

Fachdidaktische und methodische Aspekte einer
OO-Implementierung von ADTs
im Informatikunterricht der Sek II
mit Softwaretool „fujaba“

Schriftliche Hausarbeit

vorgelegt im Rahmen der
Ersten Staatsprüfung für das Lehramt
für die Sekundarstufe I/II in Informatik
von

Andreas Fischer

15. Oktober 2004
Paderborn

Gutachter: Prof. Dr. Johannes Magenheim

Inhaltsverzeichnis

1 Einleitung.....	3
2 ADTs und OO.....	5
2.1 Der Nutzen von ADTs.....	7
2.2 ADTs objektorientiert.....	8
2.3 Die klassischen ADTs.....	9
2.3.1 Der ADT Stapel.....	10
2.3.2 Der ADT Schlange.....	12
2.3.3 Der ADT Liste.....	13
2.3.4 Der ADT Binärbaum.....	15
3 UML und CASE-Tools.....	17
3.1 Die Unified Modeling Language (UML).....	17
3.2 CASE-Tools.....	18
3.3 Das CASE-Tool Fujaba.....	19
3.3.1 Überblick.....	20
3.3.2 Klassendiagramme.....	21
3.3.3 Aktivitätsdiagramme.....	22
3.3.4 Das Dynamic Object Browsing System 'Dobs'.....	25
4 Implementierung der ADTs in Fujaba.....	27
4.1 Der ADT Stack.....	27
4.1.1 Klassendiagramm.....	27
4.1.2 Aktivitätsdiagramme.....	28
4.2 Der ADT Schlange.....	29
4.2.1 Klassendiagramm.....	29
4.2.2 Aktivitätsdiagramme.....	30
4.3 Der ADT Liste.....	31
4.3.1 Klassendiagramm.....	31

4.3.2	Aktivitätsdiagramme.....	32
4.4	Der ADT Binärbaum.....	34
4.4.1	Klassendiagramm.....	34
4.4.2	Aktivitätsdiagramme.....	35
5	ATDs im Informatikunterricht mit Fujaba.....	38
6	Die Umsetzung im Unterricht.....	42
6.1	Die Unterrichtsreihe im Überblick.....	42
6.2	Die verwendeten Methoden im Überblick.....	43
6.2.1	Das Objektspiel.....	43
6.2.2	Der Zetteltest.....	44
6.2.3	Story-Driven-Modelling.....	45
6.3	Die einzelnen Phasen der Unterrichtsreihe.....	46
6.3.1	Phase 1: Die einfache Nachfolgeverkettung.....	46
6.3.2	Phase 2: Die Verkettung durch Knotenobjekte.....	54
6.3.3	Phase 3: Der ADT Schlange zur Verwaltung allgemeiner Objekte.....	59
6.3.4	Phase 4: Der ADT Liste zur Verwaltung allgemeiner Objekte.....	62
6.3.5	Phase 5: Der ADT Stack.....	70
6.3.6	Phase 6: Der ADT Binärbaum.....	72
7	Schlussbemerkungen.....	79
8	Literaturverzeichnis.....	81
9	Versicherung.....	83
10	Anhang.....	84
10.1	Glossar.....	84
10.2	CD.....	85

1 Einleitung

Im Rahmen des life³-Projektes (life-Webseite) wurde Fujaba im Anfangsunterricht der Jahrgangsstufe 11 zur Einführung in die Objektorientierung eingesetzt und evaluiert. In Seminaren an der Universität und im Rahmen der schulpraktischen Studien am Pelizaeus-Gymnasium, an dem Fujaba weiterhin im Unterricht verwendet wird, habe ich selbst Erfahrungen mit Fujaba sammeln können und an der Erarbeitung und Durchführung einer Unterrichtsreihe mit Fujaba mitgewirkt.

Aufgrund der Erfahrungen im Unterricht empfinde ich das objektorientierte Modellieren mit Fujaba als sehr ertragreich. Bei vielen Schülerinnen und Schülern konnte ich eine hohe Motivation feststellen, obwohl oder vielleicht gerade weil die Inhalte des Unterrichts recht anspruchsvoll ausgewählt waren.

Einen besonderen Vorteil bei der Arbeit mit Fujaba sehe ich in der Unterstreichung der Modellierung, während die Implementierung eher in den Hintergrund rückt und später hauptsächlich zur Überprüfung der gewonnenen Analyse- und Modellierungsergebnisse genutzt wird.

Die Schülerinnen und Schüler sollen mit dem vom life³-Projekt entworfenen Unterrichtskonzept ein objektorientiertes Grundverständnis erwerben. Ausgehend von diesem im Jahrgang 11 erworbenen Wissen und den gewonnenen Fähigkeiten und Fertigkeiten stellt sich nun die Frage, ob das Konzept in der Jahrgangsstufe 12 sinnvoll fortgeführt werden kann.

Eine spiralförmige Vertiefung ist in dem gewählten objektorientierten Paradigma möglich (Richtlinien, S. 58), meiner Meinung nach an dieser Stelle sinnvoll und förderlich, um eine „Konzept- und Spracherweiterung durch die (Fort)Entwicklung von Modellen und die Schaffung eigener neuer Sprachmittel jenseits programmiersprachlicher Vorgaben“ (Richtlinien, S. 58) auf der Grundlage des Anfangsunterrichts zu erreichen.

Im objektorientierten Paradigma nehmen Objekte, Beziehungen zwischen ihnen und die auf ihnen ausführbaren Operationen eine zentrale Stellung ein. ADTs können objektorientiert auf dieser Grundlage realisiert werden: „Die Objekte einer Klasse bilden einen abstrakten Datentyp“ (Baumann 1996, S. 278).

In der vorliegenden Arbeit soll dargelegt werden, ob und wie abstrakte Datentypen im Informatikunterricht mit Hilfe von Fujaba eingeführt, modelliert und

anschließend implementiert werden können. Dabei soll die Implementierung vollständig auf der Basis der in Fujaba verwendeten Aktivitätsdiagramme erfolgen, die zur grafischen Beschreibung der Methoden eingesetzt werden. Interessant ist dabei, ob eine umfassende Bearbeitung von ADTs im Unterricht mit Fujaba möglich bzw. sinnvoll ist.

Ausgehend von einer allgemeinen Betrachtung der Abstraktion und dem daraus abgeleiteten Konzept der abstrakter Datentypen werde ich die Vorzüge deutlich machen und die Verbindung von ADTs und Objektorientierung aufzeigen.

Anschließend wird eine Betrachtung der UML erfolgen, die sich in der Softwaretechnik zur Standard-Notation entwickelt hat. Die Darstellung durch Diagramme in UML ermöglicht es dem Anwender, die Modellierung unabhängig von einer bestimmten Programmiersprache strukturiert durchzuführen. CASE-Tools unterstützen den Entwickler bei der Modellierung mit UML. Die sich daraus ergebenden Vorteile können auch im Informatikunterricht genutzt werden, was ich in Kapitel 3 darlegen werde.

Daran anschließend stelle ich exemplarisch dar, wie eine Realisierung der hier betrachteten ADTs in Fujaba möglich ist.

In Kapitel 5 werde ich darlegen, ob das Unterrichtskonzept lehrplankonform ist und fachdidaktischen Aspekten entspricht.

Abschließend stelle ich skizzenhaft einen möglichen Verlauf für eine Unterrichtsreihe zur Behandlung von ADTs mit Hilfe von Fujaba zur Verfügung. Anregungen und Orientierungspunkte dazu erhielt ich einerseits durch die Projekt-Vorlage „Wartezimmer“, die auf dem NRW-Bildungsserver learnline (learnline-Webseite) Lehrern zur Verfügung gestellt wird, andererseits durch ein von Carsten Schulte (Schulte 2004) erarbeitetes Unterrichtskonzept.

2 ADTs und OO

Mit dem Begriff Abstraktion bezeichnet man einen Prozess, bei dem viele Details vernachlässigt werden und nur die zu einem bestimmten Zeitpunkt und in einem bestimmten Kontext wichtigen Informationen betont werden. Im täglichen Leben verwenden wir ständig solche Abstraktionen. Als Beispiel nehmen wir die Aussage „Peters Fahrrad ist grün“. Wenn der Zuhörer die Person Peter kennt und weiß, was ein Fahrrad ist, wäre es unnötig, die Person konkret zu beschreiben („der 1,86m große Junge mit den roten Haaren und den blauen Augen, der neben euch wohnt ... bei der Firma XYZ arbeitet und noch nicht verheiratet ist“). Ebenso wird der Begriff Fahrrad nicht genauer erklärt, da der Zuhörer weiß, um was es sich dabei handelt. Eine genaue Beschreibung von Peters Fahrrad ist auch nicht weiter notwendig, da z.B. Größe, Alter und weitere Ausstattung hier nicht relevant sind. Der Zuhörer benötigt nur das Wissen, um was es sich bei einem Fahrrad ungefähr handelt, nähere Details würden das Gespräch nur in die Länge ziehen und so von der eigentlichen Aussage, dass das Rad grün ist, ablenken. Einem Interessierten könnten weitere Details zur Beschaffenheit des Fahrrades später noch mitgeteilt werden, sie sind aber für das Verständnis der Aussage nicht notwendig.

So werden durch die Abstraktion einerseits nur die wichtigen Eigenschaften betrachtet, einzelne Details werden vernachlässigt. Andererseits werden die Objekte mit einem Namen versehen, so dass man sich einfach auf sie beziehen kann, ohne immer wieder die beschriebenen Eigenschaften zu nennen.

Softwareentwicklung wird immer komplexer, die resultierenden Programme stets umfangreicher. So mussten im Zuge der Zeit Mechanismen entwickelt werden, die es ermöglichen, den Überblick behalten zu können und die zu einem Zeitpunkt benötigten Informationen so gering wie möglich zu halten. Irrelevante Details sollten so weit wie möglich ausgeblendet werden. Dieser Schritt soll mit dem Prozess der Abstraktion erreicht werden.

Methoden in höheren Programmiersprachen sind Abstraktionen von Operationen. Methoden werden durch einen Namen, einige Parameter und durch ihre Wirkung beschrieben, manchmal auch zusätzlich durch ein Ergebnis. Werden diese Methoden in einem Programm benutzt, ist eine genaue Kenntnis über den Methodenrumpf irrelevant. Diese algorithmische Abstraktion besitzt mehrere

Vorteile: Es ist für den Programmierer nicht an jeder Stelle des Programmes wichtig, wie der Algorithmus funktioniert, er muss nur wissen, was er tut und wie er angewendet werden kann. Probleme können auf diese Weise so weit verfeinert werden, bis sie schließlich direkt implementiert werden können. Ein weiterer Vorteil ist die Unabhängigkeit: Der Methodenrumpf kann später bei Bedarf geändert werden, eine Änderung an weiteren Stellen im Programm ist so nicht nötig.

Die Abstraktion ist allerdings nicht nur für Operationen, sondern auch für Daten wichtig. Die Verwendung der Daten wird dann durch die spezifizierten Eigenschaften und die möglichen Operationen festgelegt. Einzelheiten zur Darstellung und Verwendung der Daten werden abstrahiert. In höheren Programmiersprachen werden einfache Datentypen mit grundlegenden zulässigen Operationen zur Verfügung gestellt (z.B. Ganzzahlen mit den Grundrechenarten). Daneben verfügen diese Programmiersprachen aber auch häufig über komplexere Datenstrukturen, bei denen z.B. mehrere der grundlegenden Datentypen in sinnvoller Weise zusammengefasst werden. Bei diesen Datentypen und Datenstrukturen ist für den Programmierer nicht wichtig, wie die Daten intern verarbeitet werden, er muss sie nur benutzen.

Ebenso können neue Datentypen bzw. Datenstrukturen von Programmierern erstellt werden. Um sie verwenden zu können, sind Spezifikationen der Schnittstelle nötig. Von der internen Struktur kann aber abgesehen, also abstrahiert werden.

Solch eine bereits vorgegebene oder auch selbst entwickelte Datenstruktur wird dann auch Abstrakter Datentyp (ADT) genannt.

Jenkins (Jenkins 1998, S. 3) definiert einen ADT sehr allgemein:

- An ADT is an externally defined data type that holds some kind of data.[...]
- An ADT has built-in operations that can be performed on it or by it.
- Users of an ADT do not need to have any detailed information about the internal representation of the operations.

Ein ADT ist also ein definierter Datentyp, der Daten enthält bzw. verwaltet. Zu dieser Verwaltung werden Operationen bereitgestellt. Dabei benötigt der Benutzer, also der Programmierer bzw. Entwickler, keine näheren Informationen über den internen Aufbau der Operationen.

2.1 Der Nutzen von ADTs

Die Eigenschaften eines ADT:

- Universalität (implementation independence)
Der einmal entworfene und implementierte ADT kann ohne Probleme [...] in jedes beliebige andere Programm einbezogen und dort benutzt werden. Unabdingbare Voraussetzung dafür ist selbstverständlich eine
- präzise Beschreibung (precise specifications)
Die Schnittstelle zwischen Implementation und Anwendung muss eindeutig und vollständig sein. Eine entsprechende Spezifikation (=Beschreibung von Softwaremodulen hinsichtlich ihrer Funktion und Schnittstellen) unterstützt auch arbeitsteiliges Vorgehen, was vor allem bei umfangreichen Software-Projekten angebracht ist. So können etwa bestimmte Teams einen oder mehrere ADT bereitstellen, während gleichzeitig andere das eigentliche Programm, das sie anwendet, verfassen. Überhaupt wird jede Art von Kommunikation in und zwischen Arbeitsgruppen durch solche Standardisierung erleichtert.
- Einfachheit (simplicity)
Diese Eigenschaft ermöglicht es dem Anwender, sich auf das Konzeptionelle seiner Arbeit zu konzentrieren. [...] Die klare Modularität (siehe unten) gestattet es, zuerst überschaubare Probleme zu lösen (verkörpert durch die einzelnen Operationen des ADT), sie dann (jedenfalls so lange, bis Fehler sichtbar werden) gewissermaßen vergessen zu dürfen und sich auf die möglicherweise sehr komplizierteren Algorithmen des eigentlichen Programms konzentrieren zu können. In diesem Sinne konstitutiv für den ADT ist das
- Verbergen von Informationen (information hiding)
Ein ADT soll als „black-box“ aufgefasst werden. Der Anwender kennt den Wertebereich und die Operationen, die darauf definiert sind; wie die Daten intern abgebildet und verarbeitet werden, also Repräsentation und Algorithmen, brauchen ihn nicht zu interessieren und sollen es auch nicht. [...] Das stellt auch einen gewissen Schutz dar:
- Geschütztheit (integrity)
Zugriffe – vor allen Dingen verändernde Zugriffe – auf Daten eines ADT geschehen grundsätzlich sehr kontrolliert, indem sie nur über die entsprechenden Operationen möglich sind. [...]
- Modularität (modularity)
Der ADT verkörpert sehr klar das modulare Prinzip. Das erlaubt übersichtliches und damit sicheres Programmieren und leichten Austausch (wie z.B. bei modernen elektronischen Geräten).[...]
(Rolleke und Sennholz 1994, S. 147)

Einer der wichtigsten Vorteile eines ADT ist die Wiederverwendbarkeit durch Modularisierung. Wie bereits beschrieben, werden von höheren Programmiersprachen wie C++ oder Java Datentypen und Datenstrukturen angeboten. Hierfür werden ebenfalls Standardoperationen zur Verfügung gestellt. Für Zeichenketten gibt es so z.B. in Java eine eigene Klasse (String). Unter anderem werden die Grundoperationen zum Vergleichen, Verbinden, Kopieren und Verändern direkt durch diese Klasse angeboten, um mit ihnen auf Objekten dieser

Klasse operieren zu können. Würde diese Klasse (oder eine vergleichbare für Zeichenketten) nicht zur Verfügung gestellt, müsste der Programmierer diese Operationen jedes Mal von Grund auf neu implementieren. Diese Arbeit bleibt ihm nun aber erspart. Er muss nicht immer das „Rad neu erfinden“, sondern kann die bereitgestellten Datenstrukturen verwenden. Die Wiederverwendung einmal erstellter Datenstrukturen in verschiedenen Kontexten und Projekten ist so möglich. Um die Eigenschaften und die daraus resultierenden Vorteile eines ADT effektiv nutzen zu können, müssen die zur Verfügung gestellten Operationen der Datenstruktur angemessen sein. So müssen alle wichtigen Manipulationen möglich gemacht werden. Gleichzeitig ist allerdings darauf zu achten, dass nur wirklich typische Anwendungen berücksichtigt werden. Es ist stets auf eine Ausgewogenheit zwischen komfortabler Handhabung und Minimalität zu achten. Hierbei müssen oft Kompromisse gemacht werden, da z.B. einzelne Anwendungsfälle Operationen erforderlich machen, die dann aber dem Prinzip der Universalität entgegenstehen.

2.2 ADTs objektorientiert

Wie bei der Entwicklung mächtiger und trotzdem überschaubarer Datentypen, gewann die Abstraktion in der allgemeinen Softwareentwicklung immer mehr an Bedeutung.

„Die Geschichte der Softwareentwicklung ist eine kontinuierliche Steigerung der Abstraktion – vom Bitmuster über Makrobefehle, Prozeduren, abstrakte Datentypen zu Objekten, Rahmenwerken, Entwurfsmustern, Komponenten, generativen Verfahren bis hin zur Model Driven Architecture (MDA). Die Objektorientierung bildet hier seit Anfang der 1990er Jahre eine Basistechnologie. Die im Vergleich zu prozeduralen Ansätzen wesentlich höheren Abstraktionsmöglichkeiten der Objektorientierung gründen sich dabei nicht nur einfach auf eine Verbesserung und Weiterentwicklung der klassischen Methoden, sie begründet auch eine neue Denkweise.“ (Oesterreich 2004, S. 16)

Quibeldey-Cirkel spricht daher auch von einem Paradigmenwechsel in der Informatik vom Struktur- zum Objekt-Paradigma (Quibeldey-Cirkel 1994, S. 15).

Sowohl Objektorientierung als auch das ADT-Konzept nutzen die Abstraktion als eine grundlegende Methode.

Der Begriff „Objektorientierung“ ist in der Literatur nicht eindeutig definiert. Meyer (Meyer 1990, S. 65ff) nennt sieben Ebenen, die die Objektorientierung nach

seiner Meinung besitzen muss:

1. Objektbasierte modulare Struktur
2. Datenabstraktion
3. Automatische Speicherplatzverwaltung
4. Klassen
5. Vererbung
6. Polymorphismus und dynamisches Binden
7. Mehrfaches und wiederholtes Erben

Mit folgender Definition stellt er den Zusammenhang von ADTs zur Objektorientierung her: „Objektorientierter Entwurf ist die Entwicklung von Softwaresystemen als strukturierte Sammlungen von Implementierungen abstrakter Datentypen.“ (aaO., S. 64)

Demnach sind Objektorientierung und abstrakte Datentypen untrennbar miteinander verbunden, die Objektorientierung baut direkt auf den ADTs auf. Unter den strukturierten Sammlungen dieser Implementierungen ist eine Sammlung von Klassen zu verstehen. Ein ADT lässt sich objektorientiert als eine oder mehrere Klassen implementieren (vgl. Jenkins 1998, S. 3).

Mit Hilfe von ADTs kann so ein grundlegendes Verständnis von Objektorientierung geschaffen werden, für ein umfassendes Verständnis sind sie vermutlich zwingend notwendig.

Nach der Hinwendung der Fachinformatik zur Objektorientierung und der Bedeutung für die aktuelle Softwareentwicklung sollte sich natürlich der Informatikunterricht in der Schule nicht davor verschließen.

2.3 Die klassischen ADTs

In diesem Abschnitt wird ein allgemeiner Überblick über die vier hier betrachteten ADTs Stapel, Schlange, Liste und Binärbaum gegeben.

Aufgrund der einfachen Strukturen von Stapel und Schlange kann das Prinzip von ADTs mit minimalem Einsatz thematisiert werden. Diese Datenstrukturen können sich die Schülerinnen und Schüler außerdem einfach vorstellen, da sie aus dem

Alltag bekannt sind. Beim Stapel wird das LIFO-Prinzip (siehe Abschnitt 2.3.1), bei der Schlange aber das FIFO-Prinzip (siehe Abschnitt 2.3.2) verwendet. In der Fachinformatik haben diese beiden grundverschiedenen Prinzipien ihre Berechtigung. Sind erste Erfahrungen mit dem ADT Schlange gemacht, kann das Konzept spiralförmig auf einem höheren Niveau anhand der Liste betrachtet werden. Des Weiteren kann der Unterschied zwischen linearen und hierarchischen Datenstrukturen erfahren werden. Der Grundgedanke eines binären Baumes und der benötigten Operationen ist zwar sehr einfach, die Implementierungen der Methoden sind aber wesentlich komplexer als die für lineare Strukturen.

ADTs sind gerade aufgrund der Abstraktion unabhängig von ihrer Implementierung. In dieser Arbeit sollen aber nur dynamische Varianten betrachtet werden, die intern über Verweise realisiert werden. Umsetzungen mit Hilfe von Arrays, wie sie häufig zur Vereinfachung und so zur Einführung im Informatikunterricht eingesetzt werden, sind in der Regel nicht dynamisch und prinzipiell anders konstruiert. Da die objektorientierte Sichtweise im Vordergrund steht, wird der Fokus der Beschreibungen in diesem Abschnitt auch auf den verketteten Strukturen liegen.

Für die einzelnen ADTs gibt es viele verschiedene Varianten, die sich einander mehr oder weniger ähneln. Es soll hier nur ein Überblick über die allgemeinen Ideen und Eigenschaften gegeben werden, eine ausführliche Betrachtung und ein Vergleich verschiedener Ausprägungen ist nicht Gegenstand dieser Arbeit und würde den Rahmen sprengen.

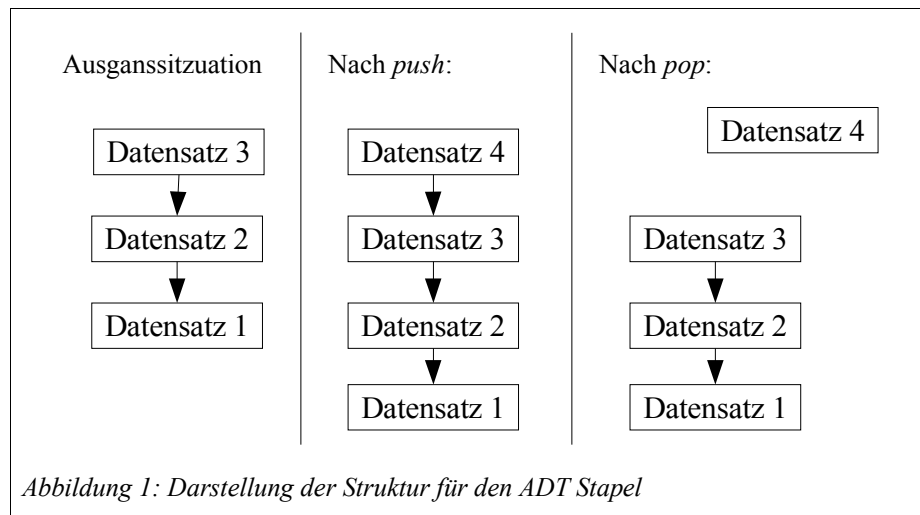
Als Grundlage für den Stapel und die Schlange wird hier die Beschreibung von Azmoodeh (1990, S. 48ff und S. 58ff) verwendet. Für die Liste und den Binärbaum werden die von Rollke und Sennholz (1994, S. 152ff und S. 176ff) gelieferten Beschreibungen zugrunde gelegt.

2.3.1 Der ADT Stapel

Der ADT Stapel wird auch als Keller oder im Englischen mit Stack bezeichnet. Diese Begriffe werden im Folgenden synonym gebraucht.

Beim Stapel ist das Einfügen bzw. Entfernen von Elementen nur an einem Ende möglich. So ist er einer der einfachsten abstrakten Datentypen.

Vorstellen kann man ihn sich als einen Stapel Bücher auf einem Tisch, auf den man oben immer wieder einzeln neue Bücher auflegen kann und die Bücher auch oben jeweils einzeln entfernt (siehe Abbildung 1).



Der Stapel arbeitet nach dem LIFO-Prinzip (Last In, First Out). Dies bedeutet, dass immer das Element als Erstes wieder ausgegeben bzw. entfernt wird, das als Letztes eingefügt wurde. Es ist stets nur das oberste Element eines Stapels bekannt, die übrigen Elemente bleiben verborgen, bis sie wieder das oberste Element sind. Dieses oberste Element wird als Kopf (engl. *top* oder *head*) bezeichnet.

Die notwendigen Operationen eines Stapels sind das Einfügen und Entfernen einzelner Elemente und das Lesen des obersten Elementes. Üblicherweise bezeichnet man die Operation zum Einfügen mit *push*, die zum Entfernen eines Elementes mit *pop*. Das Lesen des Kopfelementes wird *top* genannt. Hier hat sich auch im Deutschen die englische Bezeichnung eingebürgert. Zur Prüfung, ob der Stapel leer ist, verwendet man *empty* oder *isEmpty*.

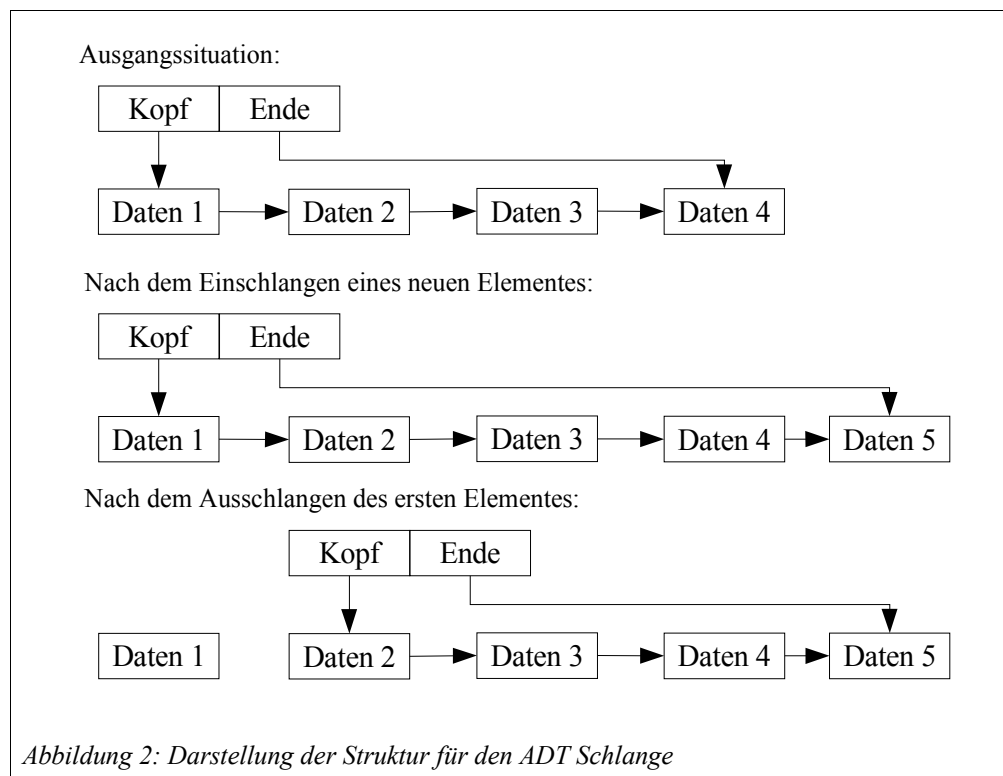
Trotz der einfachen Struktur und der recht eingeschränkten Funktionalität des Stapels wird diese Datenstruktur in vielen Bereichen in der Informatik eingesetzt, in denen der Zugriff auf Daten in umgekehrter Reihenfolge benötigt wird. So werden Stapel zur Realisierung von geschachtelten Methodenaufrufen eingesetzt. Es werden z.B. die Rücksprungadresse und lokale Daten als Datenpaket auf einen Stapel gepackt, wenn ein erneuter Methodenaufruf erfolgt. Wurde eine Methode komplett abgearbeitet, werden die Daten und die Rücksprungadresse wieder vom

Stapel geholt und weiterverarbeitet. So wird eine korrekte Programmausführung trotz beliebiger Komplexität gewährleistet.

