Studienarbeit **Universität Hamburg**

Fachbereich Informatik **Arbeitsbereich Softwaretechnik**

Entwicklung eines Werkzeugs zur

Visualisierung des Objektverhaltens

in C++-Programmen

Keno Hamer, Stephen Friedrich

Betreuung: Prof. Dr. H. Züllighoven **April 1996**

Keno Hamer
Warnholtzstr. 7
22767 Hamburg
E-Mail: `1hamer@informatik.uni-hamburg.de`

Stephen Friedrich
Mettlerkampsweg 2
20535 Hamburg
E-Mail: `1friedri@informatik.uni-hamburg.de`

Prof. Dr. H. Züllighoven
Arbeitsbereich Softwaretechnik
Fachbereich Informatik
Universität Hamburg
Vogt-Kölln-Str. 30
22527 Hamburg
E-Mail: `zuelligh@informatik.uni-hamburg.de`

Inhaltsverzeichnis

Teil A: Konzepte 5

1. Einleitung........ 5

2. Werkzeuge für die objektorientierte Software-Entwicklung 7
    2.1. Analyse und Entwurf........ 7
    2.2. Implementierung........ 8
    2.3. Und nach der Implementierung?........ 9
    2.4. Möglichkeiten eines Visualisierungs-Werkzeugs 10
3. Dynamisches Verhalten von Objekten........ 12
    3.1. Beziehungen zwischen Klassen........ 13
    3.2. Beziehungen zwischen Objekten........ 13
    3.3. Überlegungen zu graphischen Notationen 14
        3.3.1. Objektdiagramme in OMT und bei Booch........ 14
        3.3.2. Objektdiagramme als Basis für ein Software-Werkzeug........ 15
    3.4. Basis der Visualisierung: Drei Diagrammtypen 16
        3.4.1. Konstruktions-Diagramm........ 16
        3.4.2. Referenz-Diagramm........ 17
        3.4.3. Interaktions-Diagramm........ 18
4. Möglichkeiten einer integrierten Entwicklungsumgebung 19
    4.1. Bestehende Entwicklungsumgebungen........ 19
    4.2. Visualisierung der Programmausführung........ 19
    4.3. Schwierigkeiten der Integration........ 20
5. Überblick über ähnliche Arbeiten........ 21
5.1. Unterstützung während des Entwurfs........ 21
5.2. Untersuchung implementierter Systeme........ 21
5.2.1. Objektorientierte Datenflußanalyse........ 21
5.2.2. Beobachtung der Ausführung eines Programms 24
5.2.2.1. Visualisierung post-mortem........ 25
5.2.2.2. Visualisierung während des Programm-Ablaufs 28

Teil B: Der Prototyp Voop 31

6. Objektorientierung - vom Konzept zum Object-Code........ 31
    6.1. Konzepte objektorientierter Programmiersprachen 31
    6.2. Umsetzung objektorientierter Konzepte in C++........ 32
    6.3. Analyse von C++-Programmen........ 33
        6.3.1. Konstruktoren und Destruktoren........ 33
        6.3.2. Lebensdauer von Objekten........ 33
        6.3.3. Objekt-Identität über this-Pointer 34
        6.3.4. Vererbung in C++........ 34
        6.3.5. Templates........ 39
        6.3.6. Arrays........ 40
7. Voop – status quo 41
    7.1. Funktionalität und Oberfläche des Prototypen 41
    7.2. Überblick über die Architektur........ 42
        7.2.1. Der Rahmen der Anwendung........ 43
        7.2.2. Eine Meta-Ebene für das untersuchte Programm 43
        7.2.3. Elemente der graphischen Oberfläche........ 44
    7.3. Implementation, Diskussion von Alternativen........ 45
        7.3.1. Beobachtung und Steuerung des untersuchten Programms . 45
        7.3.2. Name-mangling........ 49
        7.3.3. Einsatz des Boebachtermechanismus 49

8. Möglichkeiten der Fortführung 51
8.1. fachliche Erweiterungen 51
8.1.1. dargestellte Diegramme 51
8.1.2. Objektbeziehungen 51
8.1.3. Protokollieren von Durchläufen 52
8.1.4. Konfiguration des Werkzeugs 52
8.1.5 Einbettung in eine Anwendungsumgebung 52
8.2. technische Verbesserungen 52
8.2.1. Portabilität 53
8.2.2. Performanz 53
8.2.3. Beobachten von Objekt-Verweisen 54
9. Anhang 55
9.1. Syntax der „Stabs“-Debug-Anweisungen 55
9.2. Literaturverzeichnis 57

Teil A: Konzepte

1. Einleitung

Die objektorientierte Entwurfs- und Programmiermethodik hat sich längst zum Paradigma entwickelt, das die Softwareentwicklung der 90er Jahre beherrscht. Der Erfolg gründet sich auf die Aussicht, auch komplexe Systeme geeignet modellieren zu können und zu transparenteren Programmen zu gelangen. Die Entwicklungszeiten lassen sich durch die vereinfachte Wiederverwendung von Programmtext verkürzen, die Software ist leichter verständlich und einfacher zu warten. So zumindest lauten einige Ansprüche der objektorientierten Programmierung.

Um solche Ansprüche einzulösen, sind zwei Voraussetzungen unentbehrlich: Einerseits müssen die Entwickler mit den objektorientierten Konzepten vertraut sein und diese sinnvoll einsetzen können, andererseits werden Software-Werkzeuge gebraucht, die dabei die notwendige Unterstützung leisten. Wem diese Voraussetzungen zur Verfügung stehen, der hat gute Chancen, erfolgreich objektorientiert zu entwickeln.

Erfolgreiche Softwareentwicklung bedeutet auch, von der Erfahrung und den Ergebnissen anderer zu profitieren. Objektorientierte Programmierung eignet sich hierzu wie sonst keine andere Technologie. Der erste Schritt in diese Richtung waren Klassenbibliotheken, zu denen etwas später Frameworks kamen. Die aktuelle Entwicklung spiegelt sich in der Diskussion um Entwurfsmuster (design patterns) wider, durch die eine einfache Wiederverwendung nicht nur von Source-Code, sondern auch von Abstraktionen auf Entwurfsebene gelingt.

Der Entwickler, der diese Formen der Wiederverwendung nutzen möchte, muß verstehen, in welchem Zusammenhang die Klassen einer Bibliothek, eines Frameworks oder eines Entwurfsmusters stehen. Insbesondere muß er auch überblicken, wie die Kooperation der Objekte während der Programmausführung geschieht: Das *dynamische Verhalten von Objekten* zu begreifen ist der Schlüssel zum Verständnis objektorientierter Programme.

Die Tätigkeiten der Softwareentwicklung gehen noch darüber hinaus; um beispielsweise ein Programm zu debuggen, muß das dynamische Verhalten nicht nur verstanden, sondern auch effektiv analysiert werden können. Wer nach Fehlern im Source-Code sucht oder Performanzprobleme in den Griff bekommen will, ist gezwungen, sich zielgerichtet mit dem dynamischen Verhalten des Programms auseinanderzusetzen.

In der Realität ist jeoch festzustellen, daß die verfügbaren Methoden und Werkzeuge die statischen Aspekte der Programmentwicklung (unterschiedlich) gut, die dynamischen Aspekte in der Regel aber nur unzureichend abdecken.

In imperativen Programmen gibt es im wesentlichen einen Kontrollpfad durch das Programm, der durch Prozeduraufrufe geschachtelt ist, sowie mehr oder weniger komplexe dynamische Datenstrukturen. Gerade in objektorientierten Programmen läßt sich jedoch eine gesteigerte dynamische Komplexität beobachten: Anstelle des einen Kontrollpfades durch das Programm gibt es jetzt eine Menge kleinerer Einheiten (Objekte), die zur Laufzeit entstehen, gelöscht werden und durch ihre Interaktion die Funktionalität des Programms ausmachen.

Vor diesem Hintergrund haben wir begonnen, ein Werkzeug zu entwickeln, daß das dynamische Verhalten von C++-Programmen graphisch darstellt und in Form von animierten Diagrammen auf dem Bildschirm visualisiert. Als Ergebnis können wir einen funktionsfähigen Prototypen vorweisen. Die Entwicklung dieses Programms wollen wir mit der vorliegenden Arbeit dokumentieren, darüber hinaus aber auch unabhängig von unserem Programm den Bedarf, die Konzepte und die Möglichkeiten eines derartigen Werkzeugs vermitteln.

Der Aufbau der Arbeit ist daher zweigeteilt: Der erste Teil (Kapitel 2 bis 5) beschäftigt sich mit der allgemeinen Konzeption, die wir für ein Visualisierungs-Werkzeug ausgearbeitet haben. Im zweiten Teil (Kapitel 6 bis 8) gehen wir gezielt auf den von uns entwickelten Prototypen ein.

In Kapitel 2 stellen wir die Kategorien von Werkzeugen vor, die für die objektorientierte Anwendungsentwicklung gebräuchlich sind. Das Kapitel soll die Motivation für unsere Entwicklungsarbeit deutlich machen und diese in den Zusammenhang bestehender Werkzeuge einordnen.

Kapitel 3 hat die grundlegenden Beziehungen zwischen Klassen und zwischen Objekten zum Inhalt. Die Objektrelationen werden untersucht und benannt, und es werden Formen der graphischen Darstellung vorgestellt. Wir werden drei Diagrammtypen beschreiben, die wir als geeignete Grundlage einer Programm-Visualisierung ansehen.

In Kapitel 4 werden wir eine Werkzeugvision beschreiben, die auf den Diagrammtypen aufsetzt und eine konkrete Vorstellung eines möglichen Werkzeugs schaffen soll.

Der erste Teil dieses Textes endet mit dem 5. Kapitel, in dem wir auf Arbeiten anderer Autoren eingehen, die sich mit ähnlichen Themen beschäftigt haben.

Weil das Werkzeug, das wir prototypisch entwickelt haben, der Visualisierung von C++-Programmen dient, beschreibt Kapitel 6, wie objektorientierte Konzepte in der Sprache C++ realisiert sind, und geht auf einige C++-Interna ein, mit denen wir uns notwendigerweise vertraut machen mußten.
Den Stand unserer Entwicklungsarbeit hält Kapitel 7 fest. Dort werden Funktionalität und Bedienung des Prototypen beschrieben. Der programmtechnische Aufbau wird ebenso erklärt wie die wichtigsten Entwurfsentscheidungen, die dem Programm zugrunde liegen.
Kapitel 8 geht schließlich auf die Ansatzpunkte ein, an denen die Weiterentwicklung des Prototypen sinnvollerweise erfolgen sollte.

2. Werkzeuge für die objektorientierte Software-Entwicklung

Objektorientierte Softwareentwicklung stellt einen umfangreichen Prozeß dar, der mit der Analyse des Anwendungsbereichs beginnt und in vielen Schritten zu einem lauffähigen Programm führen soll. Als Zwischenergebnisse entstehen zahlreiche Dokumente, z.B. Szenarien, Wörterbücher und Prototypen. Wir nehmen für den Entwicklungsprozeß ein zyklisches, evolutionäres Vorgehen an (vgl. [Flo95], [KGZ93]). Das impliziert, daß bestimmte Tätigkeiten im Zeitverlauf wiederholt und viele Dokumente zusätzlich erstellt oder revidiert werden.
Der Umfang dieses Prozesses und der damit einhergehende Dokumentenreichtum erfordern eine umfassende Unterstützung durch Software-Werkzeuge – den Entwicklern nicht mehr als Editor, Compiler und Linker und darüber hinaus Papier und Bleistift zur Verfügung zu stellen, ist weder der Aufgabe noch den heutigen Möglichkeiten angemessen.
Dieses Kapitel gibt zunächst eine kurze Übersicht über vorhandene Werkzeuge. Wir wollen deutlich machen, wie unser Konzept eines Visualisierungs-Werkzeugs in die Reihe der Werkzeuge einzuordnen ist, und welche Fähigkeiten es bietet.
Die meisten Entwicklungswerkzeuge lassen sich einer der „klassischen" Phasen Analyse, Entwurf, Programmierung, Test und Integration zuordnen. Auch bei einem Entwicklungsprozeß, der nicht streng nach dem Phasenmodell betrieben wird, sondern zyklisch und evolutionär angelegt ist, bleiben die Werkzeuge an eine der verschiedenen Tätigkeiten gebunden. Der größte Nutzen entsteht, wenn sich die Tools über möglichst große Bereiche der Entwicklung erstrecken und sinnvoll miteinander kombiniert werden können. Eine derartige Werkzeugunterstützung hilft, Modellbrüche zu vermeiden. Da es kein einzelnes Werkzeug gibt, das den ganzen Prozeß alleine abdeckt, besteht also der Bedarf nach einer Zusammenstellung von Werkzeugen, die

- zumindest für objektorientierte Entwicklung,
- besser noch für eine bestimmte Methode konzipiert wurden,
- und im Rahmen einer Werkzeugumgebung integrierbar sind.

2.1. Analyse und Entwurf

Das Vorgehen bei Analyse und Entwurf richtet sich nach den verwendeten Methoden und Entwurfsmetaphern. Im wesentlichen handelt es sich um Arbeiten des Untersuchens, Strukturierens und Systematisierens des Anwendungsbereichs. Von den Methoden vorgesehene Hilfsmittel sollten so weit wie möglich durch Software zur Verfügung gestellt werden. Wir können zu diesem Zweck folgendes Angebot an Tools ausfindig machen:

- Graphik-Editoren
  Fast alle Methoden verwenden graphische Notationen. Typische Beispiele sind Klassenhierarchien und Entity-Relationship-Diagramme. Um Änderungen einfach vorzunehmen und aus den z.T. sehr großen Diagrammen einzelne Sichten extrahieren zu können, ist eine Computerunterstützung ausgesprochen sinnvoll. Im einfachsten Fall kann diese Unterstützung durch allgemeine, vektororientierte Zeichen- oder Flow-Chart-Programme erfolgen.
  Beispiel: Corel Flow
- Textsysteme
  Für textuelle Entwicklungsdokumente (Szenarios, Visionen, Data Dictionaries etc.) sollten hypertext-ähnliche Dokumente, die eine einfache Verknüpfung der Texte ermöglichen, erstellt werden. Im einfachsten Fall stehen dazu Editoren für Hypertext-Standard-Formate wie HTML oder MS-Windows-Help-Format zur Verfügung.
  Wesentlich komfortabler sind auf den Entwicklungsprozeß und die Struktur der Dokumente abgestimmte Editoren.
- Klassenbrowser
  Auch wenn noch keine Klassen implementiert wurden, können Klassenbrowser schon dazu dienen, die entworfenen Klassendefinitionen übersichtlich zu verwalten. Beziehungen zwischen Klassen werden berücksichtigt.
  Beispiel: Sniff+
- Objektorientierte CASE-Tools:
  Für die verschiedenen Methoden objektorientierter Entwicklung gibt es Werkzeuge, die zum großen Teil die oben aufgezählten Werkzeuge in sich vereinen und miteinander verbinden. Aus dem Modell, das so entsteht, werden von den meisten Tools Codefragmente erzeugt, in einzelnen Fällen sind auch Fähigkeiten des Reverse Engineering enthalten, indem nachträglich aus dem Source-Code Entwurfsdokumente abgeleitet werden.

Beispiele nach [Sch94]:
- OMTool (OMT)
- Paradigm Plus (OMT, Booch, Coad-Yourdon, Fusion u.a.)
- StP („Software through Pictures“) (OMT)
- OOATool (Coad-Yourdon)
- Together++ (Coad-Yourdon)
- Rational Rose (Booch)
- Objectory (Jacobsen)

2.2. Implementierung

Das Ergebnis des Entwurfs ist ein objektorientiertes Modell, das auf einer Rechnerplattform implementiert werden kann. Die Anforderungen, die an die Werkzeuge zur Programmierung gestellt werden müssen, hängen nur noch mittelbar von der Entwurfsmethode ab. Ausschlaggebend sind nun eher technische und softwaretechnische Argumente (z.B. die Zielplattform oder Vorgaben des Auftraggebers, welche Programmiersprache zu benutzen ist). In begrenztem Maß ist es möglich, einen objektorientierten Entwurf in einer nicht-objektorientierten Sprache umzusetzen (vgl. [Mey88] Kap. 17, [RBP91] Kap. 18, [KGZ93] Kap. 8.1.). Wir gehen jedoch von der Verwendung einer objektorientierten Sprache aus. In diesem Kontext sind die wesentlichen Werkzeuge:

- Code-Generatoren
  Zu den verbreitetsten Code-Generatoren zählen visuelle Enwicklungstools zur Oberflächen-Erstellung (*GUI builder*). Ähnliche Werkzeuge liegen im Datenbank-Bereich vor, um aus Datenmodellen Programmgerüste zu erzeugen. Code-Generatoren können in der Regel nur Fragmente des Programms produzieren, die einen bestimmten Teil der Funktionalität bewirken, z.B. die Oberflächensteuerung. Der Anwendungskern wird dann in das generierte Gerüst integriert.
  Die objektorientierte Programmierung profitiert von der Verwendung verfügbarer Klassenbibliotheken, denen in vielen Fällen Code-Generatoren beiliegen und die Benutzung der Bibliothek erheblich erleichtern.
  Beispiel: StarView
- Entwicklungsumgebung
  Eine leistungsfähige Entwicklungsumgebung muß Compiler, Linker und Debugger unter einer gemeinsamen Oberfläche zusammenfassen. Ein komfortabler Editor ist unbedingt notwendig. Für objektorientierte Entwicklungsumgebungen gehören Klassenbrowser zum Standard. Die Verwendung von Versionsverwaltungen sollte unterstützt werden.
  Beispiele: Sniff+, VisualAge
- Compiler
  Für die bekannten objektorientierten Sprachen gibt es unter den gängigen Betriebssystemen verschiedene professionelle Compiler. Nachteilig sind Pre-Compiler, die eine objektorientierte Sprache in nicht-objektorientierten Code übersetzen, weil dadurch das Debuggen des übersetzten Programms u.U. erheblich erschwert wird. Im übrigen gilt bei der Übersetzung die gleiche Forderung nach *inkrementellen* Compilern, wie sie nachfolgen für Linker beschrieben wird[1].
- Linker
  Prinzipiell hängt die Verwendung eines Linkers nicht davon ab, ob Software objektorientiert entwickelt wird. Objektorientierte Konzepte ermöglichen aber, die Funktionalität eines Programms auf relativ kleine Einheiten (Klassen) zu verteilen. Die Wartezeiten für Compiler und Linker sollten daher nicht unnötig lang sein, wenn nur eine einzige Klasse geändert wurde. Wie lang diese Zeiten ausfallen, hängt davon ab, wie in der benutzten Sprache Modularisierung verwirklicht wird. Für C++ ergibt sich die Forderung nach inkrementellen Linkern, um zu vermeiden, daß jeder Linkeraufruf alle verwendeten Klassen neu bindet.

Wir haben bereits darauf hingewiesen, daß eines der großen Potentiale objektorientierter Programmierung in der Wiederverwendung von Klassen und Entwurfsmustern besteht. Die Entscheidung, welche vorhandenen Ressourcen eingesetzt werden sollen, muß teilweise schon während des Entwurfs, spätestens aber während der Implementierung gefällt werden. Zum Zeitpunkt der Implementierung müssen die Entwickler mit den benutzten Klassenbibliotheken, Frameworks und Entwurfsmustern vertraut sein. Um die Einarbeitung zu erleichtern, sind auch hier passende Werkzeuge ange-

[1] „It is unaccetable to require a ten minute compile and link cycle simply to add a field to a high-level super-class! Incrementally compiled methods and incrementally compiled … field definitions are a must for quick debugging.“ N. Meyrowitz, zitiert nach [Sch94].

bracht. Wir sehen einen Bedarf an Tools, die das dynamische Verhalten der Bibliotheken-, Framework- und Muster-Architekturen erfahrbar machen. Wir werden später auf diesen Einsatzzweck zurückkommen.

2.3. Und nach der Implementierung?

Für die bisher beschriebenen Typen von Werkzeugen können wir feststellen, daß es für die Schritte von Analyse, Entwurf und Programmierung eine Auswahl an Tools gibt, die an den Erfordernissen objektorientierter Entwicklung ausgerichtet sind. Bei den Tätigkeiten, die nach der (ersten) Implementierung stattfinden, herrschen konventionelle Werkzeuge vor, die nur eine schwache Unterstützung von objektorientierten Technologien bieten. Zum Teil sind solche Werkzeuge zwar verfügbar, aber nur wenig verbreitet. Das ist um so erstaunlicher, wenn man bedenkt, wie groß der Anteil von Wartungs- und Korrekturtätigkeiten bei der Softwareentwicklung ist. Am ehesten findet man noch Werkzeuge zum Reverse Engineering, die z.B. Klassenhierarchien und -dokumentationen aus bereits geschriebenen Programmen ableiten können.
Das wesentliche Hilfsmittel, um ein implementiertes Programm zu bearbeiten, ist ein Debugger: Das Auffinden der (immer vorhandenen) Fehler ist ohne einen guten Debugger ein ausgesprochen mühsames Unterfangen. Das Debuggen dient dem Qualitätsmerkmal *Korrektheit*; ein anderes Merkmal ist *Performanz*. Um diese zu verbessern, werden Profiler eingesetzt, mit denen der Ressourcenverbrauch eines Programms untersucht werden kann.
Betrachtet man, welche Debugger und Profiler angeboten und eingesetzt werden, so fällt auf, daß sich objektorientierte Technologien nur in einem geringen Maß durchgesetzt haben. Beim Debuggen objektorientierter Programme ist die Minimalanforderung, daß der Debugger das Konzept „Objekt“ kennt und damit in der Lage ist, statischen Typ und Attribute eines Objekts korrekt anzuzeigen. Die aktuellen Versionen der verbreiteten Debugger besitzen diese Fähigkeit, gehen aber nicht weiter auf die Eigenschaften objektorientierter Programme ein[2] – beim Debuggen ist der Entwickler gezwungen, nicht objekt-, sondern kontrollflußorientiert vorzugehen. Ein objektorientiertes Vorgehen würde es erlauben ein Objekt als Ganzes anzusprechen, das Setzen von Breakpoints auf ein oder mehrere bestimmte Objekte zu beschränken, den implizit an Methoden übergebenen this-Pointer des aufgerufenen Objekts dem Benutzer nicht zu zeigen, polymorphe Aufrufe von Methoden richtig zu erkennen und anzeigen u.v.a.
Es scheint, als seien die Debugger auf dem Stand der strukturierten Programmierung stehengeblieben, und man erhält den Eindruck, daß in der objektorientierten Softwarekonstruktion nach den Entwurfsmethoden der 90er Jahre die Implementierungssprachen der 80er und die Debugger der 70er benutzt werden.
Nach unseren Vorstellungen muß das Debuggen objektorientierter Programme grundsätzlich anders erfolgen:

- *Visualisierungen*: Wir können davon ausgehen, daß dort, wo objektorientiert entwickelt wird, auch graphikfähige Hardware vorhanden ist. Warum sollten Debugger also mit einer Benutzeroberfläche daherkommen, die offensichtlich für Zeichenterminals geschaffen wurde? Nachdem bei der Entwicklung seit langem Graphiken eingesetzt werden, um Klassen und Objekte darzustellen, müssen wir diese Form auch für Werkzeuge nutzen, die auf dem implementierten Programm arbeiten. Daß die bildliche Umsetzung eines Sachverhalts das Verständnis beträchtlich erleichtern kann, ist hinreichend bekannt. Kapitel 3 wird sich mit möglichen Darstellungsformen befassen, um den Ablauf eines objektorientierten Programms sinnvoll anzuzeigen.
  Objekt- statt kontrollflußorientiert: Der Entwickler muß davon befreit werden, sich auf dem Level von Funktionen und Anweisungen durch ein Programm zu bewegen. Das Maß der Dinge müssen die Objekte sein: Sie bilden die Abstraktionsebene, auf der entworfen und implementiert wurde, also sollte ein Debugger auch möglichst lange auf dieser Ebene bleiben. Während der Ausführung sollte der Detaillierungsgrad der Anzeige bei Objekte, ihren Attributen und Methoden liegen. Zeiger auf Objekte sollten als solche dargestellt werden und nicht als Speicheradresse erscheinen. Erst wenn es für den Benutzer wirklich interessant ist, darf das Werkzeug auf das Niveau von Anweisungen, Adressen und Funktionsaufrufen übergehen. Ein Beispiel soll dieses Verständnis von objektorientiertem Debuggen illustrieren: Um die Programmausführung zu unterbrechen, sobald eine bestimmte Methode für ein bestimmtes Objekt erreicht wird, muß in einem konventionellen Debugger ein Breakpoint auf die Codezeile bzw. auf die Einsprungadresse der Methode gesetzt

[2] Uns ist lediglich ein kommerzielles Produkt, *Look!*, bekannt, das sich grundlegend von den herkömmlichen Debuggern unterscheidet. Das Tool stellt den Programmablauf anhand der Objekte graphisch dar und verfolgt damit dieselbe Zielrichtung wie unsere Arbeit.

und danach in einen *bedingten Breakpoint* umgewandelt werden, indem ein Ausdruck eingetippt wird, der angibt, ob die Methode mit einem Verweis auf die Speicheradresse des gewünschten Objekts aufgerufen wurde. Unser Vorschlag ist hingegen, daß auf dem Bildschirm das Sinnbild des Objekts angeklickt und aus einer Liste der dazugehörigen Methoden ausgewählt wird, für welche eine Unterbrechung erfolgen soll.
- *information at your fingertips*: Die Möglichkeiten heutiger Benutzeroberflächen müssen genutzt werden, um möglichst viele relevante Informationen in angemessener Weise zu präsentieren. Wir stellen uns zum Beispiel vor, eine große Anzahl von Objekten durch einfache graphische Symbole anzuzeigen; ein Mausklick auf ein bestimmtes Objekt läßt dann ein Fenster mit zusätzlichen Angaben erscheinen[3].
- *Programme gleichzeitig ausführen und analysieren*: Graphische Benutzeroberflächen ermöglichen, mehrere Prozesse in eigenen Fenstern nebeneinander laufen zu lassen. Die Analyse eines ausführbaren Programmes ist damit in der Art möglich, daß in einem Fenster das Programm ausgeführt und normal benutzt wird. Währenddessen werden daneben die miteinander kooperierenden Objekte *online* angezeigt, ohne daß permanent Eingaben erforderlich sind, um bestimmte Attribute oder Zustände zu überwachen.

