Keno Hamer

Visualisierung von Objektinteraktionen

Diplomarbeit

Fachbereich Informatik

Universität Hamburg

Februar 1998

Keno Hamer

Planckstraße 20
22765 Hamburg
hamer@dsm.chasque.apc.org

Betreuer: Prof. Dr. Heinz Züllighoven

Vogt-Kölln-Straße 30
22527 Hamburg
zuelligh@informatik.uni-hamburg.de

Zweitbetreuer: Dr. Daniel Moldt

Vogt-Kölln-Straße 30
22527 Hamburg
moldt@informatik.uni-hamburg.de

Erklärung gemäß §23 der Diplom-Prüfungsordnung:

Hiermit erkläre ich, daß ich die Arbeit selbständig durchgeführt und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1
Einleitung

Um hochwertige und komplexe Software zu entwickeln, müssen für alle Aufgaben innerhalb der Projektzyklen angemessene Methoden und Werkzeuge zur Verfügung stehen. Für die objektorientierte Programmentwicklung sind dementsprechend Werkzeuge nötig, die das objektorientierte Paradigma gezielt unterstützen. So vielfältig das Angebot an objektorientierten Werkzeugen und Methoden mittlerweile auch ist – in vielen Publikationen wird auf ein Defizit an Werkzeugen hingewiesen, mit denen sich Struktur und Verhalten objektorientierter Systeme begreifbar machen lassen.

Sichtbar wird dieses Defizit besonders vor dem Hintergrund der Wiederverwendung. Die Fortentwicklung der Techniken, die auf dem Paradigma der Objektorientierung basieren, hat die Wiederverwendung von Komponenten und Konzepten in einem vorher nicht dagewesenen Maß ermöglicht. Wiederverwendung kann aber nur dann stattfinden, wenn Softwareentwickler[1] verstehen, wie sich die wiederverwendbaren Komponenten verhalten und wie sie einzusetzen sind. Während es früher nur sehr begrenzte technische Möglichkeiten gab, um wiederverwendbare Elemente zu entwickeln, zeichnet sich inzwischen ab, daß das Verstehen der Elemente zum limitierenden Faktor wird.

Gleichzeitig hat die erste Generation von großen objektorientierten Systemen ein Alter erreicht, in dem typische Probleme der Anpassung und Wartbarkeit auftreten: Die ursprünglichen Entwickler sind nicht mehr verfügbar, Dokumentationen sind unvollständig und inkonsistent, und es treten geänderte Anforderungen in einem Ausmaß auf, das beim Entwurf der Systeme nicht vorgesehen waren. Auch hier liegt ein Bedarf an Werkzeugen vor, mit denen die Systeme analysiert werden können.

Für wiederverwendbare Komponenten und für lauffähige Anwendungen gilt gleichermaßen, daß die *Struktur* der Systeme noch relativ einfach erschlossen werden kann. Sehr viel schwieriger ist es, das *Verhalten* eines Systems zu verstehen und zu beschreiben: In allen objektorientierten Programmiersprachen findet sich die Struktur im Programmcode wieder; das Verhalten aber entsteht erst zur Laufzeit und ist damit für den Softwareentwickler unsichtbar. Der Versuch, das Verhalten eines Systems „sichtbar“ zu machen, wird schnell zum Dilemma: Einerseits werden für Fragen bezüglich des Laufzeitverhaltens veraltete, nicht objektorientierte Werkzeu-

[1] Zugunsten der Lesbarkeit dieser Arbeit beschränke ich mich auf die Verwendung maskuliner Formen, anstatt Schreibweisen wie “Entwicklerinnen und Entwickler” oder “EntwicklerInnen” zu benutzen. Ich bitte um Verständnis.

ge (z.B. Debugger) eingesetzt, aber andererseits erschließt sich gerade erst durch die Dynamik die Leistungsfähigkeit objektorientierter Programmiersprachen und Entwurfskonzepte.

1.1 Ziele der Arbeit

Um das Verhalten von objektorientierten Systemen im Sinne des Wortes sichtbar zu machen, wird an der Universität Hamburg zur Zeit der Prototyp eines Werkzeugs entwickelt, mit dem die zur Laufzeit entstehenden Objekte und deren Dynamik visualisiert werden können. Eine Zielsetzung für das Werkzeug besteht darin, nicht nur die Visualisierung von Objekten zuzulassen, sondern auch objektübergreifende Strukturen und Verhaltensmuster anzuzeigen. Damit kann die Wiederverwendung objektorientierter Komponenten und Konzepte gezielt unterstützt werden.

Langfristig soll das Werkzeug mehrere Arten von Diagrammen anbieten; zu Beginn wird lediglich die Visualisierung in Form von Interaktionsdiagrammen ermöglicht. Ein besonderer Einsatzzweck für das Werkzeug besteht in der Anfertigung von Dokumentationen: Entwicklern von Frameworks soll es ermöglicht werden, Diagramme zu erstellen, die Aspekte des Frameworks illustrieren und den Benutzern des Frameworks dessen Verwendung erleichtern. Benötigt wird daher die Möglichkeit, Visualisierungen nicht nur zu erstellen, sondern sie anschließend auch zu bearbeiten.

Diese Arbeit steht in engem Zusammenhang mit der Entwicklung des Werkzeugs. Sie ist eine von zwei Diplomarbeiten, die die Konzepte des bisher entwickelten Werkzeugs beschreiben. Wie die beiden Arbeiten gegeneinander abzugrenzen sind, werde ich in den späteren Kapiteln erklären. Die wesentlichen Ziele dieser Arbeit sind es,

- Grundlagen der Softwarevisualisierung zu diskutieren,
- einen Überblick über Arbeiten zu diesem Thema zu geben,
- die Architektur des entwickelten Systems vorzustellen und
- die Komponenten des Systems zu erklären.

Neben den allgemeinen Aspekten des Systems wird sich ein Schwerpunkt der Arbeit mit demjenigen Teil des Werkzeugs befassen, der die grafische Darstellung in Gestalt von Interaktionsdiagrammen sowie deren Manipulation ermöglicht.

1.2 Aufbau der Arbeit

An diese Einleitung schließt sich ein einführendes Kapitel an, das sich mit dem grundlegenden Problem, Struktur und Verhalten von Software zu verstehen, auseinandersetzt und den Einsatz visueller Techniken in der Softwareentwicklung behandelt. Danach werden andere Arbeiten vorgestellt, die sich ebenfalls mit Werkzeugen zur objektorientierten Softwarevisualisierung beschäftigen.

Das folgende dritte Kapitel beschreibt Möglichkeiten, in der objektorientierten Softwareentwicklung Wiederverwendung zu nutzen. Auf die Inhalte dieses Kapitels wird im Verlauf der Arbeit mehrfach Bezug genommen.

Im vierten Kapitel, das einen zentralen Teil dieser Arbeit darstellt, werden die Fragestellungen beschrieben, die die Entwicklung des Werkzeugs begründeten, und es werden die Anforderungen an das Werkzeug diskutiert. Basierend auf den Anforderungen wird eine Systemarchitektur vorgestellt. Die Komponenten der Architektur werden in den darauffolgenden Kapiteln näher beschrieben; sie lassen sich grob nach den Aufgaben Informationserfassung, Informationspräsentation und Informationsverwaltung unterscheiden. Letztere ist Inhalt von Kapitel 5. Es beschreibt ein Modell, in dem Struktur und Verhalten von Objekten nachgebildet werden können. Zu dessen Herleitung zählt die Untersuchung von möglichen Objektzuständen und den Übergängen zwischen diesen. Wie die für die Visualisierung benötigten Informationen erfaßt werden, beschreibt Kapitel 6.

Die Präsentation der Informationen behandelt Kapitel 7, das den umfangreichsten Abschnitt dieser Arbeit bildet. Da das Werkzeug eine Visualisierung in Form von Interaktionsdiagrammen leisten soll, wird zunächst eine Notation für erweiterte Interaktionsdiagramme vorgestellt. Um die Diagramme interaktiv bearbeiten zu können, habe ich ein Framework entwickelt, das im selben Kapitel beschrieben wird. Im Anschluß daran folgt der Entwurf des Visualisierungswerkzeugs, und es wird diskutiert, mit welchen Auswirkungen die Visualisierung höherer, objektübergreifender Konzepte verbunden ist.

Im abschließenden achten Kapitel werden die Ergebnisse der Arbeit zusammengefasst, und ein Ausblick wird aufzeigen, welche offenen Fragen und welche Anknüpfungspunkte für weitere Forschungsarbeiten bestehen.

Kapitel 2
Grundlagen

2.1 Programmverstehen

2.1.1 Reverse Engineering

Als 1968 das Fachgebiet des *Software Engineering* aus der Taufe gehoben wurde, um die Qualität von Softwaresystemen zu erhöhen, wies man bereits in der Bezeichnung des neuen Gebietes darauf hin, daß diese Qualitätssteigerung durch Anleihen bei den etablierten Ingenieurswissenschaften erreicht werden sollte. *Engineering* sollte ausdrücken, daß Softwareentwicklung in einem geregelten, formalisierten Herstellungsprozeß stattzufinden habe, an dessen Ende das einsatzbereite Produkt Software steht.

Die Kritik, die am Selbstverständnis des Software Engineering geübt wird, zielt unter anderem auf diese Trennung von Produktion und Einsatz ab und weist außerdem auf die Bedeutung von kreativen und kooperativen Anteilen in der Softwareentwicklung hin [Floyd 95]. Vor dem Hintergrund dieser Kritik ist die Forderung nach einem evolutionären Vorgehen entstanden, bei dem die Softwareentwicklung nicht mit einem fertiggestellten Programm abbricht, sondern statt dessen in einen neuen Entwicklungszyklus übergeht.

Der Ansatz des evolutionären Vorgehens wird der Tatsache gerecht, daß Software auch nach Inbetriebnahme wesentlichen Änderungen unterworfen ist, weil sich die Anforderungen verändern oder weil Fehler und Schwachstellen sichtbar werden. Das herkömmliche Verständnis des Software Engineering nimmt aber einen Einschnitt vor, an dem das System „fertiggestellt“ ist und der Einsatz beginnt, und faßt die weiteren Tätigkeiten unter Wartung zusammen. Die häufig zitierten Zahlen, daß die Wartung soundsoviel Prozent des Aufwandes für ein System ausmacht, weisen darauf hin, daß der Engineering-Begriff die Weiterentwicklung eines Systems nicht zufriedenstellend erfaßt.

Im Zusammenhang mit Wartungsaktivitäten wird in den letzten Jahren mehr und mehr von *Reengineering* und *Reverse Engineering* gesprochen: Die normale Entwicklungsrichtung von Software führt über verschiedene Ebenen von Dokumenten, die zunehmend abstrakter werden; angefangen bei informellen Beschreibungen des Systems bis hin zum vom Compiler übersetzten Programm. Diese Richtung wird als *Forward Engineering* bezeichnet. Beim Reverse Engineering geht es umgekehrt darum, aus abstrakten Dokumenten weniger abstrakte zu extrahieren. Mit Reengineering bezeichnet man die Aktivitäten, die an einem Dokument oder Programm vorgenommen werden, um dessen Verständlichkeit, Wiederverwendbarkeit oder Änderbarkeit zu verbessern. Während unter Wartung üblicherweise nur Veränderungen im

Rahmen der bestehenden Strukturen eines Systems verstanden werden, bezieht sich Reengineering auf diejenigen Prozesse, in denen man diese Strukturen selbst überarbeitet [Doebele 97].

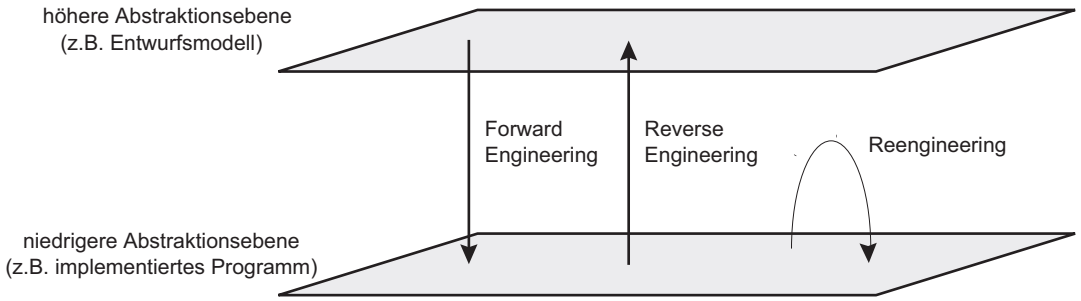


Abb. 1: Forward Engineering, Reverse Engineering und Reeingeneering

Reengineering und Reverse Engineering sind in diesem Sinne keine Begriffe, die als Gegensatz zu Software Engineering auftreten, sondern sind davon unabhängig; sie lassen sich auch auf evolutionäre Entwicklungsprozesse anwenden.

Typischerweise finden Reengineering und Reverse Engineering auf der Basis von Quelltexten statt. Da viele Systeme unzureichend oder sogar überhaupt nicht dokumentiert sind, müssen anhand des Quelltextes Dokumentationen erstellt und Entwurfskonzepte identifiziert werden. Bestimmte Aufgaben können dabei automatisch durch Software bewältigt werden, in den meisten Fällen aber müssen die Systeme von Entwicklern analysiert werden – hier sind *unterstützende* Softwarewerkzeuge gefragt.

2.1.2 Programmverstehen

Immer komplexer werdende Softwaresysteme stellen an Entwickler immer höhere Anforderungen. In der Einsicht, daß Software nur selten intuitiv verständlich ist, wurde das Verstehen von Software von diversen Autoren thematisiert und wird häufig unter dem Begriff *Programmverstehen* beschrieben: „Program understanding is the (ill-defined) deductive process of acquiring knowledge about a software artifact through analysis, abstraction, and generalization.“ [Tilley und Smith 96]

Ein Beispiel, das die Bedeutung des Themas unterstreicht, ist ein Untersuchungsergebnis von Fjeldstad und Hamlen, wonach bei Änderungen an im Einsatz befindlichen Systemen 47% des Aufwands für das Programmverstehen entfallen (für die Änderung von Code und Dokumentation werden nur 25% angegeben, für das Testen 28%) (zitiert in [Müller 97]). In einem Forschungsbericht der IBM wird das Programmverstehen als eine entscheidende Herausforderung für die aktuelle Softwaretechnik identifiziert: „ A key motivator for software tools in the 1990’s will be software evolved over decades from several thousand line, sequential programming systems into multi-million line, multi-tasking ‘business-critical’ systems. As the programming systems written in the 1960’s and 1970’s continue to mature, the focus for software tools will shift from tools to help develop new programming projects to tools to help us understand and enhance aging programming systems“ [Corbi 89]. Neben den individuellen Anpassungen und Erweiterungen, die für bestehende Software vorgenommen werden müssen, treten zudem

einschneidende Anforderungen auf, von denen eine große Zahl von Systemen gleichzeitig betroffen sind: Im Moment steht das „Jahr-2000-Problem“ im Mittelpunkt, und als nächstes wird die europäische Währungsumstellung folgen, die eine Überarbeitung von Programmen jeder Generation erfordern wird – objektorientierte Neuentwicklungen der letzten Jahre werden ebenso betroffen sein wie alte COBOL-Systeme.

Ob man nun von Wartung, Reengineering oder Reverse Engineering spricht: Das Programmverstehen besitzt offensichtlich zentrale Bedeutung. Eine bedeutende Rolle spielt das Programmverstehen aber auch während des Forward Engineerings, wenn Entwickler vorhandene Komponenten (wieder-) verwenden und sich dazu erst das Wissen über die Komponenten aneignen müssen.

Grundlegende kognitive Modelle des Programmverstehens sind Top-Down- und Bottom-Up-Vorgehen und die iterative Verfeinerung von Hypothesen. Bei der Top-Down-Methode beginnt man mit der Vorstellung von der Grundfunktionalität, die das System leistet, und untersucht mit diesem Vorwissen schrittweise die Systemkomponenten. Der Bottom-Up-Ansatz beginnt mit der Analyse des Quelltextes, indem man die Bedeutung einzelner Codefragmente herausarbeitet und die Fragmente zu sinnvollen Gruppierungen zusammenfaßt (diesen kognitiven Prozeß bezeichnet man als *chunking*). Das Zuweisen von Bedeutungen an die gebildeten Gruppierungen wird als *Concept Assignment* beschrieben [Biggerstaff et al. 94]. Bei der iterativen Verfeinerung werden Hypothesen über das System formuliert. Durch schrittweise Validierung, Verfeinerung und Erweiterung der Annahmen entsteht eine konsistente Menge von Hypothesen, durch die das untersuchte System beschrieben wird [Doebele 97].

Der häufigste Weg, den Entwickler für das Programmverstehen gehen, ist eine Mischung aus Top-Down- und Bottom-Up-Ansatz, die auch als opportunistische Strategie bezeichnet wird. Dabei wird sowohl das Vorwissen über Anwendungsbereich und Systemfunktionalität mit eingebracht als auch die Erfahrung im Verstehen von Quelltexten.

Unabhängig vom „Wie“ des Programmverstehens ist das „Wann“: Das Erarbeiten eines Programms kann systematisch erfolgen, bevor andere Tätigkeiten vorgenommen werden (Änderungen, Tests etc.), oder aber nach jeweiligem Bedarf, d.h. erst dann, wenn es für eine bestimmte Aktivität erforderlich wird. Diese „as-needed“-Strategie steht natürlich einem frühzeitigen Überblick über das Programm entgegen und ist dem systematischen Vorgehen unterlegen [Littman et al. 86].

2.1.3 Statische und dynamische Aspekte

Struktur und Verhalten sind zwei wesentliche, unterschiedliche Charakteristika von Programmen. Zur Struktur zählen alle statischen, d.h. von der Laufzeit unabhängigen Aspekte des Systems, etwa: Welche Komponenten gibt es? Wie sind Datenstrukturen aufgebaut? usw. Die dynamischen Eigenschaften kennzeichnen das Verhalten zur Laufzeit: Reihenfolgen von Funktionsaufrufen, Pfade durch bedingte Anweisungen, Arbeitsweisen von Algorithmen etc.

Zum Programmverstehen müssen Entwickler sowohl mit den statischen als auch mit den dynamischen Aspekten eines Systems vertraut werden. Um die dynamischen Aspekte zu untersuchen, gibt es zwei Wege. Der erste besteht darin, das Verhalten des Programms anhand des Quelltextes (oder einer anderen Beschreibung) zu antizipieren, ohne daß das Programm ausgeführt werden muß. Der andere Weg ist die Beobachtung des Programms während seiner Ausführung. Auch hier kann also unterschieden werden, ob die Analyse statisch oder dynamisch erfolgt.

Für die Frage nach der Struktur des Programms ist es unwesentlich, ob die Analyse statisch oder dynamisch erfolgt. Die Erkenntnisse über das Programmverhalten hängen hingegen stark von der Methode der Analyse ab:

- Die statische Analyse des Programmverhaltens kann ohne spezielle Werkzeugunterstützung erfolgen; geeignete Werkzeuge können jedoch sehr hilfreich sein. Das Verhalten kann exemplarisch (für bestimmte Fälle) „durchgespielt“ oder allgemein (für *alle* möglichen Programmausführungen) untersucht werden. Quantifizierte Aussagen (z.B. über den Verbrauch an Ressourcen) können in der Regel nicht gemacht werden.
- Die dynamische Analyse erfolgt zur Laufzeit und geschieht in der Regel exemplarisch. Die Analyse wird u.U. einfacher, weil das Augenmerk gezielt auf bestimmte Kennzeichen gerichtet werden kann und sich das Programm aus der Sicht des Benutzers (top down) erschließt, nicht aus der des Programmierers (bottom up). Für die dynamische Analyse sind geeignete Werkzeuge (z.B. Debugger, Profiler) zwingend erforderlich.

Der bereits erwähnte IBM-Forschungsbericht weist darauf hin, daß durch die dynamische Analyse neue Möglichkeiten des Programmverstehens eröffnet werden: „Studying the dynamic behaviour of an executing program can be very useful and can dramatically improve understanding by revealing program characteristics which cannot be assimilated from reading the source code alone“ [Corbi 89].

2.1.4 Werkzeugunterstützung

Das Verstehen größerer Programme ist ohne Werkzeugunterstützung nicht praktikabel. Die Werkzeuge müssen für drei Aktivitäten geeignet sein: Das Sammeln von Informationen, deren Verwaltung und deren Erschließung durch den Benutzer [Tilley und Smith 96].

- Um Informationen zu sammeln, müssen Werkzeuge automatische, halbautomatische und manuelle Verfahren bieten, mit denen Daten aus Quelltexten und anderen Dokumenten extrahiert werden können. Ein traditionelles Beispiel ist das Parsen von Quelltexten, um die Menge aller Symbole zu erhalten (also eine statische Analyse mit Compiler-basierten Techniken). Das Parsen ist automatisch möglich; das Problem bei vollautomatisch gewonnenen Informationen ist aber, daß diese in der Regel in einer sehr großen Menge generiert werden. Zum werkzeugunterstützten Sammeln von Informationen sollten daher immer Filter vorgesehen werden, um die Datenmenge eingrenzen zu können.

  Alternative Dokumente und Informationen sind dynamisch gewonnene Daten, Beschreibungen oder Kommentare. Gerade für die zuletzt genannten, informellen Kategorien endet die automatische Auswertbarkeit. Hier müssen Werkzeuge das Vorgehen des Benutzers sinnvoll unterstützen.
- Das Werkzeug muß die Verwaltung der Informationen durchführen. Ein bloßes Speichern der gewonnenen, rohen Daten ist noch nicht zufriedenstellend; hilfreicher ist es, wenn die Daten in einem Modell oder in einer Wissensbasis miteinander in Beziehung gesetzt werden. Das Modell soll im Sinne des Programmverstehens eher konzeptuell als technisch ausgerichtet sein.
- Der Benutzer muß die Information effizient erschließen können, d.h. die Werkzeuge müssen geeignete Verfahren zur Navigation, zur Analyse und zur Präsentation bereitstellen. Ansätze dazu sind hypertextbasierte Schnittstellen, erweiterte Suchfunktionen,

Konfigurierbarkeit und Programmierbarkeit durch den Benutzer, Visualisierungen und multimediale Aufbereitungen.

Auch wenn werkzeugunterstütztes Programmverstehen viele Möglichkeiten eröffnet, ist bei der Entwicklung solcher Werkzeuge immer wieder die Frage nach der Einsetzbarkeit zu stellen; nicht alles, was machbar ist, wird auch benötigt, und auch die fortschrittlichsten Techniken ändern nichts daran, daß Programmverstehen primär eine kognitive Leistung von Entwicklern ist. Werkzeuge können hierzu Unterstützung leisten, sollten aber nicht vom eigentlichen Ziel ablenken – „Tools which require complex query commands should not be used for program understanding, since often the programmer will be diverted from his primary purpose and begin struggling with secondary issues like compound query formation and syntax errors. A good deal of attention to the 'human factors' of program understanding is often required" [Corbi 89].

2.2 Grundlagen der visuellen Programmierung

Eine (potentiell mächtige) Art, Programmverstehen zu fördern, liegt darin, Software zu visualisieren. Damit verbindet man gewöhnlich nicht den Anspruch, *alle* Aspekte eines Systems zu erfassen und grafisch zu präsentieren, sondern nur bestimmte Teile davon.

Visuelle Techniken werden neben der Visualisierung von Software auch zur Erstellung von Programmen benutzt. Beide Bereiche fallen unter den Oberbegriff „Visuelle Programmierung".

Die Nutzung visueller Techniken erfolgt in der Informatik seit Jahr und Tag, und grafische Darstellungen von Computerprogrammen und Datenstrukturen sind so alt wie die Programmierung selbst. Erst in den sechziger Jahren war die Technik aber so weit fortgeschritten, daß grafische Repräsentationen nicht mehr nur auf dem Papier, sondern auch am Computer selbst vorgenommen werden konnten. Am MIT wurde in den 60er Jahren die erste interaktive Umgebung für eine visuelle Programmiersprache entwickelt [Sutherland 63]. Seitdem kamen mehr und mehr grafikorientierte Programmiersprachen und -werkzeuge hinzu. Spätestens mit der Verbreitung von grafischen Benutzeroberflächen sind seit Mitte der achtziger Jahre visuelle Programmierumgebungen nicht mehr nur Thema von Forschung und Entwicklung, sondern heute auch aus der kommerziellen Softwareentwicklung nicht mehr wegzudenken.

2.2.1 Visuelle Programmierung

Zunächst ist der Begriff *visuell* näher zu erklären. Im Zusammenhang mit Programmierung werden die Begriffe *visuell* und *textuell* gewöhnlich als Gegensatzpaar verwendet. Stefan Schiffer weist allerdings darauf hin, daß es angemessener ist – und dem Sprachgebrauch der Kognitionspsychologie entspricht –, *visuell* und *verbal* gegenüberzustellen [Schiffer 97]. Die textuelle Darstellung eines Programms ist demnach verbal, weil sie vom *verbalen Wahrnehmungsssystem* des Menschen verarbeitet wird. Grafische Darstellungen werden hingegen durch das *visuelle Wahrnehmungssystem* erfaßt. Die beiden Arten, ein Programm darzustellen, können folgendermaßen definiert werden:

- Die *textuelle (verbale) Repräsentation eines Programms* stellt das Programm als (eindimensionale) Sequenz von Anweisungen und Bezeichnern dar. Der Symbolvorrat, aus dem die Anweisungen und Bezeichner bestehen, ist eine bestimmte Menge einzelner Zeichen (Buchstaben, Ziffern, Satzzeichen, mathematische Formelzeichen).

- In der *visuellen Repräsentation eines Programms* werden Eigenschaften des Programms durch die räumliche Anordnung von Elemente beschrieben; es gibt also immer zwei oder mehr Dimensionen. Der Symbolvorrat besteht neben den von textuellen Systemen gewohnten Zeichen in der Regel aus grafischen Formen. Das Programm wird nicht nur durch die Formen, sondern auch durch deren Attribute definiert; so kann in einer visuellen Repräsentation die Stärke oder Farbe einer Linie eine bestimmte Bedeutung haben.

Visuelle Techniken werden sowohl für das Forward Engineering benutzt, indem Systeme durch visuelle Programmiersprachen spezifiziert werden, als auch für Reverse Engineering, indem Aspekte eines Programms grafisch aufbereitet und präsentiert werden.

Die Bezeichnungen, die für diese Anwendungsbereiche verwendet werden, variieren. In Brad Myers' häufig zitierter Taxonomie werden alle Systeme, in denen Softwareentwicklung durch grafische Methoden oder Oberflächen erfolgt, als visuelle Sprachen (Visual Languages) bezeichnet [Myers 90]. Seine Klassifikation definiert visuelle Programmierung (Visual Programming) und Programmvisualisierung (Program Visualization) als Unterbegriffe. Visuelle Programmierung ergibt sich nach seiner Definition durch die Möglichkeit, ein Programm „in a two- or more dimensional fashion" zu erstellen. Programmvisualisierung bedeutet, daß für ein Programm, das in konventioneller, textueller Form spezifiziert wurde, Grafik benutzt wird, um Aspekte des Programms oder seiner Ausführung zu illustrieren.

An Myers' Definitionen ist zweifache Kritik zu üben. Erstens ist es zweifelhaft, ob alle grafischen Entwicklungswerkzeuge als visuelle Sprachen bezeichnet werden können. Sprachen verfügen definitionsgemäß – gleich, ob es sich um natürliche oder formale Sprachen handelt – über ein Vokabular und eine Grammatik. Diese in allen grafischen Systemen zu benennen, ist mehr oder weniger problematisch. Bei visueller Programmierung liegen offensichtlich formale Sprachen zugrunde; deshalb erscheint es naheliegend, den Begriff der visuellen Sprachen auf grafische Programmiersprachen zu beschränken. Zweitens ist die Definition von Program Visualization unnötig eng gefaßt, wenn vorausgesetzt wird, daß visualisierte Programme in textuellen Sprachen erstellt wurden.

Anstelle von Myers' Definition benutze ich eine an [Goldberg et al. 95] und [Schiffer 97] angelehnte Fassung:

- *Visuelle Umgebungen* (visual environments) bezeichnen Werkzeuge, die den Umgang mit visuellen Repräsentationen von Programmen zulassen, unabhängig davon, ob das Programm verbal oder visuell beschrieben ist. Zu visuellen Umgebungen zählen jedoch keine Werkzeuge, die keinen direkten Bezug zu einer ausführbaren Version des Programms haben, d.h. keine Diagrammeditoren und keine grafischen CASE-Tools, die nicht zumindest Teile des Codes generieren können.
- *Visuelle Programmiersprachen* (visual programming languages) sind Programmiersprachen, die eine visuelle Syntax besitzen, d.h. Eigenschaften des Programms werden durch räumliche Informationen und durch grafische Attribute festgelegt.
- *Visuelle Programmierung* (visual programming) ist die Erstellung von Software mit visuellen Programmiersprachen.
- *VP-Systeme* sind visuelle Umgebungen, die visuelle Programmierung unterstützen.

2.2.2 Softwarevisualisierung

Das werkzeuggestützte Erzeugen visueller Darstellungen von Programmen wird als *Softwarevisualisierung* verstanden. Werkzeuge zur *Softwarevisualisierung* (program visualization) sind Teile von visuellen Umgebungen, mit denen visuelle Repräsentationen von Aspekten eines untersuchten Programms erzeugt werden. Es spielt keine Rolle, ob das Programm ursprünglich verbal oder visuell codiert wurde. Softwarevisualisierung setzt den Einsatz von Werkzeugen voraus; Handskizzen, die dieselben Sachverhalte darstellen, sind nicht unter Softwarevisualisierung einzuordnen [Schiffer 97]. Werkzeuge zur Softwarevisualisierung zählen zu den visuellen Umgebungen. Den Zusammenhang zwischen den Begriffen zeigt Abbildung 2.

Unter Programmvisualisierung fallen nach der gegebenen Definition auch Editoren, die die Syntax von textuellen Sprachen farblich hervorheben, oder Werkzeuge für „Pretty Printer"-Ausgaben. Solche Systeme nehmen eine Sonderrolle ein, weil sie kein eigenständiges grafisches Vokabular anbieten, sondern nur Ergänzungen zum textuellen Vokabular der Programmiersprachen hinzufügen, und werden in dieser Arbeit vernachlässigt. Softwarevisualisierung im engeren Sinn führt nicht zu (möglicherweise grafisch annotierten) Texten, sondern erzeugt Grafiken, Diagramme oder Animationen.

Um Softwarevisualisierung näher zu charakterisieren, sollen zunächst die Begriffe aus dem ersten Teil dieses Kapitels aufgenommen werden:

- Eine mögliche Anwendung von Softwarevisualisierung ist der Einsatz zum Zweck des Programmverstehens.
- Softwarevisualisierung kann statische und/oder dynamische Aspekte eines Programms erfassen (vgl. Abschnitt 2.1.3).
- Softwarevisualisierung kann unabhängig vom Ablauf des untersuchten Programms (statische Analyse) oder während dessen Laufzeit erfolgen (dynamische Analyse).
- Systeme zur Softwarevisualiserung erstellen im allgemeinen aus Informationen, die auf einer abstrakten technischen Ebene gewonnen wurden, grafische Dokumente, die den mentalen Modellen von Entwicklern näher kommen, und fallen damit unter das Reverse Engineering.

Systeme, die in diesem Bereich entwickelt wurden, haben sehr unterschiedliche Schwerpunkte und Ziele. Um die Systeme zu klassifizieren, haben verschiedene Autoren Einteilungen in Kategorien vorgenommen. Zu den am häufigsten zitierten Arbeiten zählen die Beiträge von Myers sowie Roman und Cox.

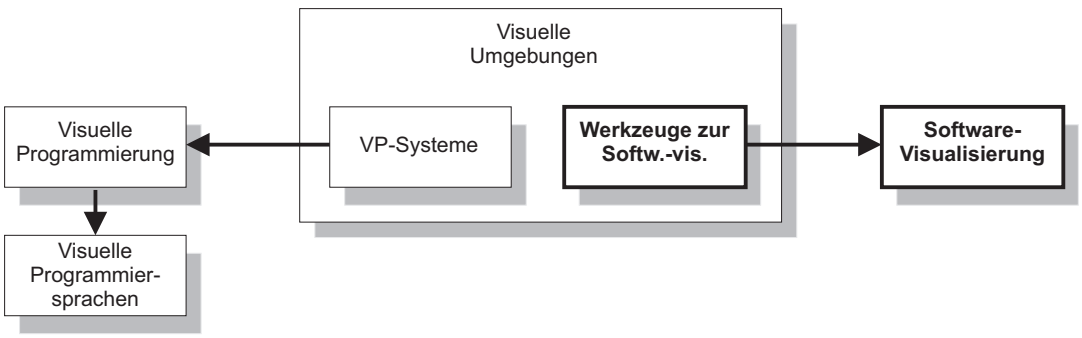


Abb. 2: Visuelle Programmierung, Visuelle Umgebungen und Softwarevisualisierung

Myers' Kategorisierung [Myers 90] erfolgt anhand von zwei Dimensionen. Die erste Achse enthält die Unterscheidung nach statischer und dynamischer Visualisierung (hier ist mit statisch gemeint, daß die Anzeige sich auf bestimmte Zustände, z.B. Schnappschüsse beschränkt, während eine dynamische Visualisierung eine kontinuierliche Animation ist). Die andere Achse zeigt auf, was Gegenstand der Visualisierung ist: Code, Daten oder Algorithmen.

Roman und Cox schlagen eine Taxonomie vor, durch die das Modell von Myers erweitert wird [Roman und Cox 93]. Ihre Kategorisierung erfolgt nach fünf Kriterien:

- *Bereich* (scope): Welche Bereiche eines Programms werden dargestellt? In Frage kommen der Quelltext, Daten, der Kontrollzustand (d.h. ein „Schnappschuß" während der Programmausführung) und das Verhalten (als Abfolge von Daten- und Kontrollzuständen).
- *Abstraktion* (abstraction): Wie weit kann das System vom technischen Level der rohen Informationen abstrahieren? Neben der direkten Repräsentation, die die Informationen unmittelbar grafisch umsetzt, gibt es strukturelle Repräsentationen, bei denen bestimmte Informationen gefiltert, hervorgehoben etc. werden, und synthetisierte Repräsentationen; bei letzteren werden Informationen visualisiert, die im untersuchten Programm zwar logisch, nicht aber explizit vorhanden sind, und im Rahmen der Visualisierung erst transformiert werden müssen.
- *Methode der Spezifikation* (specification method): Wie wird bestimmt, welche Aspekte eines Programms angezeigt werden? Dazu muß eine Abbildung von den Ereignissen im untersuchten Programm auf visuelle Ereignisse vorgenommen werden. In manchen Fällen ist diese Abbildung vordefiniert und unveränderbar. Ein gängiger Weg ist die Annotation des Originalprogramms, d.h. es werden in den Quelltext Aufrufe zum Erzeugen einer Grafik eingefügt. Weiterhin gibt es den deklarativen Ansatz, in dem die Visualisierung von Programmereignissen entkoppelt ist. Die erforderliche Abbildung kann innerhalb des Visualisierungssystems vorgenommen werden. Ein manipulativer Ansatz erlaubt, daß der Benutzer durch Beispiele vorgibt, wie Programmereignisse dargestellt werden sollen.
- *Benutzerschnittstelle* (interface): Die Benutzerschnittstelle wird durch die Art der Präsentation und der Interaktion charakterisiert. Die Präsentation kann durch mehr oder weniger komplexe grafische Elemente erfolgen und eine oder mehrere (von einander unabhängige) Sichten auf das untersuchte Programm zulassen. Hinsichtlich der Interaktion läßt sich unterscheiden, ob der Benutzer das Aussehen einer Visualisierung durch Kontrollelemente oder aber durch direkte Manipulation der angezeigten Grafik beeinflussen kann.
- *Präsentation* (presentation): Wie wird versucht, den Informationsgehalt einer Visualisierung möglichst zu erhöhen? Dazu kann z.B. zählen, die Darstellung gezielt an bestimmten analytischen Fragestellungen auszurichten, erklärende Elemente mit aufzunehmen und die Aussagefähigkeit durch verschiedene parallele Sichten auf ein Programm zu verbessern.