Eine weitere Anwendung findet der Stapel bei UPN-Taschenrechnern (Umgekehrte polnische Notation), wie sie früher z.B. von Hewlett Packard gebaut wurden (vgl. Baumann 1993, S 103).

2.3.2 Der ADT Schlange

Die Datenstruktur Schlange, im Englischen als Queue bezeichnet, erlaubt im Gegensatz zum Stapel das Einfügen von Elementen ausschließlich an einem Ende, das Lesen und Entfernen aber nur am anderen Ende (siehe Abbildung 2).



Eine solche Schlange kann man sich wie eine Warteschlange vorstellen: Man stellt sich am Ende der Schlange an, wartet, bis alle vor einem der Reihe nach bedient wurden und man selbst am Anfang der Schlange steht. Nun ist man selbst an der Reihe.

Die Schlange ist eine FIFO-Struktur (First In, First Out). Dies bedeutet, dass

Elemente in der gleichen Reihenfolge wieder lesbar sind und ausgegeben werden, wie sie eingefügt wurden.

Die Schlange benötigt als wesentliche Operationen das Einfügen (*enqueue* bzw. *einschlangen*) eines Elementes am Ende der Liste sowie das Ausgeben (*front* bzw. *lies*) und das Entfernen (*dequeue* bzw. *ausschlangen*) des ersten Elementes. Daneben wird wie beim Stapel noch eine Funktion zur Überprüfung auf enthaltene Elemente zur Verfügung gestellt (*isEmpty* bzw. *istLeer*)

Sichtbar ist stets nur das Element am Anfang der Schlange, das als *head* bzw. Kopf bezeichnet wird. Die übrigen Elemente bleiben wie beim Stapel verborgen, bis sie am Anfang stehen. Dadurch wird das Ändern der Elemente verhindert. Auch für das erste Element sind keine Operationen zur Modifikation vorgesehen.

Verwendung findet die Schlange z.B. in der Verwaltung von Druckaufträgen, Prozessen, Nachrichten, Eingabegeräten etc. durch das Betriebssystem, da hier häufig die zuerst ankommenden Daten auch als erste bearbeitet werden müssen. Durch die Schlange wird dann die korrekte Reihenfolge sichergestellt.

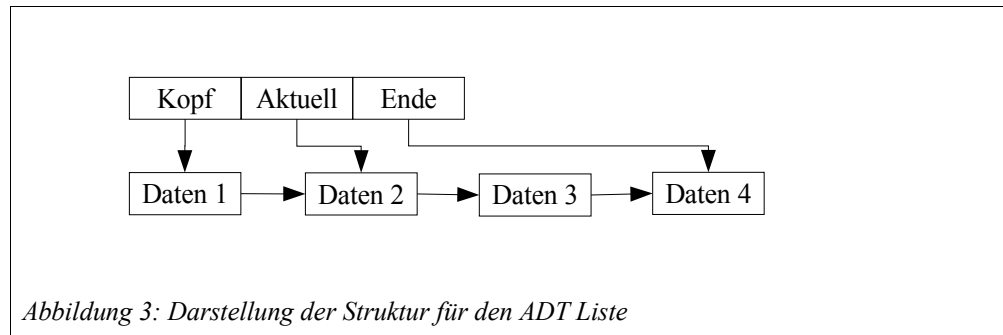
2.3.3 Der ADT Liste

Die (lineare) Liste ist die komplexeste und mächtigste der hier vorgestellten linearen Datenstrukturen. Stapel und Schlange werden häufig als besondere Liste betrachtet und durch Einschränkungen der Operationen aufbauend auf der Liste implementiert.

Eine Liste ermöglicht im Gegensatz zu Stapeln und Schlangen das Einfügen und Entfernen einzelner Elemente an beliebiger Stelle. Außerdem kann auf jedes Element zugegriffen werden, und die Elemente können auch modifiziert werden. Der Zugriff auf Elemente ist allerdings nicht direkt möglich, sondern erfolgt sequentiell. Das bedeutet, dass Elemente nicht z.B. über eine Nummerierung (wie bei Arrays) angesprochen werden können, sondern dass dies von Element zu Element über die Vorgänger-Nachfolger-Beziehung geschehen muss.

In einer Liste gibt es ein Anfangs- und ein Endelement. Üblicherweise wird aus praktischen Gründen neben dem Anfangs- und Endelement noch ein drittes Element in der Liste besonders ausgezeichnet: ein sogenanntes *aktuelles Element* (siehe

Abbildung 3). Dieses aktuelle Element wird als Bezug für viele Operationen benutzt. So wird beim Löschen z.B. das Element, das als aktuelles gekennzeichnet ist, aus der Liste entfernt oder es wird ein neues Element z.B. vor oder hinter dem aktuellen Element eingefügt.



Da in einer Liste auf jedes enthaltene Element zugegriffen werden kann und sowohl ein Stapel (LIFO) als auch eine Schlange (FIFO) als besondere Ausprägung einer Liste gesehen werden kann, besitzt die Datenstruktur Liste keine besondere Datenreihenfolge wie LIFO oder FIFO.

Im Gegensatz zu Stapel und Schlange sind für eine Liste mehr Fälle beim Einfügen eines Elementes zu betrachten:

1. Einfügen in eine leere Liste
2. Einfügen am Ende der Liste
3. Einfügen am Anfang der Liste
4. Einfügen zwischen zwei Elementen

Daher werden hier zwei Operationen verwendet: Eine zum Einfügen eines neuen Elementes vor (*fuegeEinVor*) und eine zum Einfügen nach dem aktuellen Element (*fuegeEinNach*). Zum Entfernen wird lediglich eine Operation (*entferne*) benötigt, die das aktuelle Element löscht. Daneben wird eine Funktion zum Ändern eines Elementes angeboten (*aendere*).

Um aber diese Operationen überhaupt sinnvoll einsetzen zu können, muss zunächst die Möglichkeit gegeben werden, den Verweis auf das aktuelle Element zu ändern. Hierzu werden die Operationen *findeErstes*, *findeLetztes* und *findeNaechstes* bereitgestellt.

Daneben gibt es wie bereits bei den ADTs Stapel und Schlange eine Operation zum

Lesen des aktuellen Elementes (*lies*) und eine Operation, die einen Wahrheitswert liefert, ob die Liste leer ist (*istLeer*).

Neben der hier vorgestellten linearen, einfach verketteten Liste werden häufig doppelt verkettete Listen benutzt, bei denen neben dem Nachfolgeelement auch jeweils das Vorgängerelement eines Elementes bekannt ist. Eine weitere spezielle Liste ist die sogenannte Ringliste, die keinen Anfang und kein Ende besitzt, da sie kreisförmig wie z.B. eine Perlenkette aufgebaut ist.

2.3.4 Der ADT Binärbaum

Der Binärbaum ist im Gegensatz zu den bisher betrachteten Strukturen nicht linear aufgebaut, sondern hierarchisch.

Die Einschränkung auf binäre Bäume an dieser Stelle liegt darin begründet, dass Bäume mit mehreren Nachfolger prinzipiell gleich aufgebaut sind, aber erheblich komplexere Betrachtungen erfordern. Der Binärbaum ist der Schritt von den linearen Datenstrukturen hin zu einer hierarchischen, an der die wichtigen Eigenschaften beobachtet werden können, ohne jedoch auf Besonderheiten bei Bäumen mit mehr Nachfolgern eingehen zu müssen. Die Vorstellung von einem Binärbaum mit einfachem Größer-Kleiner-Vergleich z.B. für Suchbäume ist sehr anschaulich.

Ein Binärbaum zeichnet sich dadurch aus, dass jedes Element (=Knoten) außer dem ersten Element (=Wurzel) einen Vorgänger und null bis zwei direkte Nachfolger (daher die Charakterisierung *binär*) hat, die als linker bzw. rechter Nachfolger bezeichnet werden. Elemente ohne Nachfolger sind besondere Knoten und heißen Blätter. Häufig wird auch die Bezeichnungen Vater und Sohn für Vorgänger- bzw. Nachfolgeknoten verwendet.

Die Verbindung zwischen zwei Knoten wird Kante genannt, eine Folge von Kanten heißt Pfad. Elemente, die von der Wurzel gleich weit entfernt sind, liegen auf derselben Ebene. Mit Tiefe eines Baumes wird die Anzahl der Ebenen bezeichnet.

Bäume werden typischerweise rekursiv verstanden:

Da die Nachfolger eines Knotens ihrerseits als Wurzeln von Bäumen aufgefasst werden können, d.h. als Wurzeln jeweils Teilbäume des gesamten Baumes

konstruieren, kann man binäre Bäume auch **rekursiv** definieren:

$Baum = leer$

| falls Anzahl der Knoten = 0

$Baum = Baum + WurzelKnoten + Baum$

| ansonsten

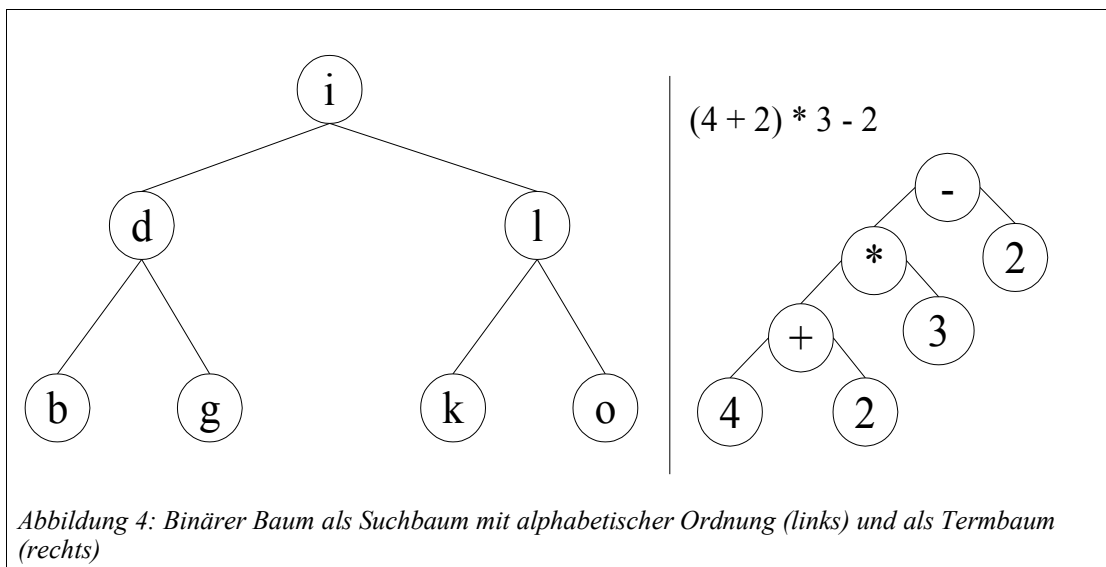
(Rollke und Sennholz, 1994, S. 117)

So kann jeder Knoten als Wurzel eines (Teil-)Baumes aufgefasst werden, Blätter sind demnach Wurzeln mit zwei leeren Teilbäumen.

Verwendung findet der Binärbaum unter anderem beim Suchbaum, Termbaum oder als Wissensbasis für sog. Expertensysteme:

Als Suchbaum bildet er durch seine Ordnung die Grundlage für die schnelle binäre Suche. Der in Abbildung 4 dargestellte Baum enthält alphabetisch geordnete Knoten. Für einen Knoten gilt, dass alle Knoten, die links von ihm sind, „kleinere“ Elemente enthalten, und alle, die rechts folgen, „größere“ Elemente enthalten.

Mit einem Termbaum können arithmetische Ausdrücke eindeutig dargestellt werden (Abbildung 4).



Bei einer Wissensbasis für Expertensysteme kann z.B. mit Hilfe von Ja/Nein-Fragen eine genaue Bestimmung vorgenommen werden, etwa: Liegt die Stadt westlich des Rheins? → Nein → Liegt die Stadt nördlich der Donau? → Ja → usw. → Ist die Stadt Berlin? → Ja.

3 UML und CASE-Tools

3.1 Die Unified Modeling Language (UML)

Erste Ideen zur Objektorientierung hat es schon in den 1970er Jahren gegeben. Bereits kurz danach hat es mit Smalltalk eine erste kommerziell verfügbare objektorientierte Entwicklungsumgebung gegeben. Sie ist zwar heute nicht mehr sehr populär, wird aber stets als Referenz für Objektorientierung angesehen. In dieser Zeit sind bereits Publikationen zum objektorientierten Programmieren erschienen, eine umfassende Methode für objektorientierte Analyse und objektorientiertes Design hat sich aber erst seit Anfang der 1990er Jahre herausgebildet. Sehr beliebt waren hier die Methoden von Grady Booch und James Rumbaugh. Auf der Grundlage ihrer und der Methoden anderer Autoren entwickelte man schließlich die Unified Modeling Language.

„Die Unified Modeling Language (UML) ist eine Sprache und Notation zur Spezifikation, Konstruktion, Visualisierung und Dokumentation von Modellen für Softwaresysteme.“ (Oesterreich 2004, S. 209)

Die UML entwickelte sich zu einem Quasi-Standard und wurde später (in der Version 1.1 von 1997) auch offiziell als Standard anerkannt. Seitdem wurde die UML stets weiterentwickelt und stellenweise korrigiert, derzeit liegt mit der Version 2.0 eine komplette Überarbeitung vor, in die auch die Ideen vieler anderer Autoren eingeflossen sind.

Die UML hat sich in der Fachwissenschaft als Standard-Notation beim Softwareengineering durchgesetzt. Sie wird hier in allen Bereichen als durchgängige Notation verwendet und nimmt somit eine Schlüsselfunktion ein. Mit ihr lassen sich zahlreiche Aspekte bei der Softwareentwicklung beschreiben, wobei dies auf einer abstrakten Ebene stattfindet, sodass sie unabhängig von einer bestimmten Programmiersprache verwendet werden kann.

Zur effektiven Nutzung von UML sollte ein Grundverständnis von Objektorientierung vorhanden sein (obwohl UML selbst nicht auf OO beschränkt ist (vgl. Arlow und Neustadt 2002, S. 5)), die wichtigen Prinzipien und Bezeichnungen sollten bekannt sein. Mit diesem Wissen kann die UML erfolgreich zur Analyse und zum Design von Software verwendet werden. Sie bietet Darstellungsmöglichkeiten,

die das Wesentliche einer Software beschreiben, die Implementation aber nicht vorwegnehmen. Aufbauend auf dem durch die UML unterstützten Entstehungsprozess kann später die Software implementiert werden. Erst hier muss auf eine konkrete Programmiersprache Rücksicht genommen werden.

Eine nähere Beschreibung über die Entstehung und die Grundlagen der UML findet sich z.B. bei Bernd Oesterreich (Oesterreich 2004).

Anhand grafischer Darstellungen ist es leichter möglich, komplexe Zusammenhänge dennoch übersichtlich darzustellen. Ein Betrachter wird aufgrund dieser Strukturiertheit schneller eine Übersicht und Einsicht in diese Zusammenhänge gewinnen können als dieses auf rein textueller Basis möglich ist.

3.2 CASE-Tools

CASE-Tools (Computer Aided Software Engineering) unterstützen den Entwickler beim Entwurf eines Softwaresystemes, besonders im Hinblick auf die Verwendung von UML:

UML was designed to incorporate current best practice in modeling techniques and software engineering. As such, it is explicitly designed to be implemented by computer-assisted software engineering (CASE) tools. [...] UML diagrams are human-readable and yet are easily rendered by CASE programs. (Arlow und Neustadt 2002, S. 5)

In der professionellen Softwareentwicklung werden solche CASE-Tools bereits erfolgreich eingesetzt.

Die CASE-Tools ermöglichen es, die Architektur einer Software mittels UML zu beschreiben, wobei ihre Hauptaufgabe darin besteht, die Konsistenz der Diagramme sicherzustellen. Darüberhinaus kann aus den erzeugten UML-Diagrammen durch diese Tools Quelltext erzeugt werden. Dies wird als Forward-Engineering bezeichnet.

Außerdem gibt es bereits CASE-Tools, die sogenanntes Reverse-Engineering ermöglichen. Hier werden für bereits vorliegenden Quellcode UML-Diagramme erzeugt. Diese lassen sich dann in der Regel leichter als der zugrunde liegende Quellcode analysieren.

Die Software kann so mithilfe dieser CASE-Tools aufbauend auf UML allgemein beschrieben werden, die Festlegung auf eine bestimmte Programmiersprache ist hier

(noch) nicht zwingend erforderlich. Eine Kenntnis der Eigenheiten und der Syntax einer Programmiersprache ist hier (fast) nicht notwendig. Dagegen muss aber die UML und ihre Verwendung bekannt sein. Je nach Umfang des eingesetzten CASE-Tools ist häufig nur die genaue Kenntnis bestimmter Diagrammtypen notwendig, da die meisten derzeit existierenden Tools immer nur einen bestimmten Teil der UML verwenden und verarbeiten können. Ebenso kann man sich aber auch bei der Nutzung auf bestimmte Bereiche beschränken, die durch das CASE-Tool mithilfe von UML bearbeitet werden sollen. So kann man z.B. mit Klassendiagrammen die Klassen- und Methodenrumpfe erzeugen und die weitere Implementierung im Quelltext vornehmen.

Für den Informatikunterricht sollte man sich auf bestimmte Bereiche der UML beschränken, da ein umfassendes Verständnis der mittlerweile sehr komplexen UML in der Schule sicher nicht erreicht werden kann. Durch die Verwendung von CASE-Tools kann den Schülerinnen und Schülern aber ein erheblicher Anteil an Arbeit bei der Modellierung und anschließenden Implementierung von Softwaresystemen abgenommen werden, wenn z.B. aus den leicht überschaubaren Klassendiagrammen direkt der Quelltext für die Klassen erzeugt werden kann. Hubwieser sieht in Code-Generatoren eine interessante Perspektive: „Damit wird den Schülern unmittelbar die Bedeutung ihrer Modellierungsergebnisse vor Augen geführt.“ (Hubwieser 2004, S. 75). Eine Verwendung von CASE-Tools im Unterricht ist somit bei der Verwendung von UML wünschenswert.

Ein Vergleich mehrerer CASE-Tools ist auf den Seiten des life³-Projektes zu finden (life-Webseite).

3.3 Das CASE-Tool Fujaba

Fujaba ist ein CASE-Tool, das es ermöglicht, Programme mithilfe der UML zu entwickeln. Fujaba steht für 'From UML to Java And Back Again' und erstellt automatisch aus UML-Diagrammen lauffähigen Java-Quellcode ('From UML to Java'). Umgekehrt bietet Fujaba aber auch die Möglichkeit des Reverse-Engineerings, sodass aus korrektem Java-Quellcode UML-Diagramme erstellt werden können ('And Back Again').

Die für diese Arbeit benutzte Fujaba-Version wurde extra für das life³-Projekt (life-

Webseiten) zum Einsatz im Anfangsunterricht Informatik in der Jahrgangsstufe 11 angepasst. Sie beschränkt sich auf Klassen- und Aktivitätsdiagramme zur Erzeugung des Quellcodes. Die in Fujaba verwendeten Aktivitätsdiagramme existieren allerdings so nicht in UML, sie sind ein Ersatz für die UML-Sequenz- und Kollaborationsdiagramme. Mit Hilfe dieser Aktivitätsdiagramme werden in Fujaba die Methoden grafisch implementiert.

Neben der eigentlichen Entwicklungsumgebung stellt Fujaba das Programm DOBS (Dynamic Object Browsing System) zur Verfügung, mit dem die internen Objektstrukturen der Java Virtual Machine dargestellt werden können. In Dobs können Objekte von Klassen erzeugt und auf diesen Objekten Methoden aufgerufen werden. Veränderungen in der Objektstruktur werden dann grafisch angezeigt.

3.3.1 Überblick

Die Oberfläche von Fujaba (Abbildung 5) präsentiert unter der Menü- und der Funktionsleiste eine zweigeteilte Ansicht. Im linken Bereich ist eine Übersicht über die Klassen- und Aktivitätsdiagramme sichtbar. Hier kann ausgewählt werden, welches Diagramm im rechten Bereich dargestellt werden soll. Mit dem darunter befindlichen Schieber kann die Darstellungsgröße der Diagramme verändert werden, sodass z.B. auch sehr komplexe Diagramme übersichtlich dargestellt werden können.

Fujaba ermöglicht es auch, den für eine Klasse oder eine Methode erzeugten Quellcode anzeigen zu lassen. Den Schülerinnen und Schülern wird so die Möglichkeit gegeben, die Auswirkungen von Veränderungen in der UML-Notation auf den Quelltext zu beobachten. So kann der Übergang von der grafischen Implementierung in Fujaba hin zu einer auf Quellcode-Ebene unterstützt werden.

In dieser Arbeit und der später beschriebenen Unterrichtsreihe wird aber nur die grafische Programmierung eingesetzt, der Quellcode wird nicht betrachtet.

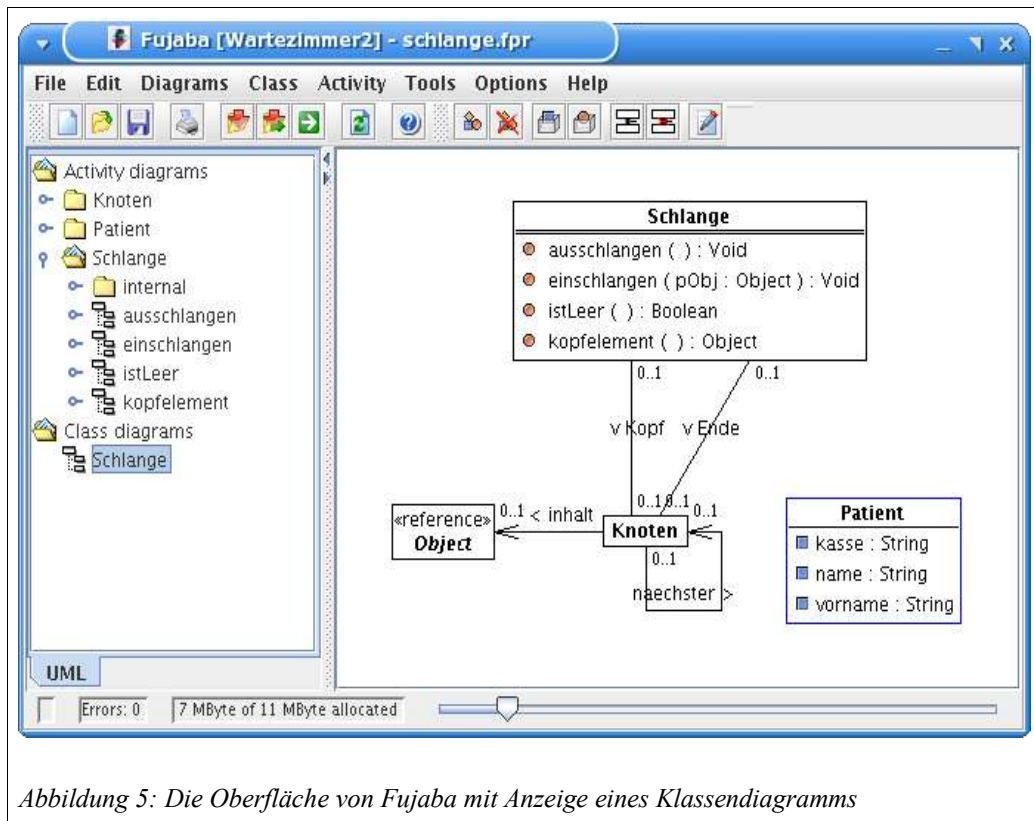
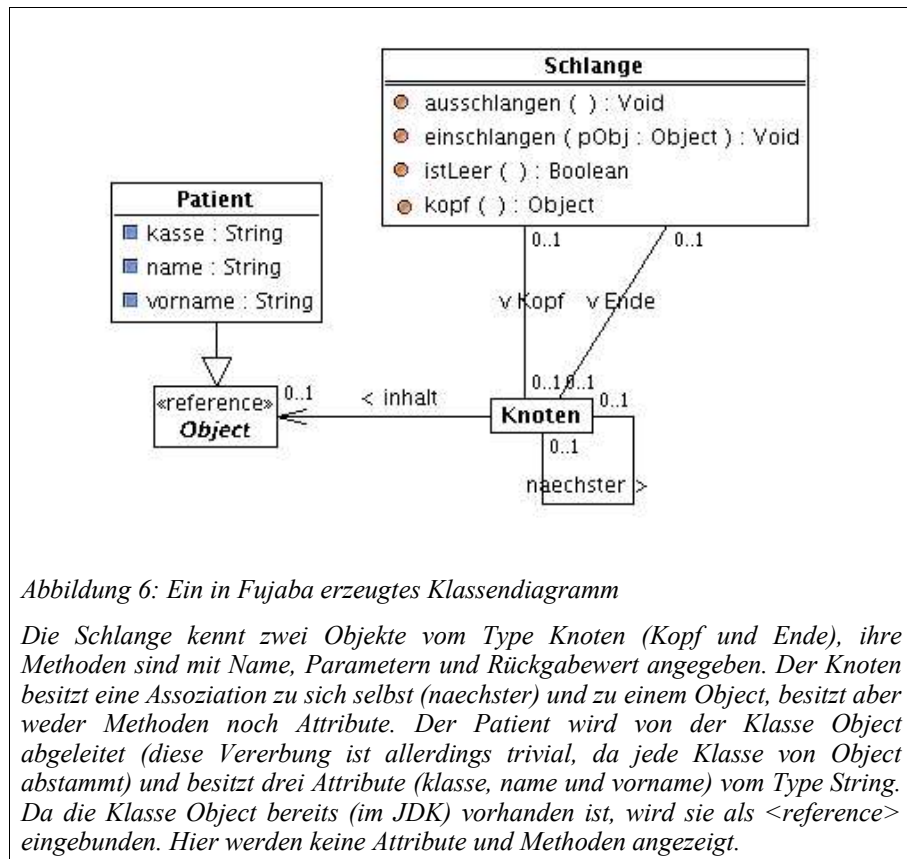


Abbildung 5: Die Oberfläche von Fujaba mit Anzeige eines Klassendiagramms

3.3.2 Klassendiagramme

Klassendiagramme werden in Fujaba in einer üblichen UML-Notation angezeigt (siehe Abbildung 6). Für jede Klasse (rechteckiger Kasten) werden Attribute (mit blauem Quadrat) und Methoden (mit rotem Kreis) angezeigt. Aus dem Klassendiagramm kann über ein kontextsensitives Menü direkt zu den einzelnen Aktivitätsdiagrammen für die Methoden gesprungen werden. Beziehungen zwischen den einzelnen Klassen werden durch die üblichen Beziehungselemente (Vererbung, gerichtete/ungerichtete Assoziation ...) angezeigt.

Eine genaue Beschreibung zu den Klassendiagrammen kann der Fujaba-Dokumentation entnommen werden (Fujaba-Webseite).