Zusammengefaßt erkennen wir den Bedarf nach einem Werkzeug, das die Ausführung eines Programms auf Basis der Objekte visuell umsetzt. Das Ziel besteht darin, Analyse, Entwurf, Implementierung und Wartung ohne größere Modellbrüche miteinander verbinden zu können. Das Werkzeug sollte daher flexibel benutzt und in eine aus anderen Werkzeugen bestehende Umgebung integriert werden können. Ein solches Tool ist dann viel mehr als „nur“ ein besserer OO-Debugger.

2.4. Möglichkeiten eines Visualisierungs-Werkzeuges

Die Unzulänglichkeiten bestehender Debugger und Profiler können als Auslöser genommen werden, um zur Konzeption eines Visualisierungs-Werkzeuges für objektorientierte Programme zu gelangen. Fehlersuche und Optimierung sind damit zwei mögliche Einsatzgebiete. Allgemeiner sehen wir die Fragestellung: „Was passiert eigentlich in meinem Programm während der Ausführung?“ Das Werkzeug, das wir uns vorstellen, dürfte diese Frage zumindest zu einem großen Teil beantworten.
Als konkrete Anwendungen stellen wir uns vor:
- *Verständnis fremder Programme und Bibliotheken*: Die Einarbeitung in unbekannte Programme und in Klassenbibliotheken wird wesentlich erleichtert, indem das Zusammenwirken der Klasseninstanzen während der Ausführung verfolgt wird. Das gilt insbesondere auch für Frameworks, bei denen der Entwickler im Grunde nur an der Verwendung der Funktionalität einer einzelnen Klasse interessiert ist, aber trotzdem das Konzept einer ganzen Bibliothek verstehen muß, und für die Benutzung von Entwurfsmustern [GHJ95].
- *Fehlersuche*: Indem das Entstehen und Verschwinden von Objekten und Beziehungen zwischen Objekten angezeigt werden, können typische Fehlerursachen der OOP schnell aufgespürt werden – Referenzen auf gelöschte Objekte, Aufrufe geerbter statt überschriebener Methoden usw.
- *Optimierung*: Wenn das Werkzeug statistische Daten erfaßt, lassen sich schnell die Abschnitte finden, die einen besonders hohen Ressourcenbedarf haben. Charakteristische Daten für die Rechenzeit sind Aufrufhäufigkeiten, Ausführungszeiten einzelner Methoden usw.
  In der graphischen Darstellung läßt sich einfach erkennen, wo durch viele oder durch speicherintensive Objekte Performanz- oder Speicherprobleme entstehen.
- *Verifikation*: Weil der Ablauf auf Objektebene dargestellt wird und damit der Abstraktionsebene des Entwurfs entspricht, kann ein Programm gut auf seine Spezifikation überprüft werden. Im Idealfall werden in einer Entwicklungsumgebung dieselben graphischen Interaktionskomponenten für Entwurfsdokumente und Ausgabe der Visualisierung benutzt. Das System bietet dann eine optimale Unterstützung für den Vergleich von spezifiziertem und implementiertem Verhalten.
- *Ausbildung*: Nicht nur für das Verständnis bestimmter Programme, sondern auch für das Erlernen objektorientierter Programmierung an sich ist es hilfreich, die Dynamik graphisch zu präsentieren. Objektorientierte Programmierung wird erst durch *dynamische* Objekte sinnvoll, die Programmieranfängern erfahrungsgemäß Probleme bereiten.

[3] Zum Vergleich: Eine vergleichbare Operation wird im Debugger gdb durch Eingabe eines Kommandos wie

```
p *(*tMyBaseClass) 0xd230
```

durchgeführt.

- *Reverse Engineering*: Zur nachträglichen Dokumentation bestehender Programme kann das Werkzeug durch Diagramme beitragen, die das dynamische Verhalten protokollieren.

Je besser das Werkzeug in eine Entwicklungsumgebung integriert ist, desto vielseitiger wird es im Arbeitsalltag verwendet werden können. Gelingt diese Einbindung wirklich, läßt es sich so bequem aufrufen und beiläufig benutzen, wie wir es heute etwa von Klassenbrowsern gewöhnt sind.
Der Praxiswert hängt natürlich davon ab, wie das Verhalten des Programms visualisiert wird. Bevor wir eine konkretere Vision unseres Werkzeugs vorstellen, geht das folgende Kapitel deswegen auf die Möglichkeiten der graphischen Repräsentation ein.

3. Dynamisches Verhalten von Objekten

„Instead of a bit-grinding processor raping and plundering data structures, we have a universe of well-behaved objects that courteously ask each other to carry out their various desires.“

D. Ingalls: Design Principles behind Smalltalk (1981)

Die Funktionalität eines objektorientierten Programms wird niemals durch ein einziges Objekt alleine bestimmt. In einem solchen Fall könnte das Programm kaum aus etwas anderem bestehen als aus konventionellem (im Sinne von nicht-objektorientiertem) Code, der in ein objektorientiertes Konstrukt wie „Klasse“ verpackt ist. Die Vorteile des OO-Modells wären damit in keiner sinnvollen Weise genutzt.

Das Wesen objektorientierter Programme besteht vielmehr im Zusammenspiel von vielen verschiedenen Objekten. Objektorientierter Entwurf bedeutet, dieses Zusammenspiel zu gestalten, indem die Beziehungen zwischen Entitäten der realen Welt erkannt und entsprechend modelliert werden. Um objektorientierte Programme zu verstehen, benötigt man nicht nur das Wissen, welche Klassen benutzt werden, sondern auch, wie diese Klassen und deren Instanzen miteinander in Verbindung stehen.

Die Relationen zwischen Klassen sind dadurch gekennzeichnet, daß sie unabhängig von der Laufzeit der Software festgestellt werden können. Wenn etwa eine Klasse eine andere beerbt, dann gilt diese Beziehung beim Entwurf genauso wie später zu einem beliebigen Zeitpunkt während der Programmausführung. Klassen verkörpern auf diese Art statische Strukturen.

Im Gegensatz dazu sind die Beziehungen zwischen Objekten dynamisch: Objekte entstehen zur Laufzeit, kommunizieren und kooperieren mit anderen Objekten und werden wieder gelöscht. Das dynamische Verhalten von Programmen wird somit weniger von Klassen als von Instanzen bestimmt. Kein objektorientiertes Entwurfsverfahren kann ohne Methoden zur Modellierung der Dynamik auskommen, die den Übergang von der Klassen- auf die Objektebene vollziehen.

Für das Werkzeug, das wir entwickeln, müssen wir sowohl für Klassen als auch für Objekte die möglichen Beziehungen kennen. Die beiden folgenden Abschnitte werden diese Relationen behandeln. Um die Inter-Objekt-Beziehungen graphisch aufzubereiten, werden dafür geeignete Notationen benötigt, womit sich der letzte Teil dieses Kapitels beschäftigt.

3.1. Beziehungen zwischen Klassen

Objektorientierte Systeme unterscheiden sich von objekt*basierten* durch die Möglichkeit, in einer Klasse Eigenschaften anderer Klassen zu erben. Vererbung ist damit die erste wesentliche Beziehung, in der Klassen zueinander stehen können. Über weitere Beziehungstypen gibt es unterschiedliche Ansichten, die zum Teil darin begründet sind, daß sich bestimmte Strukturen nicht eindeutig gegeneinander abgrenzen lassen. Zum anderen ist maßgebend, ob die Klassen von einem realitäts- oder einem modellnahen Standpunkt aus betrachtet werden. Rumbaugh orientiert sich eher an der zu modellierenden Wirklichkeit und stellt darin die Beziehungstypen Vererbung, Assoziation und Aggregation fest [RBP91]. Wir übernehmen die Einteilung von Booch, die etwas weiter differenziert ist [Boo94]. Hinsichtlich der Abstraktionsebenen von Modellierung und Implementierung steht diese Einteilung der Implementierung näher als die von Rumbaugh und eignet sich dadurch für die Analyse bestehender OO-Programme besonders.

Die Beziehungstypen nach Booch sind:

- *Vererbung*: Eine Klasse erbt die Eigenschaften einer oder mehrerer anderer Klassen. Die Klassen treten als Unter- bzw. Oberklassen in der Vererbungshierarchie auf. Eine Unterklasse steht zu ihren Oberklassen in einer „is-a“-Beziehung. Oberklassen sind eine *Generalisierung* ihrer Unterklassen, in der umgekehrten Richtung wird von *Spezialisierung* gesprochen.
- *Assoziation*: Als Assoziation bezeichnet Booch eine semantische Abhängigkeit zwischen Klassen, die während der Modellierung festgestellt wird. Im Entwurfsprozeß kann sie zur Vererbungs-, Aggregations- oder Verwendungsbeziehung verfeinert werden. Für eine Assoziation wird die Kardinalität (z.B. 1:n-Relation) bestimmt, aber noch keine Richtung.
- *Aggregation*: Eine Aggregation liegt vor, wenn zwischen Objekten einer Klasse eine *Whole-Part*-Beziehung herrscht. In der Realität ist dies der Fall, wenn ein Objekt als isolierbarer Bestandteil eines andersartigen Objekts erscheint. Enthaltene Klassen sind in der Regel durch die Aggregation nach außen gekapselt.
- *Verwendung (using)*: Durch eine Verwendungsbeziehung treten zwei Klassen in eine *Client-Supplier*-Relation, d.h. eine Klasse stellt einer anderen bestimmte Dienste zur Verfügung (vgl. auch [Mey88]). Das Protokoll für diese Dienstleistungen gibt der nach außen sichtbare Teil der

Schnittstelle der Supplier-Klasse vor. Nach dem Geheimnis-Prinzip sollten in der Schnittstelle nur Methoden, aber keine Attribute offengelegt werden.
Im Einzelfall gibt es keine eindeutige Trennung zwischen Aggregation und Verwendung.

- *Instanziierung*: Eine Instanziierung von Klassen ist durch Generizität möglich. Von einer allgemeinen generischen Klasse wird eine konkrete Klasse durch Instanziierung abgeleitet.
- *Meta-Klasse*: In Smalltalk oder CLOS können Klassen als Instanzen von Meta-Klassen gebildet werden. Da Meta-Klassen in C++ nicht unterstützt werden können, ist diese Beziehung für uns nicht von Bedeutung. Die Auswirkungen, die aus Meta-Klassen folgen, werden wir im weiteren Text nicht berücksichtigen.

3.2. Beziehungen zwischen Objekten

Aus den Beziehungen zwischen Klassen resultieren Beziehungen zwischen den Instanzen. Es ist offensichtlich, daß zwischen zwei Objekten nur eine Verbindung bestehen kann, wenn diese bereits in der statischen Beschreibung der zugehörigen Klassen bzw. deren Oberklassen implementiert wurde. Während Klassen zeitinvariant sind (zumindest wenn keine Metaklassen existieren), hat jedes Objekt eine bestimmte Lebenszeit. Ebenso sind auch Relationen zu anderen Objekten im System zeitabhängig. Beziehungen zwischen Objekten gehen also aus Beziehungen zwischen Klassen hervor, indem die Rollen der Beteiligten durch konkrete Instanzen besetzt werden und die Hinzunahme der Zeit als zusätzliche Dimension erfolgt.
Für die Beziehungstypen der Klassen bedeutet das im einzelnen:

- Vererbung wird in den gebräuchlichen objektorientierten Systemen zwischen Klassen, nicht aber zwischen Objekten realisiert. Eine Umsetzung der Vererbungsbeziehung auf Objektebene gibt es daher nicht.
- Auch für die Assoziation gemäß der vorausgegangenen Definition findet man keine Entsprechung zwischen Objekten, da assoziierte Relationen bis zur Implementierung zu anderen Beziehungen weiterentwickelt werden.
- Zur Instanziierung einer konkreten Klasse aus einer generischen kann es ebenfalls keine analoge Objektbeziehung geben: Die generische Klasse ist immer abstrakt, d.h. es kann niemals ein Objekt von ihr instanziiert werden. Unter Vernachlässigung der Meta-Klassen bleiben also Verwendung und Aggregation, für die sich die Frage nach resultierenden Objektbeziehungen stellt.
- Der Verwendungsbeziehung entspricht auf Objektebene die *Verknüpfung (link)*: Rumbaugh definiert einen link als „eine Instanz einer Assoziation; physikalische oder konzeptuelle Verbindung zwischen Objekten“ [RBP91]. Eine Verknüpfung ist unidirektional und ermöglicht es einem Objekt, eine Nachricht an das verknüpfte Objekt zu senden. Ein Datenaustausch ist über eine Verknüpfung in beiden Richtungen möglich, so daß ein Objekt der Client-Klasse durch Nachrichten an Objekte der Server-Klasse deren Methoden aufrufen und Ergebnisse zurückerhalten kann.
  Booch beschreibt Verknüpfungen als transitiv. Den an einer Verknüpfung beteiligten Objekten weist er drei mögliche Rollen zu: *actor*, *server* und *agent*. Der *actor* kann Nachrichten an den *server* senden, aber nicht umgekehrt. Wenn zwei Objekte nur indirekt miteinander verknüpft sind, nehmen die dazwischenliegenden Objekte die Funktion von *agents* an.
- Zur *Aggregation*: Booch benutzt den Begriff Aggregation nicht nur als Typ einer Klassenbeziehung, sondern auch für die daraus entstehende Whole-Part-Beziehung zwischen zwei Objekten. Für die Implementierung ist der Begriff des *physical containment* von besonderem Interesse. Er impliziert, daß die Lebensdauer eines enthaltenen Objekts unmittelbar mit der des beinhaltenden Objekts verknüpft ist. Booch unterscheidet *containment by value* und *containment by reference* in Abhängigkeit davon, ob ein Objekt als Variable oder über einen Zeiger enthalten ist [Boo94].
  Die Aggregation bildet einen Spezialfall der Verknüpfung; jedes Aggregat besitzt links auf die enthaltenen Objekte.

In der Darstellung der Objektbeziehungen sehen wir die zentrale Aufgabe des Software-Werkzeugs, an dessen Entwicklung wir arbeiten. Wir benötigen daher eine Klassifizierung von Objektbeziehungen, in die die Objektrelationen eines Programms eingeordnet werden können. Die Relationen Verknüpfung und Aggregation erscheinen uns noch nicht ausreichend, so daß wir eine eigene Klassifizierung für Objektbeziehungen vorgesehen haben. Sie kennt vier verschiedene Typen:

- *referenziert*-Beziehung: Ein Objekt kennt ein anderes Objekt über eine Zeiger- oder Referenz-Variable.

- *hat*-Beziehung: Ein Objekt enthält ein anderes Objekt im Sinne der Aggregation. Nach Boochs Terminologie liegt ein *containment by value* vor.
- *benutzt*-Beziehung: Ein Objekt nimmt Dienste eines anderen Objektes in Anspruch, d.h. es greift auf dessen Attribute zu (was vermieden werden sollte) oder ruft dessen Methoden auf. Es kann, aber muß nicht notwendigerweise eine referenziert- oder hat-Beziehung zu Grunde liegen. Dies gilt zum Beispiel, wenn eine Nachricht an ein Objekt gesendet wird, zu dem nur eine indirekte Verknüpfung besteht.
- *erzeugt*-Beziehung: Durch die erzeugt-Beziehung wird angegeben, von welchem Objekt die Erzeugung eines neuen Objekts veranlaßt wurde. Jedem Objekt kann durch diese Relation genau ein Erzeuger zugeordnet werden[4]. Ebenso wie die benutzt-Beziehung liegt bei der Erzeugung normalerweise eine referenziert-Beziehung (für dynamische Objekte) oder eine hat-Beziehung (für statische Objekte) vor. Im Gegensatz zu den anderen Typen bleibt die erzeugt-Beziehung bis zum Ende der Lebensdauer des erzeugten Objekts vorhanden, auch wenn es eventuell von keinem anderen Objekt mehr referenziert wird.

  Die erzeugt-Beziehung zeichnet sich weiterhin dadurch aus, daß dem Erzeuger zum Zeitpunkt der Erzeugung die Identität des neuen Objekt noch gar nicht bekannt ist: Die Nachricht, die das Entstehen des Objekt auslösen soll, muß an eine andere, bereits bestehende Instanz gesendet werden. Erst hinterher kann dem Erzeuger das Objekt bekannt gemacht werden.

Man kann überlegen, ob es analog zur erzeugt- auch eine *löscht*-Beziehung geben sollte. Nach unserer Vorstellung ist dies aber nur ein Spezialfall der benutzt-Relation. Man bedenke auch, daß eine löscht-Beziehung nur zu einem einzigen Zeitpunkt, nicht aber über eine Zeitspanne bestehen kann: Die Relation kommt in dem Moment zustande, wo eines der beteiligten Elemente aufhört zu existieren (nämlich das gelöschte Objekt), so daß die Relation unmittelbar hinfällig ist. Anders sieht es jedoch aus, wenn wir in einer Modellierung des Objektverhaltens Objekte auch noch nach ihrem Löschen erfassen. Während im Programm ein Objekt also nicht mehr existiert, können wir das Modell des Objekts beibehalten und dieses Modell mit dem Zustand „gelöscht" kennzeichnen. Bei einer solchen Modellierung erscheint es sinnvoll, eine löscht-Beziehung einzuführen.

3.3. Überlegungen zu graphischen Notationen

Es gibt wohl keinen Autor, der über objektorientierten Entwurf schreibt und dabei nicht auf den Wert von Diagrammen hinweist. Für die Beziehungen zwischen Klassen sind entsprechend viele graphische Notationen vorgeschlagen worden. Da es ausreichend viele Arbeiten auf diesem Gebiet gibt, werden wir uns nicht weiter mit möglichen Darstellungsalternativen beschäftigen, sondern die Diagramme der Object Modeling Technique (OMT) nach [RBP91] benutzen.
Notationen, die Objektbeziehungen veranschaulichen, findet man deutlich seltener. Wegen der Dynamik dieser Beziehungen ist es natürlich problematisch, Diagramme zu konstruieren, die - auf Papier gezeichnet - mehr darstellen als eine Momentaufnahme des Systems zu einem bestimmten Zeitpunkt. Wenn wir solche Diagramme aber auf dem Bildschirm animieren, erhalten wir Visualisierungen der Programmausführung, die Dynamik problemlos ausdrücken und vom Menschen intuitiv gut erfaßt werden können.

3.3.1. Objektdiagramme in OMT und bei Booch

Für das Objektmodell der OMT sieht Rumbaugh vor, Objekte als Rechtecke mit abgerundeten Ecken zu zeichnen [RBP91]. Dieses Sinnbild korrespondiert mit den rechteckigen Symbolen für Klassen. Während in den Bildern der Klassen die Attributnamen aufgelistet werden können, erscheinen in den Objektsymbolen an deren Stelle die konkreten Werte der Instanz. Als einzige Relation wird die Instanziierung als gepunkteter Pfeil zwischen Objekt und Klasse notiert. Als Teil des Objektmodells fällt diese Darstellung in die statische Modellierung der OMT. Die dynamischen Aspekte werden durch das dynamische und das funktionale Modell ausgedrückt. Deren Notation ist für unsere Anwendung aber ungeeignet, weil sie eher mikroskopisch die objektinternen Zustandsübergänge und Kontrollflüsse, nicht aber die makroskopischen Beziehungen zwischen Objekten darstellen kann.
Im Gegensatz zu Rumbaugh führt Booch neben dem Klassendiagramm auch ein eigenes Objektdiagramm ein [Boo94]. Knoten des Diagramms sind die Booch-typischen Wolken als Sinnbilder der Ob-

[4] In C++ kann es auch nicht-objektgebundene Funktionen geben, z.B. die Funktion *main* oder Klassenmethoden (vgl. 6.1.). Es stellt sich die Frage, ob man Objekten, die in solchen Funktionen erzeugt wurden, einen Erzeuger zuordnen kann. Eine Möglichkeit besteht in der Einführung „virtueller" globaler oder klassenabhängiger Objekte, die als Erzeuger in die Relationen aufgenommen werden. Siehe dazu Kapitel 7.

jekte, die Kanten stehen für die Verknüpfungen. Die Diagramme erhalten eine hohe Ausdrucksfähigkeit, indem die Wolken und Linien durch graphische Zusätze variiert werden. Damit lassen sich z.B. Reihenfolgen von Nachrichten, synchrone und asynchrone Ereignisse und die Sichtbarkeit der Objekte untereinander beschreiben. Insgesamt überzeugt die umfassende Notation der Objektdiagramme. Es dürfte jedoch schwierig sein, die für die Entwurfsphase konzipierte Darstellung für ein analysierendes Programm zu benutzen. Wie aussagekräftig ein Graph ist, hängt schließlich nicht nur von den zur Verfügung gestellten Symbolen ab, sondern ganz wesentlich von deren Verwendung und der Fähigkeit, Verfeinerungen und Abstraktionen sinnvoll und flexibel einzusetzen. Ein Computerprogramm, das Graphen und Animationen generiert, wird diese Fähigkeit kaum besitzen können. Vor diesem Hintergrund erscheint selbst das Objektdiagramm von Booch als schon zu komplex.

3.3.2. Objektdiagramme als Basis für ein Software-Werkzeug

Eine Notation, die unseren Anforderungen genügt, muß
- Symbole für Objekte enthalten,
- die vier definierten Objektrelationen darstellen können und
- so übersichtlich wie möglich sein.

Die Übersichtlichkeit ist vielleicht das schwerwiegendste Argument: Ein Graph, der in einem Bild alle möglichen relevanten Informationen präsentieren will, wird schon bei einer geringen Anzahl von Objekten kaum noch überschaubar und damit unbrauchbar sein. Also gilt zunächst „weniger ist mehr“, d.h. die Beschränkung auf die Darstellung der wesentlichen Inhalte. Das gilt um so mehr, weil wir als Darstellungsmedium nicht Papier, sondern einen Bildschirm verwenden werden, dessen Abmessungen fest begrenzt sind. Natürlich kann das Platzangebot durch Scrolling beliebig vergrößert werden. Bei einem animierten Graph, der sich permanent ändert, führt das aber dazu, daß entweder Änderungen außerhalb des sichtbaren Ausschnitts geschehen oder daß der sichtbare Ausschnitt mit jeder Änderung verschoben werden muß. Beide Alternativen sind nicht ideal, und das Problem kann nur dadurch entschärft werden, daß der Graph möglichst kompakt gehalten wird.
In einer Abwägung zwischen Übersichtlichkeit einerseits und Informationsgehalt andererseits halten wir folgende Forderungen für sinnvoll:
- Die Menge der angezeigten Objekte muß durch den Benutzer bestimmt werden können. Das kann zum Beispiel durch Beschränkung auf bestimmte Klassen oder auf bestimmte Zeitpunkte geschehen (etwa: alle Objekte, die erzeugt werden, nachdem der Benutzer eine bestimmte Funktion des untersuchten Programms gestartet hat).
- Statt eines einzigen Diagramms für die vier Objektrelationen können mehrere einzelne Graphen nebeneinander benutzt werden. Dadurch lassen sich die einzelnen Diagramme besser überblicken. Wir sehen keine grundsätzlichen Nachteile in diesem Ansatz.
- Wenige, wesentliche Informationen erscheinen direkt im Diagramm, zusätzliche Informationen sollten auf Anfrage einfach erhältlich sein. Zum Beispiel: Zu jedem Objekt wird immer ein Name angezeigt. Die Attribute erscheinen aber erst nach einem Mausklick auf das entsprechende Objekt.

Diese Überlegungen lassen noch in vieler Hinsicht offen, wie ein konkretes Werkzeug aussehen könnte. Sie stecken erst den Rahmen ab, den wir im nächsten Kapitel mit unserer Vision eines Werkzeuges füllen wollen.
Der zweite der drei angesprochenen Punkte wirft die grundlegende Frage auf, welche Darstellungen das Werkzeug bieten soll. Bevor wir auf die drei Diagramme eingehen, die wir uns als Ausgangsbasis vorstellen, müssen wir ein Konzept zur Benennung der Objekte entwickeln, um Objekte in den Diagrammen mit einem Namen anzeigen zu können.
Für alle Objekte, die als statische Variable deklariert sind[5], scheint es sinnvoll, den Variablennamen zu übernehmen. Dieser Name kann immer eindeutig angegeben werden. Für dynamisch deklarierte Objekte könnte man den Namen der Zeigervariable benutzen, die auf das Objekt verweist. Hier gibt es aber keine Eindeutigkeit mehr; schließlich können beliebig viele Referenzen auf ein Objekt existieren. Eine Möglichkeit besteht nun darin, den Namen derjenigen Variable zu nehmen, unter dem das Objekt zuerst referenziert (bzw. erzeugt) wurde. Eine solche Variable muß es jedoch nicht immer geben, denn die verbreiteten Programmiersprachen erlauben die Erzeugung von Objekten, ohne daß eine benannte Referenz darauf verweist.
Als nicht unbedingt schöne, aber in jedem Fall eindeutige Lösung bietet es sich an, die Objekte durch den Namen ihrer Klasse und eine fortlaufende Nummer zu identifizieren. Daß dadurch sofort auf die Klasse eines angezeigten Objekts geschlossen werden kann, wird in vielen Fällen vorteilhafter sein

[5] insbesondere Objekte, die in einer hat-Beziehung enthalten sind

als der Verweis auf Variablennamen. Nach diesem Schema bezeichnet beispielsweise *tAdresse#4* das vierte Objekt, das von der Klasse *tAdresse* instanziiert wurde.

