlichen Ablaufs, die für die Interaktions*diagramme* aufgrund ihrer statischen Beschaffenheit so wesentlich sind, an Bedeutung; ihnen kommt allenfalls noch die Funktion zu, die Spur des zeitlichen Ablaufs zu festzuhalten. Insbesondere die vertikale zeitliche Dimension und somit der Vorzug des Sequenzdiagramms verlieren dabei an Bedeutung. Wenn man zudem erlaubt, daß die Objekte in Sequenzdiagrammen wie in [GRR99] beschrieben zweidimensional angeordnet werden, dann geht das Sequenzdiagramm in der zeitlichen Projektion sogar in das Kollaborationsdiagramm über (vgl. dazu die Diskussion in Abschnitt 5.1).

Eine Kollaborationsanimation besteht zunächst aus der Anordnung von Objekten (wie sie z. B. aus einer Instanziierungsanimation hervorgegangen ist) und den Verknüpfungen zwischen diesen. Ausgehend von einem Akteur wird entlang einer Verknüpfung eine Nachricht zu einem anderen Objekt geschickt und zieht dabei einen Ariadne-Faden hinter sich her. Das Objekt reagiert auf den Erhalt einer Nachricht, indem es seinerseits Nachrichten entlang seiner Verknüpfungen verschickt oder/und ggf. das Abarbeiten der Nachricht durch die Übergabe eines Rückgabewertes quittiert.

Anders als in Kollaborationsdiagrammen besteht in deren animierter Variante die Möglichkeit, Parameter in Form anderer Objekte, die beim Nachrichtenaustausch übergeben werden, als „wandernde Verknüpfungen“ darzustellen, wie in Abbildung 4 (a) gezeigt. Dieser Parameter kann entweder dazu verwendet werden, eine dauerhafte Verknüpfung zwischen dem Empfänger und ihm herzustellen (b), oder weitergereicht (c) bzw. selbst Empfänger einer Nachricht (nicht gezeigt) werden.

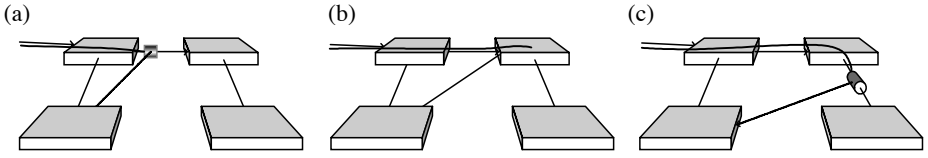

Abbildung 4: verschiedene Schnappschüsse einer Kollaborationsanimation (s. Text)

Eine Kollaborationsanimation wie die in Abbildung 4 gezeigte enthält keine Instanziierungen. Da eine Instanziierung in aller Regel mit einer Initialisierung der neuen Instanz einhergeht und diese Initialisierung parametrisiert ist, ist es sinnvoll, eine Instanziierungsanimation in eine Kollaborationsanimation einzubauen. Dazu ist es erforderlich, daß die Klassensphäre eingeblendet wird und daß die Nachricht, die die Instanziierung auslöst, von einer Instanz (oder einem Akteur) an eine Klasse gesendet wird.

Auch wenn es für die Instanziierung notwendig ist, daß das Objekt, das die Instanzzierung auslöst, die Klasse, von der die neue Instanz sein soll, kennt, ist es nicht sinnvoll, entsprechende Verknüpfungen einzuzeichnen, denn normalerweise sind alle Klassen global verfügbar. Gleichwohl ist es sinnvoll, Instanziierung als durch eine Nachricht an eine Klasse ausgelöst darzustellen, auch wenn z. B. in JAVA die Syntax der Objektkonstruktion (`new <Klassenname>([Parameterliste])`) diese Sichtweise nicht unbedingt nahelegt. Ein solcher kombinierter Instanziierungs-/Initialisierungsvorgang in Form

einer kombinierten Instanziierungs-/Kollaborationsanimation ist in Abbildung 5 dargestellt. Damit subsumiert der Typ der Kollaborationsanimation die Instanziierungsanimation.