Diese Kriterien geben eine gute Richtlinie, um verschiedene Visualierungwerkzeuge bewerten zu können. Was jedoch fehlt, ist eine Betrachtung der Einsatzkontexte, für die Visualisierungssysteme erstellt werden: Wer soll das System zu welchem Zweck benutzen? Als Beispiele seien zwei mögliche Antworten genannt: Ein Einsatzkontext ist die Verwendung in Studium oder Ausbildung. Visualisierungssysteme können hier benutzt werden, um Programmieranfängern grundlegende Konzepte zu verdeutlichen. Ein ganz anderer Einsatzkontext (mit deutlich anderen Anforderungen) entsteht in der professionellen Softwareentwicklung, wenn Fachleute Visualisierungssysteme nutzen, um Programmfehler zu lokalisieren.

2.2.3 Die Vorteile visueller Programmierung

Bisher wurde offengelassen, warum die Verwendung visueller Techniken in der Softwareentwicklung von Nutzen ist. Daß ein Bild mehr sagt als tausend Worte, ist *common sense*; in diesem Abschnitt soll versucht werden, das Thema aus kognitionswissenschaftlicher Sicht zu beleuchten.

Dem Einsatz visueller Techniken liegt offensichtlich die Annahme zugrunde, daß

- (bezogen auf visuelle Programmiersprachen) das Erstellen eines Programms mit visuellen Programmiersprachen leichter fällt als mit textuellen und daß
- (bezogen auf Softwarevisualisierung) das Verstehen eines Programms leichter fällt, wenn es in einer visuellen anstelle einer textuellen Repräsentation vorliegt.

Als empirischen Beleg dieser Annahme mag man die Tatsache ansehen, daß Entwickler und Programmierer schon immer Diagramme benutzt haben, um Eigenschaften eines Programms darzustellen, auch bevor es dazu unterstützende Software gab.

Diagramme sind als visuelle Technik anscheinend gut dazu geeignet, komplexe Sachverhalte in einer Art auszudrücken, die für das menschliche Erkennen besonders günstig ist. Das Wissen über die Funktionsprinzipien des Gehirns ist jedoch relativ klein, und daher ist es müßig zu spekulieren, ob das Gehirn Wissen in einer Art speichern kann, die mit der Struktur von Diagrammen vergleichbar ist [Petre et al. 97].

Eine Arbeit, die sich gezielt mit den Unterschieden von textuellen und visuellen Repräsentationen beschäftigt, haben Larkin und Simon veröffentlicht [Larkin und Simon 87]. Diese Unterschiede arbeiten die Autoren für die *kognitiven Programme* heraus, die auf den Repräsentationen arbeiten: Die Programme sind für

- das (Wieder-)Erkennen von Informationen,
- das Suchen nach Information und
- das Ableiten zusätzlichen Wissens

verantwortlich. Während das Erkennen nach Aussage von Larkin und Simon in visuellen Darstellungen nicht grundsätzlich efffizienter erfolgt als in textuellen, profitiert das Suchen bei visuellen Repräsentationen von der zweidimensionalen Indexierung, die durch die räumliche Anordnung gegeben ist. Da zusammenhängende Informationen häufig an nahegelegenen Orten positioniert sind, läßt sich Information in der Regel schnell finden, indem erst der ungefähre Bereich festgestellt und danach in dessen Umgebung gesucht wird; eine eindimensionale, sequentielle Suche erlaubt keine so effiziente Suche. Das größte Potential visueller Repräsentation liegt aber erst in der Ableitung von zusätzlichem Wissen: Diagramme können bereits viele Zusammenhänge explizit sichtbar machen, die aus einer ansonsten äquivalenten textuellen Darstellung erst erschlossen werden müssen [Larkin und Simon 87]. Ein anderer Vorteil visueller Darstellungen liegt in der Anzahl verfügbarer Ausdrucksmittel: Eine textuelle Repräsentation besteht aus einzelnen Zeichen, die einem relativ kleinen Alphabet entstammen; für visuelle Repräsentationen stehen Formen, Farben, Anordnungen usw. zur Verfügung. Beispielsweise kann man acht Dimensionen aufzählen, die miteinander kombinert werden können: neben horizontalen und vertikalen Koordinaten kommen Form, Farbe, Größe, Ausrichtung, Füllmuster und Helligkeit zum Einsatz [Bertin 81]. Die Zeichen, die in einer textuellen Repräsentation verwendet werden, unterscheiden sich zunächst nur in ihrer Form. Typographie erhöht die Auswahl an

Ausdrucksmitteln, reicht aber noch nicht an visuelle Repräsentationen im Sinne von Diagrammen heran.

Daß Programmierer visuelle Darstellungen benutzen, hängt letztlich auch von verschiedenen nicht objektivierbaren Kriterien ab. So kommt eine Befragung zu dem Ergebnis, daß grafischen Repräsentationen folgende Eigenschaften beigemessen werden [Petre et al. 97]:

- höherer Informationsgehalt bzw. höhere Informationsdichte,
- höhere Abstraktion und damit bessere Abbildung des Problembereichs,
- schneller erkennbare Strukturen,
- leichterer Überblick, leichter zu erfassen,
- leichter einzuprägen,
- angenehmer, weil weniger formal und „more fun".

Diese Aussagen werden teilweise durch Experimente unterstützt, teilweise aber auch widerlegt: Zitiert wird ein Versuch, in dem den Versuchspersonen das Entdecken von ähnlichen Strukturen anhand der textuellen Darstellung leichter fiel als anhand der grafischen [Petre et al. 97]. Ähnliche Aussagen greift Stefan Schiffer auf und formuliert daraus fünf Thesen, die man typischerweise als Vorteile visueller Techniken findet – etwa, daß Bilder leicht verständlich seien und das Lernen erleichtern – und für die gezeigt wird, daß sie nicht allgemein gültig sind [Schiffer 96]. In eine ähnliche Richtung geht eine Untersuchung von Alan Blackwell, der 140 Artikel zur visuellen Programmierung ausgewertet hat, um Aussagen über kognitive Theorien zu sammeln. Er kommt zu dem Schluß, „[that] most of the metacognitive beliefs found in this survey have a logical basis in introspection, cognitive theory, or folk psychology. In some cases, they are well-founded, but in others there is either logical or empirical evidence making them inappropriate." [Blackwell 96]

2.2.4 Anforderungen an die Softwarevisualisierung

Die im vorangegangen Abschnitt wiedergegebenen Arbeiten zeigen, daß es einige kognitive Vorteile von grafischen Darstellungen gibt, aber es wird auch deutlich, daß visuelle Präsentationen textuellen Formen nicht immer und nicht in allen Kontexten überlegen sind.

Wer Werkzeuge zur Softwarevisualisierung entwickelt, steht vor der Aufgabe, die kognitiven Vorzüge visueller Techniken so einzusetzen, daß die Anwender darin einen Nutzen erkennen. Petre et al. beschreiben dazu zunächst die Gründe, aus denen Benutzer Visualisierungswerkzeuge benötigen, und die Anforderungen, die sich daraus ergeben. Bei den Benutzern unterscheiden sie zwischen Fachleuten und Nicht-Fachleuten (z.B. Programmieranfängern). Für Fachleute hat Programmvisualiserung u.a. folgende Aufgaben [Petre et al. 97]:

- Die Nutzung von Visualisierungen als „externes Gedächtnis".
- Das Ermöglichen von Diskussionen mit anderen Experten.
- Visualisierung als Mensch-Maschine-Schnittstelle, um in Zusammenarbeit mit dem Computer Probleme zu lösen.

Der Nutzen für Nicht-Fachleute liegt darin, daß Visualisierungen Einblicke in die Denkweise von Experten und in die Arbeitsweise von Computern geben können. Geeignete Visualisierungswerkzeuge gehen in der Regel mit relativ kleinen Problemen (häufig mit einfachen Bei-

spielen) um und profitieren von vereinfachenden, schematisierenden Anzeigen. Die Anforderungen von Experten weichen davon stark ab [Petre et al. 97]:

- Experten wollen die *Kontrolle* über die Visualisierung haben, d.h. sie wollen selbst den Fokus der Visualisierung bestimmen können,
- sie brauchen Visualiserungswerkzeuge, die sich für *große Probleme („real-sized problems")* eignen und somit über angemessene Navigations- und Suchverfahren verfügen,
- sie brauchen *effiziente Werkzeuge*, die Visualisierungen in hoher Geschwindigkeit bieten,
- sie legen Wert auf die *Korrektheit* und *Vollständigkeit* der Anzeige *(„truth")*, d.h. keine vom System vorgegebenen Vereinfachungen oder Auslassungen.

Je mehr es sich bei den Anwendern um Experten handelt, desto vorsichtiger sollten Entwickler von Visualisierungswerkzeugen mit Annahmen über die Benutzung ihrer Werkzeuge umgehen: Die Anwender wissen als Experten am genauesten, welche Information sie zu welcher Zeit benötigen. Werkzeuge sollten daher nicht für einen bestimmten Anwendungskontext entwickelt werden, sondern für die Verwendung in möglichst vielen Kontexten offenstehen: „Software visualisation builders are in danger of designing a tool that is not available in the context in which it is most needed, since we know from observation that people reason about programs in non-standard environments." [Petre et al. 97].

2.2.5 Zusammenfassung der Kriterien

Die zuletzt genannten Anmerkungen von Petre et al. unterstreichen die Bedeutung, die dem Einsatzkontext von Visualisierungssystemen zukommt. Entwickler von derartigen Systemen müssen also zunächst eingrenzen, von welchen Benutzergruppen und für welche Fragestellungen ein System eingesetzt werden soll. Der Kontext umfaßt auch die technischen Randbedingungen, die zum Einsatz des Systems erfüllt sein müssen. Zusammen mit der Taxonomie von Roman und Cox ergibt sich damit ein Katalog von Unterscheidungsmerkmalen, nach denen Programme zur Softwarevisualisierung bewertet werden können (Tabelle 1).

Kriterium	Charakteristische Merkmale
Einsatzkontext	- Benutzergruppe des Systems - Zweck (allgemeines Verständnis, Fehlersuche...) - technische Bedingungen
Bereich	- statische oder dynamische Aspekte eines Programms - Darstellung von Code, Daten oder Verhalten
Abstraktion	- direkte Umsetzung von erfaßten Informationen oder Abstraktionen
Spezifikation	- wie wird der Umfang der Darstellung bestimmt? - muß eine Annotation (Instrumentierung) der Quelltexte erfolgen?
Benutzerschnittstelle	- benutzte Darstellungsarten - verfügbare Darstellungselemente - kann die Darstellung interaktiv manipuliert werden?
Präsentation	- Semantik der Darstellung - werden erklärende Elemente in die Darstellung aufgenommen?

Tab. 1: Unterscheidungsmerkmale für Systeme zur Softwarevisualisierung

2.3 Arbeiten zu objektorientierten Visualisierungssystemen

Nachdem in den vorangegangen Abschnitten eine allgemeine Einordnung der Softwarevisualisierung erfolgte, wird im weiteren Verlauf dieser Diplomarbeit die Visualisierung dynamischer Aspekte von objektorientierten Programmen im Mittelpunkt stehen. Es gibt eine Reihe anderer Arbeiten, die sich ebenfalls mit diesem Thema befassen. Im folgenden stelle ich diejenigen Arbeiten vor, in denen entsprechende Visualisierungswerkzeuge beschrieben werden. Da ich in den späteren Kapiteln wiederholt auf das Kriterium der Abstraktion eingehen werde, orientiere ich mich bereits an dieser Stelle am selben Merkmal: Zunächst werden Ansätze vorgestellt, die das Verhalten von objektorientierten Programmen direkt repräsentieren, d.h. auf der Abstraktionsebene, auf der Informationen über Objekte erfaßt werden. Im Anschluß folgen Arbeiten, die sich von dieser Ebene lösen und die versuchen, bestimmte Strukturen und Eigenschaften der Softwarearchitektur darzustellen, indem objektübergreifende Abstraktionen gebildet werden.

2.3.1 Visualisierung auf Objektebene

Zu den ersten Werkzeugen, mit denen die Dynamik objektorientierter Programme visuell dargestellt wurde, zählt ein erweiterter Smalltalk-Debugger: In [Cunningham und Beck 86] stellen die Autoren eine grafische Notation für Methodenaufrufe zwischen Objekten vor und präsentieren eine Erweiterung eines Smalltalk-Debuggers, der entsprechende Diagramme automatisch erzeugt. Bei der Notation handelt es sich um Objektdiagramme, in denen der Nachrichtenverkehr zwischen Objekten, besonders aber auch innerhalb der Objekte ausgedrückt werden soll. Die Notation und die zugehörige Implementierung eignen sich eher für kleine Beispiele und zu Unterrichtszwecken, aber kaum für die Analyse größerer Programme.

Für umfangreichere Probleme wurde GraphTrace entwickelt [Kleyn und Gingrich 88]. Die Autoren wollen mit dem System das Verstehen großer objektorientierter Programme erleichtern. Dazu bieten sie sowohl statische als auch dynamische Diagramme an. Die Grundstruktur der Diagramme ist baumartig; je nach Diagramm stehen die Knoten der Bäume für Klassen, Objekte oder Methoden. Der Programmablauf wird durch Animationen der Diagramme dargestellt. Dabei ändert sich der Aufbau der Graphen nicht, sondern es werden in zeitlicher Abfolge einzelne Knoten farblich hervorgehoben. GraphTrace wurde für den objektorientierten Lisp-Dialekt Strobe geschrieben. Die Laufzeitinformationen werden erfaßt durch eine Modifikation der internen Funktion, die bei Methodenaufrufen verwendet wird. Vor der eigentlichen Visualisierung muß das untersuchte Programm einmal ausgeführt werden, um die strukturellen Informationen zu erfassen, und erst in einem zweiten Durchlauf entstehen dann die Animationen.

Im Thomas J. Watson Reasearch Center der IBM hat es unter den Namen Object Visualizer und HotWire zwei Projekte zur objektorientierten Softwarevisualisierung gegeben. Zu HotWire [Laffra und Malhotra 94] gehört ein Präprozessor, der den Quelltext eines untersuchten C++-Programms instrumentiert und Aufrufe an das HotWire-Laufzeitsystem einfügt, mit dem das Programm anschließend gelinkt wird. Ziel des Systems ist es, mit so vielen unterschiedlichen Visualisierungen wie nötig die Suche nach Programmfehlern zu unterstützen. Dazu gibt es neben einer Reihe von vorgefertigten Diagrammen eine Skriptsprache, in der der Benutzer zusätzliche Darstellungsarten erstellen kann. Die vorgefertigten Darstellungen zeigen z.B. Übersichten von Klassen, Instanzen und Interaktionen der Instanzen. Vom Benutzer erstellte Skripts können Diagramme erzeugen, in denen Instanzenvariablen grafisch umgesetzt werden. So kön-

nen in Diagrammen z.B. fachliche Eigenschaften von Objekten durch grafische Attribute ausgedrückt werden.

Object Visualizer [De Pauw et al. 93] beruht ebenfalls auf Instrumentierung von C++-Quelltexten. Das System legt während der Laufzeit eines analysierten Programms eine Datenbasis an, in der alle Methodenaufrufe gespeichert werden. Diese Datenbasis wird Ereignisraum genannt und ist auf Vollständigkeit, Kompaktheit und effiziente Zugriffsmöglichkeiten ausgelegt (ausführlich in [De Pauw et al. 94]). Die Diagramme, die das System bietet, werden durch Anfragen an den Ereignisraum erzeugt. Insbesondere lassen sich durch die Anfragen sehr einfach akkumulierte Werte ermitteln, z.B. die Anzahl aller bisher erfolgten Methodenaufrufe zwischen zwei bestimmten Klassen. Dementsprechend bietet ObjectVisualizer mehrere Diagramme, die solche akkumulierten Informationen darstellen. Zusätzliche, animierte Darstellungstechniken lassen sich relativ einfach realisieren, indem die Struktur des Ereignisraums auf die vorgesehenen Dimensionen animierter Diagramme (Zeit, räumliche Anordnung und Farbe) abgebildet wird. Insgesamt eignet sich das Konzept des Systems, um einen Überblick über das Programmverhalten zu erhalten, während die Betrachtung einzelner Objektinstanzen in den Hintergrund rückt.

Ebenfalls aus einem IBM-Forschungslabor stammt das System ProgramExplorer [Lange und Nakamura 97]. Es wurde ebenfalls für C++-Programme entwickelt und kommt ohne Instrumentierung aus. Die benötigten Laufzeitinformationen werden mit Hilfe des IBM Heap View Debuggers ermittelt; das untersuchte Programm muß daher mit einem passenden IBM-Compiler übersetzt worden sein. Nachdem der Benutzer die Menge der zu erfassenden Objekte eingegrenzt hat, erzeugt das System drei Arten von Diagrammen: In einem Objektdiagramm werden alle Instanzen dargestellt sowie die Methodenaufrufe zwischen ihnen. In einem Klassendiagramm werden alle Instanzen des Objektdiagramms als ein Element dargestellt; wie im Objektdiagramm werden die Elemente entsprechend der Methodenaufrufe untereinander verbunden. Als dritte Form werden einfache Interaktionsdiagramme erzeugt.

Mit dem expliziten Ziel, die Einarbeitung in Frameworks zu erleichtern, wurde ein weiteres Visualisierungswerkzeug entwickelt [Cheng und Gray 92]. Es instrumentiert den Quelltext zunächst derart, daß eine Trace-Datei erzeugt wird, die danach vom eigentlichen Werkzeug zu einer animierten Anzeige aufbereitet wird. Welche Klassen dargestellt werden, kann der Benutzer vorher auswählen. Die Anzeige beschränkt sich darauf, Icons für die Objekte der ausgewählten Klassen aufzulisten und, im selben Bildschirmfenster, den Aufrufstack ebenfalls in Form von Icons zu präsentieren. Weil nur der Anwendungs-, nicht aber der Framework-Quelltext instrumentiert werden kann, werden Objekte von Framework-Klassen nicht angezeigt. Die Autoren sehen darin eine nützliche Vereinfachung. Die Verwendung der Trace-Datei ermöglicht die einfache Reproduzierbarkeit von Animationen, verhindert aber die Analyse zur Laufzeit des Programms.

Ein weiteres Visualisierungswerkzeug ist VizBug++ [Jerding und Stasko 94]. Das Werkzeug benutzt eine Darstellung, die verschiedenste Klassen- und Objekteigenschaften in einem Diagramm anzeigt und vom visuellen Entwurfwerkzeug GROOVE [Shilling und Stasko 92] übernommen wurde. Interessanter als die Darstellungstechnik ist das allgemeine Framework, das die Entwickler auf Basis des VizBug++-Protoypen entwickelt haben und dessen erweiterbare Architektur sie vorstellen. Für objektorientierte Programme geben sie eine Liste von Ereignissen an, die die statischen und dynamischen Programmeigenschaften vollständig beschreiben sollen; durch Instrumentierung wird der Quelltext ergänzt, so daß die Ereignisse während der

Programmausführung den Framework-Komponenten mitgeteilt werden. Das angestrebte Ziel ihrer Arbeit ist es, das Verstehen objektorientierter Programme zu erleichtern.

Das Werkzeug *Scene* [Koskimies und Mössenböck 96] erzeugt eine Art erweiterter Interaktionsdiagramme für Oberon-Programme. Für die Darstellung der Diagramme wird ein Framework benutzt, das hypertextähnliche Fähigkeiten zur Verfügung stellt und das es den Diagrammelementen ermöglicht, auf Benutzeraktionen zu reagieren. Damit entsteht eine Benutzerschnittstelle, die einfachen Zugriff auf zusätzliche Anzeigen (z.B. zugehörige Quelltexte) bietet. Auch Scene instrumentiert die untersuchten Programme durch zusätzlichen Code.

Der *Object Architecture Analyzer* [Fröhlich und Stranzinger 97]führt eine eigene Notation ein, die es erlaubt, eine Programmausführung als Folge von Icons anzuzeigen. Die Icons werden in verschiedenen Sichten verwendet, in denen z.B. der Stack, Interaktionen zwischen Objekten und die Struktur von Objekten zu sehen sind. Daß die Notation in den unterschiedlichen Diagrammen einheitlich benutzt wird, ist ein Vorteil, die Icons sind jedoch nicht intuitiv verständlich. Die Autoren verstehen den Begriff der Architektur auf der Ebene von Objekten, und auf dieser Ebene ist auch die Notation zu verstehen. Bei großen Programmen dürfte die Darstellung schnell zu Diagrammen führen, die sehr detailliert und schwer zu überblicken sind. Als Technik zur Informationserfassung dient auch hier die Instrumentierung der Quelltexte.

Bei allen bisher erwähnten Visualisierungssystemen handelt es sich um Forschungsprojekte. Ein kommerzielles Produkt wird unter dem Namen *Look!* vertrieben [West 94]. Look! wird unter Unix und Microsoft Windows angeboten und dient zum Debuggen und Verstehen von C++-Programmen. Die Programme müssen nicht instrumentiert werden, sie müssen lediglich mit Debug-Informationen compiliert worden sein. Unterstützt werden verschiedene textuelle und visuelle Darstellungen, darunter statische Klassendiagramme, zwei Arten von Objektdiagrammen für Referenzen und Erzeugungsbeziehungen, Attributwerte von Objekten und eine Liste aller erfaßten Methodenaufrufe. Diagramme, die dynamische Eigenschaften zeigen, sind animiert; die Geschwindigkeit, in der das beobachtete Programm ausgeführt wird, kann variiert werden, damit der Benutzer die Animationen verfolgen kann. Die Menge der erfaßten Informationen wird durch statische und dynamische Filter eingeschränkt. Die dynamischen Filter können z.B. erkennen, daß eine Methode mehrfach in Folge aufgerufen wurde, und bieten dann an, den Aufruf fortan zu ignorieren.

2.3.2 Visualisierung mit objektübergreifenden Abstraktionen

Nach verschiedenen Arbeiten, die am *Georgia Institute of Technology* entstanden und zu denen auch das oben erwähnte *VizBug++* zählt, ist man dort zu der Einsicht gelangt, daß es für das Verständnis großer Programme weniger auf die Animationen ankommt, sondern stärker auf das Navigieren innerhalb von Diagrammen und das Fokussieren einzelner Stellen [Jerding et al. 96]. Daher wurde eine Anzeige in Form einer „Execution Mural“ entwickelt, die sich als schematisches Interaktionsdiagramm beschreiben läßt. Das Diagramm ist entlang der Zeitachse stark komprimiert, so daß ein kompletter Programmablauf mit tausenden von Nachrichten in einem Bildschirmfenster dargestellt wird. Durch unterschiedliche Farbintensitäten werden in dieser komprimierten Ansicht Muster von Nachrichtenfolgen gekennzeichnet, die das System selbsttätig erkannt hat. Das Kriterium für die Mustererkennung ist lediglich das Wiederkehren von Nachrichtensequenzen, taugt aber dazu, einen Überblick zu erhalten und Entwurfsmuster (hier: Verhaltensmuster) zu identifizieren. Der Benutzer kann Abschnitte der Anzeige vergrößern und sich in die Muster „hineinzoomen“, bis einzelne Nachrichten und deren Namen er-

kennbar werden. Das Konzept scheint für große Probleme geeignet zu sein, denen die technische Grundlage – Instrumentierung und Trace-Dateien – noch im Wege stehen dürfte.

In [Lange und Nakamura 95] beschreiben die Entwickler des oben bereits erwähnten Program Explorer, wie sich ihr Werkzeug zum Auffinden von Entwurfsmustern in Framework-Architekturen eignet. Eine Erweiterung des Werkzeugs, etwa um grafische Symbole für Entwurfsmuster, erfolgt jedoch nicht. Vielmehr stellen die Autoren heraus, durch welche allgemeinen Eigenschaften Muster aufgefunden werden können, und beschreiben, wie die Schnittstelle des Program Explorers das Auffinden der Muster unterstützt. Voraussetzung dazu ist, daß der Benutzer eine Vorstellung davon hat, welche Muster an welchen Stellen auftreten.

In [Ortigosa und Campo 96] wird ein System namens *Luthier* vorgestellt. Das analysierte Programm muß hier vom Programmierer um Meta-Klassen ergänzt werden, mit denen zur Laufzeit Klassen und Objekte nachgebildet werden. Visualisiert wird dann das Verhalten der resultierenden Meta-Objekte. Neben der dynamischen Visualisierung einzelner Instanzen bietet das System auch eine Architekturebene: Dazu wird der Programmcode geparst, und es werden die Vererbungs- und Benutzungsbeziehungen festgestellt. Die am engsten miteinander verknüpften Klassen werden dann als (mögliches) Subsystem interpretiert und in einem Diagramm zusammengefaßt. Diese Interpretation ist natürlich nicht immer zutreffend; nach Aussage der Autoren hat es in den meisten untersuchten Beispielen dennoch zum besseren Verstehen von Anwendungsprogrammen geführt.

In einem anderen Ansatz werden Architektureigenschaften bereits bei der Instrumentierung des Quelltextes angegeben [Sefika et al. 96]. Es entsteht damit ein Modell des untersuchten Programms, das hierarchisch aus Subsystemen, Frameworks, Entwurfsmustern, Klassen, Objekten und Methoden zusammengesetzt ist. Während andere Visualiserungssysteme erst versuchen, die Architektur erkennbar zu machen, ist das Architekturmodell hier bereits durch die Instrumentierung bekannt. Die Präsentation erfolgt in mehreren Diagrammarten, darunter auch Interaktionsdiagramme und verschiedene akkumulierende Anzeigen. Eine Stärke des Systems ist die flexible Struktur des Architekturmodells, da sich die angegebenen Bestandteile beliebig kombinieren lassen und damit prinzipiell jede Programmarchitektur nachgebildet werden kann. Die Schwäche liegt in der erforderlichen Instrumentierung; diese setzt die Kenntnis der Architektur voraus, bevor der Benutzer mit der Visualisierung beginnen kann.

2.4 Zusammenfassung

Ausgangspunkt des Kapitels waren die Begriffe *Reverse Engineering* und *Programmverstehen*. Das Programmverstehen ist eine wesentliche Aufgabe bei der Pflege und Weiterentwicklung bestehender Programme wie auch beim Einsatz wiederverwendbarer Programmkomponenten. Der kognitive Prozeß des Verstehens kann durch geeignete Werkzeuge unterstützt werden. Der Einsatz von Softwarevisualisierung macht solche Werkzeuge zu potentiell mächtigen Hilfsmitteln. Systeme zur Softwarevisualisierung lassen sich nach verschiedenen Kriterien unterscheiden, nämlich nach ihrem Einsatzkontext, den dargestellten Aspekten, nach möglichen Abstraktionen, der Art, in der der Umfang der Visualisierung bestimmt wird, und nach Eigenschaften von Benutzerschnittstelle und Präsentation. Vorgestellt wurden eine Reihe von Arbeiten, die Visualisierungswerkzeuge für das dynamische Verhalten von objektorientierten Programmen beschreiben. Die Werkzeuge befinden sich fast ausnahmslos im Entwicklungsstadium; unter ihnen ist nur ein Produkt, das es bis zur Marktreife gebracht hat. Erst in jüngster Zeit

sind Arbeiten entstanden, in denen versucht wird, von der Ebene der einzelnen Objekte zu abstrahieren und objektübergreifende Architektureigenschaften zu visualisieren. Diese Ansätze haben zum Ziel, Entwurfsmuster aufzuzeigen und das Verständnis von Frameworks zu erleichtern. Auf die Eigenschaften von Entwurfsmustern und Frameworks wird im folgenden Kapitel näher eingegangen.

Kapitel 3
Wiederverwendbare Elemente

In der Einleitung wurde bereits angedeutet, daß Softwarevisualisierung gerade für den Umgang mit wiederverwendbaren Programmelementen interessant erscheint: Nur diejenigen Elemente, die von Anwendungsentwicklern verstanden werden, lassen sich wiederverwenden; Softwarevisualisierung ist hier ein potentiell mächtiges Hilfsmittel. Dieses Kapitel erklärt die Formen der Wiederverwendung aus softwaretechnischer Sicht und dient damit als Grundlage für die anschließenden Kapitel, in denen Visualisierung auf wiederverwendbare Elemente angewendet wird.

3.1 Eigenschaften objektorientierter Sprachen

Objektorientierte Softwareentwicklung ist in den neunziger Jahren zum dominierenden Paradigma der Softwaretechnik geworden. Am Anfang stand der (kommerzielle) Erfolg der objektorientierten Programmiersprachen C++ und Smalltalk; danach folgten die unterschiedlichen Methoden zur objektorientierten Analyse und zum objektorientierten Entwurf, durch die der ganze Entwicklungsprozeß von den Stärken der neuen Sprachen profitiert.

Was objektorientierte Programmiersprachen auszeichnet, sind:

- Modularität und Lokalität: Klassen sind erweiterbare strukturierte Datentypen. Durch strukturierte Datentypen werden Modularität und Lokalität unterstützt, und auf dieser Basis lassen sich bereits abstrakte Datentypen definieren. Abstrakte Datentypen verkörpern ein Grundkonzept der Wiederverwendbarkeit.
- Kapselung (Datenabstraktion): Schnittstellen und Implementationen können voneinander getrennt werden. Die Schnittstellen lassen eine implizite Art des *Programming by Contract* zu [Meyer 88]; mittels Vor- und Nachbedingungen läßt sich die Funktionalität einer Klasse (unabhängig von der Implementation) in einer Klassenschnittstelle definieren.
- Vererbung von Code: Klassen können durch Vererbung Implementationen aus Oberklassen erben, ohne daß die Oberklassen verändert werden müßten. Damit ist eine bessere Erweiterbarkeit und Änderbarkeit als in nicht-objektorientierten Sprachen möglich. Wiederverwendbare Komponenten können durch Code-Vererbung gut erweitert und damit an spezielle Einsatzkontexte angepaßt werden.
- Polymorphie: Durch Polymorphie muß zur Compilezeit nur das Vorhandensein eines Schnittstellen-Features bekannt sein, nicht aber dessen Implementierung; die Bindung an die Implementierung wird erst zur Laufzeit in Abhängigkeit vom Objekttyp

vorgenommen. Während die Vererbung von Implementationen auch ohne Polymorphie sinnvoll ist, läßt sich die Vererbung von Schnittstellen erst durch dynamisches Binden konsequent nutzen. Klassen können abstrakt beschrieben werden, und zwischen Klassen lassen sich Generalisierungs- und Spezialisierungsbeziehungen festlegen. Weil diese Beziehungen sowohl in der Analyse und im Entwurf als auch in der Programmierung benutzt werden können, kann der Software-Entwicklungsprozeß bruchloser erfolgen.

Die Aufzählung dieser Eigenschaften soll zweierlei zeigen: Erstens, daß durch objektorientierte Programmiersprachen wesentliche softwaretechnische Anforderungen erfüllt werden, wodurch diese Sprachen besonders zur Erstellung großer Systeme tauglich sind. Zweitens, daß objektorientierte Sprachen ein besonderes Potential bezüglich der Wiederverwendbarkeit von Software besitzen.

Auch wenn diese Potentiale seit mehr als zehn Jahren zur Verfügung stehen und als Argumente für Objekttechnologien ins Feld geführt werden, wird erst in den letzten Jahren verstärkt untersucht, wie die Möglichkeiten der Wiederverwendbarkeit voll ausgeschöpft werden können. Anhand von Klassenbibliotheken, Entwurfsmustern und Frameworks beschreibe ich im folgenden drei Konzepte der objektorientierten Wiederverwendung.

3.2 Klassenbibliotheken

Eine Klassenbibliothek ist eine Sammlung von Klassen, die nützliche Funktionalität bietet, ohne dabei eine bestimmte Entwurfsstruktur für die Anwendung festzulegen. Klassenbibliotheken waren die ersten wiederverwendbaren Elemente, die im Zusammenhang mit objektorientierten Sprachen aufkamen. Sie traten die unmittelbare Nachfolge der Funktionsbibliotheken an, die in anderen Programmiersprachen benutzt wurden. Auf C++ bezogen beschreibt Bjarne Stroustrup, daß die ersten Bibliotheken in der Tradition anderer Sprachen standen und die neuen Sprachkonzepte noch nicht konsequent nutzten:

„Die frühen C++-Bibliotheken zeigten die Tendenz zur Imitation von Sprachkonzepten anderer Sprachen. So stellt zum Beispiel [...] [die] ursprüngliche Prozeßbibliothek, die die allererste C++-Bibliothek überhaupt war, Simula67-ähnliche Mechanismen für Simulationen zur Verfügung, die Bibliothek für komplexe Arithmetik ist mit den in der mathematisachen C-Bibliothek enthaltenen Funktionen der Fließkomma-Arithmetik zu vergleichen, und Keith Gorlens NIH-Bibliothek ist ein C++-Aquivalent der entsprechenden Smalltalk-Bibliothek. Auch die neueren „ersten Bibliotheken“ vermitteln noch den Eindruck, daß die Programmierer Überläufer von anderen Sprachen sind. Sie scheinen mit der Erstellung von Bibliotheken zu beginnen, bevor sie die Techniken und Entwurfsmöglichkeiten von C++ gänzlich angenommen haben – und auch schätzen gelernt haben.“ [Stroustrup 94]

Ebenso wie die Funktionsbibliotheken ermöglichen Klassenbibliotheken die Wiederverwendung von *Code*, lassen aber die Architektur der Anwendung offen. Der Benutzer kann über die Architektur entscheiden, kann die Klassen der Bibliothek selektiv einsetzen und hat zu entscheiden, wann er welche Funktionalität nutzen will. Im Unterschied zu Funktionsbibliotheken bieten Klassenbibliotheken aber zwei Vorteile:

- Indem jede Klasse eine Anzahl von Methoden und Datenelementen kapselt und Klassen in Vererbungshierarchien stehen können, sind Klassenbibliotheken i.a. besser strukturiert als vergleichbare Funktionsbibliotheken und damit besser zu überblicken.

- Klassenbibliotheken lassen sich durch Vererbung leicht erweitern und anpassen. Es müssen nur die veränderten Methoden neu erstellt werden, während die restlichen Methoden einfach „hinzugeerbt“ werden. Die Anpassungen sind in der neuen Klasse gekapselt (es wird also Lokalität gewährleistet). Weil eine durch Vererbung entstandene Klasse zu ihrer Basisklasse typkompatibel ist, kann sie auch in getypten Sprachen an Stelle der Basisklasse verwendet werden.

Klassenbibliotheken gibt es für die verschiedensten Einsatzzwecke. Sie können entweder fachlich (d.h. auf ein bestimmtes Anwendungsgebiet zugeschnitten) oder technisch (unabhängig vom Anwendungsgebiet) orientiert sein. Typische Vertreter von Klassenbibliotheken dienen technischen Zwecken:

- Bibliotheken für grundlegende Datentypen und -strukturen: Listen, Bäume, String-Klassen, Arithmetik etc. (horizontale Foundation Libraries); darunter insbesondere Container-Bibliotheken, die eine Hierarchie der gebräuchlichen Datenstrukturen zur Verfügung stellen (Listen, AVL-Bäume, Mengen, Hash-Tabellen, Stacks usw.)
- Programmentwicklung für bestimmte Umgebungen, insbesondere Bibliotheken für grafische Benutzeroberflächen (vertikale Foundation Libraries)
- Persistenz von Objekten, Datenbanken

Gerade die Bibliotheken zur Entwicklung grafischer Benutzeroberflächen (*Graphical User Interface*, GUI) haben eine große Bedeutung erlangt und stehen in einem engen Zusammenhang mit den objektorientierten Programmiersprachen. Beide Technologien sind zur selben Zeit populär geworden, und sicherlich hat der Bedarf an Entwicklungssystemen für GUI-Anwendungen zur Verbreitung von Smalltalk und C++ beigetragen: Im Falle von Smalltalk gab es schon früh eine grafische Entwicklungsumgebung, und die Erstellung grafischer Anwendungen war von Anfang an relativ einfach, weswegen Smalltalk auch beliebt ist für die Entwicklung von Prototypen. C++ hat davon profitiert, daß es früh Klassenbibliotheken gab, die die in C geschriebenen grafischen Benutzeroberflächschnittstellen kapselten und die Programmierung dieser Oberflächen deutlich erleichtert haben.