3.4. Basis der Visualisierung: Drei Diagrammtypen

Anstelle eines einzigen, allumfassenden Objektdiagramms halten wir es für sinnvoller, das dynamische Verhalten eines Programms in drei Graphen auszudrücken, die einander ergänzen. Damit legen wir das Fundament für das Werkzeug, das entwickelt werden soll.
Dia Namen, die unsere drei Typen von Diagrammen tragen, sind:

- Das *Konstruktions-Diagramm*[6], das die erzeugt-Beziehungen darstellt,
- das *Referenz-Diagramm* für referenziert- und hat-Beziehungen und
- das *Interaktions-Diagramm*, das benutzt- und erzeugt-Beziehungen enthält.

Damit erhalten wir eine Minimalkonfiguration von Sichtweisen – die Möglichkeiten der Visualisierung sind durch die drei Graphen bei weitem noch nicht ausgeschöpft.

3.4.1. Das Konstruktions-Diagramm

Das Konstruktions-Diagramm ist ein gerichteter Graph. Die Knoten symbolisieren Objekte als abgerundete Rechtecke; die Kanten besagen, daß ein Objekt die Erzeugung eines anderen ausgelöst hat. Die Kanten werden mit dem Namen der Methode versehen, die den Konstruktoraufruf enthielt.
In Abbildung 3.1 gibt es also ein Objekt einer Klasse *tAdreßbuch*, das in seinem gleichnamigen Kostruktor ein Objekt der Templateklasse *tList<tAdresse>* erzeugt. Im weiteren Verlauf werden der Liste drei Objekte hinzugefügt, was durch die Methode *tList<tAdresse>::Add* geschieht und jedesmal bewirkt, daß ein neuer Listenknoten *tNode<tAdresse>* entsteht.

Abbildung 3.1

Das Diagramm enthält zu einem bestimmten Zeitpunkt alle existierenden Objekte[7]. Im Verlauf der Animation kommen fortwährend neue Knoten hinzu, und bestehende verschwinden. Durch das Diagramm erkennt man einfach, welche Objekte während eines Programmlaufs entstehen, in welcher Reihenfolge dies geschieht und, anhand der Kantenbeschriftungen, aus welchem Kontext heraus Objekte erzeugt werden.
Für die folgenden Aufgaben und Tätigkeiten erscheint uns das Konstruktions-Diagramm als besonders geeignet:

- *Einarbeitung in Klassenbibliotheken*: Die Abhängigkeiten zwischen Klassen werden besser verdeutlicht, als es durch statische Hierarchiegraphen möglich ist. Man kann schnell erkennen, von welchen Klassen überhaupt Objekte instanziiert werden.
- *Erkennen „persistenter" Objekte*: In der Animation wird deutlich, welche Objekte ihre Erzeuger überleben. Dieses Verhalten ist oft erwünscht und kann durch die Visualisierung gut vermittelt werden. Es kann aber auch auf Fehler deutlich machen, weil das überlebende Objekt eventuell von seinem Erzeuger gelöscht werden sollte. Am Ende eines Programmlaufs ist schnell zu erkennen, ob einzelne Objekte übrig bleiben, die es eigentlich nicht mehr geben soll und die dadurch nur unnötigen Speicherplatz beanspruchen[8].

Das Diagramm bietet viel Raum für Weiterentwicklungen. Auf dem Bildschirm kann zusätzliche Funktionalität hinzugefügt werden, indem ein Mausklick auf Knoten oder Kanten unterschiedlichste Informationen zu Objekten und den erzeugt-Relationen anzeigt. Die Knoten können durch graphische

---

[6] zu Konstruktions- und Referenz-Diagramm vgl. [Wes93]
[7] Wie oben erwähnt wurde, muß der Anwender die Möglichkeit haben, die Menge der angezeigten Objekte einzuschränken. Dabei entsteht das Problem, daß dem Graphen Knoten fehlen werden, um Relationen neu erzeugter Objekte einzutragen. Der Graph zerfällt in viele kleine Zusammenhangskomponenten und verliert einen großen Teil der eigentlichen Information.
[8] Dieser Punkt betrifft gezielt C++, wo keine garbage collection vorhanden ist.

Attribute ergänzt werden: Das jeweils aktive Objekt kann besonders hervorgehoben werden. Für ein gelöschtes Objekt wäre es z.B. sinnvoll, den zugehörigen Knoten nicht einfach verschwinden zu lassen, sondern ihn vorher mit einer bestimmten Farbe zu versehen. Statische und dynamische Objekte könnten durch Farben oder Muster unterschieden werden usw.

3.4.2. Das Referenz-Diagramm

Das Referenz-Diagramm ist ebenfalls ein gerichteter Graph mit denselben Knoten wie im Konstruktions-Diagramm. Es treten zwei verschiedene Arten von Kanten auf: Durchgezogene Pfeile stehen für referenziert-Beziehungen und verweisen von einem Objekt auf andere, auf die es Referenzen enthält. Der Variablenname, unter dem diese Referenz besteht, erscheint als Kantenbeschriftung. Analog werden hat-Beziehungen durch gestrichelte Pfeile angezeigt (Abbildung 3.2).
Naturgemäß besteht hier nicht mehr die Übersichtlichkeit des Konstruktions-Diagramms, weil jedes Objekt beliebig oft referenziert werden kann. Ein Knoten könnte damit eine große Zahl an eingehenden Kanten halten. Wir schlagen vor, einen anderen Weg zu gehen: Jedes Objekt, das mehrfach referenziert wird, erscheint genauso oft als Knoten. Die Knoten können so angeordnet werden, daß das Diagramm übersichtlich und lesbar bleibt. Durch eine farbige Kennzeichnung wird für solche Objekte angezeigt, daß sie mehrfach im Graph auftreten (in Abbildung 3.2 für *tAdresse#17*). Eine Menüoption könnte für ein ausgewähltes Objekt alle zugehörigen Knoten im Diagramm hervorheben.
Bei referenziert-Relationen kann es vorkommen, daß das referenzierte Objekt gelöscht wird, die Referenz aber bestehen bleibt – ein typischer Fehler in objektorientierten Programmen. Das Diagramm muß deshalb auch solche unvollständigen Beziehungen, bei denen einer der Partner nicht mehr existiert, anzeigen. Wir schlagen vor, daß Objektsymbol in diesem Fall durch ein abgerundetes Rechteck mit einem Fragezeichen zu ersetzen. Referenzen, die ordentlich auf *null* gesetzt sind, sollten der Übersichtlichkeit halber nicht im Graph auftreten.

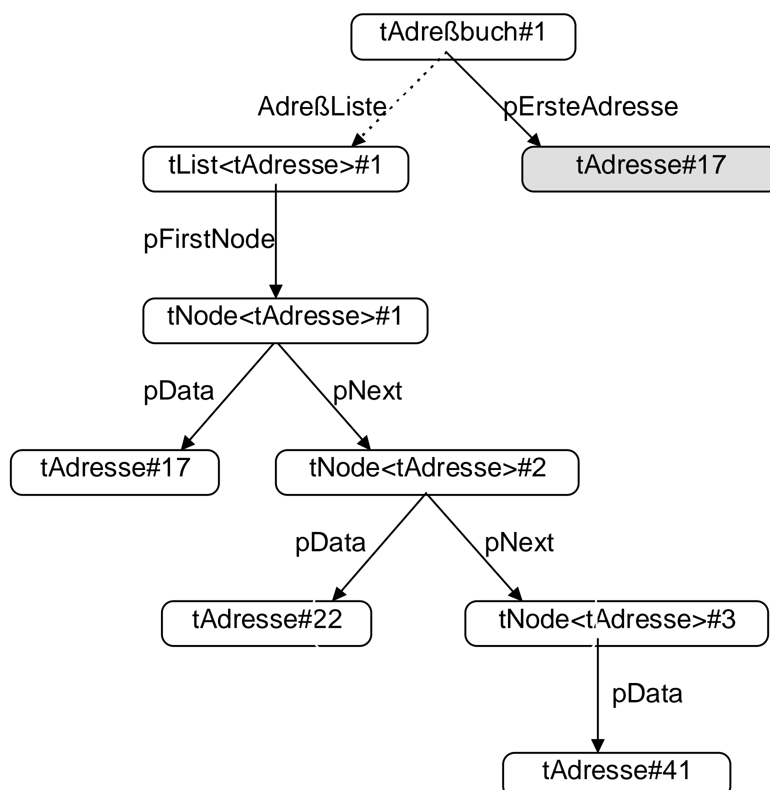


Abbildung 3.2

Das Referenz-Diagramm verändert sich genau wie das Konstruktions-Diagramm. In beiden sind zu jeder Zeit dieselben Objekte eingetragen. Zusammen ergeben sie einen umfassenden Einblick in ein untersuchtes Programm. Für sich alleine eignet sich das Referenz-Diagramm besonders gut zur Fehlersuche, um falsche oder nicht gelöschte Referenzen ausfindig zu machen.
Um die beiden vorgestellten Graphen optisch voneinander abzuheben, lassen wir das Konstruktions-Diagramm von links nach rechts, das Referenz-Diagramm von oben nach unten wachsen. Für Ergän-

zungen und Erweiterungen gelten hier dieselben Anregungen, die oben zu den Konstruktions-Diagrammen gegeben wurden.

3.4.3. Das Interaktions-Diagramm

Interaktions-Diagramme verwenden wir in der Form, wie sie für die Objectory-Methode eingeführt wurden [JCJ92]. Im Gegensatz zu den beiden zuvor besprochenen Diagrammen stellt diese Notation nicht Momentaufnahmen, sondern den zeitlichen Verlauf der Programmausführung dar. Die Zeitachse verläuft von oben nach unten. Objekte erscheinen als vertikale Linien. Die Lebensdauer eines Objekts läßt sich am Start- und Endpunkt der Linie ablesen. Außerhalb dieses Zeitraums wird die Linie nach oben und unten gestrichelt verlängert (vgl. Abbildung 3.3).
Wenn ein Objekt durch eine Nachricht aktiviert wird, erscheint die Objektlinie als Balken. Nachrichten werden durch horizontal verlaufende Pfeile symbolisiert, die mit dem Namen und den Argumenten der aufgerufenen Methode versehen werden.
Das Interaktions-Diagramm umfaßt erzeugt- und benutzt-Beziehungen. Weil die zeitliche Abfolge mehrerer Ereignisse in *einem* Graphen festgehalten wird, eignet sich das Diagramm für die Ausgabe auf dem Bildschirm genauso wie auf einem Drucker. Die Verwendung von Interaktions-Diagrammen in der Entwurfsphase ist nicht nur in Objectory vorgesehen, sondern wurde mittlerweile auch von anderen Autoren aufgegriffen [Boo94], [GHJ95]. Ein Werkzeug, das aus implementierten Programmen Interaktions-Diagramme erzeugen kann, ermöglicht also den direkten Vergleich mit den Entwurfsdokumenten. Eine weitere Anwendung besteht in der nachträglichen Dokumentation von Software.

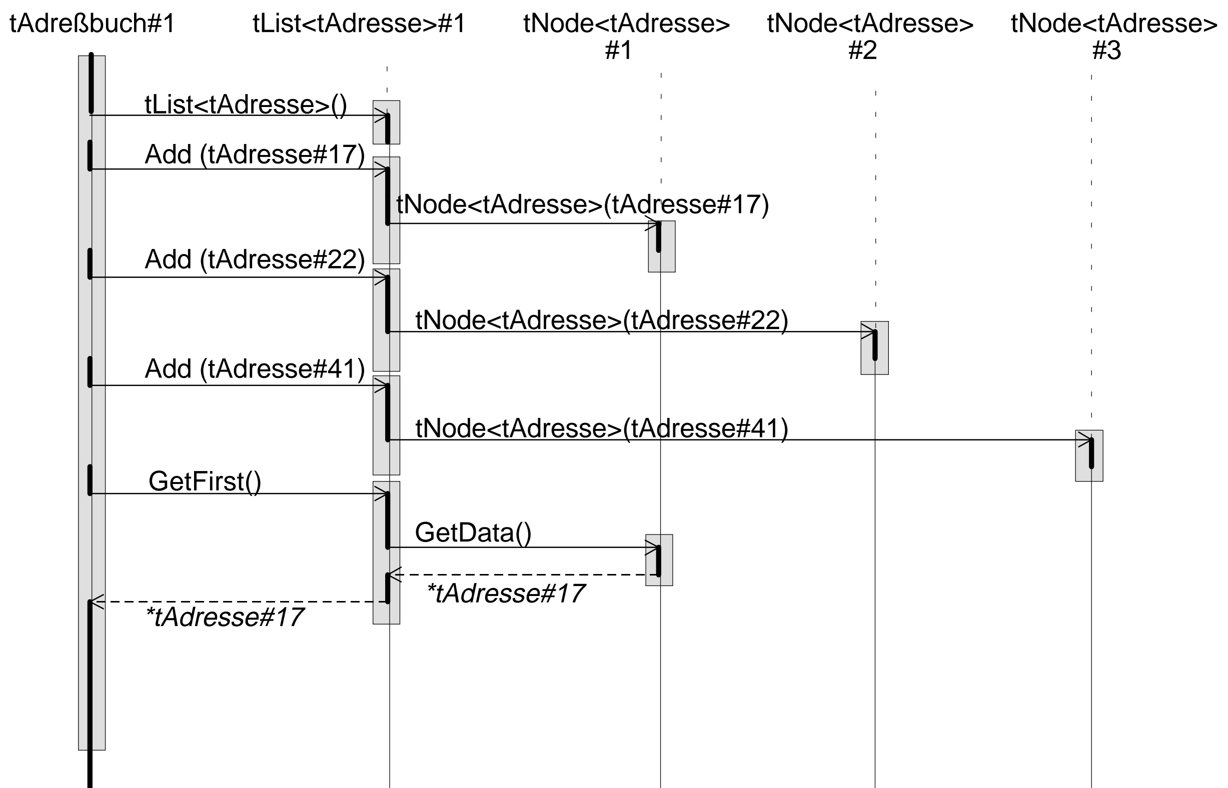


Abbildung 3.3

4. Möglichkeiten einer integrierten Entwicklungsumgebung

Dieser Abschnitt beschäftigt sich mit der optimalen Integration des hier behandelten Werkzeugs in eine Entwicklungsumgebung. Dazu ist es notwendig, kurz die allgemeinen Ansprüche an eine Entwicklungsumgebung aufzuzählen. Darüberhinaus wird auf diese Punkte jedoch nicht tiefer eingegangen werden. Insbesondere werden die Probleme, die sich bei einer möglichen Realisierung ergeben, nicht näher diskutiert.
Dagegen soll genauer dargestellt werden, wie dieses Werkzeug die bereits vorhandenen Möglichkeiten erweitern kann.

4.1. Bestehende Entwicklungsumgebungen

Ein Werkzeug wie dieses ist als Unterstützung für den Entwickler gedacht.
Es fügt sich daher möglichst so nahtlos in eine Entwicklungsumgebung ein, daß es zum integralen Bestandteil der Umgebung wird.
Information über das Programm wird wo immer möglich automatisch generiert. Vom Entwickler erzeugte Dokumentation wird dazu verwaltet. In den Analyse- und Design-Phasen sind dies weitgehend frei formulierte Texte. Während der weiteren Entwicklung kommen Beschreibungen der Architektur und im Speziellen Klassen- und Methoden-Beschreibungen dazu.
Eine Versionsverwaltung arbeitet übergreifend auf der Dokumentation und dem Source-Code.
Damit arbeitet möglichst transparent ein Locking-Mechanismus zusammen, der die gleichzeitige Arbeit mehrerer Entwickler ermöglicht und koordiniert. Im Idealfall wird die Zusammenarbeit durch weitergehende Werkzeuge zur Projektplanung unterstützt.
Eine Reihe von Darstellungs- und Editier-Möglichkeiten auf verschiedenen Abstraktions-Ebenen wird angeboten. Hier ergeben sich vielfältige Möglichkeiten durch den Übergang von Dokumentation der Analyse-Phase über Beschreibungen der Design-Phase und Beschreibungen von Klassen und Methoden, hin zu Klassen- und Methoden-Deklarationen bis zu ihrer Implementation. Unterstützt wird das Navigieren über die Dokumentation und durch Symbol-Browser, Cross-Referencer und Darstellung der Klassen-Hierarchie.
Die Darstellung wird auf sämtlichen Ebenen simultan aktualisiert.

4.2. Visualisierung der Programmausführung

Über ein Werkzeug zur Auswertung dynamischer Vorgänge während der Programmausführung können bei vielen der bereits existierenden Darstellungen zusätzlich Zusammenhänge aufgezeigt werden. Dadurch wird die Aussagekraft und der Wert dieser Diagramme teilweise erheblich gesteigert.

Die Sicht auf den Source-Code wird während der Ausführung aktualisiert. Bereits durchlaufene Code-Teile werden markiert. Der Text einer Klassen-Deklaration wird um die aktuellen Werte der Attribute eines ausgewählten Objektes ergänzt.

In der Darstellung der Klassenhierarchie wird die Klasse des momentan aktiven Objektes markiert. Zu jeder Klasse kann eine Liste der erzeugten Objekte angezeigt werden. Klassen können hier zur Beobachtung aktiviert oder deaktiviert werden.

Statistische Informationen über Klassen, wie z.B. gegenseitige Aufrufhäufigkeiten der zugehörigen Objekte als grobes Maß für den Grad der Kopplung dieser Klassen, Anzahl erzeugter Objekte, Zeitbedarf der Methoden als Anteil an der Gesamtlaufzeit, geben auf oberster Ebene einen Überblick über das Gesamtsystem. Diese Daten können zum großen Teil an bestehende Diagramme wie gebunden werden.

Durch die Beobachtung der Ausführung des Programms ergeben sich jedoch auf verschiedenen Ebenen völlig neue Darstellungen, die sinnvoll zu bestehenden Diagrammen in Beziehung gesetzt werden können:

So kann der Grad der Kopplung von Klassen auch in einem neuen Diagramm dargestellt werden, in dem die Entfernung zweier Knoten ein Maß für Aufrufhäufigkeiten ist. Dieses Diagramm ist in dem Sinne animiert, als sich dieses Maß über die Dauer der Programmausführung ändert [PKV94].

Weitere neue Möglichkeiten machen den wesentlichen Wert dieses Werkzeugs aus: Als wichtigste Erweiterung sind die in 3.4. genannten Diagramme von Objekten und Objekt-Relationen zu nennen. Diese geben einen etwas detaillierteren Einblick als über Klassen kumulierte Informationen.

Auf der untersten Ebene wird die Funktionalität eines herkömmlichen Debuggers auf objektorientierte Konzepte ausgebaut. Die Ausführung von Methoden wird an ein bestimmtes Objekt gebunden dargestellt. Breakpoints können auf ein oder mehrere bestimmte Objekte beschränkt werden. Der Aufrufstack wird mit Objekten und Methoden angezeigt. Allgemein kann ein Objekt als eine Einheit angesprochen werden.

Diese neuen Darstellungen, die durch das Werkzeug möglich werden, reichen somit von der Darstellung eines einzelnen Objektes mit seinen Attributen, über das Generieren von Interaktions-Diagrammen bis hin zu statistischen Auswertungen auf Ebene von Klassen.
Dabei ergeben sich die wertvollsten Möglichkeiten durch die gleichzeitige Darstellung verschiedenster Anzeigen. So kann z.B. eine während des Designs entstandene Vision als Text angezeigt, am ablaufenden Programm getestet, dabei ein Interaktions-Diagramm erzeugt und gleichzeitig der durchlaufene Code markiert werden.

Wo immer möglich sind die verschiedenen Sichten durch Querverweise miteinander verknüpft. Folgende Beziehungen sollten an der Oberfläche durch an die entsprechenden Symbole gebundene Verweise (über PopUp-Menüs, Buttons u.ä.) ausgenutzt werden:

- eine Klasse mit den von ihr instanziierten Objekten und umgekehrt ein Objekt mit der entsprechenden Klasse
- eine Klasse zu ihren Ober- und Unterklassen
- ein Klasse zu einer Sicht auf die Klassendeklaration im Source-Code
- ein Objekt zu einer detaillierten Sicht des Zustand, d.h. einer adäquaten Darstellung aller Attribute
- ein Objekt mit der Anweisung im Source-Code, an der es erzeugt wurde
- ein Objekt zu allen Variablen in denen es referenziert wird
- ein Objekt mit allen referenzierten Objekten
- der Aufrufstack zu Objekten und Methoden-Definitionen
- der Aufruf-Stack zu einem Interaktions-Diagramm

Zusätzlich sind natürlich die bestehenden Möglichkeiten eines Cross-Referenz-Browsers mit der Ausnutzung statischer Beziehungen zwischen Bezeichnern weiterhin vorhanden.

Es besteht die Möglichkeit, zur besseren Orientierung nach Anwählen eines Symbols den Fokus aller Darstellungen, in denen dieses Symbol eine Enstprechung hat, zu setzen.

4.3. Schwierigkeiten der Integration

Ein Problem ergibt sich durch die Forderung nach gleichzeitiger Aktualisierung aller Darstellungen:
Ein Teil der Darstellungen soll bereits vor der Übersetzung generiert werden können. Dagegen ist ein großer Teil der neuen Darstellungen von der Übersetzung und meist auch Ausführung des erzeugten Programms abhängig.
Selbst in diesem Fall, in dem bereits für die anderen Teile der Entwicklungsumgebung das Parsen des Source-Codes nötig ist, sind wesentliche Informationen erst aus dem übersetzten und gebundenen Programm zu entnehmen. Dies betrifft z.B. die Größe von Objekten und die Adressen von Methoden.
Der wesentliche Teil dabei ist die Übersetzung der Sourcen. Hier ist allein im Hinblick auf den Zeitbedarf ein inkrementeller Compiler und Linker unbedingt nötig.
Nach Möglichkeit ist aus diesen Gründen der Compiler nicht eigenständiges Produkt. Die Integration sollte so erfolgen, daß zu jedem Zeitpunkt so viel Informationen wie möglich gewonnen werden. Das verbietet es, jeweils auf Anforderung einen kompletten Übersetzungslauf zu starten. Im Idealfall werden fast alle Übersetzungsarbeiten in die Phase des Editierens des Source-Codes verlagert.

5. Überblick über ähnliche Arbeiten

Etliche Autoren betonen die Notwendigkeit und Möglichkeit in objektorientierten Programm-Systemen das Laufzeitverhalten besonders zu betrachten.
Hier sollen Arbeiten dargestellt werden, deren Zielsetzung die maschinelle Unterstützung in diesem Bereich ist.

5.1. Unterstützung während des Entwurfs

Während des Designs sehen verschiedene Entwurfs-Methodiken die besondere Berücksichtigung dynamischer Aspekte vor. In dieser Phase beschränkt sich jedoch Werkzeugunterstützung oft auf Hilfe beim manuellen Erstellen der vorgesehen Diagramme.

Um auch während des Entwurfs die intendierte Dynamik über animierte Darstellungen adäquat erfassen zu können, haben Shilling und Stasko ihr Werkzeug GROOVE (Graphical Object Oriented Visualization Environment) entwickelt [ShS94].

Es soll Programmierern erlauben, sowohl statische als auch dynamische Strukturen zu spezifizieren. Für das zeitvariante Verhalten wird eine Animation der Graphen eingesetzt.
Klassen, Objekte und Funktionen besitzen verschieden graphische Repräsentationen (Dreiecke, Kreise, Rechtecke). Pfeile zwischen den Symbolen stehen für Vererbungsbeziehungen und Methodenaufrufe. Farben werden eingesetzt, um Vererbungshierarchien zusätzlich zu verdeutlichen, indem Basisklassen dunkel, abgeleitete Klassen zunehmend heller gezeichnet werden. Die Objekte erscheinen in der Farbe der zugehörigen Klasse.
Im Gegensatz zu den meisten bekannten Methoden gibt es keine getrennten Diagramme für statische und dynamische Modellierung, sondern alle Symbole werden in einem einzigen Graph nebeneinander verwendet. Um die wiedergegebene Information trotzdem überschaubar zu halten, wird ein Fokus benutzt: Dieser kann vom Benutzer auf ein einzelnes Symbol gesetzt werden und bewirkt, daß nur die Zusammenhänge zwischen Symbolen angezeigt werden, die die ausgewählte Entität betreffen. Dazu gehören nicht nur Beziehungen wie Vererbungen, sondern auch Sichtbarkeit von anderen Objekten oder Friend-Relationen.
Die mit GROOVE erzeugten Diagramme und Animationen können einen hohen Informationsgehalt sinnvoll vermitteln. Die Autoren stellen ihr System als visuelles Entwicklungstool vor. Als „ambitonierteste" geplante Erweiterung beschreiben sie ihr Vorhaben, mit denselben graphischen Repräsentationen auch die Ausführung bestehender Programm zu visualisieren, womit sie also exakt die Zielrichtung dieser Studienarbeit verfolgen. Um dieses Vorhaben zu realisieren, arbeiteten sie zum Zeitpunkt der Veröffentlichung an Werkzeugen für eine Code-zu-Code-Transformation, die existierende Programme um Aufrufe der Animationsfunktionen von GROOVE ergänzt.

5.2. Untersuchung implementierter Systeme

Hier sollen Arbeiten dargestellt werden, deren Ziel die Untersuchung bereits implementierter Systeme ist. Ausgangspunkt ist also der Code eines vorliegenden Programms.
Für ein Werkzeug, das Aussagen über dynamische Prozesse in diesem Programm machen soll, sind zwei grundlegende Verfahren denkbar:
Eine Möglichkeit besteht in der Ausführung des zu untersuchenden Programms, wobei interessierende Ereignisse auf irgendeinem Weg registriert werden.
Andererseits ist es auch denkbar, den Source-Code des Programms im Hinblick auf mögliche Ausführungspfade zu analysieren. Für imperative Sprachen ist das Verfahren der Datenflußanalyse hierzu gut erforscht. In der im folgenden beschriebenen Arbeit wird eine Erweiterung um objektorientierte Konzepte vorgestellt.