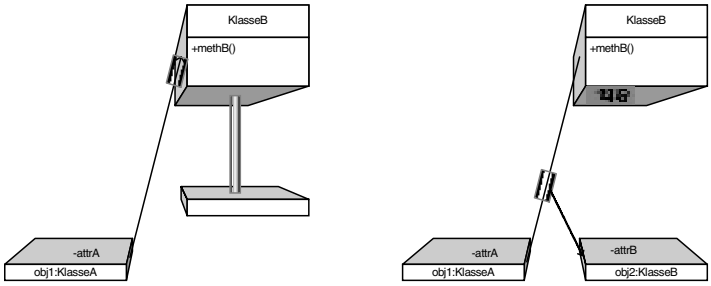

Abbildung 5: Vorgang der Instanziierung als Kollaborationsanimation

4 Stand der Arbeiten

In einem ersten Ansatz verwenden wir das UML-Modellierungswerkzeug TOGETHER der Firma TOGETHERSOFT (http://www.togethersoft.com), um aus einem objektorientierten Programm das statische Strukturdiagramm (Klassendiagramm) sowie interessierende Kollaborationsdiagramme automatisch zu erzeugen oder auch den Benutzer diese Diagramme selbst zeichnen zu lassen. TOGETHER bietet über eine Exportfunktion die Möglichkeit, seine Diagramme im XMI (= XML-Metadata-Interchange)-Format zu speichern. Als XMI-Typ sollte hierbei UNISYS Version 1.1 verwendet werden, da nur bei diesem Positionsinformationen der Diagrammelemente in der erzeugten Datei enthalten sind.

Die XMI-Dateien repräsentieren die Informationen über die einzelnen Modellelemente in hierarchischer Form. Für das Einlesen dieser Informationen haben wir ein JAVA-Programm geschrieben, das aus einer Datei mit Hilfe des XML-Parsers XERCES von APACHE (http://xml.apache.org/xerces-j/index.html) eine Baumstruktur aufbaut. Aus dem Parse tree werden dann die Dateien erzeugt, die den VRML (=Virtual Reality Modeling Language)-Code für die Darstellung der 3D-Diagramme enthalten (Abbildung 6). Dazu gehören die Beschreibungen der verwendeten geometrischen Objekte sowie deren Positions- und Bewegungsinformationen. Darüber hinaus sind Beschreibungen für das sog. Weltsystem wie Hinter- und Untergrund sowie Texturen enthalten.

Zur Komposition der geometrischen VRML-Primitive zu Ablaufanimationen sind für die einzelnen Modellelemente wie Klasse, Objekt etc. korrespondierende JAVA-Klassen definiert worden, die zu den jeweiligen Knoten des Parse tree die entsprechenden 3D-Repräsentationen liefern. Schwieriger ist die Erzeugung der Animationen: Zwar können neben den Beschreibungen für die statische Darstellung mit VRML auch Bewegungsabläufe spezifiziert werden, jedoch müssen hierbei grundsätzlich Rotationen und Skalierungen, wie sie auch bei der Veränderung des Betrachtungswinkels durch den Benutzer vorkommen, von Translationen unterschieden werden. Bei ersteren verändert nämlich das Element seine ursprüngliche Position nicht, während bei Translationen eine Beschreibung des Bewegungspfades nötig ist. Hierfür werden spezielle Skripte verwendet,

die im wesentlichen aus den Informationen für Start- und Endpunkt sowie der Bewegungsgeschwindigkeit bestehen.[1]

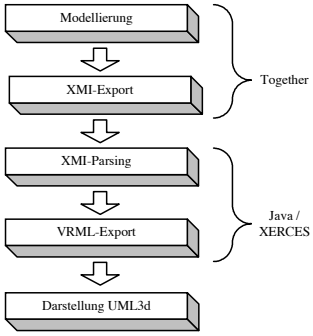

Abbildung 6: Schritte der Erzeugung einer 3D-Animation

Prinzipiell stellen die Animationen auf Basis von UML-Interkationsdiagrammen im Fall von Fallunterscheidungen und Wiederholungen (Schleifen) nicht alle möglichen, sondern lediglich exemplarische Abläufe dar, da die Variablenwerte nicht zur Verfügung stehen (vgl. die Diskussion unten). In einer fortgeschrittenen Version kann u. U. eine Debug-Schnittstelle wie z. B. die von JAVA dazu verwendet werden, aus dem laufenden Programm heraus eine grafische Animation der Abläufe zu steuern, so daß mit echten Variablenwerten animiert werden kann. Für Diagramme vom Typ der Abbildung 4, bei denen aus übergebenen Parametern neue Links eines Objekts werden, ist es ohne eine solche Schnittstelle außerdem notwendig, daß alle Zugriffe auf Instanzvariablen über Accessoren (in TOGETHER automatisch als Getter und Setter von JAVA BEANS erkannt) erfolgen.