3.3.3 Aktivitätsdiagramme

Durch Aktivitätsdiagramme werden in Fujaba die Methoden implementiert, die im Klassendiagramm festgelegt wurden.

In Abbildung 7 ist ein Fujaba-Aktivitätsdiagramm abgebildet. Die Ausführung einer Methode beginnt bei dem Methodenkopf (1) oben links. Von dort aus zeigen Pfeile, die Transitionen, den algorithmischen Ablauf an. Die wichtigsten Elemente in einem Aktivitätsdiagramm sind die Story-Pattern (2, 3 und 4).

Ein Story-Pattern übernimmt drei Aufgaben:

1. Es prüft, ob die angegebene Objektstruktur tatsächlich vorhanden ist. Hierbei wird stets von einem bekannten Objekt ausgegangen. Ein solches bekanntes Objekt ist gebunden und wird mit bound bezeichnet. Diese Objekte besitzen keine Klassenbezeichnung hinter dem Namen. Im Beispiel prüft Story-Pattern (2), ob das this-Objekt (this ist hier bound) mit der Beziehung top auf ein Objekt der Klasse Node verweist.
2. Falls die Prüfung in 1. erfolgreich war, wird die Objektstruktur erweitert, falls

dies im Story-Pattern gefordert wird. Neue Objekte werden eingefügt und Beziehungen gesetzt. Dargestellt wird dies durch das Wort `<create>` und die grüne Färbung. Außerdem können Attributwerte gesetzt werden. Im Beispiel erzeugt Story-Pattern (2) ein Objekt `newNode` und setzt die Beziehungen `top` und `next`.

3. Ebenfalls nur im Erfolgsfall der Prüfung im 1. Schritt werden Objekte oder Beziehungen gelöscht. Dies ist erkennbar durch die rote Farbe und das Wort `<destroy>`. Im Beispiel entfernt Story-Pattern (2) die Beziehung `top` auf das Objekt `first`. (Da die Beziehung `top` aber bereits im 2. Schritt dieses Story-Patterns umgesetzt wurde, wird das Löschen in diesem Fall nicht mehr in den Quelltext aufgenommen; ein späteres Löschen würde hier einen Fehler produzieren)

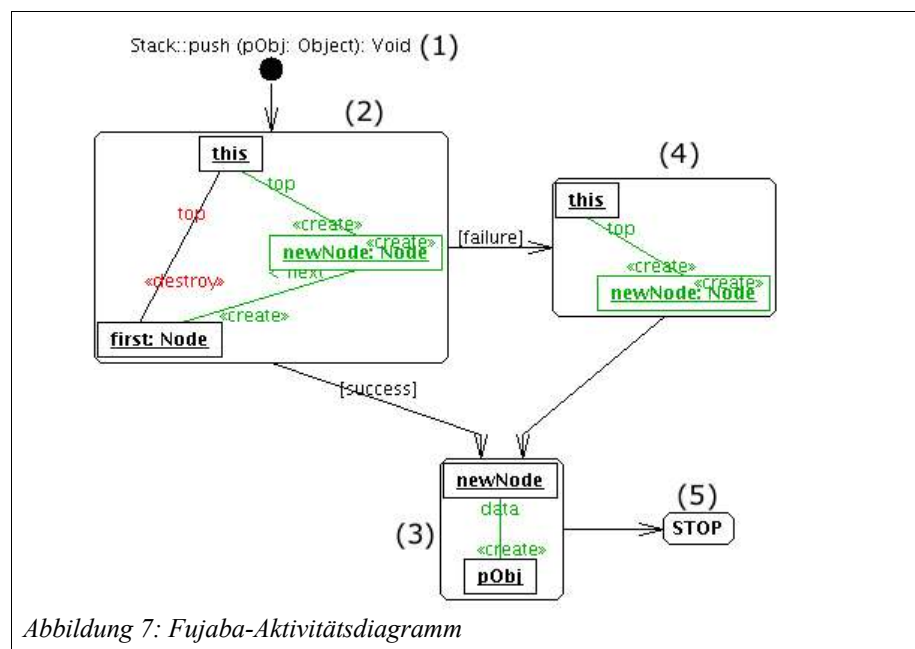


Abbildung 7: Fujaba-Aktivitätsdiagramm

Story-Pattern (2) besitzt in dem Beispiel zwei ausgehende Transitionen, die beide mit einem booleschen Ausdruck versehen sind. Wurde die Prüfung im 1. Schritt erfolgreich durchgeführt, wird der `[success]`-Transition gefolgt, andernfalls der `[failure]`-Transition. In dem Beispiel würde in Story-Pattern (2) ein neues Objekt in die Objektstruktur am Anfang des Stapels eingefügt. Falls aber noch kein Element

vorhanden war (failure), ist das neue Objekt das einzige und würde in Story-Pattern (4) eingefügt. Story-Pattern (3) erzeugt noch eine Beziehung von dem neuen Knotenobjekt zum übergebenden Datenobjekt (pObj). Die Ausführung einer Methode endet stets mit einem STOP (5).

Einige weitere wichtige Elemente eines Aktivitätsdiagrammes werden in Abbildung 8 beschrieben: Die Raute (1) ist ein sog. NOP (No-Operation). Anhand von Transitionen mit booleschem Ausdruck können hiermit Verzweigungen realisiert werden. In Aktivitätsdiagrammen kann auch direkt Java-Code in einem sog. Statement (2) verwendet werden. Mit Collaboration-Statements (3) kann in einem Story-Pattern Java-Code direkt zu einem Objekt verwendet werden. Hier wird z.B. die Methode `knotenanzahl()` auf dem Objekt, das mit `linkerBaum` bezeichnet wird, ausgeführt und das Ergebnis dann der Ganzzahl-Variablen `anzahlLinks` zugewiesen.

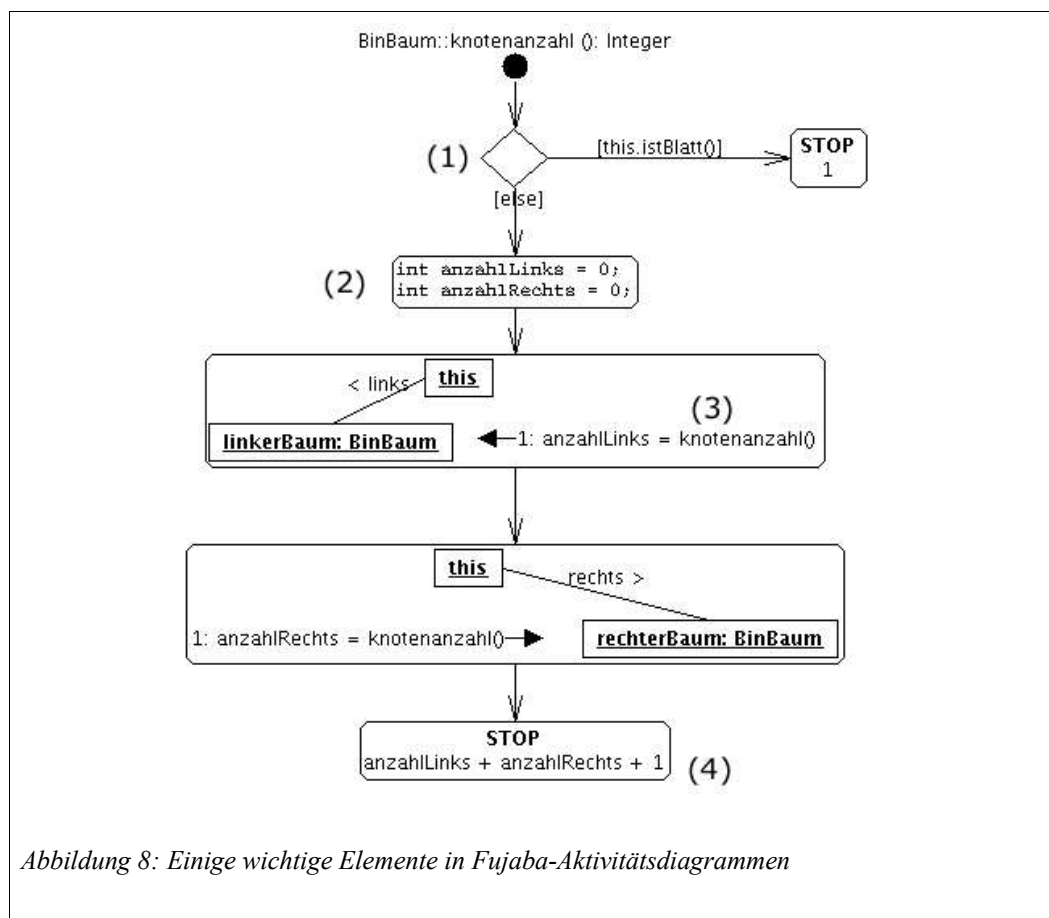


Abbildung 8: Einige wichtige Elemente in Fujaba-Aktivitätsdiagrammen

3.3.4 Das Dynamic Object Browsing System 'Dobs'

Dobs ermöglicht es, die Objektstrukturen grafisch darzustellen. Hilfreich ist besonders die Visualisierung der Beziehungen zwischen den Objekten. Durch das Ausführen der vorhandenen Methoden kann der Anwender interaktiv diese Objektstrukturen ändern.

Auf der linken Seite zeigt Dobs die bekannten Objekte an. Darunter werden kontextsensitiv die Attributwerte und Methoden eines Objektes angezeigt. Im großen Rahmen auf der rechten Seite ist die aktuelle Objektstruktur zu sehen. Objekte werden hier mit Klassennamen und (eventuell vorhandenen) Attributwerten angezeigt. Ebenfalls werden die Beziehungen zwischen Objekten mit Pfeilen dargestellt.

Verfügbare Methoden können ausgeführt werden, Parameter werden bei Bedarf mit einer Eingabeaufforderung durch den Benutzer festgelegt.

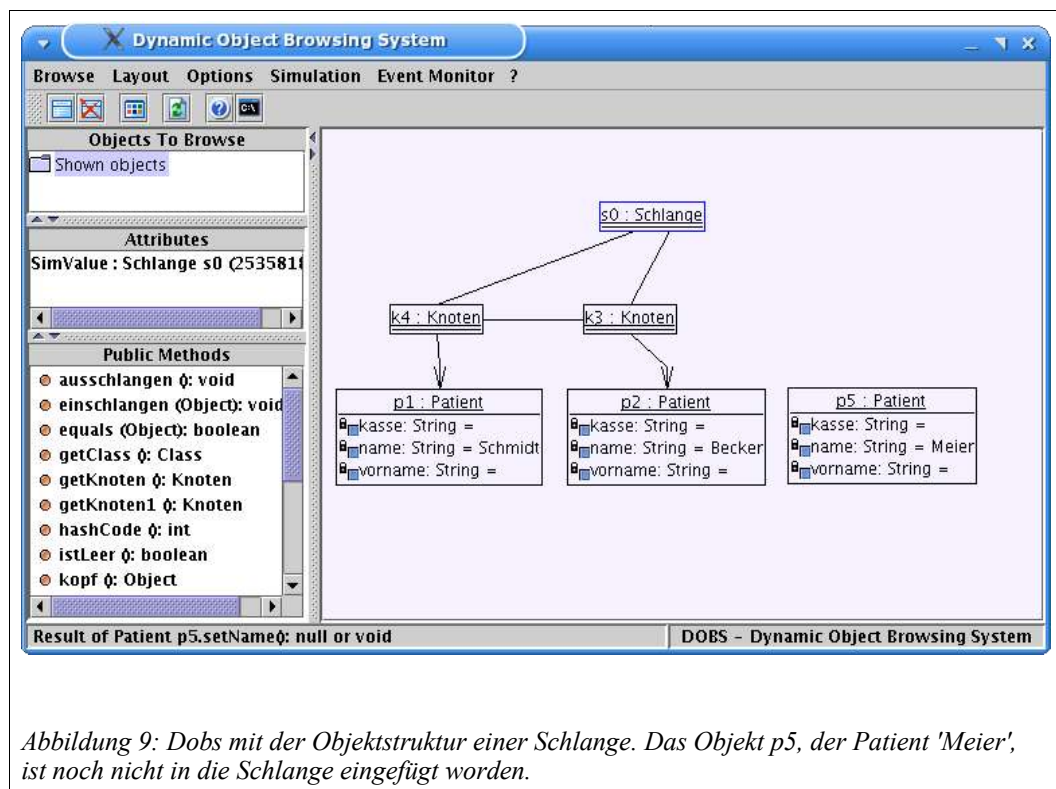


Abbildung 9: Dobs mit der Objektstruktur einer Schlange. Das Objekt p5, der Patient 'Meier', ist noch nicht in die Schlange eingefügt worden.

Die Schülerinnen und Schüler sollen Dobs benutzen, um das Verhalten der Methoden auf die Objektstrukturen analysieren zu können. Einerseits werden selbst implementierte Projekte getestet, andererseits sollen vorgegebene Projekte nur

anhand der Objektstruktur und des Laufzeitverhaltens analysiert werden.

Bei relativ kleinen Objektstrukturen ist die Benutzung von Dobs gut möglich. Werden sie aber komplexer, kann vermutlich nicht sofort die ganze Struktur erfasst werden. Objekte, die von Methoden und nicht vom Benutzer direkt erzeugt werden, zeigt Dobs nicht sofort an. Der Benutzer muss erst den Befehl 'Expand' auf einem bestimmten Objekt ausführen, danach werden dann die über Beziehungen verbundenen Objekte dargestellt. Für unerfahrene Benutzer ist dies umständlich und führt schnell zu Missverständnissen, da z.B. Objekte schon existieren, aber unsichtbar sind. So nimmt man schnell an, diese Objekte wären noch gar nicht vorhanden. Dies führt häufig zu fehlerhafter Bedienung.

Ein weiterer Nachteil bei der Darstellung in Dobs tritt beim Entfernen von Objekten auf. Im Programm wird z.B. durch eine Methode ein Objekt gelöscht, in Dobs wird es aber immer noch angezeigt. Dieses Verhalten hängt mit dem Garbage Collector von Java zusammen, der nicht mehr referenzierte Objekte dann löscht, wenn mehr Speicherplatz benötigt wird. Da aber diese Objekte noch von Dobs referenziert werden, wenn sie angezeigt werden, kann der Garbage Collector die Objekte auch nicht löschen. Somit verbleiben sie im Speicher und Dobs zeigt sie auch weiterhin an. Hier besteht die Gefahr, dass korrekt implementierte Methoden durch eine falsche Darstellung der Objektstruktur in Dobs als inkorrekt angesehen werden und eine Suche nach einem Fehler beginnt, der gar nicht vorhanden ist. Durch überstürztes Handeln können sich dann erst recht Fehler einschleichen.

Diese Eigenheiten von Dobs müssen den Schülerinnen und Schülern bekannt sein, damit das Werkzeug korrekt eingesetzt werden kann.

Damit die Schülerinnen und Schülern bei der Analyse von Objektstrukturen nicht erst die Projektdatei in Fujaba öffnen, um von dort Dobs starten zu können, sollten nur die Binärdateien zur Verfügung gestellt werden und Dobs direkt gestartet werden.¹

¹ Der Befehl „java -mx64m -cp /Pfadzu/fujaba.jar:/home/andreas/fujaba3:/de.uni_paderborn.fujaba.dobs.DobsApp“ führt (unter Linux) Dobs direkt aus und benutzt die Klassen, die sich im Startverzeichnis befinden. Praktischerweise könnte die Lehrperson eine Skript-Datei zur Verfügung stellen, die Dobs in dem Verzeichnis mit den zu untersuchenden Klassen ausführt. Unter Windows ist dieses Vorgehen ebenfalls möglich. Die Pfadangaben müssen natürlich der Installation entsprechend angepasst werden.

4 Implementierung der ADTs in Fujaba

An dieser Stelle wird ein Überblick darüber gegeben, wie die vier betrachteten ADTs Stack, Schlange, Liste und Binärbaum mithilfe von Fujaba implementiert werden können. Dargestellt wird jeweils nur eine Variante, die an die in Kapitel 2 dargestellten Beschreibungen anknüpft, wenn auch mehrere verschiedene Implementierungen möglich wären. Hierzu werden die Klassendiagramme und die Aktivitätsdiagramme ausgesuchter Methoden erläutert. Es wird auf die Darstellung aller Methoden verzichtet².

Die Implementierungen der drei linearen ADTs verwenden stets Knotenobjekte zur Verwaltung der Datenstruktur. Diese Knotenobjekte verweisen auf Objekte der Klasse *Object*, so dass beliebige Objekte als Inhalt von dem jeweiligen ADT aufgenommen werden können. In den folgenden Abschnitten wird mit 'Element' stets eine Einheit aus Knoten- und Inhaltsobjekt benannt, so dass z.B. mit 'Einfügen eines Elementes' das Einfügen eines Knotenobjektes zur Verwaltung mit gleichzeitigem Setzen der Referenz vom Knoten- auf das Inhaltsobjekt gemeint ist

Da die Algorithmen allgemein bekannt sind, werden die einzelnen Aktivitätsdiagramme nur kurz erklärt.

4.1 Der ADT Stack

4.1.1 Klassendiagramm

Der Stack bzw. Stapel wird durch die Klasse *Stack* implementiert (siehe Abb. 10). Ein Exemplar der Klasse *Stack* zeigt – sofern vorhanden – mit der Beziehung *top* auf das oberste Knotenobjekt (Klasse *Node*) des Stapels. Die Knotenobjekte referenzieren (über die Beziehung *data*) ein Inhaltsobjekt (*Object*) und kennen das nächste (*next*) Knotenexemplar.

² Die Dateien der Fujaba-Projekte sind auf der beigelegten CD enthalten, hier können die übrigen Methoden eingesehen werden

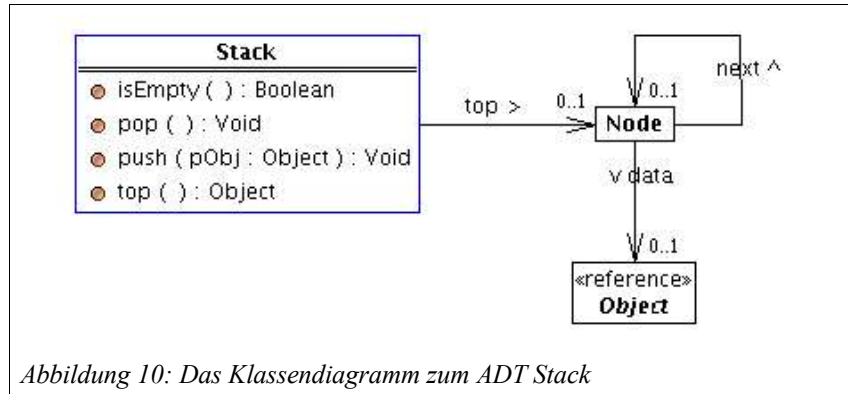
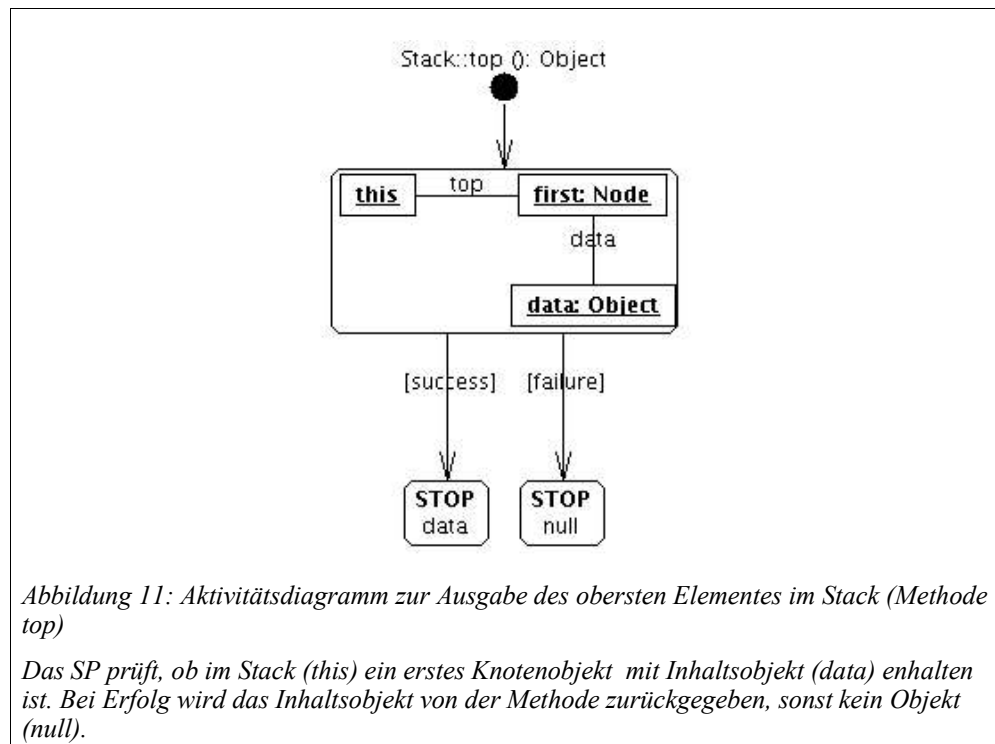


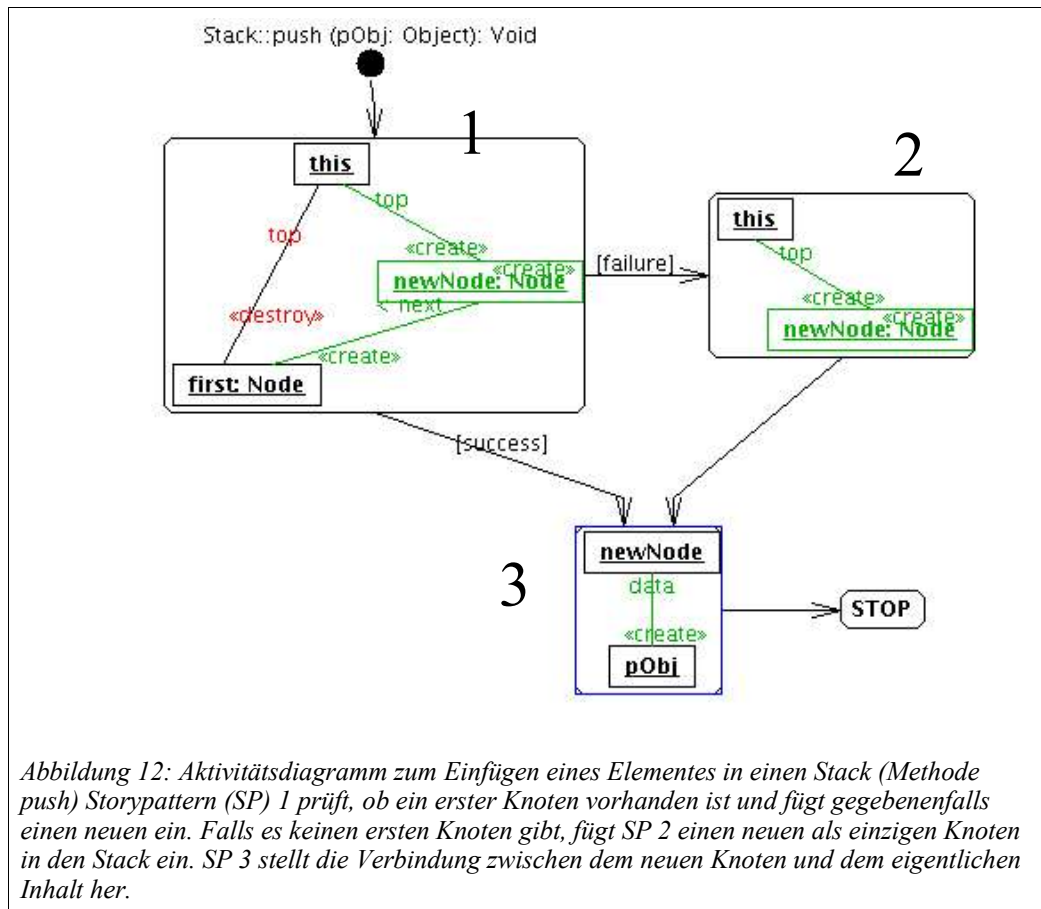
Abbildung 10: Das Klassendiagramm zum ADT Stack

4.1.2 Aktivitätsdiagramme

Exemplarisch werden hier die Aktivitätsdiagramme zu den Methoden `push()` zum Einfügen eines neuen Elementes und `top()` zur Ausgabe des obersten Elementes beschrieben. Die Methoden zum Entfernen eines Elementes und zur Überprüfung, ob der Stack leer ist, sind denen für den ADT Schlange sehr ähnlich. Auf sie wird in Abschnitt 4.2.2 eingegangen.

Abbildung 11: Aktivitätsdiagramm zur Ausgabe des obersten Elementes im Stack (Methode `top`)

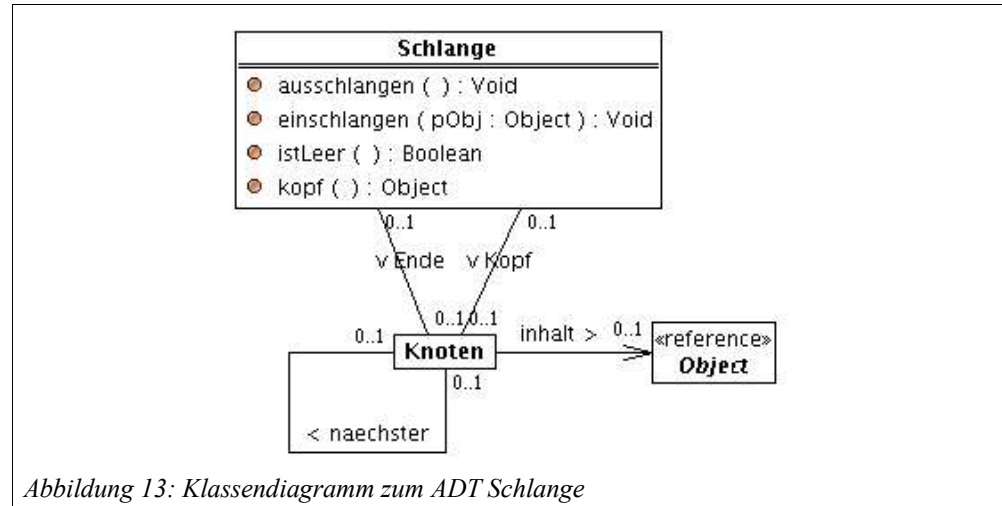
Das SP prüft, ob im Stack (`this`) ein erstes Knotenobjekt mit Inhaltsobjekt (`data`) enthalten ist. Bei Erfolg wird das Inhaltsobjekt von der Methode zurückgegeben, sonst kein Objekt (`null`).



