Schriftliche Hausarbeit

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

Michael Gerdelmann

Thema:

Didaktische Implikationen des Einsatzes von Softwaretools zur Visualisierung von Ablaufprozessen objektorientierter Software

Paderborn, 31. Mai 2000

Gutachter: Prof. Dr. Magenheim
Universität-Gesamthochschule Paderborn
Fachbereich 17: Didaktik der Informatik

Inhaltsverzeichnis

Inhaltsverzeichnis 1

Abbildungsverzeichnis 2

1 Vorwort 4

2 Didaktische Einordnung 5

2.1 Informatiksysteme 7

2.2 Dekonstruktion 9

2.3 Tools zur Visualisierung von Ablaufprozessen objektorientierter Software 10

3 Softwaretechnische Perspektive 12

3.1 Softwareentwicklungsprozeß 12

3.2 Voruntersuchung 14

3.3 Systemanalyse 15

3.4 Objekte 17

3.5 Klassen und Objekte 20

3.6 Prüfung und Test von Software 24

3.6.1 Testlauf 25

3.6.2 Theoretische Prüfung 25

3.6.3 Debug session – klassisch 26

3.6.3.1 Verhalten bei Fehlern 27

3.6.3.2 Callstack 28

3.6.3.3 Step by step 28

3.6.3.4 Breakpoints 29

3.6.3.5 Zusammenfassung 29

3.6.4 Debug session – grafisch 30

3.6.4.1 Black Box Test der kleinsten Elemente . . . 30

3.6.4.2 Grey Box Test eines Programms 31

3.6.5 Resumee 32

4 Mr. Dobs **33**
4.1 Visualisierung von Ablaufprozessen . . . 33
4.2 Fujaba . . . 33
4.3 Mr. Dobs . . . 34
4.4 Beispiel: Elevator . . . 34
4.4.1 1. Durchlauf . . . 35
4.4.2 Dekonstruktion . . . 47
4.4.3 2. Durchlauf . . . 52
4.5 Fazit . . . 57

5 Folgerungen für den Informatikunterricht **59**
5.1 Verschiedene Ebenen . . . 59
5.2 Benutzeroberfläche/Daten . . . 60
5.3 Datenobjektsystem und Quellcode . . . 64
5.4 Inhaltliche Einbettung in den Informatikunterricht . . . 65

A Evolution des Mr. Dobs **69**
A.1 Methoden kommentieren . . . 69
A.2 Pakete . . . 69
A.3 Preferences . . . 70
A.3.1 Icons für Objekte . . . 70
A.3.2 Ausblenden von Methoden . . . 70
A.3.3 User Preferences . . . 70
A.3.4 Session Preferences . . . 71

B Newsgroup Diskussion **72**

Literaturverzeichnis **81**

C Versicherung **83**

Abbildungsverzeichnis

1 Übersicht . . . 4
2 Objektevolution (2) . . . 23
3 Sichtwechsel . . . 23

4 Elevator step 1 35
5 Elevator step 2 36
6 Elevator step 3 37
7 Elevator step 4 38
8 Elevator step 5 39
9 Elevator step 6 40
10 Elevator step 7 40
11 Elevator step 8 41
12 Elevator step 9 42
13 Elevator step 10 42
14 Elevator step 11 43
15 Elevator step 12 44
16 Elevator step 13 44
17 Elevator step 14 45
18 Elevator step 15 45
19 Elevator step 16 46
20 Elevator Klassendiagramm 47
21 Person::enterElevator() (Fujaba) 49
22 Person::gotoElevator() (Fujaba) 50
23 Person::gotoElevator() (Fujaba) 52
24 Elevator step 1 53
25 Elevator step 2 53
26 Elevator step 3 54
27 Elevator step 4 54
28 Elevator step 5 55
29 Elevator step 6 56
30 Elevator step 7 56
31 Elevator step 8 57
32 Benutzeroberfläche ↔ Datenobjekte 61
33 Benutzeroberfläche ↔ Datenobjekte aus Systemsicht 61
34 Benutzeroberfläche ↔ Datenobjekte ↔ Mr. Dobs 63
35 Laufzeitebene ↔ Beschreibungsebene (1) 64
36 Laufzeitebene ↔ Beschreibungsebene (2) 65
37 Laufzeitebene ↔ Beschreibungsebene (2) 67

1 Vorwort

In meiner langjährigen Praxis als Programmierer stieß ich immer wieder auf Probleme, deren Ursache in den internen Abläufen eines objektorientierten Programms verborgen waren. Bisher gab es keine Möglichkeit diese sichtbar zu machen. Nach der Veröffentlichung der Software Mr. Dobs, steht nun ein Tool zur Verfügung was diese Abläufe grafisch visualisiert.
In der vorliegenden Arbeit werden die didaktischen Implikationen des Einsatzes von Mr. Dobs zur Visualisierung von Ablaufprozessen objektorientierter Software behandelt.
Aus der folgenden grafischen Übersicht ist die grobe Strukturierung der Arbeit zu entnehmen:

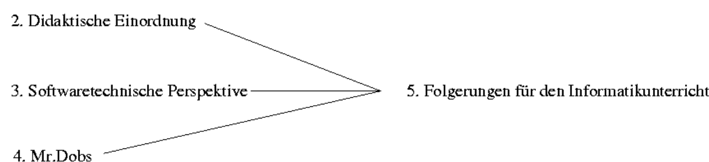


Abbildung 1: Übersicht

Kapitel 2 verfolgt die Einordnung des Themas in den fachdidaktischen Kontext.
In Kapitel 3 wird das Thema aus softwaretechnischer Sicht beleuchtet. Im Hintergund steht ein abstrakter Softwareentwicklungsprozeß, in dem für diese Arbeit signifikante Probleme erläutert werden.
Die Softwaretools Mr. Dobs und Fujaba werden in dem darauffolgenden 4. Kapitel kurz vorgestellt. Anschließend wird anhand dieser Tools eine Analyse des Informatiksystems “Elevator-Demo” vorgenommen.
Abschließend werden in Kapitel 5 die Folgerungen der vorhergehenden Kapitel auf den Informatikunterricht bezogen. Im Vordergrund steht hierbei die didaktische Plazierung des Tools Mr. Dobs in den Unterrichtsverlauf sowie die sich daraus ergebenden Vorteile.

2 Didaktische Einordnung

“Wir müssen endlich aufhören, die großen Systeme einfach hinzunehmen. Sie sind weder selbstverständlich noch unabänderlich noch übermächtig. Menschen machen sie so groß, und Menschen können sie klein machen. Sie können gleich damit anfangen.”[Sie96]

Der GI-Fachausschuß 7.3 “Informatische Bildung in Schulen” stellt fest:

> Informations- und Kommunikationstechnologien verändern die Lebens- und Arbeitswelt und sind daher von großer Gegenwarts- und Zukunftsbedeutung. Wenn mit Bildung der Anspruch verknüpft ist, Kinder und Jugendliche zur Teilhabe am gesellschaftlichen Leben zu befähigen, muss die Auseinandersetzung mit den Informations- und Kommunikationstechnologien ein Element von Allgemeinbildung sein. [GIF99]

Daraus ergeben sich folgende Anforderungen an die Inhalte einer informatischen Bildung insbesondere an den Informatikunterricht, dem damit ein fester Platz im Rahmen der Unterrichtsfächer zusteht:

1. Sie behandelt Ausschnitte der Welt unter informationstechnologischen Aspekten
   Daraus folgt die Auseinandersetzung mit informationstechnologischen Prozessen in Natur, Technik und Gesellschaft sowie Nutzbarmachung dieser Informatiksysteme.

2. Sie wird unter sozialem Kontext und im Anwendungsbereich betrachtet
   Daraus folgt die Analyse von Strukturen oder Arbeitsweisen und Induktion/ Rückführung auf induzierte Prinzipien.

3. Im Vergleich zu den Natur-, Geistes- und Gesellschaftswissenschaften leistet sie einen eigenständigen Beitrag zum Verständnis von Welt wie auch zur Bewältigung von Zukunftsaufgaben.

Anschließend stellt der Text [GIF99] einige informatische Zugänge zum grundlegenden Verständnis computerbasierter Medien dar, unter anderem das "Verarbeiten von Daten"[1]:

> 1. Welche Modellbildungen liegen einem Informatiksystem zugrunde, wie sind die Problemstellungen strukturiert?
> 2. Welches Verfahren wird bei der Softwareentwicklung angewendet und welchen Einfluss hat dies u.a. auf den Gestaltungsprozess?
> 3. Welche Algorithmen und Datenstrukturen bzw. Objekte sind in der Software implementiert?
>
> [GIF99]

Daraus können sich folgende Fragen ergeben:

1. Wie kann eine Modellbildung, die einem Informatiksystem zugrundeliegt, nachvollzogen werden? Und welche Werkzeuge sind dafür geeignet?

2. Welche Beziehungen bestehen zwischen dem Verfahren (Softwareentwicklungsprozeß) und dem Gestaltungsprozeß?

3. Auf welche Weise sollen Algorithmen, Datenstrukturen **und** Objekte einer fertigen (implemetierten) Software erkundet werden?

Welche zentralen Inhalte sollte also Informatikunterricht haben? Wäre der Begriff des Algorithmus noch im Zentrum des Unterrichts, wäre diese leicht zu beantworten, "indem anhand schrittweise immer komplexer werdender Algorithmen Grundkonzepte des Programmierens im Kleinen eingeführt werden [. . .]. Die Komplexität der erstellten Systeme ist bei solch einem Vorgehen jedoch zwangsläufig sehr begrenzt"[Ste99, S.9]. Hubwieser schlägt "als Lösung [. . .][vor], im Lehrplan die Vermittlung konkreter Modellierungstechniken anstatt programmier-technischer Details vorzuschreiben"([Hub99]).

[1] Die anderen: "Interagieren mit einem Informatiksystem" und "Kommunizieren in Netzten"

Sein Vorschlag beinhaltet eine konstruierende Vorgehensweise , in der die behandelten Systeme immer komplexer werden. Von einer abstrakteren Sicht schlägt Hubwieser etwas vor, was im kleinen beginnt und schrittweise komplexer wird. Es ist damit zwar möglich –mit diesen objektorientierten Methoden– komplexere Systeme zu **erstellen**, doch sind damit die Fragen auch nicht zu beantworten. Sie legen den Schwerpunkt auf die Analyse und Beurteilung von Informatiksystemen und nicht auf deren Entwicklung.
Unklar ist nun noch, was Informatiksysteme sind und wie sie im Informatikunterricht behandelt werden sollen.

2.1 Informatiksysteme

Es stellt sich die Frage nach dem Nutzen der Vermittlung von Informatiksystemen. Nach welchen Kriterien werden diese Systeme für den Unterricht ausgewählt und wie lassen sie sich vergleichen. Was ist ein System? Und was macht ein System komplex?

> Ein System S ist eine Menge von Elementen E_i, die durch Beziehungen miteinander verbunden sind und gemeinsam einen bestimmten Zweck erfüllen.[Leh95]

Im allgemeinen gibt es aber Systeme, die jeweils ihre spezifischen Probleme mit sich ziehen. Dies ist bei kleinen wie auch großen Systemen der Fall. Siefkes geht es dabei vordergründig um die Probleme, die wachsende Systeme mit sich bringen. Als Beispiele hierfür führt Siefkes eine Firma und eine Hochschule an. Er hält fest, daß das Wachstum solcher Systeme Einfluß auf die Veränderung aller Faktoren hat, die in direktem Zusammenhang mit dem System stehen wie beispielsweise die "[...] Räume, Technik, Rechner, Wege, mehr Technik, mehr Material."([Sie96]) Sie beziehen sich allesamt auf die Struktur des Systems mit deren Änderung ebenfalls bestehende Regeln assimiliert werden müssen.

Mit der Vergrößerung von Systemen erfolgt eine Veränderung bis in praxisrelevante Bereiche. Bei den oben genannten Beispielen wäre das der Bereich

des direkten Kontakts zu Angestellten, Kunden oder Studenten in Form eines Vortrages. Je mehr Teilnehmer desto erforderlicher wird die Präzision der Sprache: "[...] bei einem Vortrag vor vielen müssen Sie Ihre Worte gesetzter wählen als im Gespräch mit den Kollegen. Schließlich ändern sich Gefühle und Bewertungen: Je mehr Zuhörer ihre Vorlesung hat, desto mehr müssen Sie über der Sache stehen, [...] desto mehr müssen Sie sich zwingen, gut zu unterrichten"([Sie96]).

Das Wachstum eines Systems erfordert demzufolge ein ausgleichendes Verhalten aller Beteiligten. Dieser Ausgleich erfolgt über Hilfsmittel, Verhaltensregeln, die oben angesprochene Sprache sowie ein kontrollierteres Fühlen und ein angestrengteres *Wollen*. Das bedeutet, sie "[...] müssen materiell, intellektuell und emotional mehr hineinstecken; sonst wird die Kommunikation beeinträchtigt, die Arbeit gestört, das System gerät aus den Fugen"([Sie96]). Siefkes folgert daraus, daß ein System, das die zuletzt genannten Auswirkungen aufweist, ein zu großes System ist.

Neben den zu großen Systemen gibt es für ihn auch die kleinen Systeme als Mittelweg, die ebenfalls Mängel aufweisen. Jedoch ist damit eine eindeutige Bewertung großer wie kleiner Systeme nicht gegeben. Und somit hat jedes System seine spezifischen Schwächen.

Es stellt sich nun die Frage nach dem finden des *richtigen* Systems. "Systeme werden nicht dadurch klein, daß man sie zerlegt, reduziert oder sonstwie strukturell verkleinert. Systeme werden klein, wenn die Beteiligten sie nach ihren materiellen, intellektuellen und emotionalen Bedürfnissen gestalten"([Sie96]).

Ein System kann also "klein" sein, wenn die beteiligeten Personen durch ihre materiellen Möglichkeiten und die intellektuellen und emotionalen Fähigkeiten es als *klein* ansehen. Systeme sind also dadurch (zu) komplex, daß die beteiligten Personen sie als (zu) komplex ansehen, da ihnen die Möglichkeiten und Fähigkeiten fehlen es "klein" zu machen.

Daraus folgt, daß eine Schulung der intellektuellen Fähigkeiten und die Bereitstellung von geeigneten Materialien Systeme *klein* machen kann.

Wird dieses aus dem Blickwinkel des Informatikunterrichts betrachtet, kann das bedeuten, daß zum tieferen Verständnis eines Informatiksystems dieses “klein” gemacht werden sollte.

Sind aber die Informatiksysteme **zu klein**, kann das System weniger leisten und der Nutzen des Systems ist gering. Sind sie zu komplex, sind sie für die beteiligten Personen nicht handhabbar bzw. durchschaubar.

Unklar ist bisher, was überhaupt ein Informatiksystem ist.

> Informatiksysteme beinhalten soft- und hardwaretechnische Komponenten, die ihrerseits fundamentale Methoden und Ideen der Informatik und zugleich in digitaler Form materialisierte Modelle eines Realitätsausschnitts repräsentieren. Diese Modelle werden in Modellierungsprozessen entwickelt und bilden ein wesentliches Element der Systemgestaltung.[MSH99]

Es “müßte gelingen, die Komplexität einer Software eines Informatiksystems für den unterrichtlichen Einsatz didaktisch zu reduzieren, ohne daß die Einführung objektorientierter Sichtweisen beim verwendeten Beispiel zur Trivialität verkommt”[MSH99].

Ein der unterrichtlichen Situation angepaßtes “kleines Informatiksystem” scheint ein gangbarer Weg. Es könnten insofern an einem “kleinen Informatiksystem” die Erschließung von Informatiksystemen im Großen geübt und vermittelt werden.
Nachdem nun ein Vorschlag bezüglich des Inhalts gemacht wurde, stellt sich die Frage nach dessen Umsetzung.

2.2 Dekonstruktion

Die emotionalen und intellektuellen Fähigkeiten und auch die materiellen Gegebenheiten, z.B. der Entwickler, Auftraggeber und Anwender, sind es,

die "die Gestaltungsentscheidungen und die Nutzung von Informatiksystemen bestimmen. Diesen impliziten Prozessen nachzuspüren, die sich in der Software materialisiert haben, ist neben der formal-technischen Analyse von Informatiksystemen eine weitere wichtige Aufgabe der didaktischen Analyse einer systemorientierten Didaktik der Informatik. Dekonstruktion kann möglicherweise derartige Fragestellungen für den Informatikunterricht erschließen" [Mag00].
Nach der Frage nach dem *wie* stellt sich die Frage nach dem *womit*.

2.3 Tools zur Visualisierung von Ablaufprozessen objektorientierter Software

Dirk Steinkamp behandelte in seiner Diplomarbeit "Informatik-Experimente im Schullabor"[Ste99]. Er formuliert Anforderungen an eine Experimentierumgebung für den Informatikunterricht und schlägt ein DMI(direct manipulation interface) vor. Ein DMI stellt "das experimentelle Untersuchungsobjekt in den Mittelpunkt [. . .] und [entspricht] darüber hinaus der heute weitverbreiteten Art der Benutzungsschnittstelle"[Ste99, S.23]. Eine Tool zur Visualisierung von Ablaufprozessen objektorientierter Software sollte diese Eigenschaften haben.

Anforderungen an ein DMI:

- Der Benutzer soll mit einem objektorientierten Modell arbeiten können,
- die Bedienungssyntax soll möglichst durchgängig der Form Objektselektion → Funktionsaufruf entsprechen (OF-Syntax),
- die zu bearbeitenden Objekte sollen ständig sichtbar sein,
- für häufig benötigte Funktionen sollten Piktogramme vorhanden sein, die das schnellere Auffinden von Funktionen erleichtern,
- Objekte sollen durch eine direkte Referenzierung auswählbar/veränderbar sein (Verwendung von Zeigeoperationen, z. B. Bewegen, Ziehen, Klicken mit der Maus), wobei symbolisches Referenzieren durch Eintippen von Namen entfällt,
- auf jede Eingabeaktion des Benutzers soll eine unmittelbare, sichtbare Rückmeldung des Systems erfolgen (z.B. Verschieben von Objekten),

- die Interaktion soll schrittweise, einstufig und benutzergesteuert ablaufen.

[Ste99, S.23]

“Ein wesentliches Element des Experimentierens ist das dynamische Moment” [Ste99, S.26]. Es geht nicht nur darum einen statischen Zustand zu untersuchen, sondern auch mit dem Gegenstand zu interagieren.
Tools zur Visualisierung von Ablaufprozessen objektorientierter Software sollten diese Eigenschaften haben. Im Verlauf dieser Arbeit wird gezeigt, daß Mr. Dobs diese Anforderungen erfüllt.
Da diese Tools für den Informatikunterricht –und nicht nur dort– neu sind, folgt nun die Beleuchtung des Themas aus softwaretechnischer Sicht.

3 Softwaretechnische Perspektive

Im Folgenden werden einzelne Aspekte, die für das Thema dieser Arbeit aus fachinformatischer Sicht relevant sind, beleuchtet. Im Hintergrund dieses Kapitels steht ein abstrakter Softwareentwicklungsprozeß; an dem einzelne Probleme aufgezeigt werden. Der Schwerpunkt dieses Kapitels liegt auf dem Aspekt “Testen und Prüfen von Software”, da eine Visualisierung von Ablaufprozessen, in Bezug auf einen Softwareentwicklungsprozeß, dort zu verorten ist.

3.1 Softwareentwicklungsprozeß

Ein Softwareentwicklungsprozeß[2] hat zum Ziel Software, also von Computern ausführbare Programme, zu erstellen. Die Software sollte die allgemeinen Softwarequalitätsmerkmalen[3] haben. Die Wege zu diesem Ziel sind vielfältig und im Einzelnen sehr unterschiedlich. Ist das Vorhaben bzw. die Software sehr kompex und es sind mehrere Personen beteiligt, so ist es ratsam den Softwareentwicklungsprozeß entlang eines Vorgehensmodells zu gestalten.

> Diese Vorgehensmodelle müssen die Gesamtheit aller Aktivitäten, Ergebnisse, Rollen und Techniken für die Entwicklung objektorientierter Anwendungen beschreiben und dabei auch verschiedene Tätigkeitsbereiche wie die Qualitätssicherung, das Konfigurationsmanagement oder das Projektmanagement umfassen.[NS99, S.168]

Diesen (objektorientierten) Vorgehensmodellen[4] ist gemein, daß sie den Softwareentwicklungsprozeß in einzelne Phasen unterteilen[NS99, S.168].