Fachliche Klassenbibliotheken gibt es für verschiedenste Anwendungsgebiete. Einige Bibliotheken sind allgemein gehalten (z.B. Klassen für Datum und Zeit), andere sind genau für ein bestimmtes Anwendungsgebiet entworfen. Mit zunehmender Spezialisierung kann eine Bibliothek natürlich in weniger Kontexten eingesetzt werden; daher lassen sich fachliche Klassenbibliotheken grundsätzlich weniger häufig wiederverwenden als technische.

In fachlichen Klassenbibliotheken sind die Zusammenhänge zwischen den verschiedenen Klassen komplexer als zwischen technisch motivierten Klassen. Technische Klassenbibliotheken stellen dadurch einen eher losen Verbund dar und können auch aus diesem Grund leicht wiederverwendet werden; denn

- einzelne Klassen können unabhängig vom Rest der Bibliothek benutzt werden, so daß also nur ein Teil der Bibliothek, nicht aber die Bibliothek als Ganzes den Anforderungen entsprechen muß, und/oder
- der Benutzer muß nur die Klassen verstehen, die er in seiner Anwendung benutzt, aber nicht die gesamte Bibliothek.

3.3 Entwurfsmuster

Entwurfsmuster (*design patterns*) beschreiben Lösungsmöglichkeiten, die für wiederkehrende Probleme verwendet werden können. In diesem Sinne sind Entwurfsmuster weder für objektorientierte Softwareentwicklung noch für Informatik spezifisch. Der Begriff entstammt der Arbeit des Architekten Christopher Alexander [Alexander et al. 77]. Martin Fowler charakterisiert Entwurfsmuster folgendermaßen:

„The definition I use for *pattern* is an idea that has been useful in one practical context and will probably be useful in others. I use the term idea to bring out the fact that a pattern can be anything ...]. It is often said that patterns are discovered rather than invented. This is true in the sense that models turn into patterns only when it is realized that they may have a common usefulness.“ [Fowler 97]

Während Klassenbibliotheken primär die Wiederverwendung von Code erlauben, werden durch Entwurfsmuster Entwurfsentscheidungen wiederverwendet. Wo Klassenbibliotheken Funktionalität für konkrete Probleme zur Verfügung stellen, beschreiben Entwurfsmuster die Lösungsmöglichkeiten auf einer abstrakten Ebene und bieten eine Art Schablone an, die sich auf eine Menge von konkreten Problemen anwenden läßt. Entwurfsmuster kommen damit dem Alltag der Softwareentwicklung sehr entgegen, wo immer wieder ähnliche Aufgabenstellungen auftreten.

Als typisches Beispiel stelle man sich ein Kalkulationsprogramm vor, in dem Daten vom Benutzer in eine Tabelle eingegeben und automatisch als Diagramme aufbereitet werden. Wenn ein Tabellenwert modifiziert wird, müssen alle Diagramme die Änderung berücksichtigen. Abstrahiert man von dem konkreten Problem, gelangt man etwa zu folgender Beschreibung: „Ein oder mehrere Programme oder Programmkomponeten müsen auf eine bestimmte Zustandsänderung reagieren.“ Ein klassisches Entwurfsmuster, das auf diese Situation anwendbar ist, ist als Beobachter (*Observer*) bekannt [Gamma et al. 95]; die Idee des Musters lautet etwa: „Verschiedene Objekte können sich als *Beobachter* bei dem Objekt, dessen Zustandsänderung sie berücksichtigen müssen, über eine Methode *MeldeAn* registrieren lassen. Alle Beobachter müssen über eine Methode *Aktualisiere* verfügen. Sobald die Zustandsänderung eintritt, ruft das beobachtete Objekt für alle angemeldeten Beobachter die jeweilige *Aktualisiere*-Methode auf.“

Die Problemstellung dieses Beispiels ist ebenso von der Verwendung objektorientierter Technologie unabhängig wie die Idee, die allgemein hinter Entwurfsmustern steht; es liegt also die Frage auf der Hand, warum die Softwareentwicklung Entwurfsmuster erst in Zusammenhang mit objektorientierter Programmierung für sich entdeckt hat.

Eine mögliche Antwort lautet: Die abstrakte Idee, die sich hinter jedem Entwurfsmuster verbirgt, kann in objektorientierten Sprachen ausgedrückt werden, indem man abstrakte Klassen-

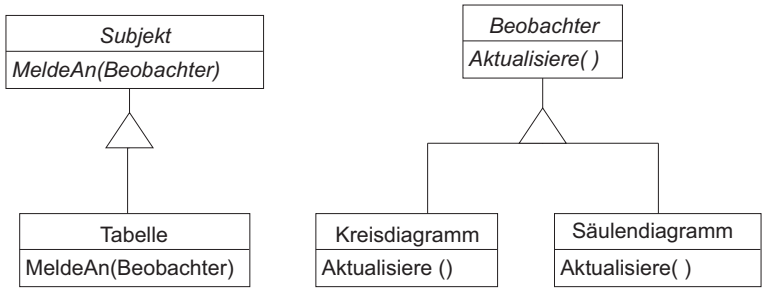


Abb. 3: Entwurfsmuster “Beobachter”

schnittstellen definiert. Für das Beobachter-Beispiel lassen sich abstrakte Klassen „Beobachter“ und (beobachtetes) „Subjekt“ definieren, in denen bereits die Existenz der Methoden *Aktualisiere* und *MeldeAn* festgelegt wird, ohne eine Implementierung dafür angeben zu müssen (vgl. Abbildung 3); Polymorphie stellt sicher, daß zur Laufzeit die richtigen *Aktualisiere-* und *MeldeAn*-Methoden für die Objekte der abgeleiteten Klassen ausgeführt werden. Indem objektorientierte Sprachen auf diese Art die Vererbung von Schnittstellen erlauben, werden viele Entwurfsmuster erst realisierbar.

Was ein Entwurfsmuster ist und was nicht, ist schwer zu sagen. Die Entscheidung darüber erfordert die Fähigkeit, Abstraktionen über den konkreten Problemen vornehmen zu können, und eine Menge praktische Erfahrung, um einzuschätzen, ob die Abstraktionen von allgemeiner Relevanz sind. Softwareentwickler können die Erfahrung anderer nutzen, indem sie auf katalogisierte Beschreibungen von Entwurfsmustern (z.B. [Gamma et al. 95]) zurückgreifen.

Der Umgang mit Entwurfsmustern bietet mehrere Vorteile:

- Die Softwareentwicklung wird beschleunigt, wenn man für ein bestimmtes Problem ein passendes Muster findet und angewendet, anstatt eine eigene Lösung für das Problem auszuarbeiten.
- Die Kommunikation unter Entwicklern wird erleichtert, wenn man sich auf bekannte Entwurfsmuster beziehen kann.
- Die Architektur von Systemen kann durch Entwurfsmuster strukturiert werden.
- Die Beschreibung von Systemen läßt sich durch Verweise auf Muster vereinfachen; unter der Voraussetzung, daß die Muster bekannt sind, lassen sich komplexe Systeme dann schneller erfassen.

Entwurfsmuster treten in vielen Fällen nicht isoliert, sondern zusammen mit anderen Mustern auf. Bestimmte Entwurfsmuster werden fast immer in Kombination mit bestimmten anderen Mustern eingesetzt.

3.4 Frameworks

Ein Framework bietet ein Gerüst zur Erstellung von Anwendungen. Dazu enthält es eine Menge kooperierender Klassen und legt Strukturen für die Architektur der Anwendungen fest.

„A framework is a class library that captures patterns of interaction between objects. A framework consists of a suite of concrete and abstract classes, explicitly designed to be used together. Applications are developed from a framework by completition of the implementations of the abstract classes. A framework can also include additional utilities to aid in the completition of end-user applications. A utility can be a code generator or algorithm, for example.“ [Rogers 97]

Frameworks gehen damit in zweifacher Hinsicht über Klassenbibliotheken hinaus:

- Der Zusammenhang zwischen den Klassen ist enger,
- das Framework bestimmt die Architektur der Anwendung.

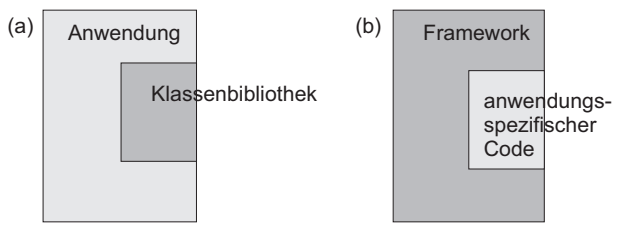


Abb. 4 : Klassenbibliothek vs. Framework

Im Extremfall schreibt der Entwickler nicht mehr seine Anwendung, in die er Teile der Bibliothek einfügt, sondern entwickelt anwendungsspezifischen Code (häufig durch Vererbung von abstrakten Framework-Klassen), der in das Framework eingefügt wird (vgl. Abbildung 4).

Entwurfsmuster werden von vielen Frameworks benutzt, um die Beziehungen zwischen den einzelnen Klassen des Frameworks auszudrücken. Insbesondere bietet es sich an, Entwurfsmuster an denjenigen Schnittstellen einzusetzen, an denen anwendungsspezifischer Code in das Framework eingesetzt werden muß. Die Nutzung von Entwurfsmustern kann für den Benutzer des Frameworks verständlicher machen, in welchem Zusammenhang die Klassen stehen, die er zur Vervollständigung des Frameworks entwickeln muß.

Ebenso wie Klassenbibliotheken können auch Frameworks technisch oder fachlich orientiert sein. Technische Frameworks sind eher horizontal angelegt, d.h. sie lassen sich in vielen Kontexten benutzen [Rogers 97]. Fachliche Frameworks sind dagegen vertikal: Ihr Einsatzkontext ist sehr eng gefaßt, so daß sie sehr viel seltener wiederverwendet werden können. Dafür sind sie aber so speziell zugeschnitten, daß sie bereits einen großen Teil der zu erstellenden Anwendung beinhalten.

Ob ein Entwickler, der ein Framework einsetzt, die internen Details (z.B. die Implementierung und interne Klassen) des Frameworks kennen muß, hängt von unterschiedlichen Gegebenheiten ab. Sogenannte *Black-Box-Frameworks*, bei denen die Kenntnis der externen Schnittstellen ausreicht, sind leichter zu benutzen, aber weniger flexibel; dem gegenüber stehen *White-Box-Frameworks*, die leichter an neue Anforderungen anzupassen, aber schwerer zu erlernen sind. Die meisten Frameworks sind vom Typ White-Box, enthalten aber auch Black-Box-Komponenten. Mit voranschreitender Evolution eines Frameworks werden gewöhnlich mehr und mehr Details vor dem Benutzer verborgen, so daß eine Entwicklung in Richtung Black-Box-Framework stattfindet [Yourdon et al. 95].

Für viele Anwendungen besteht der Bedarf, mehr als nur ein Framework einzusetzen. Auch wenn Frameworks Annahmen über bzw. Vorgaben für die Architektur der Anwendung machen, sollte sichergestellt sein, daß sich diese Annahmen und Vorgaben mit denen anderer Frameworks, die gleichzeitig zum Einsatz kommen können, vereinbaren lassen.

Frameworks bringen große Chancen, aber auch Risiken mit sich. Ihr Potential besteht darin, in Anwendungen Entwurf *und* Code aus dem Framework zu erben. Sie vereinigen damit die Vorteile von Klassenbibliotheken und von Entwurfsmustern und können die Entwicklungszeit von Anwendungen entscheidend verkürzen. Dies gilt besonders bei der Verwendung von vertikalen und von ausgereiften Black-Box-Frameworks.

Die Risiken, die mit der Entwicklung von Frameworks einhergehen, sind:

- Frameworks können zu komplex und damit zu schwer verständlich werden.
- Zu groß angelegte Frameworks können den Benutzer zwingen, Teile in seine Anwendung aufzunehmen, die nicht nötig oder sogar störend sind.

Frameworks neigen dazu, einen Absolutheitsanspruch zu entwickeln, mit dem sich andere Frameworks nicht vereinbaren lassen. Diese Problematik wird in Kapitel 7 aufgegriffen, wenn es um den Entwurf eines eigenen Frameworks geht.

3.5 Zusammenfassung

Zu Beginn des Kapitels wurden einige grundlegende Eigenschaften von objektorientierten Programmiersprachen genannt: Vererbung, Polymorphie, Lokalität und Datenabstraktion. Dank dieser Eigenschaften eignen sich objektorientierte Programmiersprachen sehr gut für die Erstellung wiederverwendbarer Komponenten. Drei wesentliche Konzepte der Wiederverwendung sind Klassenbibliotheken, Entwurfsmuster und Frameworks: Durch Klassenbibliotheken werden vor allem Implementierungen wiederverwendet, durch Entwurfsmuster in erster Linie Architekturprinzipien. Frameworks vereinigen beides, indem sie sowohl die Wiederverwendung von Architekturentwürfen als auch auch von implementiertem Code realisieren.

Die Inhalte dieses Kapitels sind für den weiteren Verlauf der Arbeit in zweifacher Hinsicht grundlegend. Einerseits wird das folgende Kapitel deutlich machen, daß Wiederverwendbarkeit durch Softwarevisualisierung unterstützt werden kann. Zum anderen wird der Entwurf eines Frameworks in Kapitel 7 besprochen.

Kapitel 4
Konzept eines Visualisierungssystems

Die vorangegangen Kapitel dienten dazu, die Grundlagen von Programmverstehen, von Softwarevisualisierung und von den Möglichkeiten objektorientierter Wiederverwendung vorzustellen. In diesem Kapitel erfolgt nun der Brückenschlag zwischen den Teilbereichen. Zunächst werden Motivation und Ziele eingegrenzt, und anschließend wird ein Konzept entworfen, nach dem sich Werkzeuge für die Visualisierung objektorientierter Programme erstellen lassen. In den weiteren Kapiteln werden die Bestandteile des vorgestellten Konzeptes detaillierter beschrieben.

4.1 Motivation und Zielbestimmung

Aus welchen Gründen möchte man das Verhalten objektorientierter Programme visualisieren? Die Vorgeschichte dieser Arbeit beginnt mit konkreten, praktischen Fragen, die während der Entwicklung größerer objektorientierter Anwendungen auftraten. In den folgenden Abschnitten werden einige dieser Fragen in zwei Beispielen aufgegriffen. Die Beispiele vermitteln die Motivation des Themas und tragen wesentlich dazu bei, die Anforderungen an das Visualisierungswerkzeug formulieren zu können.

4.1.1 Beispiel: Späte Erzeugung

Die Beschäftigung mit dem Thema Softwarevisualisierung wurde am Arbeitsbereich Softwaretechnik der Universität Hamburg unter anderem durch eine bestimmte Fragestellung motiviert: In einem am Arbeitsbereich entwickelten Framework [Lilienthal 97] wird das Entwurfsmuster *Späte Erzeugung* benutzt, um Objekte prototypisch zu generieren [Riehle und Züllighoven 94]. Das Entwurfsmuster erfordert, daß für alle Klassen, von denen Objekte erzeugt werden sollen, sowie für deren Basisklassen zu Beginn der Programmausführung Objekte als Prototypen generiert werden. Während in einer Klasse generell nicht bekannt ist, welche von ihr abgeleiteten Klassen es gibt, „weiß“ der Prototyp einer Klasse, welche Prototypen die abgeleiteten Klassen repräsentieren. Die sogenannte späte Erzeugung von Objekten erfolgt, indem zur Laufzeit ein Prototyp angewiesen wird, ein Objekt einer bestimmten, abgeleiteten Klasse zu generieren. Von welcher Klasse das Objekt sein soll, wird in einer Klausel beschrieben. Der beauftragte Prototyp muß daher in der Lage sein, ein zur Klausel passendes Objekt selbst zu erzeugen, oder er delegiert diese Aufgabe an die Prototypen, die für abgeleitete Klassen erzeugt wurden. Der Nutzen des Entwurfsmusters besteht darin, daß zur Compilezeit nur eine Oberklasse bekannt

sein muß, von der eine Instanz erzeugt werden soll. Über den konkreten Typ wird zur Laufzeit anhand der Klausel entschieden.

Das gerade beschriebene Entwurfsmuster wird an mehreren Stellen des Frameworks benutzt und hat sich bewährt. Dennoch hat es zu Fragen und Problemen geführt:

- Didaktisch: Die Erzeugung von Objekten erfolgt auf nichttriviale Art. Wie erklärt man das zugrunde liegende Prinzip den Benutzern des Frameworks?
- Anzahl der Prototyp-Objekte: Wenn man ein Programm, in dem das Framework benutzt wird, startet, werden zunächst die Prototypen-Objekte erzeugt. Wie hoch ist die Anzahl der Prototypen? Entwickler, die mit dem Framework vertraut sind, konnten diese Zahl nur schwer abschätzen, weil die Prototypen nicht explizit im Quelltext auftreten, sondern implizit durch Makros entstehen.
- Erzeugung: An wie viele Prototyp-Objekte wird eine Klausel delegiert, bis der richtige Prototyp gefunden ist, um das gewünschte Objekt zu erzeugen?

Alle drei Punkte zielen auf das dynamische Verhalten, das durch die späte Erzeugung entsteht (wohingegen die statische Architektur von Framework und Anwendungsprogramm nicht nennenswert durch das Muster beeinflußt wird). Um die Fragen zu beantworten, könnte man versuchen, das Verhalten anhand des Quelltextes nachzuvollziehen. Ein solches Vorgehen ist gerade für dieses Beispiel extrem mühselig, weil die Erzeugung der Prototypen nicht lokal in einem begrenzten Teil des Quelltextes erfolgt, sondern komplett auf den Programmcode von Framework und Anwendung verteilt ist.

4.1.2 Beispiel: Dokumentation von Frameworks

Im Beispiel des Späte-Erzeugung-Musters traten bereits zwei Rollen auf: Die Rolle des Entwicklers, der ein Framework erstellt und dazu unter anderem Entwurfsmuster verwendet, und die Rolle des Benutzers, der das Framework für eigene Applikationen einsetzt. Frameworks besitzen das charakteristische Konzept, nach dem der Benutzer des Frameworks Klassen implementiert, deren Methoden nicht sein Code, sondern der Code des Frameworks aufruft.

Im Fall des am Abreitsbereich entwickelten Frameworks gilt das insbesondere für die Klassen, in denen Funktions- und Interaktionskomponenten [Kilberth et al. 94] programmiert sind und denen damit wesentliche Anteile an der Programmausführung zukommen. Diese Klassen erstellt der Benutzer des Frameworks; instanziiert werden sie aber durch Framework-eigenen Code.

Wie macht man Benutzern des Frameworks plausibel, welche Methoden der anwendungsspezifischen Klassen zu welchem Zeitpunkt aufgerufen werden? Natürlich müssen die Entwickler ihr Framework dokumentieren und die Dokumentation an die Benutzer ausliefern. Neben den textuellen Beschreibungen der Frameworkklassen werden häufig Grafiken verwendet, die die Zusammenhänge zwischen den Klassen illustrieren. Ein Visualisierungswerkzeug könnte dazu wertvolle Hilfe leisten, wenn die Entwickler damit interaktiv Diagramme für Ausschnitte des Systemverhaltens und der Systemstruktur erzeugen können und die Diagramme in die Benutzerdokumentation einbringen [Lilienthal und Strunk 96].

Wenn die Frameworkentwickler Diagramme für Benutzer erstellen, wollen sie damit gezielt bestimmte Merkmale ihres Frameworks erklären. Es kann daher nicht ausreichen, daß Diagramme von einem Visualisierungswerkzeug erstellt und unverändert in die Dokumentation eingefügt

werden, sondern es wird der Wunsch bestehen, die Diagramme zu vereinfachen, mit Anmerkungen und Erklärungen zu versehen, bestimmte Elemente hervorzuheben usw.

Wenn etwa ein Objekt des Frameworks durch späte Erzeugung entsteht, dann soll in der Dokumentation vernachlässigt werden, welche Folge von Methodenaufrufen zwischen verschiedenen Prototyp-Objekten zustande kommt, bis die angeforderte Klausel interpretiert und das gewünschte Objekt erzeugt ist (Abbildung 5a) – für den Benutzer möchte man vermutlich nur in einem Diagramm darstellen, daß ein bestimmtes Objekt durch das Muster „Späte Erzeugung“ generiert wurde (Abbildung 5b).

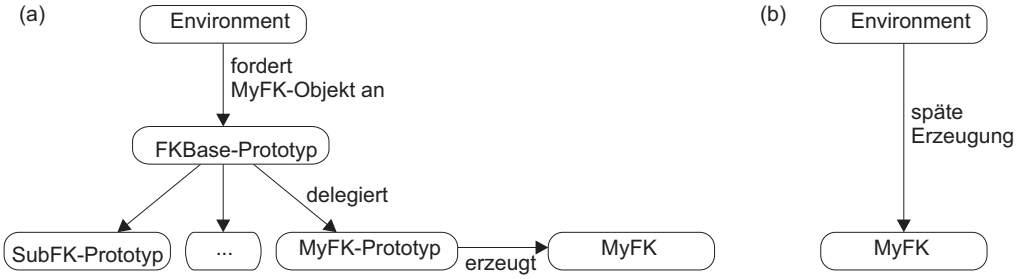


Abb. 5: (a) Späte Erzeugung als Objektdiagramm, (b) vereinfachtes Diagramm

4.1.3 Folgerungen aus den Beispielen

Die geschilderten Probleme helfen, Anforderungen an das Visualisierungswerkzeug zu konkretisieren. Ich möchte die Beispiele unter zwei Aspekten betrachten:

- Welcher Abstraktionsebene sind die Informationen, die das Werkzeug erschließen soll, zuzuordnen?
- Welches „Profil“ zeichnet die potentiellen Benutzer des Werkzeugs aus, und welche Anforderungen entstehen daraus?

Ich versuche, die Fragen in den beiden folgenden Abschnitten zu beantworten.

4.1.3.1 Zielebene: Objektorientierte Konzepte

Beide Beispiele drücken das Bedürfnis aus, das Verhalten von Programmen bzw. Frameworks zugänglich zu machen. Sie beinhalten konkrete Fragen wie:

- Wieviele Objekte einer Klasse werden erzeugt?
- Wann werden die Objekte erzeugt? Wann werden sie später benutzt?
- Wie interagieren sie miteinander?
- Welche Entwurfsmuster werden benutzt? Was bewirken die Entwurfsmuster?

Offensichtlich geht es darum, einen Überblick über bestehende Programme zu erhalten und ein grundlegendes Verständnis zu erlangen. Die Fragen bewegen sich auf einer Abstraktionsebene, auf der Objekte im Mittelpunkt stehen. Dies ist ein wesentlicher Unterschied zu herkömmlichen Programmierwerkzeugen (z.B. Debugger), die prinzipiell für dieselben Anforderungen gedacht sind, aber nicht auf der Ebene von Objekten, sondern auf der von Quelltextanweisungen arbeiten. So wie Daniel Ingalls ein objektorientiertes Programm als „a universe of well-behaved objects that courteously ask each other to carry out their various desires“ [Ingalls 81] beschreibt,

das an die Stelle eines „bit-grinding processor [that is] raping and plundering data structures“ tritt, möchte ich mit dieser Arbeit den Schritt weg von den prozessornahen Debuggern und hin zu Werkzeugen vollziehen, die das Universum der miteinader kommunizierenden Objekte erfassen.

Ziel dieses Kapitels ist es, ein fachliches Konzept für ein Werkzeug zu entwickeln, das das Verhalten objektorientierter Programme auf der Ebene von Objekten und von höheren Entwurfskonstrukten grafisch darstellt. In welcher Form die grafische Darstellung erfolgen soll, ist noch zu klären; wichtiger ist es zu bestimmen, *was* das Werkzeug anzeigen soll. *Wie* die Anzeige gestaltet wird, steht erst an zweiter Stelle.

Nach den genannten Fragestellungen muß das Visualisierungswerkzeug folgendes leisten:

- Während ein objektorientiertes Programm ausgeführt wird, zeigt das Werkzeug die Objekte an, die erzeugt werden. Dementsprechend soll auch die Destruktion der Objekte angezeigt werden.
- Das Werkzeug zeigt an, wie die Objekte miteinander interagieren, d.h. wie sie wechselseitig Methoden aufrufen.
- Das Werkzeug ermöglicht es, höhere Konzepte (wie Entwurfsmuster) in der grafischen Darstellung auszudrücken.

Das Werkzeug soll aber nicht auf unnötige Details der Objekte und ihrer Dienste eingehen. Es soll in der Lage sein, die Schnittstellen von Klassen und die Attributewerte von Objekten anzuzeigen, aber es soll nicht die einzelnen Anweisungen zeigen, die Objekte in ihren Methoden ausführen. Methoden werden somit zu unteilbaren Gebilden innerhalb der Programmausführung.

Natürlich gibt es Fragen, die nur beantwortet werden können, indem die Implementierung einer Methode genau untersucht wird. Diese Fragen zählen aber zu einer anderen Kategorie, z.B. zur Fehlersuche, aber nicht zum Verstehen der „großen“ Programmkonzepte. Sie lassen sich mit herkömmlichen Debuggern viel besser lösen oder anhand des Quelltextes beantworten. Die Integration eines Debuggers und eines Quelltexteditors ist zu diesem Zweck wünschenswert.

4.1.3.2 Zielgruppe: Programmentwickler

Die Fragestellungen, die genannt wurden, betreffen Softwareentwickler und Programmierer. Sie kommen als Benutzer von Visualisierungswerkzeugen in Frage, um Erkenntnisse über ihren eigenen Code zu erlangen, um sich in fremde Programme oder Komponenten einzuarbeiten, oder um eigene Programme anderen Entwicklern verständlich zu machen. In jedem Fall: Die potentiellen Benutzer sind Fachleute, bei denen ein vorhandenes Fachwissen vorausgesetzt werden darf. Anders als Programmieranfänger, für die Visualisierungstechniken didaktischen Wert haben können, besitzen sie professionelle Ansprüche: Ein Werkzeug muß flexibel, effizient und verläßlich sein (vgl. S. 14).

Die Forderung nach Verläßlichkeit bedeutet, daß die Implementierung Robustheit und Korrektheit sicherstellen muß. Die Effizienz des Werkzeugs muß in zweierlei Hinsicht gewährleistet sein: Erstens muß es effizient im Sinne von performant sein, und zweitens heißt das, daß die gewünschten Ergebnisse schnell und einfach erzeugt werden können. Flexibilität bedeutet schließlich, daß das Werkzeug an unterschiedliche Problemstellungen und Anwendungskontexte anpaßbar sein muß.

Diese Forderungen verlangen nach einem offenen und flexiblen System. Die Architektur des Systems muß auf Änderbarkeit und Erweiterbarkeit ausgelegt sein, und die Benutzerschnittstelle muß die unterschiedlichen Anforderungen berücksichtigen. Ein zentrales Element der Benutzerschnittstelle werden die Diagramme sein, die das Visualisierungswerkzeug erzeugt. Gerade vor dem Hintergrund vielfältiger Anforderungen muß man sich bewußt sein, daß es keine Form der Visualisierung gibt, die für alle Einsatzzwecke und Fragestellungen gleichermaßen geeignet ist, und daß Visualisierungswerkzeuge eine Ergänzung, aber kein Ersatz für andere Programmierwerkzeuge sind.

Um diesen Argumenten gerecht zu werden, sollte das Visualisierungswerkzeug

- nicht nur eine Art von Diagrammen erstellen können, sondern mehrere Arten anbieten,
- in den verfügbaren Diagrammen an die Bedürfnisse des Benutzers anpaßbar sein,
- erweiterbar sein, so daß das Werkzeug später um zusätzliche Diagramme ergänzt werden kann,
- nach Möglichkeit mit anderen, bereits vorhandenen Programmierwerkzeugen integrierbar sein.

Der letzte Punkt, die Integrierbarkeit mit anderen Werkzeugen, hängt natürlich davon ab, ob andere Werkzeuge überhaupt eine (verfügbare und dokumentierte) Schnittstelle besitzen.

4.2 Erfassen von Entwurfskonzepten

Im zweiten Kapitel wurde bereits das Kriterium „Abstraktion“ benutzt, um Werkzeuge zur Softwarevisualisierung zu kategorisieren. Im folgenden wird nun beschrieben, wie allgemeine Abstraktionen bei der Visualisierung objektorientierter Programme berücksichtigt werden können.

4.2.1 Implementierungs- und Modellierungsebene

Unter den Fragen, die durch Softwarevisualisierung beantwortet werden sollen, kann man unterscheiden zwischen Fragestellungen, die die Modellierungsebene betreffen, und solchen, die sich auf die Implementierungsebene beziehen. Zusammen mit der realen Welt der Anwendungsdomäne spannen die beiden Ebenen ein Dreieck auf, innerhalb dessen Softwareentwicklung stattfindet: Die Anforderungsermittlung untersucht die Welt der Anwendungsdomäne, und die dabei gewonnenen Erkenntnisse werden benutzt, um auf der Modellierungsebene einen Entwurf auszuarbeiten. Auf der Implementierungsebene wird der Entwurf in Programmtext umgesetzt. Der Programmtext (bzw. das übersetzte und ablaufende Programm) ist schließlich zu validieren, indem geprüft wird, ob sich das Programm konform zur Anwendungsdomäne verhält.

Die Dokumente, die Werkzeuge zur Softwarevisualisierung als Input benutzen, sind Quelltexte oder compilierte Programme und gehören damit zur Implementierungsebene. Auf dieser Ebene können nicht mehr Informationen gewonnen werden, als sich mit dem Symbolvorrat der verwendeten Programmiersprache ausdrücken lassen. Für eine objektorientierte Programmiersprache sind das z.B. die Informationen, welche Klassen im Programm verfügbar sind und aus welchen Anweisungsfolgen deren Methoden bestehen. Aus den Anweisungen geht auch hervor, wann Objekte erzeugt werden und wie Objekte miteinander interagieren.

In den vorangegangenen Abschnitten tauchten aber auch Begriffe wie „Entwurfsmuster“ und „Framework“ auf, die sich auf die Modellierungsebene beziehen. Wenn ein Visualisierungswerkzeug Daten der Implementierungsebene als Eingabe verarbeitet und Informationen über die Modellierungsebene verfügbar macht, liegt ein echter Reverse-Engineering-Schritt vor.

Die Verbindung von objektorientierten Entwurfsmethoden und objektorientierten Programmiersprachen erlaubt es, relativ bruchlos von der Modellierungsebene auf die Implementierungsebene überzugehen: Die Klassen, die im Entwurf modelliert wurden, lassen sich mit ihren Methoden und Attributen direkt auf die Implementierungsebene abbilden.

Dennoch ist die Abbildung nicht verlustfrei. Im Übergang von der Modellierungs- auf die Implementierungsebene geht vor allem Wissen über die Architektur des Systems verloren (zum Architekturbegriff vgl. [Floyd und Züllighoven 97]). Bestimmte Aspekte der Architektur treten zwar auch auf der Implementierungsebene hervor, z.B. die Zerlegung des Systems in Module (was in objektorientierten Systemen in der Regel den unterschiedlichen Klassen entspricht). Andere Gesichtspunkte der Architektur, die sich nur implizit in der Implementierung wiederfinden lassen, sind

- Architekturprinzipien, die dem Entwurf zugrunde liegen und konzeptuelle Zusammenhänge zwischen Klassen oder semantische Eigenschaften beschreiben; dazu gehören z.B. Entwurfsmuster und Entwurfsmetaphern;
- Organisationsformen, durch die Architektur strukturiert wird, z.B. durch Verwendung von Programmbibliotheken und Frameworks.

Die Architekturmerkmale eines bestimmten Systems zu kennen ist eine wesentliche Grundlage, um das System verstehen zu können.

4.2.2 Wiederverwendbare Elemente

Der Einsatz von wiederverwendbaren Elementen (vgl. Kapitel 3) findet sowohl auf Modellierungs- als auch auf Implementierungsebene statt. Wo Klassen aus Klassenbibliotheken oder Frameworks benutzt werden, erfolgt unmittelbar eine Wiederverwendung der Implementierung. Allerdings wird auf dieser Ebene die Architektur der Bibliothek oder des Frameworks ebenso wenig vollständig erfaßt, wie es auch für die Architektur des Gesamtsystems gilt. Entwurfsmuster stellen architektonische Prinzipien dar, die auf der Modellierungsebene angewendet werden und die auf der Implementierungsebene so nicht zu finden sind (schließlich zeichnen Muster sich ja dadurch aus, daß dieselbe Entwurfsidee in unterschiedlichen Implementierungen auftritt).

Welcher Zusammenhang besteht nun zwischen Softwarevisualisierung und Wiederverwendung? Konzepte oder Code müssen verstanden werden, bevor man sie wiederverwendet, und Programvisualisierung soll das Verstehen unterstützen. Daher ist von einem Visualisierungswerkzeug zu erwarten, daß sich damit Diagramme erstellen lassen, in denen wiederverwendbare Elemente kenntlich gemacht sind. Ob die Kennzeichnung automatisch oder interaktiv vorgenommen werden kann, ist eine andere Frage.

Zunächst wird ein Visualisierungswerkzeug, das das Verhalten eines objektorientierten Programms darstellt, nichts anderes anzeigen als Objekte (evtl. mit ihren Attributwerten) und Nachrichten, die während der Programmausführung von Objekt zu Objekt gesendet werden. Wie verändert sich die Darstellung, wenn man Informationen über Klassenbibliotheken, Entwurfsmuster und Frameworks einbeziehen will?

Im Fall der Klassenbibliotheken möchte man sicherlich hervorheben, welche Objekte von Klassen einer bestimmten Bibliothek instanziiert sind, d.h., man versieht die entsprechenden Objekte im Diagramm mit einer textuellen Anmerkung oder einem grafischen Symbol, oder man verändert die Anordnung der Objekte, so daß die Objekte einer Bibliothek zueinander benachbart sind. Es verändert sich damit nicht, *was* angezeigt wird, sondern nur, *wie* es angezeigt wird.

Anders sieht es für Entwurfsmuster aus: In Abschnitt 4.1.2 wurde am Beispiel der späten Erzeugung vorgeschlagen, wie man in einem Diagramm statt einer Folge einzelner Nachrichten nur noch ein Ereignis für das Auftreten eines Muster anzeigen kann. Der Inhalt des Diagramms wird also verändert, indem bestimmte Elemente entfernt und dafür andere aufgenommen werden. Es ist aber davon auszugehen, daß ein Benutzer des Werkzeugs beide Darstellungen benutzen möchte und z.B. bei Bedarf das angezeigte Muster zu der „eigentlichen“ Nachrichtensequenz expandieren will. Demnach darf das eine Diagrammelement, Muster, nicht die anderen Elemente (die einzelnen Nachrichten) unwiderruflich ersetzen, sondern beide Elemente sind gleichzeitig in einem fachlichen Modell des Diagramms vorhanden und sind miteinander verknüpft. Dargestellt wird aber immer nur eine der beiden Varianten.