Die durch den in Abbildung 6 skizzierten Ablauf erzeugten VRML-Dateien können mit einem entsprechenden Viewer oder Browser plugin angeschaut werden. Das von uns verwendete Plugin der Firma BLAXXUN (http://www.blaxxun.de) bietet unter anderem die Möglichkeit, frei definierte Viewpoints anzugeben. Dabei handelt es sich um vorgegebene Ansichten im Raum, die über ein Menü ausgewählt werden können. Indem man ihm eine verdeutlichende Sicht im Raum vorgibt, können wesentliche Aspekte wie z. B. die räumlichen Hierarchiebeziehungen für den Lernenden hervorgehoben werden. Auch das Kombinieren von einzelnen Viewpoints zu einer sog. Tour kann für den Lernenden unterstützend eingesetzt werden.

5 Diskussion

Um die Animationen einfach zu halten, ist es generell notwendig, die programmiererische Freiheit etwas einzuschränken. Zu den sinnvollen Beschränkungen gehören zunächst:

[1] VRML bietet darüber hinaus verschiedene Sensoren, die eine Animation beeinflussen können, z. B. Mausklicks und Kollisionen von Objekten. Zur Zeit sind jedoch alle Animationen in unserer Arbeit durch die programmierten Skripte im Ablauf fixiert und können nach ihrer Erzeugung nicht von außen gesteuert werden.

- Zugriff auf die Attribute einer Klasse/Instanz ausschließlich über Accessoren, damit die Zuweisung von Parametern an Attribute automatisch entdeckt werden kann;
- Typhierarchie als Abstraktionshierarchie [St00] (jede Klasse ist entweder `final` oder `abstract` deklariert), damit beim „Stanzen“ von Instanzen keine niedrigeren Klassen im Weg stehen können.

Weitere Restriktionen werden sich aus der Praxis ergeben.

Einschränkungen dieser Art, die zunächst technisch bedingt sind, müssen nicht unbedingt als solche empfunden werden, denn zum einen können sie schon aus programmierpraktischer Sicht sinnvoll, zum andern aus generellen didaktischen Überlegungen wünschenswert sein; es soll ja lediglich ein Verständnis für die prinzipiellen Abläufe, nicht für die technischen Details oder gar Idiome der objektorientierten Programme und Programmierung gefördert werden. Auch müssen natürlich die sprachspezifischen Unterschiede soweit wie möglich wegabstrahiert werden, solange man sich nicht auf eine bestimmte Sprache im Unterricht festlegen will.

Ein nicht unbeträchtliches Problem der objektorientierten Programmablaufvisualisierung, das gewissermaßen von UMLs Interaktionsdiagrammen geerbt wird, ist die Darstellung der häufig vorkommenden 1:*n*-Beziehungen, in denen ein Objekt mit mehreren anderen (gleichen Typs) assoziiert ist und mit diesen gleichförmig (in der Regel über einen Iterator) interagiert. Eine Möglichkeit ist, den Container, der die Menge der Objekte enthält und der als eigenständiges Objekt in der Modellierung für gewöhnlich unterschlagen wird, explizit zu machen und die Menge der Objekte dahinter zu verbergen. Das assoziierende Objekt kommuniziert dann nur noch mit dem Container, der die Aufgaben autonom weiterdelegiert, und nicht mehr direkt mit den Objekten. Dies kann durchaus auch guter Stil sein [Be97].

Die vollständige Animation realistischer (und damit größerer) Programme ist schon aus zeitlichen Gründen kaum sinnvoll. Aber auch kleine Programme können zu unbefriedigenden Animationen führen. So zeigt eine einem Trace entsprechende automatisch erzeugte Animation nie alle möglichen Abläufe (Interaktionen), sondern nur die, die den Eingaben entsprechen, diese dafür aber u. U. (bei Iterationen) quälend oft. Interessanter wäre hier eine Animation, die auf einer Analyse der Fallunterscheidungen basiert und die alternative Abläufe der Reihe nach anbietet oder Verzweigungen dem Betrachter zur Auswahl überläßt. Lösungen hierfür können dem Gebiet des White-box-Testens und der Programmverifikation entnommen werden.

5.1 3D-Modellierung und Animation von Modellen

Eine unserer Arbeit im Ansatz recht ähnliche ist die von Gogolla et al. [GRR99, RG00]. Die Autoren zielen jedoch nicht auf didaktische Aspekte, sondern vielmehr auf eine Erweiterung der Möglichkeiten des Modellierers ab und setzen ihre Schwerpunkte entsprechend. So wird die dritte Dimension der virtuellen Realität in erster Linie dazu genutzt, das Interessierende eines Modells vom Rest abzuheben. Gerade inspizierte Teile beispielsweise eines Klassendiagramms werden dazu in den Vordergrund geholt, während die anderen in die Ferne rücken. Dem steht in unserem Ansatz die Aufteilung des Raums in horizontale Ebenen gegenüber, wobei sich auf der untersten (der „Erdoberflä-

che“) die Instanzen bewegen und in der Sphäre darüber (der „übernatürlichen“), in vertikaler Richtung gemäß ihres Abstraktionsgrads angeordnet, die Typen.