> Eine Phase stellt eine Gruppierung von Aktivitäten zu einer plan- und kontrollierbaren Einheit dar. Die meisten Vorgehensmodelle verwenden die

[2] Gemeint ist –für diese Arbeit– ein objektorientierter Softwareentwicklungsprozeß

[3] nach DIN 66272/ISO 9126: Funktionalität, Zuverlässigkeit, Benutzbarkeit, Effizienz, Änderbarkeit und Übertragbarkeit

[4] z.B. Worldwide Solution Design and Delivery Method (WSDDM) der IBM, Fusion von Hewlett Packard, V-Modell der IABG, Rational Unified Process (RUP) von Rational, Perspective von Select Software Tools, OPEN Process Specification des OPEN-Konsortiums und AE-Modell des Informatikzentrums der Sparkassenorganisation(aus [NS99])

> zeitliche Dimension, um den Softwareentwicklungsprozeß in unterschiedliche Abschnitte oder Phasen einzuteilen. Dem klassischen Wasserfallmodell liegt ein sequentielles Durchlaufen aller Phasen zugrunde. Die aus einer Phase resultierenden Ergebnisse fallen kaskadenartig in die nächste Phase hinein.
> [. . .] in objektorientierten Vorgehensmodellen wird häufig eine Phasenstruktur verwendet, um gegenüber den Nutzern bzw. dem Auftraggeber die zeitliche Abfolge der einzelnen Vorgehensschritte auf einer höheren Abstraktionsstufe zu verdeutlichen. Beispiele für verwendete Phasennamen sind etwa **Voruntersuchung, Systemanalyse, Entwurf, Erzeugung und Einführung**. Die einzelnen Vorgehensmodelle unterscheiden sich sowohl im Zuschnitt und der Anzahl der Phasen als auch in der Festlegung ihrer zyklischen Abhängigkeiten.[NS99, S.168]

Für die Visualisierung von Ablaufprozessen objektorientierter Software ist es unerheblich, welches objektorientierte Vorgehensmodell verwendet wurde, da die Software immer am Ende des Prozesses steht. Zur Argumentation wird aber ein Modell benötigt, um u.A. Probleme des Softwareentwicklungsprozesses aufzudecken, die Auswirkungen auf das Verständnis von Ablaufprozessen objektorientierter Software haben. Dieses Modell baut auf den Phasennamen aus [NS99] auf. Das benötigte eher neutrale Metamodell definiere ich durch folgende Phasen:

Voruntersuchung: Es werden funktionale Anforderungen in Form des Anwendungsfallmodells (use-case-model) und der groben Spezifikation der Anwendungsfälle dokumentiert.

Systemanalyse: Aus den Voruntersuchungen wird ein Systemmodell erstellt, was die Anwendungsfälle umfaßt und in Beziehung zueinander setzt. Es dient zur Vorbereitung für den Entwurf.

Entwurf: Aus dem Systemmodell wird ein formaleres Klassenmodell entworfen. Das Klassenmodell ist die allgemeinere (verfeinerte) Variante des Systemmodells. In ihm müssen die Beziehungen zwischen den Systemkomponenten “formal” beschrieben und erkennbar sein (Assoziationen, Schnittstellen, . . .).

Erzeugung: Das Klassenmodell wird hier in eine Programmiersprache überführt. Des weiteren werden die Algorithmen festgelegt und implementiert.

Einführung: Die Software wird in die Nutzung überführt, d.h. die Klassen werden in eine maschinenausführbare Form übersetzt(⇒ Software) und die Software wird getestet (Wartung).

Im Fokus dieser Arbeit liegen die Phasen "Erzeugung" und "Einführung". Im Speziellen liegt der Schwerpunkt auf *Prüfung und Test von Software*(⟶3.6). Da beim Test von Software die Voruntersuchung, im speziellen die "use-cases", eine zentrale Rolle spielen, werden diese –im Rahmen des Themas– näher beleuchtet(⟶3.2). Um objektorientierte Software zu verstehen und zu diskutieren, muß der Begriff Objekt näher untersucht werden, um z.B. Mißverständnisse zu vermeiden(⟶3.4). Da eine objektorientierte Software aus Klassen generiert wird, sind die Beziehungen zwischen Objekten und Klasse näher zu untersuchen(⟶3.5).

3.2 Voruntersuchung

Im ersten Schritt eines Softwareentwicklungsprozesses muß "die Sache" bzw. "die Sachlage" untersucht werden. Die Entwickler der Software müssen über alle für die spätere Software relevanten Fakten informiert sein, darunter befinden sich zum einen die "Gegenstände" und zum anderen die Relationen. Soll z.B. eine Simulation eines Kiosks (siehe Seite 19) geschrieben werden, müssen sich die Entwickler über bestimmte Tatsachen im Klaren sein. Beispielsweise muß den Entwicklern das Warenwirtschaftssystem des Kiosks bekannt sein und sie müssen es verstanden haben. Um dieses zu erreichen, können einzelne Situationen in einem Kiosk "durchgespielt" werden (Heinz möchte eine Banane kaufen und . . .). Diese "durchgespielten" Situationen werden Anwendungsfälle genannt.

Synonym/Akronym/Übersetzung: use case [engl.]
Ein Anwendungsfall beschreibt in Klartext eine Menge konsistenter und zielgerichteter Interaktionen von Akteuren mit einem System, an deren Ende ein definiertes Ergebnis entstanden ist. Ein Anwendungsfall wird stets

> durch einen Akteur initiiert und ist eine komplette, unteilbare Beschreibung. Anwendungsfälle beschreiben das gewünschte externe Systemverhalten aus der Sicht und in der Sprache der Anwender und somit Anforderungen, die das System zu erfüllen hat. Beschrieben wird, was es leisten muß, aber nicht wie es dies leisten soll.[Oes]

Anwendungsfälle sind wenig formell und können verschiedene Ausprägungen haben. Sie können:

- eine IST-Situation beschreiben (Ist-Anwendungsfall)
- einen SOLL-Zustand beschreiben (Soll-Anwendungsfall)
- eine Essenz beschreiben (Essentielle Anwendungsfälle)
- nur die durch Software zu unterstützenden Sachverhalte beschreiben (System-Anwendungsfälle)
- auch außerhalb der Software und allgemeine geschäftliche Anwendungsfälle darstellen (Geschäfts-Anwendungsfälle)
- ganz einfach verwendet werden (Name + Kurzbeschreibung)
- semi-formal und detailliert verwendet werden (bis zu akteurspezifischen Schrittbeschreibungen)

[Oes]

Im allgemeinen können nicht alle möglichen Anwendungsfälle erfaßt werden. Es ist zwar theoretisch möglich alle definierten Anwendungsfälle einzeln umzusetzen. Dieses birgt aber verschiedenste Probleme in sich. Es würde u.a. bedeuten, daß der Programmieraufwand sehr groß ist, da jeder isolierte Fall einzeln implementiert werden müßte. Des weiteren kann es dazu führen, das die Software zu eingeschränkt –auf die Anwendungsfälle– ist. Soll z.B. zusätzlich zu den Waren Banane, Apfel und Wasserflasche noch ein Schokoriegel verkauft werden, muß die Software entsprechend erweitert werden. Eine abstraktere Beschreibung von Apfel, Banane, Wasserflasche und Schokoriegel wäre z.B. Ware, denn sie sind –bezogen auf das Beispiel Kiosk– alles Waren. Dieser Abstraktionsschritt ist Aufgabe der folgenden Phasen.

3.3 Systemanalyse

Nachdem in der ersten Phase einige der Anforderungen an die spätere Software exemplarisch definiert sind, müssen weitere Schritte zur Abstraktion

des Modells vollzogen werden, um zu einem Softwareprodukt zu kommen. Eine Software, die als allgemeine "Klasse" Ware und als spezielle Ausprägungen Banane, Apfel, usw. anbietet, ist "schneller"(s.u.) und konsistenter umzusetzen. Eine Handlung (z.B. Verkauf) ist nun nicht mehr für jede einzelne Ware zu programmieren (verkaufeApfel() oder verkaufeBirne()), sondern kann von der allgemeineren Handlung verkaufeWare() getätigt werden.
Eine Möglichkeit "zu verallgemeinern" bzw. zu abstrahieren besteht in der Erstellung von CRC[5] Karten[Amb98]:

> CRC cards are an aggressively informal technique, they also encourage discussion. In a CRC session the developers will crowd around a table each person picking up a card as they describe how the class participates in some use case. By giving the developers something to touch that represents the class, you make it easier talk through the designs. By accenting responsibilities instead of data and method you help them get away from dumb data holders and ease them towards understanding the higher level behavior of the class.[Fow, S.3]

Durch CRC-Karten kann ein System analysiert werden. Richtig eingesetzt, können die Ablaufprozesse auf höherer Ebene erschlossen werden. Dieses ist der Phase "Systemanalyse" zuzuordnen.

Zur Überprüfung der späteren Software können sie herangezogen werden um zu testen, ob die Spezifikation umgesetzt wurde.

Die Anwendungsfälle und die Systemanalyse präzisieren die Spezifikation der Software und führen zu einem "höheren Verständnis" vom Gegenstand derselben.

Der Begriff "Objekt", als Synonym für ein Element eines Systems (⟶2.1), kann im Softwareentwicklungsprozeß in jeder Phase verwendet werden. Er ist jeweils mit unterschiedlicher Bedeutung besetzt und muß, um Mißverständnisse zu vermeiden, näher beleuchtet bzw. differenziert werden.

[5] Class Responsibility Collaborator

3.4 Objekte

Was ist ein Objekt?

(I)

> **Objekt**[lat.: “das Entgegengeworfene”]
>
> 1. (a) Gegenstand, mit dem etwas geschieht od. geschehen soll [. . .]
>   (b) unabhängig vom Bewußtsein existierende Erscheinung der materiellen Welt, auf die sich das Erkennen, die Wahrnehmung richtet(Philos.);
>   (c) aus verschiedenen Materialien zusammengestelltes plastisches Werk der modernen Kunst (Kunstw.)
> 2. Satzglied, das von einem Verb als Ergänzung gefordert wird[. . .]
> 3. (a) Grundstück, Wertgegenstand, Vertrags-, Geschäftsgegenstand [. . .]
>   (b) [. . .] Gebäude
>
> [Dud90, S.542]

(II)

> **Objekt**
> Ein Objekt ist die konkrete Ausprägung des durch eine Klasse definierten Datentyps. Es hat einen inneren Zustand, der durch Attribute in Form von anderen Objekten oder Elementen der in der Programmiersprache vorgegebenen Datentypen dargestellt wird. Der Zustand kann sich durch Aktivitäten des Objekts ändern, also durch Ausführung von Operationen auf Objektdaten. Jedes Objekt hat eine Identität, so daß auch gleiche Objekte unterscheidbar sind. Im Ablauf eines Programms werden Objekte erzeugt (und wieder gelöscht), die aufgrund des Empfangs einer Botschaft hin aktiv werden. Die Menge aller möglichen Botschaften für ein Objekt heißt Schnittstelle.[Bre96, S.428]

Welcher Objektbegriff ist nun für einen Softwareentwicklungsprozeß relevant? Oder brauchen wir verschiedene Objektbegriffe? In (II) geht es primär um einen Objektbegriff, der Objekte am Ende eines Softwareentwicklungsprozesses (zur Laufzeit[6] des Programms) beschreibt. Diese Objekte stellen

[6]Zeitraum zwischen dem Start eines Programms und seinem Ende.

keine Gegenstände oder sonstige durch (I), im engeren Sinne, erfaßte Objekte dar.

Am Anfang des Softwareentwicklungsprozesses (erstellen von Anwendungsfällen) stehen Objekte, die zweckgebunden, d.h. im Hinblick auf das zu erstellende (Software-)Produkt, abstrahiert wurden. Sie sind nicht real und eher Metaphern realer Objekte. Hier paßt die Definition (I)1b. Diese **Metaobjekte**[7] werden zum Aufstellen von Anwendungsfällen benutzt.

Nun folgt ein Abstraktionsprozeß[8], der z.B. verschiedene Metaobjekte zu Objekten höherer Ordnung zusammenfaßt. Diese Objekte höherer Ordnung stellen später (Objekt-)Klassen dar. Diese Klassen bilden den Programmcode bzw. den Ursprung(sourcecode) für die Software. Nach Übersetzung des Programmcodes in eine maschinenverständliche Form, kann die Software getestet werden. Beim Test der Software, also zur Laufzeit des Programms, werden aus den Klassen Objekte generiert. Diese Objekte sollten die Eigenschaften der Objekte höherer Ordnung haben. Dieser Objekttypus ist in (II) beschrieben. Im Folgenden wird dieser Objekttypus als generiertes Objekt bezeichnet.

In Tabelle 1 sind diese unterschiedlichen Ausprägungen in Bezug auf die einzelnen Phasen eines Softwareentwicklungsprozesses dargestellt.

	Objekt (I)		
$\Longrightarrow$	Metaobjekt	$\longrightarrow$	Voruntersuchung
$\Longrightarrow$	Objekt höherer Ordnung	$\longrightarrow$	Systemanalyse
$\Longrightarrow$	Klasse	$\longrightarrow$	Entwurf / Erzeugung
$\Longrightarrow$	generiertes Objekt (II)	$\longrightarrow$	Einführung

Tabelle 1: Objektevolution

Zur Verdeutlichung der unterschiedlichen Begriffe ein kleines Beispiel:

[7] individuelle Vorstellung vom ursprünglichen Objekt

[8] Hiermit ist ein Prozeß gemein, der die komplexen Zusammenhänge strukturiert, ohne "Sinnverlust" vereinfacht und überschaubar macht.

Aufgabe: Für einen Schulkiosk(insbesondere für das Waren-Wirtschaftssystem eines Schulkiosks) soll eine Simulation[9] geschrieben werden.
Jede der am Softwareentwicklungsprozeß beteiligten Personen hat eine Vorstellung von z.B. einem Regalfach((I)1b). D.h. jede Person operiert mit einer subjektiven Ausprägung des Objektes Regalfach. Diese subjektiven Interpretationen von einem Regalfach werden als Metaobjekte bezeichnet. Nun können einzelne Anwendungsfälle mit diesen Metaobjekten durchgespielt werden, um deren Funktion, Wirkungsweise,Verantwortlichkeiten usw. auseinanderzunehmen und zu objektivieren[10]. Diese Anwendungsfälle sollten nun abstrahiert werden und zu einem Objekt höherer Ordnung führen. Diese Objekte höherer Ordnung könnten z.B. im CRC-Karten Modell [Amb98] eine Karte repräsentieren. Kommt es nun zum Entwurf und zur Erzeugung, werden Klassen aus den Objekten höherer Ordnung entworfen. Die Summe dieser Klassen bilden den Programmcode der Software. Nach Übersetzung des Programmcodes in eine maschinenverständliche Form, kann die Software Schulkiosk ausgeführt werden. Zur Laufzeit kommt es dann zur Bildung von generierten Objekten "Regalfach".

Meiner Ansicht nach bildet eine nicht trennscharfe Formulierung der verschieden Ausprägungen des Begriffs Objekt einen Kern der Probleme in den Diskussionen im Softwareentwicklungsprozeß. Zur Begründung einzelner Entscheidungen, die in einer Phase getroffen wurden, müssen zum Teil konkretere Objekte –aus vorherigen Phasen– herangezogen werden. Eine eindeutige kontextsensitive Zuordnung des Begriffs, sofern *nur* "Objekt" verwendet wird ist im allgemeinen nicht möglich und ruft Mißverständnisse hervor.
In Anhang B ist eine Diskussion einer Newsgroup aus dem Internet wiedergegeben. Die Diskussion krankt an Mißverständnissen, die auch auf die verschiedenen Ausprägungen eines Objektes zurückzuführen sind. Schon bei

[9] Nachahmung eines Kiosks: Es sollen nur bestimmte Abläufe nachgeahmt bzw. simuliert werden.

[10] [Dud90, S.542]: [...] etwas von subjektiven, emotionalen Einflüssen befreien. [...]

einer solch simplen Fragestellung gibt es Probelme mit der Kommunikation bzw. dem Verständnis untereinander.
Im komplexeren Fall, dem Softwareentwicklungsprozeß, treten alle Ausprägungen des Begriffs Objekt auf. Sie verursachen –insofern– Probleme in der Diskussion und damit im Kern des Softwareentwicklungsprozesses. Insbesondere gibt es Probleme in der Unterscheidung von Klassen und generierten Objekten. Dieses führt zu folgendem Punkt.

3.5 Klassen und Objekte

Ein Problem innerhalb eines Softwareentwicklungsprozesses stellt die Unterscheidung zwischen Klassen und generierten Objekten dar. Oder wie Hubwieser es ausdrückt, ist "ebenso sorgfältig [...] bei der Implementierung auf die saubere begriffliche Unterscheidung von Klassen als abstrakten Beschreibungen [...] und Instanzen als konkreten Ausprägungen von Klassen [...] zu achten"([Hub99]). Bei der Erstellung eines objektorientierten Programms werden Klassen programmiert. Wird das Programm ausgeführt, werden aus den Klassen Objekte generiert (Exemplare dieser Klassen). In der Diskussion über ein Programm kommt es zu einem sprachlichen und argumentativen Problem (abhängig vom Blickwinkel): Nur zur Laufzeit treten Fehler zutage bzw. sie sind "zu sehen". Kommt es zur Diskussion bzgl. dieses Fehlers, ist ein gedanklicher Wechsel zwischen den generierten Objekten und dem Quelltext zu vollziehen. Der Blickwinkel ändert sich. Semantische Fehler werden im Quelltext verursacht, treten aber erst zur Laufzeit auf. Ursache und Wirkung spielen nicht auf dem gleichen Spielfeld, sind aber miteinander verbunden. Befinden wir uns auf der Quelltextebene sind wir in der "was passiert dann" Situation. Wir erzeugen Klassen und müssen gleichzeitig die später erzeugten Objekte berücksichtigen. Befinden wir uns auf der Laufzeitebene, sind wir in der "haben wir das so gewollt?"-Situation: Verhalten sich die generierten Objekte so, wie wir uns das vorher vorgestellt und programmiert haben? In einer klaren Unterscheidung dieser beiden Fälle liegt der Kern des Problems.

> Es muß unterschieden werden zwischen der Beschreibung von Objekten und den Objekten selbst[Bre96, S.2].

Daß selbst Fachliteratur Probleme mit einer klaren Trennung hat, sollen die beiden folgenden Textauszüge zeigen:

1. A **class** is an named collection of fields that **hold data** values and methods that operate on those values.[Fla99, S.61]

   Dieser Textauszug verdeutlicht ein wenig besprochenes Problem. Das Zitat vermischt zwei Ebenen, die Klassen- und die Objektebene. Klassen sind, solange es noch kein Exemplar von ihnen gibt, abstrakt. Deshalb können sie auch keine Operationen auf ihre Daten anwenden - sie habe keine. Erst wenn ein Exemplar von ihnen existiert, können Operationen auf die nun vorhandenen Daten angewendet werden. Dies geschieht nicht im Quelltext, sondern zur Laufzeit.

2. Just as a cell ist the smallest unit of life that can survive and reproduce on its own, a class ist the smallest unit of Java code that can stand alone.[Fla99, S.61]

   Selbst dieser Vergleich ist problematisch. Es wird hier von real existierenden Objekten gesprochen und mit einer abstrakten Beschreibung verglichen. Außerdem kann eine Klasse nicht "leben" oder sich selbst reproduzieren. Nur ein Exemplar der Klasse "lebt". Klassen können Anweisungen enthalten, sich selbst zu reproduzieren, aber erst ein Exemplar(Objekt) der Klasse ist in der Lage dies durchzuführen.

Folgerung:

Die Mischung der Klassen- und Objektsicht ist problematisch und kann zu Fehlvorstellungen führen.

Beim Softwareentwicklungsprozeß von den "use cases" (Anwendungsfällen) bis zu den Klassen kann diese Fehlvorstellung Probleme aufwerfen. Dieser Prozeß ist primär ein Abstraktionsprozeß und versucht aus den Anwendungsfällen eine Klassenstruktur zu generieren. Es wird also versucht, aus

“realen Objekten” und ihren Beziehungen Klassen zu generieren. Am Ende des Softwareentwicklungsprozesses steht die Ausführung des Programms, bei dem aus den Klassen Objekte, diesmal computergeneriert, erzeugt werden. Findet keine klare Trennung zwischen Objekten und Klassen statt, kann es vorkommen, daß beim Abstraktionsprozeß zu wenig abstrahiert bzw. verallgemeinert wird und insofern z.B. Übertragbarkeit und Funktionalität des Programms zu gering sind. Es kann ein Programm entstehen, welches nicht die allgem. Softwarequalitätsmerkmale aufweist.

Ein weiteres Problem stellt die letzte Phase des Softwareentwicklungsprozesses (Einführung) dar. Bis zur Phase “Entwurf”(Erstellung von Klassen) befinden wir uns in einem Prozeß, der mehr und mehr abstrahiert. In der letzten Phase, die vielfach vergessen wird, wenn es um Softwareentwicklungsprozesse geht, werden die erstellten Klassen getestet. Es werden konkrete Objekte aus den abstrakten Klassen generiert. Dieser Schritt geht also “in die andere Richtung”. Im wahrsten Sinne “back to the roots”: Zurück zu konkreteren Objekten. Diese sind generiert und sollen das Verhalten der Metaobjekte simulieren. Man könnte hier von einer Art Parabel sprechen (siehe Abbildung 2), die bei Metaobjekten beginnt und bei generierten Objekten endet.