Frameworks sind ein spezieller Fall von Klassenbibliotheken; daher gilt hier gleichermaßen, was bereits für Bibliotheken gesagt wurde. Außerdem werden in vielen Frameworks Entwurfsmuster verwendet, die in Diagrammen dargestellt werden sollen. Ein anderes Beispiel, wie ein Diagramm Frameworks berücksichtigen kann: Wenn in einer Applikation ein Black-Box-Framework (S. 26) verwendet wird, liegt es nahe, das Framework auch tatsächlich als „Black Box“ darzustellen, d.h. auf die Anzeige der einzelnen Objekte, ihrer Verknüpfungen und Interaktionen zu verzichten und stattdessen ein einziges Symbol zu zeigen. Hier gilt dasselbe wie bei Entwurfsmustern: Das Diagramm sollte auf einem Modell basieren, das sowohl die einzelnen Objekte des Frameworks als auch das Black-Box-Konstrukt kennt.

4.2.3 Elemente 1. und 2. Ordnung

Die allgemeinen Aufgaben eines Visualisierungswerkzeugs sind es, Informationen über ein visualisiertes Programm zu erfassen, diese Informationen zu verwalten und sie grafisch zu präsentieren. Wenn wir erneut den Fall eines Entwurfsmusters nehmen, heißt das: Das Werkzeug erfaßt eine Folge von einzelnen Nachrichten. Daraus entsteht ein Diagramm, in dem die Nachrichten entweder als Folge oder zusammengefaßt zu einem Muster angezeigt werden.

Da der Benutzer die Möglichkeit haben soll, zwischen beiden Präsentationen zu wechseln, muß das Werkzeug dauerhaft sowohl die einzelnen Nachrichten als auch die Repräsentation als Muster speichern, auch wenn jeweils nur eine der beiden Varianten Bestandteil des erzeugten Diagramms ist. Das bedeutet:

- Das Visualisierungswerkzeug muß mehr Informationen verwalten, als im Diagramm dargestellt werden.
- Die verwalteten Informationen konkurrieren z.T. miteinander, indem sie dieselben Sachverhalte auf unterschiedlichen Abstraktionsebenen repräsentieren (nämlich Implementierungs- versus Modellierungsebene).

Ich möchte daher den Datenbestand, den das Visualisierungswerkzeug verwaltet, nach Elementen erster und zweiter Ordnung unterscheiden:

- Elemente erster Ordnung repräsentieren Informationen über das analysierte Programm auf der Ebene, auf der sie vom Visualisierungswerkzeug gewonnen werden. Für

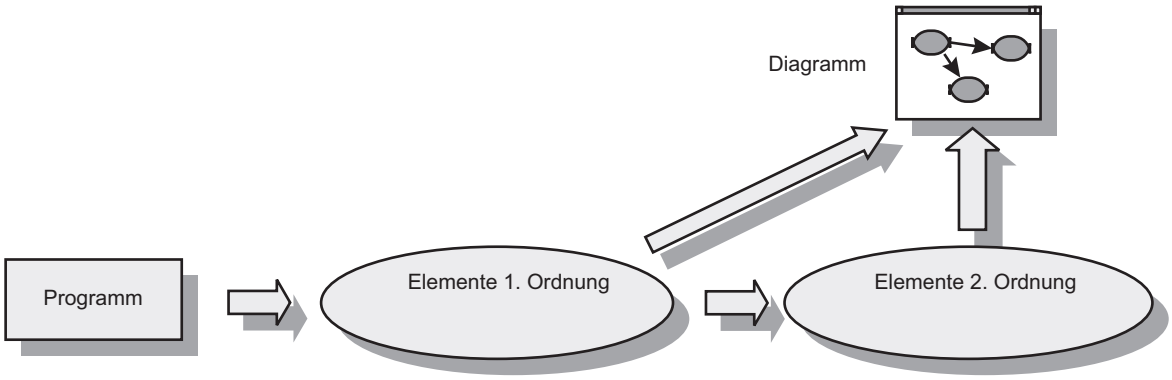


Abb. 6: Diagramme auf Basis von Elementen 1. und 2. Ordnung

objektorientierte Programme sind dies die objektorientierten Konstrukte, die mit den Mitteln der benutzten Programmiersprache ausgedrückt werden können, nämlich Klassen, Objekte, Methoden und Methodenaufrufe.[1] Elemente erster Ordnung stehen notwendigerweise auf der Implementierungsebene.

- Elemente zweiter Ordnung drücken Zusammenhänge zwischen oder Bedeutungen von anderen Elementen (erster oder zweiter Ordnung) aus. Sie stehen für (objektübergreifende) Konzepte, die auf der Modellierungsebene oder der Implementierungsebene verwendet wurden. In der Regel werden sie andere Elemente aggregieren. Die Menge der Elemente erster Ordnung wird durch die Elemente zweiter Ordnung strukturiert, denn letztere stehen für die strukturierenden Architekturkonzepte, die im untersuchten Programm verwendet wurden.

Das Visualisierungswerkzeug erzeugt, während es die Ausführung eines Programms beobachtet, Informationen erster Ordnung. Darauf aufbauend können dann Informationen zweiter Ordnung angelegt werden. Der Benutzer sieht ein Diagramm, in dem beide Formen repräsentiert sind (Abbildung 6).

Elemente zweiter Ordnung kann man nicht eindeutig oder vollständig benennen, da die Architektur von Software unterschiedlich strukturiert sein kann, und so gibt es auf der Modellierungsebene verschiedene Ansätze; auch auf der Implementierungsebene können verschiedene Konzepte auftreten, wie die folgende Liste möglicher Elemente zweiter Ordnung zeigt:

- Element „Klassenbibliothek“: faßt alle Klassen der Bibliothek zusammen sowie alle Objekte, die von den Klassen instanziiert werden.
- Element „Subsystem“: faßt bestimmte Objekte zusammen, die zur Laufzeit eine konzeptuelle Einheit darstellen.
- Element „Entwurfsmuster“: faßt mehrere Methodenaufrufe, eventuell auch Objekte zusammen.
- Element „Prozeß“: faßt alle Objekte zusammen, die bei der Ausführung eines verteilten Programms in einem Prozeßraum angesiedelt sind.

1 Eine Methode ist ein statisches Konstrukt, Methodenaufrufe (synonym: Nachrichten) sind dynamisch

Die Liste kann beliebig fortgeführt werden; für diese Arbeit ist es nicht entscheidend, welche Elemente zweiter Ordnung man definiert, sondern daß die Architektur des Visualisierungswerkzeugs es zuläßt, auf die Elemente erster Ordnung eine zweite Kategorie aufzusetzen.

4.3 Arten von Diagrammen

Es gibt eine Vielzahl von Darstellungstechniken, um Aspekte objektorientierter Programme in Diagrammen abzubilden. Die Darstellungstechniken haben zwei mögliche Ursprünge:

- Sie stammen aus dem Repertoire einer objektorientierten Entwurfsmethode oder
- wurden für Systeme zur Softwarevisualisierung entwickelt.

Diagramme, die als Bestandteile von Entwurfsmethoden auftreten, dienen zunächst einmal dem Forward Engineering. In der Regel werden sie von Softwareentwicklern per Hand oder am Bildschirm gezeichnet. Solche Diagramme für die Softwarevisualisierung zu benutzen bedeutet, sie (anhand abstrakter Dokumente wie des übersetzten Programms oder des Quelltexts) durch Visualisierungssoftware generieren zu lassen (Reverse Engineering).

Während die in den Entwurfsmethoden benutzten Diagramme zweidimensional sind (vgl. Kap. 2), arbeitet man in der Softwarevisualisierung häufig mit animierten Diagrammen, in denen in Form der Zeit eine dritte Dimension verwendet wird.

4.3.1 Diagramme der UML

Auch wenn es eine hohe Zahl von Entwurfsmethoden mit jeweils eigenen Diagrammen und Notationen gibt, kann man die Menge aller Darstellungstechniken auf wenige Grundtypen reduzieren. Der zur Zeit vollständigste Katalog ist durch die Zusammenarbeit von Grady Booch, James Rumbaugh und Ivar Jacobson entstanden, die ihre Ansätze in der *Unified Modeling Language* (UML, siehe [Rational 97]) vereint haben. Die UML umfaßt alle relevanten Diagrammtypen des objektorientierten Entwurfs:

- *Klassendiagramme* zeigen die statische Struktur des Entwurfs, namentlich die existierenden Klassen, deren Aufbau und deren Beziehungen untereinander. *Objektdiagramme* zeigen einzelne Instanzen und deren Beziehungen. Sie sind dazu gedacht, Ausschnitte der Systemstruktur zu einem Zeitpunkt der Programmausführung zu beschreiben, etwa um Beispiele für die Verkettung von Datenstrukturen zu zeigen. Klassen- und Objektdiagramme sind nicht eindeutig voneinander getrennt und können miteinander vermischt werden, indem Klassendiagramme Objekte enthalten.
- *Use-Case-Diagramme* stellen Zusammenhänge zwischen *Use Cases* und *Aktoren* dar. Ein Use Case ist ein zusammenhängender Teil der Systemfunktionalität. Aktoren stehen für Rollen, die von Personen und Objekten außerhalb des Systems eingenommen werden und die mit dem System interagieren. Welche Aktoren mit welchen Use Cases zusammenhängen, wird in Use Case Diagrammen dargestellt (siehe auch [Jacobson et al. 92]).
- *Sequence diagrams* zeigen die Interaktion von Objekten entlang einer Zeitachse. In ihnen werden Nachrichten, aber keine Assoziationen zwischen Objekten berücksichtigt.
- *Collaboration diagrams* stellen ebenfalls Interaktionen zwischen Objekten dar und enthalten zusätzlich deren Assoziationen. Im Gegensatz zu *sequence diagrams* zeigen sie keine Zeitverläufe.

- *Zustandsdiagramme (statechart diagrams)* geben Zustandsübergänge innerhalb eines Objekts an. In ihnen wird modelliert, wie ein Objekt auf Ereignisse, die von außen eintreffen, reagiert.
- *Aktivitätsdiagramme (activity diagrams)* sind ähnlich wie Zustandsdiagramme, stellen aber keine Abfolge von Zuständen, sondern von Tätigkeiten dar. In ihrer Idee und Notation sind sie an Petrinetze angelehnt. Sie eignen sich, um die einzelnen Arbeitsschritte darzustellen, die innerhalb einer Methode erfolgen.
- *Implementationsdiagramme (implementation diagrams)* gibt es in zwei Ausprägungen: *Komponentendiagramme* stellen die statische Organisation des Systems dar, die durch Module, linkbare Bibliotheken, ausführbare Programme etc. gegeben ist. *Verteilungsdiagramme (deployment diagrams)* zeigen, auf welchen physischen Komponenten die Ausführung zur Laufzeit stattfindet, z.B. auf welchen Computern die Teile einer verteilten Anwendung ablaufen.

4.3.2 Diagramme zur Softwarevisualisierung

Während man in den Entwurfsmethoden wenige Grundtypen von Diagrammen bestimmen kann, die in unterschiedlichen Varianten immer wieder auftreten, existiert für die Softwarevisualisierung kein solcher Konsens an Diagrammtypen. Dafür gibt es mehrere Gründe: Insgesamt liegen im Bereich der Softwarevisualisierung weniger Arbeiten vor, so daß man wiederkehrende Schemata seltener findet; die Prototypen, die zur Visualisierung verwendet werden, stellen häufig nur kleine Ausschnitte von Softwaresystemen dar und brauchen daher keinen umfangreichen, ausgefeilten Diagrammvorrat; weil unterschiedliche Zielsetzungen und technische Ansätze verfolgt werden, weichen die Darstellungsmittel entsprechend voneinander ab.

Immerhin ist es naheliegend, daß sich Werkzeuge zur Softwarevisualisierung an den von Entwurfsverfahren bekannten grafischen Notationen orientieren. Schließlich kann man davon ausgehen, daß die Notationen ihre Praxistauglichkeit bereits unter Beweis gestellt haben, und man kommt den Benutzern des Werkzeugs entgegen, denen die Darstellung bereits vertraut ist und die die Visualisierungen daher schneller erfassen können.

Unter Umständen können Visualisierungswerkzeuge Diagramme genau in der Form erzeugen, wie sie in der UML oder einer anderen Methode benutzt werden. Das gilt vor allem dann, wenn statische Aspekte eines Programms visualisiert werden; Software-Entwicklungsumgebungen wie etwa *Sniff+*, die die Quelltexte eines Softwareprojekts analysieren und daraus ein Klassendiagramm zeichnen, werden damit zu relativ einfachen, aber recht weit verbreiteten Werkzeugen zur Softwarevisualisierung. Während die Klassendiagramme der UML etc. über einen vergleichsweise großen Vorrat an Ausdrucksmitteln verfügen und sich mühelos mit Informationen überfrachten lassen, sind die von Visualisierungswerkzeugen erzeugten Diagramme zugunsten der Übersichtlichkeit schlicht (oder: spartanisch) gehalten.

Die Visualisierung dynamischer Programmaspekte kann ebenfalls in Form von Diagrammen, wie sie UML verwendet, erfolgen. Hier kommen z.B. Interaktions-, Aktivitäten- und Zustandsdiagramme in Frage. Wenn dynamische Aspekte visualisiert werden, muß die zeitliche Abfolge der Programmereignisse adäquat dargestellt werden. Interaktionsdiagramme beinhalten bereits eine (implizite) Zeitachse. Für andere Diagrammtypen kann eine animierte Anzeige benutzt werden, um die Abhängigkeit vom Zeitverlauf auszudrücken.

Um die Beziehungen und Interaktionen von Objekten zu visualisieren, kann man die UML-Objektdiagramme verwenden. Doch auch hier tritt das Problem auf, daß Objektdiagramme zwar ausreichende Notationen enthalten, um alle Aspekte in einem Diagramm darzustellen, daß ein solches Diagramm wegen der großen Informationsmengen, die in der Softwarevisualisierung anfallen, nicht mehr handhabbar ist. Deswegen ist es sinnvoll, Subtypen von Objektdiagrammen zu bilden, die jeweils nur einen Teil der Information darstellen. In [Hamer und Friedrich 96] wird eine Aufteilung in Konstruktions- und Referenz-Diagramme vorgeschlagen:

- *Konstruktionsdiagramme* zeigen alle zu einem Zeitpunkt vorhandenen Objekte als Knoten an. Gerichtete Kanten zwischen den Knoten zeigen an, welches Objekt die Erzeugung eines anderen Objekts veranlaßt hat. Konstruktionsdiagramme haben immer eine Baumstruktur (aber nicht unbedingt ein einziges Wurzel-Objekt, so daß das Diagramm in mehrere Zusammenhangskomponenten zerfallen kann).
- In *Referenz-Diagrammen* sind Objekte ebenfalls durch Knoten symbolisiert. Die Kanten stehen hier für Referenzen, durch die die Objekte untereinander verknüpft sind. Im Gegensatz zu Konstruktionsdiagrammen können die Verknüpfungen beliebig (n:m) sein, so daß keine Baumstruktur mehr entsteht, sondern ein Netz. Um dennoch zu einer übersichtlicheren baumartigen Darstellung zu gelangen, kann man mehrfach referenzierte Objekte als mehrere Knoten anzeigen, muß dabei aber kenntlich machen, welche unterschiedlichen Knoten sich auf dasselbe Objekt beziehen. Ein anderer Vorschlag sieht vor, nur die erste Referenz im Diagramm abzubilden, die für die Objekte angelegt wird. Damit erhält man eine „first-reference hierarchy“, die sich in einer Baumstruktur anlegen läßt [West 93].

Die Interaktionen zwischen den Objekten werden durch die beiden Diagrammtypen noch nicht abgedeckt. Um diese Informationen zu zeigen, kann man entweder einen dritten Subtyp von Objektdiagrammen bilden oder eine andere Darstellung benutzen; aus dem UML-Vorrat kommen Interaktionsdiagramme und *collaboration diagrams* in Frage. Von diesen Alternativen besitzen Interaktionsdiagramme den Vorteil der vorhandenen Zeitachse und einer relativ großen Verbreitung.

Über die genannten Formen von Diagrammen hinaus verwenden Visualisierungswerkzeuge häufig auch zusätzliche textuelle Darstellungen, um z.B. den (Aufruf-) Stack oder Objektattribute anzuzeigen. Mitunter werden auch solche Darstellungen als Diagramme bezeichnet. Auch wenn diese Anzeigen mit grafischen Attributen, Symbolen oder Icons angereichert sind, bleiben sie in ihrer Struktur eindimensional, während Diagramme durch eine zweidimensionale Indizierung gekennzeichnet sind (S. 10). Solche zusätzlichen Darstellungen können von großem Nutzen sein, sollten aber nicht als Diagramme bezeichnet werden.

4.3.3 Eine Suite von Diagrammen

Unter den beschriebenen Diagrammtypen gibt es nicht „*das* eine, einzig richtige“ Diagramm, das für alle Aufgaben der objektorientierten Softwarevisualisierung geeignet wäre. Statt nach einem solchen Über-Diagramm zu suchen, sollte man in einem Visualisierungswerkzeug lieber eine Anzahl von zusammenhängenden visuellen und textuellen Darstellungen vereinigen, die der Benutzer nach seinen Bedürfnissen einsetzen kann.

Für den Bereich der statischen Programmaspekte schlage ich drei Präsentationsformen vor:

- Klassendiagramme, die die vorhandenen Klassen in ihrer Vererbungshierarchie anzeigen.

- Komponentendiagramme (vgl. UML), in denen angezeigt wird, welche Komponenten es gibt und wie diese untereinander durch Benutzt-Beziehungen verknüpft sind. Muster für die Komponentenbildung sind z.B.: alle Klassen, die in einer Bibliothek gelinkt sind, bilden eine Komponente; alle Klassen, deren Quelltext im selben Verzeichnis liegt, bilden eine Komponente.
- Ein Klasseninspektor, der (als textuelles Werkzeug) alle Methoden und Attribute einer Klasse anzeigt.

Als Kombination von Diagrammtypen, die die dynamischen Aspekte aufzeigen, bietet sich die Kombination der folgenden Visualisierungsformen an [Lilienthal und Strunk 96]:

- Konstruktionsdiagramme
- Referenzdiagramme
- Interaktionsdiagramme

Zwei zusätzliche textuelle Darstellungen vervollständigen die dynamische Sicht:

- Ein Nachrichteninspektor, der eine Liste der zuletzt versendeten Nachrichten anzeigt.
- Ein Objektinspektor, der alle aktuellen Attributbelegungen eines Objekts auflistet.

4.4 Entwurf der Systemarchitektur

In den vorangegangenen Abschnitten wurden die allgemeinen Anforderungen und Möglichkeiten für die objektorientierte Softwarevisualisierung behandelt. Auf der Grundlage dieser allgemeinen Gedanken möchte ich nun das Konzept eines Visualisierungswerkzeugs beschreiben, das zur Zeit an der Universität Hamburg entwickelt wird. Der Prototyp trägt den Namen *VOOP2*; dieser Name ist angelehnt an ein vorheriges Projekts namens *VOOP*, was als Akronym für *Visualisierung objektorientierter Programme* stand [Hamer und Friedrich 96]. Ergebnis des Projekts VOOP war ein Prototyp, mit dem sich Konstruktionsdiagramme erstellen lassen. Als problematisch erwiesen sich vor allem der hohe Laufzeitoverhead und die Architektur, die zu sehr auf eine einzige Art von Diagramm zugeschnitten war. Diese Erkenntnisse haben die Entwicklung von VOOP2 von Beginn an geprägt.

4.4.1 VOOP2: Die grundsätzliche Idee

VOOP2 ist der Idee nach weniger ein einzelnes Werkzeug, sondern eine erweiterbare Zusammenstellung von miteinander verknüpften Werkzeugen. Die Grundfunktionalität kann etwa so zusammengefaßt werden:

- VOOP2 beobachtet die Ausführung eines objektorientierten Programmes und generiert Informationen über Elemente 1. Ordnung. Um welche Elemente es sich handelt, ist bekannt: Klassen, Objekte, Methoden und Nachrichten.
- Die Informationen können um Elemente 2. Ordnung erweitert werden. Für diese Elemente wurden einige Vorschläge gemacht, aber es ist offen, ob die Liste der vorgeschlagenen Elemente praktikabel oder vollständig ist. Offen ist weiterhin, inwieweit Elemente 2. Ordnung automatisch generiert werden können oder vom Benutzer zugewiesen werden sollen.

- Der Benutzer kann in unterschiedlichen Sichten bzw. Diagrammen die Informationen erschließen. Im vorangegangenen Abschnitt wurde eine Kombination von Darstellungen vorgeschlagen, die VOOP2 bieten soll. Ob diese Kombination den Bedürfnissen der Benutzer entspricht, wurde nicht diskutiert.

Mit anderen Worten: Sicher ist, daß VOOP2 einen funktionalen Kern besitzen muß, der auf einem zu analysierenden Programm arbeitet und die Elemente 1. Ordnung zur Verfügung stellt. Sicher ist auch, daß diese Elemente in einem Diagramm dargestellt werden. Weiterhin soll es möglich sein, mit Elementen 2. Ordnung umzugehen. Unsicher aber bleibt, welche Arten von Diagrammen erzeugt werden, und unsicher ist ebenfalls, welche Elemente 2. Ordnung später definiert werden.

Man könnte nun versuchen, diese Unsicherheiten auszuräumen und für die offenen Punkte Antworten zu ermitteln. Aber das ist sehr spekulativ, solange ungewiß ist, wie handhabbar ein Werkzeug von VOOP2 überhaupt sein kann, welche Einschränkungen aus Performancegründen nötig sind usw. Stattdessen soll eine Systemarchitektur entworfen werden, die offen angelegt und unter Umständen erst später um die zusätzlichen Features erweitert wird. Auf der Basis dieser Architektur kann dann ein Prototyp entwickelt werden, um Erfahrungen für den weiteren Ausbau von VOOP2 zu sammeln.

Bevor ich den Entwurf der Architektur vorstelle, möchte ich einige technische Grundlagen diskutieren und dabei Entscheidungen begründen, die wir für die Funktionsweise von VOOP2 gefällt haben. Gemäß der in Abschnitt 2.1.4 (siehe S. 8) beschriebenen Aufgabenbereiche behandele ich die Entscheidungen hinsichtlich Informationserfassung, -verwaltung und -präsentation.

4.4.2 Informationserfassung

Eine wesentliche Entscheidung ist darüber zu fällen, wie man die Informationen, die präsentiert werden sollen, erfassen möchte. Benötigt werden sowohl statische als auch dynamische Daten; statische Daten besagen z.B., welche Klassen es gibt und in welcher Relation sie zueinander stehen, dynamische sind diejenigen, die zur Laufzeit anfallen, d.h. welche Nachrichten versendet werden etc. Im folgenden gehe ich davon aus, daß ein *compiliertes* Programm untersucht wird, d.h. das Programm liegt in einer auf der jeweiligen Maschinenebene ausführbaren Form vor und kann unabhängig davon ausgeführt werden, ob der Quelltext vorliegt oder nicht. Für interpretierte Programme können die benötigten Informationen auf anderem Weg gewonnen werden, indem zum Beispiel der Interpreter manipuliert wird.

Statische Informationen können durch Parsen des Quelltextes erhalten werden. Alternativ kann vorausgesetzt werden, daß beim Übersetzen des Programms Debug-Informationen in das erzeugte Programm eingefügt werden. Ein mit Debug-Informationen compiliertes Programm enthält zwischen dem ausführbaren Code Informationen über die Programmstruktur, z.B. über die Namen von Typen und die Namen und Adressen von Variablen und Funktionen. Durch Decodierung der Informationen kann anhand des übersetzten Programms dessen Struktur ermittelt werden.

Für die Gewinnung dynamischer Informationen gibt es ebenfalls zwei mögliche Verfahren, nämlich die Instrumentierung von Quelltexten und wiederum die Verwendung von Debug-Informationen.

- Instrumentierung (auch: Annotation) des Quelltextes bedeutet, daß der ursprüngliche Quelltext um Anweisungen erweitert wird. Die zusätzlichen Anweisungen können dann z.B. Aufrufe von Zeichenroutinen sein, durch die Diagramme gezeichnet werden. Solche Anweisungen können manuell oder automatisch eingefügt werden.

  Zur automatischen Instrumentierung durchläuft der Programmtext vor der eigentlichen Compilierung einen Präprozessor, der den Quelltext parst und die Annotation vornimmt. Manuelle Instrumentierung wird direkt vom Entwickler vorgenommen und ist in dem Sinne sehr flexibel, so daß ein Programmierer Annotationen gezielt an den Stellen vornehmen kann, die von Interesse sind.

  Nachdem der instrumentierte Quelltext übersetzt ist, enthält das ausführbare Programm unmittelbar den Code, der für die Analyse bzw. die Visualierung notwendig ist; dieser Code wird genauso effizient ausgeführt wie der eigentliche Programmcode. Der Laufzeit-Overhead, der durch Analyse und Präsentation entsteht, ist damit relativ klein.

  Der Pferdefuß dieses Verfahrens ist die vor der Visualisierung erforderliche Neuübersetzung.

- Wenn das Programm mit Debug-Informationen compiliert wurde, können aus diesen Informationen z.B. Funktionsnamen und -adressen ermittelt werden. Das Visualisierungssystem muß dann wie ein konventioneller Debugger vorgehen: Das Programm wird an bestimmten Stellen mit Breakpoints (Haltepunkten) versehen, an denen die Programmausführung unterbrochen wird. Auf die Unterbrechung wird reagiert, indem z.B. ein angezeigtes Diagramm aktualisiert wird; danach kann die Programmausführung fortgesetzt werden.

  Während ein Programm entwickelt wird, übersetzt der Programmierer es gewöhnlich mit Debug-Informationen. Auch Klassenbibliotheken und Frameworks, deren Quelltexte nicht zugänglich sind, werden meistens in einer mit Debug-Informationen übersetzten Version ausgeliefert. Die Forderung, daß diese Informationen vorliegen müssen, bedeutet in der Praxis also keine Einschränkung.

  Der Nachteil gegenüber der Quelltext-Instrumentierung liegt im Laufzeit-Overhead, der durch das Setzen von Breakpoints und das Interpretieren der von Breakpoints ausgehenden Unterbrechungen entsteht.

Wir haben uns für die zweite Möglichkeit entschieden, um nicht vom Vorliegen der Quelltexte abhängig zu sein und um das Recompilieren zu vermeiden, das durch Instrumentierung erforderlich wäre.

Im Zusammenhang mit der Informationserfassung ist auf die Menge der anfallenden Informationen hinzuweisen. Während des Ablaufs eines objektorientierten Programms werden in der Regel viel mehr Objekte erzeugt und Nachrichten versendet, als man in sinnvoller Art und Weise präsentieren kann. Außerdem sinkt die Ausführungsgeschwindigkeit stark, wenn permanent Informationen über das Programm erfaßt und verarbeitet werden müssen. Deswegen muß es Möglichkeiten geben, Art und Umfang der erfaßten Informationen nach Bedarf zu bestimmen. Dazu kann man sowohl die Informationserfassung als auch die -präsentation beschränken, d.h.

- nicht alle anfallenden Informationen müssen erfaßt werden, und
- nicht alle erfaßten Informationen müssen angezeigt werden.

In welcher Art die Informationsmenge einschränkt wird, sollte dem Benutzer des Werkzeugs überlassen werden.

4.4.3 Informationsverwaltung

Das Visualisierungswerkzeug soll mehrere Ansichten eines Programmablaufs in verschiedenen Diagrammen bieten können (s. o.). Es muß daher eine zentrale Stelle geben, die die Laufzeitinformationen verwaltet und mehreren Klienten zur Verfügung stellt. Wir haben uns entschieden, ein Modell des ablaufenden Programms zu konstruieren. Es verändert sich dynamisch und bildet zu jeder Zeit den Zustand des analysierten Programms ab. Wir erhalten damit ein *Programmodell*, in dem es für jedes Objekt, das im Programm vorhanden ist, ein korrespondierendes beschreibendes Objekt gibt. Dieses beschreibende Objekt bezeichnen wir als *Metaobjekt*, um es von seinem Gegenstück im analysierten Programm zu unterscheiden.

Auf den ersten Blick mag dieser Ansatz redundant erscheinen, weil ein Spiegelbild des beobachteten Programms angelegt wird. Jedoch gibt es keine Instanz, die einen Zugriff auf die originalen Objekte erlaubt – sie befinden sich „irgendwo“ im Speicherbereich des Programms, aber es gibt keinen Index, keine Symboltabelle oder ähnliches, um auf die Objekte zuzugreifen. Im Programmodell werden die Metaobjekte hingegen strukturiert gespeichert und lassen sich über Suchfunktionen etc. erschließen. In den Metaobjekten können auch zusätzliche Daten gespeichert werden, die die echten Objekte betreffen, dort aber nicht vorhanden sind, z.B. wann und in welchem Kontext das Objekt erzeugt wurde oder von welchen Stellen aus es referenziert wird.

Die Verwaltung der Metaobjekte beansprucht natürlich Rechenzeit. Andererseits ergeben sich Effizienzgewinne, weil Zugriffe auf die Metaobjekte billiger sind als auf die zugehörigen Originale, da im letzteren Fall Operationen über die Grenzen von Prozeßräumen notwendig sind.

Auch wenn es wünschenswert ist, immer ein vollständiges Programmmodell zu führen, das zu jedem existierenden Objekt ein Metaobjekt enthält, wird man es aus den oben beschriebenen Gründen vorziehen, die Anzahl der erfaßten Metaobjekte in einem vernünftigen Rahmen zu halten.

Das Programmodell enthält also Metaobjekte, in denen das dynamische Verhalten des Programms abgebildet wird. Weitere Elemente wie Beschreibungen von Klassen und von Funktionen sind nötig, um die statischen Aspekte des Programms zu beschreiben.

Mit diesen Elementen ist das Programmodell ein Datenbehälter, der sich passiv verhält, d.h. er wird mit Informationen versorgt, die gespeichert werden und über eine Schnittstelle abgefragt werden können. Das Auffrischen des Programmodells erfolgt kontinuierlich während des Ablaufs des beobachteten Programms; hier muß es wohldefinierte Ereignisse geben, die ein „Updaten“ des Programmodells auslösen. Ein solches Ereignis markiert somit jeweils eine Veränderung des Programmodells. Die Klienten, die das Programmodell in irgendeiner Form in Anspruch nehmen, müssen ebenfalls Kenntnis von den Ereignissen haben, um dem Benutzer das veränderte Programmodell zu präsentieren.

Das Programmodell stellt ein allgemeines objektorientiertes Modell des Programms dar, berücksichtigt aber noch keine speziellen Anforderungen, die für bestimmte Zwecke der Analyse oder Visualisierung auftreten können. Wenn beispielsweise Aussagen über die Vorgeschichte eines aktuellen Zustands gemacht werden sollen, kann das Metamodell dazu keine Auskunft geben. Ein Werkzeug, das solche Informationen benötigt, muß hier sein eigenes Datenmodell pflegen; Grundlage dazu bleibt aber das Programmodell.

In der ersten Ausbaustufe von VOOP2 werden alle Elemente des Programmodells zu Elementen erster Ordnung gehören. Elemente 2. Ordnung sollen hinterher als zusätzliche Modellelemente integriert werden. Wenn man Elemente 2. Ordnung einführen möchte, von denen man weiß, daß sie nur von einem einzigen Werkzeug in Anspruch genommen werden, sollten diese Elemente nicht im Programmodell, sondern im werkzeugeigenen Datenmodell angesiedelt werden. Dieser Aspekt wird anhand eines Beispiels in Kapitel 7 diskutiert.

4.4.4 Informationspräsentation

Die im Programmodell erfaßten Daten sollen in verschiedenen Diagrammen dargestellt werden; alle Diagramme, die zu einer Zeit angezeigt werden und sich auf dasselbe Programm beziehen, sollen natürlich zueinander konsistent sein. Änderungen des Programmodells sollten in allen angezeigten Diagrammen gleichzeitig sichtbar werden.

Änderungen können auch vom Benutzer vorgenommen werden, indem er z.B. die Bezeichnung eines Metaobjekts ändert. Auch hier sollten natürlich alle Diagramme die vorgenommene Änderung gleichzeitig berücksichtigen.

Hinsichtlich der Größe, die eine automatisch erzeugte Visualisierung in kürzester Zeit erreicht, sollte die Oberfläche der Werkzeuge zulassen, daß Benutzer bereits erstellte Diagramme zusammenfassen, Teile davon ausblenden oder strukturieren können. Die Werkzeuge sollten daher eine interaktive Oberfläche besitzen, die die direkte Manipulation von Diagrammen erlaubt. Wenn wir davon ausgehen, daß Elemente 2. Ordnung vom Benutzer angelegt werden können, dann wird es auch hier wünschenswert sein, die Elemente durch unmittelbare Interaktion mit einem angezeigten Diagramm zu definieren.

4.4.5 WAM: Werkzeug, Automat und Material

Die Entwicklungsmethode WAM [Kilberth et al. 94] ist ein Modell zur objektorientierten Anwendungsentwicklung. Sie wird bestimmt von dem Leitbild, daß die Benutzer von Software Experten ihres Anwendungsgebietes sind, qualifizierte Tätigkeiten verrichten und dazu über Wissen und Erfahrung verfügen. Sie führen ihre Tätigkeiten eigenverantwortlich und situationsabhängig durch. Software muß so beschaffen sein, daß das eigenverantwortliche und situative Vorgehen unterstützt wird.

Das Leitbild orientiert sich an der Arbeit in einer Werkstatt, in der ein Handwerker seine Arbeit verrichtet; er verfügt über Fachwissen und bestimmt selbst über die Art, in der er seine Arbeit ausführt. Seine Tätigkeit weist zwar immer wiederkehrende Muster auf, die Ausführung variiert aber von Mal zu Mal. Der Arbeitsgegenstand sind unterschiedliche Materialien, die verändert, miteinander verbunden, ineinander überführt werden usw., bis das gewünschte Arbeitsergebnis vorliegt. Neben seinem Fachwissen und seiner Erfahrung stehen dem Ausführenden Werkzeuge zur Verfügung. Natürlich hat jedes Werkzeug einen bestimmten Zweck, zu dem es eingesetzt wird; *wie* die Benutzung erfolgt, kann sich in gewissen Freiräumen bewegen und ist dem Ermessen des Handwerkers überlassen. Neben den Werkzeugen stehen ihm Automaten zur Verfügung. Sie führen Routinetätigkeiten aus, die immer wieder auf dieselbe Art verrichtet werden.

Das Leitbild der Werkstatt wird auf die Softwareentwicklung übertragen und läßt den Benutzern der Software die Rolle der fachlich qualifizierten Handwerker zukommen. Dem Werkstatt-Vorbild entsprechend, werden die Entwurfsmetaphern Werkzeug, Material und Automat eingeführt. Mit den Entwurfsmetaphern werden Softwareentwicklern Hilfsmittel zur

Verfügung gestellt, an denen sie sich während der Modellierung eines Systems orientieren können.

Die erste Metapher beschreibt Materialien: Als Materialien werden die Arbeitsgegenstände der Anwendung modelliert. Arbeitsgegenstände werden durch fachliche Umgangsformen gekennzeichnet; solche Umgangsformen sind beispielsweise allgemeine Prüfungen, ob der Zustand des Materials zulässig und konsistent ist. Andere Umgangsformen sind spezifisch für ein ganz bestimmtes Material. Ein Beispiel aus dem Finanzwesen ist das Material „Konto“, das über Umgangsformen wie „einzahlen“, „auszahlen“ und „Zinsen berechnen“ verfügt (vgl. z.B. [Bäumer et al. 95]).