4.2 Der ADT Schlange

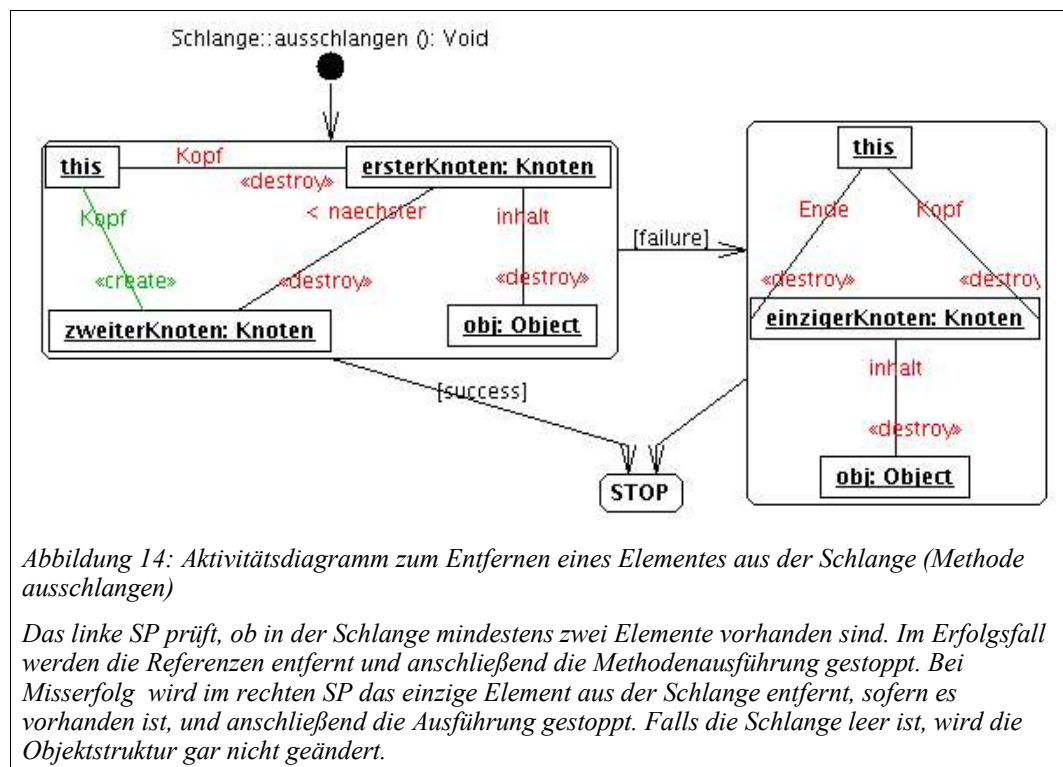
4.2.1 Klassendiagramm

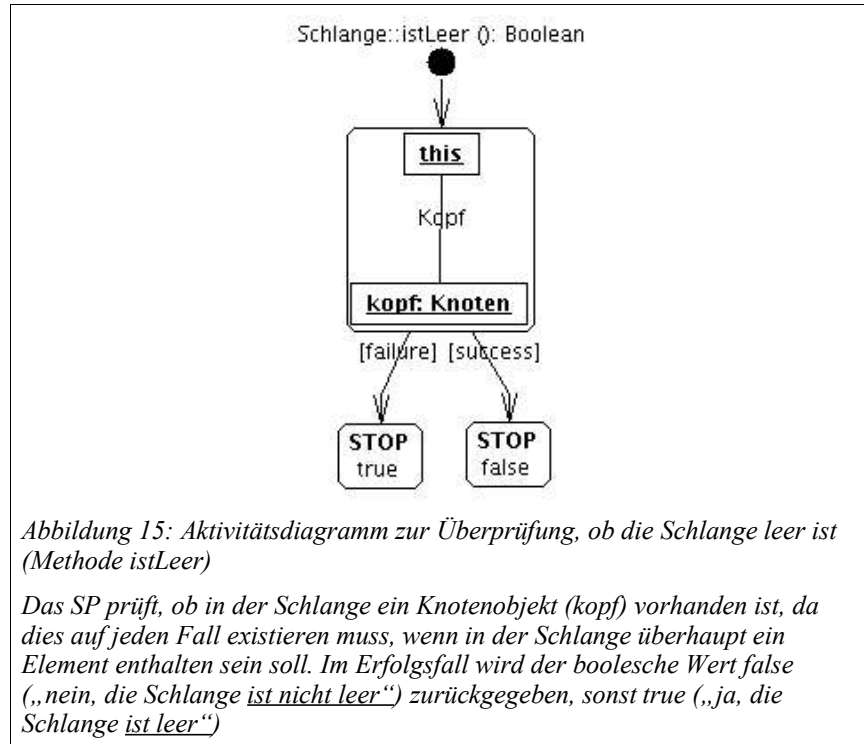
Das Klassendiagramm (Abbildung 12) für den ADT Schlange entspricht dem für den Stack (hier allerdings mit deutschen Bezeichnungen) ergänzt um eine Beziehung zur Auszeichnung des letzten (Knoten-)Elementes.



4.2.2 Aktivitätsdiagramme

Hier werden die Aktivitätsdiagramme zur Implementierung der Methoden *ausschlangen* und *istLeer* dargestellt. Die beiden Methoden *einschlangen* und *kopf* sind ganz ähnlich zu implementieren wie die Methoden *push* und *top* für den Stack.

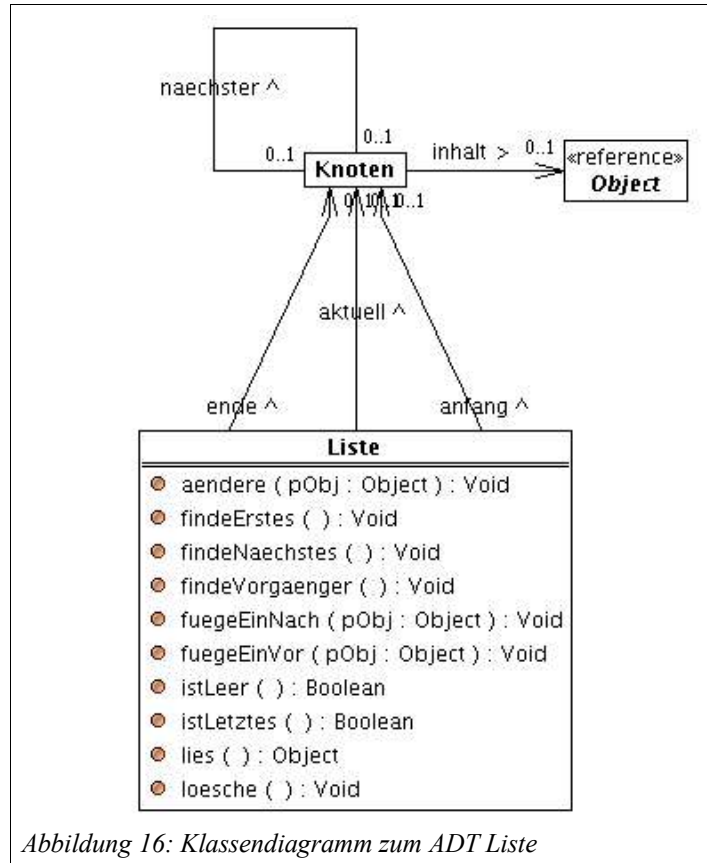




4.3 Der ADT Liste

4.3.1 Klassendiagramm

Das Klassendiagramm zur Liste baut auf dem der Schlange auf. Es gibt jetzt aber drei Beziehungen zur Klasse `Knoten`: *anfang*, *aktuell* und *ende*. Die Methoden werden entsprechend der Struktur angepasst. Es sind hier nur einige grundlegende Methoden implementiert. Klassen, die für speziellere Einsatzgebiete gedacht sind, könnten von dieser Klasse `Liste` abgeleitet werden und weitere Methoden (z.B. zur Suche oder zum Zählen der Elemente) implementieren.

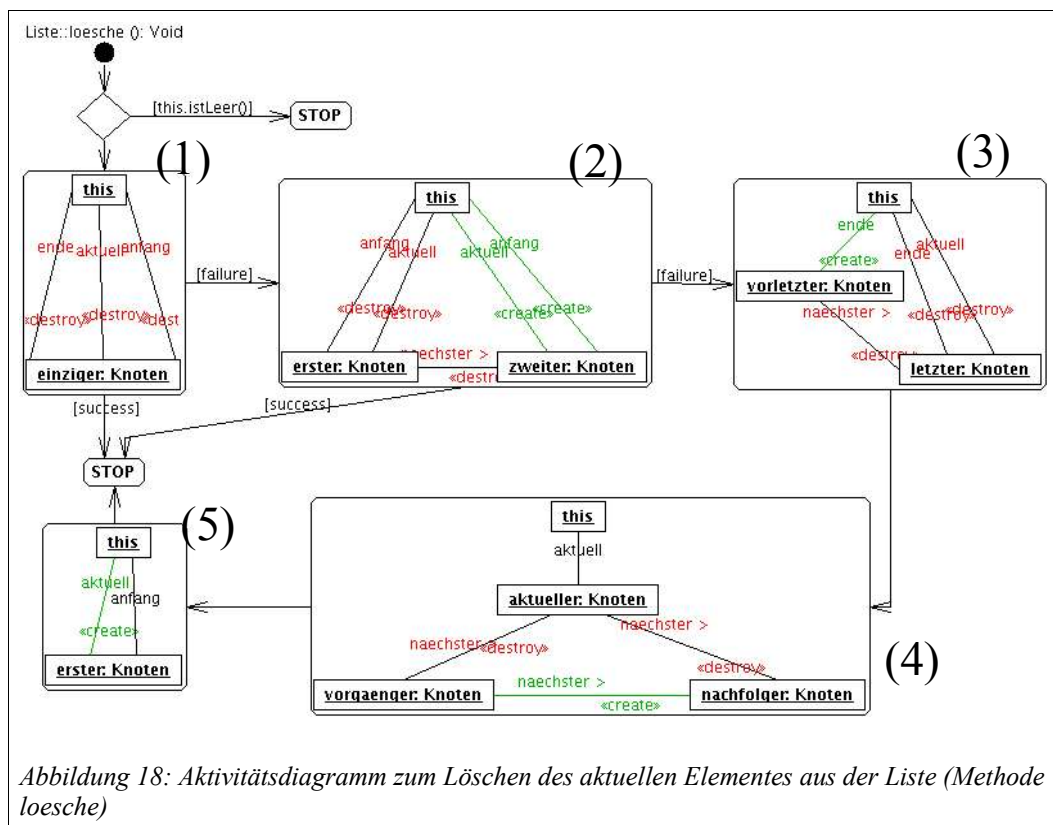
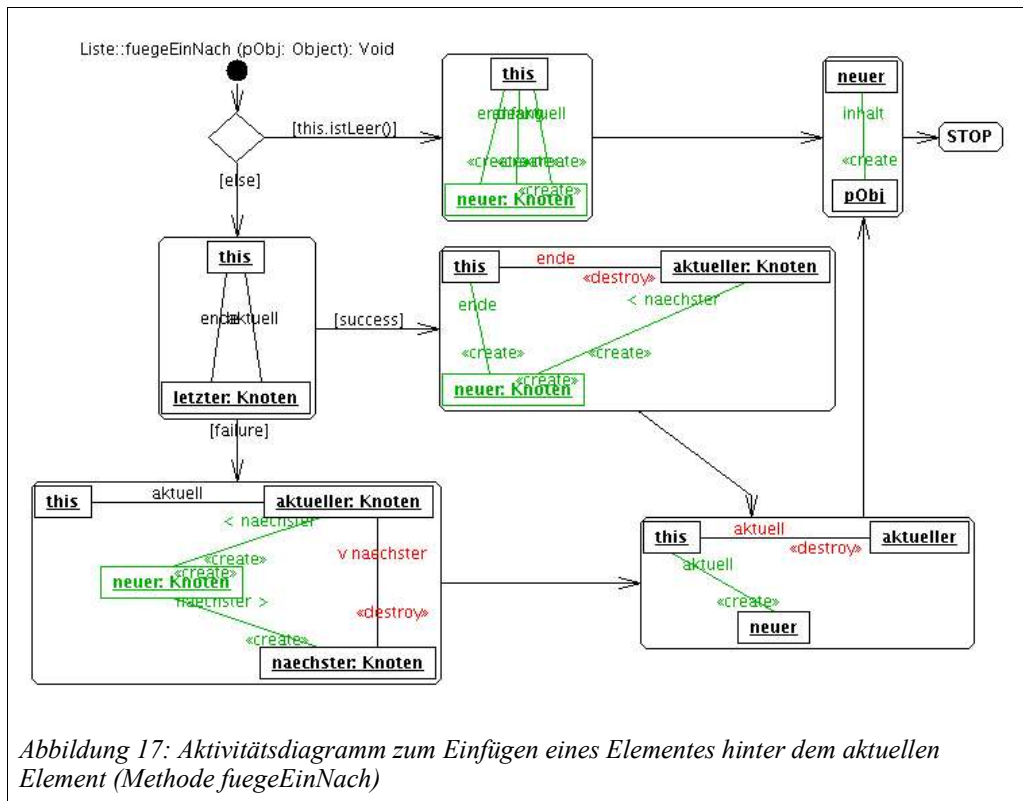


4.3.2 Aktivitätsdiagramme

Die Methoden für den ADT Liste sind teilweise schon recht umfangreich, da z.B. beim Einfügen oder Löschen eines Elementes mehrere verschiedene Fälle betrachtet werden müssen, die in den Aktivitätsdiagrammen entschieden und anschließend umgesetzt werden müssen.

Die Funktionsweise der Methode *fuegeEinNach* (Abbildung 17):

Zunächst wird geprüft, ob die Liste leer ist. Ist dies der Fall, wird die erste Zeile bis zum Stop abgearbeitet: Ein Knotenobjekt wird eingefügt und anschließend das übergebene Datenobjekt (pObj) daran 'angehängt'. Falls die Liste nicht leer war, prüft das SP unter dem NOP (über den else-Zweig), ob nur ein Element vorhanden ist. Bei Erfolg wird (über das SP nach success) der neue Knoten am Ende eingefügt, sonst (failure) hinter dem aktuellen und vor dem folgenden Knoten. Das SP unten rechts setzt die Beziehung aktuell auf den neuen Knoten. Anschließend wird über das gleich Story-Pattern wie auch im Fall der leeren Liste das Datenobjekt an den Knoten angehängt.



Die Funktionsweise der Methode *fuegeEinNach* (Abbildung 18):

Erste Zeile: Falls die Liste Leer ist, muss kein Element entfernt werden.

Die Story-Pattern (1) – (4) übernehmen jeweils einen speziellen Anwendungsfall:

(1) löscht, wenn nur ein Element in der Liste ist, (2) löscht, wenn das erste Element entfernt werden soll. (3) löscht das letzte Element und (4) wird ein Element im Inneren der Liste löschen. Das erste Element ist, falls überhaupt noch eines in der Liste enthalten ist, nach dem Löschvorgang stets das aktuelle Element (siehe Story-Pattern (2) und (5)).

4.4 Der ADT Binärbaum

4.4.1 Klassendiagramm

In Abbildung 19 ist das Klassendiagramm zum Binärbaum dargestellt. Hier gibt es im Gegensatz zur den drei linearen Datenstrukturen einige Besonderheiten: Es gibt nicht die Aufteilung in verwaltende Klasse (*Stack*, *Schlange*, *Liste*) und Knoten (-klasse). Wie bereits in Abschnitt 2.3.4 erwähnt, kann jeder Knoten als (Teil-) Baum angesehen werden. So übernimmt ein Knoten die Aufgabe zur Ordnung der Daten im Baum und stellt gleichzeitig die Methoden zur Verfügung. Über die Beziehungen *links* und *rechts* werden jeweils die linken bzw. rechten Teilbäume referenziert. Der dargestellte Binärbaum kann zur Verwaltung beliebiger Objekte verwendet werden, die die dargestellte Schnittstelle *Comparable* implementieren. Im Beispiel ist dies durch die Klasse *Patient* erfolgt, die durch die Methode *compareTo* eine Vergleichsfunktion anbieten muss.

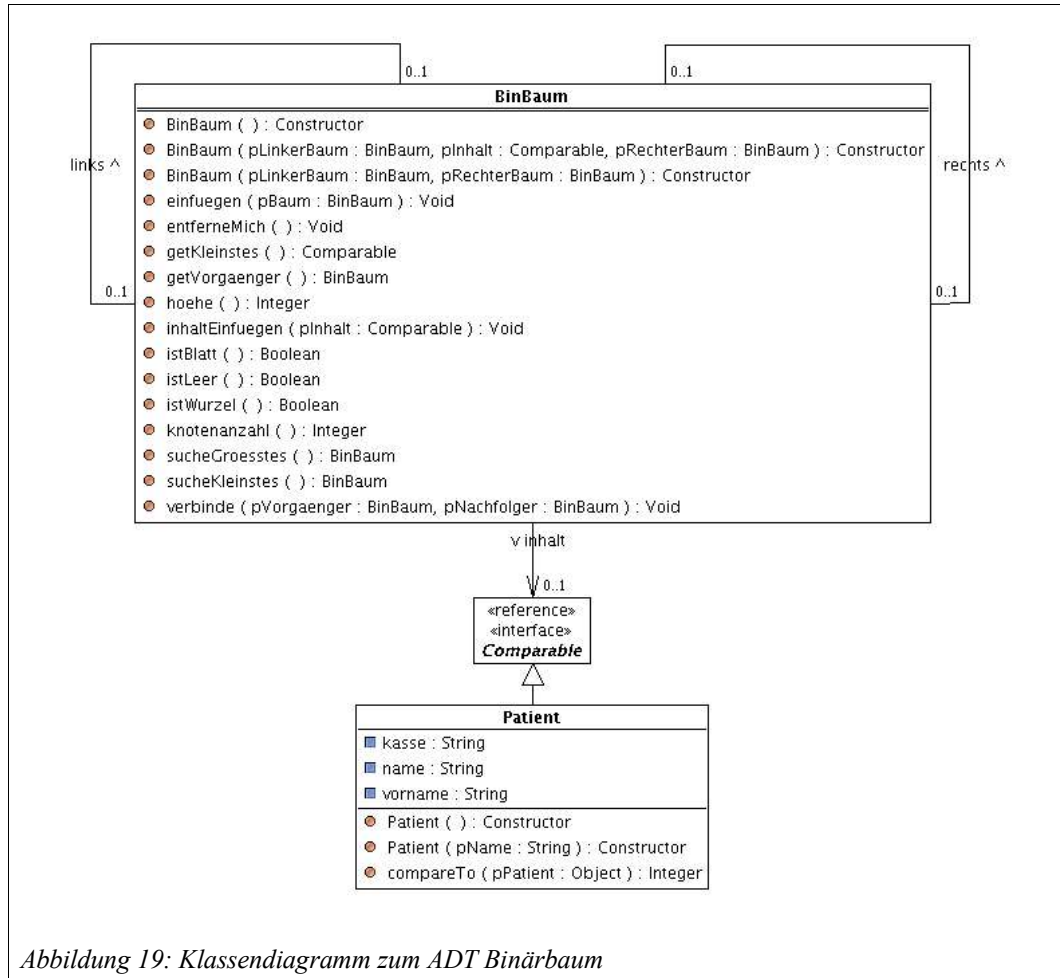


Abbildung 19: Klassendiagramm zum ADT Binärbaum

4.4.2 Aktivitätsdiagramme

Das Klassendiagramm zum Binärbaum ist recht einfach, die Implementierung der Methoden ist dagegen teilweise sehr aufwändig.

Exemplarisch sei die Methode *inhaltEinfuegen* in Abbildung 20 hier beschrieben, weil das sortierte Einfügen in den Binärbaum eine der zentralen Operationen ist, damit der binäre Baum auch stets eine korrekte Ordnung besitzt.

Stroy-Pattern (1) prüft, ob das aktuelle BinBaum-Objekt bereits ein Inhaltsobjekt referenziert. Falls nicht, wird (über failure) das übergebene Inhaltsobjekt *plnhalt* hier eingefügt und anschließend die Ausführung der Methode beendet.

Hat das aktuelle BinBaum-Objekt aber schon ein Inhaltsobjekt erhalten, muss mit Hilfe der Verzweigung (2) entschieden werden, ob der Inhalt links oder rechts vom aktuellen Baumknoten einsortiert werden muss.

Dazu wird in (3) auf dem linken Nachfolger durch ein Collaboration-Statement die

Methode rekursiv aufgerufen. Das heißt, dass nun ausgehend von diesem Nachfolger als Wurzel eines Baumes das Inhaltsobjekt eingefügt wird. Nach erfolgter Einfügung in einem der Rekursionsaufrufe würde diese Methode (über success) an dieser Stelle dann beendet.

Falls aber kein Nachfolger vorhanden war, wird auch das Collaboration-Statement nicht ausgeführt, so dass auch kein rekursiver Aufruf stattfindet. Dadurch ist aber klar, dass das übergebene Inhaltsobjekt hier als Blatt eingefügt werden muss, was in Story-Pattern (4) realisiert ist.

Das Einfügen im rechten Teilbaum erfolgt analog.

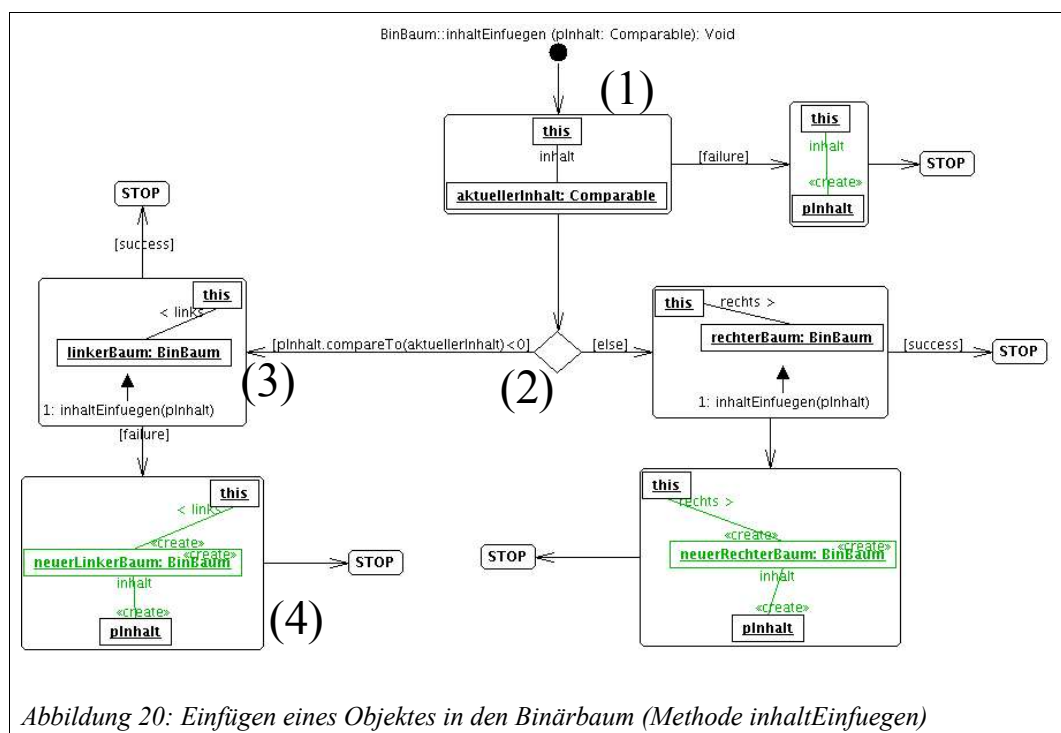


Abbildung 20: Einfügen eines Objektes in den Binärbaum (Methode *inhaltEinfuegen*)

Eine Traversierung des Baumes wird in der Methode *knotenanzahl* implementiert, wie sie z.B. auch für die Suche in einem Baum benötigt wird. Abbildung 21 zeigt das zugehörige Aktivitätsdiagramm.

Ist das BinBaum-Objekt *this* ein Blatt, so gibt es nur einen Knoten und die Methode kann bereits in der ersten Zeile verlassen werden. Andernfalls werden im Story-Pattern (1) zwei Variablen zur Verfügung gestellt, in denen die Anzahl der Knoten im rechten bzw. linken Teilbaum gespeichert werden. Durch die Story-Pattern (2) und (3) wird die Methode ähnlich wie beim Einfügen auf dem linken bzw. rechten

Teilbaum rekursiv aufgerufen, so dass jeder Knoten besucht wird und auch nur einmal gezählt wird.

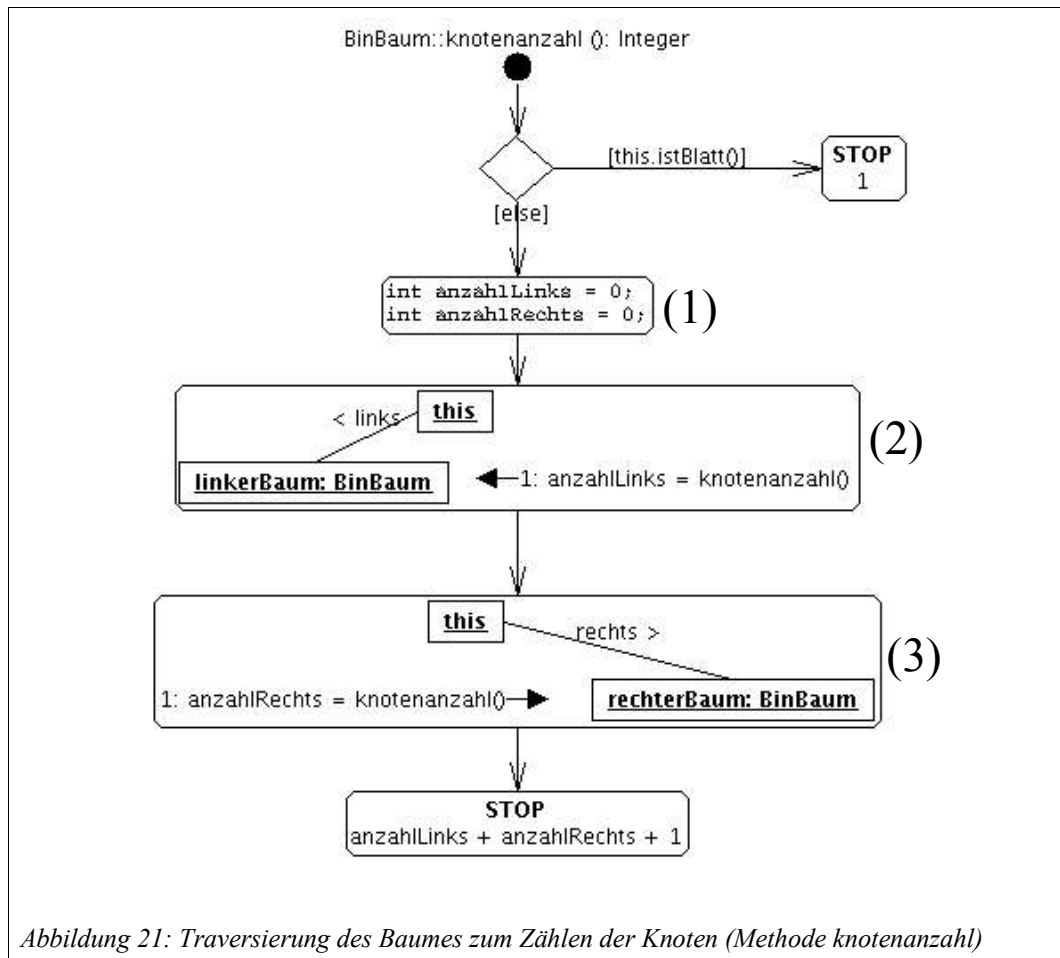


Abbildung 21: Traversierung des Baumes zum Zählen der Knoten (Methode `knotenanzahl()`)

5 ATDs im Informatikunterricht mit Fujaba

Grundsätzlich ergibt sich die Frage, ob eine Thematisierung, Modellierung und Implementierung von ADTs mit Hilfe von Fujaba unter fachdidaktischen Aspekten möglich und sinnvoll ist. Zur Beantwortung dieser Frage sollen nun einige im Lehrplan formulierte Aspekte betrachtet werden.