5.2.1. Objektorientierte Datenflußanalyse

Den meisten objektorientierten Programmiersprachen liegt ein imperatives Sprachkonzept zugrunde. So ist auch bei C++ als Erweiterung von C die herkömmliche Datenflußanalyse ohne prinzipielle Änderungen anwendbar.
Dabei geht jedoch jede durch die Objektorientierung unterstützte Abstraktion verloren.

In [STK94] stellen die Autoren mit der "Hierarchical Data Flow Analysis" (HDFA) eine Methode vor, mit der Datenfluß-Analyse auf Objekt-Niveau durchgeführt werden kann.

Die herkömmliche Datenflußanalyse geht von Variablen aus, für die ein Zustand und Operatoren auf diesem Zustand definiert sind.
Die HDFA dagegen betrachtet eine Hierarchie von Variablen, Objekten, Klassen und Komponenten. Für jede dieser Ebenen sind Zustand und Operatoren unter Rückgriff auf die darunter liegende Ebene definiert.
Für Variablen werden die Definitionen der klassischen Datenflußanalyse übernommen. Bestimmend für Objekte ist die Menge ihrer Attribute (Variablen), für Klassen die Menge ihrer Instanzen und für Komponenten die dazugehörigen Klassen.
Für jeden möglichen Ausführungpfad entsteht durch die Analyse auf jeder Ebene eine Folge von Operatoren, die auf Inkonsistenzen untersucht werden kann.

HDFA ist also eine Erweiterung herkömmlicher Verfahren. In der konventionellen Datenflußanalyse sind die zentralen Begriffe Zustand und (Datenfluß-)Operatoren für Variablen definiert:

Der Zustand einer Variablen ist ihr Wert.

Drei Operatoren sind definiert:
- **D**efine: eine Variable wird definiert, wenn sich ihr Zustand, also ihr Wert ändert
- **K**ill: (oder Undefine) eine Variable wird gelöscht, wenn ihr Zustand nicht mehr zugreifbar ist
- **U**se: eine Variable wird referenziert, ohne daß sich ihr Zustand ändert

HDFA definiert nun eine Hierarchie für objektorientierte Programme, die für die Datenflußanalyse genutzt wird. Zustand und Datenfluß-Operatoren werden pro Hierarchie-Ebene definiert.

1) Wiederverwendbare Komponenten (reusable components)
   Hiermit sind Mengen von Klassen gemeint, die eine klar defnierte und abgegrenzte Funktionalität gegenüber dem Rest des Systems haben.
   (Eiffel und Smalltalk verwenden hier die Begriffe "Cluster" bzw. "Categorie", wie in C++ wird dieses Konzept jedoch von der Sprache selbst nicht unterstützt.)

2) Klassen
   Der Zustand einer Klasse besteht aus den Zuständen aller Klassenvariablen und aller Instanziierungen (Objekte) dieser Klasse.
   Für jede Klassenvariable und für jedes Objekt der Klasse wird ein eigener Datenfluß-Thread, d.h. eine eigener Folge von Operatoren, betrachtet.
   - Define class D_c Eine Klasse wird definiert, wenn sich ihr Zustand ändert, d.h. wenn ein Objekt dieser Klasse erzeugt wird oder eine Klassenvariable defniniert wird.
   - Use class U_c Eine Klasse wird benutzt, wenn eine Klassenvariable oder ein Objekt dieser Klasse ohne Zustandsänderung referenziert wird.
   - Kill class K_c Der Thread eines Objektes ist mit der Destruktion des Objektes beendet, der Thread einer Klassenvariable mit dem Löschen dieser Variable.

3) Objekte
   Der Zustand eines Objektes besteht aus den Zuständen aller Attribute dieses Objektes.
   - Define object D_o Ein Objekt wird definiert, wenn es erzeugt wird oder wenn eine Methode ausgeführt wird, die den Zustand des Objektes ändert.
   - Use object U_o Ein Objekt wird benutzt, wenn es referenziert wird, ohne daß sein Zustand sich ändert.
   - Kill object K_o Das Objekt wird gelöscht und ist nicht mehr zugreifbar.

4) Attribute
   Klassenvariablen und Objektvariablen werden Attribute gennant. Zustand und Operatoren sind wie für Variablen in der konventionellen DFA definiert.

Globale Variablen und zu Methoden lokale Variablen werden ebenfalls wie inder konventionellen DFA behandelt.

Aufgrund dieser Definitionen bestehen Beziehungen (Abhängigkeiten) zwischen den diversen Operatoren auf den verschiedenen Hierachie-Stufen.

So führt z.B. die Benutzung U_a einer Objekt-Variablen zur Benutzung U_o des Objektes, während das Löschen einer Objekt-Variablen K_a zur Definition eines Objektes D_o führt.

Da in einem Statement auf Attribut-Ebene mehrere Operatoren auftreten können, gibt es eine Prioritätsregel, die besagt, welcher Operator auf Objekt-Ebene auftritt:
Der Kill-Operator hat Vorrang vor dem Define-Operator und der Define-Operatore hat Vorrang vor dem Use-Operator.

Die HDFA baut auf diesen Abhängigkeiten zwischen den Datenfluß-Operatoren auf.

Die Analyse wird in zwei Phasen unterteilt:

1) DFA in drei Schritten auf der Klassen-, der Objekt- und der Attribut-Ebene.
   Dabei wird die Analyse auf Attribut-Ebene genutzt um Mehrdeutigkeiten aus den anderen Ebenen zu entscheiden:
   Deklarationen, Zuweisungen und Kontroll-Anweisungen führen eindeutig zu Datenfluß-Operatoren.
   Dagegen sind Anweisungen, die Methoden- oder Prozedur-Aufrufe enthalten prinzipiell mehrdeutig.
   Deshalb wird in der Analyse auf Attribut-Ebene der Rumpf der Methode bzw. Prozedur ausgewertet.

2) Auflösen der verbleibenden Mehrdeutigkeiten aus Phase 1) durch die zwischen den Datenfluß-Operatoren bestehenden Abhängigkeiten.

Aufspüren von Anomalien

Beim Datenfluß-Testen werden entsprechend den Kontroll-Anweisungen Pfade durch das Programm gewählt und die dabei entstehenden Datenfluß-Operatoren ausgwertet.
Eine unsinnige Sequenz von Datenfluß-Operatoren heißt Anomalie.

So ist die Benutzung eines Objektes nach seinem Löschen solch eine Anomalie.

Die mehrfache Definition eines Attributes ohne zwischenzeitige Benutzung gilt auch als Anomalie.

HDFA soll den Fokus bei der Datenfluß-Analyse objektorientierter Programme auf den Zustand von Objekten legen. Durch die Ausnutzung der in objektorientierten Programmen natürlich vorhandenen Hierarchie sind Anomalien auch auf Objekt- und Klassen-Ebene feststellbar.

Bewertung

Bei der HDFA entstehen für jeden möglichen Ausführungs-Pfad im Programm viele Sequenzen von Datenfluß-Operatoren.
Diese können maschinell auf Anomalien untersucht werden.
Für die praktische Einsetzbarkeit ist es aufgrund der großen Anzahl unbedingt nötig, daß ebenfalls maschinell mit hinreichender Genauigkeit entschieden werden kann, welche Anomalien wirklich Codierungs- oder Design-Fehler darstellen.
So tritt z.B. bei beiden folgenden Code-Auszügen die Definition eines Objektes ohne zwischenzeitliche Benutzung zweimal nacheinander auf. Während es sich im ersten Fall um einen Fehler handelt, ist dieser Effekt im zweiten Fall gewollt.

1)

```
a = aStack.Pop();
if(a = b) ...
```

2)

```
aStack.Push(a);
aStack.Push(b);
```

Während in diesem Fall die Anomalie noch leicht per Hand entscheidbar ist, kann bei komplexen Fällen über mehrere Objekte und Methoden eine Entscheidung schon sehr viel - menschliche - Arbeit verlangen.

Die Komplexität der Analyse steigt mit der Programm-Komplexität extrem an. Jede zusätzliche Klasse, Methode, Variable, jedes neu erzeugte Objekt steigert die Gesamt-Komplexität.
C++ ist zudem eine sehr komplexe Sprache, die viele Ausnahmen und Mehrdeutigkeiten enthält, die in der Analyse berücksichtigt werden müßten.

Mit der DFA kann jedoch zumindest theoretisch für jeden denkbaren Ausführungspfad das Auftreten von Anomalien vermieden werden.

Für die meisten in der Praxis auftretenden Systeme ist der Aufwand der DFA so enorm, daß die Analyse entweder schon am Rechenaufwand scheitert, das Nachvollziehen der Anomalien für Menschen in erträglicher Zeit nicht möglich ist, oder der Aufwand aus wirtschaftlichen Gründen den Nutzen bei weitem übersteigt.

Lediglich bei kleineren und kritischen Applikationen erscheint der Einsatz von DFA möglich. Inwieweit auch dann andere Verfahren der Programm-Verifikation sinnvoller sind, sollte im Einzelfall geprüft werden.

5.2.2. Beobachtung der Ausführung eines Programms

Gegenüber der Datenflußanalyse entspricht das Beobachten des laufenden Programms der Aufzeichnung der auftretenden Ereignisse (Operatoren) entlang nur eines Pfades. Dabei können in der Regel nicht alle Kombinationen getestet werden, doch eine genügend große Zahl von Testläufen ermöglicht meist die Abdeckung der praktisch auftretenden Fälle.

Die technischen Randbedingungen haben für diese Anwendung naturgemäß großen Einfluß auf Konzeption und Möglichkeiten entsprechender Verfahren.
So führt ein Ansatz beim Object-Code eines zu untersuchenden Programms in wichtigen Punkten zu einem anderen Verhalten als der Ansatz beim Source-Code.
Noch wichtiger erscheint uns jedoch, inwieweit eine Auswertung schon direkt während der Laufzeit möglich ist.
Im besten Fall ist dadurch eine Abbildung der Interaktion des Benutzers mit dem Programm auf die internen objektorientierten Strukturen simultan möglich. Dabei entsteht zum einen eine Benutzer-Sicht, d.h. das untersuchte Programm stellt sich unverändert dar und interagiert wie vorgesehen mit dem Benutzer.
Zum anderen wird eine Entwickler-Sicht geboten, die Einblick in die Interna des Programms gibt. Hier sind Informationen auf verschiedenen Abstraktionsstufen möglich. Darstellungen des Source-Codes, von Objekten und Nachrichten bis zu statistischen Informationen über das Gesamtsystem bieten ein zusammenhängendes Bild.
Wesentlich hierbei ist, daß beide Sichten simultan dargestellt und aktualisiert werden. Damit ist es erstmals möglich, diese Sichten auf einem angemessenen Niveau in Beziehung zu setzen. Bestimmtes Verhalten des untersuchten Programms an seiner Oberfläche kann dazu führen, daß in der Entwickler-Sicht Objekte, Methoden im Source-Code usw. genauer betrachtet werden. Umgekehrt kann z.B. ein beobachtetes Objekt-Verhalten die Notwendigkeit einer anderen Benutzung des untersuchten Programms bewirken.
Durch diese Kombination bietet sich eine komplett neue Perspektive, die allein schon für Entwickler sehr hilfreich ist. Sämtliche Implementierung erfolgt über eine statische Beschreibung in Form von Source-Code. Im Normallfall ist nur das Ergebnis eines Programmablaufs sichtbar, jetzt dagegen im Idealfall ein Bild des Programms selbst in vita.
Darüberhinaus bieten sich Möglichkeiten zum Testen, Debugging, Einarbeitung in fremde Programme, ... die andere Verfahren nicht ermöglichen.

Einige Werkzeuge zur Visualisierung arbeiten post-mortem, d.h. bieten erst nach kompletten Ablauf des untersuchten Programms die Möglichkeiten aufgetretene Ereignisse darzustellen.
Es sind also zwei getrennte Phasen nötig, wobei in der ersten Phase sämtliche Ereignisse während des Programm-Ablaufs registriert werden. In einer zweiten Phase werden die gesammelten Daten ausgewertet, wobei statische wie auch animierte Darstellungen möglich sind.
Hierbei geht gerade der wesentliche Vorteil der Beobachtung des Programms verloren. Die Benutzung des Program muß völlig unabhängig von den durch die Beobachtung gewonnenen Informationen bleiben. Erkenntnisse aus der späteren Visualisierung können bestenfalls in einem weiteren Programmlauf berücksichtigt werden. Fehlerhaftem oder einfach unerwartetem Verhalten des beobachteten Programms kann nicht direkt weiter nachgeforscht werden.
Der Schwerpunkt bei diesem Vorgehen, liegt deshalb in anderen Bereichen wie z.B. Schulung, Dokumentation, Profiling.
Unabhängig vom Vorgehen fallen recht große Datenmengen an, die effizient verwaltet werden müssen.

5.2.2.1. Visualisierung post-mortem

Diese Systeme können einfacher aufgebaut sein, als die Ein-Phasen-Systeme mit sofortiger Visualisierung. Zum Zeitpunkt der Darstellung sind sämtliche Informationen bereits vorhanden. So ist z.B. das Layout animierter Darstellungen einfacher durchzuführen, da alle zukünftigen Änderungen bekannt sind.
Die Geschwindigkeit des Ablaufs des beobachteten Programms in der ersten Phase kann deutlich höher liegen, da lediglich alle anfallenden Daten effizient gespeichert werden müssen. Dagegen wird bei sofortiger Visualisierung bei komplexen Strukturen mit vielen beteiligten Objekten die Bedienbarkeit des untersuchten Programms u.U. erheblich beeinflußt.
Sämtliche Darstellungen können während der zweiten Phase beliebig abgespielt werden, so ist z.B. auch eine wiederholte Animation interessanter Abschnitte leicht möglich. Dabei kann der Detaillierungsgrad der Anzeige verändert werden, so daß das gleiche Szenario auf unterschiedlichen Abstraktionsebenen erfaßt werden kann.
Eine solche Trennung von Darstellung und Programmablauf hat also Vorteile, kann aber mit den Ein-Phasen-Systemen nur mit erheblicher Komplexität für Benutzer und Entwickler des Werkzeugs erreicht werden: Performanz ist hier für Entwickler wesentlich bedeutender. Die Möglichkeit der temporären Trennung von Visualisierung und Programmablauf wie sie nötig wäre, um ein wiederholtes Abspielen zu ermöglichen, bedeutet hier ebenfalls Aufwand für den Entwickler, aber auch zusätzliche Schwierigkeiten bei der Benutzung.
In der zweiten Phase wird genau eine bestimmte Folge von Ereignissen visualisiert, die durch Interaktion des Benutzers mit dem untersuchten Programm entstanden ist.
Diese Interaktion kann jedoch während der Darstellung kaum nachvollzogen werden. Medium dieser Interaktion waren die Oberflächenelemente des untersuchten Programms. Da diese zum Zeitpunkt der Animation nicht mehr vorhanden sind, fehlt eine adäquate Darstelllungsmöglichkeit der erfolgten Benutzung des beobachteten Programms. Lediglich die Ergebnisse z.B. eines Mausklicks werden in Form von Nachrichten an Objekte sichtbar. Auf diesem Niveau scheint es sehr schwierig bis unmöglich, auf Interaktionsfolgen und -muster des Benutzers rückzuschliessen. So bleibt als einziges das implizite Wissen des Benutzers selbst, der in der Regel sowohl das untersuchte Programm als auch das Werkzeug benutzt.
Dies ist jedoch nicht der Fall, wenn zur Dokumentation und zur Schulung die Visualisierungskomponente allein benutzt wird.

Die Schwierigkeiten Interaktion nachzuvollziehen fallen natürlich abhängig von der Art des untersuchten Programms unterschiedlich ins Gewicht. Bei komplexer Anwendungslogik mögen sie sogar zu vernachlässigen sein. Das andere Extrem bilden reaktive oder stark oberflächen-orientierte Systeme, wie jede Art von Editoren. Hier besteht der wesentliche Teil der Applikation häufig aus der Koordination von Eingaben und Darstellungen, was eine Visualisierung post-mortem fast sinnlos macht.

Mit Graphtrace wird in [KIG88] ein Werkzeug beschrieben, das in Strobe geschriebene Programme behandelt. Strobe ist eine Erweiterung von Common Lisp. Dadurch ist die Beobachtung des ablaufenden Programmes recht einfach über eine Änderung der low-level Lisp-Funktionen zur Nachrichten-Weiterleitung möglich.

Der Zielsetzung dieser Arbeit deckt sich ansonsten weitgehend mit Voop: "a motion picture of the entire system's dynamic behaviour"

Verschiedene Views stellen in Graphen-Form statische (structural) und dynamische (behavioral) Aspekte dar.

strukturelle (statische) Sichten:
- taxonomische Views:
  - Progeny Graph: Vererbungsbeziehung
  - Ancestry Graph: zeigt zusätzlich wie die Vererbungsbeziehung ausgenutzt wird: bei Aufruf einer Methode wird ggf. die Oberklasse hervorgehoben, die die Methode implementiert.
- part-whole graph

dynamische Sichten:
- method invocation graph: ein Baum, der die Schachtelung von Nachrichten darstellt. Ein depth-first-Durchlauf entspricht der Historie von versendeten Nachrichten. Wiederholte Aufrufe derselben Methode werden als ein Knoten dargestellt. Bei Rekursion werden aber mehrere Knoten mit dem gleichen Namen gezeichnet.
- object invocation graph: die Knoten zeigen nur den Objektnamen. Damit wird ein mehr genereller Überblick ermöglicht.
- object-slot invocation graph: jeder Knoten enthält ein bestimmtes Object mit einer bestimmten Methode.

Um die dynamischen Views darzustellen ist ein zwei-Phasen Vorgehen nötig:
In einem Record-Lauf des Programms werden sämtliche Informationen gesammelt.
Das Programm muß dann erneut gestartet werden, um eine animierte Darstellung zu erhalten. Dabei werden dann sämtliche Graphen von Anfang an vollständig dargestellt, wobei sich die Animation auf das Hervorheben (highlighting) der gerade aktiven Teile beschränkt. In dieser einfachen Möglichkeit der Animation und des Layouts liegt der Hauptgrund für das Vorgehen in zwei Phasen.

Der Schwerpunkt liegt - sicherlich auch aufgrund dieses Ansatzes - auf der Benutzung des Werkzeuges zum Programmverständnis.
Es wird hervorgehoben, daß dieses Verständnis Vorbedingung von Wiederverwendbarkeit ist.

Die Verwendung als Debugger und als Profiler ist als Erweiterung angedacht. Außerdem könnten Icons aus dem Bereich der Anwendung den Nachrichtenfluß spezifischer darstellen.

Anders als mit herkömmlichen Werkzeugen soll es mit Graphtrace möglich sein, die Vorgänge in einem System sehr detailliert darzustellen, aber auch größere Zusammenhänge zu erfassen.
Herkömmliche Programmier-Umgebungen bieten oft die Möglichkeit, durch statische Analyse des Codes die (möglichen) Schachtelungen von Funktionsaufrufen darzustellen. Durch Polymorphie verliert diese Darstellung an Aussagekraft und muß durch dynamisch generierte Diagramme wie den "method invocation graph" ersetzt werden.

Interessant an dem beschriebenen System ist die Möglichkeit für den Anwendungsprogrammier sogennante generator functions zu definieren.
Damit können auch Zusammenhänge, die von Strobe nicht direkt als Sprachkonstrukt unterstützt werden, in GraphTrace dargestellt werden.

Z.B. gibt es in Strobe keine syntaktische Möglichkeit part-whole-Beziehungen auszudrücken. Ein Applikationsprogrammier modelliert diese Beziehung auf höherer Ebene. Die generator function dazu muß dann angeben, wie ein entsprechender Beziehungsgraph aus einem Objekt mit den entsprechenden Links konstruiert werden kann.
Entsprechend ist es möglich jede Form von höheren, d.h. fachlich motivierten Beziehungen einzubringen.
Diese Fähigkeit wird natürlich wesentlich dadurch unterstützt, daß es sich hier um ein interpretierendes System handelt, in dem die direkte Verbindung von Graphtrace mit dem zu untersuchenden Programm möglich ist.

In einer übersetzenden Entwicklungsumgebung wäre eine ähnliche Funktionalität nur über eine Bibliothek erreichbar, die die Verbindung zum Werkzeug herstellt.

In [ChG92] wird ein Werkzeug beschrieben, das ebenfalls hauptsächlich das bessere Verständnis bestehender Programme ermöglichen soll, aber für C++ vorgesehen ist.

Hauptgedanke bei der Entwicklung dieses Werkzeuges ist die Einarbeitung in bestehende Frameworks zu vereinfachen.
Diese Frameworks sind üblicherweise nur auf einer Methodenebene dokumentiert.
Weitergehende Informationen sind oft nur aus dem Source-Code zu entnehmen: "All to often programmers are forced to become source code archaeologists."

Die Design-Ebene ist häufig nicht dokumentiert. Gerade das Design des Frameworks muß aber verstanden werden, um einen sinnvolle Verwendung des Frameworks zu ermöglichen.

Die Autoren heben zwei Methoden heraus, die bisher bereits zur Erleichterung des Programm-Verständnisses genutzt werden:

1) Analyse des Source-Codes, Aufbau einer Datenbank mit Informationen über die Struktur des Programms. Möglichkeiten, diese Datenbank abzufragen. (z.B. der Ansatz der C++-Entwicklungsumgebung Sniff++)
2) Beispiel Applikation mit einem Source-Level-Debugger untersuchen.

Beide Methoden setzen bereits ein gewisses Verständnis des Frameworks voraus.

Das hier beschriebene Werkzeug setzt früher an, indem das zu untersuchende Programm vor der Übersetzung um Trace-Funktionen erweitert wird.
Während des Ablaufs erzeugt das so instrumentierte Programm ein Trace-File, das von einer weiteren Komponente in eine animierte Sicht dieses Ablaufs umgesetzt werden kann.

Erzeugte Objekte werden durch Icons dargestellt und nach Klassen gruppiert. Für jede Klasse kann dabei aus einer Bibliothek von Icons eines gewählt werden.
Der momentane Call-Trace wird als Folge von Icons dargestellt mit jeweils dem Methoden-Namen daneben.
Die Auswahl der zu beobachtenden Klassen kann bereits vor der Übersetzung getroffen werden und bei der Animation weiter eingeschränkt werden. Dadurch ist auch der Call-Trace nicht unbedingt vollständig.

Der Begriff "Jo-Jo-Problem" wurde in [TGP89] geprägt:

> In tiefen Klassen-Hierarchien in denen Polymorphismus genutzt wird, kommt es oft zu Sequenzen von Methoden-Aufrufen für dasselbe Objekt auf verschiedenen Ebenen der Hierarchie.
> Dies ist mit statischen Beschreibungen wie auch mit den textbasierten Darstellungen von Debuggern schwer nachzuvollziehen.

Eine Werkzeug wie dieses ermöglicht hier eine sehr viel schneller erfaßbare, graphische Darstellung.

Die Benutzung als Debugger wird als nebensächlich angesehen.

Durch die Beobachtung über Änderung des Source-Codes und die Animation nach dem eigentlichen Lauf des Programms wird der Einsatzbereich des Werkzeuges stark eingeschränkt.

Die Autoren erkennen die Möglichkeiten kaum, die in einer anderen Vorgehensweise liegen.

So sind sie auch der Meinung, daß der Bedarf an Werkzeugen dieser Art mit verbesserten Dokumentationen von Frameworks geringer werden wird.

Das Werkzeug gibt es in einer Unix-Version, sowie in einer Version für das MacApp-Framework, in der einige Eigenheiten dieses Frameworks mit ad-hoc-Methoden berücksichtigt sind.

5.2.2.2. Visualisierung während des Programm-Ablaufs

Sehr viel weniger Arbeiten beschäftigen sich mit Werkzeugen, die ein Programm direkt während seines Ablaufs visualisieren können. Dabei ist sicherlich die gesteigerte technische Komplexität der wichtigste Grund.

Im T. J. Watson Research Center der IBM wurde ebenfalls ein System zur Visualisierung der Ausführung objektorientierter Programme erarbeitet. Dieses System wurde prototypisch für die Sprache C++ implementiert [PHK93].

Bemerkenswert ist die zugrundeliegende, recht formale und allgemeingültige Modellierung. Prinzipiell ist eine Visualisierung sowohl während des Programm-Ablaufs als auch post-mortem möglich. Das Konzept ist plattform- und sprachunabhängig, bisher aber nur für C++ angewendet. Es ist darauf angelegt mehrere Darstellungen simultan anzuzeigen und kann einfach um neue Visualisierungen ergänzt werden.
Die Arbeitsweise beruht auf einer Instrumentierung des Source-Codes. Die bisher vorgestellten Diagramme befinden sich alle auf einer sehr hohen Abstraktionsstufe (meist klassen-orientiert).

Ein allgemein verwendbarer Annotator arbeitet als Preprozessor. Er wird dabei von einem Skript (annotation script) gesteuert, was prinzipiell die Verwendung beliebiger Sprachen erlaubt.
Während der Instrumentierung wird das Programm um ein eigenes RTTI-System (Run-time Type Information) ergänzt, was es möglich macht, zur Laufzeit Objekte entsprechend ihres Typs zu untersuchen.
Der ergänzte Code erzeugt während des Ablaufs einen Fluß von Ereignissen.

Ein eigenes Protokoll dient zur Kommunikation zwischen dem beobachteten Programm und dem außerhalb liegenden Werkzeug.

Um kritische Ressourcen effizient zu nutzen, wurde ein Konzept entwickelt, nach der die für die Visualisierung notwendigen Informationen
- möglichst vollständig erfaßt,
- möglichst kompakt gespeichert
- und für schnelle Zugriffe ohne großen Verwaltungsaufwand

zugänglich sind.

In [PKV94] wird hauptsächlich dieser Teil der Arbeit vorgestellt. Die Autoren charakterisieren die Ausführung eines Programms als Folge von interessierenden Ereignissen. In objektorientierten Systemen sind solche Ereignisse z.B. Methodenaufrufe, die als Punkte in einem vierdimensionalen Ereignisraum (event space) eingetragen werden können. Die Dimensionen dieses Raums sind Klassen, Instanzen, Methoden und Zeitpunkte. Jedes Ereignis kann damit als 4-Tupel beschrieben werden.
Die Ereignisse können als differentielle oder als integrale Information zur Verfügung gestellt werden: Im ersten Fall wird jedes Ereignis einzelnd gemeldet, im zweiten werden Ereignisse akkumuliert.