Die Metapher Werkzeug wird für die Programme verwendet, mit denen die Materialien interaktiv bearbeitet werden; die Programme werden als Software-Werkzeuge entworfen, die den Benutzern die Materialien präsentieren und die Bearbeitung ermöglichen. Der interaktive Anteil eines Werkzeugs übernimmt die Präsentation des Materials und besitzt die Fähigkeit, von einem Benutzer Anweisungen entgegenzunehmen. Das Durchführen dieser Anweisungen ist unabhängig von der Interaktion und findet auf einer fachlichen Ebene statt; für diesen Anteil ist die funktionale Seite des Werkzeugs verantwortlich. Die Trennung von Interaktion und Funktion ist ein wichtiges Architekturprinzip der Werkzeugkonstruktion.

Die dritte Metapher ist die des Automaten. Während Werkzeuge Materialien nur reaktiv, d.h. auf Anweisung des Benutzers verändern, können Automaten selbständig Materialien bearbeiten. Für das Beispiel des Materials „Konto“ kann man sich einen Automaten vorstellen, der in vorgegebenen Zeitabständen alle Konten auf die Einhaltung ihres Kreditrahmens überprüft und selbständig die Sperrung eines Kontos veranlaßt, falls die Kreditlinie überzogen wurde. Das Verhalten von Automaten läßt sich beeinflusen, indem man sie konfiguriert. Ein Software-Automat wirkt im Verborgenen; er arbeitet, nachdem er einmal gestartet wurde, im Hintergrund. Für das Konfigurieren des Automaten oder das Anzeigen seiner Aktivität sind Werkzeuge zuständig; der Automat selbst hat keine sichtbare Oberfläche.

4.4.6 Architektur des Visualisierungswerkzeugs nach WAM

Wenn man nun die Entwurfsmetaphern auf das Konzept von VOOP2 anwendet, wird man die Komponenten, die Visualisierungen am Bildschirm darstellen, als Werkzeuge ansehen. Das Material, das damit in den Werkzeugen angezeigt wird (und durch die Werkzeuge verändert werden kann), ist das Programmodell. Das untersuchte Programm ist ebenfalls als Material von VOOP2 anzusehen. Das Erstellen und Aktualisieren des Programmodells ist schließlich eine Aufgabe, die von einem Automaten erledigt werden kann. Die Elemente dieses Ansatzes können den drei Teilbereichen der Visualisierung – Informationserfassung, Informationsverwaltung und Informationspräsentation – zugeordnet werden:

- Die Informationserfassung kann von einem Automaten übernommen werden. Der Automat beobachtet das ablaufende Programm und schreibt das Programmodell fort. Er arbeitet damit auf zwei Materialien, nämlich dem analysierten Programm und dem Programmodell. Da der Automat steuerbar sein soll, muß es ein Werkzeug geben, das die Konfiguration übernimmt.
- Die Informationsverwaltung erfolgt im Programmodell, das damit als zentrales Material der Anwendung identifiziert werden kann. Für das Anlegen des Materials ist in erster Linie der Automat zuständig, für das Präsentieren des Materials die Werkzeuge.

- Die Präsentation der Daten in Form von Diagrammen übernehmen Werkzeuge. Jedes Diagramm ist das Material eines Werkzeugs (während das Programmodell gemeinsames Material aller Werkzeuge ist). Vorgesehen sind mehrere Werkzeuge, die mit unterschiedlichen Diagrammen unterschiedliche Sichten auf das Programmodell bieten.Ein alternativer, ebenfalls zutreffender Sprachgebrauch ist, das Visualisierungssystem als Ganzes (inklusive Materialien und Automat) als Werkzeug zu verstehen und für jeden Diagrammtyp ein *Subwerkzeug* vorzusehen.
  Im weiteren Verlauf dieses Kapitels wird eine technische Trennung der einzelnen Komponenten vorgeschlagen, durch die es angebracht ist, von Werkzeugen statt von Subwerkzeugen zu sprechen und den Ausdruck System (oder Visualisierungssystem) zu benutzen, wenn die Gesamtheit der Komponenten gemeint ist.

Der daraus resultierende Systementwurf basiert also auf dem Automaten, der die Ausführung eines Programms steuert und überwacht. Sobald der Automat feststellt, daß Ereignisse vorliegen, auf die er (nach seiner gegenwärtigen Einstellung) reagieren muß, verändert er das Programmodell und teilt allen aktiven Werkzeugen die Änderung mit. Für diese Mitteilung läßt sich das in Kapitel 3 beschriebene Beobachter-Entwurfsmuster verwenden (siehe S. 70).

Ein Werkzeug, das über ein solches Ereignis informiert wird, kann nun den geänderten Zustand des Programmodells erfragen und darauf reagieren, indem es das angezeigte Diagramm fortschreibt oder ein werkzeuginternes Material an den neuen Zustand anpaßt. Abbildung 7 stellt die Architektur dar.

4.5 Abhängigkeit von der Programmiersprache

An dieser Stelle ist es noch unerheblich, nach welchem Paradigma die visualisierten Programme programmiert sind; eine Festlegung auf die Visualisierung objektorientierter Programme ist noch nicht erfolgt. Fragt man jedoch nach den Umgangsformen das Metamodells und den Funktionen, die für Automat und Werkzeuge benötigt werden, wird man sofort auf Eigenschaften

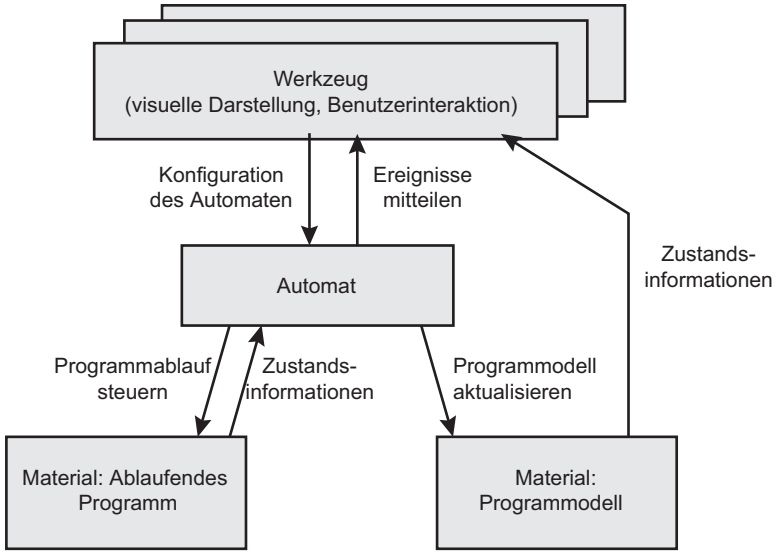


Abb. 7: Werkzeuge, Automat und Materialien für das Visualisierungssystem

treffen, die spezifisch für die Analyse objektorientierter Programme sind. Diese Eigenschaften werden in den anschließenden Kapiteln näher spezifiziert.

Bisher spielte es ebenfalls keine Rolle, in welcher Programmiersprache die zu visualisierenden Programme formuliert wurden. Die Architektur und die Schnittstellen der Archtekturkomponenten sollten unabhängig von Programmiersprachen sein. Die Präsentation soll objektorientierte Konstrukte und Konzepte darstellen, die ebenfalls für alle objektorientierten Sprachen gleich sind. Lediglich einige Spracheigenschaften weichen voneinander ab, für die geringe Anpassungen des Metamodells und der anzeigenden Werkzeuge nötig wären: etwa Metaklassen in Smalltalk, „Vererbung“ von Interfaces in Java, Mehrfachvererbung mit virtuellen Basisklassen in C++.

Entscheidender aber ist der Aufwand, den man zum Erhalt von Informationen über das analysierte Programm benötigt. Die Informationserfassung, wie ich sie oben beschrieben habe, ist ausgerichtet auf übersetzende, nicht aber auf interpretierende Sprachen. Informationen über das Laufzeitverhalten interpretierter Programme können im Vergleich zu compilierten Programmen einfacher gewonnen werden, weil innerhalb eines Interpreters ein Zugriff auf den interpretierten Quelltext wie auch auf die vom Programm verwalteten Daten relativ leicht ist. Verfügt die Programmiersprache zusätzlich über Meta-Objekt-Informationen[2] (wie es in Java der Fall ist), wird die Informationserfassung weiter vereinfacht. Daß unter diesen Umständen Programmvisualiserung mit geringem Aufwand realisiert werden kann, zeigt Chris Laffra anhand von Java [Laffra 97].

Die Programmiersprache, für die VOOP2 Visualiserungen erzeugen soll, ist C++. Die Entscheidung zugunsten dieser Sprache hat mehrere Gründe: Die Sprache ist an Universitäten, vor allem aber auch in der Praxis weit verbreitet und wird mittelfristig eine der wichtigsten Programmiersprachen sein. Für C++ liegen umfangreiche Klassenbibliotheken und Frameworks vor, worin wir viele Anwendungsmöglichkeiten (und einen hohen Anwendungsbedarf!) für ein Visualisierungssystem sehen. So stammen auch die zu Beginn dieses Kapitels vorgestellten Beispiele aus einem C++-Framework, und in dem bereits in der Einleitung erwähnten Forschungsprojekt wird ebenfalls C++ eingesetzt. Außerdem besitzt C++ bestimmte „kritische“ Spracheigenschaften (z.B. das Fehlen einer Garbage Collection), wodurch Programmfehler auftreten, deren Beseitigung nach qualifizierter Werkzeugunterstützung verlangt – das verfügbare Angebot an Speichercheckern etc. bestätigt diese Aussage.

4.6 Einordnung in Roman-Cox

Zum Abschluß dieses Kapitels möchte ich eine Einordnung von VOOP2 in die Familie der Softwarevisualisierungssysteme vornehmen. Ich beziehe mich auf die Taxonomie von Roman und Cox [Roman und Cox 93], die in Abschnitt 2.2 vorgestellt wurde:

- Bereich: VOOP2 erfaßt primär zwei Bereiche, nämlich Daten (data state) und Verhalten (behaviour). Daten werden vor allem im Referenzdiagramm visualisiert (die visualisierten Daten sind diejenigen Objektattribute, die auf andere Objekte verweisen), das Verhalten in Interaktions- und Konstruktionsdiagrammen.

2 Meta-Objekt-Informationen ermöglichen es, zur Laufzeit zu ermitteln, zu welcher Klasse ein Objekt zählt.

Wenn VOOP2 Klassendiagramme umfaßt, kann als dritter Bereich Codevisualisierung genannt werden; dies spielt für VOOP2 aber nur eine untergeordnete Rolle.

- Abstraktion: Alle drei Ebenen, die Roman und Cox angeben, werden von VOOP2 unterstützt: Zunächst einmal soll VOOP2 *direkte Repräsentation* leisten, d.h. Informationen werden als Elemente 1. Ordnung erkannt und angezeigt. Durch die Notwendigkeit, die Anzeige auf einen Teil der Informationsmenge beschränken zu können, ist bereits das Kriterium für *strukturelle Repräsentation* erfüllt. Wenn in einer späteren Ausbaustufe auch Elemente 2. Ordnung eingeführt werden, wird eine *synthetisierte Repräsentation* ermöglicht.
- Methode der Spezifikation: Neben der eher experimentellen manipulativen Spezifikation sehen Roman und Cox Annotation oder Deklaration vor. Mit VOOP2 haben wir uns gegen Code-Instrumentierung und damit gegen die Spezifikation durch Annotation entschieden. Die Spezifikation erfolgt stattdessen deklarativ, indem innerhalb von VOOP2 Optionen und Filter zur Visualisierung festgelegt werden.
- Benutzerschnittstelle: Die Benutzerschnittstelle von VOOP2 kann noch nicht eingeordnet werden, da bisher noch nicht betrachtet wurde, wie die Diagramme realisiert werden sollen. Auf diesen Punkt wird später (Kapitel 7) eingegangen.
- Präsentation: Erklärende Präsentation ist möglich, indem Elemente 2. Ordnung in Diagramme aufgenommen werden. Andere Aspekte der Präsentation hängen davon ab, wie die Diagramme später realisiert werden.

Auch wenn der für den Benutzer sichtbare Teil des Visualisierungssytems bisher nicht behandelt wurde, zeigt die Einordnung von VOOP2 in die Taxonomie von Roman und Cox bereits, daß das System als Visualisierungswerkzeug einen weiten Teil der möglichen Anforderungen und der Potentiale, die Visualisierungssysteme bieten, abdeckt.

<table>
<tr><th>Bereich</th><th>Abstraktion</th><th>Spezifikation</th><th>Benutzer-schnittstelle</th><th>Präsentation</th></tr>
<tr><td>Code</td><td>direkt</td><td>Annotation</td><td rowspan="4">Vgl. Kapitel 7</td><td>Erklärende Präsentation</td></tr>
<tr><td>Daten</td><td>strukturell</td><td>Deklaration</td><td rowspan="3">Vgl. Kapitel 7</td></tr>
<tr><td>Kontroll-zustand</td><td>synthetisiert</td><td>Manipulation</td></tr>
<tr><td>Verhalten</td><td></td><td></td></tr>
</table>

Dunkelgraue Flächen kennzeichnen Eigenschaften, die durch das Konzept von VOOP2 gewährleistet sind.
Hellgraue Flächen hängen von der Realisierung der einzelnen Werkzeuge ab.

Tab. 2: Eigenschaften von VOOP2

4.7 Zusammenfassung

In diesem Kapitel wurden zunächst Beispiele für Fragestellungen geschildert, die sich beim Verstehen von objektorientierten Programmen ergeben; Softwarevisualisierung kann das Verstehen erleichtern. Insbesondere können sich (professionelle) Entwickler schneller in Frameworks einarbeiten, wenn ihnen geeignete Visualisierungswerkzeuge zur Verfügung stehen.

Unter dem Namen VOOP2 wurde das Konzept eines Systems vorgestellt, das für diesen Einsatzzweck entwickelt wird. Die Architektur gliedert sich in einen Automaten, der die für die Visualisierung notwendigen Information erfaßt, in ein Programmodell, das diese Informationen verwaltet, und in eine Reihe von Werkzeugen, die die Informationen in unterschiedlichen Diagrammen präsentieren.

Im Programmodell werden die Informationen nach Elementen erster und zweiter Ordnung unterschieden. Elemente erster Ordnung beschreiben die grundlegenden Bausteine eines objektorientierten Programms: Klassen, Methoden und Objekte. Die Elemente zweiter Ordnung sind nicht a priori festgelegt; es kann sich bei ihnen um Konzepte oder Konstrukte handeln, die von der Ebene der ersten Ordnung abstrahieren und die die Architektur eines Programms nachbilden.

Indem das allgemein gehaltene Programmodell von VOOP2 von mehreren Werkzeugen benutzt werden kann, ist das Konzept offen für verschiedenartige Diagramme. Durch die Möglichkeit, Elemente 2. Ordnung einzuführen, können spätere Diagramme auf unterschiedlichen Abstraktionsstufen stehen.

Unter den üblichen Diagrammen zeichnen sich Interaktionsdiagramme dadurch aus, daß sie die Dynamik eines Programms zeigen, ohne animiert werden zu müssen. Daher eignen sie sich auch für (gedruckte) Dokumentationen. Außerdem profitieren sie davon, hinreichend bekannt zu sein. Aus diesen Gründen erstellt das erste zu VOOP2 zählende Werkzeug Interaktionsdiagramme. Bevor dieses Werkzeug in Kapitel 7 behandelt wird, befassen sich die folgenden beiden Kapitel mit dem Programmodell und dem Automaten.

Kapitel 5
Modellierung der Dynamik von Objekten

Im vorangegangen Kapitel wurde das *Programmodell* als zentrales Material des Visualisierungssystems vorgestellt. Es wurde beschrieben, daß im Programmodell alle Informationen verwaltet werden sollen, die über einem untersuchten Programm erfaßt und für Visualisierungswerkzeuge zur Verfügung gestellt werden. Das Programmodell ist damit anwendungsfachlich begründet; denn es verkörpert das untersuchte Programm, das aus Sicht des Softwareentwicklers, der das Visualisierungssystem einsetzt, ein fachliches Objekt darstellt.

Das Programmodell soll diejenigen Aspekte von Programm und Ausführung abbilden, die zur Visualisierung des Programmverhaltens bekannt sein müssen. Wie in Abschnitt 4.1.1 beschrieben, geht es darum, die Objekte und ihre Interaktionen sichtbar zu machen.

In diesem Kapitel werden die Bestandteile des Programmodells beschrieben. Es handelt sich dabei um mehrere Typen, die den Elementen 1. Ordnung entsprechen und zusammen ein *Metamodell* ergeben. Der Entwurf des Metamodells erfolgte gemeinsam mit Stephen Friedrich, der die im folgenden vorgestellten Klassen und den in Kapitel 6 behandelten Automaten implementiert hat.

5.1 Verhalten von objektorientierten Programmen

5.1.1 Programm und Ausführung

Weil es in dieser Arbeit um die Visualisierung des Programm*verhaltens* geht, muß man sich klarmachen, daß das Programm ein statisches Gebilde ist. Erst wenn es gestartet wird, entsteht mit der Ausführung eine dynamische Instanz des Programms.

Nun kann man sowohl Programm als auch Ausführung auf verschiedenen Abstraktionsebenen betrachten. Auf der niedrigsten Ebene ist das Programm ein Folge von Binärziffern, in den Mikroprozessorbefehle codiert sind; auf einer sehr viel höheren Ebene werden Programm und Ausführung durch Entwurfskonzepte repräsentiert. Wir präzisieren die Begriffe daher folgendermaßen:

- Als *Programm* verstehen wir die ausführbare Datei, die durch das Compilieren und Linken eines Quelltextes entstanden ist. Das Programm stellt eine Anwendung dar. Die Abstraktionsebene, auf der wir uns mit der Anwendung beschäftigen, ist die Ebene des Quelltextes. Wenn von Bestandteilen des Programms gesprochen wird, geschieht das auf dieser Ebene; Bestandteile des Programms werden durch die Sprachmittel der verwendeten

Programmiersprache ausgedrückt.
Weiterhin wird vorausgesetzt, daß das Programm vom Compiler erzeugte Debug-Informationen enthält, in denen die Bestandteile des Programms beschrieben sind.

- Das Programm ist statisch, seine dynamische Instanz ist die *Ausführung*. Die Ausführung ist eine Folge von Operationen, die in der verwendeten Programmiersprache zur Verfügung gestellt werden.

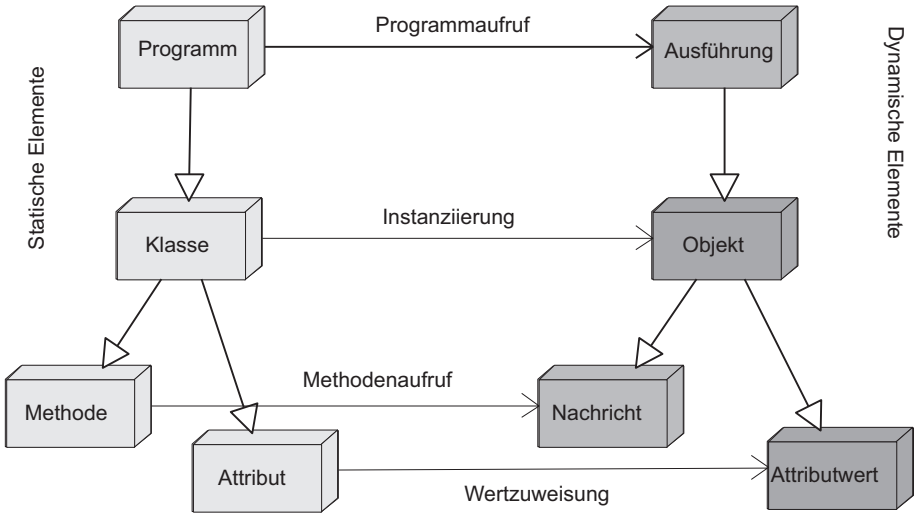
5.1.2 Klassen und Objekte

Beschäftigen wir uns daher mit den grundlegenden objektorientierten Konzepten, nämlich Klassen und Objekten. Eine Klasse beschreibt „a set of potential run-time objects characterized by the operations available on them (the same for all instances of a given class), and the properties of these operations. These objects are called the instances of the class. Classes and objects should not be confused: 'class' is a compile-time notion, whereas objects only exist at run-time. This is similar to the difference that exists in classical programming between a program and one execution of that program“ [Meyer 88] (S. 476).

Bezogen auf die Programmiersprache Eiffel schreibt Bertrand Meyer, daß Klassen durch zwei Arten von Eigenschaften (features) beschrieben werden. „There are two kinds of features: routines [...], that is to say operations; and attributes, that is to say data items associated with objects of the class“ [Meyer 88] (S. 477). Diese Charakterisierung trifft auch auf C++ zu, nur wird dort ein anderer Sprachgebrauch benutzt; Routinen werden als Methoden oder member functions bezeichnet.

Damit können Programm und Ausführung näher beschrieben werden:

- Die grundlegenden statischen Elemente eines objektorientierten Programms sind *Klassen*. Eine Klasse legt fest, welche Attribute Objekte der Klasse besitzen, und definieren *Methoden*, die für die Objekte der Klasse aufgerufen werden können. Methoden stellen ebenfalls ein statisches Konstrukt dar.
- Das grundlegende dynamische Konzept objektorientierter Sprachen ist das *Objekt*. Objekte werden zur Laufzeit von Klassen instanziiert. Der Zustand eines Objekts ergibt sich aus



Die horizontalen Pfeile zeigen, wie statische Elemente zur Laufzeit dynamisch wirksam werden.

Abb. 8: Fachliches Modell eines objektorientierten Programms

dessen Attributwerten.
Objekte können miteinander in Verbindung stehen, indem sie sich über Attributwerte referenzieren oder aneinander Methoden aufrufen. Der Aufruf einer Methode wird in objektorientierten Systemen als ***Nachricht*** bezeichnet. So wie Objekte die dynamische Verkörperung der statischen Klassen sind, sind Nachrichten das dynamische Analogon zu den Methoden.

Das anwendungsfachliche Modell, das wir damit erhalten, faßt Abbildung 8 zusammen. Dieses Modell gilt für sequentielle Programmiersprachen; in nebenläufigen Programmen müßte die Möglichkeit berücksichtigt werden, daß dieselbe Methode zur Laufzeit von unterschiedlichen Seiten aktiviert werden kann. In diesem Fall stehen dem statischen Konstrukt der Methode mehrere unabhängig, dynamische Instanziierungen *(stack frames)* gegenüber. Auf die Eigenschaften nebenläufiger Programme werde ich nicht näher eingehen.

5.1.3 Beziehungen zwischen Klassen und Objekten

Wenn eine objektorientierte Anwendung entwickelt wird, besteht eine der zentralen Tätigkeiten darin, die Klassen der Anwendung zu identifizieren und miteinander in Beziehung zu setzen. Dieser Schritt entspricht der Entity-Relationship-Modellierung, die zu den richtungsweisenden Vorläufern der objektorientierten Anwendungsentwicklung zählt.

Im Unterschied zu Entity-Relationship-Modellen erlaubt ein objektorientierter Entwurf, durch Vererbungsbeziehungen Spezialisierungen von Klassen zu modellieren. Unter den Beziehungen, die zwischen Klassen bestehen können, wird Vererbung stets an erster Stelle genannt, weil sie für objektorientiertes Vorgehen charakteristisch ist. Die Zahl an Beziehungstypen, die in der Literatur aufgezählt werden, variieren von Autor zu Autor (vgl. z.B. [Schäfer 94], S. 82, S. 106 und S. 126). Die unterschiedlichen Angaben hängen in der Regel davon ab, ob die Sichtweise stärker an Analyse und Entwurf oder an der Programmierung orientiert ist.

Die in der Analysephase identifizierten Relationen sind typischerweise Assoziation, Generalisierung und Aggregation. Assoziation ist der allgemeine Fall einer semantischen Beziehung zwischen Klassen [Booch 94], der im weiteren Prozeß durch einen der anderen Relationstypen ersetzt wird. Generalisierung (bzw. Spezialisierung) drückt sogenannte „is-a“-Beziehungen aus; sie besagen, daß bestimmte Klassen als spezielle Untertypen anderer Klassen angesehen werden können, und stellen das abstrakte Konzept von Vererbung dar. Aggregationen sind „part-of“-Beziehungen, d.h. Objekte einer Klasse treten als Teile von Objekten einer anderen Klasse auf.

Die Relationen, die sich im Hinblick auf den Programmentwurf definieren lassen, sind:

- Vererbung: Durch Vererbung werden Generalisierungs-/Spezialisierungsbeziehungen realisiert. Von einer (Ober-)Klasse werden Unterklassen abgeleitet, die mit der Oberklasse typverträglich sind. Die Eigenschaften der Oberklasse werden an die Unterklassen vererbt und können dort modifiziert bzw. erweitert werden.
- Containment: Aggregationen werden als Containment-Beziehungen realisiert. Üblicherweise wird für das aggregierte Objekt eine Instanz der zugehörigen Klasse in der Klassendefinition der aggregierenden Klasse deklariert. Enthaltendes und enthaltenes Objekt sind damit fest aneinander gebunden.

- Verwendung: Verwendung ist eine losere Beziehung als Containment. Sie entsteht, wenn ein Objekt eine Referenz auf ein anderes Objekt derselben oder einer anderen Klasse besitzt.
- Generische Instanziierung: Wenn eine objektorientierte Programmiersprache Generizität erlaubt, lassen sich mittels Klassenschablonen (Templates o.ä.) parametrisierte Klassen generieren.
- Metaklassen-Instanziierung: In einigen Sprachen (z.B. CLOS und Smalltalk) können Klassen als Instanzen anderer Klassen erzeugt werden; sie stellen dann Instanzen von Metaklassen dar.

Relationen zwischen Klassen sind statisch: Sie werden im Entwurf festgelegt und gelten unabhängig von der Laufzeit des Programms.

Aus den statischen Relationen zwischen Klassen entstehen zur Laufzeit des Programms dynamische Relationen zwischen Objekten. Mit den Objektrelationen wird beschrieben, in welchem Verhältnis Objekte, die zur selben Zeit existieren, miteinander stehen. Objektrelationen gehen aus den Containment- und Verwendungsrelationen der Klassen hervor [Hamer und Friedrich 96]:

- Benutzt-Relation: Ein Objekt *benutzt* ein anderes Objekt, indem es dessen Methoden aufruft oder auf dessen Attribute zugreift. Dies ist der allgemeinste Fall; jeder der drei folgenden Relationen ist auch immer eine Benutzt-Relation.
- Referenziert-Relation: Ein Objekt *referenziert* ein anderes Objekt, indem es einen Verweis auf das Objekt als eigenes Attribut besitzt. Dies ist im allgemeinen auch die Voraussetzung, durch die der Aufruf von Methoden am anderen Objekt möglich wird.
- Hat-Relation: Eine Objekt *beinhaltet* ("*hat*") ein anderes Objekt, wenn das letztgenannte Objekt fest zum Speicherbereich des ersten Objekts gehört. Diese Beziehung folgt aus Aggregation bzw. Containment.
- Erzeugt-Relation: Ein Objekt erzeugt ein anderes Objekt, indem es dessen Konstruktion veranlaßt.

Objekte, die in einer Benutzt-Beziehung stehen, können miteinander interagieren: Sie können wechselseitig Methoden aufrufen und, sondierend oder modifizierend, auf Attribute zugreifen.

5.1.4 Lebenszyklus eines Objekts

Nachdem die Beziehungen zwischen Objekten behandelt wurden, wird der Fokus nun auf das einzelne Objekt[1] verengt.

Jedes Objekt wird zu irgendeinem Zeitpunkt der Programmausführung erzeugt und zu einem späteren Zeitpunkt wieder gelöscht. In der dazwischenliegenden Zeit ist es entweder aktiv, d.h. die Programmausführung befindet sich gerade in einer Methode dieses Objekts, oder passiv.

Für die Objekterzeugung (Konstruktion) sind zwei Schritte nötig:

- Zuerst muß, der Größe des Objekts entsprechend, ein Speicherbereich alloziert werden.

1 Wir gehen von C++-Objekten aus.

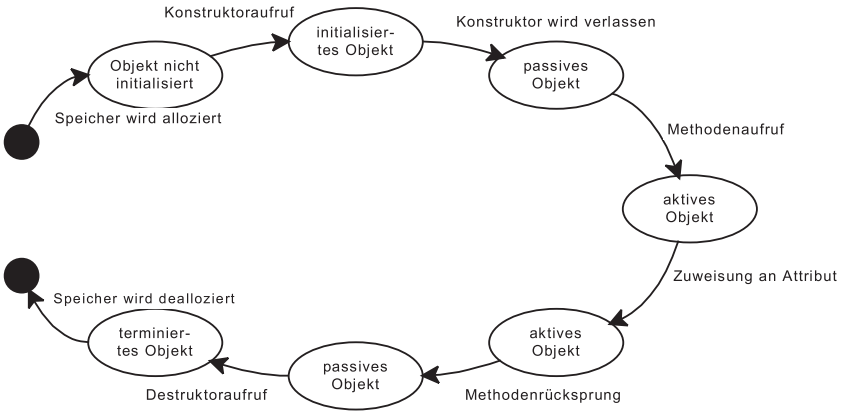


Abb. 9: Lebenszyklus eines Objekts (Bsp.)

- Danach muß das Objekt initialisiert werden. Dazu zählen einerseits technisch bedingte Initialisierungen (z.B. das Anlegen eines Verweises auf die virtuelle Methodentabelle) und andererseits das Ausführen der Initialisierungsanweisungen, die der Programmierer für die Klasse und die Oberklassen des Objekts implementiert hat. Außerdem müssen alle in diesem Objekt enthaltenen Objekte konstruiert werden.

Die beiden Schritte erfolgen in der Regel durch eine Anweisung, in C++ durch die Kombination von new-operator und Name bzw. Signatur des Konstruktors:[2]

```
pObjektName = new tKlassenname(Parameter);
```

Das Allozieren des Speicherbereichs kann entfallen, wenn das Objekt statisch deklariert wurde. In diesem Fall wurde der benötigte Speicher schon zur Compilezeit zur Verfügung gestellt.

Nachdem das Objekt angelegt wurde, ist es so lange passiv, bis eine seiner Methoden aufgerufen wird. Durch den Methodenaufruf wird das Objekt aktiv, und es kann eigene Operationen ausführen oder Methoden anderer Objekte aufrufen. Vor allem kann es in dieser aktiven Phase auch seine Attribute und damit seinen Zustand verändern.

Schließlich wird irgendwann der Destruktor des Objekts aufgerufen. Dies kann durch das Objekt selbst geschehen, erfolgt in der Regel aber von einem anderen Objekt aus. Erst wird der Code des Destruktors selbst durchlaufen, dann folgen die Destruktoren der enthaltenen Objekte und der Oberklassen. Danach kann das Objekt dealloziert werden, d.h., der durch das Objekt belegte Speicher wird wieder freigegeben. Das Schema des Lebenszyklus zeigt Abbildung 9.

Dieser Lebenszyklus gilt auch für verteilte und für persistente Objekte. Verteilte Objekte können innerhalb ihres Lebenszyklus zwischen verschiedenen Orten migrieren. Persistente Objekte können das Programm, in dem sie angelegt wurden, überleben und während eines späteren Programmablaufs wieder aktiviert (und gelöscht) werden. Sie haben also den selben Lebenszyklus, der sich dann aber über mehrere Programmausführungen erstrecken kann. Geht man von einem Von-Neumann-Rechner aus, muß es aber in beiden Fällen Stellvertreter-Objekte (Proxies)

2 Für Namen von Variablen und Typen gilt hier die Konvention, daß Zeigervariablen mit dem Buchstaben *p* beginnen, Typen mit dem Buchstaben *t*.

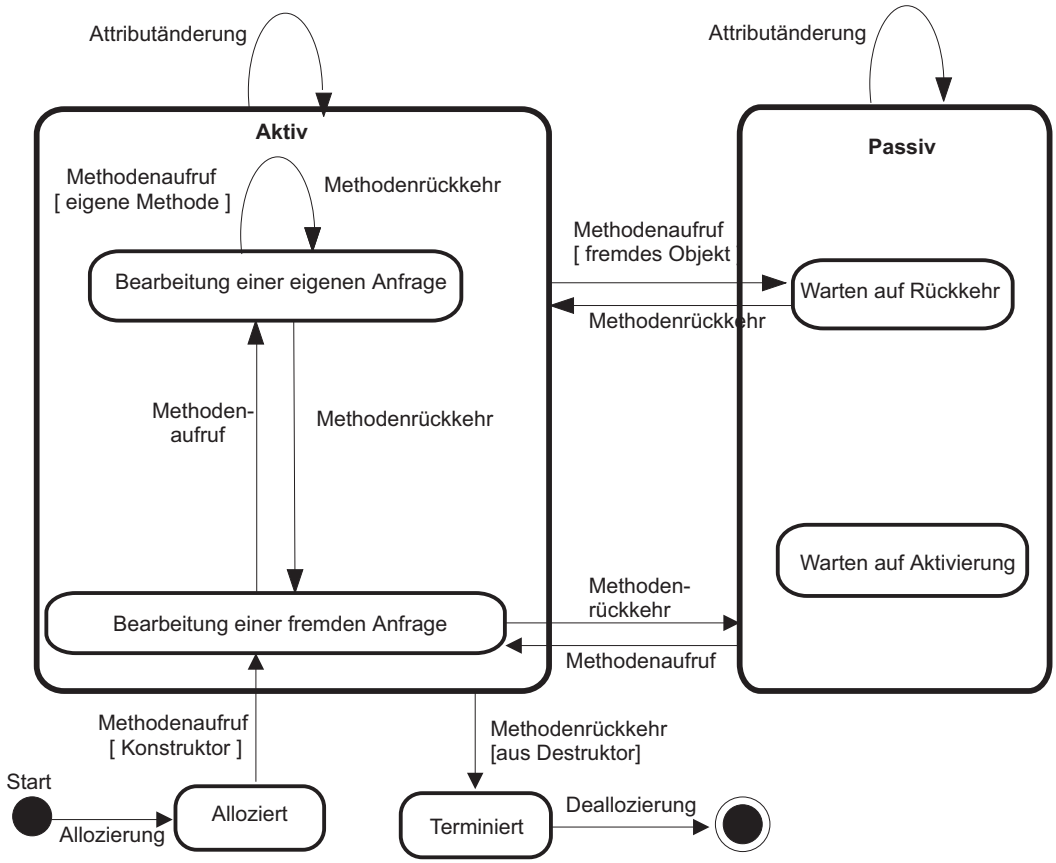


Abb. 10: Zustandsmodell eines Objekts (zur Notation siehe [Rational 97])

geben, die sich im Prozeßraum des ausgeführten Programms befinden und dort die Rolle des entfernten bzw. extern gespeicherten Objekts übernehmen. Diese Stellvertreter sind als „normale“ Objekte lokal zum Programm und überdauern dessen Ausführung nicht.

Mit Hilfe des schematischen Lebenszyklus läßt sich das in Abbildung 10 gezeigte Zustandsdiagramm herleiten. Die meisten Übergänge zwischen den Zuständen des Objekts ergeben sich durch Methodenaufrufe bzw. aus Rücksprüngen aus aufgerufenen Methoden. Zustandsübergänge durch Änderungen von Attributen sind sowohl im aktiven als auch im passiven Zustand möglich; letzteres bedeutet, daß die Attribute eines Objekts von einem fremden Objekt manipuliert werden – das spricht zwar nicht für guten Programmierstil, kommt aber doch vor und ist somit im Zustandsdiagramm berücksichtigt.

Änderungen der Attribute sind von Bedeutung, wenn man neben den Objektinteraktionen auch die Referenzierungs-Relation verfolgen will. In diesem Fall liegen relevante Ereignisse immer dann vor, wenn einer Variablen, in der eine Referenz auf ein Objekt gehalten wird, ein neuer Wert zugewiesen wird; diese Variable ist selbst Attribut eines Objekts.