Gemäß dem Lehrplan sind den Schülerinnen und Schülern in der Jahrgangsstufe 11 die notwendigen Voraussetzungen zu vermitteln, um den Anforderungen der Qualifikationsphase (Jahrgangsstufe 12 und 13) zu entsprechen. Es wird ein curricular spiralförmiger Aufbau empfohlen. In der Jahrgangsstufe 12 werden die erworbenen Kenntnisse auf höherem Niveau und in komplexeren Zusammenhängen erweitert und vertieft. Der Unterricht sollte dabei stets handlungsorientiert sein. (Richtlinien, S. 24)

Eine fachmethodische Leitlinie im Bereich Modellieren und Konstruieren ist zunächst die Gewinnung von Informatikmodellen. Hierzu werden Probleme eingegrenzt und strukturiert, um anhand eines reduzierten Modelles einen Lösungsansatz zu entwerfen.

Im nächsten Schritt, der Abstraktion von Daten und Algorithmen, werden die zu lösenden Probleme in Teilprobleme zerlegt. „Die Lösung der Teilprobleme in Form von Modulen läuft auf die Verwendung oder Entwicklung geeigneter abstrakter Datentypen und der zugehörigen Verarbeitungsalgorithmen hinaus.“ (aaO., S. 12)

Zur Realisierung, Überprüfung und Weiterentwicklung von Lösungen macht man sich leistungsstarke Werkzeuge nutzbar, die allerdings nur in einer dem Problem angepassten Komplexität angewendet werden sollen. (aaO., S.13)

Diese sehr allgemein gehaltenen Gesichtspunkte sollen im Weiteren auf diese Frage hin konkretisiert werden.

Schwill erscheint das Unterrichtsprinzip der fundamentalen Ideen „gerade für den Informatikunterricht als besonders tragfähiges Konzept“ (Schwill 1993, S. 28). Das Prinzip der fundamentalen Ideen wurde 1960 von J.S. Bruner erstmals formuliert. Nach diesem fachneutralen didaktischen Prinzip soll sich der Unterricht „in erster Linie an den Strukturen (den sog. **fundamentalen Ideen**) der zugrundeliegenden Wissenschaft orientieren“ (aaO, S. 2). Bruner selbst hat den Begriff der

fundamentalen Iden nicht genau charakterisiert (aaO., S. 8), aber Schwill definiert ihn seinerseits:

„Definition“:

Eine **fundamentale Idee** (bezgl. einer Wissenschaft) ist ein Denk-, Handlungs-, Beschreibungs- oder Erklärungsschema, das

- (6) in verschiedenen Bereichen (der Wissenschaft) vielfältig anwendbar oder erkennbar ist (**Horizontalkriterium**),
- (7) auf jedem intellektuellen Niveau aufgezeigt und vermittelt werden kann (**Vertikalkriterium**),
- (8) in der historischen Entwicklung (der Wissenschaft) deutlich wahrnehmbar ist und längerfristig relevant bleibt (**Zeitkriterium**),
- (9) einen Bezug zu Sprache und Denken des Alltags und der Lebenswelt besitzt (**Sinnkriterium**).

Für die abstrakten Datentypen weist er entsprechend seiner Definition nach, dass sie zu den fundamentalen Ideen der Informatik zählen (aaO., S. 25). Somit ist auf dieser Basis eine Orientierung des Informatikunterrichts an dem Konzept der abstrakten Datentypen durchaus zu rechtfertigen. Die Frage ist nur noch, wie eine Umsetzung im Unterricht erfolgen kann oder sollte.

Für Hubwieser besitzt der Bereich der Modellierung eine „immense Bedeutung für die Allgemeinbildung“ und darf in keinem Informatikunterricht fehlen (Hubwieser 2004, S. 85). Weiter schlägt er vor, durch die Modellierung „eine *Hilfsebene* zwischen der Problem- und der Implementierungsebene“ (aaO., S. 90) zu erzeugen, so dass im Unterricht nicht auf programmiertechnische Details eingegangen werden muss. Im Unterricht kann somit der Modellierungsprozess in den Vordergrund rücken, die Programmierung sollte zurückweichen.

Leider verfügte man bis vor kurzem nicht über geeignete Techniken, um diesen Modellierungsvorgang im Unterricht systematisch und in angemessener Tiefe umsetzen zu können. [...] Inzwischen haben sich jedoch auf dem Gebiet der Softwareentwicklung Modellierungstechniken durchgesetzt, die aufgrund ihrer Anschaulichkeit und Beschreibungsmöglichkeit geeignet erscheinen, genau diese methodische Lücke zu schließen. Die Softwaretechnik verwendet inzwischen vor allem objektorientierte Entwurfsmethoden, die auf diesen Techniken aufsetzen. Dazu gehören die Entwicklungsmethoden von Rumbaugh et al. (1991), Booch (1994) und Jacobson et al. (1991), die inzwischen unter Beteiligung der drei Erfinder zur *Unified Modeling Technique* (UMT) verschmolzen sind und weiterentwickelt wurden (siehe Booch, Rumbaugh, Jacobson (1997)). Einige dieser relativ neuen Techniken ermöglichen eine durchaus altersgemäße Modellierung einfacher Sachverhalte und damit eine direkte Umsetzung unseres didaktischen Ansatzes. (Hubwieser 2004, S. 85f)

Mit der Bezeichnung „Unified Modeling Technique“ ist hier vermutlich die Unified

Modeling Language gemeint. Die Methode von Rumbaugh wurde früher mit Object Modeling Technique (OMT) bezeichnet, das Ergebnis der Zusammenführung mit der Methode von Booch wurde zunächst Unified Method (UM) genannt und später dann in Unified Modeling Language (UML) umbenannt (Oesterreich 2004, S. 18).

Eine Verwendung der UML und der damit verbundenen Modellierungstechniken als Hilfsmittel für die Modellierung scheint mir somit für den Informatikunterricht legitim und passend.

Zusammen mit dem in den Kapiteln 2 und 3 dargelegten Eigenschaften und Möglichkeiten ist meines Erachtens ein Unterrichtsansatz zur Modellierung von abstrakten Datentypen mit Hilfe der UML interessant.

In Abschnitt 3.2 habe ich bereits die Vorzüge der Quelltext-Generierung bei der Verwendung von CASE-Tools im Unterricht dargelegt. Fujaba nimmt gegenwärtig unter den CASE-Tools nach meinem Kenntnisstand eine besondere Stellung ein. Diese ergibt sich aus der Gesamtheit der folgenden Punkte:

- Aus UML-Klassendiagrammen wird automatisch Quelltext erstellt.
- Methoden können auf grafischer Ebene durch Aktivitätsdiagramme beschrieben werden, der Quelltext wird ebenfalls automatisch erzeugt.
- Dobs ermöglicht die Darstellung und Veränderung der erzeugten Objektstrukturen zur Laufzeit.
- Die life³-Version bietet aus didaktischen Gründen eine für den Unterricht reduzierte Funktionalität (durch die Begrenzung auf Klassen- und Aktivitätsdiagramme), die somit sach- und adressatengemäß ist.
- Es ist kostenlos erhältlich.
- Es ist plattformunabhängig verwendbar, da es in Java implementiert ist.

Auch unter der von Moll (2002, S.48) erarbeiteten Kriterienliste zur Auswahl von CASE-Tools für den Unterricht kann der Einsatz von Fujaba gerechtfertigt werden:

- Die Sach- und Adressatengemäßheit ist gegeben.
- Die Oberfläche und Funktionalität sind zur Verwendung im Unterricht angepasst.
- Die Verwendung von Begriffen und Bezeichnungen aus der objektorientierten

Softwareentwicklung ist gegeben.

- Die Darstellung der Diagramme ist übersichtlich, zoombar und eingrenzbar.
- Der automatisch erzeugte Quellcode ist direkt ausführbar.

Die Dokumentationsmöglichkeit in Fujaba ist eingeschränkt. Kommentare können lediglich zu Methodenköpfen aufgenommen werden. Eine detaillierte Dokumentation ist aber wegen der durchgängig grafischen Darstellung auch nicht immer notwendig. Somit schätze ich diesen Kritikpunkt als nicht so schwer wiegend ein.

Zusammenfassend stellt sich mir eine Modellierung von ADTs mit anschließender Implementierung in Fujaba im Informatikunterricht gerechtfertigt und sinnvoll dar.

6 Die Umsetzung im Unterricht

6.1 Die Unterrichtsreihe im Überblick

Die hier dargestellte Unterrichtsreihe stellt einen möglichen Unterrichtsverlauf zur Behandlung von ADTs im Informatikunterricht der Sekundarstufe II dar. Der Verlauf ist skizzenhaft zu verstehen: Es werden verschiedene Methoden für den Unterricht exemplarisch dargestellt und Alternativen aufgezeigt. Es wird hier keine konkrete Planung für eine oder mehrere Unterrichtseinheiten mit entsprechendem Verlaufsplan durchgeführt.

Die hier vorgestellte Unterrichtsreihe ist für eine Lerngruppe in der Jahrgangsstufe 12 an Gymnasien konzipiert. Im Anfangsunterricht der Jahrgangsstufe 11 soll bereits mit Fujaba gearbeitet worden sein, sodass an dieser Stelle keine Einarbeitung in UML oder Fujaba mehr nötig ist. Die Beherrschung grundlegender Modellierungstechniken wird ebenfalls vorausgesetzt.

Das Vorgehen der Unterrichtsreihe greift die Idee aus dem Projekt „Wartezimmer“ auf, das auf dem NRW-Bildungsserver [learnline](#) (learnline-Webseite) zu finden ist: In einer Arztpraxis gibt es ein Wartezimmer, in dem Patienten warten. Anhand dieses Beispiels können verschiedenen Datenstrukturen entwickelt werden.

Die Arztpraxis bietet hier einen Kontext für den informatischen Inhalt. Die Unterrichtsreihe folgt somit dem Ansatz „Lernen im Kontext der Anwendung“, der gemäß dem Lehrplan für den Erwerb einer Informatik-Kompetenz erforderlich ist (Richtlinien, S. 17).

Methodisch habe ich mich an einem von Carsten Schulte entworfenen Konzept zur Veranschaulichung abstrakter Datentypen in Fujaba orientiert. Die dort eingesetzten Verfahren und die dazu gemachten Erfahrungen habe ich hier an einigen Stellen verwendet.

Die Unterrichtsreihe gliedert sich in 6 Phasen:

Phase	Kurzbeschreibung der Phase	Kurzbeschreibung der Ziele
1	Die einfache Nachfolgeverkettung	Erste Idee von (Warteschlangen), einfache Nachfolgeverkettung
2	Die Verkettung durch Knotenobjekte	Verkettung durch Knotenobjekte, um das Einfügen eines Elementes in mehrere Warteschlangen zu ermöglichen
3	Der ADT Schlange	Die Schlange soll beliebige Objekte aufnehmen können
4	Der ADT Liste	Datenstruktur, die das vorzeitige Verlassen des Wartezimmers und die Behandlung von Notfällen ermöglicht
5	Der ADT Stack	Analyse des ADT Stack, Eigenschaften, Anwendungen
6	Der ADT Binärbaum	Analyse der Baumstruktur, einfache Implementierung

6.2 Die verwendeten Methoden im Überblick

An dieser Stelle erfolgt eine kurze Beschreibung der im Unterricht mit Fujaba häufiger verwendeten Methoden.

6.2.1 Das Objektspiel

Das Objektspiel dient der Überprüfung von Klassendefinitionen oder CRC-Karten. CRC-Karten (Class-Responsibilities-Collaborators) sind Karteikarten, auf denen der Name, die Verantwortlichkeiten in Form von Wissen und Fähigkeiten und die bekannten „Mitarbeiter“ vermerkt sind. Die Verantwortlichkeiten werden weitgehend durch kurze Beschreibungen festgelegt und nicht formal.

Basierend auf den CRC-Karten oder den bereits vorliegenden Klassendiagrammen werden nun den Schülerinnen und Schülern Rollen zugeteilt. In diesen Rollen sind sie auf die vorgegebenen Verantwortlichkeiten und Mitarbeiter beschränkt. Sie erhalten sog. Objektkarten, auf denen sie ihr aktuelles Wissen festhalten müssen, da

zwar die Schülerin oder der Schüler natürlich immer den gesamten Kontext betrachten kann, das von ihnen verkörperte Objekt allerdings nicht.

Nun wird durch die Schülerinnen und Schüler ein Szenario durchgeführt. Hierbei ist stets genauestens darauf zu achten, dass die Rollen auch strikt eingehalten werden. Während des Objektspieles können immer wieder Unklarheiten oder Fehler aufgedeckt werden, die sich aus den vorgegebenen (Rollen-)Definitionen ergeben. Z. B. kann aufgrund fehlenden Wissens oder unzureichender Fähigkeiten ein Objekt seine Aufgaben nicht erfüllen. So können die Klassen bzw. CRC-Karten überprüft und gegebenenfalls angepasst werden.

Wenn die Lerngruppe das Objektspiel bereits gut beherrscht, kann es passieren, dass die einzelnen Vorgänge so schnell durchgeführt werden, dass der Überblick verloren geht und wichtige Details übersehen werden, die einen Fehler darstellen. Somit ist immer genau aufzupassen, ob die Rollen wirklich von den Schülerinnen und Schülern eingehalten werden.

Die entstandenen Objektstrukturen können zur besseren Übersicht auch an der Tafel oder auf Folie festgehalten werden, sodass die Zusammenhänge besser klar gemacht werden können.

6.2.2 Der Zetteltest

Der Zetteltest (Schulte 2004, S. 15ff) dient dem Testen von Aktivitätsdiagrammen, indem der Ablauf schrittweise an einem durch ein Szenario gegebenes Objektdiagramm durchgeführt wird. Dazu wird an dem vorgegebenen Objektdiagramm mit Hilfe des Aktivitätsdiagrammes gearbeitet:

Ausgehend von einem konkreten Methodenaufruf werden in der Objektstruktur die gebundenen Objekte ermittelt. Damit klar ersichtlich ist, welche Objekte bereits gebunden sind, werden diese Objekte mit Karten abgedeckt, die den im Aktivitätsdiagramm verwendeten Bezeichner tragen. Anschließend wird überprüft, ob die im Aktivitätsdiagramm vorgegebene Objektstruktur so wiedergefunden werden kann, um die benötigten Objekte ebenfalls zu binden. Ist dieser Schritt erfolgreich durchgeführt, werden die angegebenen Veränderungen in der Objektstruktur eingetragen.

So kann mit dem Zetteltest einerseits die Implementierung geprüft werden, andererseits aber auch das Verhalten von Aktivitätsdiagrammen und Story-Pattern verdeutlicht werden.

6.2.3 Story-Driven-Modelling

Schulte verwendet ein vereinfachtes Verfahren des Story-Driven-Modelling. Es basiert auf einem ausführlicheren Verfahren (Diethelm, Geiger, Zündorf 2004) zur Implementierung von Methoden in Fujaba.

Beim (vereinfachten) SDM wird folgendermaßen vorgegangen:

- 1) Fälle stichwortartig sammeln
- 2) Story-Boarding für unbekannte Fälle
 - 2a) eventuell Klassendiagramm ergänzen
 - 2b) eventuell Zetteltest
- 3) Zusammenführen der Fälle zu einem Aktivitätsdiagramm
- 4) Zetteltest

Hier wird also ein Problem zunächst in seine Teilprobleme zerlegt (1), die anschließend einzeln mit Hilfe des Story-Boarding (2) gelöst werden. Das hier verwendete Story-Boarding ist als Erweiterung zum bereits beschriebenen Objektspiel anzusehen. Bei der Durchführung werden die Objekte und jegliche Veränderungen der Objektstruktur grafisch an der Tafel oder auf Folie festgehalten. Dies ermöglicht eine bessere Anschaulichkeit der Objektstruktur. Wurde der Fall komplett durchgespielt, wird das zu diesem Fall gehörige Aktivitätsdiagramm erstellt. Nachdem die Diagramme für alle Fälle erfolgreich erarbeitet wurden, müssen sie in eine einzige Methode zusammengeführt werden (3). Abschließend sollte noch einmal ein Test erfolgen, ob die einzelnen Fall-Diagramme richtig kombiniert wurden (4).

In Abschnitt 6.3.4 werde ich das Vorgehen dieses Verfahrens exemplarisch anwenden.

6.3 Die einzelnen Phasen der Unterrichtsreihe

6.3.1 Phase 1: Die einfache Nachfolgeverkettung

Die Schülerinnen und Schüler entwickeln anhand einiger Beispiel-Situationen eine erste Idee von Warteschlangen. Anschließend sollen die Klassen Wartezimmer und Patient erarbeitet werden. In dieser Phase soll noch eine einfache Nachfolgeverkettung der Patientenobjekte verwendet werden, bei der ein Patientenobjekt das Nachfolgeobjekt kennt.

Hierzu analysieren die Schülerinnen und Schüler in Partner- oder Gruppenarbeit einige Situationen (Abbildung 22), in denen (Warte-)Schlangen auftreten und stellen die wesentlichen Gemeinsamkeiten zur Entwicklung der Idee einer (Warte-)Schlange heraus.

Untersuchen Sie die folgenden Situationen. Stellen Sie die Gemeinsamkeiten heraus.

- (1) Mehrere Personen wollen an der Supermarktkasse bezahlen
- (2) Zu Stundenschluss wollen alle Schüler ihr Dokument ausdrucken
- (3) Das Parkhaus ist voll. Die wartenden Autos werden nach und nach eingelassen
- (4) Der Andrang am Schulkiosk ist groß, es gibt aber nur eine Bedienung
- (5) In der Arztpraxis warten die Patienten im Wartezimmer

Abbildung 22: Aufgabenstellung zum Problem Warteschlange

In einem anschließenden Unterrichtsgespräch werden die Ergebnisse zusammengetragen und die abstrahierten Merkmale einer Warteschlange festgehalten (siehe Abbildung 23).

Es kann vorkommen, dass der vierte Punkt nicht von den Schülerinnen und Schülern genannt wird, da nicht explizit zwischen Bedienen und Verlassen der Warteschlange unterschieden wird. In diesem Fall sollte die Lehrperson diesen Punkt motivieren. Bei der Umsetzung von Warteschlangen übernimmt häufig die Methode *ausschlangen* auch das Auslesen des ersten Elementes.

Abstrakter Datentyp: Warteschlange

- Es bildet sich eine Warteschlange
- Die Reihenfolge muss eingehalten werden
- Neue müssen sich hinten anstellen → einschlangen
- Der Erste ist an der Reihe → kopf
- Der Erste verlässt die Warteschlange → ausschlangen
- Die Anzahl der Wartenden ist nicht bekannt

Abbildung 23: Mögliches Ergebnis des Unterrichtsgesprächs

Punkt sechs kann auch fehlen, da manchmal die Warteschlangen die Anzahl der enthaltenen Elemente festhalten. Da aber die Probleme der Aufgabenstellung diesen Aspekt zulassen, wurde er hier auch aufgeführt.

Die Bezeichnung der Methoden muss nicht während der Sammlung der Ergebnisse stattfinden, dies wird vermutlich anschließend erst erfolgen. Wenn das Tafel-/Folienbild komplettiert wurde, können die Punkte, die eine Methode benötigen, durch einen Methodennamen ergänzt werden.

Nachdem die Merkmale einer Warteschlange erarbeitet wurden, soll nun anhand einer Beispielaufgabe eine Warteschlange modelliert werden.

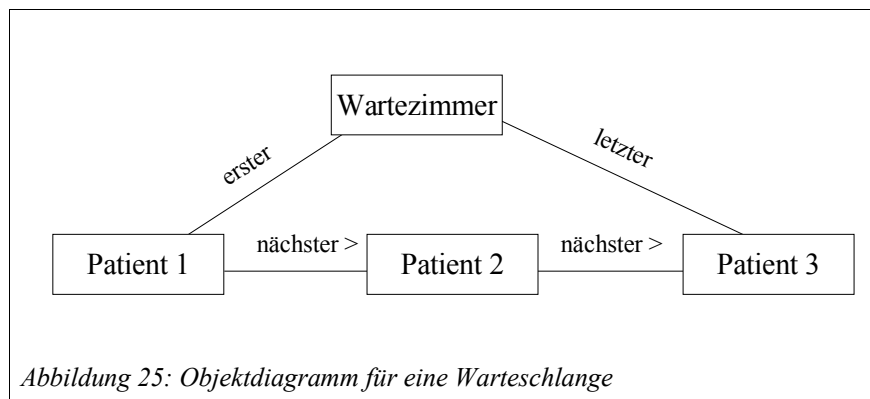
Aufgabenstellung:

Die Daten der Patienten im Wartezimmer einer Arztpraxis sollen von der EDV verwaltet werden. Neue Patienten werden in die Warteschlange aufgenommen, die Patienten gehen der Reihe nach zur Behandlung zum Arzt.

Wie kann diese Situation objektorientiert modelliert werden?

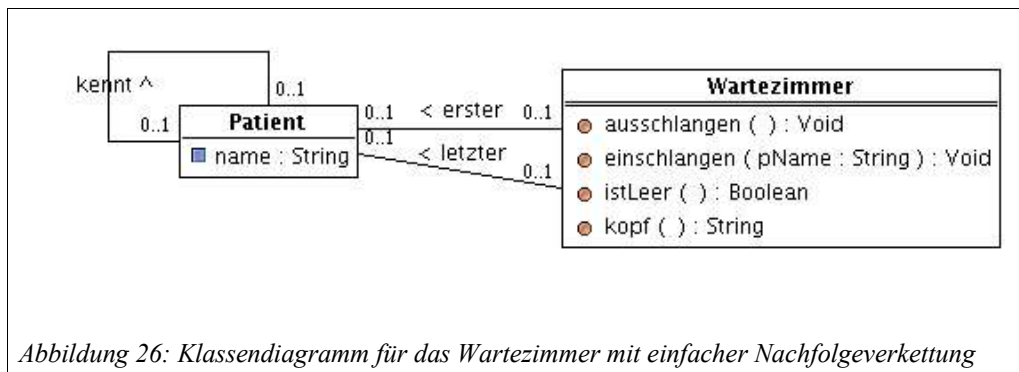
Abbildung 24: Aufgabenstellung zum Problem Warteschlange

Durch Ideen und Anregungen der Schülerinnen und Schüler wird im Unterrichtsgespräch nun eine anschauliche Darstellung einer Warteschlange an der Tafel oder auf Folie erarbeitet (siehe Abbildung 25).



Hierdurch soll die Übertragung in ein Klassendiagramm erleichtert werden, das jetzt aus dieser Objektstruktur erstellt werden soll. Es empfiehlt sich, die Objektdiagramme auf Folie zu zeichnen, da sie dann in zukünftigen Stunden (z. B. auch in Phase 2) schnell zur Hand sind, ohne dass wieder neue Diagramme an die Tafel gezeichnet werden müssen.

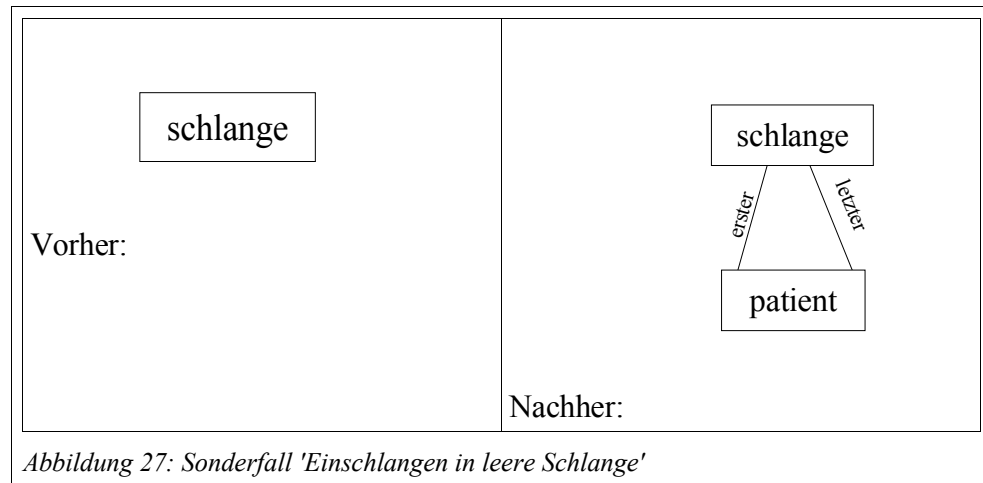
Bei der Übertragung in ein Klassendiagramm müssen die Parameter und Rückgabewerte der Methoden noch festgelegt werden. Ein mögliches Klassendiagramm ist in Abbildung 26 zu sehen.



An dem bereits erstellten Objektdiagramm (Abbildung 25) kann der allgemeine Fall beim Einschlangen anschaulich an der Folie (bzw. an der Tafel) dargestellt werden. Dies sollte anhand eines Objektspieler geschehen. Die nicht mehr benötigten Kanten werden weggewischt, die neuen mit farbiger Kreide eingezeichnet.

Der nächste Schritt besteht darin, den Sonderfall beim Einfügen eines Elementes in eine leere Schlange zu untersuchen. Zur Verdeutlichung sollte dieser Sonderfall

nicht nur verbal beschrieben werden, sondern auch grafisch durch ein Objektdiagramm an der Tafel oder auf Folie visualisiert werden (siehe Abbildung 27).



Da beim Einschlangen nur die zwei Fälle 'Einschlangen in eine bestehende Liste' und 'Einschlangen in eine leere Schlange' betrachtet werden müssen, kann die Methode `einschlangen` implementiert werden. Mit den vorliegenden Objektdiagrammen der zwei möglichen Fälle sollte es den Schülerinnen und Schülern möglich sein, im Unterrichtsgespräch eine Übertragung in ein Aktivitätsdiagramm in Fujaba zu bewerkstelligen.

Dies könnte aufgrund der Einfachheit der Methode an der Tafel erfolgen. Hierbei ist allerdings auf die exakte Darstellung der einzelnen Symbole und der Beschriftungen zu achten, da sonst das „Abtippen“ durch die Schülerinnen und Schüler erschwert werden würde. Um diese Darstellungsfehler zu vermeiden, ist es daher ratsam, das Aktivitätsdiagramm direkt in Fujaba zu erzeugen und über einen Beamer zu präsentieren. So werden Fehler und damit verbundene Diskussionen vermieden.

Noch vor dem „Abtippen“ durch die Schülerinnen und Schüler sollte allerdings der Ablauf der Methode mit dem Zetteltest analysiert werden. So kann sichergestellt werden, dass jeder die Funktionsweise verstanden hat.

Wenn die Schülerinnen und Schüler die Methode `einschlangen` (siehe

Abbildung 28) in Fujaba übertragen und die Funktionalität in Dobs überprüft haben, schließt sich die Implementierung der Methode ausschlangen an. Dies sollte aufgrund der ausführlichen Erarbeitung der Methode einschlangen von den Schülerinnen und Schülern in Partner- oder Gruppenarbeit erfolgen, um den Grad des selbstständigen Arbeitens zu erhöhen.