Die Autoren stellen eine effiziente Struktur vor, um integrale Information über Methodenaufrufe zu verwalten, die dann die Basis für Animationen und Analysen des dynamischen Programmverhaltens bilden. Diese Struktur nennen sie *call frame*; sie enthält zu jedem Methodenaufruf die Angaben über Klasse, Instanz und Name der aufrufenden sowie der aufgerufenen Methode. Durch Projektionen der call frames auf verschiedene Ebenen ergeben sich verschiedene Sichten auf den Programmablauf. Eine Sicht ist beispielsweise ein *inter-class call cluster*, der Aufrufe zwischen Klassen animiert (d.h. Nachrichten zwischen Objekten werden den dazugehörigen Klassen zugeordnet), eine andere die *inter-class call matrix*, in der die kumulierten Anzahlen der Aufrufe dargestellt werden. Der Schwer-

punkt der Arbeit liegt jedoch nicht auf den Visualisierungen, sondern auf der Verwaltung des Ereignisraums.
Das vorgestellte Konzept erlaubt es, auf einfache Art auf die gesammelten Informationen zuzugreifen. Im Gegensatz zu anderen Ansätzen werden als Schwerpunkt nicht die Instanzen, sondern im gleichen Maße auch die Klassen und Methoden betrachtet. Dadurch eignet sich die Datenbasis gut für unterschiedliche statistische Auswertungen und damit für das Profiling. Weniger allgemeine Verwendungszwecke werden nicht extra berücksichtigt; für ein Werkzeug, das gezielt zur Fehlersuche eingesetzt werden soll, erscheint die Modellierung nicht vorteilhaft. In unseren Augen ist unter solchen Voraussetzungen ein fachlich motivierter objektorientierter Entwurf zweckmäßiger als das abstraktere Konzept des Ereignisraums.

Eine graphische Notation zur Darstellung von Objekten und Nachrichten wird in [Cub86] vorgestellt. Es wird jedoch nur ein kleiner Teil der vorstellbaren Zusammenhänge aufgezeigt.
Ergänzend haben die Autoren einen vorhandenen Debugger in einem Smalltalk-80-System so erweitert, daß Nachrichten aufgezeichnet und automatisch in Graphen umgesetzt werden; manuelle Korrekturen des Graph-Layouts sind während der Erstellung möglich.
Zustände der Objekte und zeitliche Reihenfolge der Nachrichten werden jedoch nicht abgebildet, eine Animation des Programmverhaltens findet nicht statt. Die Graphen beschränken sich darauf, den logischen Zusammenhang von verschiedenen Objekten und deren Methoden wiederzugeben.
Jedes Objekt wird als ein Rechteck gezeichnet, neben dem links der Klassenname steht. Um eine Basisklasse anzuzeigen, wird das Rechteck horizontal geteilt, so daß im oberen Teil die am meisten abgeleitete Klasse, darunter die Basisklasse erscheint. Nachrichten werden durch Pfeile verkörpert, an deren Spitze der Name der Methode steht, die durch die Nachricht aufgerufen wird. Intention der Autoren ist, insbesondere auch die Nachrichten zu veranschaulichen, die innerhalb eines Objektes gesendet werden. Dies ist zum einen der Fall, wenn ein Objekt in einer Methode eine andere eigene Methode aufruft, zum anderen, wenn (eventuell gleichnamige, weil überladene) geerbte Methoden der Basisklassen benutzt werden.

Die Graphen fangen nicht einen einzigen Zeitpunkt des laufenden Programms ein, sondern stellen eine logische Folge von Nachrichten dar. Wiederholt gesendete Nachrichten werden nicht mehrfach gezeichnet. Die Notation kann daher zwar bestimmte Kausalzusammenhänge von verschiedenen Klassen wiedergeben, erlaubt aber keine exakte Analyse des tatsächlichen Laufzeitverhaltens.
Einsatzzweck der entwickelten Diagramme ist für die Autoren hauptsächlich die Verwendung in der Ausbildung, um das Verständnis für das Zusammenspiel von Klassen und Objekten zu unterstützen.
Vereinfachend wirkt hier sicherlich die Wahl von Smalltalk zur Implementation und die enge Beschränkung auf den Einsatz zur Erzeugung bestimmter Diagramme.

Ein Werkzeug namens "Look" zur Untersuchung von C++-Programmen wird in [Wes93] beschrieben. Inzwischen wird es für viele Plattformen kommerziell vertrieben.
Es wurde entworfen mit dem Ziel der möglichst allgemeinen Einsetzbarkeit in Bereichen wie Testen, Debugging, Einarbeitung in fremde Programme und Frameworks sowie Schulung. Damit soll eine Steigerung der Produktivität, der Wiederverwendbarkeit und Qualität von Programmen erreicht werden.
Look bietet dazu etliche animierte und statische Darstellungen, weitgehende Konfigurierbarkeit und einen kompletten, integrierten Debugger.
Sämtliche Darstellungen werden während der Ausführung des beobachteten Programms simultan aktualisiert.
Das zu beobachtete Programm ist unverändert zu bedienen, währenddessen bietet Look gleichzeitig den Einblick in die internen Abläufe auf objektorientierter Ebene. Dabei ist jederzeit ein Eingriff in das untersuchte Programm möglich. Dazu bietet der Debugger Funktionen auf Source-Code-, Objekt- und Klassen-Ebene an, und sämtliche Anzeigen können während des Laufs konfiguriert und ein- oder ausgeblendet werden.
Das Werkzeug arbeitet unabhängig vom Source-Code allein mit dem Object-Code, der dafür natürlich mit Debug-Informationen vorliegen muß.
Damit deckt sich die Zielsetzung in vielen Punkten mit unseren Vorstellungen eines optimalen Werkzeugs.

Die Autoren sehen, daß die Notwendigkeit für ein Werkzeug dieser Art, aber auch seine Möglichkeiten erst mit der Objektorientierung entstanden.
Eine dezentrale Struktur interagierender Objekte steigert die dynamische Komplexität erheblich. Andererseits bieten Klassen mit ihren Methoden, Vererbungs- und Referenzbeziehungen und Instanzen eine dritte Ebene über der Source- und der Maschinen-Code-Ebene, die maschinell erfaßt und dargestellt werden kann.

Während des Programm-Ablaufs können Objekte und ihre sich ändernden Beziehungen und der aktuelle Fluß von Nachrichten animiert werden. Das Layout dieser Diagramme bereitet dabei besondere Schwierigkeiten, da sowohl die Menge der Objekte als auch ihre Beziehungen dynamisch sind und auftretende Änderungen im Allgemeinen nicht vorhergesagt werden können.
Objekte werden in den dazugehörigen Diagrammen als graphische Elemente dargestellt, wobei je nach Speicherart (static, auto, heap) die Darstellung eine andere Form annimmt. Folgende Informationen können in Objektdiagrammen angezeigt werden:
Referenzierungen: zwischen einem Objekt und allen daraus referenzierten Objekten
Erzeugungsbeziehungen: zwischen einem Objekt und allen daraus erzeugten Objekten
Nachrichten: zwischen Sender- und Empfänger-Objekt
Klassenzugehörigkeit
Eine von diesen Beziehungen ist in einer Darstellung von Objekten für das Layout bestimmend. Unabhängig davon, wonach sich das Layout richtet, sind jedoch alle Informationen in jedem Diagramm darstellbar, wobei der Nachrichten-Fluß immer dargestellt wird.
Für jedes Objekt kann weitere Information visualisiert werden, wie es ist aktiv, es wurde referenziert, es wurde gelöscht, während es aktiv war, Größe im Speicher u.a.

Zusätzlich sind statische Beschreibungen wie ein Klassen-Diagramm und eine Sicht auf den Source-Code mit dem gerade ausgeführten Code vorgesehen.
Diese verschiedenen Sichten können gleichzeitig angezeigt und aktualisiert werden und ermöglichen in Kombination ein besseres Verständnis des Programmablaufs.
So liegt ein möglicher Einsatzzweck auch in dem Verständnis von C++-Details wie der Konstruktions-/Destruktionsreihenfolge von Objekten, temporären Objekten, Auflösung von Methoden-Aufrufen in einer Vererbungshierarchie, Typ-Konvertierungs-Operatoren u.a.
Diverse Fehler in der Speicher-Benutzung können automatisch erkannt werden.

Die Animation kann in ihrer Geschwindigkeit kontrolliert und jederzeit unterbrochen werden. Zu jedem Zeitpunkt kann von den Objektdarstellungen zu der darunterliegenden Sicht auf Source-Code und Variablen-Inhalte übergegangen werden. Die schrittweise Ausführung auf Nachrichten- oder Source-Ebene ist möglich.
Look bietet alle wesentlichen Möglichkeiten eines Debuggers ergänzt um objektorientierte Möglichkeiten.

In jedem Programm-System von signifikanter Größe wird es bei weitem zu viele Objekte und Nachrichten geben, um sinnvoll eine vollständige Animation zu erlauben.
Deshalb ist eine statische Filterung nach Modulen, Klassen, Methoden und Funktionen vorgesehen, die einfach über die Diagramme anwählbar ist.
Eine zusätzliche dynamische Filterung kann Objekte und Nachrichten berücksichtigen, muß aber in einer C++-ähnlichen Syntax programmiert werden.

Eine Animation auf Design-Ebene, d.h. vor der Implementation ist angedacht, aber nicht vorgesehen. Als Hauptschwierigkeit wird hier das aus CASE-Werkzeugen bekannte Problem der Konsistenz von Design und Implementierung während der Evolution des Systems gesehen.

Teil B: Der Prototyp Voop

Unsere Vorstellungen von einem visuellen Analysewerkzeug bildeten den Ausgangspunkt zur Realisierung eines solchen Programms für die Sprache C++, dem wir den Arbeitsnamen *Voop* gaben: Visualisierung objektorientierter Programme. Die Entwicklung von Voop nahm den Schwerpunkt unserer Studienarbeit ein; das Ergebnis ist ein einsetzbarer Prototyp, dessen Voraussetzungen, Entwicklung und Weiterführung in den nachfolgenden Kapiteln beschrieben werden.

6. Objektorientierung - vom Konzept zum Object-Code

Zu Beginn der Arbeit stand für uns fest, daß Voop nicht auf den C++-Source-Code eines zu untersuchenden Programms angewiesen sein sollte, sondern allein mit dessen Object-Code auskommen muß. Der Object-Code entsteht durch das Compilieren und Linken des Source-Codes. Wir müssen daher verstehen, wie Konstrukte objektorientierter Sprachen in C++ verwirklicht sind und in Object-Code transformiert werden. Dieser Abschnitt soll den Zusammenhang darstellen von

- allgemeinen objektorientierten Konzepten,
- deren Verwirklichung in C++
- und die prinzipielle Umsetzung in ein ablauffähiges Programm, also die Übersetzung in Object-Code.

6.1. Konzepte objektorientierter Programmiersprachen

Hier soll es vor allem darum gehen, objektorientierte Konzepte aufzuführen und eindeutig zu benennen. Damit kann in den folgenden Abschnitten dargestellt werden, welche Effekte sich aus diesen objektorientierten Abstraktionen auf darunterliegenden Ebenen ergeben. Zur näheren Erläuterung dieser Konzepte verweisen wir auf die Literatur.
Informatik beschäftigt sich mit der Analyse informationsverarbeitender Systeme, ihrer Abstraktion in Form von Modellen und deren Konkretisierung als Simulation auf Automaten. Die Objektorientierung tritt mit dem Anspruch auf, mit dem zentralen Konzept des Objektes auf diesen Ebenen (Analyse, Abstraktion, Konkretisierung) eine homomorphe Beschreibung zu bieten.

- Auf der Modellebene ist ein *Objekt* die Kapselung von Verhalten und Zustand zu einer Einheit. Objekte sind von außen nur über ein Protokoll zugreifbar. Dieses Protokoll besteht aus einer Menge von *Nachrichten*, die ein Objekt empfangen kann.
- Der Empfang einer Nachricht führt zur Aktivierung des Objektes und der Ausführung einer durch die Nachricht bezeichneten *Methode*, die einen Algorithmus implementiert. Die Nachricht kann Objekte als Argumente an die Methode mitliefern. Eine Nachricht kann ein Objekt an den Sender der Nachricht zurückliefern.
- Ein Objekt kapselt eine Menge von *Attributen*; das sind Objektvariablen (oder auch *Instanzenvariablen*), die nur in Methoden des Objektes zugreifbar sind.
- Als *Zustand* eines Objektes wird die Zusammenfassung aller Werte der Objektvariablen bezeichnet. Der Zustand ist damit nur durch Methoden des Objekts selbst veränderbar[9].
- *Klassen* sind Schemata für gleichartige Objekte. Sie sind Vorlage für das Erzeugen und Löschen von Objekten. Objekte werden deshalb als *Instanzen* oder auch Exemplare einer Klasse bezeichnet. In getypten OO-Sprachen bilden Klassen die Grundlage des Typsystems.
- Sogenannte *Klassenvariablen* sind an die Klasse gebunden, d.h. unabhängig von den Objekten der Klasse besteht nur eine Ausprägung der Variablen. Diese sind in der Regel von allen Objekten der Klasse zugreifbar.
- *Klassenmethoden* sind an die Klasse gebundene Methoden, die deshalb nur auf Klassenvariablen und nicht auf den Zustand einzelner Objekte zugreifen können. Einige Sprachen beschreiben Klassen wiederum selbst als Objekte einer *Meta-Klasse*.
- Eine Klasse kann von einer anderen *erben*, d.h. ihre Methoden und Attribute übernehmen. Dabei können Methoden *redefiniert*, d.h. durch eine andere Implementation ersetzt werden. Zusätzlich können in der Unterklasse Attribute und Methoden ergänzt werden. Das direkte Erben von mehr

---

[9] In vielen objektorientierten Sprachen ist diese Beschränkung vorgegeben. In C++ ist es guter Programmierstil keinen anderen Zugriff zu erlauben.

als einer Klasse wird als *Mehrfachvererbung* bezeichnet. *Wiederholtes Erben* ist das mehrmalige, direkte oder indirekte Erben von derselben Klasse.
- Als *Signatur einer Nachricht* wird Anzahl und Typ ihrer Argumente verstanden. Die *Signatur einer Methode* gibt die Anzahl der Argumente an, ob ein Rückgabewert geliefert wird und, bei statisch typisierten Programmiersprachen, den Typ der Argumente und des Rückgabewertes.
- *Kovarianz* ist die Möglichkeit, in redefinierten Methoden den Typ eines Arguments auf eine Unterklasse des ursprünglichen Typs einzuschränken. *Kontravarianz* meint die Erweiterung solch eines Typs auf eine Oberklasse.
- Als *Überladen* wird die Möglichkeit bezeichnet, mehreren Methoden gleiche Namen zu geben. Anhand der Signatur einer Nachricht muß die zugehörige Methode gefunden werden können.
- *Generizität* bedeutet, daß Klassen oder Methoden mit Typen parametrisiert werden können.
- Bei statisch getypten Sprachen wird der Typ, mit dem eine Variable deklariert ist, als *statischer Typ* der Variablen bezeichnet. Als – eingeschränkte – *Polymorphie* wird die Fähigkeit von Variablen bezeichnet, auch Objekte jeder Oberklasse ihres statischen Typs aufnehmen zu können. Der *dynamische Typ* einer Variable ist der Typ des enthaltenen Objekts.
- Als *dynamische* (oder *späte*) *Bindung* wird die Zuordnung von Nachrichten zu Methoden zur Laufzeit des Programms anhand des dynamischen Typs einer Variablen verstanden.
- Einige Sprachen bieten explizite Formen, um *Delegation* auszudrücken: Damit ist die Fähigkeit von Objekten gemeint, Nachrichten nicht selbst zu verarbeiten, sondern an andere Objekte zur Bearbeitung weiterzuleiten.

6.2. Umsetzung objektorientierter Konzepte in C++

Die Programmiersprache C++ konnte sich seit Mitte der achtziger Jahre als am weitesten verbreitete objektorientierte Sprache durchsetzen. Ein wesentlicher Grund für diesen Erfolg liegt in der Verbreitung von C, auf dessen Fundament C++ entstand.
Für das Programm Voop wählten wir C++ als Implementierungssprache und als die Quellsprache der von Voop untersuchten Programme. Für diese Entscheidung gibt es eine Reihe von verschiedenen Gründen:
- *Verbreitung von C++*
  Unser Ziel war die Entwicklung eines im Programmieralltag einsetzbaren Softwarewerkzeugs. Die große Anzahl bestehender C++-Programme bietet daher ein größeres Anwendungsfeld, als es für andere objektorientierte Programmiersprachen der Fall wäre.
- *Fehlen einer garbage collection in C++*
  Im Gegensatz zu anderen Sprachen (z.B. Eiffel) besitzt C++ keine garbage collection: Anstelle des Laufzeitsystems ist der Programmierer für das Löschen von nicht mehr benötigten Objekten selbst verantwortlich. Für das Beheben der Fehler, die in diesem Zusammenhang häufig gemacht werden, wird ein Programm wie Voop besonders interessant.
- *Einheitliche Sprache für Implementierung und Analyse*
  Voop mußte nicht zwangsläufig in derselben Sprache implementiert werden, in der auch die analysierten Programme geschrieben wurden. Für beide Zwecke ist jedoch eine sehr gute Kenntnis der entsprechenden Programmiersprache notwendig. Deshalb hielten wir es für unbedingt sinnvoll, uns auf eine einzige Sprache zu konzentrieren.
- *Vorhandene C++-Entwicklungswerkzeuge*
  Für C++ standen uns gute professionelle Tools zur Verfügung, andererseits konnten wir aus der „public domain“ die frei verfügbare Gnu-Software (Compiler, Debugger und weitere Werkzeuge) samt der zugehörigen Sourcen nutzen.
- *Nähe zu Unix*
  Die Verwendung von C++ erlaubte uns, die in C vorliegenden Standard-Bibliotheken des Betriebssystems Unix zu nutzen; der Gegenstand des Projektes führte zwangsläufig zu Problemen, die nur durch eine Programmierung nahe am Betriebssystem zu lösen waren: Kapitel 7 wird darauf näher eingehen.

Die Programmiersprache C++ unterscheidet sich von anderen, „reinen“ objektorientierten Sprachen wie Eiffel oder Smalltalk grundsätzlich dadurch, daß sie (nahezu) Source-Code-kompatibel zu einer nicht-objektorientierten Sprache, nämlich C, ist. Damit kommt es zu einem hybriden Ansatz, der die Vermischung von objektorientiertem und von konventionellem Code zuläßt. Eine saubere Programmierung im Sinne des objektorientierten Paradigmas ist im wesentlichen möglich und wird immer wieder gefordert. Demgegenüber steht aber die häufige Verwendung bestehenden C-Codes, nicht zuletzt auch die lange C-Vergangenheit vieler C++-Programmierer. Als Konsequenz müssen wir da-

von ausgehen, daß jedes C++-Programm über Funktionen verfügt, die nicht an Objekte bzw. an Klassen gebunden sind.
Von den unter 6.1 genannten Merkmalen objektorientierter Sprachen findet man in C++ beinahe alle Kennzeichen wieder. Die Ausnahmen bilden Metaklassen, Delegation und Ko/Kontravarianz. Einige Bezeichnungen weichen in C++ vom üblichen Sprachgebrauch der Objektorientierung ab[10]: Statt von Ober- und Unterklassen spricht man hier auch von Basis- bzw. von abgeleiteten Klassen. Methoden sind als *Element-Funktionen* (*member functions*) bekannt, Attribute als (*Daten-) Elemente*. *Konstruktoren* und *Destruktoren* dienen als spezielle Element-Funktionen der Initialisierung bzw. dem sauberen Löschen von Objekten mit ihren Abhängigkeiten. Anstelle von Nachrichten werden *Aufrufe* von Element-Funktionen benutzt. Im folgenden werden wir die vertrauten Begriffe von Methoden und Attributen als Synonyme für Element-Funktionen und Daten-Elemente verwenden.

6.3. Analyse von C++-Programmen

Die Entwicklung eines Werkzeuges wie Voop erfordert ein umfassendes Verständnis davon, was zur Laufzeit eines C++-Programms vorgeht. Dazu reicht es nicht aus, die Abstraktionsebene des objektorientierten Entwurfs zu beherrschen. Auf einer sehr viel niedrigeren Stufe der Abstraktion müssen auch die technischen Umsetzungen der Sprache bekannt sein. Der folgende Abschnitt befaßt sich mit den wichtigsten Zusammenhängen, die für Voop von Bedeutung sind. Teilweise handelt es sich um Aspekte, die schon zur Sprachdefinition von C++ gehören, andere entstehen erst bei der Implementierung von C++-Compilern durch technische Erfordernisse.

6.3.1. Kontruktoren und Destruktoren

Konstruktoren dienen zur Initialisierung von Objekten einer Klasse. Durch Überladen ist es möglich, zu einer Klasse mehrere Konstruktoren zu implementieren, die sich durch ihre Signatur unterscheiden. In Destruktoren lassen sich Anweisungen ausführen, die für das saubere Entfernen eines Objektes erforderlich sind. Die explizite Definition von Konstruktoren und Destruktoren ist nicht zwingend erforderlich; verzichtet man darauf, erzeugt der Compiler automatisch einen Standard-Konstruktor und einen Standard-Destruktor (die beide keine Aufrufparameter besitzen) sowie einen Copy-Konstruktor[11].
Wir übernehmen den Begriff Struktor von Stroustrup [Str92] als Bezeichnung für einen Konstruktor oder Destruktor.
Bevor die Anweisungen im Konstruktor einer Klasse ausgeführt werden, erfolgt der Aufruf der Basisklassen-Konstruktoren. Umgekehrt werden die Destruktoren der Basisklassen aufgerufen, nachdem der Rumpf der abgeleiteten Klasse ausgeführt wurde. In welcher Reihenfolge die Aufrufe bei Objekten mit mehreren Basisklassen erfolgen, wird im Abschnitt 6.3.4. erläutert.
Eine Besonderheit ergibt sich, wenn Konstruktoren oder Destruktoren *inline* programmiert werden. Eine Inline-Methode wird in C++ durch die Spezifikation „inline“ explizit gekennzeichnet oder dadurch erreicht, daß die Methodendefinition innerhalb der Klassendeklaration erfolgt. Wird im Source-Code eine solche Inline-Methode aufgerufen, fügt der Compiler an der entsprechenden Stelle des Object-Codes nicht den Aufruf der Methode, sondern deren (compilierten) Code direkt ein. Dieses Sprachkonstrukt eignet sich besonders für häufig verwendete kurze Methoden (z.B. Operatoren); Ziel ist es, den Aufwand für Parameterübergabe, Sichern der Register auf dem Stack etc. zu vermeiden. Dies ist eines der Sprachkonstrukte, das den Prinzipien der Objektorientierung widerspricht, da durch inline-Methoden die Möglichkeit zur dynamischen Bindung verloren geht. Auch Struktoren können inline programmiert werden[12]. Daraus folgt, daß es im Object-Code des Programms nicht mehr eine einzige Adresse gibt, an der sich ein bestimmter Konstruktor befindet, sondern sehr viele identische Codefragmente, die alle denselben Konstruktor verkörpern. Wenn das Laufzeitverhalten des Programms beobachten werden soll, indem die Aufrufe von Konstruktoren registriert werden, muß also unter Umständen eine ganze Anzahl von Programmadressen überwacht werden, obwohl es auf einer höheren Ebene nur einen einzigen Konstruktor gibt.

[10] Die Begriffe geben wir hier gemäß der deutschen Übersetzung von [Str92] wieder.
[11] Die Erzeugung dieser impliziten Kon- und Destruktoren ist compilerabhängig.
[12] Dies gilt insbes. auch für automatisch vom Compiler erzeugte Kon- und Destruktoren. Der Compiler gcc legt diese inline an, soweit das möglich ist (nicht möglich z.B. bei Hierarchien mit virtuellen Methoden, wo Aufrufe über die Methodentabelle erfolgen müssen).

6.3.2. Lebensdauer von Objekten

Betrachtet man das Verhalten von Objekten während der Programmausführung, steht an erster Stelle die Frage nach der Lebensdauer der einzelnen Objekte. Es kommt natürlich vor, daß ein bestimmtes Objekt beim Start des Programmes erzeugt wird und bis zum Ende erhalten bleibt[13]. Dieser Fall stellt jedoch eine Ausnahme dar. Die Fälle, in denen die Lebensdauer sehr gering ist, überwiegen deutlich. Deswegen geht es zunächst darum, die Erzeugung und das Löschen von Objekten zu betrachten, wofür in C++ Konstruktoren und Destruktoren benutzt werden.

Eine grundsätzliche Unterscheidung wird zwischen statischen und dynamischen Objekten[14] getroffen [Str92]:

- Für ein statisches Objekt wird eine Variable als Instanz einer Klasse deklariert, z.B. durch

  ```
  MyClass AnObject;
  ```

  Die Lebensdauer des Objektes entspricht dessen Gültigkeitsbereich im Programmtext, d.h. für ein Objekt, das lokal in einem Block deklariert wird, erfolgt der Konstruktoraufruf beim Eintritt in den Block. Entsprechend wird der Destruktor aufgerufen, sobald die Ausführung das Ende des Blokkes erreicht. Diese Aufrufe werden implizit durch den Compiler erzeugt.
- Um ein dynamisches Objekt zu erzeugen, wird zunächst ein Pointer deklariert:

  ```
  MyClass * pAnObject;
  ```

  Die Deklaration legt den statischen Typ fest und führt lediglich dazu, daß es einen Zeiger auf Objekte dieser Klasse gibt – das Objekt selbst muß erst durch ein explizites *new* erzeugt werden:

  ```
  pAnObject = new MyDerivedClass;
  ```

  Der auf das *new* folgende Bezeichner gibt dabei den dynamischen Typ des Objektes an. Durch *new* wird einerseits der notwendige Speicher auf dem Heap zur Verfügung gestellt (alloziert), andererseits ein Konstruktor der Klasse aufgerufen. Rückgabewert von *new* ist die Heap-Adresse, an der das neu erzeugte Objekt liegt.
  Das Beseitigen eines Objektes geschieht, indem *delete* für den Pointer aufgerufen wird. Dadurch wird der Destruktor der Klasse ausgeführt und anschließend der Speicher, den das Objekt belegt hat, wieder freigegeben (dealloziert). Wenn ein *delete* für denselben Zeiger mehrfach erfolgt, kommt es in der Regel zu unkontrollierten Fehlern. Ein wiederholtes *new* mit Zuweisung auf dieselbe Variable ist hingegen unkritisch, so lange ein Referenz auf das vorige Objekt erhalten bleibt.