Wir erhalten damit die Ereignisse, die in der Programmausführung durch unsere objektorientierte Sichtweise erfaßt werden:

- Konstruktion eines Objekts,
- Destruktion eines Objekts,
- Methodenaufruf,

- Methodenrücksprung,
- Attributänderung.

Diese Ereignisse müssen den Werkzeugen, die als Klienten im Visualiserungssystem auf Informationen warten, mitgeteilt werden. Das kann entweder durch das Programmodell oder durch andere Kommunikationswege geschehen.

5.2 Das Metamodell

Von den fachlichen Begriffen, die in den Abschnitten 5.1.1. und 5.1.2. identifiziert wurden, modellieren wir „Klasse“, „Objekt“ und „Methode“ als Elemente des Programmodells, verzichten aber auf eine Modellierung von „Programm“, „Ausführung“ und „Nachricht“:

- Programm ist nach der verwendeten Beschreibung ein sehr technischer, Ausführung ein sehr abstrakter Begriff. Die Eigenschaften, die für die objektorientierte Visualisierung interessant sind, befinden sich bei den Elementen von Programm und Ausführung, nämlich bei Klassen, Objekten, Methoden und Nachrichten.
- Nachrichten werden nicht als Programmodell-Elemente abgebildet, denn die Idee des Programmodells war darin begründet, jeweils den aktuellen Zustand der Programmausführung nachzubilden. Eine Nachricht ist konzeptuell aber nur ein Ereignis ohne zeitliche Ausdehnung und findet sich daher im Programmodell nicht wieder.

Doch auch wenn die Nachrichten nicht als Elemente des Modells auftreten, sind sie ein entscheidender Teil der visualisierten Programmausführung und müssen den Visualisierungswerkzeugen bekanntgemacht werden. Wir fordern daher, daß der Automat die Werkzeuge durch einen Ereignismechanismus informiert, wenn in der Programmausführung eine relevante Nachricht aufgetreten ist.

Ein Ereignis versendet der Automat ebenfalls, wenn sich der Zustand des Programmodells ändert. Da das Programmodell neben den statischen Elementen für Klassen und Methoden als dynamische Elemente lediglich Objekte abbildet, tritt eine Zustandsänderung des Modells also genau dann auf, wenn sich ein Element der Menge aller Objekte ändert.

Somit bleiben mit Klassen, Objekten und Methoden also drei grundlegende Typen von Programmodell-Elementen. Um diese Typen zu beschreiben, gibt es die Klassen `tClass`, `tObject` und `tFunction`. Zusammen ergeben sie das *Metamodell*: Ein Modell, das die notwendigen Bausteine zur Verfügung stellt, um das Verhalten eines objektorientierten Programms abzubilden.

5.2.1 Modellierung von Klassen

Für die Beschreibung von Klassen werden im Metamodell Objekte vom Typ `tClass` benutzt. Die Schnittstelle von `tClass` lautet:

```
enum tClassType { CLASS, STRUCT, UNION };
enum tAccess { PUBLIC, PROTECTED, PRIVATE };

class tClass
{
   public:
```

```
    const string& GetName(void);
    const string& GetFullName(void);

    bool HasConstructor(void);
    bool HasDestructor(void);

    tClassType GetType(void);

    const vector<const tInheritanceInfo*>* GetBaseClasses(void);
    const vector<const tContainmentInfo*>*
                          GetContainedObjectClasses(void);
    const vector<const tAssociatedClassInfo*>*
                          GetAssociatedClasses(void);

    const vector<const tFunction*>* GetMethods(void);

    const tFile* GetFile(void);
    int GetSize(void);
};
```

Den Namen der von einer `tClass`-Instanz beschriebenen Klasse erhält man durch die Methode `GetName`. Da C++ es zuläßt, eine Klasse innerhalb der Klassendefinition einer anderen Klasse zu deklarieren, liefert die Methode `GetFullName` den Namen der Klasse einschließlich aller umgebenden Klassen. Definiert man z.B. innerhalb einer Klasse `A` eine zweite Klasse `B`, so erhält man für `B` mittels GetName den Wert „B“ und mittels `GetFullName` den Wert „A::B“; innerhalb eines Programms sind die von `GetFullName` zurückgegebenen Klassennamen immer eindeutig.

In C++ gibt es die Konstrukte Struktur (`struct`) und Union (`union`), die konzeptuell keine Klasse sind, sich technisch aber genauso verhalten. Unter Umständen geht aus den Debug-Informationen nicht einmal hervor, um welches der drei Konstrukte es sich handelt, so z.B. beim STABS-Format [Menapace et al. 93]. Daher gehen wir davon aus, daß sich `tClass` sowohl auf Klassen, Strukturen und Unions beziehen kann, und sehen die Methode `GetType` vor, die den richtigen Typ angibt. Im Zweifelsfall werden Strukturen und Unions dann als Klassen aufgefaßt.

Die Methoden `HasConstructor` and `HasDestructor` geben an, ob für die Klasse Konstruktoren bzw. Destruktoren definiert wurden. Im nächsten Kapitel wird erklärt werden, daß das Vorhandensein der Konstruktoren und Destruktoren eine Voraussetzung für die Erfassung von Laufzeitinformationen ist. Ebenfalls technisch erforderlich ist die Methode `GetSize`, die die Größe des Speicherbereichs angibt, den ein Objekt der Klasse beansprucht.

Kommen wir nun zu den Relationen, die zwischen Klassen bestehen können. Über Vererbungsbeziehungen gibt die Methode `GetBaseClasses` Auskunft, indem sie alle Basisklassen liefert. Der Rückgabewert ist ein Vektor, der für jede direkte Basisklasse eine Struktur vom Typ `tInheritanceInfo` enthält:

```
struct tInheritanceInfo
{
    tClass*  pClass;
    bool     IsVirtual;
    tAccess  Access;
```

```
   int      Offset;
};
```

Neben einem Verweis auf das `tClass`-Objekt, das die Basisklasse beschreibt, enthält die Struktur die Attribute `IsVirtual` und `Access`, die angeben, ob die Vererbung öffentlich (public), geschützt (protected) oder privat bzw. virtuell oder nicht-virtuell erfolgt (siehe z.B. [Stroustrup 92]). Die Variable `Offset` bezieht sich auf das Speicherlayout von Objekten: In den Speicherbereich, der von einem Objekt belegt wird, ist für jede Basisklasse ein Datenbereich eingebettet, der als ein Objekt der Basisklasse interpretiert werden kann. `Offset` gibt an, an welcher Adresse (relativ zur Adresse des abgeleiteten Objekts) dieser Bereich beginnt. Wie die Methode `GetSize` ist `Offset` aus technischen Gründen für die Informationserfassung notwendig.

Containment-Beziehungen führen zu einem ähnlichen Objektlayout, in dem der Speicherbereich eines Objekts ein vollständiges anderes Objekt einschließt. `GetContainedObjectClasses` erzeugt einen Vektor mit Informationen über die enthaltenen Objekttypen. Die Struktur, die dafür benutzt wird, ist ähnlich wie `tInheritanceInfo`:

```
struct tContainmentInfo
{
   tClass*  pClass;
   tAccess  Access;
   int      Offset;
};
```

Die Bedeutung der Felder von `tContainmentInfo` ist analog zu den gleichnamigen Feldern in `tInheritanceInfo`.

Die Methode `GetAssociatedClasses` bezieht sich auf die Verwendungs-Relationen zu anderen Klassen. Sie liefert einen Vektor aller Instanzenvariablen, mit denen in Objekten der Klasse andere Objekte referenziert werden. Dazu wird die Struktur `tAssociatedClassInfo` benutzt:

```
struct tAssociatedClassInfo
{
   tClass*  pClass;
   tAccess  Access;
   int      Offset;
};
```

Die Felder sind identisch mit denen von `tContainedInfo`, müssen aber etwas anders interpretiert werden: An der mit `Offset` bezeichneten relativen Adresse liegt nun nicht mehr ein vollständiges Objekt, sondern lediglich eine Zeigervariable; diese Variable referenziert ein Objekt, dessen statischer Typ durch `pClass` beschrieben wird; der dynamische Typ kann eine davon abgeleitete Klasse sein.

Es bleiben noch die Generizitäts- und die Metaklassenbeziehungen: Letztere entfällt, weil C++ keine Metaklassen kennt. Generische Klassen haben wir nicht berücksichtigt, weil die Klassenschablonen (templates), die vom Compiler zur Instanziierung generischer Klassen verwendet werden, weder im Programm noch während dessen Ausführung wiederzufinden sind. Sollten wir später feststellen, daß explizite Informationen über Generizität von Nutzen sind, müßte eine Klasse eingeführt werden, die Klassenschablonen beschreibt, und `tClass` muß dann auf eine

eventuell benutzte Klassenschablone und auf die zur Instanziierung benutzten Klassen verweisen.

Die Methoden, über die die von einem `tClass`-Objekt beschriebene Klasse verfügt, werden gesondert in Objekten des Typs `tMethod` beschrieben (s.u.).

Die Methode `GetFile` gibt an, in welcher Quelldatei die Klasse definiert wurde.

5.2.2 Modellierung von Methoden

Das Metamodell stellt die Klasse `tFunction` zur Verfügung, mit der Eigenschaften von Methoden beschrieben werden können. Sie kann auch für herkömmliche, d.h. nicht objektgebundene Funktionen verwendet werden.

```
class tFunction
{
   public:
   const string& GetName(void);
   const string& GetArgs(void);
   bool IsLocal(void);
   bool IsMember(void);
   const tClass* GetClass(void);
   bool IsVirtual(void);
   bool IsConstructor(void);
   bool IsDestructor(void);
   bool IsGenerated(void);
   bool IsStatic(void);
   tAccess GetAccess(void);

   tFile* GetDefFile(void);
   unsigned int GetLineNo(void);
   tSourceFile* GetSourceFile(void);
   tObjectFile* GetObjectFile(void);

   bool HasDebugInfo(void);
};
```

Objekte der Klasse `tFunction` beschreiben Methoden. Name und Signatur der Methode werden mittels `GetName` und `GetArgs` ermittelt. Mit `IsLocal` wird erfragt, ob die Methode lokal innerhalb einer anderen Methode definiert wurde.

In C++ können neben den objektgebundenen Methoden auch herkömmliche, d.h. nicht an Klassen gebundene Funktionen auftreten, die für die dynamsiche Analyse ebenfalls interesant sein können. Daher bilden wir alle Funktionstypen in `tFunction` ab. Die Methode `tFunction::IsMember` gibt Auskunft, ob es sich bei einer Funktion um eine Methode (member function) handelt. Ist dies der Fall, liefert `GetClass` das zugehörige `tClass`-Objekt, und `IsConstructor` bzw. `IsDestructor` geben an, ob es sich um einen Konstruktor oder Destruktor handelt. Da C++-Compiler unter Umständen selbsttätig Konstruktoren und Destuktoren erzeugen, ist `IsGenerated` vorgesehen, um festzustellen, ob die Methode vom Programmierer oder vom Compiler erstellt wurde (vgl. [Lippman 96], S. 32). Schließlich wird durch `IsStatic` erfragt, ob es sich um eine Klassen- oder Instanzenmethode handelt.[3] `GetAccess` gibt an, ob die Methode in ihrer Klasse privat, geschützt oder öffentlich deklariert wurde.

Die Quelldateien, in denen die Funktion oder Methode deklariert und definiert wurde, erhält man mit `GetDefFile` und `GetSourceFile`. `GetLineNo` bezeichnet die Programmzeile, in der die Implementierung der Funktion beginnt. `GetObjectFile` gibt den Namen der compilierten Objektdatei an, in der sich der Funktionscode befindet.

Ein rudimentäres `tFunction`-Objekt kann auch für Funktionen angelegt werden, zu denen das übersetzte Progremm keine Debug-Informationen enthält, denn die Namen von Funktionen werden in jedem Fall in den übersetzten Code hineingeschrieben, um anhand der Namen das Programm linken zu können. Dann bleiben aber die meisten in `tFunction` erfaßten Informationen unzugänglich, so daß die Methode `HasDebugInfo` benötigt wird, um festzustellen, ob es eine vollständige gültige Beschreibung der Funktion gibt.

5.2.3 Modellierung von Objekten

Auch wenn die modellierten Objekte die wichtigsten (oder zumindenst hinsichtlich des Laufzeitverhaltens interessantesten Objekte) sind, besitzt die Metamodell-Klasse `tObject` die einfachste Schnittstelle:

```
class tObject
{
   public:
   tClass* GetClass(void);
   string GetName(void);
   long GetID(void);
};
```

Die geringe Anzahl der Methoden ist darauf zurückzuführen, daß die meisten Informationen über Objekte bereits bei der jeweiligen Klasse erfragt werden können. Um welche Klasse es sich handelt, teilt ein Objekt über die Methode `GetClass` mit. Die Methode `GetName` liefert den Namen eines Objekts (in der bisherigen Implementierung der Klassenname). Um die Objekte eindeutig zu kennzeichnen, wird ein Objektidentifizierer (OID) benutzt, der durch `GetID` ermittelt werden kann.

Wenn ein Programm ausgeführt wird, sind die meisten Elemente, die im Programmodell existieren, Instanzen von `tObject`. Sie werden in dieser Arbeit auch als *Metaobjekte* bezeichnet. Den Ausdruck habe ich bereits im vorangegangen Kapitel benutzt, um die Objekte des Programms und ihre Gegenstücke im Programmodell voneinander zu unterscheiden. An dieser Stelle wird also der Ursprung des Begriffs deutlich: Metaobjekte sind Instanzen der Metamodell-Klasse `tObject`.

5.3 Zusammenfassung

Das Metamodell stellt die Bausteine zur Verfügung, aus denen sich das Programmodell zusammensetzt. Es gibt drei grundlegende Typen, die der Beschreibung von Klassen, von Methoden und von Objekten dienen. Gemäß der Unterscheidung nach Elementen erster und zweiter Ordnung handelt es sich bei den drei Typen um die erste Kategorie. Abschnitt 7.2 wird anhand eines

[3] Klassenmethoden (static member functions) sind für eine Klasse definiert und werden im Gegensatz zu den üblichen Objektmethoden nicht für eine Instanz der Klasse, sondern für die Klasse selbst aufgerufen.

Beispiels zeigen, wie das Metamodell um zusätzliche Elemente zweiter Ordnung erweitert werden kann.

Kapitel 6
Automat zur Erfassung von Laufzeit-Informationen

6.1 Aufgaben des Automaten

Der Automat, der für die Erfassung der Laufzeitinformationen zuständig ist, wurde im vierten Kapitel als Teil der Systemarchitektur vorgestellt. Wir haben dem Automaten den Namen *ControlEngine* gegeben, weil er der „Motor“ des Visualisierungssystems ist und die Instanz darstellt, die das visualisierte Programm kontrolliert (wobei diese Kontrolle nach strengen Regeln erfolgt und von den Werkzeugen abhängig ist, die auf den Automaten aufsetzen).

Die Aufgaben der ControlEngine sind im einzelnen:

- Das Ermitteln der statischen Informationen zu einem Programm, d.h., es müssen die `tClass` und die `tFunction`-Objekte des Programmodells angelegt werden.
- Das Ausführen des Programms, d.h. Starten und Unterbrechen.
- Das Ermitteln von dynamischen Informationen während der Laufzeit, d.h. Kenntnis darüber zu erhalten, welche Objekte erzeugt und gelöscht werden, wie sie miteinander interagieren und ihre Zustände ändern.
- Die Pflege des Programmodells während der Programmausführung, d.h., zu den in der Programmausführung entstehenden Objekten müssen `tObject`-Instanzen im Modell angelegt (respektive wieder gelöscht) werden.
- Das Benachrichtigen der Visualisierungswerkzeuge, wenn sich der Zustand des Programmodells geändert hat.

In diesem Kapitel werden die wesentlichen technischen Voraussetzungen behandelt, unter denen die statischen und dynamischen Programminformationen erfaßt werden können. Weiterhin wird die Schnittstelle erklärt, mit dem sich der Automat den Visualisierungswerkzeugen präsentiert. Ein Beispiel demonstriert schließlich das Schema, nach dem der Automat von Werkzeugen benutzt wird.

Stephen Friedrich hat die ControlEngine entworfen und implementiert und wird in einer separaten Arbeit eine detailliertere Beschreibung des Automaten, dessen Funktionalität und Architektur geben. Ich beschränke mich in diesem Kapitel auf die Informationen, die für das Verständnis der vorliegenden Arbeit benötigt werden.

6.2 Technische Grundlagen

6.2.1 Statische Informationserfassung

Die ControlEngine muß statische Informationen erfassen, um daraus die Metamodell-Elemente der Typen `tClass` und `tFunction` anzulegen. Dazu werden Klassen- und Funktionsbeschreibungen benutzt, die vom Compiler als Debug-Informationen in den Code eines übersetzten Programms geschrieben werden. Der Compiler codiert die Beschreibungen in kompakten Zeichenfolgen, die zwischen dem ausführbaren Code stehen. Um die Informationen zu nutzen, müssen diese Zeichenfolgen vor der Programmausführung geparst und decodiert werden.

Das Format der Debug-Informationen ist primär von Betriebssystem- und Compilerversionen abhängig und erst sekundär von der benutzten Programmiersprache. Unter Unix ist das STABS-Format gebräuchlich, das für verschiedene Programmiersprachen benutzt wird [Menapace et al. 93]. Natürlich kennt das Format für die unterschiedlichen Sprachen spezifische Konstrukte; es gibt aber eine Basis von sprachunabhängigen STABS-Codes. In diesen sind z.B. für Variablen die Speicheradressen und Variablennamen codiert, für benutzerdefinierte Typen der Aufbau der Typen und für Funktionen Funktionsname, Adresse und Signatur. Weil das STABS-Format älter als C++ ist, wurden die bestehenden Konventionen um die zusätzlichen Spracheigenschaften ergänzt, womit die Eigenschaften der Programmiersprache leider nicht an allen Stellen zufriedenstellend erfaßt werden. So geht aus den STABS-Anweisungen beispielsweise nicht hervor, ob ein C++-Typ als Klasse, Struktur oder Union definiert wurde. Diese Zweifelsfälle sind zum Glück Ausnahmen, so daß man davon ausgehen kann, daß aus den Debug-Anweisungen korrekte Klassen- und Methodenbeschreibungen erzeugt werden können.

6.2.2 Dynamische Informationserfassung

Als dynamische Informationen müssen zur Laufzeit diejenigen Ereignisse erkannt werden, die laut Abschnitt 5.1.4 für die Objektvisualisierung relevant sind, nämlich Konstruktion und Destruktion von Objekten, Methodenaufrufe und -rücksprünge sowie Zuweisung von Objektreferenzen.

Würde man den Quelltext des untersuchten Programms instrumentieren, könnte man die für diese Ereignisse in Frage kommenden Stellen leicht identifizieren und dort zusätzlich Anweisungen zur Informationserfassung einfügen. Da wir uns aber gegen die Instrumentierung entschieden haben (vgl. Abschnitt 4.4.2), kann nur „von außen“ auf das Programm zugegriffen werden. Das bedeutet, daß das Programm geladen und als eigener Prozeß ausgeführt wird. Die Ausführung wird unterbrochen, wenn ein zuvor gesetzter Haltepunkt erreicht ist. Der Zustand des unterbrochenen Programms kann analysiert werden, indem man den sogenannten *Stack Frame* untersucht: Dieser enthält die Registerbelegung des Mikroprozessors und damit auch den *Program Counter*, dessen Wert angibt, an welcher Stelle sich die Programmausführung gerade befindet. Daraus muß man nun ableiten, wie die Unterbrechung zu interpretieren ist, welche Daten zu untersuchen sind usw. Dann wird die Anweisung gegeben, die Programmausführung fortzusetzen, bis der nächste Haltepunkt angetroffen wird.

Wie wendet man dieses prinzipielle Vorgehen nun an, um die relevanten Ereignisse zu erhalten? Am einfachsten läßt es sich für das Ereignis „Methodenaufruf“ erklären: In den Debug-Informationen findet man die Einsprungadresse einer Methode; diese Adresse markiert die Stelle, zu der bereits alle Parameter des Aufrufs in den aktuellen Stack Frame kopiert, aber noch keine Anweisungen der Methode ausgeführt wurden. Die ControlEngine muß auf diese Adresse

einen Breakpoint setzen. Wenn die Adresse erreicht wird, erhält die ControlEngine eine Benachrichtigung des Betriebssystems, womit die Unterbrechung signalisiert wird. Die ControlEngine läßt sich nun den Stack Frame der Programmausführung mitteilen und wertet ihn aus: Aus dem Wert des Program Counters kann sie ableiten, in welcher Methode die Unterbrechung erfolgt ist. Damit ist bekannt, welche Methode aufgerufen wird, aber noch nicht, für welches Objekt dies geschehen ist. Letzteres geht aus einem der Register im Stack Frame hervor, das vereinbarungsgemäß die Adresse des betroffenen Objekts enthält. Diese Objektadresse wird in C++ als *this-Pointer* bezeichnet. Die ControlEngine muß nun im Metamodell nach einem Objekt mit dem ermittelten this-Pointer suchen und kann dann an alle Visualisierungswerkzeuge ein Ereignis versenden, das sich informell mit den Worten „Methode *m* wurde für Objekt *o* aufgerufen“ beschreiben läßt. Danach veranlaßt die ControlEngine die Fortsetzung des Programmablaufs.

Anhand dieses Anwendungsfalles erkennt man zwei charakteristische Eigenschaften:

- Der Automat arbeitet nicht auf der Basis von Objekten, sondern von Methoden. Wenn ein Werkzeug alle Aktivitäten eines einzigen Objekts verfolgen will, müssen auf alle Methoden der Klassen Haltepunkte gesetzt werden. Die Unterbrechung erfolgt dann unabhängig vom Objekt, und es ist Aufgabe der ControlEngine oder des Werkzeugs zu überprüfen, ob die Methode für das zur Diskussion stehende Objekt oder für ein anderes Objekt der Klasse aufgerufen wurde.
- Die Identifizierung von Objekten erfolgt über den this-Pointer, der beim Methodenaufruf übergeben wird. Die Identifizierung wird dadurch erschwert, daß die Zuordnung von Objekten und this-Pointern in keiner Richtung eindeutig ist: Unterschiedliche Objekte können denselben this-Pointer besitzen, wenn das eine Objekt im anderen enthalten ist, und dasselbe Objekt kann bei Mehrfachvererbung unter verschiedenen this-Pointern auftreten (vgl. [Hamer und Friedrich 96]).

Analog wird vorgegangen, um Methodenrücksprünge zu verfolgen. Auch für die Konstruktion und die Destruktion von Objekten wird dasselbe Prinzip genutzt, indem die Konstruktion als Spezialfall eines Methodenaufrufs (in Form des Konstruktoraufrufs) angesehen wird; für die Destruktion gilt das Entsprechende.

Damit wird implizit vorausgesetzt, daß eine Objekterzeugung einem Konstruktoraufruf gleichkommt. Das ist in der C++-Programmierung aber nicht zwingend notwendig, denn es können auch Objekte erzeugt werden, ohne daß die Klasse über einen Konstruktor verfügen muß. Somit entfällt für konstruktorlose Klassen die Möglichkeit, Objekterzeugung durch Haltepunkte zu beobachten. Wir fordern daher, daß eine Klasse, deren Objekte von der ControlEngine erfaßt werden sollen, über Konstruktoren (und entsprechend über Destruktoren) verfügen *muß*. Diese Forderung klingt zwar nach einer starken Einschränkung, wird aber in der Realität dadurch relativiert, daß Compiler an vielen Stellen Default-Konstruktoren und -Destruktoren erzeugen. Beispielsweise geschieht dies bereits, wenn eine Klasse eine virtuelle Methode besitzt (vgl. [Hamer und Friedrich 96], [Lippman 96] S. 32 ff.).

Von den fünf identifizierten Ereignistypen bleibt noch die Attributänderung: Hier versagt das bei den übrigen Ereignisse anwendbare Schema. Bei den vorangegangen Fällen ist ausreichend, Haltepunkte an den Einsprungadressen der Methoden zu setzen. Dahingegen muß man eine ungleich höhere Zahl von Haltepunkten benutzen, um Attributänderungen zu erfassen: Prinzipiell muß die Programmausführung bei jeder Zuweisung unterbrochen werden. Aus Performance-Gründen ist ein solches Vorgehen nicht vertretbar. Ich werde diese Problematik in der vorlie-

genden Arbeit offenlassen, da das Beobachten von Attributänderungen für das im weiteren Verlauf konzipierte Werkzeug nicht notwendig ist.

6.3 Schnittstelle der ControlEngine

6.3.1 Unterscheidung von Ereignissen

Wenn die Attributänderungen vernachlässigt werden, bleiben folgenden Ereignisse, die die ControlEngine berücksichtigen muß:

- Konstruktion und
- Destruktion von Objekten sowie
- Methodenaufrufe und
- Methodenrücksprünge.

Daß Objektkonstruktion und -destruktion als Aufrufe spezieller Methoden erfaßt werden, ist ein internes technisches Detail; aus fachlicher Sicht liegen unterschiedliche Kategorien von Ereignissen vor. In die Schnittstelle der ControlEngine müssen Konstruktion und Destruktion deswegen explizit aufgenommen werden.

Um die Anzahl der zu verarbeitenden Ereignisse in vernünftigem Rahmen zu halten, muß die ControlEngine einstellbar sein, d.h., ihr wird mitgeteilt, welche Ereignisse sie verfolgen soll. Visualisierungswerkzeuge, die die Dienste der ControlEngine in Anspruch nehmen, melden dazu an, über welche Ereignisse sie informiert werden wollen.

Die Anmeldung für Ereignisse muß bereits differenziert erfolgen und nicht etwa nach einem „Alles oder nichts“-Verfahren, bei dem sich ein Werkzeug nur pauschal für Objektkonstruktionen anmelden kann und danach mit Konstruktionsereignissen überflutet wird. Die naheliegendste Art, Ereignisse zu differenzieren, richtet sich nach den Klassen; im Falle der Konstruktion soll sich ein Werkzeug also für eine bestimmte Klasse anmelden, so daß es immer dann benachrichtigt wird, wenn ein Objekt dieser Klasse erzeugt wird. Nachdem ein Objekt erzeugt worden ist, kann man verlangen, daß eine Anmeldung für Ereignisse möglich sein soll, die genau dieses Objekt betreffen (sinngemäß wird die ControlEngine z.B. von einemWerkzeug beauftragt: „benachrichtige mich, wenn die Methode *m* für das Objekt *o* aufgerufen wird“).

Wenn sich Werkzeuge also für Ereignisse anmelden, kann die Anmeldung abhängig vom Ereignistyp weiter präzisiert werden, indem eine Klasse oder ein bestimmtes Objekt, bei Methodenaufrufen und -rücksprüngen auch die jeweilige Methode angeben wird.

<table>
<tr><th>Ereignistyp</th><th colspan="3">Anmeldung kann erfolgen für...</th></tr>
<tr><td>Konstruktion</td><td rowspan="4">bestimmte Klasse</td><td colspan="2" rowspan="2"></td></tr>
<tr><td>Destruktion</td></tr>
<tr><td>Methodenaufruf</td><td rowspan="2">bestimmtes Objekt</td><td rowspan="2">bestimmte Methode</td></tr>
<tr><td>Methodenrücksprung</td></tr>
</table>

Tab. 3: Ereignistypen

Natürlich könnte man die Anmeldung noch nach weiteren Kriterien differenzieren, so daß Werkzeuge nur unter bestimmten Bedingungen über Ereignisse informiert werden. Bedingungen könnten z.B. den Zustand von Objekten betreffen (etwa: „benachrichtige mich, wenn ein Objekt der Klasse *k* gelöscht wird und für dieses Objekt *o* die Bedingung *b(o)* gilt“).

Wir wollen uns jedoch auf die elementaren Ereignisse beschränken, so wie auch das Metamodell zunächst nur die elementaren Konstrukte – Klasse, Objekt, Methode – kennt. In Abschnitt 5.2 wurde bereits die Möglichkeit werkzeugspezifischer Schnittstellen für das Metamodell erwähnt; in derselben Art könnte man die ControlEngine um spezifische Schnittstellen für spezialisierte Ereignisse erweitern. Vor diesem Hintergrund bietet es sich an, auch die elementaren Ereignissen in Schnittstellen zu kapseln, die von der „Kernfunktionalität“ der ControlEngine (nämlich dem Laden und Ausführen eines Programms) getrennt sind. Wir sehen zwei Schnittstellen vor: eine für Kon- und Destruktion, die zweite für die Beobachtung von Methoden. Zusammen mit dem eigentlichen Kern der ControlEngine werden wir den Automaten in drei Klassen modellieren:

- Die Klasse `tControlEngine`, die Operationen zum Laden, Starten und Unterbrechen eines Programms bereitstellt,
- die Klasse `tClassObserver`, deren Schnittstelle den Werkzeugen ermöglicht, sich für die Beobachtung von Objektkonstruktion und Destruktion anzumelden, und
- die Klasse `tFunctionObserver`, bei der sich Werkzeuge für die Beobachtung von Funktionen und Methoden anmelden können.

Diese drei Klassen werden in den folgenden Abschnitten beschrieben.

6.3.2 tControlEngine

Die Klasse `tControlEngine` wird ebenso wie die Klassen `tClassObserver` und `tFunctionObserver` als Singleton [Gamma et al. 95] implementiert, d.h. es gibt von der Klasse immer genau eine Instanz. Über die Klassenmethode `Get` erhält man einen Zeiger auf die Instanz. Die Schnittstelle von `tControlEngine` lautet:

```
class tControlEngine
{
   public:
   // open an executable, given a pathname
   static tControlEngine* Get(const string& Executable);

   void SetArguments(const string& Arguments = "") ;
   // run or continue the program
   void Run(void);
   // stop the program as soon as possible
   // (if currently stopped on an event of interest:
   //  don't automatically resume execution)
   void Stop(void);
   void Reset(void);
};
```

Die zum Singleton-Muster gehörende `Get`-Methode erhält als Parameter den Namen des Programms, das geladen werden soll. Bevor das Programm gestartet wird, kann man mit `SetArguments` Kommandozeilenparameter setzen, die dem Programm übergeben werden. Der Start der Programmausführung erfolgt durch den Aufruf von `Run`; zur Unterbrechung wird `Stop` auf-

gerufen. Danach läßt sich die Ausführung mit einem weiteren `Run`-Aufruf fortsetzen. Die Methode `Reset` setzt das Programm nach einer Unterbrechung zurück, so daß das nächste `Run` zu einem Neustart des Programms führt.

6.3.3 tClassObserver

Die Klasse `tClassObserver` ist die Schnittstelle zu den Konstruktions- und Destruktionsereignissen. Ihre Aufgabe besteht darin, `tObject`-Instanzen im Programmodell anzulegen und zu löschen, und Werkzeuge, die sich für Ereignisse angemeldet haben, beim Eintreten der Ereignisse zu benachrichtigen. Die Benachrichtigung erfolgt nach dem Entwurfsmuster „Functor" [Coplien 92]; dazu werden von der Klasse `tStructionEvent` Funktionsobjekte *(functors)* instanziiert und vom Werkzeug an das `tClassObserver`-Objekt übergeben. Die Funktionsobjekte beinhalten, welche Methode ausgeführt werden soll, wenn ein Ereignis auftritt, und welche Parameter der Methode zu übergeben sind.

Die Klasse `tClassObserver` besitzt das folgende Interface:

```
class tClassObserver
{
   public:
   static tClassObserver* Get(tControlEngine* pObserver);

   // get notified when objects are constructed and destructed:
   void Register(tClass* pClass,
      tStructionEvent* pUponConstruction,
      tStructionEvent* pUponDestruction);

   void Unregister(tClass* pClass,
      tStructionEvent* pUponConstruction,
      tStructionEvent* pUponDestruction);

   void RegisterForAllObjects
      (tStructionEvent* pUponConstruction,
       tStructionEvent* pUponDestruction);

   void UnregisterForAllObjects
      (tStructionEvent* pUponConstruction,
       tStructionEvent* pUponDestruction);

   // keep track of object constructions and destructions,
   // but don't send events
   void Trace(tClass* pClass);
   void Untrace(tClass* pClass);

   void TraceAllObjects(void);
   void UnTraceAllObjects(void);
};
```

Die `Get`-Methode taucht auch hier als Teil des Singleton-Musters auf (vgl. `tControlEngine`). Mit der Methode `Register` melden sich Werkzeuge für eine bestimmte Klasse an, d.h., sobald ein Objekt der Klasse erzeugt oder gelöscht wird, wird ein korrespondierendes Objekt im Programmodell angelegt bzw. entfernt, und das Werkzeug wird benachrichtigt. Die Methoden, die zur Benachrichtigung am Werkzeug aufgerufen werden sollen, werden in den Funktionsobjekten `pUponConstruction` und `pUponDestruction` übergeben. Die Methode `Regi-`

sterForAllObjects wird benutzt, um sich für die Beobachtung aller Klassen anzumelden. Mit Aufrufen von Unregister und UnregisterForAllObjects können sich Werkzeuge wieder abmelden und werden fortan nicht mehr benachrichtigt.

Da damit nur die Objektkonstruktionen und -destruktionen erfaßt werden, für die auch Benachrichtigungen an Werkzeuge ausgehen sollen, werden noch zusätzlich Methoden eingeführt, um auch ohne Benachrichtigungen Objekte im Programmodell abzubilden. Die Methoden Trace und TraceAllObjects veranlassen, daß für die Objekte einer bestimmten bzw. aller Klassen entsprechende Programmodell-Objekte vorhanden sein sollen. Mit Untrace und UnTraceAllObjects lassen sich die Anweisungen stornieren.

6.3.4 tFunctionObserver

Bei der Klasse tFunctionObserver können sich Werkzeuge anmelden, um Funktionen und insbesondere Methoden zu beobachten. Als Call-Back-Mechanismus dienen auch hier Funktionsobjekte (Funktoren), in denen die Instanz mitgeteilt wird, die beim Aufruf einer Funktion aktiviert werden soll. Optional kann auch die Rückkehr aus einer Funktion gemeldet werden.

Wenn es sich bei der zu beobachtenden Funktion um eine Methode handelt, reicht es aus, bei der Anmeldung zur Beobachtung nur das tFunction-Objekt zu übergeben; da dieses implizit auf die zugehörige Klasse verweist, muß die Klasse nicht extra angegeben werden.

Die Schnittstelle von tFunctionObserver beschränkt sich zunächst darauf, daß sich Werkzeuge als Beobachter für eine bestimmte Funktion oder für alle Funktionen anmelden können. Zukünftige Erweiterungen könnten die Anmeldung für alle Methoden einer Klasse oder für den Methodenaufruf an einem bestimmten Objekt beinhalten.

Die Schnittstelle der Klasse lautet folgendermaßen:

```
class tFunctionObserver
{
   public:
   static tFunctionObserver* Get(tControlEngine* pObserver);

   void Register(tFunction* pFunction,
      tFunctionEvent* pUponEntry);
   void Register(tFunction* pFunction,
      tFunctionEvent* pUponEntry, tFunctionEvent* pUponExit);
   void Unregister(tFunction* pFunction,
      tFunctionEvent* pUponEntry);
   void Unregister(tFunction* pFunction,
      tFunctionEvent* pUponEntry, tFunctionEvent* pUponExit);

   // get notified on every method call of every object
   // first object is sender, second recipient
   void RegisterForAllMethods
      (tFunctionEvent* pUponEntry);
   void RegisterForAllMethods
      (tFunctionEvent* pUponEntry, tFunctionEvent* pUponExit);
   void UnregisterForAllMethods
      (tFunctionEvent* pUponEntry);
   void UnregisterForAllMethods(tFunctionEvent* pUponEntry,
   tFunctionEvent* pUponExit);
};
```

Die `Register`- und `RegisterForAllMethods`-Operationen liegen mit je zwei Signaturen vor: Es können entweder ein oder zwei Funktoren übergeben werden. Der erste beinhaltet die zu aktivierende Adresse im Falle eines Methodenaufrufs, die zweite (und optionale) die entsprechende Adresse für die Rückkehr aus einer Methode. Die Funktoren gehören zur Klasse `tFunctionEvent`, in der auch die Signatur des Call-Back-Aufrufs definiert ist. Die Signatur hat die Form *(Objekt o_1, Methode m_1, Objekt o_2, Methode m_2)*. Mit diesen Parametern wird dem benachrichtigten Werkzeug mitgeteilt, daß die Methode m_2 für das Objekt o_2 aufgerufen wurde und daß dieser Aufruf von der Methode m_1, die für Objekt o_1 aktiv war, ausging.