In den seltensten Fällen sind alle Tests erfolgreich und das Programm muß korrigiert werden. Es muß also wieder in die Klassensicht “umgeschaltet” werden (siehe Abbildung 3). Dieser Prozeß muß solange wiederholt werden, bis alle Tests erfolgreich sind.

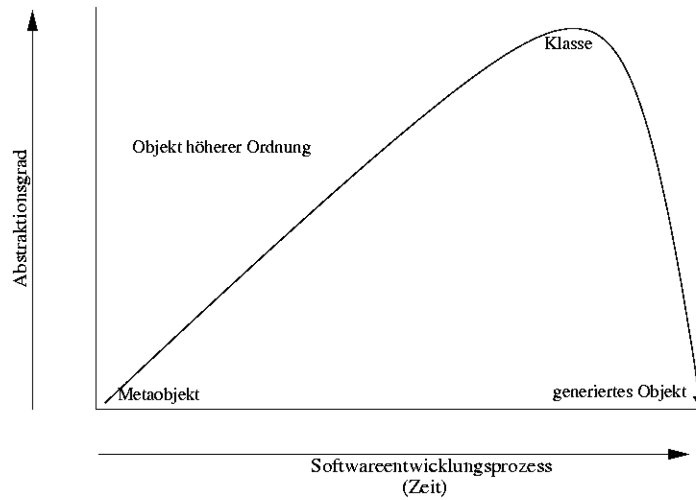


Abbildung 2: Objektevolution (2)

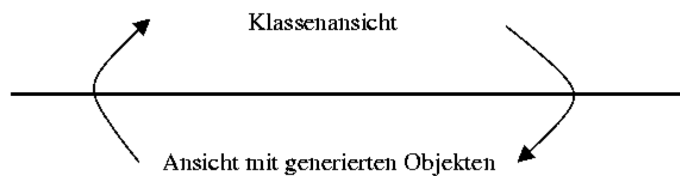


Abbildung 3: Sichtwechsel

Dieser Prozeß alterniert zwischen den beiden Ebenen und stellt hohe kognitive Anforderungen –wie oben besprochen– an die Entwickler (im Unterricht).

Mit Softwaretools zur Visualisierung von Ablaufprozessen objektorientierter Software sollte die Objektsicht gestärkt und des weiteren eine klare Trennung zwischen Klassen und deren Objekten vollzogen werden.

3.6 Prüfung und Test von Software

Vor der Auslieferung[11] bzw. der Finalisierung von Software, muß ein Programm auf Korrektheit geprüft werden. Für die Prüfung von Software sind zwei Fragestellungen von Bedeutung

> - Wurde in [jeder] [. . .] Phase des Software-Erstellungsprozesses "richtig" gearbeitet, d.h. wurden die Phasenvorgaben konsistent, vollständig und frei von unnötiger, leerer Redundanz in die Phasenergebnisse umgesetzt ("V e r i f i k a t i o n")?
> - Wurde das "richtige" System erstellt, d.h. entspricht das fertiggestellte System den Nutzungsanforderungen ("Va l i d a t i o n")?
>
> [Bel98]

Verifikation sollte eine Selbstverständlichkeit für ein Entwicklungsteam sein. Aber auch die Validation ist Aufgabe des Entwicklungsteams. Denn

> [. . .] die [. . .] Komplexität der Software und deren Prüfung, berechtigen aber keineswegs die "Bananas"-Methode, deren Einsatz in der Praxis vielfach beobachtet wird: "Software reift bei dem Kunden nach!". Denn diese (Un)Methode verursacht hohe Wartungskosten von Programmen, die als "fertig und ausgetestet" ausgeliefert werden, jedoch noch voller Fehler stecken. In diesen Fällen "ertestet" der Anwender etwa die Hälfte aller Mängel. Mit anderen Worten, der Kunde wird zum unbezahlten "Testingenieur", oder gar zum "Versuchskaninchen".[Bel98]

Deshalb ist es wichtig vor der Auslieferung die Korrektheit der Software zu prüfen. Hierzu werden im wesentlichen drei Arten von Testverfahren angewandt.

> Wenn die dynamische Prüfung von Software sich auf die Funktionsvielfalt und -vollständigkeit des erstellten Produktes bezieht, wird die Software als **Black-Box** betrachtet, in deren Struktur man nicht "hineinblickt". Man beobachtet die Eingabe- und Ausgabegrößen der Testläufe und entscheidet dann (entsprechend der Systemspezifikation), ob aufgrund der

[11] Synonym für "Software ist fertig und kann von anderen benutzt werden".

> zufriedenstellenden Ergebnisse eine Abnahme erfolgen kann (Black-Box-Test oder spezifikationsorientierter Test). Wichtig ist dabei, daß alle relevanten Eigenschaften des abzunehmenden Systems durch solche Ein-/Ausgabegrößen charakterisiert werden.
> Betrachtet man beim Testen das vorliegende Programm primär anhand seiner Struktur, so spricht man von **White-Box**-Test (auch Glass-Box-Test oder Strukturtest genannt): Das Programm ist hier "gläsern", man möchte alle relevanten Teile, z.B. Verzweigungen, Pfade von Anweisungen usw. prüfen. Bei dieser Testart bildet die vorliegende Implementation des Prüflings die Grundlage des Testvorgangs. Dies kann zwingend notwendig sein, weil z.B. keine Spezifikation vorhanden ist; mit anderen Worten, bei Strukturtests geht man oft implementierungsorientiert vor.
> Natürlich ist eine Mischung der beiden Testarten (**Grey-Box**-Test) möglich.[Bel98]

Zur Durchführung dieser Tests gibt es verschiedene Arten der Durchführung und verschiedene Umgebungen zur Durchführung. Im Folgenden werden einzelne besprochen, wobei die für diese Arbeit relevanten Arten ausführlicher behandelt werden.

3.6.1 Testlauf

Wird das Programm schlicht gestartet, kann ein Black-Box-Test durchgeführt werden. Hierbei werden –exemplarisch– signifikante Eingaben gemacht und beobachtet, wie sich das Programm verhält. Es soll primär überprüft werden, ob das Programm den Spezifikationen bzw. den Ansprüchen genügt. Des weiteren kann von Unkundigen[12] überprüft werden, was das Programm leistet. Interna des Programms, wie die Objektstruktur, Ablaufprozesse oder Algorithmen bleiben verborgen.

3.6.2 Theoretische Prüfung

Hierbei wird der Quelltext des Programms "top down" auseinandergenommen. Danach werden anhand theoretischer Prüfungen, wie z.B. mit Schleifeninvarianten oder Kompression, einzelne Teile des Programms auf Korrektheit überprüft. Sind diese Fragmente korrekt, werden diese wieder "bottom

[12] d.h. Personen, die die internen Strukturen des Programms nicht kennen, aber den Nutzen bzw. Sinn des Programms kennen.

up"zusammengesetzt und wieder getestet. Dieser Prozeß setzt sich bis zur obersten Ebene fort. Sind alle Prüfungen erfolgreich, ist das Programm korrekt. Da größere Programme für solche Tests zu komplex sind, werden diese Prüfungen in der Regel nur auf "Funktionenebene" durchgeführt. Im Allgemeinen kann es kein Verfahren geben, um Programme zu testen($\Leftarrow$ theoretische Informatik). Im Einzelfall können aber spezielle Tests durchgeführt werden. Da hier nur am Quelltext gearbeitet wird, kommen keine Objekte und deren Beziehungen vor. Ablaufprozesse sind, da es sich um Quelltext handelt, nur schwer zu erkennen. Bei diesem Test handelt es sich um einen White-Box-Test.

3.6.3 Debug session – klassisch

"Debug sessions" sind Laufzeitsitzungen in denen ein Programm mit "debug informations" ausgeführt wird.

Diese Sitzungen werden in speziellen Umgebungen durchgeführt. Dies kann z.B. ein einzelner Debugger wie z.B. der *gdb*[13] sein oder eine "all in one" Entwicklungsumgebung wie z.B. *Visual Studio*[14] oder *Forté* [15].
"Debug informations" sind Informationen, die im Programm enthalten sind, aber zur *simplen* Ausführung des Programms nicht benötigt werden. Sie liefern in der Entwicklungsphase bei ungewollten (Fehlern) oder gewollten ("breakpoints") Programmunterbrechungen zusätzliche Informationen. Diese Informationen liefern –je nach Entwicklungsumgebung bzw. Compiler– verschiedene hilfreich gedachte Informationen über den momentanen Programmstatus. Zum Beispiel kann eine Zeile aus dem Quelltext ausgegeben werden, an der das Programm unterbrochen wurde oder sogar der Callstack[16] des Programms. Im Folgenden werden einige Eigenschaften bzw. Funktionen von Debuggern dargestellt.

[13] The GNU Debugger

[14] Eine Entwicklungsumgebung für mehrere Sprachen (C++,Java,Basic,...) mit integrierten Debugger

[15] Eine Entwicklungsumgebung für Java mit integriertem Debugger

[16] Ein Stack, in dem vermerkt ist, von welchen Objekten bzw. Funktionen die aktuelle Funktion aufgerufen wurde.

3.6.3.1 Verhalten bei Fehlern

Sofern ein Fehler in einer “debug session” auftritt, bricht das Programm ab. Anhand der “debug informations” kann der Debugger die Zeile und die Datei identifizieren, die den Abbruch hervorgerufen hat, und diese Informationen ausgeben. Da es sich bei diesen Fehlern nicht um syntaktische handelt, enthält in der Regel nicht die ausgegebene Quelltextzeile den Fehler, sondern der vorher behandelte Code. Diese logischen Fehler sind schwerer zu finden und erfordern viel Erfahrung.

Beispiel:
Enthalte ein Programm ein Feld von begrenzter Größe(Array)

$$A = (a_i)_{i=0\ldots(n-1);n\in\mathbb{N}}$$

. Das Programm bricht mit einer “Index out of range”[17] Fehlermeldung ab. Welche Fälle können vorliegen?

1. i und/oder n entsprechen nicht der Spezifikation und werden statisch bestimmt.
2. i und/oder n entsprechen nicht der Spezifikation und werden zur Laufzeit bestimmt.
3. Der Speicher(der Maschine/Umgebung/...), der für das Anlegen des Feldes nötig ist, reicht nicht aus.

Sofern 1) zutrifft sind nur die Zahlenwerte an die Spezifikation anzupassen (trivial).
2) Sind i oder n nicht statisch (sie werden berechnet), kann die Fehlersuche komplexer werden, da ein logischer Fehler im Algorithmus vorliegt und dieser überprüft werden muß. Der Fehler liegt also nicht an der Stelle vor, an der das Programm abbricht. Es handelt sich um einen sogenannten Folgefehler, der an dieser Stelle zutage tritt.
Der dritte Fall ist *im allgemeinen* nicht zu beantworten. Er hängt mit

[17] Es wurde versucht auf einen nicht vorhandenen Feldeintrag zuzugreifen

den Gegebenheiten zusammen, die nicht auf logische Fehler im Programm zurückzuführen sind, sondern auf die Umgebung, in der das Programm läuft. Hier kann entweder etwas an den Gegebenheiten verändert oder das Programm kann, z.B. auf den begrenzten Speicher, angepaßt werden.

3.6.3.2 Callstack

Ein Hilfsmittel zum Aufspüren logischer Fehler ist der Callstack. Ein Callstack ist der aktuelle Stapel von Funktionen, die einander aufgerufen haben. Befindet sich das Programm –zur Laufzeit– in der Funktion C, die von B aufgerufen wurde und diese von A, so enthält der Callstack (A, B, C). Ist Funktion C abgearbeitet, und das Programm befindet sich in B, enthält der Callstack (A, B). In einem Callstack ist also der aktuelle Programmstatus bzgl. der Funktionsaufrufe und deren Hierarchie vermerkt. Die Funktion, die "'oben auf dem Stapel liegt", hat den aktuellen Fokus.
Zur Fehlerfindung in imperativen Programmen bzw. Programmteilen stellt der Callstack ein wichtiges Hilfsmittel dar, da bei imperativen Programmen die Funktionen im Vordergrund stehen und durch den Callstack deren Verknüpfung zur Laufzeit gezeigt wird. Zur Fehlerfindung in objektorientierten Programmen ist er im allgemeinen ungeeignet, da in ihnen die Objekte im Vordergrund stehen und der Callstack nur die Funktionen einzelner Objekte ausgibt. Angenommen: (im Quelltext) Klasse **K**(ind) wird von Klasse **V**(ater) abgeleitet. *dummy()* ist Memberfunktion von V und wird in **K** nicht überschrieben. Das Programm wird übersetzt und gestartet. Das Programm bricht in der Funktion dummy() ab. Die Funktion *dummy()* hat den Fokus. Nun stellen sich die Fragen: Zu welchem Objekt gehört die Funktion *dummy()* und aus welcher Klasse wurde dieses Objekt generiert? Mit Hilfe des Callstacks können diese Fragen nicht beantwortet werden.

3.6.3.3 Step by step

Grundsätzlich ermöglichen Debugger eine "step by step"-Ausführung des Programms. Dazu werden nacheinander die einzelnen Anweisungen des Programms abgearbeitet. Die Visualisierung beschränkt sich auf die Anzeige

der jeweiligen Zeilen im Quelltext und die Ausgabe der aktuellen Variablen- bzw. Objektwerte. Parallel dazu “läuft” das Programm. Des weiteren gibt es die Möglichkeit “step-outs” und “step-ins” zu machen. Bei einem “step-in” wird versucht, sofern es sich bei der aktuelle Zeile im Quelltext um einen Funktionsaufruf handelt, in diese Funktion zu springen und dort weiter zu gehen(eine Ebene höher im Callstack). Andernfalls wird die nächste Zeile behandelt. Befindet man sich in einer Funktion, kann ein “step-out” gemacht werden, wobei die Funktion ohne “steps” bis zum Ende ausgeführt wird und der Fokus wieder bei der “rufenden Funktion” liegt.
Ähnlich zum Callstack ist die “step by step” Ausführung des Programms zu bewerten. Für imperative Programme bzw. Programmteile stellen sie ein wichtiges Hilfsmittel zur Fehlerfindung dar. Da auch hier nur die Funktionen im Vordergrund stehen, ist diese Debuggerfunktion für objektorientierte Programme im allgemeinen ungeeignet.

3.6.3.4 Breakpoints

Des weiteren können “breakpoints” definiert werden. Diese Breakpoints unterbrechen das Programm zur Laufzeit und springen in den “step by step”-Modus. Breakpoints werden entlang des Quelltextes gesetzt. Sie werden nicht als besonderer Code im Quelltext implementiert, sondern extern durch die Entwicklungsumgebung[18]. Sie teilen dem Debugger in einer “debug session” mit, an welchen Stellen das Programm zu unterbrechen ist.

3.6.3.5 Zusammenfassung

Allen klassischen Debuggern fehlt eine grafische Visualisierung des aktuellen Programmstatus. Die Visualisierung beschränkt sich auf die Ausgabe der aktuellen Variablen und Objektwerte. Insbesondere kann auch nur der Status der Objekte angezeigt werden, deren Funktionen im Callstack liegen. Ein Gesamtüberblick über das laufende Programm fehlt völlig. Die “step by step”-Methode unterstützt nicht die Objektsicht. Das Programm erscheint

[18] Die interne Implementierung ist von Entwicklungsumgebung zu Entwicklungsumgebung unterschiedlich, vom Verhalten aber vergleichbar.

hier als linearer “Spaghetticode”, in dem mit den aus Basic oder Fortran bekannten *GOTO*-Anweisungen durch das Programm gesprungen wird. Da auch nur Funktionen –dazu noch im Quelltext– angezeigt werden, kann hier von einer Ansicht gesprochen werden, die das oben besprochene Problem der Klassen- und Objektsicht verwischt und damit die Fehlvorstellung verstärkt. In Visual Studio unterscheiden sich beispielsweise die Implemetierungsebene und die “debug session” nur durch ein kleines Fenster, was die aktuellen Werte der Attribute und lokalen Variablen der Fokusfunktion anzeigt. Es ist zwar ersichtlich, in welcher Zeile des Quelltextes sich das Programm befindet, aber nicht in welchem Objekt. Nur die ausgegebenen Objektattribute könnten das Objekt identifizieren. Wieviele und welche Objekte parallel existieren ist nicht ersichtlich.

Diese Testart ist nur von Kennern des Programmcodes und der Programmiersprache sinnvoll einsetzbar, da sie primär entlang des Quelltextes arbeitet und Algorithmenabläufe überwacht. Sie gibt textuell immer nur einzelne Zusammenhänge (zwischen Objekten) aus und läßt insofern nur schwierig Rückschlüsse auf die “globale” Programmstruktur zu. Deshalb ist diese Testart für Unkundige eher ungeeignet.

3.6.4 Debug session – grafisch

Eine andere Art des Testens besteht darin, das Programm zur Laufzeit mit seinen Bestandteilen (Objekten) zu beobachten. Dazu sind spezielle Tools nötig, die fähig sind, ein Programm zur Laufzeit in seine Objekte zu zerlegen und deren Beziehungen anzuzeigen. Des weiteren sollte es möglich sein, einzelne Objekte “per Hand” zu generieren. Es sollten die Objekte selbst, die Beziehungen der Objekte untereinander, deren Daten(Attribute) und ihre Funktionen darstellbar sein. Diese neue Programmkategorie hat im Moment nur einen einzige Vertreter: *Mr. Dobs*[19].

3.6.4.1 Black Box Test der kleinsten Elemente

In der Phase des Softwareentwicklungsprozesses, in der Klassen geschrie-

[19] Mr. Dobs ist Teil des Fujaba-Paketes, was an der U-GH Paderborn entwickelt wird.

ben werden, ist es meist nicht möglich –ohne großen Aufwand– einzelne Klassen zu testen. Bisher ist es nötig, zum Test einzelner Klassen, ein spezielles Hauptprogramm (in Java eine spezielle Klasse mir der Memberfunktion *main()*) zu schreiben, daß zur Laufzeit Objekte aus ihnen generiert. Soll der Test modifiziert werden, muß das Hauptprogramm umgeschrieben werden. Natürlich ist hierzu die Kenntnis der Programmiersprache vorausgesetzt.
Mit neueren Tools, wie Mr. Dobs, ist es nun möglich einzelne Objekte aus den Klassen zu generieren und sogar deren öffentliche Memberfunktionen[20] aufzurufen. Einzelne Klassen können nun so auf deren Funktion zur Laufzeit geprüft werden. Attribute bzw. deren Werte können erfragt und angezeigt werden. Des weiteren können auch mehrere Objekte aus einer Klasse generiert werden. Alle so generierten Objekte werden als Kasten grafisch dargestellt. Im Unterschied zu klassischen Tools werden also alle generierten Objekte angezeigt und nicht nur ein einzelnes.
Es ist nun möglich die Funktionsweise von Klassen und deren Zusammenspiel zu testen. Es ist also kein spezialisiertes Hauptprogramm nötig, um diese Tests durchzuführen.
Ist z.B. Objekt B in Objekt A enthalten (als Attribut), wird dieses durch eine Linie visualisiert. In klassischen Tools kann dies nur durch den Callstack geschlossen werden, sofern man sich im Objekt B aufhält. Befindet sich das Programm nicht in Objekt B, gibt es keine Möglichkeit, diese Beziehung "zu sehen". Sie kann nur implizit durch den Typ einer Variable oder eines Attributs geschlossen werden.
Der Begriff Ablaufprozeß bekommt in Mr. Dobs eine andere Bedeutung. Es ist nicht der Ablauf eines Algorithmus (einer Methode) zu sehen, sondern der Ablauf eines Programms (System aus generierten Objekten) als Ganzes. Die Abläufe innerhalb der Methoden bleiben verborgen, weshalb auch vom "Black Box Test der kleinsten Elemente" gesprochen werden kann.

3.6.4.2 Grey Box Test eines Programms

Ist die Phase Erzeugung abgeschlossen, kann das Programm "als Ganzes" getestet werden. Visuelle Debugger bieten die Möglichkeit "hinter die Ku-

[20] In Java und in C++ werden solche Funktionen mit *public* markiert. Sollen diese Funktionen nicht öffentlich sein, wird das Schlüsselwort *private* verwendet.

lissen" eines Programms zur Laufzeit zu schauen. Grafische Debugger, wie Mr. Dobs, sind in der Lage jederzeit einen Programmstatus anzuzeigen. Die Algorithmen der Methoden bleiben verborgen und können nur als "Black Box" (⟶3.6.4.1) getestet werden. Eine "step-by-step" Ausführung und ein "Callstack" aus klassischen Debuggern fehlen. Diese scheinbare Schwäche ist aber eine Stärke des Mr. Dobs. Die Aufgabe des Mr. Dobs ist es, ein objektorientiertes Programm (eine Kombination und Komposition einzelner Objekte (System)) als Ganzes darzustellen und nicht als linearen Prozeß.