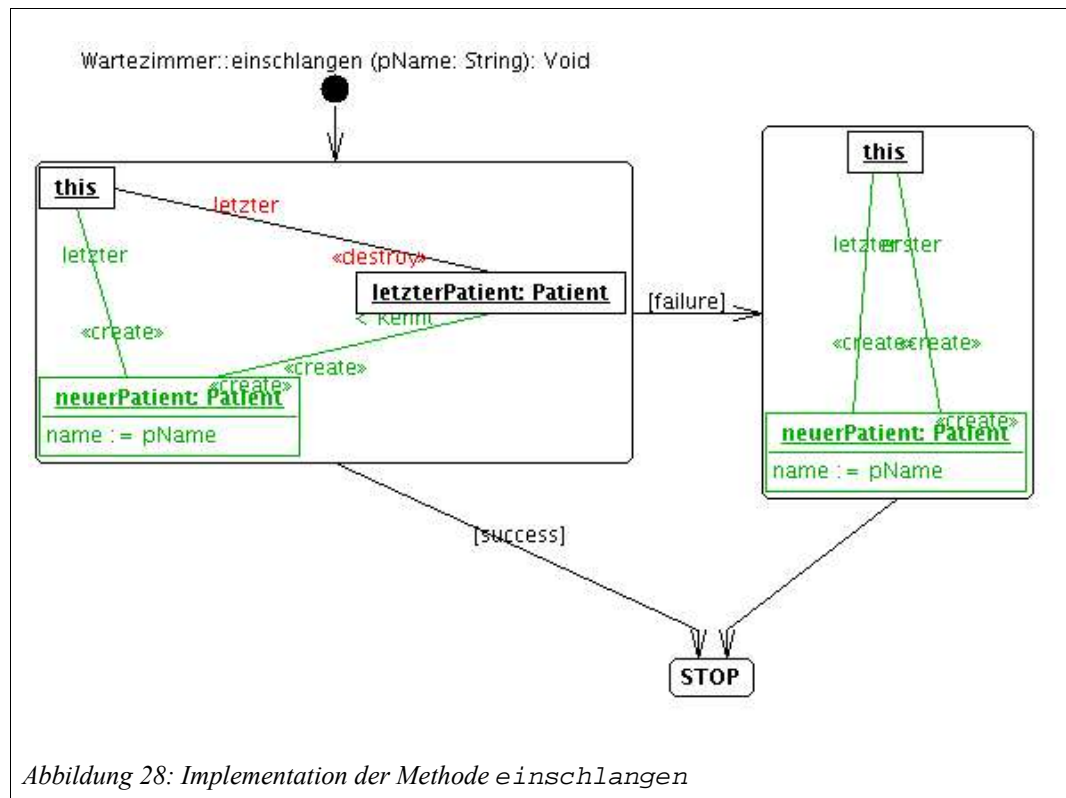


Abbildung 28: Implementation der Methode `einschlangen`

Beim Entfernen eines Elementes müssen insgesamt drei Fälle betrachtet werden:

- Entfernen eines Elementes aus einer vorhandenen Liste mit mindestens zwei Elementen
- Entfernen eines Elementes aus einer vorhandenen Liste mit nur einem Element
- Entfernen eines Elementes aus einer leeren Schlange

Die Lerngruppe sollte darauf hingewiesen werden, alle Fälle (selbstständig) zu finden und diese genau zu analysieren. Ebenfalls sollen Objektdiagramme für alle

Fälle gezeichnet werden, um daraus wieder das Aktivitätsdiagramm ableiten zu können. Dieses Finden und Analysieren kann auch gut von den Schülerinnen und Schülern als Hausaufgabe erledigt werden. Die Ableitung in ein Aktivitätsdiagramm erfolgt dann aber nach einer Besprechung der Objektdiagramme im Unterricht wieder in Partner- oder Gruppenarbeit.

Da die Storypattern in Fujaba zunächst die angegebene Objektstruktur überprüfen, kann die Implementation der Methode `ausschlangen` ganz ähnlich zu der von `einschlangen` erfolgen. Je nach Reihenfolge der Betrachtung der drei möglichen Fälle variiert aber der Grad der Schwierigkeit. In der oben angegebenen Reihenfolge ist das Aktivitätsdiagramm dem von `einschlangen` sehr ähnlich, denn es wird nur noch ein zusätzliches STOP benötigt, das aufgerufen wird, wenn kein Element in der Schlange vorhanden ist. Ebenso gut könnte aber auch zuerst der Fall 3, das Entfernen eines Elementes aus einer leeren Schlange, betrachtet werden. Hier muss dann ein Storypattern eingefügt werden, das prüft, ob dieser Fall vorliegt. Alternativ kann dieser Fall auch durch eine zusätzliche Methode `istLeer` getestet werden, die aber leicht zu implementieren ist. Abbildung 30 zeigt eine mögliche Implementierung von `istLeer`.

Es ist daher sehr produktiv – gerade auch im Hinblick auf die komplexeren Methoden für Listen und Binärbäume – die Schülerinnen und Schüler darauf hinzuweisen, die Auswirkungen bei unterschiedlicher Reihenfolge der einzelnen Fälle auf die Aktivitätsdiagramme zu prüfen. Eine solche Sensibilisierung sollte allerdings auch schon im Anfangsunterricht erfolgen, damit ein vorausschauendes Programmieren gefördert wird.

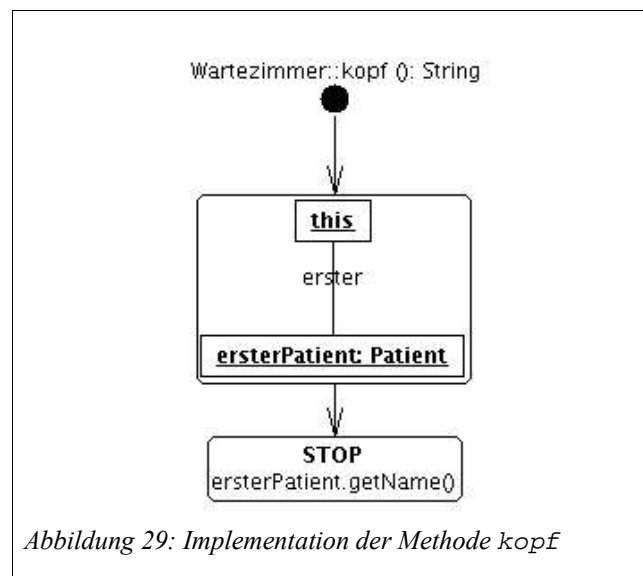
Zur Komplettierung der Warteschlange muss noch die Methode `kopf` zum Lesen des ersten Elementes implementiert werden.

Hier sind zwei Szenarien zu betrachten:

1. es existiert ein erstes Element -> Rückgabe einer Referenz auf dieses Patientenobjekt
2. die Schlange ist leer -> es wird 'null' (kein Objekt) zurückgegeben

Die Implementierung von `kopf` (siehe Abbildung 29) sollte den Schülerinnen und

Schülern keine Probleme bereiten, sodass sie die Methode auch in einer Hausaufgabe erarbeiten können. In diesem Fall würde ein Aktivitätsdiagramm auf Papier ausreichen, damit in den Gruppen in der darauffolgenden Stunde alle Aktivitätsdiagramme nebeneinander gelegt und verglichen werden können. Unterschiede werden diskutiert und anschließend kann das Ergebnis in Fujaba eingegeben werden. Bei diesen Diskussionen sollte die Lehrperson sich die erarbeiteten Lösungen ansehen und bei Unklarheiten Hilfestellung leisten. Natürlich kann es auch vorkommen, dass einzelne Gruppen durch nicht gemachte

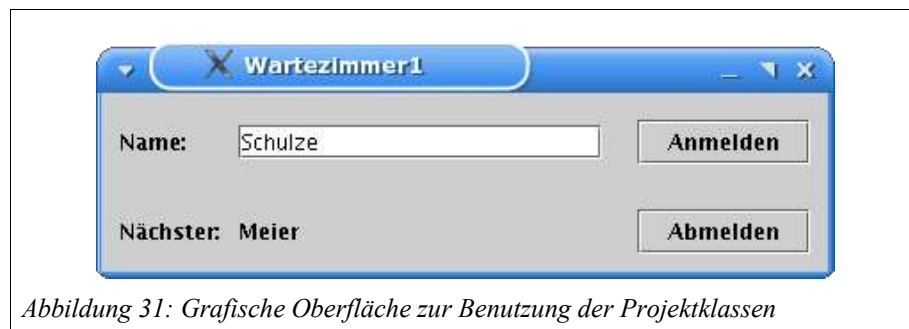
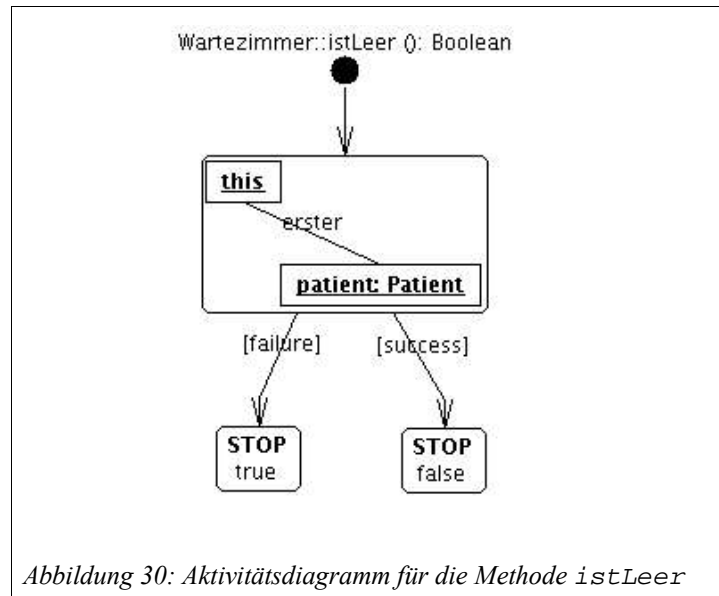


Hausaufgaben oder Scheitern keine Implementierung vorliegen haben. Falls dies bei nur einer Gruppe zutrifft, kann sicherlich mit kurzer Hilfe eine eigenständige Implementierung durch die Gruppe erfolgen. Bei mehreren Gruppen ist jedoch eine Besprechung im Plenum vorzuziehen, da sonst doch zu viel Zeit für die Hilfe aufgebraucht würde.

Zur Ergänzung kann nun noch die Methode *istLeer* durch die Lerngruppe erarbeitet werden, sofern dies nicht bereits geschehen ist. Die Methode sollte abschließend in Dobs überprüft werden.

Um den Schülerinnen und Schülern einen ersten Einblick in die Vorteile von Abstraktion zu geben, kann an dieser Stelle eine einfache grafische Oberfläche (siehe Abbildung 31) zur Verfügung gestellt werden, die die erstellten Klassen Warteschlange und Patient verwendet, um die in der Aufgabenstellung geforderte

Verwaltung der Patienten im Wartezimmer zu übernehmen.



Eine solche Oberfläche kann zum Beispiel mit Hilfe von NetBeans³ oder Eclipse⁴ erstellt werden.

Neben den selbst in Fujaba erstellten Klassen muss stets die Datei `fujaba-runtime.jar` vom Java-Interpreter gefunden werden können, damit ein Ausführen möglich ist.

Damit eine solche grafische Oberfläche die zur Verwaltung der Schlange bereitgestellten Klassen erfolgreich benutzen kann, ist bereits bei der Implementierung darauf zu achten, dass die Schnittstelle für den Benutzer korrekt implementiert wird: Klassen, Methoden und Verknüpfungen müssen genau so benannt werden, wie es in der Spezifikation festgelegt wurde. Im Unterricht wurde hier zwar nicht zu Beginn eine komplette Spezifikation der Schnittstelle festgelegt,

³ NetBeans IDE ist ein Open-Source-Produkt, erhältlich unter <http://www.netbeans.org>

⁴ Eclipse ist ein Open-Source-Produkt, erhältlich unter <http://www.eclipse.org>

aber nach und nach wurden die Bezeichner gewählt. Den Schülerinnen und Schülern sollte somit bereits vor dem Ableiten des Klassendiagramms aus den Objektdiagrammen klar gesagt werden, dass die Bezeichner so übernommen werden müssen und keine Veränderungen vorgenommen werden dürfen. Die Schülerinnen und Schüler sollen besonders auf die korrekte Groß- und Kleinschreibung achten.

So kann die Wiederverwendbarkeit von ADTs bzw. allgemeiner die Wiederverwendbarkeit durch Abstraktion thematisiert werden: Eine Oberfläche benutzt verschiedene Implementationen der Warteschlange. Hilfreich ist es hier, wenn (mindestens) zwei lauffähige Warteziimmerschlangen vorliegen, die sich in der Implementation zumindest einer der selbst erstellten Methoden unterscheiden (z. B. in der angesprochenen Reihenfolge der Fallbetrachtung). So kann die Lerngruppe sehen, dass nicht wichtig ist, wie die Methoden funktionieren, sondern, was sie tun.

Der Anwender – hier also der Programmierer der grafischen Oberfläche – kann von der Implementation der Schlange absehen, er kann abstrahieren.

Falls keine unterschiedlichen Lösungen vorliegen, kann die Lehrperson eine geeignete Version präsentieren, um die Unabhängigkeit von der internen Realisierung zu demonstrieren.

6.3.2 Phase 2: Die Verkettung durch Knotenobjekte

In dieser Phase sollen die Schülerinnen und Schüler eine Klasse Schlange entwickeln, die zur Verkettung Knotenobjekte nutzt. Diese Knoten referenzieren Patientenobjekte, die den eigentlichen Inhalt der Schlange darstellen. Die Knotenobjekte dienen lediglich der Verwaltung der Reihenfolge und der jeweiligen Inhaltsobjekte (hier noch vom Typ Patient). Die Klasse Knoten muss ebenfalls implementiert werden. Sie besitzt aber lediglich eine Beziehung zu sich selbst und zur Klasse Patient.

Die Schülerinnen und Schüler sollen erkennen, dass die Verwendung eines Inhaltsobjektes in mehreren Schlangen möglich ist und nicht für jede Schlange ein neues Objekt (mit gleichem Informationswert) erzeugt werden muss. Redundanzen werden so vermieden.

Zum Ende dieser Phase sollen die Unterschiede zu der Verkettung in Phase 1 herausgestellt werden.

Teilweise müssen die Patienten – vor allem ältere – noch vor der Behandlung durch den Arzt zum Blutdruckmessen ins Labor. Diese Patienten sollen wie alle im Wartezimmer in der richtigen Reihenfolge auf die Behandlung warten, gleichzeitig aber auch noch in der richtigen Reihenfolge zur Blutdruckmessung.

Abbildung 32: Aufgabenstellung in Phase 2

Zunächst müssen die Schülerinnen und Schüler überhaupt die Problematik erkennen. Dabei werden zwei Fragen zentrale Bedeutung haben:

1. Warum soll nicht für jede Abteilung eine eigene Warteschlange mit eigenen Patientenobjekten erstellt und verwaltet werden?
2. Warum kann nicht ein Patient in der Schlangenstruktur aus Phase 1 in zwei oder mehr Schlangen enthalten sein?

Frage 1 ist praktischer Natur: Ein Patient, der in mehrere Warteschlangen (hier: Arztbesuch und Blutdruckmessung) aufgenommen werden müsste, würde bei völlig eigenständigen Warteschlangen auch mehrfach als Objekt im System angelegt. Dadurch wird einerseits mehr Speicher verbraucht, andererseits müssten eventuell die Daten dann auch mehrfach eingegeben werden, obwohl immer der gleiche Datensatz vorliegt. Dies würde zu Redundanzen führen. Dadurch kann es weiter zu Inkonsistenzen führen, wenn sich die Daten ändern oder zusätzliche Daten aufgenommen werden sollen (z. B. die Werte für Puls und Blutdruck). Somit ist es von Vorteil, pro Patient im Wartezimmer nur ein Objekt anzulegen, dieses aber in mehreren Warteschlangen zu nutzen.

Frage 2 zielt auf die interne Verkettung mit einem Nachfolger ab: Ein Objekt der

Klasse Patient aus Phase 1 kennt seinen Nachfolger. In der Struktur war für ein Patientenobjekt genau ein Nachfolger berücksichtigt. Wartet ein Patient nun sowohl auf die Blutdruckmessung, den Arztbesuch und evtl. noch auf eine Blutentnahme, müsste das Patientenobjekt z. B. zwei oder drei Nachfolger kennen. Dies ist aber nicht möglich.

Die Problematik sollte in einem Unterrichtsgespräch diskutiert und erste Lösungsansätze erarbeitet werden

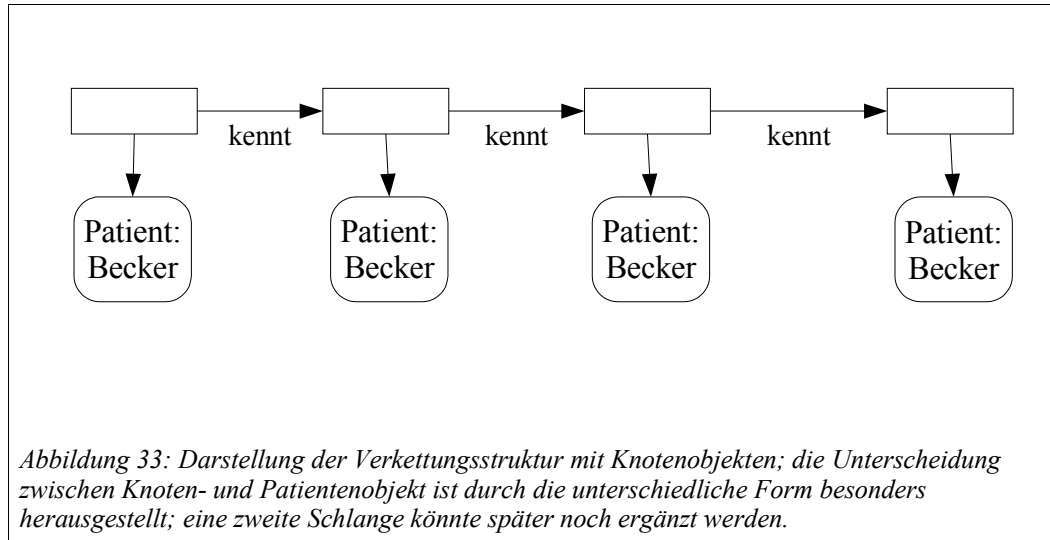
Da die Einführung von Knotenobjekten zur Verwaltung der Verkettungsstruktur von den Schülerinnen und Schülern aller Voraussicht nach noch nicht eigenständig bewerkstelligt werden kann, sollte dies durch die Lehrperson mit Hilfe einer grafischen Darstellung motiviert werden.

Hierzu werden im folgenden verschiedene Möglichkeiten vorgestellt, die bei Bedarf auch gut kombiniert werden können:

1. Anschauliche (statische) Darstellung der Verkettungsstruktur
2. Erarbeitung an der Tafel bzw. auf Folie aus den bisherigen Strukturen
3. Darstellung der Objektstruktur in Dobs
4. Erkundung einer bereitgestellten Implementation in Dobs durch die Schülerinnen und Schüler (vgl. Schulte 2004, S.19)

Bei der Darstellung der Verkettungsstruktur in 1. wird eine Grafik (siehe Abbildung 33) präsentiert, anhand derer die Schülerinnen und Schüler selbstständig in einem begleitenden Unterrichtsgespräch die nötigen Klassen Schlange, Knoten und Patient mit den zugehörigen Beziehungen entwickeln können. Die wichtigen Überlegungen sollte die Lerngruppe einbringen, die Lehrperson gibt gegebenenfalls kleine Impulse.

Die Erarbeitung auf Folie bzw. an der Tafel in 2. ist dynamischer Natur: Ausgehend von der bisherigen Struktur (hier können die Folien aus Phase 1 genutzt werden) kann in einem Unterrichtsgespräch durch Änderung der Bezeichnung (Patientenobjekt wird zu Knotenobjekt) und das Einfügen neuer Patientenobjekte die neue Struktur entwickelt werden. Die Gemeinsamkeiten und Unterschiede werden dann herausgestellt und gesichert. Zur Verdeutlichung wird nun noch eine



zweite Schlange in die Zeichnung aufgenommen, die einige der bereits bestehenden Patientenobjekte enthält. Diese Eigenschaft wird nun noch ergänzt.

Die Darstellung der Objektstruktur in Dobs in 3. ähnelt der an der Tafel. Allerdings ist hier nicht mehr die Dynamik gegeben. Objekte sind Instanzen einer bestimmten Klasse und diese Zugehörigkeit kann in Dobs zur Laufzeit nicht geändert werden. Es ist somit nicht möglich, die Patientenobjekte zu Knotenobjekten durch Entfernen der Klassenbezeichnung und Eintragen von „Knoten“ umzufunktionieren, wie dies an der Tafel möglich ist. Ein Herauslösen der Patientenobjekte aus der Schlange und ein Einfügen von Knotenobjekten, an die die Patienten anschließend gebunden werden, ist aufgrund der vielen Schritte nicht zu empfehlen. Außerdem müsste bereits eine Klasse Knoten im Klassendiagramm vorhanden sein. Somit ist eigentlich nur eine Nebeneinanderstellung der beiden Strukturen möglich, wodurch eine Verwendung von Dobs als ausschließliches Präsentationsmedium ungünstig erscheint. Hierzu sind als Medien Tafel oder Folie ebenso gut geeignet.

Die eigenständige Erkundung einer von der Lehrperson bereitgestellten Impementation der Schlange durch die Schüler ist dem Unterrichtsentwurf von Schulte entnommen:

Sie sollen nur mit der gegebenen Implementation (in Dobs) die Aufgabe lösen, einige Patienten nur auf den Arzt, andere aber auf den Arzt und die Blutdruckuntersuchung warten zu lassen. Anschließend sollen sie die Unterschiede zwischen der Implementation der Schlange und der vorangegangenen Verkettung der Patienten [...]

herausstellen. (Schulte 2004, S. 19)

Die Ergebnisse werden in einem Unterrichtsgespräch gesammelt und gegebenenfalls diskutiert.

Bei der eigenständigen Untersuchung in Dobs ist eventuell Hilfestellung durch die Lehrperson notwendig, damit die Schülerinnen und Schüler die Objekte günstig anordnen, um die Verknüpfungen und somit die gesamte Struktur erkennen zu können. In der Arbeit mit Dobs ungeübte Schülerinnen und Schüler müssen bei der Bedienung, etwa beim Erzeugen von Objekten, Aufrufen von Methoden und Anzeigen/Expandieren der Objektdarstellung, unterstützt werden.

Optional kann sich an dieser Stelle im Unterricht eine Implementierung der neuen Schlangenstruktur anschließen. Mit dem Wissen über die einfache Patientenverkettung in Phase 1 und die in dieser Phase erarbeiteten Unterschiede und Vorzüge sollten die Schülerinnen und Schüler eigenständig das Klassendiagramm und die nötigen Aktivitätsdiagramme in Partner- oder Gruppenarbeit erstellen können.

Es ist allerdings zu überlegen, ob eine Implementierung an dieser Stelle wirklich sinnvoll ist. Bisher kann die neue Schlangenstruktur lediglich Objekte der Klasse Patient verwalten. In der nächsten Phase wird eine Verallgemeinerung der hier erarbeiteten Struktur vorgenommen, sodass beliebige Objekte als Inhalt referenziert werden können. Die beiden Implementationen sind sich recht ähnlich.

Es ist allerdings möglich, die Diagramme aus Phase 1 in Fujaba im Sinne der neuen Struktur zu erweitern. In der nächsten Phase könnte dann diese Version erneut angepasst werden. Bei kleinen Projekten wie diesen ist das in Fujaba noch recht gut möglich. Das Klassendiagramm sollte keine Probleme bereiten und bei drei oder vier Methoden können auch die Schülerinnen und Schüler noch den Überblick behalten. Allerdings sind durch die Meldungen des Compiler bei Fehlern die fehlerhaften Stellen nicht so einfach zu finden, wodurch weniger versierte Schülerinnen und Schüler vielleicht überfordert resignieren.

Erfahrene Programmierer werden hier erkennen, dass das Ändern von Programmcode z. B. mit einer Suchen/Ersetzen-Funktion doch häufig einfacher ist als das händische Ändern aller Aktivitätsdiagramme.

6.3.3 Phase 3: Der ADT Schlange zur Verwaltung allgemeiner Objekte

Nun soll die in Phase 2 erstellte Schlangenstruktur dahingehend erweitert werden, dass die Schlange beliebige Objekte verwalten kann. So wird die Klasse *Schlange* universell einsetzbar.

Die Schülerinnen und Schüler müssen die Klassenhierarchie in Java kennen lernen. Hierbei müssen vor allem die allgemeine Klasse *Object* und ihre Besonderheiten erarbeitet werden.

Daneben benötigen die Schülerinnen und Schüler noch Kenntnis darüber, wie in Fujaba bekannte bzw. vorhandene Klassen eingebunden werden können

Sofern dies nicht bereits im Unterricht geschehen ist, muss zunächst die Klassenhierarchie und die Vererbung in Java behandelt werden.

Minimales Wissen für diese Phase:

- eine Klasse erweitert genau eine Klasse
- eine Klasse erbt die Eigenschaft und Methoden der höheren Klasse
- eine Klasse kann (somit) wie die höhere Klasse behandelt werden
- *Object* ist die oberste Klasse
- Folgerung: jedes Objekt einer Klasse kann als *Object* angesehen werden

Da Vererbung und Hierarchie in Java nun einmal vorgegeben sind, können sich die Schülerinnen und Schüler hier nichts Eigenes ausdenken. Die Erkundung sollte anhand von Java-Dokumentationen – etwa der Dokumentation von Sun⁵ – erfolgen. Die Erarbeitung der oben aufgeführten benötigten Eigenschaften kann in einem Unterrichtsgespräch erfolgen. Die Ergebnisse werden anschließend stichpunktartig festgehalten.

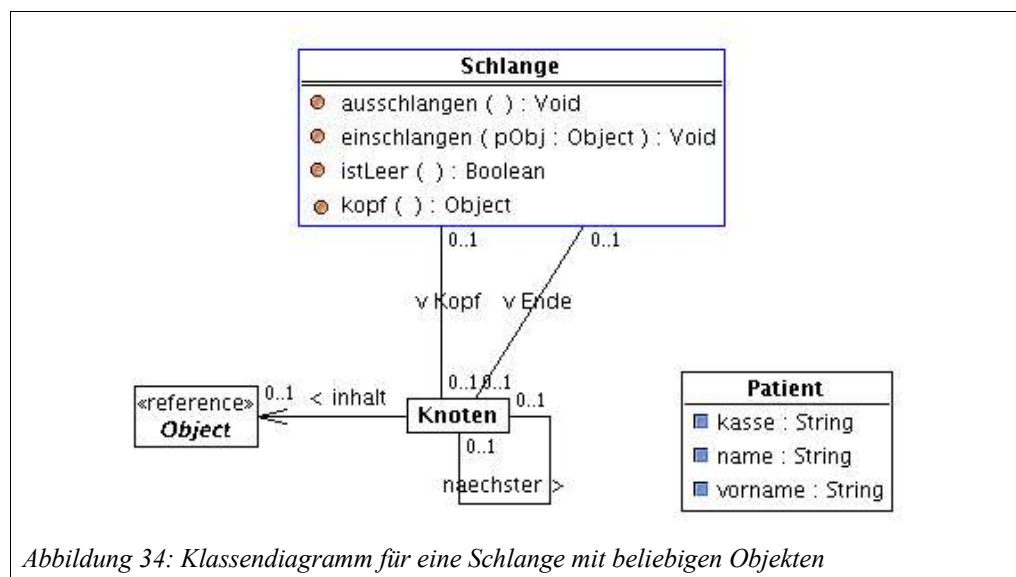
Der zweite neue Inhalt in dieser Phase: Das Verwenden vorhandener Klassen in Fujaba. Externe Klassen, d.h. solche, die nicht selbst in Fujaba im aktuellen Projekt

⁵ Die (englische) Dokumentation ist kostenlos unter <http://java.sun.com> verfügbar

implementiert werden, können dadurch eingebunden werden, indem im Klassendiagramm eine Klasse mit dem selben Namen angelegt und diese Klasse als „Reference“ deklariert wird. Fujaba stellt diese Klasse als Kasten mit dem Namen und dem Hinweis <reference> dar. Attribute und Methoden werden nicht aufgeführt.