Während `RegisterForAllMethods` die Beobachtung aller Funktionen und Methoden auslöst, bezieht sich das einfache `Register` auf genau eine Methode, die als erster Parameter übergeben wird. Die Abmeldung geschieht jeweils durch `Unregister`-Aufrufe.

6.3.5 Beispiel

Zum Schluß dieses Abschnitts soll ein Beispiel vereinfacht darstellen, wie die ControlEngine benutzt wird. Wir gehen dazu von einem Werkzeug aus, das in einem Konstruktionsdiagramm alle erzeugten Objekte anzeigt und die Objekte aus der Anzeige entfernt, sobald sie im ausgeführten Programm gelöscht werden. Daraus entsteht folgendes Szenario:

Das Werkzeug ruft `tControlEngine::Get(Programmname)`, woraufhin die ControlEngine das Programm *Programmname* lädt. Das Werkzeug holt sich nun mit `tClassObserver::Get` einen Zeiger auf die Instanz von `tClassObserver` und ruft für diese Instanz `RegisterForAllObjects`. Dabei übergibt es zwei Funktionsobjekte, *Functor1* und *Functor2*. Anschließend wird die Programmausführung gestartet.

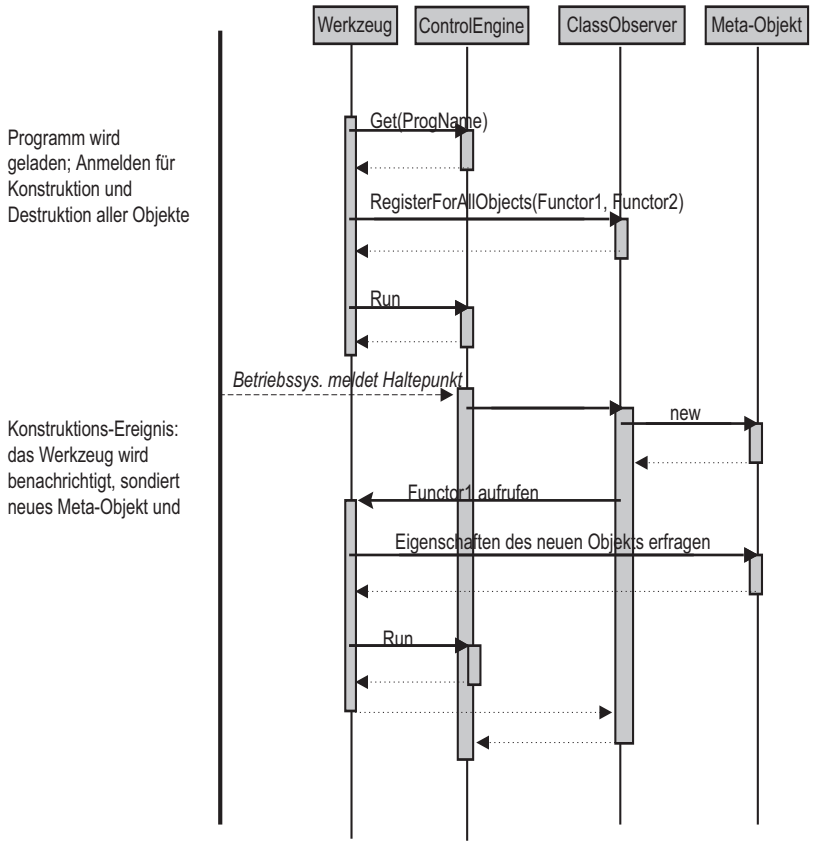


Abb. 11: Benutzung der ControlEngine

Durch das `RegisterForAllObjects` wurden von der ControlEngine Haltepunkte auf alle Konstruktoren und Destruktoren gesetzt. Sobald das erste Objekt erzeugt, d.h. der erste Konstruktor erreicht wird, erfolgt eine Unterbrechung, und die ControlEngine wird aktiviert. Sie delegiert die Behandlung an die `tClassObserver`-Instanz, welche auf das Konstruktionsereignis reagiert, indem sie ein neues Objekt im Programmodell anlegt und das als Beobachter angemeldete Werkzeug benachrichtigt. Die Benachrichtigung erfolgt duch das Aktivieren des Funktionsobjekts Functor1. Functor1 ist konzeptueller Bestandteil des Werkzeugs und übernimmt die Bearbeitung des Ereignisses: Es sondiert die Eigenschaften des neu erzeugten Programmodell-Objekts, das es als Parameter mit der Aktivierung erhalten hat, und sorgt dafür, daß die Veränderung des Programmodells auf dem Bildschirm erscheint. Wenn das geschehen ist, sendet es an die ControlEngine den Befehl, die Programmausführung fortzusetzen. Das Interaktionsdiagramm in Abbildung 11 veranschaulicht dieses Szenario.

6.4 Systementwurf als Mehrprozeßmodell

In Software-Entwicklungsprojekten kann es vorkommen, daß Entwicklungs- und Zielplattformen voneinander abweichen: Softwareentwickler benutzen für die Entwicklung andere Betriebssysteme und Systemumgebungen als diejenigen, auf denen die entwickelte Software später ablaufen wird.

Ein professionelles System zur Softwarevisualisierung sollte diesen Umstand berücksichtigen und daher zulassen, daß die Informationserfassung und die Informationspräsentation auf verschiedenen Plattformen stattfinden. Der für die Informationserfassung zuständige Automat muß auf derselben Plattform (der Zielplattform) ablaufen wie das untersuchte Programm, um die Programmausführung kontrollieren zu können. Die Visualisierungswerkzeuge sollen hingegen auf der Entwicklungsplattform zur Verfügung stehen. Für unseren Systementwurf bedeutet das, daß die ControlEngine und die Visualisierungswerkzeuge als eigenständige Programme ausgeführt werden können.

Da die ControlEngine damit notwendigerweise als gesondertes Programm realisiert wird, bietet es sich an, die einzelnen Visualsierungswerkzeuge nicht in einem Programm zu bündeln, sondern sie ebenfalls voneinander zu trennen. Durch dieTrennung wird motiviert, die Systemkomponenten als Werkzeuge (und nicht als Subwerkzeuge) zu verstehen.

Aus technischer Sicht bedeutet der Ansatz, daß das Visualisierungssystem einen Mehrprozeßraum bildet. Der Automat und jedes einzelne Werkzeug sind unterschiedliche Prozesse, die voneinander getrennt in eigenen Adreßräumen laufen, unter Umständen sogar auf verschiedenen Geräten und unter verschiedenen Betriebssystemen.

Natürlich müssen die Komponenten des Systems miteinander kommunizieren. Normale Funktions- bzw. Methodenaufrufe, die im Ein-Prozeß-Modell ausreichend sind, versagen hier, weil sie nicht über Prozeßgrenzen hinweg erfolgen können. Sie müssen ersetzt werden durch Formen der Inter-Prozeß-Kommunikation (*inter process communication*, IPC), z.B. durch standardisierte Remote-Procedure-Calls (RPC). Wir können davon ausgehen, daß eine Form der IPC verwendet wird, die von den Orten der Prozesse abstrahiert. Man muß also nur wissen, *daß* die Komponenten in unterschiedlichen Prozessen ablaufen; es ist aber unerheblich, ob die Prozesse auf demselben Computer oder unter demselben Betriebssystem gestartet wurden und ob die Kommunikation direkt oder über ein Netzwerk erfolgt.

Der Zeitbedarf, der für die Inter-Prozeß-Kommunikation anfällt, übersteigt immer den Bedarf, der durch lokale Kommunikation innerhalb eines Adreßraums entsteht. Auch wenn wir annehmen, daß die Mehrkosten in einem vertretbaren Rahmen bleiben, sollte der Einsatz von IPC von vornherein klein gehalten werden.

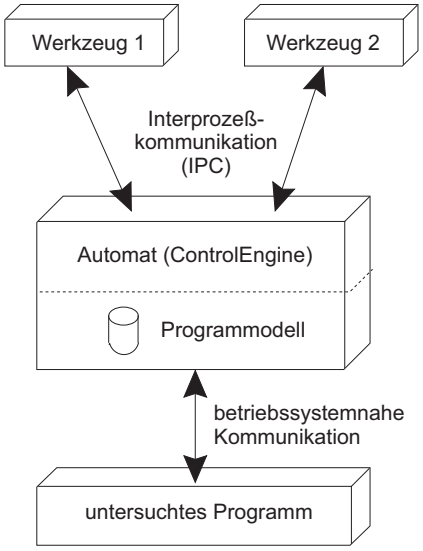


Abb. 12: Mehrprozeßmodell
(jeder Block bedeutet einen Prozeß)

Im vorliegenden Entwurf wird der wesentliche Anteil an Kommunikation um das Programmodell herum stattfinden. Nachdem Automat und Werkzeuge schon auf getrennte Prozesse verteilt wurden, ist das Schicksal des Programmodells noch offen. Es bestehen drei Möglichkeiten, um das Programmodell in den Mehrprozeßraum einzuordnen:

- Das Programmodell wird ein eigenständiger Prozeß.
- Das Programmodell wird einem der Werkzeug-Prozesse zugeordnet.
- Das Programmodell wird im Prozeß des Automaten implementiert.

Die erste Lösung kann ausgeschieden werden: Sie verspricht keine Vorteile, maximiert aber den Aufwand an IPC, weil das Programmodell sowohl mit Automat als auch mit Werkzeugen über Prozeßgrenzen hinweg kommunizieren muß.

Von den verbleibenden Möglichkeiten ist die letzte vorteilhafter. Der Automat muß enger mit dem Programmodell zusammenarbeiten als die Werkzeuge, weil er erstens mehr Informationen erfaßt als angezeigt werden. Zweitens muß der Automat, um Programmereignisse richtig zu interpretieren, bei jedem Ereignis mehrere Rückfragen an Programm und Programmodell richten, während ein solcher Interpretationsbedarf bei den Werkzeugen entfällt und auf deren Seite häufig schon eine gezielte Anfrage an das Programmodell ausreicht, um auf ein Ereignis zu reagieren. Das Prozeßdiagramm in Abbildung 12 zeigt die so entstehenden Prozesse.

Was genau wird nun in der Inter-Prozeß-Kommunikation zwischen Automaten- und Werkzeugprozessen übertragen? Dieser Kommunikationsweg wird von Werkzeugen benutzt, um den Automaten zu steuern, Ereignisse von ihm entgegenzunehmen und um Anfragen an das Programmodell zu stellen. Hinsichtlich der zu übertragenden Daten bedeutet das:

- Um einem Werkzeug ein Ereignis mitzuteilen, ruft der Automat per IPC eine bestimmte Funktion im Werkzeug-Prozeß auf. Als Parameter werden Daten übermittelt, die das Ereignis genauer beschreiben. Ein Funktionswert wird nicht zurückgegeben.
- Für die Steuerung des Automaten ruft ein Werkzeug Funktionen im Automatenprozeß auf. Je nach Funktionen werden Parameter übergeben. Eventuell werden Funktionsergebnisse zurückgegeben, die aus einfachen Datentypen bestehen, etwa Boole'sche Werte, die angeben, ob der Automat ein Kommando korrekt entgegennehmen konnte.
- Anfragen an das Programmodell erfolgen als IPC-Funktionsaufrufe, enthalten in der Regel Parameter, die die Anfrage spezifizieren, und liefern als Rückgabewert Elemente des Modells, z.B. Informationen über Klassen, Objekte und Methoden des beobachteten Programms.

Die ersten beiden Fälle stellen relativ einfache, parametrisierte Funktionsaufrufe dar und sind hinsichtlich der IPC nicht besonders kritisch. Der dritte Fall hingegen wirft ein Problem auf, das es im Einprozeßmodell nicht gibt: Die Anfragen an das Programmodell werden mit der Übermittlung von Metamodell-Instanzen beantwortet. Die Elemente sind komplexe Objekte mit diversen Attributen, darunter auch Verweise auf andere Objekte. Im Einprozeßmodell werden diese Anfragen einfach Zeiger auf den Speicherplatz des Objektes zurückgeben, denn das Werkzeug befindet sich im selben Adreßraum wie das Programmodell und kennt dann mit dem Zeiger auch das zugehörige Objekt.

Im Mehrprozeßmodell kann das Werkzeug mit einem Zeiger, den es vom Programmodell bekommt, zunächst einmal nichts anfangen; das zugehörige Objekt ist im Programmodell, aber nicht im Werkzeug lokalisiert. Damit das Werkzeug auf das Objekt zugreifen kann, sind nun zusätzliche IPC-Aufrufe nötig. Dabei gibt es zwei mögliche Vorgehensweisen: Entweder erhält das Werkzeug eine komplette Kopie des Objekts, auf die es fortan lokal zugreifen kann, oder es führt bei jedem Zugriff auf das Objekt einen erneuten IPC-Aufruf aus. Der letztgenannte Ansatz ist wegen der unnötig hohen Zahl an IPC-Operationen ungünstig, aber auch die Übertragung von Objekt-Kopien kann nicht überzeugen:

- Die Objekte können sehr groß werden. Elemente des Programmodells sind z.B. Metaobjekte, die die „eigentlichen“ Objekte im untersuchten Programm repräsentieren. So wie die Original-Objekte über beliebig viele Attribute verfügen und damit sehr groß werden können, wachsen auch die Metaobjekte entsprechend an.
- Nicht jedes Werkzeug benötigt alle Eigenschaften, die in den Elementen des Programmodells bekannt sind. So kann es sein, daß ein Werkzeug nur das Vorhandensein eines Objekts anzeigt und dazu nicht mehr als die Existenz und die Bezeichnung des Metaobjekts kennen muß, während ein anderes Werkzeug alle Referenzen eines Objekts auf andere Objekte darstellen soll und dazu die einzelnen Attribute des Metaobjekts benötigt.
- Wenn die Werkzeuge Kopien von Programmodell-Elementen erhalten, entsteht das Problem, Programmodell und Kopien konsistent halten zu müssen (was im übrigen zusätzliche IPC-Operationen erfordert).

Damit ist das Übermitteln von Objekt-Kopien keine zufriedenstellende Lösung.

Eine geschicktere Lösung besteht darin, daß in den Werkzeugen sogenannte *Proxy*-Objekte angelegt werden, die als lokale Stellvertreter der eigentlichen Programmodell-Elemente agieren. Wenn ein Werkzeug nun eine Anfrage an das Programmodell verschickt, erhält es als Ergebnis ein Proxy. Alle weiteren Aufrufe, die das Objekt betreffen, setzt das Werkzeug dann an das Proxy ab, welches die Aufrufe entweder selbst bearbeiten kann oder sich per IPC an das Programmodell wendet. Wenn ein Proxy etwa eine Klasse des beobachteten Programms beschreibt und vom Werkzeug erstmals nach dem Namen der Klasse gefragt wird, läßt sich das Proxy den Namen vom Programmodell schicken und speichert ihn, so daß spätere Anfragen lokal bearbeitet werden können. Sowohl das Programmodell als auch die Werkzeuge verfügen jeweils über eine Schnittstelle, die die Interprozeßkommunikation hinter den Proxies verbergen. Das beschriebene Vorgehen ist als Entwurfsmuster unter den Namen „Remote-Proxy“ [Gamma et al. 95] und „Ambassador“ [Coplien 92] bekannt.

Es wurde bereits festgestellt, daß nicht jedes Werkzeug dieselben Eigenschaften eines Programmodell-Elements kennen muß. Um die IPC-Kommunikation zu minimieren, müßten in jedem Werkzeug die Proxies gleich so angelegt werden, daß sie genau die vom Werkzeug benö-

tigten Informationen enthalten. Dafür sind also werkzeugspezifische Proxy-Schnittstellen zwischen Programmodell und Automat nötig.

Solche werkzeugspezifischen Schnittstellen bringen ein weiteres Potential mit sich, indem in ihnen neue Elementtypen des Metamodells definiert werden können. Das Programmodell selbst kann unverändert bleiben, wenn sich die neuen Typen intern auf bereits vorhandene Modellelemente abbilden lassen. Im weiteren Verlauf dieser Arbeit (vgl. Abschnitt 7.7) wird z.B. erörtert werden, wie sich Entwurfsmuster als Elemente 2. Ordnung in das Metamodell integrieren lassen. Auf der Werkzeugseite stellen sich die zusätzlichen Modellelemente genauso wie die Basiselemente durch Proxies dar; auf der Programmodell-Seite ist die werkzeugspezifische Schnittstelle dafür zuständig, die Abbildung auf die Basiselemente des Programmodells vorzunehmen.

Als technischen Clou kann man die Proxy-Schnittstellen als dynamische Bibliotheken realisieren, die erst zur Laufzeit an das Metamodell gebunden werden; in der Sprache der Betriebssysteme werden dazu „Dynamic Link Libraries“ (DLL) oder „Shared Objectfiles“ (SO) eingesetzt. Das Programm, das den Automatenprozeß und das Programmodell enthält, muß dann weder neu übersetzt noch neu gelinkt werden, aber trotzdem werden die spezifischen Schnittstellen im selben Prozeßraum ausgeführt und können dadurch effizient mit dem Programmodell kommunizieren. Abbildung 13 faßt dieses Konzept zusammen.

Ausgangspunkt dieses Abschnitts war die Forderung, Entwicklungs- und Zielplattform voneinander trennen zu können. Das beschriebene Mehrprozeßmodell erlaubt diese Trennung und benutzt dabei effiziente Kommunikationsmechanismen zwischen den Prozessen. Die Konzepte, die in dieser Arbeit ausgearbeitet werden, gelten gleichermaßen für das Ein- und das Mehrprozeßmodell.

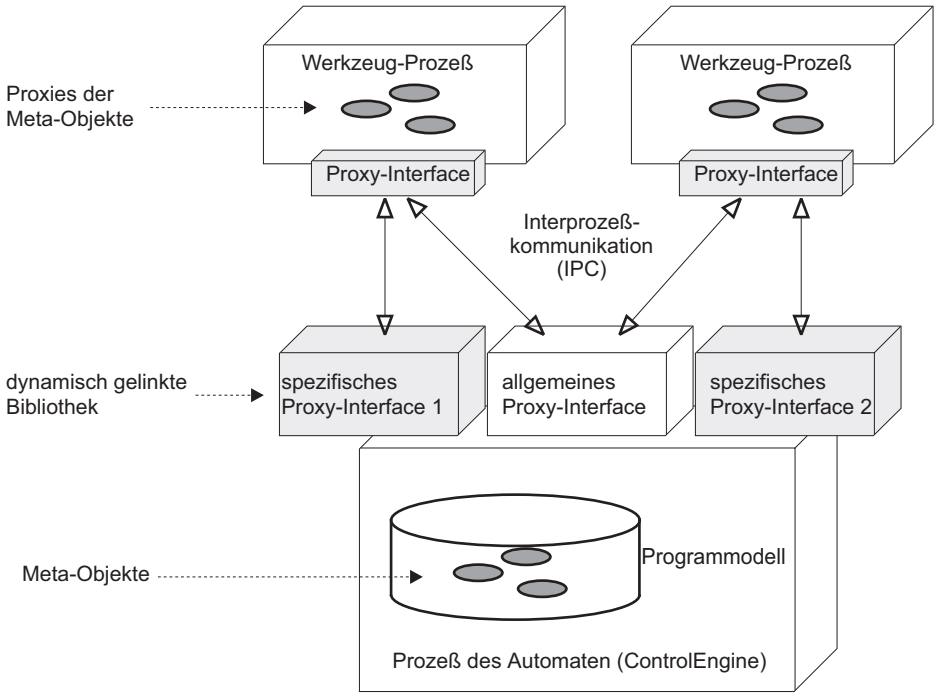


Abb. 13: Mehrprozeßmodell, erweitert durch dynamisch gelinkte Bibliotheken

6.5 Zusammenfassung

Zu Beginn des Kapitels wurde erklärt, wie die Informationen, die das Programmodell enthalten soll, erfaßt werden können. Die Vorgehensweise entspricht der eines Debuggers; das untersuchte Programm muß mit Debug-Informationen compiliert werden. Das Visualisierungssystem enthält einen Automaten mit der Bezeichnung *ControlEngine*, der für das Auswerten der Debug-Informationen zuständig ist. Dieser Automat schreibt während der Programmausführung das Programmodell fort und benachrichtigt Werkzeuge über dessen Veränderungen. Die Werkzeuge müssen sich zuvor für die unterschiedlichen Arten von Ereignissen angemeldet haben. Für die Anmeldung stehen bisher zwei Schnittstellen bereit, die den Werkzeugen erlauben, Objektkonstruktionen und -destruktionen zu beobachten (`tClassObserver`) und sich über Methodenaufrufe informieren zu lassen (`tFunctionObserver`). Weitere Schnittstellen können später hinzukommen.

Beim jetzigen Stand der Implementierung befinden sich Automat, Programmodell und Werkzeuge im selben Prozeßraum. Der Entwurf ist aber so ausgelegt, daß Automat und Werkzeuge in verschiedene Prozesse aufgeteilt werden können. Diese Alternative wurde im letzten Abschnitt des Kapitels diskutiert.

Kapitel 7
Das Werkzeug IDTool

In den vorangegangenen Kapiteln über ControlEngine und Metamodell wurden die Grundlagen der Informationserfassung und -verwaltung von VOOP2 beschrieben. Auf dieser Basis arbeiten die Werkzeuge zur Präsentation. Im Zusammenhang mit dieser Arbeit habe ich den Prototypen eines Werkzeugs für Interaktionsdiagramme erstellt. Der Name des Prototyps ist *IDTool* (für Interaction Diagram Tool). Zu Beginn dieses Kapitels werde ich die grundlegenden Anforderungen an den Prototypen behandeln. In Abschnitt 7.2 wird eine Notation für die Interaktionsdiagramme beschrieben, die das IDTool erzeugen wird. Die Gestaltung der Benutzerschnittstelle ist Inhalt des Abschnitts 7.4. Anschließend beschreibe ich den Entwurf und die Implementierung des Werkzeugs; zu diesem Themenbereich gehören die Abschnitte 7.5 und 7.6. Im Anschluß daran wird diskutiert, wie Elemente 2. Ordnung im Werkzeug berücksichtigt werden können.

7.1 Anforderungen an das IDTool

Der Prototyp *IDTool* ist das erste Werkzeug, das im Rahmen des Konzepts von VOOP2 entwikkelt worden ist. Seine Aufgabe ist es, die Objekte des Programmodells und die zugehörigen Methodenaufrufe als Interaktionsdiagramm zu präsentieren und Änderungen des Programmodells zu ermöglichen. Zusätzliche Anforderungen sind, daß der Benutzer den Umfang des Diagramms durch Filter einschränken und die erzeugten Diagramme bearbeiten kann.

Die Erstellung aussagefähiger Visualisierungen soll dadurch erleichtert werden, daß den Benutzern eine nachträgliche Änderung von erzeugten Diagrammen möglich ist. Grundsätzlich kann eine solche Bearbeitung entweder innerhalb des IDTool oder aber in einem anderen Werkzeug (u.U. einem existierenden Grafikeditor) erfolgen. Wenn man die Ergebnisse zur Dokumentation von Frameworks, Entwurfsmustern o.ä. verwenden will, benötigt man noch keine Bearbeitungsmöglichkeiten innerhalb des IDTool; der Benutzer könnte Diagramme mit dem IDTool erzeugen, sie dann mit einem vorhandenen Grafik-Werkzeug editieren und in eine Dokumentation integrieren. Trotzdem bietet das IDTool eine eigene Editor-Funktionalität: Dadurch kann der Benutzer schon während der Visualisierung Veränderungen am Diagramm vornehmen, die für den weiteren Programmablauf wirksam sind und noch während der Untersuchung eines Programms zu einer verständlicheren Anzeige führen. Was aber kann das Bearbeiten bedeuten? Die darunter fallenden Tätigkeiten kann man mit drei Punkten charakterisieren:

- Das Diagramm wird um zusätzliche Elemente erweitert. Der Benutzer fügt z.B. Kommentare ein, um die Darstellung zu erläutern, oder ergänzt das Diagramm um zusätzliche Objekte.
- Elemente werden aus dem Diagramm entfernt. Da die automatisch erstellten Diagramme in der Regel viele Elemente enthalten, die für den Benutzer uninteressant sind, entfernt er z.B. einzelne Objekte, um das Diagramm übersichtlicher zu machen.
- Elemente des Diagramms werden durch neu eingefügte Elemente ersetzt. Damit lassen sich die in den Kapitel 4 eingeführten Informationen 2. Ordnung einbeziehen. Beispielsweise könnte der Benutzer eine Anzahl vorhandener Objekte durch ein Darstellungselement, das ein Subsystem repräsentert, ersetzen, oder statt einer Folge von Methodenaufrufen ein Entwurfsmuster in das Diagramm einfügen.

Insbesondere der letzte Punkt macht deutlich: Wenn vom Benutzer Änderungen am Diagramm vorgenommen werden, bedarf es also auch einer Anpassung des zugrunde liegenden fachlichen Modells, das Konzepte wie „Subsystem“ und „Entwurfsmuster“ berücksichtigen soll.

7.2 Interaktionsdiagramme

Interaktionsdiagramme sind in der Softwareentwicklung mittlerweile in diversen Varianten verwendet worden. Zu Beginn der Entwicklung des IDTool habe ich die unterschiedlichen Varianten verglichen und daraus eine eigene Notation zusammengetragen. Während bei einer Skizze auf dem Papier situativ neue Darstellungsmittel eingeführt, Kommentare notiert und Umdeutungen vorgenommen werden können, ist der Benutzer einer Grafiksoftware dauerhaft auf die Ausdrucksmittel festgelegt, die die Softwareentwickler vorgesehen haben. Um mit dem IDTool aussagekräftige Diagramme erstellen zu können, sollte also sichergestellt sein, daß der Vorrat an Gestaltungsmitteln zukünftige Anforderungen möglichst weit abdeckt. Dazu werde ich in diesem Abschnitt eine Reihe von grafischen Elementen vorstellen, die ich als Bestandteile einer erweiterten Form von Interaktionsdiagrammen einführe.

7.2.1 Varianten von Interaktionsdiagrammen

Die Verwendung von Interaktionsdiagrammen in der objektorientierten Softwareentwicklung geht auf Rumbaugh [Rumbaugh et al. 91] und Jacobson [Jacobson et al. 92] zurück. In Jacobsons Methode zeigt sich dieses Modellierungsmittel als eine schlüssige Formalisierung der zuvor entwickleten Szenarios (use cases). Jacobson führt die Diagramme als *interaction diagrams* ein. Bezeichnungen, die von anderen Autoren benutzt werden, sind *event trace diagrams* (Rumbaugh), *fence-style object interaction diagrams* [Yourdon et al. 95], *timeline diagrams* [Coleman et al. 94] und *sequence diagrams* [Rational 97].

Allen Varianten von Interaktionsdiagrammen ist gemeinsam, daß Objekte als vertikale Linien und Nachrichten als horizontale Pfeile wiedergegeben werden. Eine implizite Zeitachse verläuft von oben nach unten, so daß die Anordnung der Nachrichten deren chronologischer Reihenfolge entspricht. Bei den dargestellten Objekten handelt es sich nicht unbedingt um programmiersprachliche Objekte (d.h. Instanzen von Programmklassen); einige Autoren benutzen die Diagramme auch, um die Anwendungswelt (oder Interaktionen von Anwendungswelt und modelliertem System) zu beschreiben. In diesem Fall können Personen, Institutionen usw.

als Objekte auftreten, und die Pfeile versinnbildlichen Kommunikationsabläufe, Tätigkeiten, Operationen etc.

Das in Abbildung 15 gezeigte Beispiel verkörpert die einfachste Variante von Interaktionsdiagrammen und entspricht etwa der Notation von Rumbaugh und Jacobson. Erweiterte Notationen zeigen explizit den Kontrollfluß, bilden Rückgabewerte ab, berücksichtigen Unterschiede zwischen synchroner und asynchroner Kommunikation und den Zeitbedarf von Operationen [Rational 97]. Die im folgenden besprochene Notation entspricht in weiten Teilen den Interaktionsdiagrammen in [Lilienthal und Strunk 96]. Abbildung 14 zeigt dasselbe Beispiel wie Abbildung 15 in der erweiterten Notation.

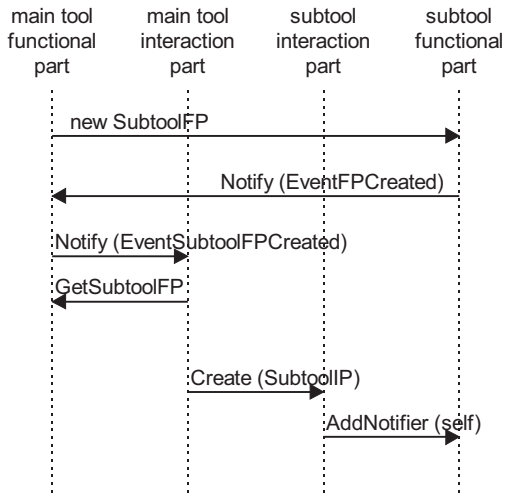


Abb. 15: Einfaches Interaktionsdiagramm

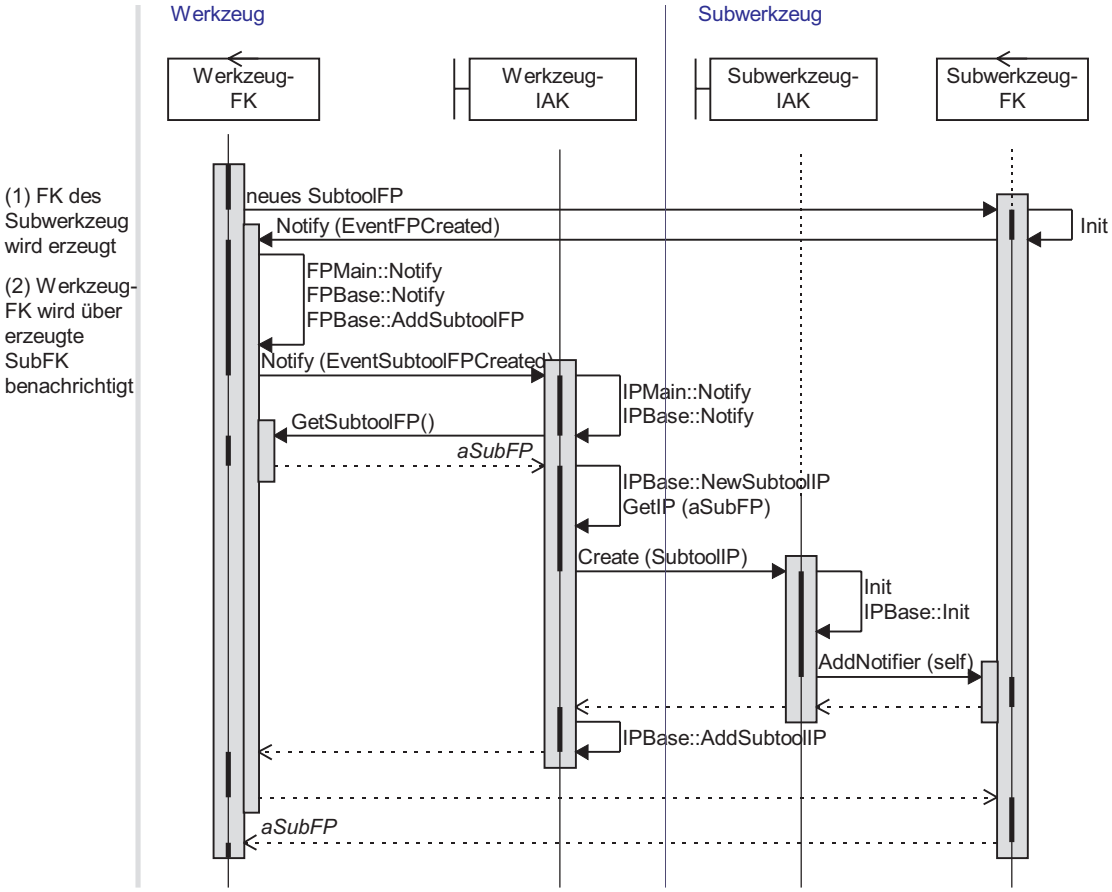


Abb. 14: Erweitertes Interaktionsdiagramm

7.2.2 Notation für erweiterte Interaktionsdiagramme

7.2.2.1 Aufbau des Diagramms

Auf der linken Seite des Interaktionsdiagramms befindet sich eine Spalte, die für textuelle Beschreibungen der Operationen bestimmt ist. Rechts daneben wird die Systemabgrenzung eingetragen (Abbildung 16). Die Systemabgrenzung kann Quelle und Ziel von Ereignissen darstellen, die von einem nicht modellierten Element des Systems eintreffen bzw. an ein solches Element gerichtet sind.

Daran schließen sich die Objekte an. Für jedes Objekt gibt es eine Objektlinie und ein Objektsymbol. Oberhalb der Objektsymbole können für einzelne Objekte oder für Gruppen von Objekten Beschreibungen eingetragen werden.

Wenn Objekte, die nicht zum eigentlichen System zählen, in das Diagramm mit aufgenommen werden sollen, können diese auf der linken Seite der Systemabgrenzung erscheinen (Abbildung 17).

7.2.2.2 Objektsymbole

Jede Objektlinie wird durch einen Objektnamen identifiziert. Bei automatischer Erzeugung des Diagramms kann als Name derjenige Variablenname gewählt werden, unter dem das Objekt zum Zeitpunkt seiner Erzeugung bekannt ist. Eine solche (benannte) Variable muß es nicht zwingend geben oder muß nicht eindeutig sein; aus diesem Grund wird man unter Umständen auf künstliche Bezeichungen ausweichen, z.B. auf eine Kombination aus Klassennamen und einer fortlaufenden Nummer.