3.6.5 Resumee

Jede der vorgestellten Testarten hat ihre Stärken und ihr spezielles Einsatzfeld. Während der Testlauf zur "einfachen" Funktionsüberprüfung benutzt werden kann, kann mit einer theoretischen Prüfung und mit einer klassischen "debug session" eine Überprüfung der Algorithmen gemacht werden. Die bisherige Lücke, des Debuggens auf Objektebene wird nun durch Mr. Dobs ausgefüllt.

4 Mr. Dobs

4.1 Visualisierung von Ablaufprozessen

Wie in 3.6 dargestellt gibt es im wesentlichen zwei Arten der Visualisierung von Ablaufprozessen. Zum eine die eher (Quell-)Textbasierte Variante (⟶3.6.3) und zum anderen die grafische (⟶3.6.4). Die textbasierte Variante ist eher zur Visualisierung von Abläufen innerhalb von Algorithmen geeignet (also der Methoden einer Klasse), wohingegen die grafische Variante (Mr. Dobs) zur Visualisierung von Abläufen innerhalb objektorientierter Software bzw. deren Objektstruktur zur Laufzeit geeignet ist (⟶3.6).
Mr. Dobs als eigenständiger Teil der Entwicklungsumgebung Fujaba stellt eine Plattform dar, mit der grafisch Ablaufprozesse innerhalb eines Informatiksystems (Software in Java) visualisiert werden können.
Zunächst aber zum Gesamtpaket:

4.2 Fujaba

> The primary topic of the FUJABA[21] project and environment is Round Trip Engineering with UML, SDM, Java and Desgin Patterns :
>
> - FUJABA combines UML class diagrams, UML behaviour diagrams, SDM story diagrams, and design patterns to a powerful, easy to use, yet formal system design and specification language.
> - Furthermore the FUJABA environment supports the generation of JAVA-sourcecode out of the whole design which results in a executable prototype, ideally.
> - Moreover the way back is provided, too (to some extend so far), so that the reading-in and parsing of JAVA-code and the regaining of the design, respectivly the patterns, becomes available.
>
> [Pro99]

FUJABA stellt also Java Programme grafisch durch UML- und SDM-Diagramme dar. Durch diese Diagramme kann ein Java Programm analysiert werden. In 4.4.2 wird FUJABA zur weiteren Analyse des Elevator-Demos eingesetzt.

[21] From UML to Java And Back Again

4.3 Mr. Dobs

Mr. Dobs (DOBS - Dynamic Object Browsing System) ist ein grafischer Debugger. Er analysiert Java `class`-Dateien[22] und stellt die Möglichkeit zur Verfügung, einzelne Objekte aus ihnen zu generieren. Es kann das Verhalten der Objekte zur Laufzeit beobachtet werden und darüber hinaus können sie "benutzt" werden. Im Unterschied zu den klassischen Tools wird hier kein Quelltext angezeigt, sondern "nur" Symbole für die generierten Objekte. Des weiteren werden zu den Objekten die "öffentlichen" Attribute, mit ihren Werten, und die "öffentlichen" Methoden ausgegeben[23]. Des weiteren werden die Beziehungen zwischen den Objekten angezeigt. Zur näheren Erläuterung folgt nun eine Beispielsitzung mit dem "Elevator-Demo" (wird zusammen mit Fujaba ausgeliefert).

4.4 Beispiel: Elevator

Das hier besprochene Beispiel simuliert einzelne Aspekte eines Fahrstuhls. Das Programm (es existiert kein "Hauptprogramm"; eine Methode *main()* fehlt) liegt nur als Bytecode vor. Die folgende Sitzung spiegelt meine erste Erkundungstour des Elevator-Demos mit Mr. Dobs wider.
Ziel dieser Sitzung war es, die Funktionalität von Mr. Dobs zu erkunden und zu versuchen, aus den einzelnen Zuständen, Rückschlüsse auf das dem Elevator-Demo zugrundeliegende Modell zu ziehen.
Zur Struktur des Programms:
Eine Haus (Klasse:House) umfaßt einen Fahrstuhl (Klasse:Elevator) mehrere Stockwerke (Klasse:Level) und mehrere Personen (Klasse:Person). Die Klassen Elevator, Level und Person sind untereinander über Referenzen und Funktionen verknüpft.
Das Haus dient im wesentlichen als Container für die einzelnen Akteure des Demos:
In der folgenden Beispielsitzung gibt es einen Fahrstuhl, vier Stockwerke und zwei Personen.

[22] In Bytecode übersetzte Java Klassen (Quelltextdateien)

[23] Es werden **nicht** die Algorithmen der Methoden angezeigt, sondern **nur** die Definitionen der Methoden.

4.4.1 1. Durchlauf

Starten wir nun die Beispielsitzung in Mr. Dobs. Hierzu wird über die Funktion *New* (als Schaltfläche oder als Menüpunkt in "Browse") ein Objekt "House" erzeugt.

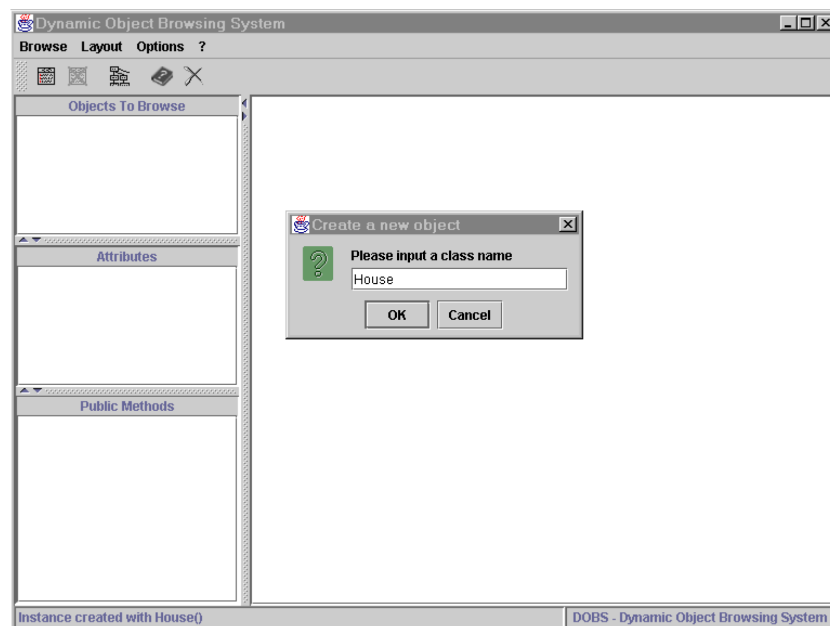


Abbildung 4: Elevator step 1

Wie in Abbildung 5 zu sehen, bekommt das Objekt den internen Namen `h0`. Im Fenster "Public Methods" sind die öffentlichen Funktionen aufgelistet, die auf `h0` wirken können. Im Fenster "Attributes" sind alle öffentlichen Attribute des Objektes mit ihren Werten abgebildet. Im Hauptfenster erscheint ein Kasten mit dem Bild eines Hauses. Dieser Kasten repräsentiert das Objekt `h0`, generiert aus der Klasse "House". `h0` ist noch leer, d.h es existieren noch keine Objekte vom Typ Level, Elevator oder Person.
Es können nun (auch mit '*New*') die weiteren bzw. benötigten Objekte erzeugt und mit den jeweiligen Funktionen von "House" in das Haus integriert

werden(z.B. mit `SetElevator()`). In der hier besprochenen Beispielsitzung erledigt diese Aufgabe die Funktion `createObjekts()`. Durch den Aufruf der Funktion `createObjekts()` werden die in dieser Sitzung benötigten Objekte generiert. Sie generiert einen Fahrstuhl, zwei Personen und vier Stockwerke.

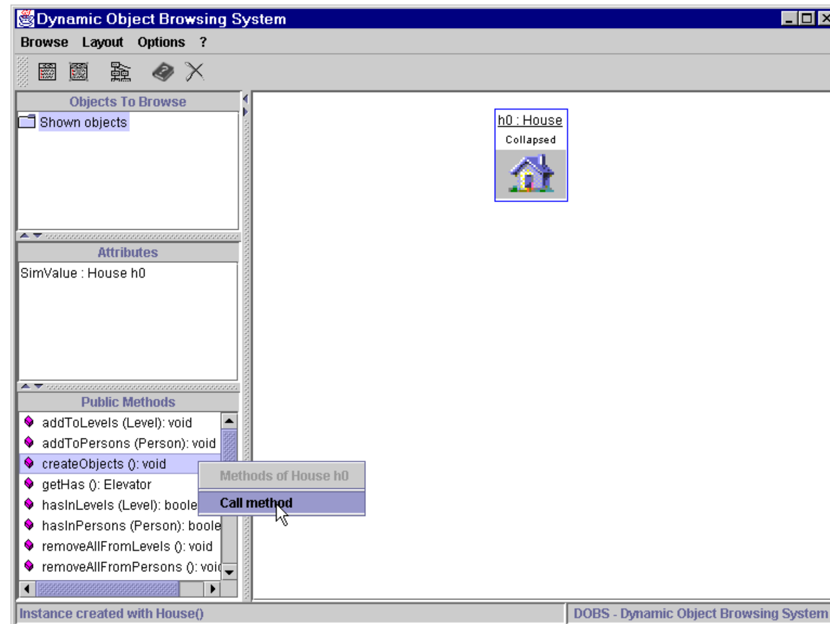


Abbildung 5: Elevator step 2

Wie in Abbildung 5 zu sehen ist, wird dazu einfach im Fenster "public Methods" die Funktion `createObjekts` mit der Maus ausgewählt und durch "klicken" der rechten Maustaste erscheint ein Kontextmenü. Nach dem "Klicken" des Menüpunktes "Call method" wird diese ausgeführt. Es werden die oben genannten Objekte generiert und in das Objekt "House" integriert.

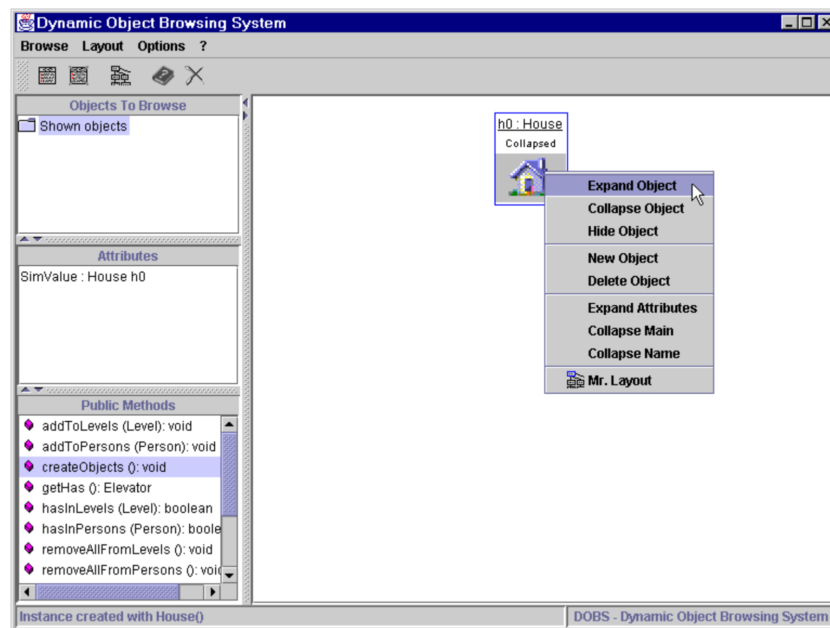


Abbildung 6: Elevator step 3

Nach dem Ausführen der internen Dobs-Funktion *Expand Objekt*(zu erreichen über die rechte Maustaste angewand über dem Kasten von "h0:House"– siehe Abbildung 6) erscheinen nun neue Objekte im Hauptfenster, die mit Linien untereinander verbunden sind (Abbildung 7).

Die Linien symbolisieren Relationen zwischen den einzelnen Objekten.
Die reale Stockwerknummer ist in den "Level-Objekten" im Attribut `number` vermerkt. Sollte in den "Objektkästen" *Collapsed* stehen, sind die Attribute verborgen. Durch einen "Doppelklick" in das entsprechende Objektkästchen werden sie angezeigt.
Abbildung 7 zeigt alle aktuellen Relationen zwischen den Objekten. Zum einen sind alle "neuen" Objekte mit dem Haus verbunden, zum anderen sind einzelne Objekte untereinander verbunden. Durch einen "Klick" auf die Linien, werden diese beschriftet:

`p1`[24]**(person) is at** `l4` person.1(`p1`) ist im ersten Stockwerk.

`p2` **persons.2 house** `h0` person.2(`p2`) ist im Haus(`h0`).

Aus dem Diagramm ist folgender Status abzulesen:
Person1 ist in Stockwerk 1, Person2 ist in Stockwerk 4 und der Fahrstuhl ist in Stockwerk 1. Stockwerke, Personen und der Fahrstuhl sind im Haus. Die letzte Relation ändert sich in den nun folgenden Phasen nicht und kann somit aus Übersichtsgründen ausgeblendet(versteckt) werden. Aktionen finden nur in bzw. zwischen den Objekten von Haus statt, nicht im Haus selbst.

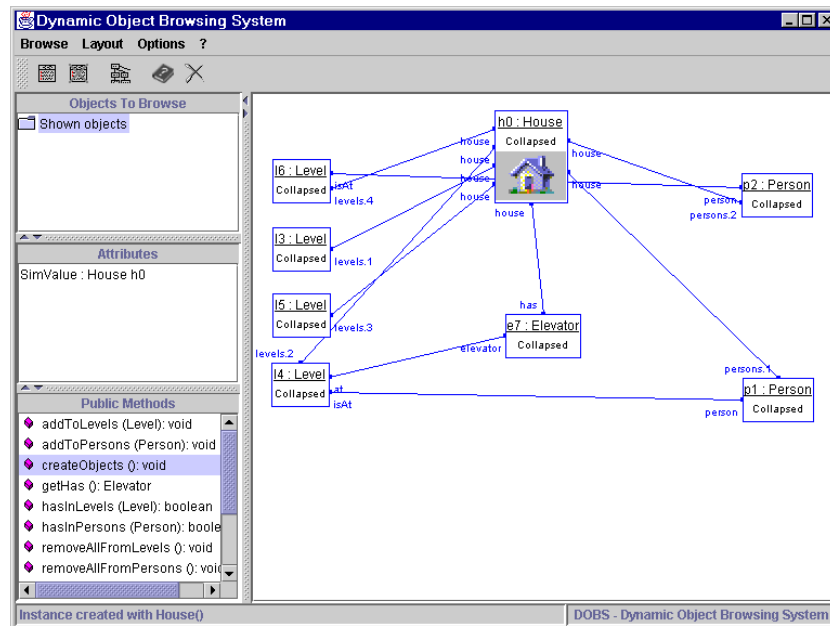


Abbildung 7: Elevator step 4

Abbildung 8 zeigt dieses “verstecken”. Dazu wird im Kontextmenü vom Objekt `h0` der Menüpunkt “Hide Object” angewandt. Abbildung 9 zeigt das nun bereinigte Diagramm.

Es sind nun nur noch die Akteure diese Sitzung zu sehen. Das Objekt `h0` ist in diesem Diagramm versteckt.

Stellen wir uns nun die Aufgabe, daß Person1 in Stockwerk1 zur Person2

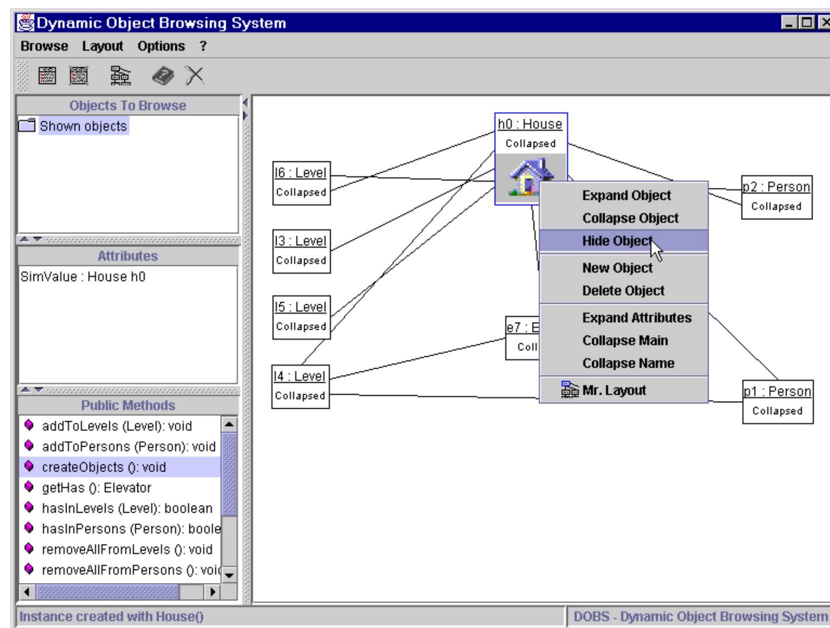


Abbildung 8: Elevator step 5

nach Stockwerk4 befördert werden soll. Zunächst muß Person1 mitteilen, in welches Stockwerk sie will. Dazu wird im Hauptfenster Person1 ausgewählt und die Funktion `setWantsTo()` aufgerufen. Wie im “Public Methods” Fenster zu sehen, hat die Funktion einen Parameter vom Typ Level. Ein Dialog (⟶10) erscheint.

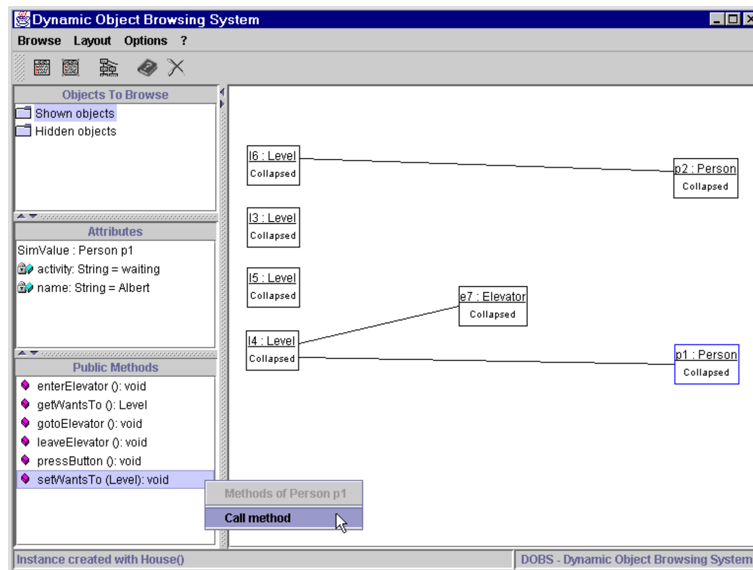


Abbildung 9: Elevator step 6

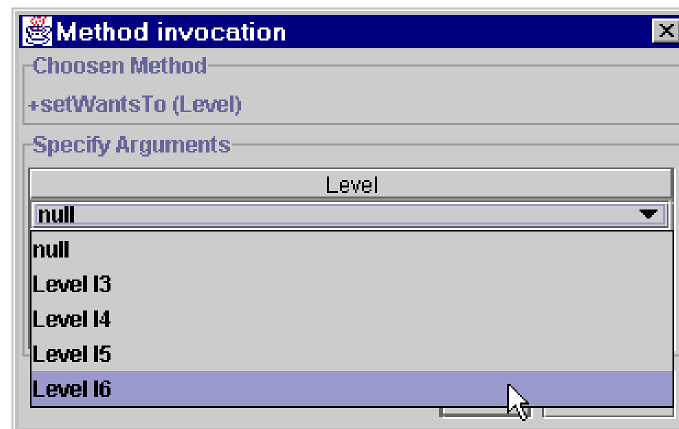


Abbildung 10: Elevator step 7

Der in Abbildung 10 gezeigte Dialog erfragt den Parameter vom Typ Level. Im Kasten “Specify Arguments” sind alle Parameter der Funktion aufgelistet – in diesem Fall ein Parameter vom Typ Level. Im darunterliegenden

Eingabefeld wird durch klicken ein Menü geöffnet, was alle möglichen Objekte, vom Typ Level, zur Auswahl stellt. Zusätzlich zu den Leveln `l3` bis `l6` wird noch `null` angegeben. `null` bezeichnet das "Nullobjekt". Das "Nullobjekt" ist kein Objekt, was benutzt werden kann. Es dient als Synonym für "kein Objekt". Sollte das "Nullobjekt" ausgewählt werden, bedeutet das, daß die Funktion mit einem leeren Parameter aufgerufen wird und insofern –bezogen auf dieses Beispiel– keine Entscheidung bezüglich des Zielstockwerkes getroffen würde. Da Person1 in Stockwerk4 möchte, wählen wir aber `l6`.