6.3.3. Objekt-Identität über this-Pointer

Die Identifizierung von Objekten geschieht in C++ mittels des *this-Pointers*. Dieser Zeiger bezeichnet die Speicheradresse, an der ein Objekt auf dem Heap oder Stack angelegt wurde. Bei statisch deklarierten Objekten kann der Compiler den this-Pointer als feste Adresse (oder als festen Offset zum Stack-Pointer) in den Code einfügen. Bei dynamisch erzeugten Objekten, deren Speicherbereich von *new* alloziert wird, erhält man den this-Pointer als Rückgabewert des *new*-Aufrufs.
Der this-Pointer wird bei allen Methoden implizit als erster Parameter übergeben, damit der Methode bekannt ist, für welches Objekt sie aufgerufen wird. Ausnahmen sind die (als *static* deklarierten) Klassenmethoden.
Die Zuordnung von Objekten und this-Pointern ist weder in der einen noch in der anderen Richtung eindeutig: Mehrfachvererbung führt dazu, daß auf ein bestimmtes Objekt unter verschiedenen this-Pointern zugegriffen wird. Andererseits kann Aggregation bewirken, daß verschiedene Objekte an derselben Speicheradresse liegen und dadurch denselben this-Pointer besitzen. Beide Fälle werden im folgenden Abschnitt verdeutlicht.

[13] Persistente Objekte werden hier nicht berücksichtigt, da sie nicht zum Sprachumfang von C++ zählen.
[14] Exakter wäre es, von statisch bzw. dynamisch deklarierten Objekten zu sprechen.

6.3.4. Vererbung in C++

Jede C++-Klasse kann von mehreren Basisklassen direkt erben (dies gilt nicht für alle objektorientierten Sprachen, vgl. z.B. Oberon). Während eine Sprache wie Eiffel ermöglicht, gezielt bestimmte Features von mehreren Oberklassen zu erben, gilt in C++ eher ein „alles oder nichts“. Neben der Zugriffskontrolle (d.h. ob ererbte Element *public*, *protected* oder *private* sein sollen) bleibt lediglich die Wahlmöglichkeit, eine Klasse virtuell oder nichtvirtuell zu beerben. Die Zugriffskontrolle wirkt sich nur auf die Compilierung aus und ist zur Laufzeit völlig irrelevant; virtuelle Basisklassen beeinflussen das Laufzeitverhalten entscheidend.
Die Eigenschaften von abgeleiteten Klassen in C++ werden im folgenden anhand dreier Stufen erklärt:

- Einfache Vererbung,
- mehrfache Vererbung ohne virtuelle Basisklassen und
- mehrfache Vererbung mit virtuellen Basisklassen.

Die Arten der Vererbung spiegeln sich im Speicherlayout der Objekte wieder, d.h. in der Anordnung von eigenen oder ererbten Datenelementen und Zeigern auf virtuelle Methoden-Tabellen. Für die Programmierung von Voop ist es entscheidend, das Speicherlayout zu kennen. Verbindliche Vorgaben gibt es dafür in C++ jedoch nicht. Das Layout, an das wir uns halten, entspricht der Konvention, die in [EIS90] als „normales Layout“ bezeichnet wird und die wir auch beim GNU Compiler fanden.

(1) Einfache Vererbung
Im einfachsten Fall erbt eine Klasse A von genau einer Klasse B (Abbildung 6.1 (a)). Damit gilt die „is a“-Beziehung: Jedes Objekt von A ist auch ein Objekt der Klasse B und kann sowohl über Zeiger auf A als auch über Zeiger auf B referenziert werden. In beiden Fällen haben die Zeiger denselben Wert, nämlich den des this-Pointers des Objektes der Klasse A. Im Speicher findet man entsprechend an derselben Adresse sowohl ein Objekt der Klasse A als auch der Klasse B: An dieser Adresse liegt zunächst ein vollständiges Objekt der Basisklasse B. Daran schließen sich die Datenelemente an, die in der Klasse A Erweiterungen gegenüber der Klasse B darstellen (vgl. Abbildung 6.1 (b)).

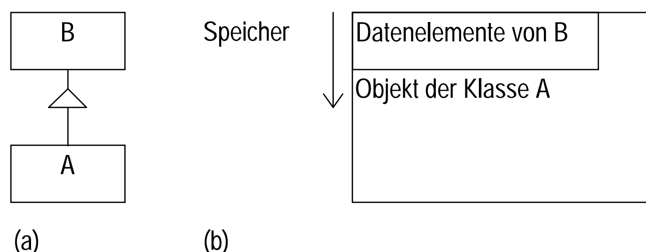


Abbildung 6.1

Wie oben bereits angesprochen wurde, werden die Konstruktoren der Basisklassen vor dem Konstruktor der abgeleiteten Klasse ausgeführt. Bei Erzeugung eines Objektes der Klasse A wird also erst der Konstruktor von B, danach der von A abgearbeitet. Werden jedoch die Einsprungadressen der Konstruktoren betrachtet, so kann die umgekehrte Reihenfolge festgestellt werden: Der Grund dafür liegt in der technischen Realisation der Aufrufreihenfolge. Der Programmzähler erreicht den Konstruktor von A, führt dort einen Prolog aus, der u.a. den Aufruf des B-Konstruktors beinhaltet, so daß als nächstes die Einstiegsadresse des Konstruktors von B erreicht wird. Darauf folgt der eigentliche Rumpf des Konstruktors von B, dann der Rücksprung zu A und die Ausführung der Anweisungen im A-Konstruktor.
Bei der Destruktion entsprechen die Reihenfolgen von Erreichen und Ausführen einander. Erst wird der Destruktor der abgeleiteten Klasse erreicht und ausgeführt, dann der Basisklassenkonstruktor aufgerufen und abgearbeitet.

Exkurs: Einbettung eines Objektes vs. Ableitung

Das folgende Beispiel soll zeigen, daß die Kenntnis des Objektlayouts ebenso wenig wie die Aufrufreihenfolge der Konstruktoren Rückschlüsse auf den zugrunde liegenden Assoziationstypen zuläßt. Wir stellen dazu zwei Fälle der Schemata „A erbt von B“ und „A enthält B“ gegenüber.
Für zwei Klassen Rechteck und Quader könnte es beispielsweise folgende Realisierungen geben:

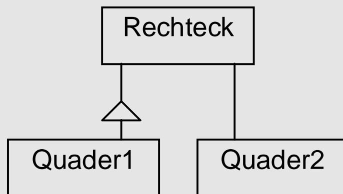
```
class Rechteck {
    public:
    int x,y; // Seitenlaengen
    Rechteck(void) {x=0; y=0;};
    ~Rechteck(void) {};
    int GibFlaeche(void) {return x*y};
    ...
};

class Quader1: public Rechteck {
    public:
    int z; // Hoehe
    Quader1(void) {z=0;};
    ~Quader(void){};
    int GibVolumen(void) {return GibFlaeche()*z;};
    ...
};
```

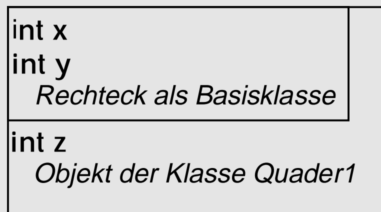
oder

```
class Quader2 {
    Rechteck Grundflaeche;
    int z; // Hoehe
    Quader2(void) {z=0;};
    ~Quader2(void){};
    int GibVolumen(void)
            {return Grundflaeche.GibFlaeche()*z;};
    ...
};
```

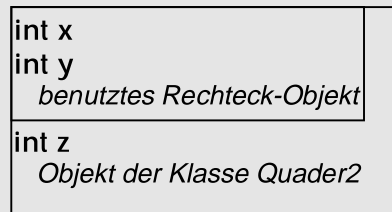
Die Anordnung von Objekten der Klassen Quader1 und Quader2 im Speicher ist identisch, jedes beginnt mit einem Rechteck-Objekt (Abbildung 6.2).



(a) Klassen Rechteck, Quader1,Quader2



(b) Speicherlayout von Quader1



(c) Speicherlayout von Quader2

Abbildung 6.2

Auch die Reihenfolge, in der die Konstruktoren aufgerufen werden, gibt keinen Aufschluß über den Typ der Assoziation. In beiden Fällen wird bei der Konstruktion eines Quaders der Quader-Konstruktor vor dem Rechteck-Konstruktor erreicht. Dieser erhält als impliziten Parameter den this-Pointer des Rechteck-Objekts, welcher mit dem this-Pointer der Instanziierung von Quader1 bzw. von Quader2 übereinstimmt. Die Ausführung erfolgt in beiden Fällen in der umgekehrten Reihenfolge, also Rechteck *vor* Quader.

Anhand dieses Verhaltens ist offensichtlich keine Unterscheidung möglich, ob eine Aggregation oder eine Spezialisierung vorliegt. Das Programm Voop sollte jedoch in der Lage sein, diesen Unterschied festzustellen und den Benutzer im ersten Fall über *ein* erzeugtes Objekt, im zweiten Fall über *zwei* neue Objekte zu informieren.

Daher muß Voop sich aus dem Source-Code oder aus Debug-Symbolen das Wissen verschaffen, welche Beziehung zwischen verschiedenen Klassen besteht.

(2) Mehrfache Vererbung ohne virtuelle Basisklassen

Mehrfachvererbung geschieht in C++, indem bei der Klassendefinition die direkten Basisklassen aufgezählt werden:

```
class A: public B, C, protected D {...};
```

Durch Mehrfachvererbung erhält eine abgeleitete Klasse zu mehreren, i.a. nicht verwandten Klassen eine „is a“-Beziehung. Ein Objekt der abgeleiteten Klasse kann über Zeiger auf die verschiedenen Basisklassen referenziert werden. Wegen der Typprüfung des Compilers können bei solchen Zugriffen nur die Methoden und Attribute verwendet werden, die für den statischen Typ der Zeigervariable zulässig sind. Je nachdem, als welche der Basisklassen das Objekt benutzt wird, besitzt dasselbe Objekte von Fall zu Fall unterschiedliche this-Pointer. Dieses Verhalten erklärt sich aus dem Layout im Speicher. Wie bei der Einfachvererbung beginnt der von einem Objekt belegt Speicherbereich mit dem Datenbereich des ersten Basisklassenobjektes. Danach folgen dann entsprechend die Abschnitte für alle weiteren direkten Basisklassen (siehe Abbildung 6.3). Die möglichen this-Pointer für das Objekt zeigen jeweils auf den Beginn eines dieser Bereiche.
Daß die this-Pointer der Basisklassen-Objekte unterschiedlich sind, entsteht nicht zwangsläufig dadurch, daß zu jeder Basisklasse ein eigener Datenblock gespeichert wird. Wenn eine Basisklasse keine Daten enthält und keine Speicherung eines Pointers auf die virtuelle Methoden-Tabelle nötig ist, könnte theoretisch der zugehörige this-Pointer mit dem Pointer für die folgende Basisklasse zusammenfallen. Der Compiler fügt jedoch an dieser Stelle ein Byte ein, so daß sich unterscheidbare Adressen für den Zugriff auf das Objekt ergeben[15].

[15] Dieses Auftreten verschiedener this-Pointer muß von Voop unbedingt beherrscht werden. Wenn ein Objekt erzeugt wird, soll Voop dieses Objekt erfassen und mit seinem this-Pointer (dem „ersten“, der auf die am meisten abgeleitete Klasse zeigt) speichern. Bei späteren Zugriffen auf dieses Objekt muß das Objekt auch dann richtig identifiziert werden, wenn es nicht mit seinem eigentlichen, gespeicherten this-Pointer auftritt, sondern mit einem etwas höher liegenden, der auf ein eingebettetes Basisklassen-Objekt zeigt.

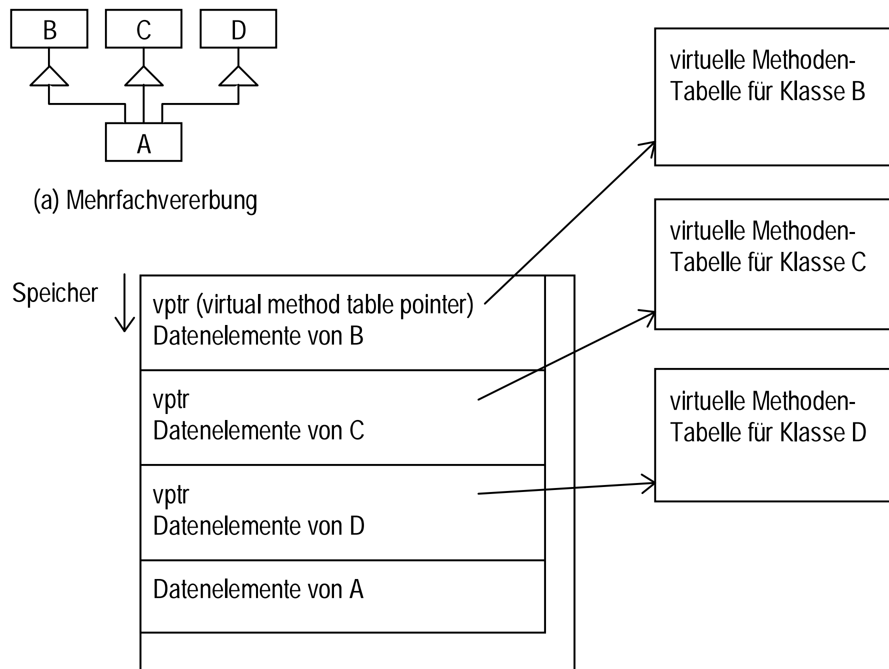


Abbildung 6.3

Bei der Erzeugung werden alle Konstruktoren der Basisklassen durchlaufen. Der Durchlauf erfolgt analog einer Tiefensuche im Vererbungsgraphen von links nach rechts, d.h. in der Reihenfolge, in der die direkten Basisklassen angegeben wurden, werden deren Konstruktorfolgen nacheinander abgearbeitet[16]. Die Ausführung der Destruktoren findet in umgekehrter Reihenfolge statt.

Wenn eine Klasse durch mehrere unterschiedliche Vererbungsbeziehungen als Basis einer abgeleiteten Klasse auftritt (also wiederholt geerbt wird), findet sich diese Klasse auch mehrfach im Speicherlayout wieder. Dementsprechend werden bei der Erzeugung und bei der Destruktion des abgeleiteten Objektes auch mehrmals die gleichen Kon- bzw. Destruktoren aufgerufen.

(3) Mehrfache Vererbung mit virtuellen Basisklassen
Ein anderes Objekt-Layout ergibt sich durch Verwendung des Schlüsselwortes *virtual*: Für jedes Vorkommen der gleichen Basisklasse mit einer virtual Deklaration, wird ein gemeinsames Sub-Objekt benutzt. Bei der Konstruktion ist für dieses Sub-Objekt natürlich nur noch *ein* Konstruktoraufruf erforderlich, entsprechendes gilt für die Destruktion. Virtuelle Basisklassen werden vor den nicht-virtuellen erzeugt: Die Ausführung der Konstruktoren geschieht zunächst für alle virtuellen Basisklassen in der oben beschriebenen Reihenfolge, danach für alle restlichen Basisklassen nach demselben Schema. Bei der Destruktion wird in umgekehrter Reihenfolge vorgegangen.
Während bei nicht-virtuellen Basisklassen mehrere Sub-Objekte der gleichen Basisklasse im Speicher angelegt werden, treten bei virtueller Vererbung Pointer auf das gemeinsame Sub-Objekt an deren Stelle.

[16] cf. [Str92] R.12.6.2

6.3.5. Templates

Das Konzept von generischen Klassen wird in C++ durch Templates verwirklicht. Im Source-Code findet sich z.B. für einen Stack eine Klassendefinition

```
template<class Elem> class Stack {
    public:
            Stack(void);
            ~Stack(void);
            void Push(Elem NewElem);
            Elem Pop(void);
    protected:
            LinkedList<Elem> MyElementList;
};
```

Im Object-Code trifft man nicht mehr auf das abstrakte Klassen-Template, sondern nur noch auf eine Anzahl von konkreten Realisierungen, den Template-Klassen: Die beiden Deklarationen

```
Stack<int> MyIntegerStack;
Stack<tString> MyStringStack;
```

führen dazu, daß der Compiler zwei Klassen *Stack<int>* und *Stack<tString>* erzeugt. Eine allgemeine Klasse wie *Stack* oder *Stack<Elem>* findet sich im Object-Code nicht wieder, vielmehr besitzt jede Realisierung desselben Klassen-Templates ihre eigenen Methoden[17].

6.3.6. Arrays

Durch ein Array werden mehrere, im Speicher direkt aufeinanderfolgende Objekte mit identischer Struktur und identischer Lebensdauer[18] erzeugt. Die einzelnen Objekte unterscheiden sich weder für statische noch für dynamische Arrays prinzipiell von anderen statischen bzw. dynamischen Objekten. Es stellt sich jedoch die Frage, ob es ausreichend ist, die Elemente eines Arrays lediglich als beliebige unabhängige Objekte zu betrachten – in vielen Fällen scheint die Behandlung eines Array als Ganzes sinnvoller. Wenn beispielsweise bei der Konstruktion eines Arrays Hunderte gleichartiger Objekte erzeugt werden, ist es für den Benutzer eines Programms wie Voop vermutlich wenig hilfreich, wenn er nacheinander über die Erzeugung jedes einzelnen Objekts informiert wird; *eine* Meldung über die erfolgte Konstruktion des Arrays dürfte sehr viel sinnvoller sein. Dennoch haben die Elemente fortan eine eigene Existenz und müssen als einzelne und unterscheidbare Objekte behandelt werden[19].

[17] Es ist also nicht möglich, mit *einem* Breakpoint auf dem Konstruktor des Klassen-Templates die Erzeugung aller zugehörigen Template-Klassen zu registrieren.

[18] Ein Array von Verweisen auf Objekte ermöglicht es natürlich einzelne Objekte zu allozieren und zu deallozieren.

[19] In den Debug-Informationen des Objektcodes findet man für ein Array natürlich nur ein Symbol, d.h. das Array wird als Ganzes behandelt. Ein Array von Objekten einer Klasse ist somit ein eigener Typ, aber keine eigene Klasse. Während bei Templates für jede generierte Klasse Konstruktoren etc. vorhanden sind, durch die die Templateklasse eindeutig bestimmt werden kann, verwendet der Compiler bei Arrays genau die Methoden der Klasse, über der das Array deklariert wurde. Wie also soll man bei einem dynamisch erzeugten Objekt feststellen, ob es Teil eines Arrays ist? Alleine die Beobachtung, daß eventuell direkt zuvor eine größere Anzahl von Objekten derselben Klasse entstanden ist, gibt keine Sicherheit. Man müßte also den Source-Code an der Stelle kennen, von der aus die Erzeugung erfolgte, oder aber Interna heranziehen, indem man etwa feststellt, daß ein *new*-Operator ein Vielfaches des Speichers alloziert, den ein Objekt der betreffenden Klasse benötigt.

7. Voop - status quo

7.1. Funktionalität und Oberfläche des Prototypen

In dem Werkzeug, wie es bisher vorhanden ist, sind einige der Möglichkeiten prototypisch implementiert.

Behandelt werden Executables, die mit dem Gnu-C++-Compiler gcc ab Version 2.6.3 unter dem Betriebssystem SunOS 4.x übersetzt sind.
Eine Version des Gnu-Debuggers gdb ab 4.11 wird benötigt.

Der mit Debug-Informationen versehene Object-Code des Programms wird gelesen und analysiert. Danach kann das Programm unter Kontrolle von Voop gestartet werden. Eine Unterbrechung ist jederzeit möglich.
Das Programm stellt sich dabei für den Benutzer unverändert dar und kann wie gewohnt bedient werden, wobei es von Voop überwacht wird.
So wird die Erzeugung und das Löschen von Objekten registriert, wobei eine Einschränkung auf bestimmte Klassen erfolgen kann.

Das Hauptfenster von Voop bietet über Menüs und Werkzeug-Leiste die Möglichkeit, ein Programm zu laden, es zu starten, zu unterbrechen, schrittweise auszuführen und zu beenden sowie die einzelnen Werkzeuge aufzurufen:

Abbildung 7.1 - Das Hauptfenster von Voop

Ein einfaches Werkzeug ist zuständig für die Auswahl von Klassen, die beobachtet werden sollen (Abbildung 7.2). Es arbeitet über das tProgram-Objekt mit den tClass-Objekten und veranlaßt die Beobachtung der gewählten Klassen und die Aktualisierung der angezeigten Objekte. Die Auswahl kann natürlich auch während des Programm-Ablaufs geändert werden.

Abbildung 7.2 - Auswahl der zu beobachtenden Klassen

Weiterhin ist als bisher einzige Diagrammart das Konstruktions-Diagramm als Darstellung der Erzeugungs-Beziehungen zwischen Objekten implementiert (Abbildung 7.3)

Dargestellt werden vorhandene Objekte als abgerundete Rechtecke, mit dem Namen der Klasse (und einer fortlaufenden Nummer). Relationen zwischen Objekten werden als beschriftete Verbindungen zwischen den Rechtecken gezeichnet.

Dabei entsteht eine Baum-Struktur. Ein Knoten mit seinen Kindern stellt ein Objekt mit allen daraus erzeugten Objekten dar.
Die Kanten dazwischen sind mit dem Namen der Methode beschriftet, aus der heraus der Konstruktor-Aufruf erfolgt. Wenn dies eine geerbte Methode ist, wird zusätzlich der Name der entsprechenden Oberklasse genannt.

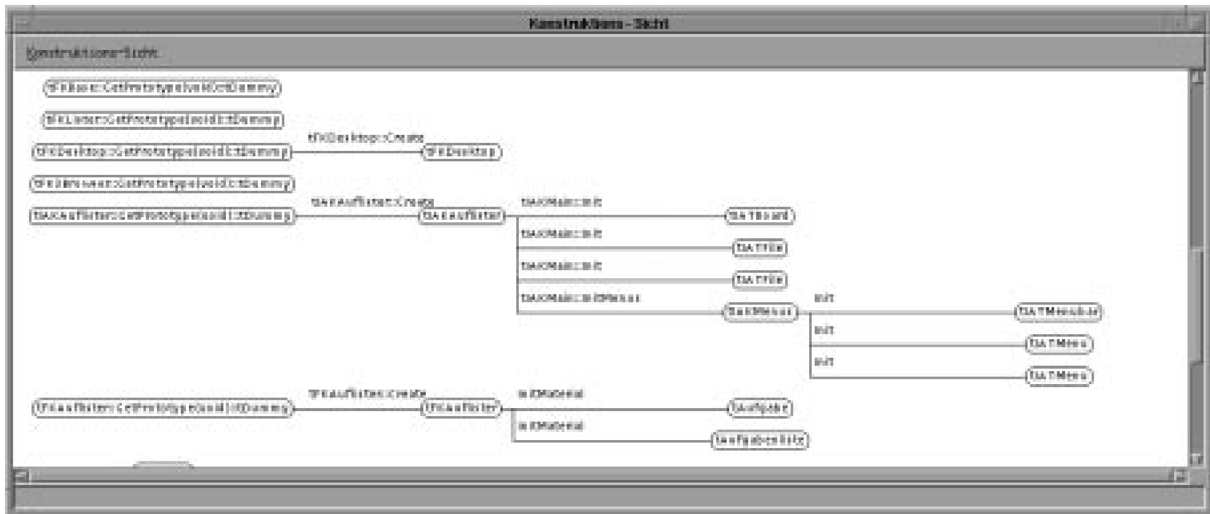


Abbildung 7.3 - Konstruktions-Diagramm

Abbildung 7.3. zeigt den Start einer Applikation, die das FIAK-Framework der Werkzeug und Material-Metapher nutzt. Zum Auflister-Werkzeug wurden Funktions- und Interaktions-Komponente über späte Erzeugung konstruiert. In der Methode InitMaterial der Klasse tFKAuflister wurde je ein Objekt der Klassen tAufgabe und tAufgabenliste konstruiert, wogegen die Objekte der Klassen tIATBoard und tIATFile in der zum Framework gehörenden Klasse tIAKMain erzeugt wurden, von der tIAKAuflister eine Unterklasse ist.

Bei dem momentanen Stand der Implementation ist die Performanz für den praktischen Einsatz deutlich zu niedrig. Nur bei sehr wenigen vorhandenen Objekten wird eine akzeptable Geschwindikeit erreicht.

Für eine sinnvolle Einsetzbarkeit wäre außerdem eine Sicht auf den Source-Code unbedingt nötig. Diese ließe sich ohne größere Änderungen integrieren.

Weitere Darstellungsarten erforderten noch Anpassungen in der Beobachtungs-Komponente von Voop.