Dieses Verfahren ist nötig, um in Fujaba die vom Java-SDK bereitgestellte Klasse *Object* verwenden zu können.

In Abbildung 34 zeigt die referenzierte Klasse *Object*.



So kann nun die Schlangenstruktur erweitert werden, dass beliebige Objekte als Inhalte aufgenommen werden können. Das zugehörige Klassendiagramm ist in Abbildung 34 zu sehen.

Um eine vernünftige Verwendung der neuen Schlange gewährleisten zu können, müssen die Methoden allerdings noch angepasst werden. Einerseits müssen die Parameter auf den Typ *Object* umgestellt werden, wo bisher *Patient* verwendet wurde, damit auch tatsächlich Objekte beliebigen Typs übergeben werden können. Andererseits ist auch eine Anpassung der in den Aktivitätsdiagrammen verwendeten Typen notwendig.

Probleme könnte den Schülerinnen und Schülern in dieser Phase die alleinstehende Klasse *Patient* und die fehlende Beziehung zur Klasse *Knoten* bereiten. Im

Klassendiagramm wird deutlich, dass die Klasse *Knoten* den Patienten nicht als Inhalt kennt. Die Klasse *Patient* ist zwar im Klassendiagramm enthalten, hat aber keine Verbindung mehr zur Schlangenstruktur. Wenn allerdings in den Eigenschaften der Klasse angegeben wird, dass sie (direkt) von *Object* abgeleitet wurde (in Fujaba: „derived from“), wird die Vererbung durch einen Pfeil angezeigt und so die Verbindung deutlich.

Sollte es hier zu Verständnisproblemen kommen, muss noch einmal auf die Hierarchie und die besondere Bedeutung der Klasse *Object* eingegangen werden. Zur Verdeutlichung kann z. B. auch in *Dobs* eine Methode (etwa `setInhalt` für den *Knoten*) mit *Object*-Parameter ausgeführt werden. In diesem Fall bietet *Dobs* alle möglichen Objekte an, die auswählbar sind. Da hier aber alle *Dobs* bekannten Objekte auswählbar sind, kann den Schülerinnen und Schülern klar gemacht werden, dass ein *Object*-Parameter alle Objekte aufnehmen kann. Der *Patient* wird so als Inhalt von einem *Knoten*-Objekt aufgenommen. Die Schlange kann nun allgemein benutzt werden.

Der Implementierung in dieser Phase sollte wieder mehr Bedeutung zukommen als in Phase 2, die ja auch aus oben dargelegten Gründen entfallen konnte. Entweder wird das in Phase 2 erstellte Projekt nun wieder erweitert oder ausgehend von den Ergebnissen aus den Phasen 1 und 2 neu implementiert.

Besonders im Hinblick auf die nächsten Phasen sollten die Schülerinnen und Schüler eine lauffähige Version der allgemeinen Schlange vorliegen haben. Zum einen kann dieses Projekt später um die Liste erweitert werden, zum anderen müssen die Schülerinnen und Schüler sich hier bereits mit den *Knoten* und den allgemeinen Objekten beschäftigen. Aufgrund der noch recht einfachen Struktur der Schlange können die Grundlagen erfahren werden, die bei den Listen direkt angewandt werden sollen, da dort nicht der Umweg über die einfache Nachfolgeverkettung erfolgen wird.

Die Implementierung sollte im Unterricht wieder in Partner- oder Gruppenarbeit erfolgen.

Anhand des bei den Erläuterungen zur Einbindung externer Klassen erzeugten Klassendiagramms und der durch die Schülerinnen und Schüler in den früheren Phasen erstellten Diagrammen sollten die Lernenden in der Lage sein, diesen

allgemeinen ADT Schlange eigenständig zu implementieren.

In der Implementationsphase hat die Lehrperson sicherzustellen, dass das Ziel, ein lauffähiges Projekt zu erstellen, von allen Gruppen erreicht wird. Bei Problemen werden Hilfestellungen gegeben. Wahlweise kann dies auch durch Mitschülerinnen oder Mitschüler geschehen. Zu überlegen ist, ob diese Hilfen individuell oder durch die Präsentation einer fertigen Lösung im Plenum geschehen soll. Der Vorteil einer Präsentation liegt in der sich anschließenden Diskussion und den Fragen, die geklärt werden können. Außerdem lernen die Schülerinnen und Schüler, ihre Lösungen und den Entscheidungsprozess zu präsentieren. Ruhige Schülerinnen und Schüler können hier zeigen, dass sie an der Arbeit in der Gruppe beteiligt sind und das Wissen verinnerlicht haben.

Nach der erfolgreichen Implementierung schließt sich eine Phase der Erkundung der Datenstruktur in Dobs an. Die Schüler sollen dazu eine Praxissituation mit Blutdruckmessungen gemäß der Problemstellung simulieren.

In einem anschließenden Unterrichtsgespräch werden kurz aufgetretene Probleme und Beobachtungen zusammengetragen.

6.3.4 Phase 4: Der ADT Liste zur Verwaltung allgemeiner Objekte

Im Unterricht wird nun der ADT Liste als Erweiterung des ADT Schlange eingeführt. Die Liste bietet umfangreichere Verwaltungsfunktionen. So können z. B. Objekte an beliebiger Stelle in die Schlange eingefügt oder entfernt werden. Außerdem ist der Zugriff auf jedes einzelne Element möglich.

Motiviert wird die Verwendung einer Liste durch das Problem, dass Patienten das Wartezimmer vorzeitig verlassen oder Patienten aufgrund eines Notfalls vorgezogen werden müssen.

Ziel ist die Erstellung einer Datenstruktur, die Patienten als beliebige Objekte (vgl. Phase 3) aufnehmen kann und das gegebene Problem löst.

Zunächst wird im Unterricht auf die Eigenschaften der Datenstruktur Liste und

zugleich die sich ergebenden Unterschiede zur Schlange eingegangen. Dies sollte durch eine Analyse der Problemstellung (siehe Abbildung 35) durch die Schülerinnen und Schüler geleistet werden können.

In der Arztpraxis kommt es immer wieder vor, dass Patienten das Wartezimmer vorzeitig verlassen, weil sie nicht so lange warten wollen. Außerdem werden manche Patienten aufgrund eines Notfalls vorgezogen, weil sie schnelle Hilfe benötigen.

Welche Schwierigkeiten ergeben sich nun bei der Wartezimmer-Verwaltung mit den bekannten Strukturen? Wie könnten sie gelöst werden? Halten Sie Ihre Überlegungen fest.

Abbildung 35: Problemstellung zur Erarbeitung des ADT Liste

Die Bearbeitung kann in Partner- oder Gruppenarbeit, in einem Unterrichtsgespräch oder auch individuell in Form einer Hausaufgabe erfolgen. Die Bearbeitung in einer Hausaufgabe hätte den Vorteil, dass sich die Schülerinnen und Schüler intensiv mit dem Problem auseinandersetzen müssten. Die Ergebnisse werden in der folgenden Unterrichtsstunde zusammengetragen.

Nun werden im Unterricht die nötigen Methoden mit ihren Eigenschaften festgelegt. Hierbei ist eine lenkende Moderation durch die Lehrperson unerlässlich. Es darf hier keine einfache Sammlung der benötigten Funktionen stattfinden, sondern die Schülerinnen und Schüler müssen auch die besonderen Fälle und Funktionalitäten erkennen oder darauf hingewiesen werden.

Abbildung 36 zeigt eine mögliche Aufstellung der Ergebnisse des Unterrichtsgesprächs.

Der ADT Liste

1. 3 besondere Knoten: `anfang`, `aktuell`, `ende`
2. Einfügen an beliebiger Stelle → `fuegeEinVor`, `fuegeEinNach`
3. Löschen an beliebiger Stelle → `loesche`
4. Ausgabe des aktuellen Elementes → `lies`
5. Bewegen der aktuell-Beziehung → `findeErstes`, `findeLetztes`,
`findeNaechstes`, `findeVorgaenger`
6. Ändern des aktuellen Elementes → `aendere`

Abbildung 36: Mögliches Ergebnis der Untersuchung der Problemstellung

In den vorangegangenen Phasen wurde auf eine explizite Spezifikation der Methoden verzichtet, da diese noch recht einfach waren und in Phase 1 nach und nach entwickelt und später nur noch angepasst wurden. An dieser Stelle sollte allerdings schon vor weiteren Überlegungen zur Implementierung tatsächlich eine genaue Beschreibung der Methoden vorgenommen werden. Parameter, Rückgabewerte und Verhalten müssen festgelegt werden. An dieser Stelle muss dieses Vorgehen als grundlegend für ADTs thematisiert werden. Die Spezifikation der einzelnen Methoden kann arbeitsteilig in Gruppen- oder Partnerarbeit geschehen, wahlweise auch als Hausaufgabe.

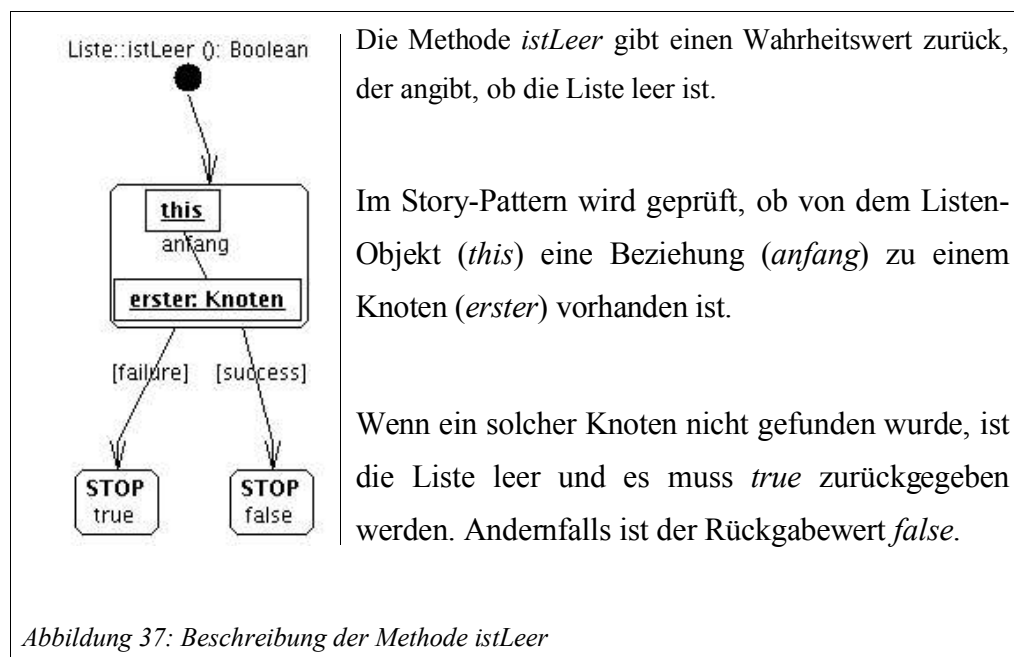
Analog zur Schlange aus der letzten Phase wird nun ein Klassendiagramm erstellt. Da einige der benötigten Methoden nicht sehr umfangreich sind, kann auf eine Implementierung durch die Schülerinnen und Schüler teilweise verzichtet werden. Es ist daher sinnvoll, diese Methoden in einer Fujaba-Projektdatei vorzugeben. Die Schülerinnen und Schüler analysieren aber die zugehörigen Aktivitätsdiagramme, um die Arbeitsweise zu verstehen. Es bietet sich an, dass sie z. B. die Methode `istLeer`, `findeErstes` und `findeNaechstes` beschreiben und darauf die sehr ähnlichen Methoden `findeLetztes` und `findeVorgaenger` eigenständig programmieren. Dazu können sehr gut Arbeitsblätter mit den

Aktivitätsdiagrammen genutzt werden, auf denen die Beschreibung direkt eingetragen werden kann.

Eine mögliche Beschreibung ist exemplarisch für die Methode `istLeer` in Abbildung 37 angegeben.

Auch die selbst erstellten Methoden können zunächst auf Arbeitsblättern angefertigt werden. Dies bringt eine Änderung der verwendeten Methodik in den Unterricht und löst etwas vom Softwaretool ab, das ja in der letzten Phase intensiv genutzt wurde und auch bei der Implementierung der schwierigeren Methoden für die Liste genutzt werden wird.

Durch die Verwendung von Arbeitsblättern zur Erkundung und zur Übertragung der Methoden ist die Bearbeitung als Hausaufgabe sehr gut möglich. Die Ergebnisse werden dann in der nächsten Unterrichtsstunde kurz präsentiert. So kann der Zeitaufwand für diese einfachen Methoden minimiert werden.



Diese Erkundungs- und Transferaufgaben leisten eine gute Vorarbeit: Die Verwendung der drei auszeichnenden Objektreferenzen `erstes`, `aktuell`, `letztes` wird klar. Bei den komplexeren Methoden werden sie den Schülerinnen und Schülern so weniger Probleme bereiten.

Mit Hilfe des (vereinfachten) Story-Driven-Modelling sollen nun die weiteren Methoden des ADT Liste entwickelt und implementiert werden .

Das Vorgehen im Einzelnen wird am Beispiel 'Füge ein Objekt nach dem aktuellen ein' exemplarisch dargestellt.

Die Schritte für das vereinfachte SDM im Überblick

- 1) Fälle stichwortartig sammeln
- 2) Story-Boarding nun für unbestimmten Fall
 - 2a) eventuell Klassendiagramm ergänzen
 - 2b) eventuell Zetteltest
- 3) Zusammenführen der Fälle zu einem Aktivitätsdiagramm
- 4) Zetteltest

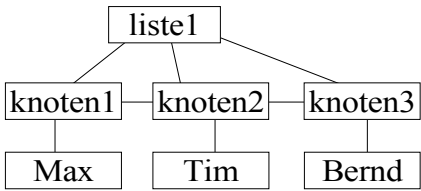
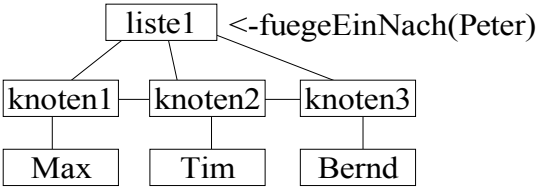
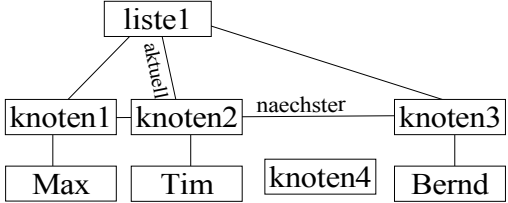
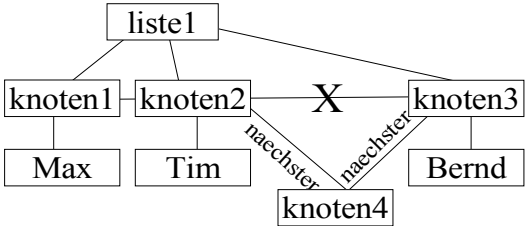
1) Fälle stichwortartig sammeln

Es sind drei Fälle für die Methode `fuegeEinNach` zu betrachten:

1. Die Liste ist leer → Einfügen als einziges Element in vorhandene Liste
2. Das letzte Element ist das aktuelle → Einfügen am Ende der Liste
3. Das aktuelle Element ist irgendwo in der Mitte der Liste (nicht am Ende)
→ Einfügen in der Mitte

2) Story-Boarding

Beim Story-Boarding werden die in einem Objektspiel ermittelten Objektstrukturen in Objektdiagrammen festgehalten. Ziel ist die Erstellung von einzelnen Aktivitätsdiagrammen für jeden einzelnen Fall. Die einzelnen Zwischenschritte im Ablauf werden in Tabelle 1 dargestellt.

Story-Boarding für den Fall 3 (Einfügen irgendwo in der Mitte der Liste)	
<p style="text-align: center;">  </p> <p>Ausgangssituation</p>	<p>Als erstes wird die Ausgangssituation beschrieben. Hierzu wird ein Objektdiagramm erstellt. Die Liste ist leer.</p> <p>(Zur Übersicht wurden die Bezeichnungen der Beziehungen hier teilweise weggelassen, zum besseren Verständnis sollten sie aber im Unterricht in das Diagramm aufgenommen werden)</p>
<p style="text-align: center;">  </p> <p>Konkreter Methodenaufruf</p>	<p>Nun folgt der konkrete Aufruf der Methode. Der Pfeil zeigt an, auf welchem Objekt die Methode ausgeführt wird.</p>
<p>1. Schritt</p> <p style="text-align: center;">  </p>	<p>Die einzelnen Schritte:</p> <p>Es wird ein Knotenobjekt erzeugt, das hinter den aktuellen Knoten (knoten2) eingefügt werden soll.</p>
<p>2. Schritt</p> <p style="text-align: center;">  </p>	<p>Der bisherige Verweis von knoten 2 auf knoten 3 wird entfernt, der neue Knoten (knoten4) in die Liste eingefügt.</p>

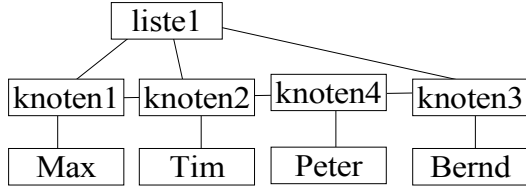
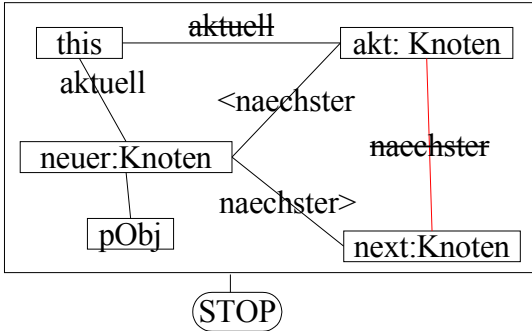
<i>Story-Boarding für den Fall 3 (Einfügen irgendwo in der Mitte der Liste)</i>	
 <p>3. Schritt</p>	<p>Das Objekt 'Peter' wird noch an den neuen Knoten 'angehängt'</p> <p>(je nach Szenario müssen evtl. noch mehr Einzelschritte durchgeführt werden)</p>
<p>Das resultierende Aktivitätsdiagramm:</p> <pre>Liste::fuegeEinNach(pObj:Object):Void</pre> 	<p>Nun folgt die Ableitung des Aktivitätsdiagrammes. Hierzu wird der Methodenaufruf wie in Fujaba übernommen und der minimale Kontext bestimmt. Dazu zählen alle Objekte, die an den Veränderungen (Suchen von Objekten, Erzeugen, Löschen von Links/Objekten) beteiligt sind.</p> <p>(In der Zusammenführung der Fälle wird das Setzen der aktuell-Beziehung und das Einhängen des Objektes pObj aus diesem Story-Pattern entfernt, da dies sonst in mehreren Story-Pattern vorkommen würde, was sehr ungünstig ist.)</p>

Tabelle 1: Überführung von Objektdiagrammen in ein Aktivitätsdiagramm für den 3. Fall (Einfügen in der Mitte einer Liste) durch Story-Boarding

Das gewonnene Aktivitätsdiagramm wird anschließend anhand der vorgegebenen Objektstruktur getestet (z. B. mit dem Zettel-Test) und gegebenenfalls korrigiert. Danach folgt die Überführung von Objektdiagrammen für die anderen beiden Fälle ebenfalls mit Hilfe des Story-Boarding, sodass man am Ende für jeden Fall ein Aktivitätsdiagramm erstellt hat. Diese müssen in dem nun folgenden Schritt miteinander kombiniert werden, damit eine Methode alle benötigten Fälle verarbeiten kann.

3) Zusammenführen der Fälle zu einem Aktivitätsdiagramm

Bei der Zusammenführung der einzelnen anhand des Story-Boarding gewonnenen Aktivitätsdiagramme müssen die einzelnen Story-Pattern in geeigneter Abfolge mit Transitionen verbunden werden. Eventuell müssen in den Aktivitätsdiagrammen noch kleine Änderungen vorgenommen (z. B. Angleichen der Bezeichner) oder mit Hilfe zusätzlicher NOPs oder Story-Pattern Verzweigungen ermöglicht werden, um einen korrekten Ablauf zu erreichen.

Schulte (2004, S. 25) schlägt vor, das Zusammenführen der Aktivitätsdiagramme in Form eines Diagramm-Puzzles durchzuführen, wobei dies wahlweise auf Papier oder direkt in Java passieren kann.

4) Zetteltest

Abschließend sollte das gesamte Aktivitätsdiagramm mit dem Zetteltest überprüft werden. Wenn die Zusammenführung im letzten Schritt bereits mit Fujaba durchgeführt wurde, kann die Methode aber auch in Dobs getestet werden. Treten hier Fehler auf, muss aber vielleicht doch der Zetteltest noch angewendet werden, um sie aufdecken und beseitigen zu können.

An dieser Stelle ist das (vereinfachte) Story-Driven-Modelling zur Erstellung des Aktivitätsdiagrammes für eine Methode abgeschlossen. Eine mögliche Implementierung der gesamten Methode ist in Abschnitt 4.3.2 (Abbildung 17) dargestellt.

Bei der Implementierung der Methode `fuegeEinVor` können die Schülerinnen und Schüler das Verfahren Story-Driven-Modelling noch einmal eigenständig durchführen und so einüben, der Unterschied zur Methode `fuegeEinNach` ist nur sehr gering. Wenn danach die Methode `loesche` umgesetzt werden soll, wird es dagegen schon schwieriger, denn es müssen vier Fälle untersucht werden und der Aufbau ist wesentlich anders. Trotzdem sollte es den Schülern möglich sein, auch diese Methode in Fujaba zu implementieren.

Nach der Realisierung aller grundlegenden Methoden sollte der hier erstellte ADT Liste wieder in den Kontext Praxis eingebracht werden. Ausgehend von der Problemstellung, die auf die Erstellung des ADT Liste hinführen sollte, kann auch

in dieser Phase wieder eine Überprüfung in Dobs stattfinden. Das bedeutet, dass die Schülerinnen und Schüler eine dem Problem entsprechende Situation in einer Arztpraxis simulieren. Hierfür ist es günstig, wenn das Projekt mit dem ADT Schlange um den ADT Liste erweitert wurde, da dann in Dobs direkt die beiden Datentypen vorhanden sind, die zur Simulation nötig sind.

Um den produktiven Einsatz der erzeugten Klassen demonstrieren zu können, kann durch die Lehrperson wieder wie in Phase 1 eine (nun umfangreichere) grafische Oberfläche zur Verfügung gestellt werden, die diese Klassen zur Verwaltung der Warteschlangen benutzt.

Im weiteren Verlauf des Unterrichts können noch einige weitere interessante Aspekte am ADT Liste thematisiert werden. Es bieten sich dazu z. B. das Suchen in einer Liste oder auch das Verfahren des Sortierens durch Mischen (Griesel und Postel 1992, S. 37, Ottmann und Widmayer 2002) an. Gerade das Sortieren durch Mischen ist sehr interessant, da hier das Prinzip *Teile-und-Herrsche (Divide-And-Conquer)* verwendet wird und so an dieser Stelle geeignet in den Unterricht eingebracht werden kann. Das Divide-And-Conquer-Prinzip zählt für Schwill (1993, S. 43) ebenfalls zu den fundamentalen Ideen der Informatik.

Eine Reflexion zum ADT Liste und zum SDM sollte diese Phase abschließen.

6.3.5 Phase 5: Der ADT Stack

In dieser Phase wird die Datenstruktur Stack – bzw. Keller oder Stapel – behandelt. Diese Phase ist nicht direkt in das Projekt Wartezimmer integriert, eine direkte Anwendung im Projekt findet nicht statt. Sie ist somit als Ergänzung zu sehen, da die zwar sehr einfache, aber dennoch wichtige lineare Datenstruktur Stack bisher im Unterricht nicht behandelt wurde.

Der Stack findet z. B. bei der Ausführung von Rekursionen Verwendung. So kann anhand dieser Datenstruktur das Prinzip der Rekursion wiederholt und vertieft werden, wenn eine erste Behandlung im vorangegangenen Unterricht bereits stattgefunden hat. Falls die Rekursion noch nicht bekannt ist, kann sie aufbauend

auf diese Datenstruktur anschließend im Unterricht erfolgen. Bei der Implementierung der Methoden für den Binärbaum ist die Rekursion unerlässlich. Eine allgemeine Einführung in die Rekursion soll aber an dieser Stelle nicht erfolgen. Sie wird in der nächsten Phase bei der Behandlung des ADT Binärbaum vorausgesetzt.

Der Lerngruppe sind die bisher behandelten linearen Datenstrukturen bekannt. Eigenschaften von Schlangen und Listen sollten die Schülerinnen und Schüler nennen und Unterschiede herausstellen können. Die Datenstruktur Stack kann als besondere Form der Liste angesehen werden, bei der Einfügen, Lesen und Entfernen von Elementen nur am Anfang möglich ist. Es gibt ähnlich der Schlange nur vier Operationen (push, pop, top, isEmpty), sodass die Schülerinnen und Schüler die Objektstruktur in Dobs in Partner- oder Gruppen erkunden können. Hier soll ihnen eine fertige Version zur Ausführung in Dobs gegeben werden, die sie ohne weitere Hinweise analysieren. Es empfiehlt sich, die Klasse mit einem neutralen Namen (etwa: Analyseklasse) zu bezeichnen, sodass von den Schülerinnen und Schülern noch keine Rückschlüsse auf die zugrunde liegende Struktur gezogen werden kann. Damit klar ist, was sie überhaupt machen sollen, ist eine klar formulierte Aufgabenstellung vorzugeben, anhand der die Eigenschaften des Stacks und die Unterschiede zu den bekannten Datenstrukturen herauszuarbeiten sind:

Untersuchen Sie in Dobs das Verhalten der Klasse 'Analyseklasse'.

1. Erstellen Sie dazu ein Objekt der Klasse und mehrere Objekte der Klasse 'Daten'.
2. Stellen Sie Vermutungen zum Verhalten der vorhandenen Methoden an
3. Fügen Sie nun die 'Daten'-Objekte nach und nach in die Datenstruktur ein und/oder entfernen Sie sie auch wieder.
4. Beschreiben Sie das beobachtete Verhalten der Datenstruktur

Abbildung 38: Aufgabenstellung zur Analyse des ADT Stack in Dobs

Aufgrund der Einfachheit soll an dieser Stelle keine Implementierung durch die Schülerinnen und Schüler erfolgen, da die Unterschiede zur Implementation der

Schlange nur sehr gering sind.

Nach der Erkundungsphase werden die Ergebnisse in einem Unterrichtsgespräch gesammelt. Hier sollte sich die Vorstellung von einem Stapel ergeben, sodass die Ergebnissammlung (Abbildung 24) eine Überschrift bekommen kann.

Der ADT Stapel/Stack	
1. Einfügen von Elementen immer an einem Ende	→ <i>push</i>
2. Auslesen des zuletzt eingefügten Elementes	→ <i>top</i>
3. Entfernen von Elementen an dem gleichen Ende	→ <i>pop</i>
4. Test, ob die Datenstruktur leer ist	→ <i>isEmpty</i>
5. LIFO-Prinzip (Last In – First Out)	

Abbildung 39: Mögliches Tafelbild mit den Ergebnissen der Analyse des ADT Stack

Anschließend kann eine Betrachtung des Klassendiagramms und der Aktivitätsdiagramme erfolgen. Es genügt, den Ablauf der Methoden kurz beschreiben zu lassen. Durch Vorlage der Diagramme auf Papier können die Schülerinnen und Schüler sich dazu jeweils Stichpunkte notieren und so die Ergebnisse und die Diagramme in ihrem Heft bzw. in ihrem Ordner sichern. Ein Vergleich mit den Methoden der Schlange, die zeitgleich an die Wand projiziert werden sollten, wird die Eigenheiten und Unterschiede vermutlich sehr schön herausstellen können.