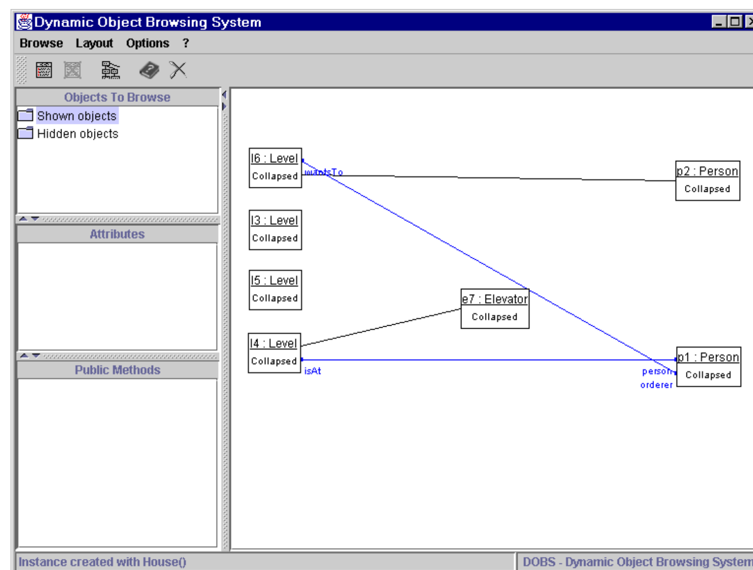


Abbildung 11: Elevator step 8

In Abbildung 11 ist der veränderte Status des Programms abgebildet. Es ist eine neue Line hinzugekommen, die symbolisiert, daß Person1 nach Stockwerk 4 (`l6`) möchte(`p1 wantsTo l6`). Des weiteren wird noch aus Sicht des Stockwerkes angezeigt, daß Stockwerk4 von Person1 angefordert wurde(`l6 orderer p1`).

Da sich der Fahrstuhl im Stockwerk von Person1 aufhält, kann Person1 in den Fahrstuhl gehen. Dieses geschieht über die Funktion `enterElevator()`.

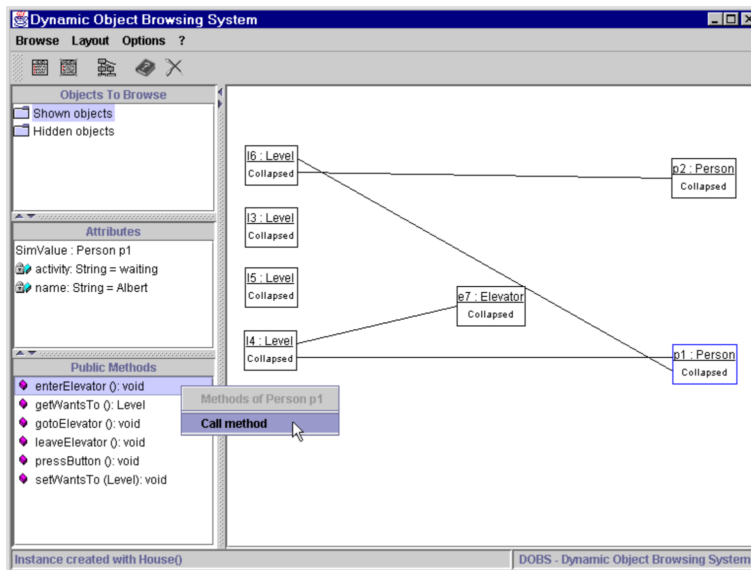


Abbildung 12: Elevator step 9

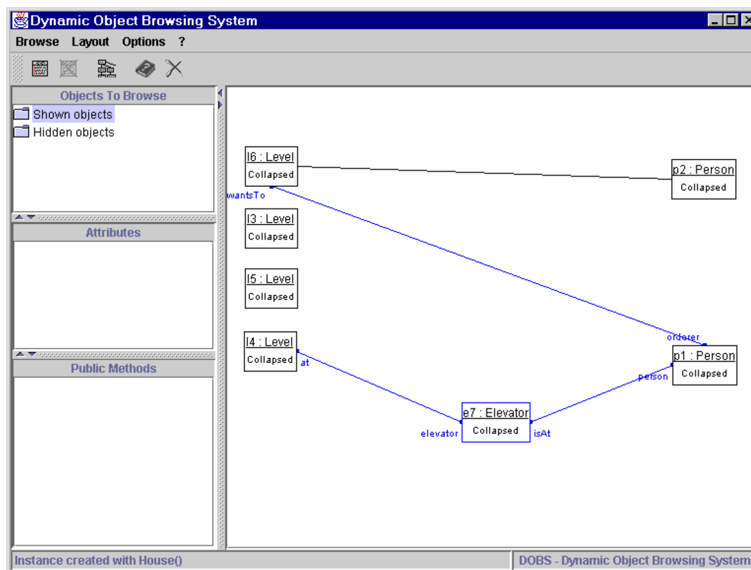


Abbildung 13: Elevator step 10

In Abbildung 13 ist zu sehen, daß Person1 sich nicht mehr im Stockwerk1 sondern im Fahrstuhl befindet. Der Fahrstuhl befindet sich aber weiterhin im ersten Stock.

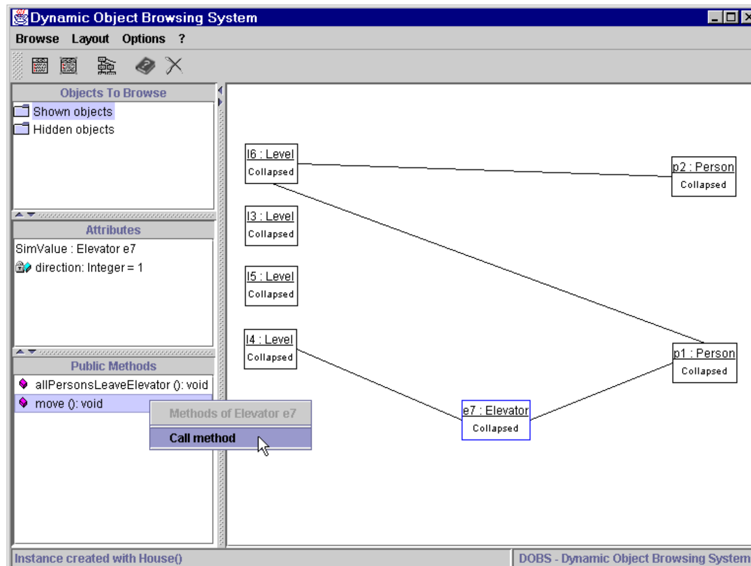


Abbildung 14: Elevator step 11

Über die Funktion `move()` vom Fahrstuhl, bewegt sich der Fahrstuhl ein Stockwerk weiter.

Nach dem Aufruf der Funktion hat sich der Fahrstuhl ein Stockwerk weiterbewegt und befindet sich nun in Stockwerk 2. In den Abbildungen 15 bis 17 sind die einzelnen Stadien nach wiederholtem Aufruf der Funktion `move()` vom Fahrstuhl gezeigt.

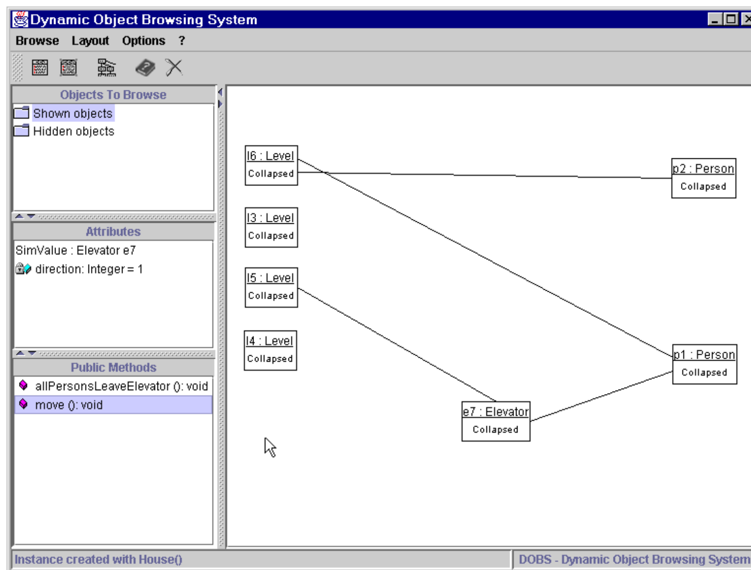


Abbildung 15: Elevator step 12

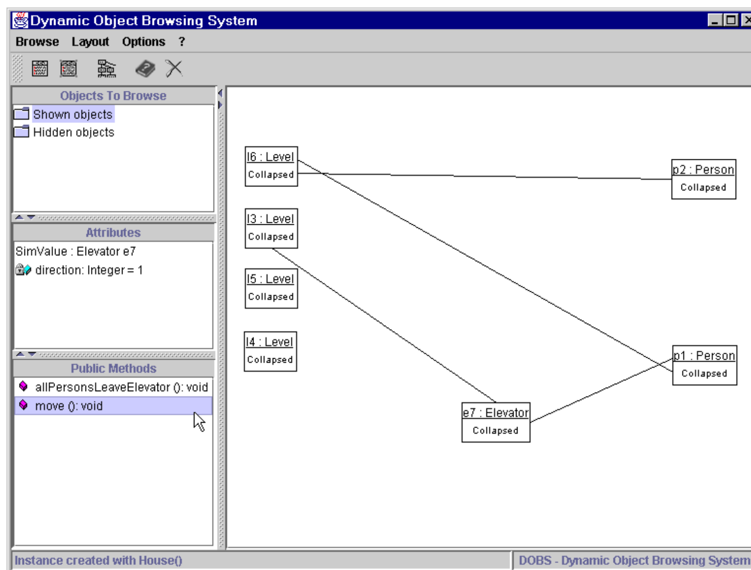


Abbildung 16: Elevator step 13

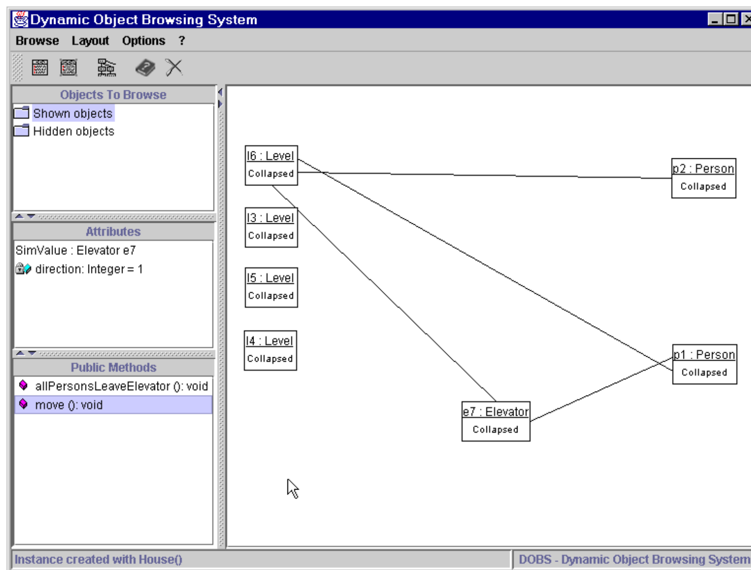


Abbildung 17: Elevator step 14

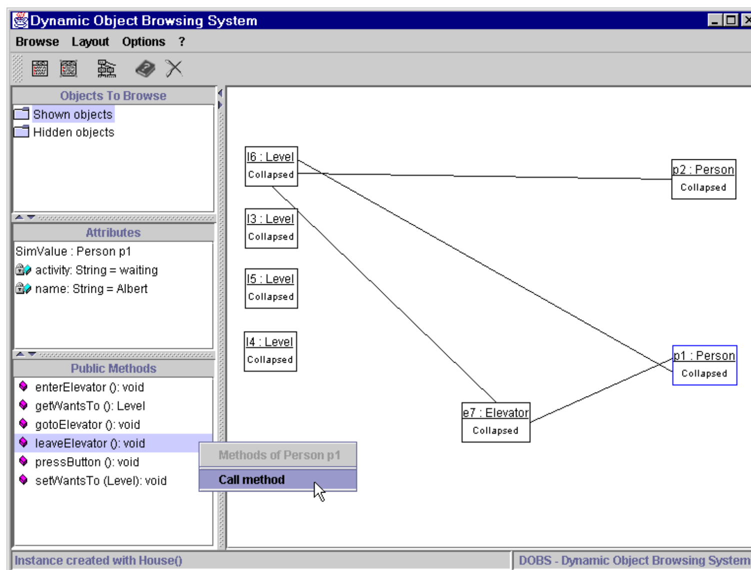


Abbildung 18: Elevator step 15

Der Fahrstuhl ist nun in Stockwerk4 angekommen und Person1 ist weiterhin im Fahrstuhl. Da Person1 in das Stockwerk4 möchte, kann Person1 nun den Fahrstuhl über die Funktion `leaveElevator()` verlassen (siehe Abbildung 18).

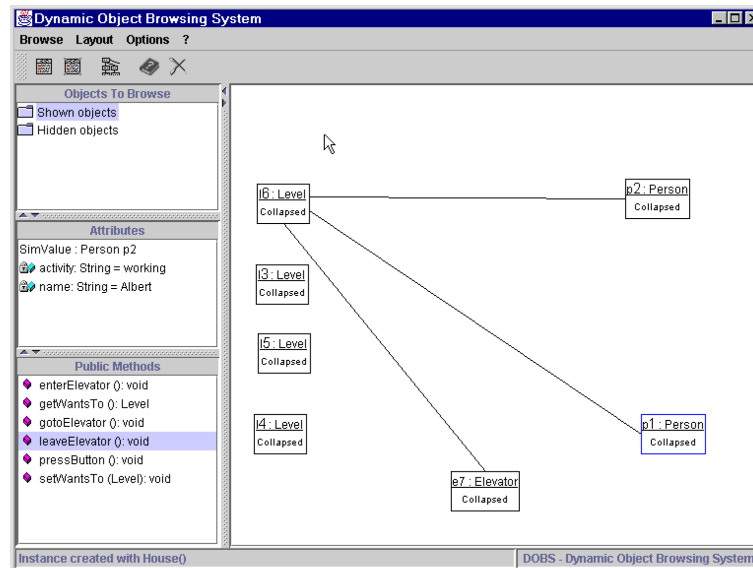


Abbildung 19: Elevator step 16

Abbildung 19 zeigt den Endzustand der Simulation. Person1 ist im Stockwerk4 bei Person2.

Es stellen sich nun die Fragen: Welches Modell steckt hinter dem "Elevator-Demo"? Und sind die Klassen im Sinne der Ersteller bedient worden?

Im vorigen Testlauf traf der Benutzer für Person `p1` die Entscheidung, in welches Stockwerk sie sollte. Der Benutzer ließ die Person in den Fahrstuhl einsteigen und wieder verlassen. Des weiteren gab es für die Person nur zwei Orte: im Stockwerk oder im Fahrstuhl.

Bei näherer Betrachtung der Abbildung 18 ist ein Attribut `activity` zu sehen, was den Wert "waiting" hat. Welche Bedeutung hat dieses Attribut? Im Verlauf des Testlaufs hat sich der Wert nicht verändert. Erst nach dem letzten Schritt (Abb. 19) hat das Attribut den Wert "working" bekommen. Lassen wir nun `p1` wieder in das Stockwerk 1 zurückkehren. Dazu

führen wir wieder die Funktionen `setWantsTo()` und `enterElevator()` aus. Überraschenderweise ist `p1` weiterhin im Stockwerk und nicht im Fahrstuhl. Welcher Unterschied –außer, daß `p1` im vierten Stockwerk ist und `activity` den Wert "working" hat– besteht zur Ausgangsituation (siehe Abbildung 9)?
Zur Beantwortung dieser Fragen ist es wohl nötig einen Blick in das Klassendiagramm und die "Activities" zu werfen.

4.4.2 Dekonstruktion

In Abbildung 20 ist das Klassendiagramm des Elevator-Demos, wie es sich in Fujaba darstellt, abgebildet.

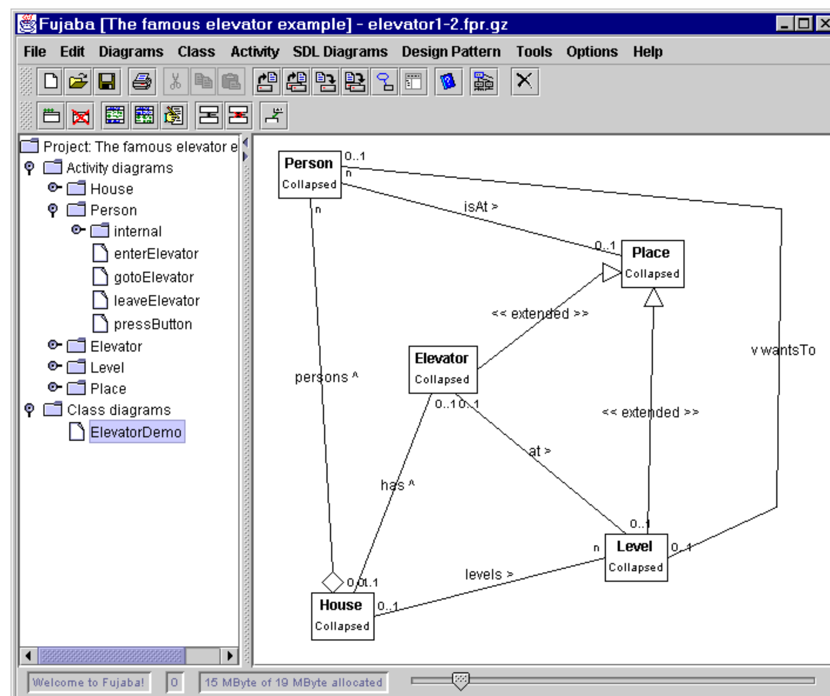


Abbildung 20: Elevator Klassendiagramm

Hier finden sich die Klassen der generierten Objekte aus dem Testlauf wieder. Zusätzlich ist noch eine Klasse `Place` zu sehen, die durch die Klassen `Elevator` und `Level` abgeleitet wird (`Elevator <<extended>> Place` und

`Level <<extended>> Place`).
Eine Person ist an (`isAt >`) einem Platz (`Place`), wobei der Fahrstuhl und die Stockwerke Plätze sind. Durch die Beschriftung der Kante (auf Seiten der Person mit `n` und auf Seiten des Platzes mit `0..1`) ist ausgedrückt, daß mehrere Personen (`n`) sich an einem Platz aufhalten können und daß eine Person sich höchstens an einem Platz aufhält (`0..1`).
Analog können die anderen Kanten interpretiert werden.
Eine Person ist in **einem** Haus und **mehrere** Personen können sich im Haus aufhalten. Durch `wantsTo` zwischen `Person` und `Level` ist der Wunsch der jeweiligen Person ausgerückt, in welches Stockwerk sie möchten.

Kommen wir nun zu den "Aktivitäten" der Klassen und zur Beantwortung der Fragen aus 4.4.1. Ein Problem, was aufgetreten war, ist, daß `p1` den Fahrstuhl nicht ein zweites mal betreten konnte. Schauen wir uns dazu einmal die Methode `enterElevator()` an. In Abbildung 21 ist diese Funktion abgebildet, wie sie sich in Fujaba darstellt.

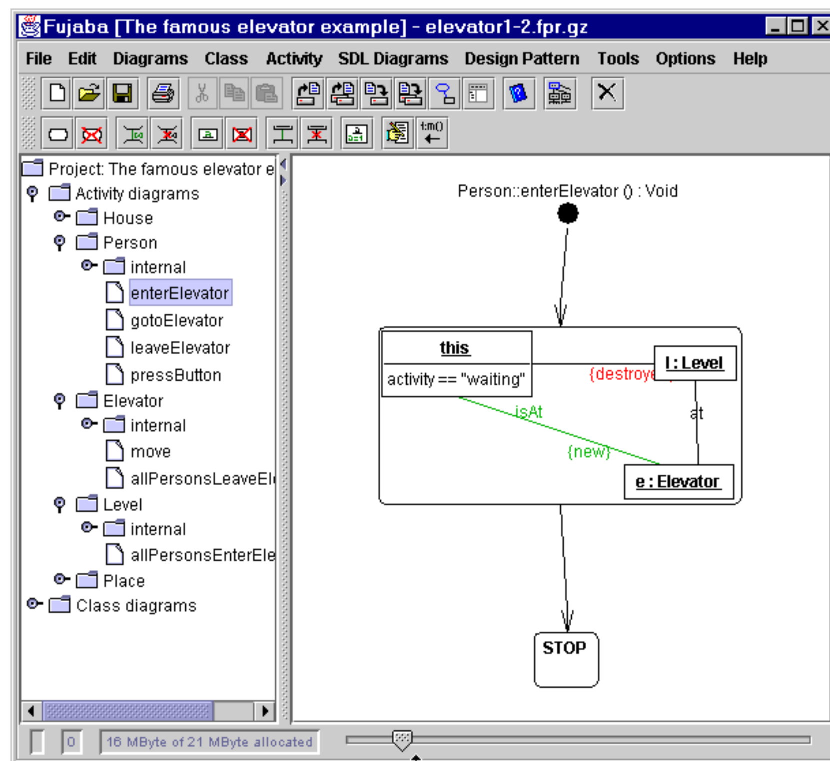


Abbildung 21: Person::enterElevator() (Fujaba)

Wird diese Methode ausgelöst, wird die Beziehung von `this` zum Stockwerk (`Level`) gelöst (`destroyed`) und eine neue Beziehung zum Fahrstuhl (`Elevator`) hergestellt (`new`). Warum betritt nun die Person nicht ein zweites Mal den Fahrstuhl? Der Schlüssel liegt in `activity==waiting`. Es wird also behauptet bzw. verlangt, daß `activity` den Wert `waiting` hat. Sollte dieses nicht zutreffen, bricht die Funktion ab und `this` ist weiterhin im Stockwerk. Springen wir nun zurück zur Dobssitzung (siehe Abbildung 19) und stellen fest, daß das Attribut `activity` von `p1` den Wert "working" hat. `p1` kann also den Fahrstuhl nicht betreten, weil das Attribut `activity` den falschen Wert hat. Welche Aufgabe hat nun dieses Attribut? Schauen wir uns dazu eine bisher noch nicht in Erscheinung getretene Methode von `Person` an: `gotoElevator()`.