7.2. Überblick über die Architektur

7.2.1. Der Rahmen der Anwendung

Unser Werkzeug ist in ein kleines Framework eingebettet, dessen Klassen prinzipiell unabhängig von Voop, jedoch von ihrer Struktur her auf das Einsatzgebiet zugeschnitten sind:
Eine Applikation muß von tApplication erben, und eine Instanz von ihr muß global deklariert werden. In unserem Fall ist tVoop als Unterklasse von tApplication deklariert.
Eine Methode von tApplication fungiert als Handler für Daten auf der Standard-Eingabe. Andere Methoden behandeln Signale des Betriebssystems (SIGINT, SIGQUIT...). Mit dem Aufruf der Methode Init() wird das System gestartet.
Die Klasse tFileHandler dient dazu, Eingaben auf Unix-File-Deskriptoren asynchron zu behandeln. Beliebige Klassen können sich über einen Beobachtermechanismus bei tFileHandler für bestimmte File-Deskriptoren eintragen.
Andere Programme können mit Hilfe der Klasse tProcess gestartet und über ihre Standard-Ein/Ausgabe gesteuert werden.
Ein Standard für Zusicherungen, Vor- und Nachbedingungen ist über die abstrakte Klasse Fehlermeldung und die Konkretisierung FehlerMeldungFuerTty sowie die Makros ENSURE, ASSERT und REQUIRE vorgesehen.
Die am Arbeitsbereich im Rahmen einer Diplomarbeit entstandene Behälter-Bibliothek ConLib wird intensiv genutzt.

7.2.2. Eine Meta-Ebene für das untersuchte Programm

Die Klassen:

Alle interessanten statischen und dynamischen Eigenschaften des zu untersuchenden Programms stellen sich in Voop als Objekte folgender Klassen dar:

statisch:

tClass	Für jede im Programm vorhandene Klasse wird eine Instanz von tClass erzeugt, die Informationen wie den Klassennamen und die Größe und Struktur von Objekten dieser Klasse enthält
tArray	Das Vorhandensein von C-Arrays in C++-Programmen ist einer der vielen Kompromisse, die C++ zu Gunsten der Kompatibilität eingeht. Wir stellen Arrays als Instanz von tArray dar.
tType	Eine abstrakte Oberklasse zu tClass und tArray, die wo möglich eine Gleichbehandlung dieser Klassen erlaubt.
tFunction	Für jede Methode und jede Funktion existiert ein Objekt dieser Klasse.
tAddressInfo	Informationen über Adresse einer Funktion im Object-Code und Definition im Source-Code sind hier gespeichert. Hauptgrund hierfür sind inline-Methoden, die mehrere Instanziierungen haben können.

dynamisch:

tObject	Jedes während des Programmlaufs erzeugte Objekt einer beobachteten Klasse wird auf der Seite von Voop über eine Instanz von tObject dargestellt. Hier werden als wichtigste Elemente die Klassenzugehörigkeit des Objektes durch einen Verweis auf das entsprechende tClass-Objekt sowie der Wert des this-Pointers gespeichert und der Zugriff auf Attribute des Objektes selbst gekapselt.
tCreationRelation	Instanzen von tObject stehen zueinander in Erzeugungs-Beziehungen. Hier wird zu jedem Objekt festhehalten, von welchem Objekt es erzeugt wurde und welche Objekte es erzeugt hat.

Das untersuchte Programm wird vertreten durch eine Instanz der Klasse

tProgram — Hier werden Zugriffsmethoden auf Klassen und Objekte des Programms für den Rest des Systems angeboten.
Über einen Beobachter-Mechanismus können sich Werkzeuge hier für interessierende Ereignisse als Beobachter anmelden.

Diese Klassen bilden damit auf das untersuchte Programm bezogen eine Meta-Ebene, die Informationen über das Programm von einer übergeordneten Ebene aus zugreifbar macht.

Instanziierung dieser Klassen

Weitere Klassen dienen zum Erzeugen dieser Informationen während des Ladens des Programms (statisch) und während seines Ablaufs (dynamisch) sowie zur möglichst effizienten Speicherung und Verwaltung der dabei erzeugten Objekte.

Erzeugen der statischen Informationen:

Die statischen Informationen über das Programm werden aus dem mit Debug-Informationen versehenen Object-Code entnommen. Unter Unix ist das Stabs-Debug-Format am weitesten verbreitet [MKM93]. Die Klasse tStabsExec ist dafür zuständig, aus einem in diesem Format vorliegenden Object-File tClass-, tFunction- und tAddressInfo-Objekte zu erzeugen. (Zur besseren Erweiterbarkeit sollte diese Funktionalität in einer Oberklasse abstrakt beschrieben werden.)
Sämtliche tAddressInfo-Objekte sind in einem Behälter der Klasse tAddressTable gespeichert. Dieser Behälter ist intern nach den Adressen sortiert, so daß die zu einer Adresse gehörende Methode während des Programmablaufs effizient ermittelt werden kann.

Erzeugen dynamischer Informationen während des Programmablaufs:

Während das Programm läuft, muß eine Komponente des Systems für die beobachteten Klassen Instanzen von tObject und tCreationRelation erzeugen und löschen. In Voop ist hierfür ein Objekt der Klasse tObserver zuständig.
Alle erzeugten tObject's werden in einen Container der Klasse tObjectContainer aufgenommen, der über den this-Pointer des dazugehörigen Objekts den Zugriff ermöglicht.
Im Gegensatz zum Erzeugen der statischen Informationen wird hierfür auf einen externen Debugger zurückgegriffen: Die abstrakte Klasse tDebugger fordert hierfür eine Funktionalität, die in tGnuDebugger unter Zuhilfenahme des Debuggers gdb der Free Software Foundation implementiert ist. Die Klasse tGdbProcess als Unterklasse von tProcess sorgt hierfür für das Starten dieses Debuggers in einem eigenen Prozess und die Kommunikation mit dem Debugger.
Prinizipiell erfolgt die Beobachtung durch Setzen von Breakpoints auf Methoden.
Da bisher nur die Beobachtung des Erzeugens und Löschens von Objekten vorgesehen ist, sind vorläufig nur Konstruktoren und Destruktoren von Interesse. In tObserver muß hierbei die teilweise recht komplexe Aufrufreihenfolge für Basisklassen-Struktoren und Struktoren für enthaltene Objekte berücksichtigt werden.

7.2.3. Elemente der graphischen Oberfläche

Die graphische Oberfläche des Werkzeugs ist mit Hilfe von wxWindows implementiert. Diese Bibliothek erleichtert die Programmierung graphischer Benutzungsoberflächen wesentlich und stellt außerdem ansatzweise ein Framework für interaktive Systeme dar. wxWindows ist unter Microsoft Windows und X-Windows weitgehend platformunabhängig einsetzbar.

Beim Start des Werkzeugs wird als erstes ein Objekt der Klasse tCommandPad erzeugt, welches das Hauptfenster der Anwendung darstellt. Es enthält lediglich eine Menüzeile und eine Werkzeugleiste (toolbar).

Nach dem Laden eines Programms kann über das Hauptfenster ein Dialog zur Auswahl der interessierenden Klassen aufgerufen werden, der in den Klassen tClassSelectionFrame und tClassSelectionPanel implementiert ist. Die allgemeine Möglichkeit aus einer Liste eine Menge von Objekten auszuwählen ist bereits in den Oberklassen tSelectionFrame und tSelectionPanel vorgesehen.

Die Klasse tCreationView kapselt die Darstellung der Objekte und ihrer Erzeugungsbeziehungen in einem Fenster.

7.3. Implementation, Diskussion von Alternativen

In diesem Abschnitt soll auf die konkrete Implementation eingegangen werden. Es werden mögliche Alternativen diskutiert und die gewählte Methode begründet.

7.3.1 Beobachtung und Steuerung des untersuchten Programms

Voop soll den Ablauf eines anderen C++-Programms visualisieren. Dazu ist auf einer beliebigen Ebene eine Anbindung an dieses Programm nötig.
Benötigte Informationen lassen sich unterscheiden, in statische, das sind solche, die aus dem Source-Code entnommen werden können (ohne Datenfluß-Analyse o.ä.), und dynamische, die durch geeignete Mechanismen während des Ablaufs des Programms gewonnen werden müssen:

statisch:
- vorhandene Klassen mit ihren Methoden
- Vererbungs-, Aggregations- und mögliche Referenzbeziehungen zwischen Klassen bzw. Objekten

dynamisch:
- vorhandene Objekte
- versendete Nachrichten

Hier gibt es zwei Möglichkeiten, die unterschiedliche Kosten und Nutzen mit sich bringen und damit erheblichen Einfluß auf die Anwendbarkeit und den Anwendungsbereich des Werkzeuges haben:

- die Analyse und evtl. Ergänzung des Source-Codes vor der Übersetzung durch einen Präprozessor.

  Nachteile:

  Erneutes Übersetzen und Binden ist in jedem Fall nötig, wenn dynamische Informationen ermittelt werden sollen.
  Programm-Teile, die nicht in Source-Format vorliegen, können nicht untersucht werden.
  Ohne größeren Aufwand durch zusätzlichen Einsatz einer anderen Methode sind Techniken des traditionellen Debuggings, wie Breakpoints, zeilenweises Abarbeiten des Source-Codes u.a. nicht möglich.
  Ein Standard-C++-Präprozessor und ein C++-Parser muß zur Änderung des Source-Codes benutzt werden.

  Vorteile:

  Der Performanz-Verlust ist sehr gering, da die Bindung an das zu untersuchende Programm durch die direkte Code-Einfügung sehr eng ist.
  Eine Unabhängigkeit vom verwendeten Compiler ist weitgehend gegeben.
  Sämtliche im Programm-Text enthaltenen Informationen können verwendet werden.

- Analyse des mit Debug-Informationen übersetzten Object-Files, Steuerung über Betriebssystem-Funktionen
  Nachteile:
    Abhängigkeit vom verwendeten Compiler besteht.
    Für jedes Debug-Format entsteht Programmier-Aufwand.
    Analyse des Object-Codes beim Laden kostet Zeit.
    Der Ablauf wird stärker verlangsamt, da Eingriffe in das laufende Programm Aufrufe von Betriebssystem-Funktionen erfordern, die zu einem Wechsel des Adreßraums führen.
    Die Semantik der Source-Code-Ebene ist durch den Compiler bereits aufgelöst, d.h. nur Informationen aus dem generierten Debug-Format können benutzt werden.
  Vorteile:
    Das zu untersuchende Programm kann ohne Änderungen sofort benutzt werden. D.h. es ist weder neues Übersetzen noch Binden nötig (solange die interessierenden Teile mit Debug-Informationen kompiliert sind, was während der Entwicklung üblich ist).
    Während des Ablaufs kann die Auswahl der Klassen, Objekte, Methoden, Variablen usw., die beobachtet werden sollen, beliebig verändert werden.
    Die volle Funktionalität eines traditionellen Debuggers ist integrierbar.

Es wäre möglich, beide Arten zu vereinen, um damit auch ihre Vor- und Nachteile geschickt zu kombinieren. So kann z.B. der Source-Code analysiert werden, um sämtliche statischen Informationen direkt auf dieser semantisch höchsten Ebene zu erhalten und dann den laufenden Prozeß über einen Debugger zu beobachten.
Dabei fällt natürlich ersteinmal der Programmier-Aufwand für beide Methoden an, d.h. im wesentlichen der Einsatz eines Parsers mit Präprozessors und das Lesen der Debug-Informationen und der Einsatz eines Debuggers. Dadurch wird dieser Ansatz für den Rahmen unserer praktischen Arbeit schon unrealistisch.
Außerdem treten dabei auch einige Nachteile zusammen auf, so ensteht größerer Zeitbedarf sowohl für die Analyse des Programms, als auch für dessen Ablauf. Eine erneute Übersetzung ist nötig, aber dennoch besteht eine Abhängigkeit vom enstandenen Object-File-Format.
Als wichtig erscheint uns auch die Schwierigkeit, Informationen aus zwei so verschiedenen Ebenen (vor und nach der Transformation durch den Übersetzer) logisch zusammenzubringen und während des Ablaufs konsistent zu halten.

Optimal wäre sicherlich die Integration in eine komplette Entwicklungsumgebung mit eigenem Übersetzer und Debugger (siehe auch Kapitel 4).

Für die praktische Einsetzbarkeit scheint der Ansatz beim Source-Code des Programms weniger geeignet. Die größten Restriktionen sind dabei die Notwendigkeit der erneuten Übersetzung und die mangelnde Flexibilität.
Insbesondere bei etwas größeren Projekten wie z.B der Nutzung dieses Werkzeugs zur Einarbeitung in Frameworks oder Bibliotheken erscheint der Zwang den Source-Code zur Verfügung zu haben verbunden mit dem u.U. sehr großen Zeitbedarf für eine Neuübersetzung als zu einschränkend.
Die Verwendung als objekt-orientierter Debugger ist kaum möglich. Ein Werkzeug, das nur auf dieser Methode aufbaut wäre im wesentlichen ein Profiler auf Objekt-Ebene und hätte damit zu viel an Einsatzmöglichkeiten verloren.

Für Voop haben wir uns deshalb dafür entschieden, beim Object-Code des Programms anzusetzen und einen Debugger zur Ablaufsteuerung zu verwenden.

Debugger

Ursprünglich war vorgesehen, einen vorhandenen Debugger zu benutzen, sowohl um statische Informationen über das Programm zu erhalten als auch zur Steuerung und Beobachtung des Programms.

Die Benutzung von SUN-Workstations mit dem Betriebssystem SUN OS 4.1.x oder Solaris war als Rahmenbedingung vorgegeben.
Für diese Systeme existieren einige Debugger, jedoch verfügt keiner von ihnen über ein API, d.h. eine direkte Programmierschnittstelle. Leistungsfähige Debugger sind hier der gdb der Free Software Foundation und der herstellereigene dbx, die beide textbasiert sind.
Diese Eigenschaft ermöglicht es, diese Debugger - obwohl sie nicht dafür vorgesehen sind - von anderen Programmen aus zu steuern. Dazu wird ausgenutzt, daß unter Unix sämtliche Ein/Ausgabe-

Kanäle einheitlich über sogenannte File-Deskriptoren identifiziert werden. Dadurch ist es möglich, die konkreten Ausprägungen wie Dateien, named und unnamed pipes, sockets und Ähnliches in weitem Umfang gleich zu behandeln.
Der Debugger wird nun als eigener Prozeß gestartet, wobei die Standard-Eingabe, -Ausgabe und -Fehlerausgabe mit dem steuernden Programm verbunden sind.
Folgendes Vorgehen ist dazu nötig:

- Ein/Ausgabe-Kanäle öffnen (pipes, sockets oder ein virtuelles Terminal)
- einen zweiten Prozeß erzeugen (per Systemfunktion fork())
- diesen Prozeß durch den gewünschten Debugger ersetzen, wobei stdin, stdout und stderr auf die vorher geöffneten Kanäle umgelenkt sind.
- Kommandos können jetzt über stdin gesendet werden, auf stdout erscheinen reguläre Ausgaben auf stderr Fehlermeldungen.

Ein Problem stellt sich durch die nötige Synchronisation der beiden parallel laufenden Prozesse. Da auch Interaktion mit dem Benutzer erfolgen soll, ist ein blockierendes Lesen der Debugger-Ausgabekanäle nicht möglich. Der Debugger startet das zu untersuchende Programm als dritten Prozeß. Kommandos auf seiner Standard-Eingabe werden vom Debugger nur verarbeitet, wenn dieser Prozeß nicht läuft, d.h. wegen Erreichen eines Breakpoint oder Auftreten eines Signals gestoppt ist.
Ein Polling ist ebenfalls nicht möglich, gerade da noch andere Prozeße laufen, die dadurch verlangsamt würden.
Solange das steuernde Programm ebenfalls textbasiert ist, d.h. Interaktion nur über die Standard-Ein/Ausgabe stattfindet, ist ein Multiplexen über die Systemfunktion select() möglich, die mehrere File-Deskriptoren auf Daten überwacht.
Dies war während der ersten Phase der Entwicklung von Voop der Fall.
Die einzige auf diesem System zur Verfügung stehende graphische Oberfläche X-Windows bietet eine Funktion, die das Eintreffen von Daten über sogenannte Events signalisiert (XtAppAddInput()).
Nach der Integration einer graphischen Oberfläche mit Hilfe von wxWindows konnte sehr leicht (im wesentlichen durch Austausch von select() gegen XtAppAddInput()) die bisher bestehende Funktionalität erhalten bleiben.

Die Art der Benutzung anderer textbasierter Programme durch Steuerung über Standard-Ein/Ausgabe ist generell anwendbar. In Voop haben wir deshalb die allgemeine Klasse tProcess dafür entworfen.

Aufgrund der Fähigkeiten der Debugger haben wir uns für den gdb entschieden. Dieser unterstützt C++ weitergehender als der dbx, liegt im Source-Code vor, ist auf den meisten Plattformen vorhanden und sollte in der aktuellen Version 4.15 ausgereift sein.

Um vom konkret eingesetzen Debugger unabhängig zu sein, ist die wesentliche Funktionalität in der abstrakten Klasse tDebugger vorgegeben.

Leider hat sich mit fortschreitender Implementierung herausgestellt, daß der vorgesehene Ansatz sämtliche Informationen über den Debugger zu erhalten, nicht durchgehalten werden kann:
Viele der dabei auftretenden Probleme haben ihre Ursache darin, daß gdb nicht dafür vorgesehen ist, von einem anderen Programm gesteuert zu werden.

- So wird sämtliche Information über die Struktur des geladenen Programmes aufbereitet und in Form von Source-Code dargestellt. Dies ist sicher sinnvoll, wenn gdb wie üblich direkt von einem Benutzer eingesetzt wird. Für ein anderes Programm folgt daraus jedoch die Notwendigkeit diesen Source-Code zu parsen.
  Das allein wäre noch recht einfach, da es sich lediglich um Deklarationen handelt.

- Wichtige Informationen sind nicht oder nur sehr umständlich abfragbar, da sie normalerweise für einen Benutzer nicht relevant sind.
  Das betrifft einerseits schon so einfache Dinge, wie alle im Programm vorhandenen Klassen. Es ist hier nur möglich alle definierten Typnamen auflisten zu lassen und jeweils einzelnd dazu die Definition abzufragen, um herauszufinden, ob es sich um eine Klasse handelt. Das alleine bedeutet einen enormen Zeitbedarf.
  Weiter ist es z.B. nicht möglich den Offset eines Objekt-Attributes relativ zum Beginn des Objektes abzufragen. Dies kann frühestens nachdem das erste Objekt einer Klasse erzeugt wurde über einen Vergleich von Adressen ermittelt werden.

- Bei Mehrfachvererbung tritt dieses Problem noch deutlicher auf:

Ein Objekt der Klasse C, die als Unterklasse von A und B definiert ist, kann dynamisch unter dem Typ von A oder dem Typ von B angesprochen werden. Der this-Pointer ist hierbei unterschiedlich, obwohl es sich um das gleiche Objekt handelt.
Daher ist es notwendig zu wissen, mit welchen Offsets auf den this-Pointer des am meisten abgeleiteten Objekts das Objekt noch angesprochen werden kann, d.h. technisch an welchem Offset zu der Adresse des Objekts jeweils die Attribute einer Oberklasse liegen.

Während diese Probleme mit einigem Aufwand prinzipiell zu umgehen sind, sind einige Fehler im Debugger sehr viel schwerwiegender:

- Um Informationen über Objekte und Variable zu bekommen ist es nötig Ausdrücke in C++-Syntax von gdb auswerten zu lassen. Dazu muß i.A. der Gültigkeitsbereich für einen Bezeichner mit angegeben werden (z.B. E::x für ein in der Oberklasse E des aktuellen Objektes deklariertes x). Dies ist in gdb prinzipiell vorgesehen, führt jedoch in einigen Fällen nur zu Fehlermeldungen, in anderen Fällen zu falschen Ergebnissen.

- Vererbung wird von gdb in etlichen Punkten falsch behandelt.
  Ein Attribut x einer Klasse A, das durch ein gleichnamiges Attribut in der Unterklasse B verdeckt wird, sollte unter A::x weiterhin ansprechbar sein.
  Hier wurde grundsätzlich das in B definierte x angezeigt. Nach Einsatz einer neueren Version des gdb, wurde bei der ersten Abfrage das eine und weiteren Abfragen das andere x ausgegeben.

- Bei wiederholtem, nicht-virtuellem Erben von einer Klasse sind damit in jedem Fall die Attribute dieser Klasse nicht mehr zugreifbar.

- Geschachtelte Klassendefinitionen werden teilweise von gdb falsch behandelt. Beim Versuch die Struktur in C++-Form auszugeben kommt es dabei zu Endlosausgaben. Von einem Benutzer ist dies leicht zu erkennen, und die Ausgabe kann auch abgebrochen werden. Für ein Werkzeug ist dieser Fall jedoch kaum vernünftig zu behandeln.
  Dies ist umso schwerwiegender als auch Deklarationen von struct's nach C++-Sprachdefinition als Klassen-Deklaration gewertet werden. Häufig werden solche Deklarationen jedoch einfach genutzt, um interne Daten zu einer Struktur zusammenzufassen, ohne daß dabei eine Klasse im Sinne der Objektorientierung gemeint ist.

Zusammen führten die Probleme lediglich eine Schnittstelle auf C++-Ebene zu haben, viele Informationen nur sehr umständlich zu erhalten und vor allem die mit vertretbarem Aufwand nicht zu umgehenden Fehler in gdb dazu, den Debugger nicht zu benutzen um an Informationen über Programm- und Klassen-Struktur zu gelangen.

Damit entstand also Notwendigkeit den Object-Code direkt zu lesen und die enthaltenen Debug-Informationen zu interpretieren, um auf diesem Weg Informationen über die statische Struktur des Programms zu erhalten.
Damit entsteht natürlich eine Abhängigkeit vom Format des Object-Codes und auch vom Format in dem die Debug-Informationen vorliegen.

Die erste Abhängigkeit wird von der Bibliothek bfd (binary file descriptor) stark gemildert, die es weitgehend unabhängig vom Object-Code-Format erlaubt, Symbolinformationen aus executables zu lesen [Cha91]. Diese Bibliothek stammt ebenfalls aus dem gnu-Projekt der FSF und wird auch von gdb selbst verwendet.

Die Abhängigkeit vom Debug-Format ist jedoch nicht zu umgehen. Für jedes unterstützte Format muß ein spezieller Interpreter programmiert werden. Unter Unix ist das Stabs-Format fast das einzig vorhandene, was diese Abhängigkeit etwas entschärft. Informationen über die Struktur des Programms werden in komprimierter Form in besondere Symbole gestellt, die zusammen mit den anderen in der Symboltabelle des ausführbaren Programms enthalten sind.
Die Klasse tStabsExec ist in unserem Werkzeug für das Lesen und Interpretieren der Symboltabelle zuständig.
Das Stabs-Format ist in [MKM93] dokumentiert. Es wurde ursprünglich für einen bestimmten Pascal-Compiler entworfen und hat sich dann auch in anderen Compilern und Sprachen durchgesetzt. Für C++ wurde es um einige Konstrukte erweitert.

Durch diese Entwicklung gibt es keinen einheitlichen Standard. Verschiedene Compiler weichen hier leicht voneinander ab. Wir haben uns hier auf den Compiler gcc beschränkt, der ebenfalls aus dem gnu-Projekt stammt.
Leider sind auch für diesen Compiler die Abweichungen nur sehr unvollständig dokumentiert.
Dies führte wiederum zu einigem, nicht vorhergesehenen Aufwand: Etliche Testprogramme wurden übersetzt und die vom Compiler erzeugten Debug-Symbole per Hand mit der Dokumentation verglichen. Gerade durch die Komplexität von C++ wurden hier erst im Laufe der Implementation einige Konstellationen deutlich, die in dieser Form in der Dokumentation nicht erwähnt sind.
Zur Unterstützung weiterer Compiler wie z.B. des CC von SUN sind hier deren Eigenheiten zu berücksichtigen. Da die prinzipiell möglichen Abweichungen von der Dokumentation und die undokumentierten Konstellationen bereits für den gcc getestet sind, sollte die Unterstützung weiterer Compiler mit vertretbarem Aufwand möglich sein.

7.3.2. Name-Mangling

Bei Zusammenführen mehrerer compilierter Object-Files zu einem ausführbaren Programm löst der Linker Referenzen einfach nach dem Namen des Symbols auf.
Um typsicheres Binden zu ermöglichen, wird deshalb jeweils die Signatur und teilweise auch der Gültigkeitsbereich von Bezeichnern vom Übersetzer in den Namen codiert. Dieser Codierung wird als name-mangling bezeichnet.
Das Dekodieren (demangling) macht aus so einem Symbolnamen für den Benutzer eines Debuggers, bzw. unseres Werkzeuges wieder lesbare Namen.
Die Art der Codierung ist nicht normiert, jedoch macht Stroustrup in [EIS90] einen Vorschlag, der von den meisten Übersetzern weitgehend eingehalten wird.
Der gcc weicht nur leicht von diesem Verfahren ab. Die Besonderheiten sind in [Sta94] dokumentiert.
Die erwähnte Bibliothek bfd bietet zum Demangling Funktionen an, auf die sich unser Werkzeug stützt.
Ein Fehler des gcc beim mangling von Methoden-Namen einer geschachtelten Klasse trat beim Testen unseres Werkzeuges auf. Für diesen Spezialfall mußten wir in tStabsExec eine Sonderbehandlung vorsehen.