Was die Animation von Interaktionen angeht, gibt es deutliche Parallelen zwischen der Arbeit von Gogolla et al. und der unseren. Allerdings ist das Verschieben von Kugeln oder Zylindern von Objekt zu Objekt als Symbol des Nachrichtenaustauschs („Stimulus“ bei Gogolla et al.) in beiden Arbeiten kein übermäßig origineller Beitrag – er drängt sich ja geradezu auf. Neu ist bei uns die Verwendung eines Ariadne-Fadens als Symbolisierung eines Threads sowie die Darstellung der Parameterübergabe beim Methodenaufruf als mit der Nachricht mitgeführte Links auf die betreffenden Objekte. Als Nachtrag zur dreidimensionalen Animation des Sequenzdiagramms in [GRR99], das bei uns ja deswegen keine Verwendung findet, weil die Zeit in Animationen in ihrer natürlichen Dimension aufgelöst wird, ist anzumerken, daß dieses in ein animiertes Kollaborationsdiagramm übergeht, wenn der Benutzer zu einer Draufsicht wechselt (in der die Lebenslinien nicht mehr zu sehen sind).

Wenn man UML zur Erklärung von objektorientierten Programmabläufen verwenden will, muß man wohl auch ausführbares UML betrachten. Das eignet sich jedoch aus zwei Gründen nicht für unserer Zwecke: Erstens muß man die Visualisierungen, die ja gerade dem Begreiflichmachen noch nicht oder nur schlecht verstandener Abläufe dienen sollen, bei der Modellierung selbst erstellen, und zweitens ist die Semantik der UML, zumindest was die Spezifikation von Abläufen angeht, ungenügend. So haben Ansätze wie z. B. die UML Virtual Machine [Ri01], die Modelle in UML ausführbar machen sollen, eben das Problem, daß UML mit seinen dynamischen Diagrammtypen Programmabläufe zwar gut illustrieren, nicht jedoch in ihrer möglichen Vielfalt spezifizieren kann. Demgegenüber steht die in einer deklarativen Sprache formulierte Aktionssemantik [Me98], die sich jedoch für didaktische Zwecke kaum eignet.

5.2 Visualisierung des Programmablaufs

JINSIGHT (http://www.research.ibm.com/jinsight/) benutzt eine modifizierte Java Virtual Machine, um während der Ausführung eines Programms Trace-Information aufzuzeichnen. Diese Information kann anschließend mit einem Viewer ausgewertet werden. Die angebotenen Diagrammtypen, die allesamt nicht animiert sind, visualisieren kumulative Daten des Programmablaufs wie die Häufigkeit bestimmter Methodenaufrufe, aber auch Informationen zum Grad und Mustern der Vernetzung von Daten. JINSIGHT stellt also typische Profiler-Information zur Verfügung; den objektorientierten Programmablauf erklärt jedoch keiner der angebotenen Diagrammtypen.

DePauw et al. definieren einen vierdimensionalen kanonischen Ereignisraum mit den Achsen Zeit, Methoden, Klassen und Instanzen. Anhand von punktförmigen Einträgen in dieses Koordinatensystem kann der Programmablauf dynamisch (also animiert) visualisiert werden. Da Darstellungen dieser Art jedoch schwierig zu interpretieren sind, werden verschiedene Projektionen vorgeschlagen. Eine, die die Zeit als Dimension erhält, die also ebenfalls animiert ist, zeigt die Kopplung von Klassen anhand der Häufigkeit davon ausgehender gegenseitiger Methodenaufrufe. Andere Sichten zeigen (teilweise dynamische) Histogramme, die kumulative Informationen enthalten, aus denen jedoch der Ablauf wie bei JINSIGHT nicht ersichtlich ist. [DKV94]

Die Visualisierung auf der Basis von erweiterten Konturdiagrammen, wie sie in [JB96] vorgeschlagen wird, erlaubt zwar theoretisch die animierte Darstellung des Ablaufs eines objektorientierten Programms, die Darstellung ist jedoch zu nah an der (Organisation der) Hardware (Stapel und Heap), um für die Veranschaulichung im eingangs skizzierten didaktischen Sinn tauglich zu sein. Für einen Lehrfilm, der sich auch mit der Abbildung auf die Maschine befaßt, ist sie jedoch durchaus von Interesse.