In Abbildung 21 ist zu sehen, daß, falls die Person in einem Stockwerk ist

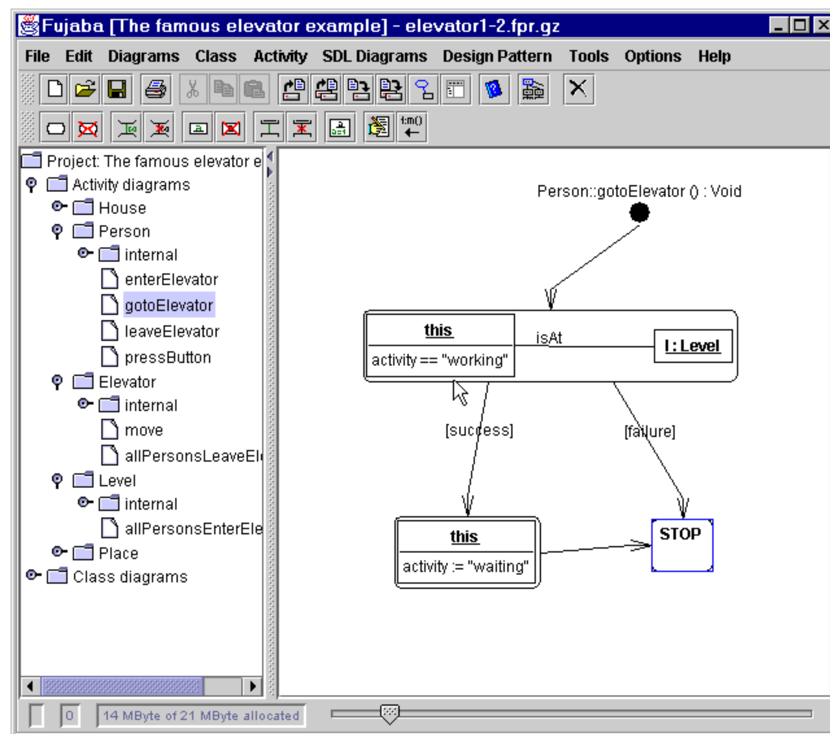
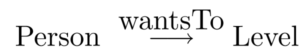


Abbildung 22: Person::gotoElevator() (Fujaba)

und `activity` den Wert “working” hat (`[success]`), `activity` auf “waiting” gesetzt wird. Falls nicht (`[failure]`), verändert sich nichts.
Dieses bedeutet, daß `p1` zuerst zum Fahrstuhl gehen muß (`gotoElevator()`), um ihn dann erst zu betreten (`enterElevator()`).
Hieraus läßt sich schließen, daß das Modell, was den Programmierern vorschwebte, ein wenig komplexer war, als das in 4.4.1 verwendete:
Eine Person befindet sind bei der Arbeit (`activity==working`) und kann den Fahrstuhl nicht benutzen. Geht diese Person zum Fahrstuhl (`gotoElevator()`), kann sie ihn benutzen. Dieses drückt sich durch `activity==waiting` aus. Eine Person hält sich also nicht einfach in einem Stockwerk auf, sondern ist in ihm entweder am Arbeiten oder wartet am Fahrstuhl. Eine Designentscheidung der Programmierer war es, daß für diese Unterscheidung nicht zwei unterschiedliche *Plätze* (z.B `Work` und `atElevator`, abgeleitet von `Level`), entworfen wurden, sondern diese Aussage “nur” durch ein Attribut ausge-

drückt wird.

Nun könnte die Person das Stockwerk auswählen, in das sie will. Doch diese Methode steht (in Fujaba) nicht zur Verfügung (siehe Abbildung 21). Die Funktion `setWantsTo()` tritt in Fujaba nicht auf. Nur im Klassendiagramm existiert eine Assoziation `wantsTo`.



$$\text{Person} \xrightarrow{\text{wantsTo}} \text{Level}$$

Fujaba generiert automatisch zu jeder Assoziation `get-` und `set-`Methoden, was bedeutet, daß (nach dem Export der Klassen nach Java) die Klasse Person die Methoden `setWantsTo() und getWantsTo()` hat.
Die Methode `setWantsTo()` gehört also nicht zum Modell der Programmierer bzw. sie war von ihnen nicht als Methode zur "Bedienung von Außen" gedacht. *Von Außen* wird in Java durch das Schlüsselwort `public` ausgedrückt. Nur reicht hier dieses eine Schlüsselwort nicht aus um zwei unterschiedliche *Außen* auszudrücken. Zum einen gibt es ein *ausserhalb der Klasse*(als Schnittstelle zu anderen Klassen) und zum anderen ein *ausserhalb des Programms* (als Schnittstelle zur Steuerung des Programms). Durch die Javasyntax gibt es keine Möglichkeit, dieses auszudrücken, deshalb erscheinen in Mr. Dobs auch die "internen" Methoden (`public` innerhalb des Programms).
Wenn nun die Methode `setWantsTo()` von den Programmierern nicht vorgesehen war, wie wird bestimmt, in welches Stockwerk eine Person möchte? Schauen wir uns dazu die Funktion `pressButton()` an.
Startbedingung ist, daß `this` (die Person) im Fahrstuhl und der Fahrstuhl in einem Stockwerk ist. Trifft dies zu (`success`), wird eine lokale Variable `rnd`, durch eine Zufallszahl ($\in \{1..4\}$), bestimmt. Diese Zahl symbolisiert die Stockwerknummer (1 bis 4). Nun wird erfragt, ob die Person im Haus ist und ob das durch `rnd` bestimmte Stockwerk im selben Haus ist. Trifft dies zu, wird das interne Attribut `wantsTo` auf das Zielstockwerk (`level` mit `number==rnd`) gesetzt.
Also soll das Zielstockwerk nicht durch den Benutzer ausgewählt werden, sondern wird "per Zufall" bestimmt. Dieses impliziert eine weitere Mo-

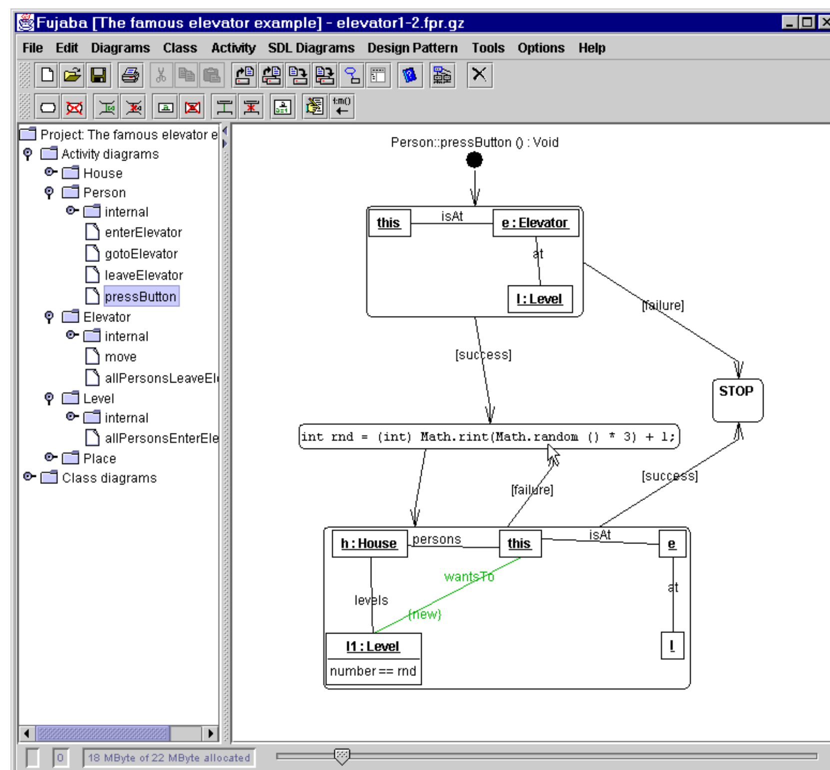


Abbildung 23: Person::gotoElevator() (Fujaba)

delländerung.

Schauen wir uns nun eine Dobs-Sitzung nach dem angepaßten Modell an.

4.4.3 2. Durchlauf

Starten wir die Sitzung wie in 4.4.1 und erhalten folgende Situation:
Als erstes lassen wir die Person (`p2`) den Fahrstuhl betreten (`enterElevator()`).

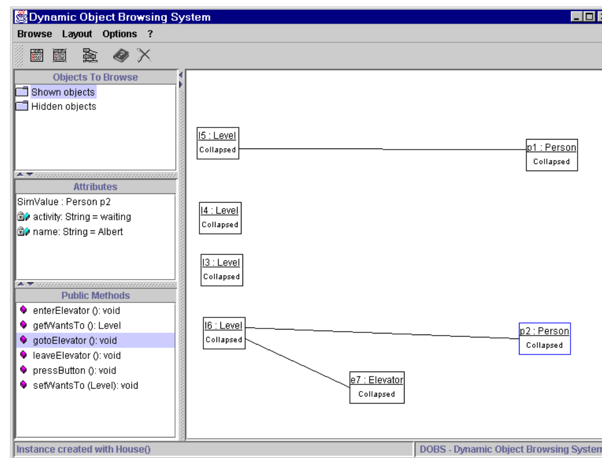


Abbildung 24: Elevator step 1

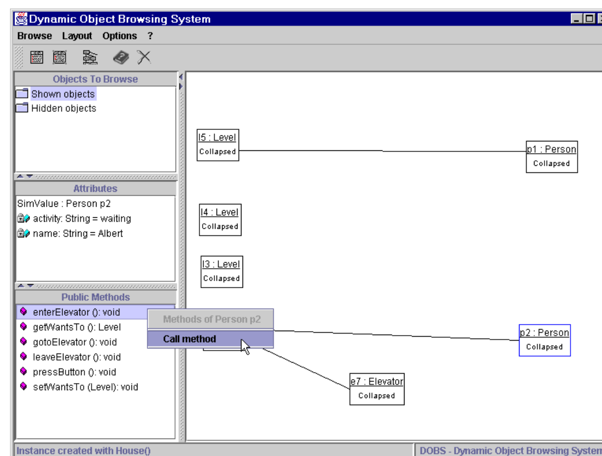


Abbildung 25: Elevator step 2

Im Fahrstuhl angekommen, benutzen wir die `pressButton()`-Funktion, um durch den Zufallszahlengenerator ein Zielstockwerk zu bestimmen.
Es wurde der zweite Stock “ausgewürfelt”.

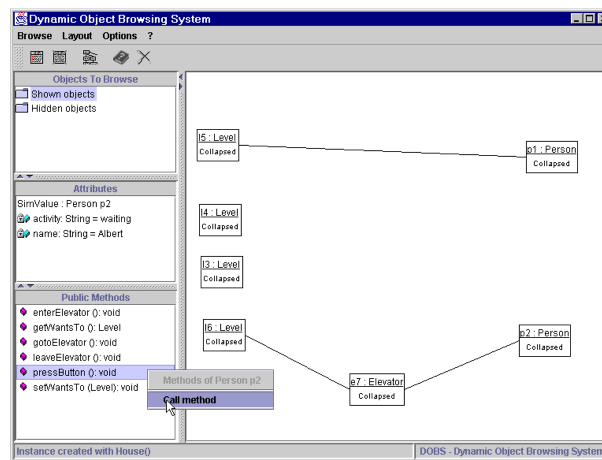


Abbildung 26: Elevator step 3

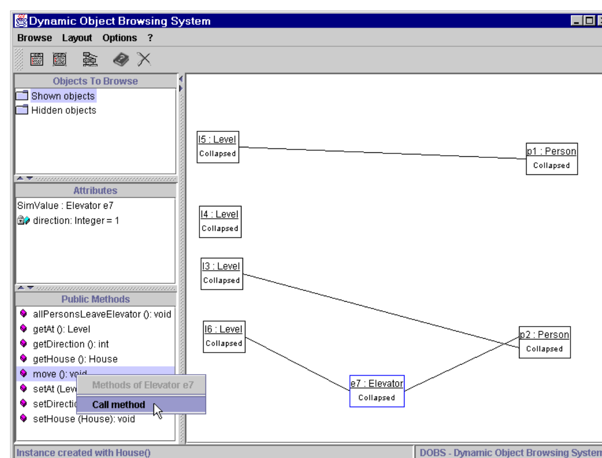


Abbildung 27: Elevator step 4

Nach einem `move()` des Fahrstuhls befindet er sich im zweiten Stockwerk und wir können `p2` aussteigen lassen.
Im zweiten Stock angekommen und ausgestiegen, befindet sich `p2` bei der Arbeit(activity=working).
Um zu überprüfen, ob im vorherigen Abschnitt das richtige Modell gefunden

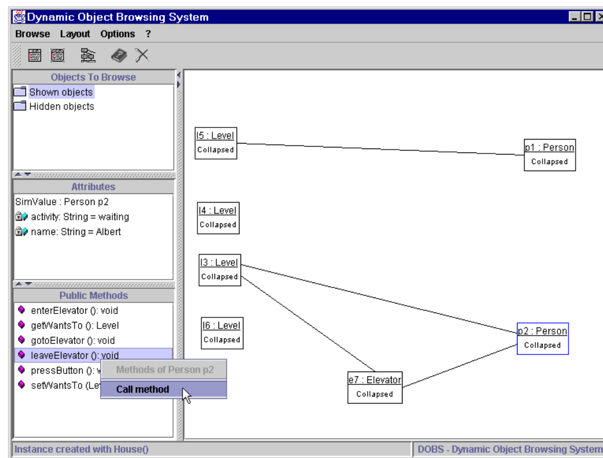


Abbildung 28: Elevator step 5

wurde, lassen wir `p2` erneut in ein anderes Stockwerk fahren.

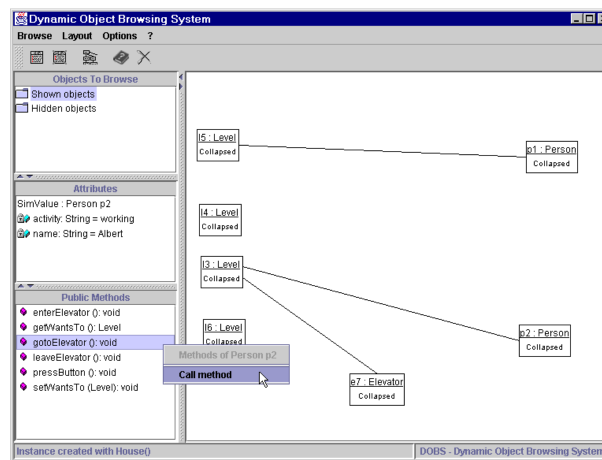


Abbildung 29: Elevator step 6

Wie in Abbildung 30 zu sehen, hat `activity` nun den Wert `waiting` und kann somit den Fahrstuhl betreten.

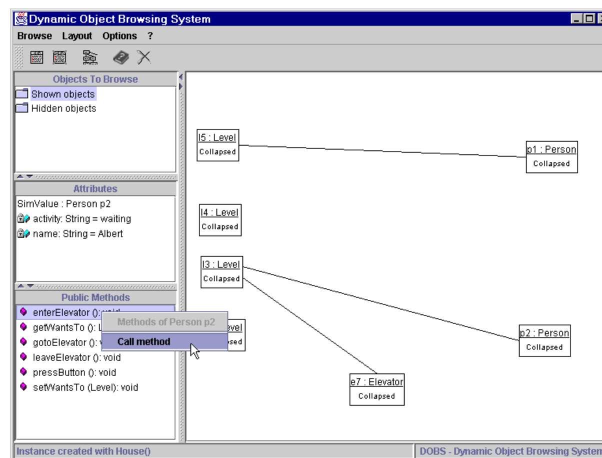


Abbildung 30: Elevator step 7

In Abbildung 31 sehen wir, daß die Person den Fahrstuhl betreten hat. Daraus folgt, daß wir nun dem Modell, was der Programmierung zugrunde lag, nähergekommen sind. Zumindest kann das Modell Elevator bedient werden.

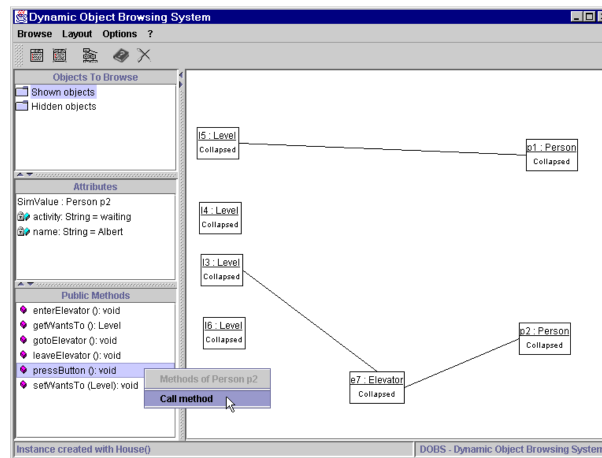


Abbildung 31: Elevator step 8

4.5 Fazit

Die Benutzung des Elevator-Demos folgte für mich aus den Benennungen der Methoden und Attribute. Zum Teil habe ich die Bedeutungen richtig geschlossen, zum Teil nicht. Andere Benuter können die Benennungen anders interpretieren und insofern das Demo anders benutzen. Eine Kommentierung der Attribute und Methoden wäre diesbezüglich zur Findung der richtigen Bedienung des Modells zuträglich (⟶A.1).

Ich habe vor dem Testlauf versucht die Diagramme, die in Fujaba dargestellt werden, zu verstehen. Dieses schlug fehl, da mir die Bedeutung der Notationen nicht klar war. Nachdem ich mit Mr. Dobs einen ersten Durchlauf gemacht habe, konnte ich einige Diagramme entschlüsseln und verstehen. Daraus könnte folgen, daß Mr. Dobs nicht nur zur Erkundung eines "fremden" Programms eingesetzt werden kann, sondern auch *implizit* zum Verständnis einer grafischen Ausgabe, wie der von Fujaba, beitragen kann.

Im Nachhinein stelle ich fest, daß ich noch weitere Informationen aus dem ersten Durchlauf hätte folgern können. Durch die Beziehungen der Person `p1` zum Stockwerk `l4` (⟶Abb. 12) **und** zum Fahrstuhl `e7` (⟶Abb. 13) hätte ich eine "Oberklasse" vermuten können, da sonst `isAt` der Person nicht zu beiden (exklusiv) eine Beziehung haben könnte. Im Klassendiagramm ist

diese “Oberklasse” mit `Place` bezeichnet.
Daraus folgt für mich, daß durch mehrfache Benutzung von Mr. Dobs als Analysewerkzeug, dessen Ausgaben immer mehr Rückschlüsse auf das der getesteten Software zugrundeliegende Modell zulassen.

Es hat sich gezeigt, daß Mr. Dobs alle Anforderungen einer Benutzerschnittstelle mit direkter Manipulation aufweist und deshalb als Experimentierumgebung eingesetzt werden kann.

Folgerung:

Mr. Dobs als DMI[a] kann zur Analyse und zum Test objektorientierter (Java-) Programme eingesetzt werden.

[a] direct manipulation interface

5 Folgerungen für den Informatikunterricht

Wie in Kapitel 2 gesagt, sollten Informatiksysteme im Informatikunterricht behandelt werden. Zur "Rollenbestimmung" des Mr. Dobs und zur Aufdeckung der Vorteile wird hier zunächst näher auf die einzelnen Ebenen eines Informatiksystems eingegangen. Anschließend werden Vorschläge gemacht, wie bzw. wann Mr. Dobs im Unterricht eingesetzt werden kann.