7.3.3. Einsatz des Beobachtermechanismus

Der u.a. in [GHJ95] beschriebene Beobachtermechanismus ('observer-pattern') wird in unserem Werkzeug benutzt, um einzelne Komponenten des Systems zu entkoppeln und Informationen in Form von Ereignissen zu verbreiten.
Dabei meldet sich ein Beobachter (observer) bei einem beobachteten Objekt (notifier) für interessierende Ereignisse an. Durch dieses explizite Bekanntmachen kann die Notifier-Klasse völlig unabhängig vom Beobachter implementiert sein.
In unserem System wird dieser Mechanismus zwischen den folgenden Klassen genutzt:

Notifier:	Observer und Methode, die das Ereignis empfängt:
tFileHandler	tApplication::StdInHandler In tApplication wird die Möglichkeit genutzt, über tFileHandler asynchron von Daten auf der Standard-Eingabe informiert zu werden. tApplication::QuitHandler, ::SignalHandler Vom Betriebssystem stammende Signale werden auf diese Handler umgelenkt, um sauber darauf reagieren zu können.
tFileHandler	tProcess tProcess enthält die Methoden RegisterStdOut und RegisterStdErr, die jedoch prinzipiell nur an tFileHandler delegiert werden.
tGdbProcess	tGnuDebugger::InterpreteOutput tGnuDebugger erbringt die in tDebugger geforderte Funktionalität unter Rückgriff auf den in tGdbProcess gekapselten gdb. Ausgaben vom gdb werden als Ereignis gemeldet.

tObserver	tClass Bei einem tClass-Objekt kann der Wunsch angemeldet werden, Objekte dieser Klasse zu beobachten. Die eigentliche Beobachtung wird an tObserver delegiert.
tObserver	tCreationView::ObserveHandler Damit die Anzeige der Objekte aktualisiert werden kann, wird das Erzeugen und Löschen von Objekten an tCreationView gemeldet.
tDebugger	tCommandPad::UpdateStatus An der Oberfläche wird ein Hinweis ausgegeben, ob das untersuchte Programm gerade läuft, gestoppt oder beendet ist.
tDebugger	tObserver::BreakpointHandler Bei Auftreten eines Breakpoints wird in tObserver untersucht, ob ein Objekt erzeugt oder gelöscht wurde und gegebenenfalls ein Ereignis weitergemeldet.

Weitere Verwendung findet der Beobachtermechanismus in den Klassen der wxWindows-Bibliothek. Als Benutzer dieser Bibliothek sind lediglich die entsprechenden abstrakten (oder leeren) Handler-Methoden zu (re)implementieren.

Implementation:

Leider kann der Beobachtermechanismus in seiner allgemeinen Form in C++ nicht so implementiert werden, daß das zugrundeliegende Entwurfmuster eine direkte Entsprechung in den Sprachkonstrukten findet.
Ein wesentlicher Grund hierfür ist das Verständnis von Pointern auf Member-Funktionen als an die Klasse gebundene Attribute:

```
class a {
        public:
                void print(int x);
        ...
};
```

Die Addresse &a::print der Methode print hat den Typ `void (a::*)(int)`, d.h. ist an die Klasse a gekoppelt. Dadurch eignet sich dieses Konstrukt nicht, um Verweise auf Methoden an allgemein verwendbare Notifier-Klassen zu übergeben.
Hier müßte der C++-Begriff von Addressen eine wesentliche Änderung gegenüber C erfahren, was der Grund ist, warum auf diese allgemeinere Semantik verzichtet wurde [Str92].

Eine Lösungsmöglichkeit liegt in der Einführung von Beobachter-Oberklassen, mit Handler-Methoden, die in den konkreten Beobachterklassen implementiert werden. Soll nicht jedes mögliche Ereignis an die gleiche Methode gemeldet werden, ist es dann jedoch notwendig für jede Notifier-Klasse eine entsprechende Oberklasse zu deklarieren und in einer Beobachterklasse u.U. von mehreren Oberklassen zu erben. Eine einzige allgemeine Beobachterklasse verbietet es vor allem, den Ereignissen spezifische Argumente mitzugegeben. Nur dadurch können jedoch in vielen Fällen sondierende Operationen beim meldenden Objekt (mit zusätzlichem Zeitbedarf) vermieden werden.

Um diese Nachteile zu vermeiden, haben wir einen anderen Weg gewählt: Bei Notifier-Klassen können nur statische Member-Funktionen angemeldet werden. Da diese bis auf die Sichtbarkeit ihrer Deklaration den C-Funktionen entsprechen, entfallen hier die oben genannten Restriktionen. Um nun die nötige Bindung an das zu benachrichtigende Objekt zu erreichen, wird schon bei der Anmeldung der this-Pointer des Objektes mitgegeben und bei Auslösen des Ereignisses als Argument weitergereicht. Über diesen Pointer kann dann in der statischen Member-Funktion die gewünschte Methode dieses Objekts aufgerufen werden.
Bei dieser Vorgehensweise läßt sich durch den Einsatz von Makros der Programmieraufwand gering halten. Jedoch fanden wir, daß der Aufwand in den Notifier-Klassen explizit Listen der zu benachrichtigenden Funktionen und Objekte zu verwalten, das Entwurfmuster weniger verschleiert, als es der Einsatz von Makros tut. Durch den Einsatz der Behälter-Bilbiothek Conlib entsteht dabei wenig zusätzlicher Code.

8. Möglichkeiten der Fortführung

Nachdem wir in dieser Arbeit unsere Ideen, unser Vorgehen und unser erreichtes Ergebnis in Form des Prototypen Voop beschrieben haben, wollen wir abschließend auf die wichtigsten Ansatzpunkte eingehen, an denen das Programm überarbeitet und erweitert werden kann. Wir unterscheiden zwischen der fachlichen Weiterentwicklung, die Leistungsumfang und Erscheinungsbild des Werkzeugs betrifft, und der technischen Verbesserung, die Interna von Voop, aber auch die daraus resultierenden Restriktionen zum Gegenstand hat.

8.1. fachliche Erweiterungen

Auch wenn wir nahezu unbegrenzte Möglichkeiten sehen, die Funktionalität eines Werkzeugs wie Voop zu erweitern, beschränken wir uns hier auf die Aspekte, die in unserer Werkzeugvision wesentlich sind und deren Umsetzung realistisch erscheint.

8.1.1. dargestellte Diagramme

Unser Prototyp bringt das Konstruktions-Diagramm auf den Bildschirm. Dessen Möglichkeiten sind noch nicht ausgereizt, so daß die Weiterarbeit in der Tiefe (nämlich an diesem Diagramm) wie auch in der Breite (das betrifft die anderen Diagramme) nötig ist.

- *Konstruktions-Diagramm*: Das Konstruktions-Diagramm sollte um die in 3.4.1 angesprochenen Features erweitert werden. Konkrete Anregungen sind:
    - Jedes Objektsymbol erhält eine Identität als Graphikobjekt und kann direkt Nachrichten der graphischen Oberfläche erhalten. Mit den Objekten kann dann durch Mausaktionen interagiert werden:
        - Ein Mausklick öffnet ein kontextsensitives Menü. Das Menü ermöglicht, die Werte der Objektattribute und andere objektspezifische Information anzuzeigen. Dazu können auch statistische Daten gehören, vgl. unten. Weitere Optionen erlauben, in einem Editorfenster die Klassendefinition anzuzeigen und die Klasse in einem Hierarchiebaum darzustellen. Die Anzeige der Objekte, die von dem ausgewählten direkt oder indirekt erzeugt wurden, kann unterdrückt werden, um den Graphen auf das Wesentliche zu beschränken.
        - Wenn gleichzeitig mehrere Fenster mit verschiedenen Diagrammen geöffnet sind, wird in allen Fenstern das Symbol des angeklickten Objekts durch entsprechende Farbgebung gekennzeichnet.
        - Das Verschieben von Objektsymbolen mit der Maus ist möglich, damit der Benutzer das Layout des Diagramms nach seinen Bedürfnissen optimieren kann.
    - Bevor gelöschte Objekte vom Bildschirm verschwinden, verändern sie ihre Farbe, um die Destruktion zu signalisieren.
    - Objekte, die durch eine hat-Beziehung in ihrem Erzeuger enthalten sind, bekommen von Anfang an eine eigene Farbe oder Schattierung.
    - Durch eine weitere Farbe wird das Objekt hervorgehoben, das zur Zeit aktiv ist.
- *Referenz-Diagramm*: Das Referenz-Diagramm ist gemäß 3.4.2 zu implementieren. Bedienung und graphische Semantik müssen dem Konstruktions-Diagramm entsprechen.
- *Interaktions-Diagramm*: Bei der Realisierung des Interaktions-Diagramms sollte in Erinnerung bleiben, daß sich dieses Diagramm auch zum Ausdrucken eignet. Das macht aber nur Sinn, wenn die Druckergebnisse in einem handhabbaren Format vorliegen und nicht erst aus vielen Einzelblättern zu einem Graphen zusammengefügt werden müssen. Zusätzlich zu den genannten Eigenschaften der anderen beiden Diagramme sind deswegen Optionen vorzusehen, mit denen die Objekte (hier als Objektlinien) gewählt werden können, die im Druck erscheinen.

8.1.2. Objektbeziehungen

Die Ausführungsgeschwindigkeit des analysierten Programms hängt stark davon ab, wie viele Klassen beobachtet werden. In der Regel wird man daher die Beobachtung auf eine überschaubare Zahl begrenzen. Zwangsläufig verliert man dadurch aber Kenntnis von Objektbeziehungen. Wenn etwa ein Objekt A ein anderes Objekt B erzeugt und das wiederum die Konstruktion von C bewirkt, ist es bisher notwendig, die Klasse von B zu überwachen um die Relation „A erzeugt B erzeugt C" zu registrieren. Andernfalls werden A und C als vermeintlich unabhängig voneinander entstandene Objekte angezeigt. Es wäre sinnvoll, für solche Fälle transitive Relationen zu erfassen und dann für das genannte Beispiel die Beziehung „A erzeugt indirekt C" anzuzeigen.

8.1.3. Protokollieren von Durchläufen

Es ist naheliegend, die in einem Programmdurchlauf aufgezeichneten Ereignisse in einem Protokoll festzuhalten. Das Protokoll kann dann benutzt werden, um die Visualisierung des Durchlaufs beliebig oft zu wiederholen. Weil in den Wiederholungen die zeitaufwendigen Debuggerzugriffe entfallen, wird sich die Animationsgeschwindigkeit beträchtlich erhöhen.

8.1.4. Konfiguration des Werkzeugs

Alle vom Benutzer vorgenommenen Einstellungen müssen gespeichert werden. Zu jedem analysierten Programm sind dazu die für die Beobachtung ausgewählten Klassen und Anzeigeoptionen zu sichern.
Protokollierte Durchläufe können ebenfalls gespeichert werden. Damit wären also die dynamischen Informationen zu einem Programmlauf vorhanden. Wenn zusätzlich die einmal erfaßten statischen Informationen über Aufbau der Klassen etc. persistent gemacht werden, läßt sich die Visualisierung des Ablauf später beliebig wiederholen, ohne daß dazu das Programm selbst notwendig ist.

8.1.5. Einbettung in eine Anwendungsumgebung

Schon jetzt benutzt unser Prototyp mehrere Fenster gleichzeitig. Dieses Konzept ist auf weitere Fenster und Programmsichten zu erweitern. Intern betrifft das die Fenster für noch zu entwickelnde Diagramme. Extern sollte versucht werden, Schnittstellen zu fremden Werkzeugen zu schaffen, z.B. zu Klassenbrowsern und Editoren. Anzustreben ist eine enge Kooperation der Werkzeuge, so daß der Editor beispielsweise den jeweils aktuellen Source-Code zeigt.

8.2. technische Verbesserungen

Wenn der Prototyp Voop in der Praxis eingesetzt werden soll, wird man beim jetzigen Stand unserer Arbeit auf zwei kritische Bereiche stoßen:

- Anforderungen an die Systemsoftware und
- Anforderungen an Rechenzeit.

Die vorrangigen technischen Probleme, die wir haben, betreffen somit Portabilität und Performanz.

8.2.1. Portabilität

Für die Erstellung des Protoypen haben wir das Betriebssystem SunOS 4, den Gnu-Compiler G++ und den Gnu-Debugger gdb als Basis benutzt. Auch wenn es sich dabei an keiner Stelle um exotische, sondern um weit verbreitete Systemsoftware handelt, wäre es wünschenswert, eine Übertragung auf anderer Betriebssysteme und die Benutzung anderer Compiler und Debugger zu ermöglichen.

Ansatzpunkte sind:

- *Betriebssystem*: Die Abhängigkeit vom Betriebssystem SunOS entsteht durch die Funktionen, die wir zur Kommunikation zwischen Voop, Debugger und untersuchtem Programm benutzen (*inter process communication*, IPC). Für andere Unix-Derivate wären hier Anpassungen nötig. Eine Übertragung auf andere Betriebssysteme als Unix erzwingt eine Umstellung auf die dort verfügbaren Mechanismen zur IPC.

  Das bringt auch eine Übertragung auf andere graphische Oberflächen mit sich. Die von uns verwendete Bibliothek wxWindows ist portabel und existiert z.B. auch in einer Version für Microsoft Windows. Die von uns programmierte Klasse tXTTy, die X-Windows-Fenster für Debugger und untersuchtes Programm öffnet, müßte aber neu implementiert werden.

  Das Object-Code-Format der analysierten Programme ist natürlich vom Betriebssystem abhängig. Wir verwenden die GNU-Bibliothek „BFD“, um auf den Object-Code zuzugreifen, und decken damit die Formate ELF, COFF und XCOFF ab. BFD wird von der Klasse tStabsExec benutzt. Eine Portierung auf Systeme mit einem anderen Object-Code-Format hätte also Änderungen in tStabsExec zur Folge.
- *Compiler*: Die Klasse tStabsExec liest die Angaben über statische Strukturen von Programmen aus Debug-Informationen. Die Debug-Informationen müssen im STABS-Format vorliegen, das unter Unix gebräuchlich ist. Die Syntax der STABS-Anweisungen variiert leider etwas zwischen verschiedenen Compilern. Unser Referenz-Compiler ist gcc; die Abweichungen des Compilers Sun CC haben wir berücksichtigt, soweit sie unserer Dokumentation [MKM93] zu entnehmen waren. Andere Compiler, die STABS unterstützen, werden hier zusätzliche Arbeit erfordern. Für abweichende Debug-Formate muß die Klasse tStabsExec als ganzes ersetzt werden.
- *Debugger*: Der benutzte Debugger (derzeit gdb) kann relativ leicht „ausgewechselt“ werden, indem von der abstrakten Klasse tDebugger eine weitere Unterklasse abgeleitet und anstelle von tGnu-Debugger benutzt wird.

8.2.2. Performanz

Die Zeit, die für einen Programmlauf unter Voop benötigt wird, hängt stark von der Anzahl der Klassen ab, die zur Beobachtung ausgewählt wurden. Als Faktoren für den Zeitbedarf sehen wir:

- *Sockets als Inter-Prozeß-Kommunikation*: Voop und Debugger kommunizieren über Sockets miteinander, d.h. jede Anweisung an den Debugger wird zeichenweise von einem Prozeß zu einem anderen geschickt. Wenn man keinen externen Debugger benutzte, sondern etwa eine entsprechende C-Bibliothek, wäre diese ineffiziente Art der Kommunikation unnötig und könnte durch normale Funktionsaufrufe ersetzt werden.
- *Overhead durch Breakpointverwaltung des Debuggers*: Bei der Untersuchung eines Programms von realistischer Größe werden schnell mehrere tausend Breakpoints gleichzeitig gesetzt. Damit nimmt der Verwaltungsaufwand zu und kostet Rechenzeit. Gerade gdb ist nicht auf diese große Anzahl eingestellt und benötigt hier unverhältnismäßig viel Zeit.
- *Analysieren des Aufrufstacks*: Bei jedem erreichten Breakpoint veranlaßt Voop den Debugger, über mehrere Ebenen im Aufrufstack zu wandern, um die Reihenfolge und Argumente der zuletzt aktiven Methoden zu ermitteln. Um diesen Stack zu konstruieren, benötigt gdb auffällig viel Zeit. Schon bei wenigen Ebenen liegt die Antwortzeit im Bereich von Zehntelsekunden.

- *Overhead durch Verwaltung der Objekte in Voop*: Je mehr Breakpoints gesetzt sind, desto höher wird die Zahl der von Voop erfaßten Objekte sein. Damit nimmt auch der Zeitaufwand zu, um bei Erreichen eines Breakpoints nach den zugehörigen Objekt-Informationen zu suchen.

Der größte Teil der Rechenzeit wird nicht in Voop verbraucht, sondern im Debugger oder auf dem Weg dazwischen. Schritte für die Zukunft könnten also sein,

- den Debugger gdb durch einen effizienteren Debugger zu ersetzen,
- nach einer effizienteren Art der Kommunikation zum Debugger zu suchen,
- ganz auf den externen Debugger zu verzichten, indem die Debugger-Funktionalität in C++-Klassen implementiert wird.

Die Zeit, die zum Laden des Programmes verbraucht wird, entsteht durch das Parsen der Debug-Anweisungen in der Klasse tStabsExec. Deren Methoden können verbessert werden, um die Ladezeit zu verkürzen.

8.2.3. Beobachten von Objekt-Verweisen

Voop ist bisher nur darauf ausgelegt, Konstruktor- und Destruktoraufrufe zu überwachen und daraus ein Konstruktions-Diagramm anzufertigen. Um das Referenzierungs-Diagramm zu verwirklichen, ist es notwendig, die Belegung aller Pointer, die auf Klasseninstanzen zeigen, zu verfolgen. Da wir nach wie vor unabhängig vom Source-Code arbeiten wollen, müssen wir davon ausgehen, daß die Adressen von Zuweisungsoperationen auf Pointern unbekannt sind; Breakpoints kommen dafür also nicht in Frage. Was man vielmehr benötigt, sind Watchpoints. Ein Versuch mit Watchpoints des gdb zeigte aber, daß die Programmausführung dadurch unerträglich langsam wird[20]. Hier müßte also noch grundlegend untersucht werden, auf welche Art die Belegung von Pointern sinnvoll überwacht werden kann. Eine Möglichkeit ist, für alle angezeigten Objekte jeweils die enthaltenen Referenzen zu überprüfen (z.B. bei jedem registrierten Methodenaufruf).

[20] Dieser Versuch fand auf einer Sun-Workstation mit SPARC-Prozessor statt. Es gibt andere Rechnerarchitekturen, deren Memory Management Unit (MMU) Watchpoints unterstützt. Nur auf solchen Systemen scheint uns die Verwendung von Watchpoints sinnvoll.

9. Anhang

9.1. Syntax der „Stabs“-Debug-Anweisungen

Um die Typinformationen aus den Debug-Anweisungen zu ermitteln, mußten wir eine Grammatik für die Stabs-Typbeschreibungen entwickeln. In vereinfachter Backus-Naur-Form ist der Aufbau der „LSYM“-Anweisungen angegeben. Die Grammatik basiert auf der Beschreibung in [MKM93] und berücksichtigt die Erweiterungen und Abweichungen, die wir in den vom Compiler gcc erzeugten Anweisungen fanden. Wie erheben keinen Anspruch auf Vollständigkeit und Korrektheit – darin lag unser Ziel nicht, vielmehr ging es uns nur darum, eine Arbeitsgrundlage für unsere Implementierung der Klasse tStabsExec zu erhalten.

```
Stabs               = .stabs "lsymstring",128,0,0,0

lsymstring          = namedef | tagdef
typedef             = classname ':' 't' defnumber
tagdef              = classname ':' ['T'] ['t'] defnumber '=' 's' size [baseclasses]
                      {members} ';' ['~%' defnumber ';']

classname           = identifier
size                = number

baseclasses         = '!' numberofbases ',' basedescr {basedescr}
numberofbases       = number
basedescr           = virtualchar visibilitychar offset ',' defnumber ';'
virtualchar         = '0'|'1'
                      // other characters should be possible, should be ignored by debugger
visibilitychar      = '0'|'1'|'2'
                      // other characters should be possible, should be ignored by debugger

member              = nonstaticmember | staticmember | virtbaseptr | vtableptr | method

nonstaticmember     = membername ':' ['/' mvisibilitychar] defnumber [definition] ','
                      bitoffset ',' bitsize ';'

staticmember        = membername ':' ['/' mvisibilitychar] defnumber ':' staticname ';'

membername          = identifier
mvisibilitychar     = visibilitychar | '9'
definition          = arraydef | ptrdef | refdef
bitoffset           = number
bitsize             = number

staticname          = identifier;

ptrdef              = '=' '*' defnumber
refdef              = '=' '&' defnumber
arraydef            = '=' 'a' rangedescr {[defnumber '='] 'a' rangedescr} defnumber

rangedescr          = 'r' defnumber ';' bound ';' bound ';'
bound               = 'J'
                      | ['A' | 'T' | 'a' | 't'] number

virtbaseptr         = '$vb' defnumber ':' defnumber ',' offset ';'

vtableptr           = '$vf' defnumber ':' defnumber [ptrdef [arraydef]] ',' offset ';'
```

```
method                  = methodname '::' methoddescr {methoddescr} ';'
methoddescr             = methodtype | functiontype

methodtype              = defnumber ['=' '#' methodargs ';'] arguments ';'
                           visibilitychar modifierchar virtualdescr
methodargs              = '#' defnumber [definition]
                          | defnumber [definition] {',' defnumber [definition]}

functiontype            = defnumber '=' 'f' defnumber [definition] arguments ';'
                          visibilitychar modifierchar '?'

arguments               = ':' identifier
                          // the mangled argument list or structor name
modifierchar            = 'A'|'B'|'C'|'D'
virtualdescr            = '.'
                          | '*' vtindex ';' defnumber ';'
vtindex                 = ['-'] number

offset                  = number

defnumber               = number

number                  = digit {digit}
digit      = '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'
```

9.2. Literaturverzeichnis

[Bal94] Thomas Ball: Efficiently Counting Program Events with Support for Online Queries. In: ACM Transactions on Programming Languages and Systems Jg. 16 Nr.5, 1994

[BMW94] Ted J. Biggerstaff, Bharat G. Mitbander, Dallas E. Webster: Program Understanding and the Concept Assignment Problem. In: Communications of the ACM Jg. 37 Nr. 5, 1994

[Boo94] Grady Booch: Object-Oriented Analysis and Design with Applications. 2nd ed. Benjamin/Cummings 1994

[CaT94] Wengtong Cai, Stephen J. Turner: An approach to the Run-Time Monitoring of Parallel Programms. In: The Computer Journal Jg. 37 Nr. 4, 1994

[Cha91] Steve Chamberlain: The Binary File Descriptor Library, Cygnus Support, 1991

[ChG92] G. Cheng, N. A. B. Gray: A Program Visualisation Tool. In TOOLS Pacific 1992

[CuB86] Ward Cunningham, Kent Beck: A Diagram for Object-Oriented Programs. In: Object-Oriented Programming Systems, Languages, and Applications Conference 1986, ACM SigGraph

[ElS90] Margaret A. Ellis, Bjarne Stroustrup: The Annotated C++ Reference Manual. Addison-Wesley 1990

[Flo95] Christiane Floyd: Software Engineering: Kritik und Perspektiven. In: Jürgen Friedrich et. al.: Informatik und Gesellschaft. Spektrum Akademischer Verlag 1995

[GHJ95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: Design Patterns: Elements of Reuseable Object-Oriented Software. Addison-Wesley 1995

[Ing81] D. Ingalls: Design Principles behind Smalltalk. In: Byte Jg. 6 Nr. 8, 1981

[JCJ92] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, Gunnar Overgaard: Object-Oriented Software Engineering – A Use Case Driven Approach. Addison-Wesley 1992

[KGZ93] Klaus Kilberth, Guido Gryczan, Heinz Züllighoven: Objektorientierte Anwendungsentwicklung. 2. Aufl. Vieweg 1993

[KlG88] Michael F. Kleyn, Paul C. Gingrich: Graphtrace - Understanding Object-Oriented Systems Using Concurrently Animated Views. In: Object-Oriented Programming Systems, Languages, and Applications Conference 1988

[McK94] John D. McGregor, Timothy D. Korson: Integrated Object-Oriented Testing and Development Processes. In: Communicatins of the ACM Jg. 37 Nr. 9, 1994

[MKM93] Julia Menapace, Jim Kingdon, David MacKenzie: The „stabs“ Debug Format. Cygnus Support 1993

[Mey88] Betrand Meyer: Object-Oriented Software Construction. Prentice Hall 1988

[Mye83] Brad A. Myers: A System for Displaying Data Structures. In: Proceedings of the ACM SIGGRAPH Computer Graphics ´83 Conference

[Mye86] Brad A. Myers: Visual Programming, Programming by Example, and Program Visualization: A Taxonomy, Conference Proceedings CHI'86

[PHK93] Wim De Pauw, Richard Helm, Doug Kimelman, John Vlissides: Visualizing the Behaviour of Object-Oriented Systems. In: Object-Oriented Programming Systems, Languages, and Applications Conference 1993, ACM SigGraph

[PKV94] Wim De Pauw, Doug Kimelman, John Vlissides: Modeling Object-Oriented Program Execution. In: ECOOP

[RBP91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, William Lorensen: Object-Oriented Modeling and Design. Prentice Hall 1991

[Rum95] James Rumbaugh: OMT: The object model. In: Journal of Object-Oriented Programming Jg. 7 Nr. 8, 1995

[Sch94] Steffen Schäfer: Objektorientierte Entwurfsmethoden. Addison-Wesley 1994

[ShS94] John J. Shilling, John T. Stasko: Using Animation to design object-oriented systems. In: Object Oriented Systems Jg. 1 Nr. 1, 1994

[Sma95] Julian Smart: User Manual for wxWindows 1.63. Edinburgh 1995

[SRMb90] SunOS Reference Manual, Vol.II, 2: System Calls, Sun 1990

[SRMc90] SunOS Reference Manual, Vol.III, 5: File Formats, Sun 1990

[SSO90] Sun System Servives Overview, Sun 1990

[Sta94] Richard Stallmann: The GNU C++ Renovation Project, Free Software Foundation 1994

[STK94] Satish Subramanian, Wie-Tek Tsai, Shekhar H. Kirani: Hierarchical data flow analysis for OO programs. In: Journal of Object-Oriented Programming Jg. 7 Nr. 2, 1994

[StP94] Richard M. Stallmann, Roland H. Pesch: Debugging with GDB. Ed. 4.12, 1994

[Str92] Bjarne Stroustrup: The C++ Programming Language. 2nd ed. Addison-Wesley 1992

[Wes93] Alan West: Animating C++ Programs. Dynamic C++ Animation White Paper, Objective Software Technology Ltd.,1993

[WWW90] Rebecca Wirfs-Brock, Brian Wilkerson, Lauren Wiener: Designing Object-Oriented Software. Prentice Hall 1990