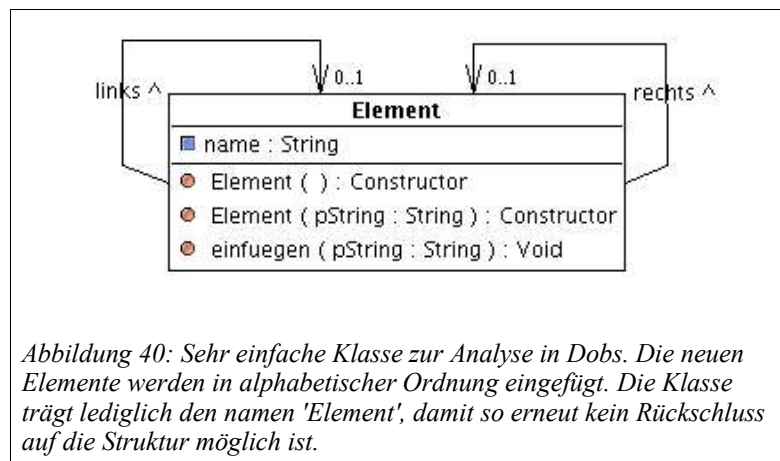
Zum Abschluss dieser Phase sollte die Bedeutung des Stapels in der Informatik behandelt werden. Dazu eignet sich eine Betrachtung von Methodenaufrufen (siehe dazu Abschnitt 2.3.1), insbesondere bei der Rekursion am Beispiel der Türme von Hanoi oder beim Back-Tracking-Verfahren.

6.3.6 Phase 6: Der ADT Binärbaum

Die in den vorhergehenden Phasen behandelten Datenstrukturen waren alle linearer Strukturen. In dieser Phase sollen die Schülerinnen und Schüler nun eine nicht-

lineare, hierarchische Struktur kennenlernen: Den binären Baum.

Zum Einstieg sollten die Schülerinnen und Schüler auch hier die Struktur zunächst in Dobs analysieren. Dabei sollen die Schüler den besonderen Aufbau eines (binären) Baumes erfahren können: Ein Element besitzt bis zu zwei Nachfolger. Es bietet sich an, der Lerngruppe eine entsprechende Klasse zur Verfügung zu stellen, bei der die eingefügten Elemente alphabetisch angeordnet werden, so dass die Ordnung in Dobs auch durch die Schülerinnen und Schüler schnell erkannt werden kann. So werden sie vermutlich die Objekte übersichtlich anordnen können. Eventuell ist hierbei aber auch eine Hilfestellung durch die Lehrperson nötig. Abbildung 40 zeigt eine Klasse, die zu Analysezwecken in Dobs ausreichen sollte.



Führen Sie folgende Schritte in Dobs durch:

1. Erstellen Sie ein Objekt der Klasse 'Element' mit dem Parameter 'Helmer'
2. Fügen Sie den Namen 'Meyer' ein.
3. Fügen Sie den Namen 'Orbke' ein.
4. Fügen Sie nacheinander die Namen 'Schmitt', 'Nehm', 'Nolte', 'Domar', 'Ahlen', 'Böttcher' ein und ordnen Sie die Objekte geeignet an.
5. Beschreiben Sie anschließend den Aufbau der Objektstruktur.
6. Beschreiben Sie die Wirkung der Methode *einfuegen* und ziehen Sie Rückschlüsse auf einen angemessenen Algorithmus.

Abbildung 41: Aufgabenstellung zur Analyse der Baumstruktur in Dobs

In Abbildung 41 ist eine mögliche Aufgabenstellung formuliert. Die Bearbeitung sollte in Partnerarbeit erfolgen.

Das Einfügen der Elemente in Dobs sollte keine Schwierigkeiten bereiten. Eine geeignete Anordnung der Elemente ist dagegen eventuell nicht ganz so einfach. Für eine korrekte Beschreibung der Struktur ist sie aber wahrscheinlich nicht nur hilfreich, sondern notwendig. Die Beschreibung der Wirkung von *einfüegen* sollte ebenfalls bewerkstelligt werden können, zumindest dann, wenn auch die Struktur erkannt werden konnte. Einen geeigneten Algorithmus werden vermutlich nicht alle Schülerinnen und Schüler liefern können. Die Bearbeitung der Schritte 5 und 6 kann auch in einer Hausaufgabe erfolgen. Dazu sollten sich aber alle das Objektdiagramm abzeichnen oder ausdrucken. Am besten wäre natürlich, wenn alle Schülerinnen und Schüler auch zu Hause die Möglichkeit hätten, die vorhergehenden Schritte noch einmal in Dobs durchführen zu können, um optimale Voraussetzungen zu haben.

Die Ergebnisse werden (evtl. in der nächsten Stunde) in einem Unterrichtsgespräch zusammengetragen. Da die Begrifflichkeiten noch nicht bekannt sind, werden hier vermutlich ungenaue Formulierungen verwendet werden. Aufgrund der Kenntnisse über die linearen ADTs werden die Schülerinnen und Schüler aber sicherlich keine Probleme haben, einander zu folgen.

Im Unterricht sollte sich trotzdem eine Phase anschließen, in der die allgemeinen Begriffe im Zusammenhang mit Bäumen kennengelernt werden. Hierzu zählen unter anderem *hierarchische Struktur, Knoten, Wurzel, (rechter/linker) Nachfolger, Kante, Teilbaum, Pfad, Tiefe...* Gerade mit dem Begriff *Teilbaum* kann eine rekursive Definition eines (binären) Baumes erfolgen.

Daraufhin kann dann eine Klasse *Baum* erstellt werden, ähnlich der einfachen Klasse für die Analyseaufgabe in Dobs (siehe Abbildung 40).

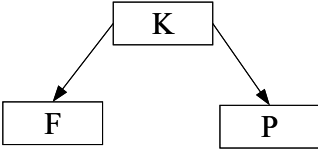
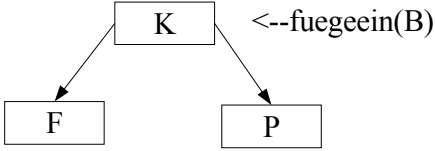
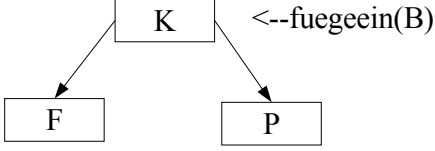
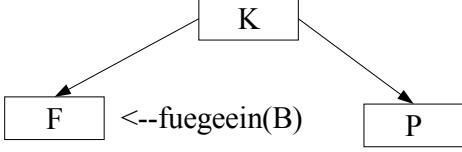
Anschließend kann die hier noch einfache Methode *einfüegen* mit Hilfe des SDM implementiert werden. Die Rekursion wird hier als bekannt vorausgesetzt. Falls im bisherigen Unterricht die Rekursion noch nicht thematisiert wurde, kann das an dieser Stelle im Unterricht geschehen. In dieser Arbeit wird darauf aber nicht eingegangen.

Das Vorgehen im SDM:

1. Fälle finden:

Es muss jeweils für das aktuelle Wurzelement entschieden werden, ob links oder rechts eingefügt werden muss. Daher wird dieses Problem aufgeteilt in „Einfügen immer links“ und „Einfügen immer rechts“.

2. Story-Boarding für den Fall „Einfügen immer links“

Story-Boarding zum Einfügen eines Elementes in einen Baum „immer links“	
 <p>Ausgangssituation:</p>	Einfacher kleiner Baum als Ausgangssituation
<p>Konkreter Methodenaufruf:</p> 	Einzeichnen des Methodenaufrufs
<p>Schritt 1:</p> 	Es muss ein Vergleich durchgeführt werden, das B muss demnach links von dem K-Objekt eingeordnet werden. Da K aber bereits einen linken Nachfolger hat, ist ein einfaches Einhängen nicht möglich.
<p>Schritt 2:</p> 	Das Einhängen „direkt links unter“ dem K war nicht möglich, somit muss nun versucht werden, das B an dem linken Teilbaum einzusortieren. Der Aufruf erfolgt rekursiv.

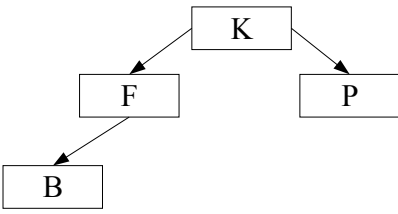
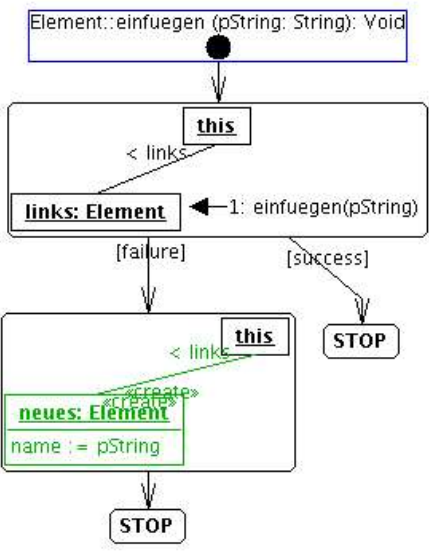
Story-Boarding zum Einfügen eines Elementes in einen Baum „immer links“	
<p>Schritt 3:</p> 	<p>Die Prüfung ergibt, dass das Element links vom F einsortiert werden muss. Da dort noch kein Element hängt, wird das B eingefügt.</p>
<p>Das zugehörige Aktivitätsdiagramm:</p> 	<p>Bei der Überführung in das Aktivitätsdiagramm wurde jetzt vernachlässigt, dass erst geprüft wurde, ob das Element links oder rechts einzusortieren ist.</p>

Tabelle 2: Beispielhaftes Story-Boarding zum Einfügen eines Elementes, das ganz links einsortiert werden muss

Das Aktivitätsdiagramm sollte anschließend noch einmal getestet werden. Für ein „Einfügen immer rechts“ kann das Story-Boarding analog durchgeführt werden oder aber das Aktivitätsdiagramm mit guter Argumentation einfach „gespiegelt“ werden.

Durch einen Vergleich mittels NOP können dann die beiden Aktivitätsdiagramme zusammengeführt werden.

Das zusammengeführte Aktivitätsdiagramm ist in Abbildung 42 dargestellt.

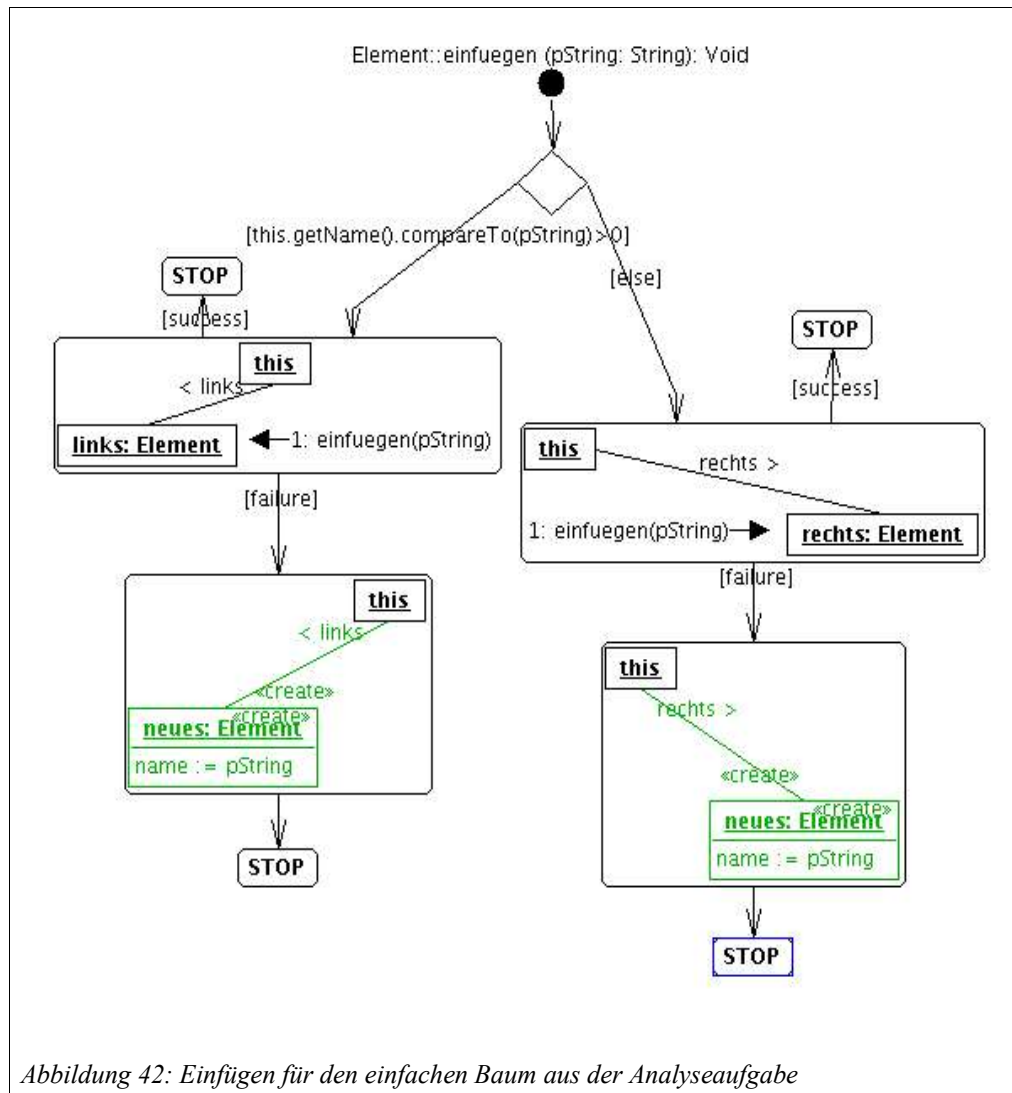


Abbildung 42: Einfügen für den einfachen Baum aus der Analyseaufgabe

Im weiteren Verlauf der Phase kann die Erarbeitung eines Baumes erfolgen, der – analog zur Schlange und zur Liste – beliebige Elemente aufnehmen kann. Eine exemplarische Implementierung eines solchen Binärbaumes ist in Abschnitt 4.4 dargestellt. Dafür ist die Verwendung des *Comparable*-Interfaces notwendig. Es ist also gegebenenfalls eine Thematisierung dessen im Unterricht notwendig.

Ein solcher Baum könnte dann wieder in den Kontext unserer Arztpraxis eingebunden werden. Denkbar ist z.B. die Verwaltung der Patienten in einem Baum. Die Patienten, die zum Arzt kommen, werden dann in die Wartezimmerschlange(n) eingefügt. Die Komplexität einer solchen Objektstruktur würde allerdings den Rahmen von Dobs bei weitem überschreiten, da eine übersichtliche Darstellung wahrscheinlich nicht mehr gegeben sein wird. Es ist also

angebracht, wieder eine grafische Oberfläche zu erstellen, die die vorhandenen Klassen zur Verwaltung der Daten nutzt.

Die Implementierung dieses Baumes kann teilweise vorgegeben werden, da sie insgesamt doch recht aufwendig ist.

Der Themenbereich (Binär-)baum ist an dieser Stelle natürlich noch längst nicht erschöpft. Im folgenden Unterrichtsverlauf könnten noch viele weitere Aspekte behandelt werden. Dazu zählen z.B. Analytische Betrachtungen, Balancieren von Bäumen und weitere Baum-Arten (AVL-Bäume, B-Bäume...), um nur eine kleine Auswahl zu nennen. Dies soll allerdings nicht mehr im Rahmen dieser Arbeit erfolgen.

7 Schlussbemerkungen

In der vorliegenden Arbeit wurde der Frage nachgegangen, ob und wie das im Anfangsunterricht der Jahrgangsstufe 11 eingesetzte life³-Unterrichtskonzept in der Stufe 12 fortgesetzt werden kann. Gegenstand des Unterrichts war dabei das Konzept der ADTs, anhand dessen eine typische Vorgehensweise bei der Entwicklung von Lösungsmethoden erarbeitet werden kann: Probleme können mehrfach in Teilprobleme zerlegt werden, die nun leichter zu lösen sind. Zusammen ergeben diese später eine Lösung für das Gesamtproblem .

Um mich der Problematik zu nähern habe ich mich zunächst mit der Bedeutung von Abstraktion für die Informatik befasst, die die Entwicklung von abstrakten Datentypen begründet hat. Die Eigenschaften von ADTs sind stets nur sehr allgemein formuliert. Aufgrund dessen habe ich eine Charakterisierung der zu betrachtenden ADTs (Stack, Schlange, Liste und Binärbaum) vorgenommen, wie ich sie auch später implementiert und in der Unterrichtsskizze verwendet habe.

In Kapitel 3 schließt sich eine Betrachtung der UML und von CASE-Tools und ihrer Bedeutung für die heutige Softwareentwicklung an. Aufgrund meiner Erkenntnisse erachte ich einen Einsatz der UML auch im Informatikunterricht als sinnvoll. Für mich ist eine Unterstützung durch CASE-Tools dabei für den Unterricht genauso erstrebenswert.

Anschließend habe ich exemplarisch die Implementierung der ADTs in Fujaba vorgenommen. Dazu lässt sich feststellen, dass auch die Realisierung recht komplexer Methoden in Fujaba möglich ist. Allerdings ist die Implementierung teilweise sehr aufwändig, umständlich und unübersichtlich.

In Kapitel 5 habe ich dargestellt, dass die Inhalte und die angewandte Methodik der anschließend beschriebenen Unterrichtsreihe dem Lehrplan entsprechen und unter fachdidaktischen Gesichtspunkten Anwendung finden können.

Die Verwendung des durchgehenden Beispiels 'Arztpraxis' in der dargestellten Unterrichtsreihe stellt insgesamt einen Kontext dar und schafft so eine förderliche Geschlossenheit. Daraus ergibt sich vermutlich auch für die Schülerinnen und Schüler ein einheitliches und umfassendes Bild der behandelten Inhalte. Außerdem bleibt der Problembereich der gleiche, die Probleme sind darin jeweils leicht different.

Nach gemeinsamem Zugang werden die Schülerinnen und Schüler stets dazu angehalten, Lösungen zunehmend selbständig zu erarbeiten, größtenteils in Partner- oder Gruppenarbeit. Zusammenarbeit wird gefördert und erfordert somit die Auseinandersetzung miteinander. Sozialkompetenz wird dadurch erworben und erweitert.

Fujaba ermöglicht die Umsetzung der in der Modellierungsphase erarbeiteten Diagramme in Quellcode. Die erstellten Diagramme sind in der Regel gut überschaubar und ermöglichen es so, den Sinn schnell zu erfassen. Außerdem erleichtern sie es den Schülerinnen und Schülern, die erarbeiteten Lösungen im Unterricht zu präsentieren. Mit Hilfe von Dobs können die normalerweise nicht sichtbaren Objektstrukturen in der Java Virtual Machine anschaulich dargestellt und direkt manipuliert werden.

Es ist durchaus denkbar, sowohl im Grund- als auch im Leistungskurs eine solche Unterrichtsreihe zu ADTs unter Verwendung von Fujaba durchzuführen. Im Leistungskurs sind dazu allerdings in einzelnen Bereichen zusätzliche verwandte Themenbereiche und tiefgreifendere Analysen auf höherem Niveau einzuflechten.

Ausgehend von dem bisher Geschriebenen möchte ich zukünftig die vorgestellte Unterrichtsreihe konkret umsetzen und die hier gewonnenen Erkenntnisse überprüfen.

8 Literaturverzeichnis

Arlow und Neustadt 2002; Arlow, Jim; Neustadt, Ila: UML and the Unified Process. Practical object-oriented analysis and design. Pearsons Education Limited 2002.

Azmoodeh 1990: Azmoodeh, Manoochehr: Abstract Data Types and Algorithms. Macmillan Education Ltd 1990.

Baumann 1993: Baumann, Rüdiger: Informatik für die Sekundarstufe II. Band 2. Ernst Klett Schulbuchverlag 1993.

Baumann 1996: Baumann, Rüdiger: Didaktik der Informatik. 2. Auflage. Ernst Klett Verlag 1996.

Burkert, Griesel und Postel 1992: Burkert, Jürgen; Griesel, Heinz; Postel, Helmut: Informatik heute – Algorithmen und Datenstrukturen. Schrödel Schulbuchverlag 1992.

Claussen 1993: Claussen, Ute: Objektorientiertes Programmieren: mit Beispielen und Übungen in C++. Springer-Verlag 1993.

Diethelm, Geiger und Zündorf 2004: Diethelm, Ira; Geiger, Leif; Zündorf, Albert: Systematic Story Driven Modeling. Technical Report, University of Kassel. www.se.eecs.uni-kassel.de/se/fileadmin/se/publications/SDM04.pdf (zuletzt besucht am 15.10.2004)

fujaba-Webseite: <http://www.fujaba.de> (zuletzt besucht am 15.10.2004)

Hubwieser 2004: Hubwieser, Peter: Didaktik der Informatik. Grundlagen, Konzepte, Beispiele. 2. Auflage. Springer-Verlag 2000.

Jenkins 1998: Jenkins, Michael S.: Abstract data types in Java. McGraw-Hill Companies 1998.

learnline: <http://www.learn-line.nrw.de>. Das Projekt Wartezimmer ist nur für Lehrerinnen und Lehrer in einem geschützten Bereich zugänglich, es wurde mir aber in einer gespiegelten Version über den Server der Universität Paderborn zur Verfügung gestellt (zuletzt besucht am 15.10.2004)

life-Webseite: <http://life.upb.de> (zuletzt besucht am 15.10.2004)

- Moll 2002: Moll, Stefan: Objektorientierte Modellierung unter Einsatz eines CASE-Tools im Informatikunterricht der Jahrgangsstufe 11. In: Schubert, Magenheimer, Hubwieser und Brinda 2002. S. 43-52.
- Meyer 1990: Meyer, Bertrand: Objektorientierte Softwareentwicklung. Aus dem Amerikanischen übersetzt von Werner Simonsmeier. Carl Hanser Verlag 1990. [Orig.: Object-oriented Software Construction. 1998]
- Oesterreich 2004: Oesterreich, Bernd: Objektorientierte Softwareentwicklung. Analyse und Design mit der UML 2.0. 6. Auflage. Oldenbourg Wissenschaftsverlag GmbH 2004
- Ottmann und Widmayer 2002: Ottmann, Thomas; Widmayer, Peter: Algorithmen und Datenstrukturen. 4. Auflage. Spektrum Akademischer Verlag 2002.
- Quibeldey-Cirkel 1994: Quibeldey-Cirkel, Klaus: Das Objekt-Paradigma in der Informatik. Teubner 1994.
- Richtlinien: Ministerium für Schule, Wissenschaft und Forschung (Hrsg.): Richtlinien und Lehrpläne für die Sekundarstufe II – Gymnasium/Gesamtschule in Nordrhein-Westfalen. Informatik. Ritterbach Verlag 1999.
- Rollke und Sennholz 1994: Rolleke, Karl-Hermann; Sennholz, Klaus: Grund- und Leistungskurs Informatik. Cornelsen Verlag 1994.
- Schulte 2004: Schulte, Carsten: Fujaba als Werkzeug zur anschaulichen Darstellung ausgewählter abstrakter Datentypen. Schriftliche Hausarbeit, Universität Paderborn 2004.
- Schwill 1993: Schwill, Andreas: Fundamentale Ideen der Informatik. URL: <http://ddi.cs.uni-potsdam.de/Forschung/Schriften/ZDM.pdf> (zuletzt besucht am 15.10.2004)

9 Versicherung

Ich versichere, dass ich die schriftliche Hausarbeit selbstständig verfasst und keine anderen Quellen und Hilfsmittel als die angegebenen benutzt habe. Alle Stellen der Arbeit, die dem Wortlaut oder dem Sinn nach anderen Werken entnommen sind, habe ich in jedem einzelnen Fall unter genauer Angabe der Quelle deutlich kenntlich gemacht. Das gleiche gilt auch für die beigegebenen Zeichnungen, Kartenskizzen und Darstellungen. Zudem versichere ich, dass ich den softwarepraktischen Teil der Arbeit selbstständig entwickelt habe.

Datum

Unterschrift

10 Anhang

10.1 Glossar

Aktivitätsdiagramm

Grafische Darstellung für die Implementation einer Methode in Fujaba.

Collaboration Statement

Ermöglicht Methodenaufrufe in Story-Pattern. Wird nur bei erfolgreicher Prüfung der Objektstruktur ausgeführt und ist an ein bestimmtes Objekt gebunden.

Dobs

Abkürzung für 'Dynamic Object Browsing System'. Grafischer Debugger. Stellt Objektstrukturen dynamisch zur Laufzeit grafisch dar. Ermöglicht das Ausführen von Methoden direkt auf den angezeigten Objekten.

Failure

Ergebnis eines Story-Pattern, wenn die Abarbeitung nicht erfolgreich war. Dieses Ergebnis kann über Transitionen ausgewertet und so der Ablauf der Methode bestimmt werden.

Link

Beziehung zwischen Objekten.

NOP (No operation)

Wird in Aktivitätsdiagrammen verwendet. Besitzt selbst keine Funktionalität, ermöglicht aber z. B. Verzweigungen und Schleifen, da mehrere ein- und ausgehende Transitionen möglich sind. Dargestellt durch eine Raute.

Objektdiagramm

Grafische Darstellung von Objektstrukturen. Objekte werden als Kasten, Beziehungen durch Pfeile oder Linien symbolisiert. Dobs verwendet Objektdiagramme zur Darstellung.

Objektspiel

Methode zum Überprüfen von CRC-Karten oder Klassendefinitionen. Es werden mit Objektinstanzen einzelne Anwendungsfälle durchgespielt, anhand derer der Entwurf geprüft werden kann.

Reference

Zum Einbinden externer Klassen werden sie in Fujaba als <Reference> deklariert.

Statement

Element in Aktivitätsdiagrammen. Bietet die Möglichkeit, Java-Quellcode direkt in ein Aktivitätsdiagramm einzufügen.

Story-Boarding

Verfahren zur Übertragung von Objektdiagrammen in Aktivitätsdiagramme.

Story-Driven-Modelling

Verfahren zur Modellierung von Methoden. Anhand einzelner Szenarien werden Aktivitätsdiagramme erarbeitet, die später zu einem alle Fälle abdeckenden Aktivitätsdiagramm vereinigt werden.

Story-Pattern

Objektdiagramm in einem Aktivitätsdiagramm. Besitzt drei Aufgaben: 1. Prüfung der Objektstruktur 2. Erweitern der Objektstruktur durch neue Objekte und Beziehungen 3. Löschen vorhandener Objekte/Beziehungen. Ergebniswert failure (Fehler bei der Objektstruktur) oder success (Erfolg).

Success

Ergebnis eines Story-Pattern, wenn die Abarbeitung erfolgreich war. Siehe Failure.

Transition

Verbindungen zwischen den Bestandteilen eines Aktivitätsdiagrammes. Darstellung in Form eines Pfeiles, der die Ausführungsrichtung angibt.

10.2 CD

Auf der beiliegenden CD finden Sie:

- Die verwendete Fujaba-Version
- Fujaba-Projekte mit Grafiken (Klassen- und Aktivitätsdiagramme ...)
- Die Fujaba-Dokumentation von der Fujaba-Webseite
- Vergleich mehrerer CASE-Tools