5.1 Verschiedene Ebenen

> Dekonstruktion kann den Blick auf verschiedene Abstraktionsebenen und Repräsentationsformen von Informatiksystemen eröffnen: Die Benutzungsoberfläche und den dort vorhandenen interaktiven grafischen Elementen und ikonischen Repräsentationen von Softwarefunktionalitäten, die Funktionen der Software und deren potenzielle Einbettung in den sozialen Interaktionskontext des Informatiksystems, den Quellcode als textuelle Repräsentationsform der Software des Informatiksystems, den Assemblercode oder gar den binären Maschinencode, der zugleich auch die Verbindung zur Hardwarearchitektur herstellt. [Mag00]

Jeder der Blickwinkel stellt für sich ein eigenes Problemfeld dar. Doch ist bei deren Behandlung im Unterricht auf eine klare Trennung der einzelnen Blickwinkel zu achten, um Fehlvorstellungen analog zu denen in 3.5 dargestellten, zu vermeiden. Vor allem in der Unterscheidung der Ebenen *Quelltext* und *Programm zur Laufzeit* (ohne Benutzerberfläche) liegt ein Kern von Fehlvorstellungen, da das eine ein System beschreibt und das andere eine spezielle Ausprägung eines Systems ist(⟶3.5).
Aber auch die Ebenen *Benutzeroberfläche* und *Programm zur Laufzeit* können Fehlvorstellungen hervorrufen. Einer Benutzeroberfläche muß nicht das gleiche oder auch nur ein ähnliches System zugrundeliegen, welches den internen Programmstrukturen zugrundeliegt. Ein Ziel bei der Entwicklung der Software "Schulkiosk" ist es, daß die Oberfläche die Interna des Programms wiederspiegelt (Klassen/Objektstruktur), doch kann dadurch die Fehlvorstellung hervorgerufen werden, daß dieses in jedem Programm so ist.
In der Praxis ist es zum Teil nicht möglich, eine klare Trennung zwischen Oberfläche(-n Klassen) und Daten (-Klassen) zu ziehen. Zum Beispiel: wie

soll ein Fortschrittsbalken den internen Fortschritt eines Berechnungsalgorithmus (einer Datenklassen/-objekt-Methode)) widerspiegeln, wenn der Algorithmus keine Nachrichten an die Oberfläche sendet[25]?

Folgerung:

Bei einer Betrachtung von Informatiksystemen auf verschiedenen Ebenen ist auf deren klare verbale und kognitive Trennung zu achten.

5.2 Benutzeroberfläche/Daten

Eine Software, wie z.B. der "Schulkiosk", beinhaltet eine Benutzeroberfläche und einen Satz von Datenklassen zur internen Datenverarbeitung. Die Benutzeroberfläche bzw. die Objekte, die aus den Oberflächenklassen generiert wurden, beinhalten Elemente zur Steuerung der Datenobjekte (Waren in ein Regal legen, bezahlen, ...). Andersherum beeinflußt der Status der Datenobjekte die Oberfläche, denn die Oberfäche sollte nur Daten anzeigen, die durch Datenobjekte repräsentiert werden (Berechnungen werden in den Datenklassen gemacht und nicht in der Oberfläche). Eine Benutzeroberfläche kann (und soll) also "nur" eine Veränderung der Datenobjekte bzw. deren Struktur "anstoßen". Anschließend soll und muß sie diese Veränderungen wiederspiegeln. Die Kommunikation zwischen diesen Ebenen geschieht über vorher definierte Schnittstellen.

Die Datenklassen repräsentieren bzw. beinhalten und verwalten Informationen und eine Oberfläche stellt sie dar. Die interne Verwaltung und deren Repräsentanten in der Oberfläche unterliegen unterschiedlichen Regeln und Zielsetzungen. Zum Beispiel kann aus Geschwindigkeitsgründen intern ein binärer Sortierbaum zur Datenstrukturierung verwendet werden, der aber

[25] Wenn überhaupt, ist dies nur über einen Umweg machbar: Es wird ein "Dummyobjekt"(der Datenklassen) benötigt, welches den internen Status des Algorithmus wiedergibt. Der Algorithmus ändert zur Laufzeit den Status des "Dummyobjektes", welcher vom Fortschrittsbalken ausgelesen und visualisiert wird.

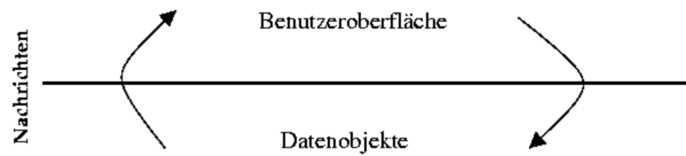


Abbildung 32: Benutzeroberfläche ↔ Datenobjekte

aus Übersichtsgründen in der Oberfläche durch eine (lineare) Liste dargestellt wird. Daraus folgt, daß “im allgemeinen” keine Rückschlüsse, ausgehend von einer Benutzeroberfläche, auf die intere Datenorganisation gemacht werden können.
Ein System[26], das die Datenobjekte repräsentieren, muß sich also im allgemeinen nicht in der Oberfläche wiederspiegeln. Insofern kann eine Oberfläche ein anderes System implizieren.

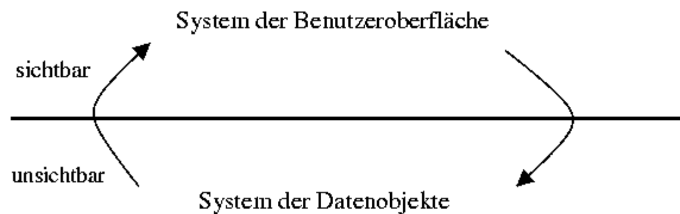


Abbildung 33: Benutzeroberfläche ↔ Datenobjekte aus Systemsicht

Da eine Oberfläche sichtbar für den Benutzer ist, ist deren System zu erkennen bzw. zu erschließen. Das System der Datenobjekte ist für die Benutzer des Programms verborgen (Black Box). Es existiert zwar eine Abbildung vom Datenobjektsystem zum System der Benutzeroberfläche und umgekehrt, doch müssen sie nicht Deckungsgleich sein. Verspricht eine Oberfläche eine spezielle Funktion, muß ein Datenobjektsystem, diese nicht unbedingt erfüllen (Beispiel: siehe A.2).
Ein Benutzer muß wissen, wie z.B. seine Textverarbeitung zu bedienen ist.

[26]Nach der allgemeinen Definition ⟶[Dud90]

Er muß wissen, wie er Daten eingibt und verwaltet. Er entwickelt "für sich" eine Vorstellung von dem System, was er aus der Oberfläche bzw. dessen Bedienung geschlossen hat. Die interne Datenverwaltung des Programms ist verborgen und für ihn in erster Linie uninteressant. Eine andere Textverarbeitung kann eine völlig andere Benutzeroberfläche haben und insofern auch ein anderes System implizieren, aber auf dem gleichen Datenobjektsystem beruhen. Andersherum betrachtet ist im wesentlichen die Bedienung der Textverarbeitungen z. B. MS Word und Star Office sehr ähnlich, d.h. das System bzw. dessen Verständnis, was zur Benutzung erforderlich ist, kann bei beiden das gleiche sein. Das interne Datenobjektsystem **ist** aber ein anderes. Das heißt –im allgemeinen– erfordert ein Benutzeroberflächensystem kein spezielles Datenobjektsystem (modulo Anforderungen) und ein Datenobjektsystem erfordert kein spezielles Benutzeroberflächensystem.
Wenn nun die Systemfunktionen einer Software (Benutzeroberfläche + Datenobjektsystem) erkundet werden sollen, eine Software analysiert oder gar dekonstruiert werden soll, ist es zwar ohne weiteres (in bezug auf die Mittel) möglich ein System einer Benutzeroberfläche zu ergründen, doch **darf** daraus nicht automatisch auf das Datenobjektsystem geschlossen werden.

"Die Funktionale Einbettung der Software in den Kontext sozialer Interaktionen des Informatiksystems erfolgt nicht zuletzt über die Benutzerschnittstelle" [Mag00]. Das heißt, eine Benutzeroberfläche(-schnittstelle) sollte auf den Einsatzkontext zugeschnitten sein. Andererseits soll eine Software die allgemeinen Softwarequalitätsmerkmale aufweisen. Zur Lösung dieses Spannungsfeldes wird in der Praxis ein Datenobjektsystem entworfen, was die allgemeinen Softwarequalitätsmerkmale aufweist, und eine Benutzerschnittstelle, welche die Anforderungen, der mit diesem Programm interagierenden Personen, erfüllt. Ein Ziel des Informatikunterrichts kann es sein, dies zu verdeutlichen:

Folgerung:

Eine Software gliedert sich in ein Benutzeroberflächensystem und ein Datenobjektsystem. Diese beiden Systeme unterlagen bei der Entwicklung des gesamten Informatiksystems unterschiedlichen Zielsetzungen und sind insofern im allgemeinen nicht deckungsgleich.

Es stellt sich nun die Frage:

Wie können Datenobjektsysteme und deren interne Abläufe analysiert werden?

Auf dieser abstrakten Ebene kann Mr. Dobs die Rolle der "Oberfläche" einnehmen, da es zum einen auf die Schnittstellen der Datenobjekte zugreifen kann und zum anderen jederzeit den aktuellen Status des Datenobjektsystems anzeigt.

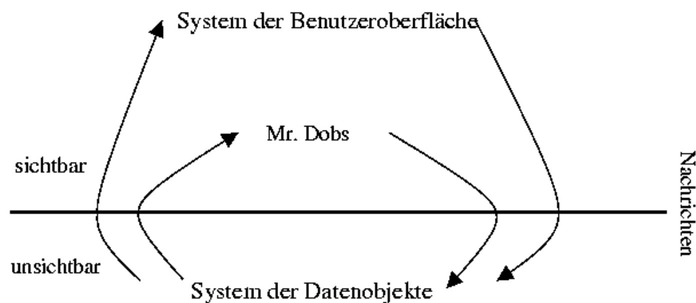


Abbildung 34: Benutzeroberfläche ↔ Datenobjekte ↔ Mr. Dobs

Durch Mr. Dobs ist es desweitern möglich eine durch die Benutzeroberfläche angestoßene Veränderung des Status des Datenobjektsystems zu betrachten. Es können also die Auswirkungen einer Benutzerinteraktion auf das Datenobjektsystem angezeigt werden.

Folgerung:

Mr. Dobs kann zur Steuerung und Analyse eines Datenobjektsystems eingesetzt werden. Insofern kann er die Rolle des Benutzeroberflächensystems einnehmen.

5.3 Datenobjektsystem und Quellcode

Wie in 3.5 gezeigt, liegt ein Problemfeld, bei der Erstellung von Software und auch bei der Analyse einer Software, in der Unterscheidung von Klassen und Objekten.

Wie in 3.6.3 gezeigt, sind die klassischen Debugger nicht in der Lage Objektstrukturen anzuzeigen und sind insofern nicht dazu geeignet das Problemfeld aufzulösen. Mit Mr. Dobs ist dies möglich, da zum einen das Datenobjektsystem angezeigt wird und zum anderen eine klare Trennung zwischen Laufzeitebene und Beschreibungsebene gemacht wird (Es existiert keine Quelltextebene in Mr. Dobs).

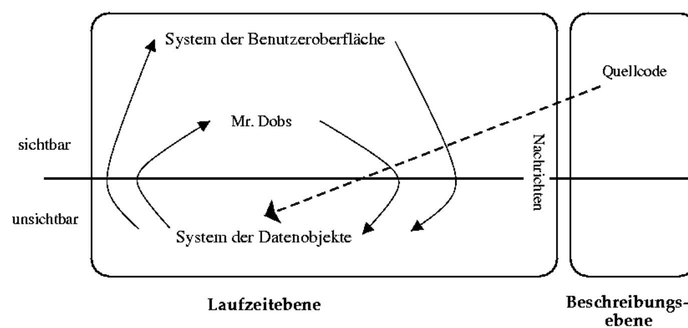


Abbildung 35: Laufzeitebene ↔ Beschreibungsebene (1)

Wie in Abbildung 35 zu sehen, stellt Mr. Dobs eine Debuggerfunktionalität

auf Datenobjektebene zur Verfügung. Das durch den Quelltext **beschriebene** Datenobjektsystem, ist zur Laufzeit durch Mr. Dobs sichtbar. Da Mr. Dobs grafische Ausgaben erzeugt, ist zu vermuten, daß ein Umschalten in eine textuelle Beschreibung (Quelltext), in bezug auf die Erkennung von Strukturen, problematisch ist. Eine grafische Darstellung des Quelltextes wäre hilfreich. Tools wie Fujaba und Together J können diese leisten und wären in diesem Kontext sicher zu empfehlen.

Abbildung 36: Laufzeitebene ↔ Beschreibungsebene (2)

In Abbildung 36 ist ein solches Szenario dargestellt.

Folgerung:

Durch Tools wie Fujaba und Mr. Dobs ist es möglich , im Rahmen der Programmentwicklung und Programmwartung, auf eine textuelle Darstellung eines Programms zu verzichten.

5.4 Inhaltliche Einbettung in den Informatikunterricht

Im folgenden sollen Vorschläge und Hinweise für einen systemorientierten Informatikunterricht dargelegt werden. Im Mittelpunkt steht hier die An-

wendung von Mr. Dobs. Von E. Lehmann wurde in “Komplexe Systeme, Eine fundamentale Idee im Informatikunterricht” eine Behandlung dieser im Informatikunterricht vorgeschlagen [Leh95]. Diese sollen mit ingenieurmäßigen Methoden konstruiert oder gewartet werden.

	Vorgehensmodell A	**Vorgehensmodell B**	**Vorgehensmodell C**	
Benutzen		Ein vorhandenes Softwareprodukt benutzen	Ein unfertiges Softwareprodukt benutzen	**1**
Analysieren	Ein neues Softwareprodukt	analysieren	analysieren	**2**
Planen, Zerlegen, Konstruieren, Integrieren	in Teamarbeit konstruieren		in Teamarbeit erweitern	**3**
Benutze, Warten	und warten	und in Teamarbeit warten	sowie benutzen	**4**
	Ein komplexes System weitgehend selbstständig konstruieren mit Methoden des Software-Engineering	*Ein vorhandenes komplexes System weitgehend selbständig mit Methoden des Reengineering warten*	*Ein vorhandenes Teilsystem mit den Methoden des Software (Re-)Engineering erweitern*	

Tabelle 2: Komplexe Systeme im Informatikunterricht nach [Leh95]

Wie in Kapitel 5 gesagt, kann Mr. Dobs auf den Ebenen *Analysieren*(2) und *Warten*[27](4) eingesetzt werden. Der Vorschlag (Forderung) von Lehmann wurde in Hinblick auf die fehlenden Tools zur Durchführung einzelner Unterichtsphasen kritisiert. Mit Mr. Dobs scheint nun ein Tool vorzuliegen, was diese Unterrichtsphasen durchfürbar(er) macht.
Abbildung 37 zeigt schematisch die unterrichtliche Einbettung und Rolle von Mr. Dobs in den Kontext des Informatikunterrichts. Hieraus ergeben sich nach Lehmann[Leh95] (⟶Tabelle 2) folgende Möglichkeiten:

[27] Warten wird hier zum einen als Phase der Fehlerbeseitigung und als Phase der Erweiterung der Funktionalität der Software verstanden

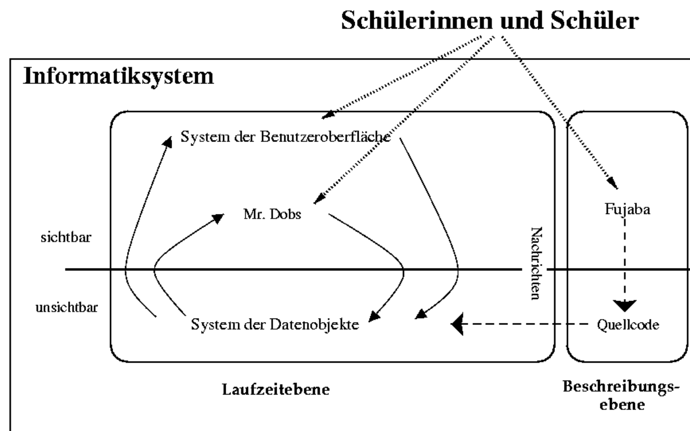


Abbildung 37: Laufzeitebene ↔ Beschreibungsebene (2)

A4,B4 In diesem Kontext kann Mr. Dobs in bezug auf die Fehlerfindung hilfreich sein. Speziell können Fehler im Datenobjektsystem erkannt (⟶3.6.4.2) und durch Experimente (Tests) die Wirkungsweise einzelner Komponenten erkundet (⟶3.6.4.1) werden.

B2 Hier kann Mr. Dobs die internen Abläufe und Datenobjektstrukturen der Software aufdecken bzw. visualisieren. Dadurch können Hypothesen in Hinblick auf die Programmstruktur angestellt werden (⟶4.5 u. 5.3). Insofern kann Mr. Dobs zum Verständnis und zur Dekonstruktion des Informatiksystems beitragen (⟶4).

C2 Mit Mr. Dobs kann die Funktion einzelner Komponenten einer unfertigen Software getestet werden. Des weiteren können durch Experimente mit Mr. Dobs die Leistung des bisherigen Systems erkundet werden(⟶3.6.4.1). Die Unzulänglichkeiten des vorhandenen Systems können festgestellt werden und so zu konkreten Erweiterungsvorschlägen führen.

Des weitern wäre es sicherlich interessant, Mr. Dobs[28] im Unterricht mit

[28]Die Diagramme des Mr. Dobs können als Mindmap einer anderen Person/eines anderen Personenkreises gesehen werden.

Mindmaps zu vergleichen.

Eine der wichtigsten Eigenschaften des Mr. Dobs ist die Möglichkeit der direkten Manipulation eines Programms zur Laufzeit.

A Evolution des Mr. Dobs

Dieses Kapitel soll einzelne noch nicht beleuchtete Eigenschaften von Mr. Dobs darstellen. Des weiteren werden Vorschläge zur Erweiterung von Mr. Dobs gemacht. Diese Vorschläge ergeben sich aus den vorherigen Kapiteln. Der dieser Arbeit zugrundeliegende Mr. Dobs ist im Fujaba-Paket, in der Version 2.0.1, enthalten. Alle Screenshots, Eigenschaften und Kommentare beziehen sich auf dieses Release.

A.1 Methoden kommentieren

Ein Problem, was sich beim Testen eines "fremden" Programms gestellt hat, war, daß die Wirkungsweise und der "Sinn" der einzelnen Methoden eines Objekts verborgen waren. Soll mit Mr. Dobs ein Programm erkundet werden, ist es nötig den Einsatzzweck einzelner Methoden in Mr. Dobs angezeigt zu bekommen. Andernfalls wäre es notwendig, das Programm auch mit Fujaba zu analysieren. Um nicht das Problemfeld "verstehen von Algorithmen" aufzumachen, würde in Mr. Dobs eine "einfache" textuelle Kommentierung der Methoden ausreichen. Ich denke an ähnliche Texteinblendungen, wie bei "Tool Tips".

A.2 Pakete

Wie in 4.4 gezeigt, besteht ein Problem im Erkennen von Funktionen, die intern vom System gebraucht bzw. benutzt werden und den Funktionen, die zur Interaktion mit dem System bestimmt sind.
In C++ und Java existieren Schlüsselwörter zur Steuerung des Zugriffs auf die Funktionen und die Attribute einer Klasse. Duch das Schlüsselwort `private` werden Funktionen und Attribute markiert, die nur von der Klasse selbst benutzt werden sollen bzw. können. Das Schlüsselwort `protected` erlaubt es den "Kind"-Klassen auf diese Funktionen und Attribute zuzugreifen. Und durch das Schlüsselwort `public` werden Funktionen und Attribute einer Klassen für alle sichtbar. In Java besteht zusätzlich die Möglichkeit, Klassen zu einem Paket zusammenzufassen und Funktionen und Attribute durch `package` zu markieren. Auf so markierte Funktionen und Attribute,

können nur Klassen aus dem Paket selbst zugreifen.
Bezogen auf das Elevator-Demo könnten alle Klassen zu einem Paket `Elevatordemo` gehören und alle Assoziationen aus dem Klassendiagramm (⟶Abb. 20) als `package` markiert werden. Dadurch wären z.B. die Memberfunctions `setWantsTo()` und `getWantsTo()` aus der Klasse `Person` in Mr. Dobs nicht mehr sichtbar. Bei einer Erkundung des Elevator-Demos mit Mr. Dobs ständen sie also nicht mehr zur Verfügung und es wäre eher möglich, das "richtige" System zu finden.
In Fujaba existieren (in der Oberfläche) **alle** Möglichkeiten dieses auszudrücken, doch beim Export in Java Quelltextdateien werden diese Angaben ignoriert.
Eine Möglichkeit Methoden zu verbergen ist in A.3.2 beschrieben.

A.3 Preferences

Die folgenden Abschnitte beziehen sich auf Einstellungen, die Mr. Dobs anbietet bzw. anbieten sollte.

A.3.1 Icons für Objekte

In Mr. Dobs besteht die Möglichkeit Objekte nicht nur mit dem Namen ihrer Klasse anzuzeigen, sondern auch durch Icons. Damit besteht die Möglichkeit, Objekte eines Typs nicht nur anhand ihres (Klassen-)Namens zu identifizieren, sondern auch durch ein Bild (Icon).