Ein recht anschaulicher visueller Trace rekursiver Prozedur- oder Funktionsaufrufe wurde von Koike und Aida vorgestellt [KA95]. Die Sequenz der Kontrollstrukturen werden durch eine Aneinanderreihung dreidimensionaler Formen dargestellt, die spiralförmig angeordnet werden, wobei die Rekursion in die räumlich Tiefe führt. Die Aufrufe sind jedoch nicht an Objekte gebunden, so daß zumindest eine objektorientierte Adaptation dieses Ansatzes notwendig würde.

Jerding et al. [JSB97] haben die Erfahrung gemacht, daß die Animation des Programmablaufs nicht so aufschlußreich über das Programmverhalten ist wie die zusammenfassende (kumulative) Darstellung des gesamten Ablaufs kombiniert mit der Möglichkeit, interessierende Details genauer darzustellen. Allerdings ist ihre Arbeit im Bereich der Visualisierung realer objektorientierter Systeme (mit mehr als 100.000 Methodenaufrufen) angesiedelt und nicht unter didaktischen Aspekten zu verstehen.

6 Schluß

Animiertes UML wie es hier vorgestellt wurde soll nicht UML oder gar die Möglichkeiten des Modellierens generell erweitern. Es soll vielmehr die anschaulichen Aspekte UMLs dazu verwenden, dem Lernenden ein konkretes Bild von den Abläufen in objektorientierten Programmen zu vermitteln und dadurch die klassische Vorstellung von der Instruktionsmaschine ersetzen. Zwei Einsatzweisen sind geplant: Durch die Produktion von Lehrfilmen, die, didaktisch sinnvoll aufgebaut, die prinzipiellen Abläufe intuitiv erklären ganz so, wie es in anderen Disziplinen bereits seit langem üblich ist, und durch die anschauliche Visualisierung des Ablaufs von Programmen, die der Lernende selbst erstellt hat. Wir sind der Überzeugung, daß animiertes UML einen guten Beitrag im Unterricht des objektorientierten Programmierens leisten kann.

Literatur

[Be97] K Beck *Smalltalk best practice patterns* (Prentice Hall, Upper Saddle River 1997).

[DKV94] W DePauw, D Kimelman, J Vlissides „Modeling object-oriented program execution“ *Proceedings of the 8th European Conference, ECOOP'94* (1994) 163–182.

[GRR99] M Gogolla, O Radfelder, M Richters „Towards three-dimensional representation and animation of UML diagrams“ in: R France B Rumpe (Hrsg) *Proc. 2nd Int. Conf. Unified Modeling Language (UML'99)* Springer LNCS 1723 (Berlin 1999) 489–502.

[JB96] B Jayaraman, CM Baltus „Visualizing Program Execution“ Proceedings of the 1996 IEEE Symposium on Visual Languages (VL '96) IEEE (1996).

[JSB97] DF Jerding, JT Stasko, T Ball „Visualizing interactions in program executions“ 1997 *Int'l Conference on Software Engineering* (ICSE'97) Proc. ACM Press (1997) 360–370.

[KA95] H Koike, M Aida „A bottom-up approach for visualizing program behvior“ in: *11th IEEE International Symposium on Visual Languages* (1995) 91–98.

[Me 98] SJ Mellor, SR Tockey, R Arthaud, P Leblanc „An action language for UML: proposal for a precise execution semantics“ in: *UML'98: Beyond the Notation* Springer LNCS 1618 (1999) 307–318.

[RG00] O Radfelder, M Gogolla „On better understanding UML diagrams through interactive three-dimensional visualization and animation“ in V Di Gesu, S Levialdi, L Tarantino (Hrsg) *Proc. Advanced Visual Interfaces (AVI'2000)* ACM Press (2000) 292–295.

[Ri01] D Riehle, S Fraleigh, D Bucka-Lassen, N Omorogbe. “The Architecture of a UML Virtual Machine.” In: *Proceedings of the 2001 Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'01)* ACM Press (2001).

[St00] F Steimann „Abstract class hierarchies, factories, and stable designs“ *Communications of the ACM* 43:4 (2000) 109–111.

[Zü01] H Züllighoven „Softwareentwicklung“ In: G Schwabe, N Streitz, R Unland (Hrsg) *CSCW-Kompendium* Springer (Heidelberg 2001).