A.3.2 Ausblenden von Methoden

Es besteht die Möglichkeit einzelne Methoden eines Objekttypus (generiert aus einer Klasse) auszublenden. Dadurch besteht die Möglichkeit den Fokus in Bezug auf eine Aufgabe auf die "relevanten" Methoden zu richten. Weiterhin können vererbte Methoden von "Eltern"-Objekten als ganzes oder einzeln ausgeblendet werden.

A.3.3 User Preferences

Mr. Dobs merkt sich alle vorgenommenen Einstellungen(s.o.). In Mehrbenutzerbetriebssystemen (Unix/Linux, Windows NT) werden diese lokal für

einen speziellen User vermerkt, so daß jeder User immer nur mit seinen Einstellungen arbeitet.

A.3.4 Session Preferences

Wünschenswert z.B. für den Einsatz in der Lehre wäre es, wenn bestimmte Einstellungen –assoziiert z.B. zu Projekten– lad- und speicherbar wären. Bezogen auf die in 4.4 durchgeführte Sitzung (Session) wäre ein profile zur Fokusssierung auf die wesentlichen Objekt(member)funktionen hilfreich gewesen. So könnte ein Satz Standartprofiles z.B. für eine "debug session" oder eine "Analyse session" (Schnittstellen zu einer Oberfläche aus-/einblenden) hilfreich sein.

B Newsgroup Diskussion

Folgende Newsgroup Diskussion ist im Internet unter `http://www.deja.com/usenet` zu finden. In der ersten Message wird eine Frage zum Thema: "Unterschied zwischen Objekten und Variablen" gestellt. An dem insgesamt 61 Messages umfassenden Thread beteiligen sich mehrere Leute, die aus unterschiedlichen Blickwinkeln, Vorerfahrungen, Wissen und daraus resultierendem Vokabular versuchten, die Frage zu klären. Diese Diskussion soll stellvertretend für z.B. eine Unterrichtssituation sein und auch dessen Verlauf und möglichen Ausgang exemplarisch verdeutlichen. Ich habe einige Messages, zur Fokussierung auf den Diskussionsverlauf gekürzt oder weggelassen.
Für die interessierten Leserinnen und Leser einige Fragen zur Ergründung der Probleme der Diskussion:

1. Aus welcher Perspektive werden die Begriffe Variable und Objekt gesehen? (Laufzeitebene/ Quelltextebee/ Java Sprachkonstrukt/...)
2. Welche Vorerfahrung haben die Verfasser in bezug auf objektorientierte Programmiersprachen und OOD? (Ergibt sich meist aus dem Vokabular)
3. Wieviel Erfahrung haben die einzelnen Verfasser in bezug auf Programmieren?

Forum: de.comp.lang.java
Thread: [LANG] Begriffsproblem, Variable ↔ Object
Message 1 of 61

Subject:[LANG] Begriffsproblem, Variable ↔ Object
Date:03/01/2000

Hallo Zielgruppe,
Ich hätte da mal wider eine saudumme Anfängerfrage ;-) Ich versuche mit Äaron Walsh; Die Java-Bibelßeit einiger Zeit Java zu lernen. Ich weiß nicht, ob der sich so blöde ausdrückt, dass ich es nicht verstehe, oder ob

ich so blöde bin, es nicht zu verstehen. Beispiel:

```
import java.applet.*;
import java.awt.*;
public class blahfasel extends ...
  int h;
  int v;
  AudioClip myAudio;

  public void start() {
    myAudio.loop();
  }
```

Mein Verständnis-Schwachpunkt setzt da ein, wo er meint, myAudio sei eine Variable. Nach stundenlangem Blättern und langsam erhöhtem Frustpegel meine ich, das ist ein Object der Klasse AudioClip, die in java.awt definiert ist?
Denn wie kann ich auf eine _Variable_ mit einer Konstruktion myAudio.loop() zugreifen? BTW, kennt jemand dieses Buch? "Taugt" es was? Ich stoße hier dauernd auf Ungereimtheiten und Ungenauigkeiten, die ich als Anfänger nicht interpretieren kann.

Message 2 of 61 :

[...] myAudio ist eine Variable der das Interface AudioClip zugewiesen wird.
[...]

Message 3 of 61 :

[...] > myAudio ist eine Variable der das Interface AudioClip zugewiesen wird.
Sch**** und ich dachte ich hätt es jetzt verstanden :-(
Gut, ich frag jetzt nicht, was "Interface" in dem Zusammenhang bedeutet
[...]

Message 4 of 61 :

[...] Nochmal um es zusammen zu fassen :
Variablen sind nur Platzhalter genauso wie in Pascal oder Delphi. Sie bezeichnen eine bestimmte Stelle im Speicher, wo der Wert der Variablen

gespeichert wird. myAudio ist eine Variable. Ein Objekt ist nun aus einem ganz anderen Gebiet. Vielleicht hast du in Pascal schon mal objektorientiert programmiert, ging zumindest mit Borland Pascal 7.0. Auch dort gibt es Objekte die in Variablen gespeichert werden. Soll heissen, die Variable ist nur der Name eines Objektes bzw. zeigt an, wo dieses Objekt im Speicher zu finden ist. In BP7.0 wurde das Konzept der Rekords um Prozeduren erweitert, so dass man einem Rekord spezielle Prozeduren zuordnen konnte, diese konnten dann über VariablenName.MethodenName aufgerufen werden, also genau wie in Java. [...]

Message 6 of 61 :

[...] Wenn man streng nach OO-Konzepten urteilt, ist myAudio weder eine Variable noch ein Objekt, sondern eine Assoziation bzw. Aggregation. [...]

Message 7 of 61 :

[...] Richtig ist: myAudio ist eine Variable vom Typ AudioClip. Da in Java alle Objekte auf dem Heap leigen, finden sich in den Variablen keine Objekte, sondern Verweise auf die wirklichen Objekte auf dem Heap. Die Variable myAudio kann also einen Verweis auf ein Objekt aufnehmen, welches eine Instanz einer Klasse sein muß, die das Interface AudioClip implementiert. Klar? [...]

Message 8 of 61 :

[...] myAudio ist aber auch eine Variable. Der Begriff "Variable" bezeichnet einfach nur einen veränderlichen Wert innerhalb eines Programms. Das kann sowohl ein Objekt als auch eine Primitive sein. Insofern ist jedes Objekt auch eine Variable. Primitive (z.B. 'ìnt i') sind ebenfalls Variablen, aber keine Objekte. [...]

Message 9 of 61 :

[...]

```
new ConstructorEinerKlasse();
```

erzeugt eine Objektinstanz, aber keine Variable, die dieses Objekt referenziert. [...]

Message 10 of 61 :

[...] myAudio ist eine Variable, denn sie ist variabel:

```
myAudio=new xAudio(5);
myAudio=otherAudio;
```

Ich kann ihr verschiedene (=variabel) objekte der klasse AudioClip zuweisen. Die variable ist sozusagen der behälter für die objekte. [...]

Message 11 of 61 :

[...] Der Begriff Variable bezeichnet einen Namensplatzhalter (Benenner) für ein ein Datum (Wert). Ob der Wert nun ein ganzes Objekt ist oder ein skalarer Wert (int, boolean, usw.) ist erst mal egal. Eine Objekt-Variable ist an sich auch nur ein Namensplatzhalter für ein "Referenz"-Wert (welcher wiederum auf eine Instanz einer Klasse verweist). [...]

Message 12 of 61 :

[...] myAudio ist eine Referenzvariable - eine Zeigervariable, und zwar eine, die auf objekte vom typ AudioClip zeigt, bzw. zeigen kann (sie kann naemlich auch auf "null" zeigen und auch nicht initialisiert sein.

und wenn die auf ein objekt zeigt (myAudio = new AudioClip()), dann kann man auf eigenschaften und methoden des entsprechenden objektes unter zuhilfenahme einer referenzvariablen zugreifen.

ein verstaendnis von oop - wie in einem anderen beitrag angemerkt - halte ich fuer das verstaendnis dieser sache nicht erforderlich. vielmehr hilft es zu verstehen, wie zeiger im allgemeinen funktionieren, und wie man zur laufzeiut speicher anfordert und dann eine zeigervariable auf diesen speicherbereich zeigen laesst, um so auf einzelne elemente (im falle von datenstrukturen) in diesem speicherbereich zuzugreifen.

das einzige, was dann noch hinzukommt, um java zu verstehen, ist, dass in dem speicherbereich nicht nur datenelemente liegen, sondern auch methoden, auf die man fast genauso zugreift, wie auf die daten elemente.

ein ganz wichtiger unterschied zu pascal und c/c++ ist, dass komplexe datentypen (also jede art von klassen und objekt) IMMER (ob nun explizit oder implizit) mit new angefordert werden muss - und das variablen von dieser klasse IMMER zeigervariablen sind, und niemals selbst objekte von diesem typ darstellen.

das hier geht z.b. nicht:

```
AudioClip myAudio;
myAudio.play();
```

das geht nicht, weil die referenz/zeigervariable myAudio ja noch auf gar kein reales AudioClip objekt zeigt.

und noch schlimmer: myAudio hat, da gerade erst deklariert, im moment noch einen zufaelligen wert, ist also uninitialisiert! zum glueck weist der java compiler einen aber auf probleme mit uninitialisierten variablen hin.

das hier ist der richtige weg:

```
AudioClip myAudio;           // nicht initialisiert, zeigt auf kein objekt
                             // und IST auch KEIN objekt

myAudio = new AudioClip();   // neues objekt vom typ AudioClip
                             // erzeugen und myAudio darauf
                             // zeigen lassen, damit...

myAudio.play();              // ...auch endlich funktioniert,
                             // bzw. ohne probleme zur laufzeit
                             // moeglich ist
```

(ob audioclip objekte nun so erstellt werden, und man sie so abspielt, weiss ich nicht, es sollen bloss beispiele sein.) [...]

Message 14 of 61 :

[...] [in Message 1:]
> Mein Verständnis-Schwachpunkt setzt da ein, wo er meint, myAudio
> sei eine Variable. Nach stundenlangem Blättern und langsam
> erhöhtem Frustpegel meine ich, das ist ein Object der Klasse
> AudioClip, die in java.awt definiert ist?

Vielleicht arbeitet der Autor hier nicht ganz sauber mit der Begriffstrennung.

Na ja, also in der prozeduralen/imperativen Programmierung, nennt man die Dinger, in denen man Infos speichert Variablen und in der objektorientierten Programmierung nennt man eben diese Dinger Objekte. Objekt und Variable meint also im Kern dasselbe.

Ähnliches gilt für Klassen. Klassen legen den Aufbau der von ihnen erzeugten Objekte fest, genauso wie Datentypen den Aufbau von Variablen festlegen.

Im Grunde kann man also Klasse/Typ und Objekt/Variable synonym verwenden.

Dann kommt noch das Interface, also die Schnittstelle hinzu. Auch das gibt es in beiden Paradigmen: Nur in der OOP ist die Schnittstelle in der Klasse festgelegt, d.h. die Methoden einer Klasse bilden ihre Schnittstelle. In der imperativen Programmierung baut man Module in denen es meist für dieses eine Modul globale Variablen gibt und halt etliche Funktionen, die mit diesen Variablen arbeiten. Von anderen Programmteilen aus kann man dann halt nur auf die Funktionen zugreifen und die Variablen sind quasi privat - wie bei Objekten und Klassen.

Daher kann man im Grunde auch in prozeduralen Sprachen wie ISO-Pascal oder C durchaus objektorientiert Programmieren und daher werden viele Begriffe synonym verwendet.

Ach ja: In Java musst du natürlich noch zwischen dem allgemeinen Begriff einer Schnittstelle im obigen Sinne und dem Schlüsselwort ïnterfaceünterscheiden (letzteres ermöglicht quasi eine Art Mehrfachvererbung in einer problemlosen Variante im Gegensatz z.B. zur Sprache C++). [...]

Message 15 of 61 :

[...] [in Message 14:]
> Na ja, also in der prozeduralen/imperativen Programmierung, nennt man
> die Dinger, in denen man Infos speichert Variablen und in der
> objektorientierten Programmierung nennt man eben diese Dinger
> Objekte. Objekt und Variable meint also im Kern dasselbe.

Nein. Das sind zwei völlig verschiedene Dinge. Ein Objekt ist ein Exemplar einer Klasse (die den Aufbau der Objekte beschreibt und ihnen somit eine einheitliche Schnittstelle gibt), also praktisch eine Menge von Daten, auf

die in der Klasse definierte Operationen ("Methoden") angewendet werden können.

Eine Variable ist ein Name oder Platzhalter für ein "Ding". In Java können diese Dinge primitive Werte oder Objektreferenzen (also Verweise auf Objekte oder die "null"-Referenz, also ein leerer Verweis) sein.

Direkt mit Objekten arbeitet man in Java sowieso nicht, sondern nur mit Objektreferenzen, die mit verschiedenen Arten von Namen bezeichnet werden können.

Beispiel: "new MyClass()" erzeugt ein neues Objekt als Exemplar der Klasse "MyClass". Aus syntaktischer Sicht ist "new MyClass()" nun ein Name für die Referenz auf dieses Objekt. Man kann mit diesem Namen z.B. eine Methode aufrufen, die das Objekt versteht, also etwa "new MyClass().init()". Oder man fürt einen neuen Namen, also eine Variable, ein und weist dieser die durch "new MyClass()" bezeichnete Objektreferenz zu: "MyClass x =3D new MyClass()".

Da Java eine streng typisierte Sprache ist und die Variable x mit dem Typ "MyClass" qualifiziert wurde, verhindern Compiler und Laufzeitumgebung, daß man an x eine Referenz auf ein Objekt zuweisen kann, das nicht die in der Klasse "MyClass" beschriebene Schnittstelle besitzt oder daß man über den Namen "x" eine Methode aufruft, die Exemplare von "MyClass" nicht verstehen.

Mit primitiven Werten ist es auch nicht anders, aber man kann das Prinzip schlechter erkennen. Ein int-Literal wie "42" ist auch nur ein Name für den int-Wert 42. Mit "int i =3D 42" wird dieser Wert der Variablen i zugewiesen. i ist jetzt also ebenfalls ein Name für den Wert 42.

Hoffentlich habe ich mich bei den Definitionen nicht irgendwo verzettelt. Falls doch, finden sich in dieser Gruppe bestimmt genügend Leute, die darauf hinweisen werden :-) [...]

Message 18 of 61 :

[...]

>> Habe gerade nachgeschlagen:

>>

>> MyClass MyObject; // Deklaration eines Objekts

>

>Deklaration einer Referenz-Variable, die in Zukunft auf ein Objekt

>der Klasse MyClass zeigen soll.

[. . .]

Ich finde den Terminus “Deklaration einer Referenz-Variablen” im uebrigen sehr fraglich. Der Ausdruck “Deklaration eines Objekts” trifft es irgendwie besser.

[. . .]

Message 19 of 61 :

[. . .] Eine Variablendeklaration ist kein Objekt ist keine Variable! (mit dem Zusatz: es gilt auch für statische Objekte)

> Ich finde den Terminus
> “Deklaration einer Referenz-Variablen” im uebrigen sehr fraglich.
> Der Ausdruck "Deklaration eines Objektstrifft es irgendwie
> besser.

Nein, nicht wenn mann mit mehr technischen Verständniss an die Sache rangehen möchte. Wenn eine Variable deklariert wird und diese für ein Objekt ist und das Objekt noch nicht existiert, dann haben wir eine Referenz-Variable und kein Objekt. Wird von der Deklaration eines Objektes ausgegangen, dann sollte die Repräsentation eines Objektes (die Attribute) ja dann schon mit MyClass MyObject; erzeugt werden. [. . .]

Message 24 of 61 :

[. . .]

“Der Unterschied zwischen richtig und fast richtig ist wie der Unterschied zwischen einem Blitz und einem Gluehwuermchen.”

Sorry, wenn ich jemanden mit dem Gebohre genervt habe, aber ich denke, die Diskussion hat einiges verdeutlicht.

[. . .]

Message 29 of 61 :

[. . .]

> Na ja, wie man’s nimmt. :)

Also. Erzeugt die Deklararion code oder nicht?

Langsam gehst Du mir auf den Keks. Wenn Du keine Ahnung hast, solltest Du Dich informieren, bevor Du hier in einer so autoritativen Weise postest, als hättest Du den Computer erfunden [...]

Message 30 of 61 :

[...] >> Aber es wäre so oder so eh nur Erbsenzähler, also was soll's.

> Interessant. Wenn Du was nicht raffst, ist es Erbsenzählerei. Wenn
> andere aber nach Deiner erlauchten Meinung Deinen Ausführungen nicht
> folgen können, sind sie Möchtegernprogrammierer.

> Naja, mach nur weiter so. Ich ziehe mich hiermit aus der Diskussion
> zurück.

Diskussion? Interessante Diskussionskultur. [...]

Message 32 of 61 :

[...] PS.: ohje, ich glaub wir sollten uns das Ursprungsposting mal wieder anschauen. Ich glaube, die Frage wurde nun schon hinreichend beantwortet... [...]

Aber das Rad der Diskussion drehte sich weiter...
Im folgenden werden die Messages immer länger und zum Teil auch schärfer im Ton.

Ich kann mir vorstellen, diese Diskussion im Unterricht (kurz vor dem Abi) unter ähnlicher Fragestellung zu behandeln, da zum einen zur Beantwortung der Frage eine klare Vorstellung von den Ebenen Laufzeit und Quelltext nötig ist und zum anderen die Diskussion den Blick auf andere -auch korrekte- Sichtweisen öffnet.
Innerhalb eines Softwareentwicklungsprozeß werden Diskussionen von noch unterschiedlicheren (Interessen-) Gruppen geführt (Anwender/ Auftraggeber/ Entwickler/...).

Literaturverzeichnis

[Amb98] Amber, Scott W.: *CRC Modelling: Bringing the Communication Gap Between Developers and Users.* AmbySoft Inc. White Paper, www.ambysoft.com, 1998. 5, 10

[Bel98] Belli, Fevzi: *Methoden und Hilfsmittel für die systematische Prüfung komplexer Software.* Informatik-Spektrum (Springer), (21):337–346, 1998. 11

[Bre96] Breymann, Ulrich: *C++: Eine Einführung.* Hanser, 3 Auflage, 1996. 3.4, 3.5

[Dud90] Duden: *Das Fremdwörterbuch*, Band 5. Dudenverlag, 1990. 3.4, 10, 26

[Fla99] Flanagan, David: *JAVA in a Nutshell,A Desktop Quick Reference.* O'REILLY, 3 Auflage, 1999. 1, 2

[Fow] Fowler, Martin: *Why Use Analysis and Design Techniques?* http://www2.awl.com/cseng/titles/0-201-89542-0/techniques/whyUse.htm. 5

[GIF99] *GI-Fachausschuß 7.3: Informatische Bildung und Medienerziehung*, 1999. 2, 2, 1

[Hub99] Hubwieser, Peter: *Modellieren in der Schulinformatik.* Log In, 19(1):24–29, 1999. 1, 3.5

[Leh95] Lehmann, Eberhard: *Komplexe Systeme, Eine fundamentale Idee im Informatikunterricht.* Log In, 15(1):29–37, 1995. 2.1, 5.4, 2, 28

[Mag00] Magenheim, Johann S.: *Informatiksystem und Dekonstruktion als didaktische Kategorien.* U-GH Paderborn, FB 17, DDI, internes Papier, 2000. 2.2, 5.1, 27

[MSH99] Magenheim, Johann S., Carsten Schulte und Thorsten Hampel: *Dekonstruktion von Informatiksystemen als Unterrichts-*

methode - Zugang zu objektorientierten Sichtweisen im Informatikunterricht. In: *Informatik und Schule*, Seiten 149–165, 1999. 2.1

[NS99] NOACK, JÖRG und BRUNO SCHIENMANN: *Objektorientierte Vorgehensmodelle im Vergleich*. Informatik-Spektrum (Springer-Verlag), 22:166–180, 1999. 3, 4

[Oes] OESTEREICH, BERND: *Glossar für das Themengebiet UML*. http://www.oose.de/uml. 3.2

[Pro99] PROJEKTGRUPPE, FUJABA: *Fujaba Homepage*. U-GH Paderborn, http://www.uni-paderborn.de/cs/ag-schaefer/ag_dt/PG/Fujaba/main.html, 1999. 21

[Sie96] SIEFKES, DIRK: *Umdenken auf kleine Systeme - Können wir zu einer ökologischen Orientierung in der Informatik finden?* Informatik-Spektrum (Springer), (19):141–146, 1996. 2, 2.1

[Ste99] STEINKAMP, DIRK: *Informatik-Experimente im Schullabor*. DDI Universität Dortmund, 9 1999. 1, 2.3

C Versicherung

Ich versichere, daß ich die schriftliche Hausarbeit einschließlich evtl. beigefügter Zeichnungen, Kartenskizzen und Darstellungen selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Alle Stellen, die dem Wortlaut oder dem Sinne nach anderen Werken entnommen sind, habe ich in jedem Falle unter genauer Angabe der Quelle deutlich als Entlehnung kenntlich gemacht.