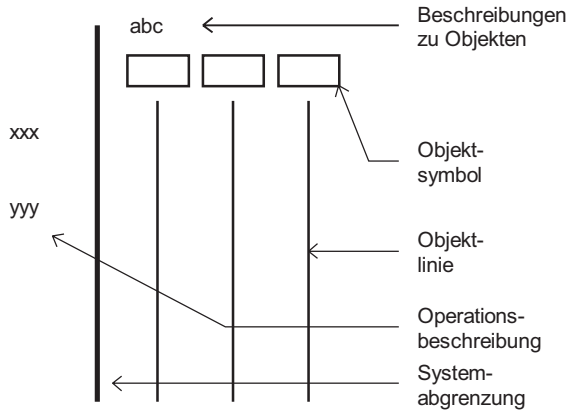


Abb. 16: Schematischer Aufbau des Diagramms

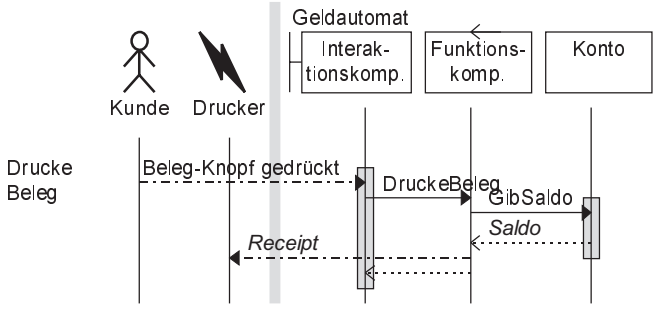


Abb. 17: Objekte innerhalb und außerhalb des modellierten Systems

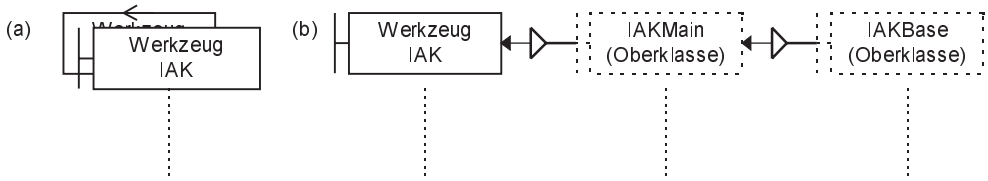


Abb. 18: (a) Gruppieren von Objekten, (b) explizite Darstellung von Oberklassen

Es bietet sich an, den Namen mit einem grafischen Symbol zu verbinden, das zusätzliche Eigenschaften des Objekts ausdrücken kann. Möglichkeiten für solche Symbole findet man in den Klassen- und Objektdiagrammen der verschiedenen OO-Methoden. In den abgebildeten Beispielen wird auf den Zeichenvorrat von *Mainstream Objects* [Yourdon et al. 95] zurückgegriffen. Angelehnt an das Model-View-Controller-Entwurfsmuster [Goldberg 84] werden hier verschiedene Symbole für Entity-, Interaktions- und Funktionsobjekte eingesetzt. Auch für Personen, Maschinenschnittstellen usw. können eigene Symbole benutzt werden (vgl. Abbildung 17).

Wenn mehrere Objekte zu einem Element zweiter Ordnung (wie Subsystem, Prozeß etc.) zusammengefaßt werden, können entweder ein neues Symbol und ein neuer Name vergeben werden, oder die Objektsymbole werden versetzt hintereinander (eventuell auch übereinander) angeordnet (Abbildung 18a).

Statt mehrere Objekte zusammenzufassen, kann man auch umgekehrt ein Objekt in seine Oberklassen-Anteile zerlegen. Für jede Oberklasse würde dann eine eigene Objektlinie im Diagramm auftreten, so daß einfach zu erkennen ist, in welchen Klassen die Methoden, die für ein Objekt aufgerufen werden, definiert worden sind. Um deutlich zu machen, daß sich die Objektlinien auf dasselbe (Programm-) Objekt beziehen, sollte dies durch eine Verbindung der Objektsymbole verdeutlicht werden (Abbildung 18b).

7.2.2.3 Objektlinien

Zu jedem Objekt gibt es eine vertikale Linie, die sich durch das gesamte Interaktionsdiagramm zieht. Auf dieser Linie sind verschiedene Objektzustände kenntlich zu machen, nämlich

- die Phasen außerhalb der Existenz des Objektes, d.h. vor der Konstruktion und nach der Destruktion,
- aktive Phasen, in denen eine Methode des Objektes ausgeführt wird (die Kontrolle kann aber zwischenzeitlich an andere Objekte delegiert werden),
- Phasen, in denen der Kontrollfluß des Programms in einer Methode des Objektes ist.

Abbildung 19a faßt die verschiedenen Phasen zusammen und gibt entsprechende Notationen an.

7.2.2.4 Systemabgrenzung

Die Systemabgrenzung kann formal als eine spezielle Objektlinie angesehen werden, die links von allen Objekten des Systems gezeichnet wird und alle Objekte zusammenfaßt, die nicht zum eigentlichen System zählen.

Optional können auf der linken Seite der Systemabgrenzung Aktoren und Objekte aus der Systemumgebung dargestellt werden (Abbildung 17).

7.2.2.5 Nachrichten

Zwei grundlegende Arten von Nachrichten werden unterschieden:

- Methodenaufrufe – erfolgen synchron und haben eine Delegation des Steuerflusses zur Folge. Es gibt *immer* eine korrespondierende, synchrone Rückkehr.
- Ereignisse – erfolgen asynchron und haben keine direkte Auswirkung auf den Steuerfluß.

Für Methodenaufrufe, Rückkehr und Ereignisse werden unterschiedliche Pfeile verwendet (Abbildung 19b).Die Pfeile haben immer Objekte oder die Systemabgrenzung als Anfangs- und Endpunkt. Oberhalb des Pfeiles wird bei Methodenaufrufen der Name der aufgerufenen Operation angegeben und optional die übergebenen Parameter. Wenn eine Methode einen Wert als Ergebnis zurückgibt, kann zu dem Pfeil, der die Rückkehr symbolisiert, der Typ des Ergebnisses oder der Variablenname, unter dem das aufrufende Objekt das Ergebnis verwendet, eingetragen werden.

Durch einen gewinkelten Pfeil werden Nachrichten verkörpert, die Objekte an sich selbst senden. Wahlweise steht eine Notation aus [Lilienthal und Strunk 96] zur Verfügung, die solche Nachrichten in einem Kasten untereinander auflistet (Abbildung 19c).

Wenn Nachrichten Elemente 2. Ordnung darstellen (z.B. Entwurfsmuster), sollte dies durch grafische Attribute kenntlich gemacht werden.

7.2.2.6 Beschreibung von Nachrichten

Jede Nachricht kann mit einem erklärenden Text versehen werden, der im Diagramm als Operationsbeschreibung auf derselben Höhe der Nachricht am linken Rand erscheint (vgl. Abbildung 17).

7.2.2.7 Horizontale und vertikale Trennlinien

Zwischen zwei Objekten kann eine vertikale Trennlinie eingefügt werden, um z.B. verschiedene Systemkomponenten zu unterscheiden. Horizontale Trennlinien können zwischen Nachrich-

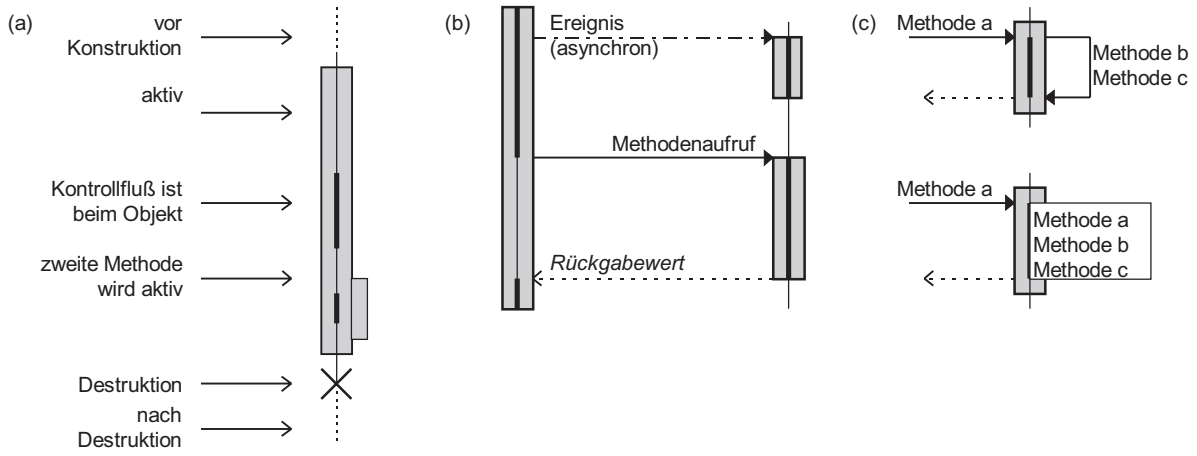


Abb. 19: (a) Phasen einer Objektlinie, (b) Nachrichten, (c) Aufruf eigener Methoden

ten verwendet werden und grenzen beispielsweise Transaktionssequenzen oder Tätigkeiten voneinander ab.

7.2.2.8 Anmerkungen und Verweise

Im Diagramm können an beliebigen Positionen Anmerkungen auftreten, die sich auf eine Nachricht oder auf ein Objekt beziehen (Abbildung 20a). In einem Diagramm kann auf andere Diagramme verwiesen werden, in denen Operationen oder Objekte verfeinert oder zusätzliche Nachrichtenabfolgen gezeigt werden (Abbildung 20b).

7.2.2.9 Zusammenfassung

Die Auflistung der Diagrammelemente ist wichtig, weil sie die Grundlage für ein fachliches Modell von Interaktionsdiagrammen bildet. Die grafische Gestaltung der vorgestellten Elemente ist an dieser Stelle zweitrangig; eventuell sollte es in einem Diagrammeditor dem Benutzer überlassen werden, sich zwischen alternativen Symbolen zu entscheiden und auf bestimmte Ausdrucksmittel ganz zu verzichten. So sollte es auch möglich sein, das in Abbildung 14 gezeigte Beispiel optional in der einfachen, in Abbildung 15 verwendeten Form darzustellen.

7.3 Gestaltung des IDTool

Das IDTool wurde zu Beginn dieses Kapitels als ein Werkzeug zur (automatischen) Erstellung und Bearbeitung von Interaktionsdiagrammen beschrieben. Die Softwarevisualisierung wird durch die Einbindung in VOOP2 möglich.

Das grafische Vokabular des Werkzeugs sind die Elemente erweiterter Interaktionsdiagramme, die oben beschrieben wurden. Wie sich die Diagramme bzw. die Elemente bearbeiten lassen, kann grundsätzlich auf zwei Arten realisiert werden:

- Editieren durch Deklaration bedeutet, daß – unabhängig vom dargestellten Diagramm – Kriterien formuliert werden, nach denen bestehende Diagramme verändert werden. So könnte ein Benutzer in einem Dialog angeben, daß alle Objekte der Klasse *xyz* aus dem Diagramm entfernt werden sollen. Dieses Vorgehen kann man als Filterung a posteriori bezeichnen.
- Editieren durch direkte Manipulation heißt, daß der Benutzer im angezeigten Diagramm einzelne Elemente selektieren und manipulieren kann. In der von grafischen Editoren bekannten Weise können dann einzelne Diagrammelemente gelöscht, verschoben etc. werden.

Diese beiden Formen des Editierens sind komplementär, d.h., sie können beide in einem Werkzeug realisiert werden. Gerade aber die direkte Manipulation erlaubt es, Diagramme an die jeweiligen Bedürfnisse anzupassen und, besonders auch für Dritte, einzelne Aspekte sichtbar zu machen. Das kann geschehen, indem Teile des Diagramms, die nicht relevant sind, verborgen oder entfernt werden, indem durch textuelle Anmerkungen oder grafische Annotationen zusätzliche Information festgehalten oder der Abstraktionsgrad verändert wird.

7.3.1 IDTool als Diagrammeditor

Das IDTool erlaubt die Bearbeitung durch direkte Manipulation. Da das Werkzeug damit über gute Möglichkeiten zur interaktiven Bearbeitung von Diagrammen verfügt, gibt es keinen

Grund, die Bearbeitung auf Diagramme zu beschränken, die zuvor als Visualisierung generiert worden sind; vielmehr kann das IDTool dann genauso gut zum Zeichnen von Interaktionsdiagrammen benutzt werden, die *ausschließlich* manuell erstellt werden. In diesem Sinne fällt bei der Entwicklung des Visualisierungswerkzeugs ganz nebenbei ein Editierwerkzeug ab. Die Möglichkeit, ein Visualisierungswerkzeug somit auch als Editor zu nutzen, findet man bei vergleichbaren Arbeiten (siehe Abschnitt 2.3) nicht.

Die erste Version des IDTool-Prototypen stellt nur eine Teilmenge der beschriebenen Interaktionsdiagramm-Elemente zur Verfügung:

- Objekte (als Objektlinien mit einem einheitlichen, rechteckigen Objektsysmbol),
- Methodenaufrufe und -rücksprünge,
- Aufrufe von Methoden desselben Objekts als gewinkelte Pfeile,
- aktive Phasen,
- die Systemabgrenzung.

Diese Elemente können zum Beispiel selektiert, verschoben, gelöscht oder mit einer Beschriftung versehen werden. Die Manipulationen sind z.T. nur in bestimmter Art möglich; so lassen sich Objekte etwa nur horizontal, nicht aber vertikal verschieben. Solche Beschränkungen wurden bewußt eingeführt, um die optische Grundstruktur des Diagramms zu erhalten und um „unsinnige“ Darstellungen (z.B. Methodenaufrufe, die nicht mit Objekten verbunden sind) zu verhindern.

Selbstverständlich werden die Beziehungen zwischen den Diagrammelementen erhalten, wenn der Benutzer einzelne Elemente manipuliert, d.h. beispielsweise, daß beim Verschieben eines Objekts auch alle mit dem Objekt verbundenen Methodenaufrufe verschoben werden.

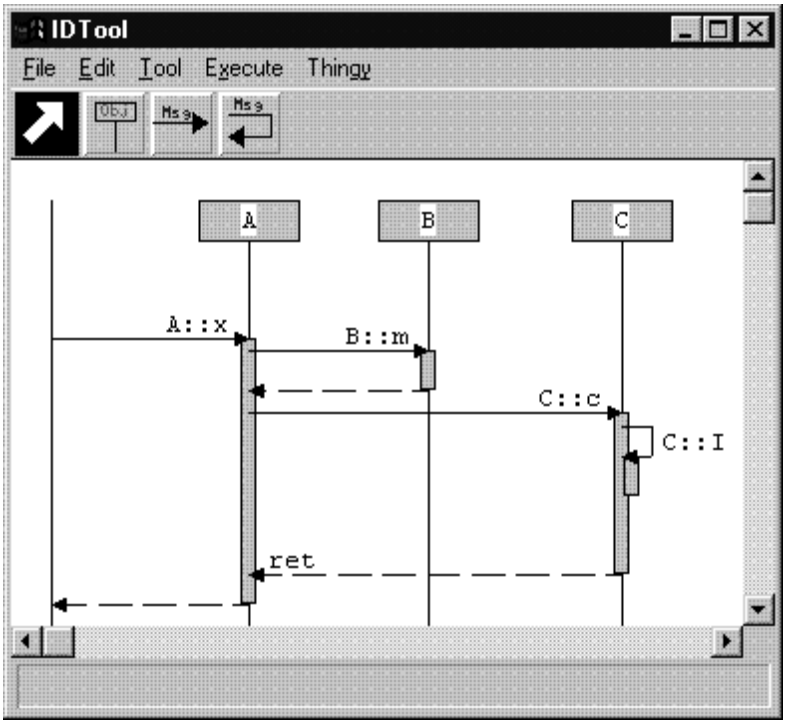


Abb. 21: Das IDTool

Neben den Möglichkeiten der direkten Manipulation bietet das IDTool in seiner bestehenden Implementierung die wesentlichen Fähigkeiten, die man von einem Editor erwartet:

- Diagramme können gespeichert und wieder geladen werden.
- Das Drucken von Diagrammen ist möglich.
- Es gibt eine Undo- und Redo-Funktion, um beliebig viele Aktionen rückgängig zu machen oder wiederherzustellen.
- Die Anzeige kann beliebig skaliert ("gezoomt") werden.
- Diagramme können in andere Programme exportiert werden.[1]

Ein Screenshot des IDTool zeigt Abbildung 21.

7.3.2 IDTool zur Softwarevisualisierung

Wenn das IDTool nicht nur zur Diagrammbearbeitung eingesetzt wird, sondern auch zu seinem eigentlichen Zweck, der Generierung von Diagrammen, müssen Steuerungsmöglichkeiten für den Ablauf des zu visualisierenden Programms und für die Auswahl der zu erfassenden Daten bereitgestellt werden. Genauer gesagt werden Steuerungsmöglichkeiten für die ControlEngine benötigt, um (1) ein Programm zu laden, zu starten, zu unterbrechen und fortzusetzen, und um (2) zu definieren, welche Ereignisse die ControlEngine behandeln soll.

Ich habe die Steuerungselemente in das IDTool integriert. Die Ablaufsteuerung wird über Menüeinträge vorgenommen, und auf diesem Weg kann ebenfalls ein Werkzeug geöffnet werden, mit dem die Menge der zu erfassenden Informationen festgelegt wird. Eine Handhabungsvision für die Benutzung des IDTool verläuft z.B. folgendermaßen:

> „Der Benutzer startet das Programm IDTool. Auf dem Bildschirm erscheint das IDTool-Fenster. Im Menü *Execute* wählt der Benutzer den Eintrag *Load executable*. Es erscheint daraufhin ein (modaler) Dialog, in dem der Name einer ausführbaren Datei eingegeben werden kann. Der Benutzer gibt einen Dateinamen an und schließt den Dialog durch Klicken des zum Dialog gehörenden OK-Button. Das gewählte Programm wird nun geladen; es entsteht eine gewisse Wartezeit, bis die Programmdaten eingelesen und ausgewertet sind. Nach dieser Wartezeit wählt der Benutzer im Menü *Execute* den Eintrag *Select Classes*. Es öffnet sich ein Werkzeugfenster mit zwei nebeneinander stehenden Listboxen. Die linke Listbox zeigt alle Klassennamen, die im geladenen Programm vorhanden sind. Mit den vorhandenen Buttons kann der Benutzer alle Klassen oder einzelne Klassen in die rechte Listbox übernehmen, um die Objekte dieser Klassen in die Visualisierung aufzunehmen. Der Benutzer aktiviert in diesem Werkzeug den *Übernehmen*-Button, um seine Klassenauswahl festzuhalten.[2]
>
> Im Execute-Menü des IDTool wählt der Benutzer nun *Start*. Das geladene Programm wird ausgeführt, und es wird zeitgleich ein Interaktionsdiagramm angelegt. Das Diagramm enthält alle Objekte der zuvor ausgewählten Klassen und zeigt darüber hinaus diejenigen

1 Der Export von Diagrammen funktioniert über das Metafile-Format von Microsoft Windows. In der Unix-Version nicht verfügbar.

2 Das Werkzeug für die Klassenauswahl verhält sich nicht modal, während der zuvor erwähnte Dialog zum Öffnen einer Datei modal ist und damit gegen den Werkzeugbegriff von WAM verstößt (vgl. 4.4.5. und [Kilberth et al. 94], S. 128f.).

Objekte an, an denen die ausgewählten Objekte Methoden aufrufen. Der Benutzer kann Programmausführung und Diagrammerzeugung unterbrechen, indem er im *Execute*-Menü *Stop* wählt. Durch den Menüeintrag *Single Step* setzt er die Ausführung bis zum nächsten im Diagramm berücksichtigten Ereignis fort. “

Es wird vorkommen, daß die Auswahl der Klassen noch nicht alle Objekte zeigt, die für die Fragestellung des Benutzers interessant sind. In diesem Fall wird die Auswahl geändert, und das geladene Programm kann entweder fortgesetzt oder erneut gestartet werden.

7.3.3 Einordnung in die Taxonomie von Roman u. Cox

Ich habe bereits in Abschnitt 4.6 beschrieben, welchen Platz VOOP2 in der Kategorisierung von Softwarevisualisierungssystemen nach Roman und Cox [Roman und Cox 93] einnimmt. Offen blieben die Einordnungen nach den Dimensionen Benutzerschnittstelle und Präsentation, die ich hier nun für das IDTool nachreiche.

Roman und Cox unterteilen das Kriterium Benutzerschnittstelle in die beiden Subkriterien Grafisches Vokabular und Interaktion. Als gebräuchliche Bestandteile des grafischen Vokabulars nennen sie fünf Komponenten:

- Einfache grafische Objekte. In diese Rubrik fallen die Symbole, die das IDTool in Diagrammen verwendet.
- Zusammengesetzte Objekte. Damit sind grafische Objekte gemeint, die der Benutzer aus bereits vorhandenen grafischen Elementen zusammensetzen kann – das IDTool bietet diese Möglichkeit nicht.
- Visuelle Ereignisse. Darunter sind das Erscheinen, Verschwinden und Verändern von grafischen Elementen zu verstehen. Erst durch visuelle Ereignisse entsteht ein animiertes Diagramm. Das IDTool kennt als visuelles Ereignis das Erscheinen von neuen Diagrammelementen. Einmal eingefügte Elemente werden aber nicht automatisch entfernt oder verändert.
- Welten. Dieser meines Erachtens etwas irreführende Begriff betrifft die Anordnung der Elemente im Diagramm; eine „Welt“ entsteht, indem das Visualisierungssystem die angezeigten Elemente durch geometrische Anordnung zueinander in Beziehung setzt. In diesem Sinn schafft auch das IDTool eine „Welt“, da die Anordnung nicht willkürlich ist, sondern eine Bedeutung – die zeitliche Abfolge von Methodenaufrufen – ausdrückt.
- Mehrere Welten. Durch separate Anzeigen, die verschiedene Inhalte durch jeweils andere Anordnungen der grafischen Elemente ausdrücken, kann ein Visualisierungssystem mehrere Welten aufbauen. Das IDTool arbeitet nur mit einer Welt.

Das zweite Subkriterium ist die Interaktion, die durch Kontrollelemente (wie Menüs, Buttons etc.) oder durch direkte Manipulation, d.h. durch Interaktion mit den Diagrammelementen, erfolgt. Das IDTool verwendet beide Arten der Interaktion; die direkte Manipulation ist dabei die für das IDTool charakteristische Art der Interaktion.

Neben der Benutzerschnittstelle nehmen Roman und Cox die Präsentation als eigenes Kriterium in ihre Taxonomie auf und meinen damit die Semantik der erzeugten Visualisierung. Sie beschreiben dieses Kriterium unter vier Gesichtspunkten:

- Interpretation der Grafik. In der Annahme, daß Interaktionsdiagramme unter Softwareentwicklern allgemein bekannt sind, benötigen die vom IDTool erzeugten Diagramme keine

zusätzliche Erläuterung und zeichnen sich dadurch positiv aus: „Visualizations that require a little or no explanation are clearly preferable.“ [Roman und Cox 93]

- Analytische Präsentation. Darunter ist die Abstraktion von der technischen Ebene der Programmausführung zu verstehen. Das IDTool bietet eine analytische Präsentation, wenn Elemente 2. Ordnung dargestellt werden (mit diesem Thema wird sich der letzte Abschnitt dieses Kapitels befassen).
- Erklärende Präsentation. Bei einer erklärenden Präsentation lenkt das Visualisierungssystem die Aufmerksamkeit des Benutzers auf bestimmte Aspekte, indem automatisch Erklärungen oder grafische Hervorhebungen hinzugefügt werden. Das IDTool sieht solche Techniken nicht vor.
- Orchestrierung. Visualisierungssysteme können nicht nur eine Anzahl von Sichten auf einen Sachverhalt bieten („Mehrere Welten“, s.o.), sondern auch ähnliche Sachverhalte in mehreren Sichten nebeneinander stellen. Beispielsweise können so mehrere Ausführungen desselben Programms miteinander verglichen werden. Das IDTool bietet hierfür keine besondere Unterstützung.

Erklärende Präsentation und Orchestrierung findet man am ehesten in Systemen, die Algorithmen visualisieren und dabei die Eigenschaften unterschiedlicher Algorithmen hervorheben und gegenüberstellen. Eine spätere Erweiterung des IDTool halte ich in diesen beiden Richtungen für weniger interessant. Lohnender wird es sein, die Möglichkeiten der analytischen Präsentation zu untersuchen und damit ein besseres Verständnis des untersuchten Programms zu fördern.

7.4 Materialien des IDTool

7.4.1 Metamodell und zusätzliche Materialien

Die folgenden Abschnitte befassen sich mit der Architektur und der Implementierung des IDTools. Am Anfang steht die Frage, mit welchen Materialien das Werkzeug umgeht – wie in den bisherigen Kapiteln beschrieben wurde, verfügt VOOP2 über ein zentral verwaltetes Material, nämlich das Programmodell. Dennoch benutzt das IDTool zusätzliche eigene Materialien, wofür es mehrere Gründe gibt:

- Das IDTool muß zu den dargestellten Elementen grafische Informationen kennen, die im Programmodell nicht enthalten sind, z.B. die Position, an der sich ein Element innerhalb des Diagramms befindet.
- Interaktionsdiagramme können um Elemente erweitert werden, die für andere Werkzeuge völlig unerheblich sind, z.B. um textuelle Anmerkungen. Solche Elemente sollten aus dem Programmodell ferngehalten werden und müssen daher im IDTool als Materialien verwaltet werden.
- Während das Programmodell immer nur den aktuellen Zustand der Programmausführung kennt, stellen Interaktionsdiagramme auch die Vorgeschichte dar und müssen daher Informationen über bereits gelöschte Programmobjekte ebenso kennen wie die Abfolge aller beobachteten Methodenaufrufe (zur Erinnerung: für Methodenaufrufe gibt es keine Metamodell-Elemente, sondern sie werden durch Reaktionsmechanismen von der ControlEngine mitgeteilt).

Die zuletzt genannten Informationen über frühere Objekte und Methodenaufrufe könnten auch für andere Werkzeuge als das IDTool interessant sein. In diesem Fall besteht die Möglichkeit, das Metamodell dynamisch um neue Elemente zu erweitern (vgl. Abschnitt 4.4). Ansonsten können die erforderlichen Daten in einem Material verwaltet werden, das nur dem IDTool zugänglich ist.

7.4.2 Semantische und grafische Materialanteile: Subjects und Views

Wenn das IDTool also nicht nur das Programmodell als Material benutzt, sondern auch eigene Materialien verwaltet, kann man bei diesen Materialien grafische und semantische Anteile voneinander unterscheiden:

- Ein Teil der Eigenschaften betrifft ausschließlich die Darstellung der Diagrammelemente, so daß man hier von *grafischen Umgangsformen* und von *grafischen Attributen* sprechen kann. Eine grafische Umgangsform ist beispielsweise das Verschieben eines Objektes innerhalb des Diagramms durch den Benutzer, und Beispiele für grafische Attribute sind Informationen über die Anzeigeposition des Objekts. Der semantische Inhalt des Diagramms wird durch diese Art von Manipulationen nicht verändert.
- Auf der anderen Seite stehen die *semantischen Umgangsformen* und Attribute, die für jedes Diagrammelement ein spezifisches Verhalten ermöglichen. Ein Beispiel für eine semantische Umgangsform mit Objekten ist das Zusammenfassen mehrerer Objekte zu einer Komponente; semantische Attribute einer Nachricht wären etwa Sender, Empfänger und der Zeitpunkt, zu dem die Nachricht verschickt wird.

Weil es eine deutliche Unterscheidung zwischen dem grafischen und dem semantischen Anteil eines Darstellungselements gibt, kann man diese beiden Anteile in getrennten Klassen modellieren. Der Vorteil liegt dann in der *losen* Kopplung der beiden Anteile, so daß man die gleiche semantische Klasse für andere grafische Repräsentationen wiederverwenden kann und sich verschiedene grafische Darstellungen ein „semantisches Material“ teilen dürfen. Trotz dieser Unterscheidung sind sowohl die semantischen als auch die grafischen Materialien fachlich motiviert (vgl. z.B. [Bäumer et al. 95]).

Im unten vorgestellten Tool-Canvas-Framework führe ich zu diesem Zweck die abstrakten Klassen `tSubject` und `tView` ein, in deren Spezialisierungen die semantischen bzw. die grafischen Materialanteile modelliert werden. Jedem *View*[3] ist genau ein *Subject* zugeordnet, umgekehrt kann aber ein *Subject* durch mehrere *Views* repräsentiert werden. Man muß sich klarmachen, daß ein *View*-Objekt zunächst einmal eine *Sicht* auf das Material *Subject* darstellt, aber auf der anderen Seite selbst Material ist: Denn zum *View* zählen etwa Angaben, wo und wie etwas auf dem Bildschirm dargestellt wird, und wenn ein Diagramm gespeichert werden soll, müssen auch diese Informationen persistent gemacht werden.

Wenn in Diagrammen des IDTool Objekte dargestellt werden, gibt es dazu also zwei Materialklassen `tObjectSubject` und `tObjectView`. Die Klasse `tObjectSubject` enthält Methoden, die z.B. eine Referenz auf ein zugehöriges Meta-Objekt (im Programmodell) oder einen Objektidentifizierer (OID) liefern. `tObjectView` bietet Methoden an, um die Position oder Größe des angezeigten Objekts zu manipulieren oder um das Objekt vom Bildschirm zu entfernen.

[3] Zur Notation: *View* steht für ein Objekt des Typs `tView`, *Subject* für ein Objekt des Typs `tSubject` usw.

Das Werkzeug IDTool verfügt, der Konstruktionsmethode WAM folgend, über eine Funktions- und eine Interaktionskomponente (FK und IAK, siehe [Kilberth et al. 94]). Materialien werden üblicherweise von der FK verwaltet; die IAK muß dann Methoden der FK nutzen, um auf die Materialien zugreifen zu können. Bei den Materialklassen `tView` und `tSubject` weiche ich aber von dieser Praxis ab und ordne die *Subjects* eher der Funktionskomponente, die *Views* eher der Interaktionskomponente des IDTool zu. Dieser Ansatz basiert auf der Überlegung, daß es eine Reihe von grafischen Manipulationen gibt, die der Benutzer an der IAK vornimmt und die für die FK nicht interessant sind: Muß man etwa die FK mit dem Verschieben eines Darstellungselements an eine andere Bildschirmposition belasten? Weil die Klasse `tView` hier eine ambivalente Rolle (Material und Materialsicht) einnimmt, räume ich der IAK einen direkten Zugriff auf die `tView`-Objekte ein (was der Rolle von *Views* als Material*sicht* entspricht). Dieser Ansatz ist jedoch problematisch. Grundsätzlich gilt, daß die *Views* fachlich motivierte Materialien sind und damit der FK zugeordnet werden sollten. Diese Forderung soll gewährleisten, daß IAKs keine Kenntnisse über fachliche Aspekte haben müssen und dadurch um so leichter durch andere IAKs ersetzt werden können. Wenn die IAK nun aber selbst für die Views verantwortlich ist, besteht die Gefahr, daß sie zu viel Wissen über die Materialien und deren Umgangsformen enthält und damit um so schwerer auszutauschen ist. Ich halte die Zuordnung der Views zur IAK für vertretbar, wenn sichergestellt wird, daß dieses Wissen minimal bleibt. Der folgende Abschnitt wird z.B. zeigen, wie Funktionen zum Anordnen der *Views* in einer eigenen Klasse gekapselt werden können (dabei handelt es sich dann um eine Strategieklasse [Gamma et al. 95]). Damit ist es möglich, die FK auf einen Kern zu reduzieren, der nur die semantischen Eigenschaften eines Diagramms kennt. Indem unterschiedliche IAKs (mit verschiedenen *View*-Klassen, Strategieklassen etc.) auf diese FK aufgesetzt werden, kann dieselbe FK für diverse Ausprägungen von Diagrammen benutzt werden.

Die Überlegungen über *Subjects* und *Views* gelten für andere Werkzeuge, die zum Visualisierungssystem hinzukommen sollen, genauso wie für das IDTool. Aus diesem Grund sind die Materialklassen für *Subjects* und *Views* als abstrakte Oberklassen in dem bereits erwähnten Tool-Canvas-Framework enthalten, das ich im folgenden Abschnitt vorstelle.

7.5 Das Tool-Canvas-Framework

7.5.1 Hintergrund

Der Vorläufer des Visualisierungssystems VOOP2 wurde unter dem Namen VOOP entwickelt [Hamer und Friedrich 96] und war in der Lage, Visualisierungen in Form von Konstruktionsdiagrammen zu erzeugen. Für die Anzeige der Diagramme wurden damals diverse Klassen implementiert, die für Darstellung der Objekte, deren Verbindungen usw. zuständig waren. Die Anzeige bot keine Möglichkeit zur Interaktion oder zur direkten Manipulation; es war aber vorgesehen, in späteren Ausbaustufen derartige Möglichkeiten zu schaffen: „Mit den Objekten kann dann durch Mausaktion interagiert werden... Ein Mausklick öffnet ein kontextsensitives Menü. Das Menü ermöglicht, die Werte der Objektattribute und andere objektspezifische Informationen anzuzeigen...“ [Hamer und Friedrich 96]. Die Anforderungen an die Benutzerschnittstelle wiesen damit Merkmale auf, wie sie nun beim IDTool wiederkehren: Es sollen Diagramme angezeigt werden, die aus einem vorgegebenen Vorrat an Symbolen bestehen, eine bestimmte Struktur besitzen und auf Benutzerinteraktionen reagieren können. Diese Merkmale werden auch bei anderen Diagrammen auftreten, die für VOOP2 noch entwickelt werden sollen. Daher lag es nahe, eine wiederverwendbare Lösung zu finden oder neu zu entwickeln, die als Grundlage für die Diagrammbearbeitung im IDTool wie auch für weitere Werkzeuge dient.

7.5.2 Ziel des Frameworks

Bei der Implementierung des IDTool bestand ein großer Teil des Aufwands darin, die Benutzerschnittstelle zu programmieren und die direkte Manipulation zu ermöglichen. Ich habe vor der Entwicklung des Werkzeugs mehrere Klassenbibliotheken und Frameworks für diesen Einsatzzweck untersucht. Die Anforderungen waren:

- Unterstützung von Grafik, d.h. grafikspezifische Routinen zum Zeichnen, zum Selektieren von bereits gezeichneten Objekten, zur Ausgabe auf Druckern, gepufferte Anzeige etc.;
- Unterstützung portabler Programme (mindestens Unix und Microsoft Windows) in C++;
- nach Möglichkeit freie Verfügbarkeit der Bibliothek / des Frameworks;
- aktuelle Bibliothek / Framework, d.h. wird auch in Zukunft gepflegt;
- zusammen mit dem am Arbeitsbereich Softwarechtechnik erstellten Framework *BibV30* [Lilienthal 97] verwendbar.

Nach Untersuchung von InterViews [Linton et al. 92], Unidraw [Vlissides 90], Fresco [Churchill 95], wxWindows [Smart 95] und Qt [Troll 97] fiel die Wahl schließlich auf wxWindows und die auf wxWindows basierende Bibliothek OGL (Object Graphics Library) [Smart 95a]. Im Laufe der Entwicklung rückte die Bedeutung der von OGL angebotenen Funktionalität mehr und mehr in den Hintergrund, so daß ich mich entschied, auf OGL zu verzichten und stattdessen aus dem entwickelten Code ein eigenes Framework zu extrahieren.

Der Aufwand, ein eigenes Framework für die Grafik zu schaffen, ist vor allem deswegen angebracht, weil VOOP2 später über mehrere Werkzeuge zur Generierung von Diagrammen verfügen soll. Das IDTool ist also das erste aus einer Reihe von Werkzeugen, das die Grafikfähigkeiten des entwickelten Frameworks nutzt. Die wesentlichen Anforderungen an die grafische Benutzerschnittstelle sind unabhängig von der Art der Diagramme, die editiert werden sollen. Die gemeinsamen Kennzeichen sind, daß die Diagramme direkt editierbar sind und daß

ihre Struktur aus einem relativ einfachen Netz von Kanten und Knoten besteht (was in Verbindung mit Benutzerschnittstellen häufig unter *strukturierter Grafik* verstanden wird). Das Framework muß also insbesondere vorsehen, daß es Konzepte wie Kanten und Knoten gibt und daß die dadurch ausgedrückten Beziehungen auch bei der Manipulation des Diagramms erhalten bleiben.

Als konzeptuelles Vorbild habe ich mich beim Entwurf des Frameworks an Unidraw orientiert. Unidraw selbst wurde nicht verwendet, weil es seit einigen Jahre nicht mehr weiterentwickelt wurde und nur auf Unix, nicht aber auf anderen Plattformen implementiert ist. Das Framework Unidraw dient der Programmierung objektorientierter Grafikeditoren, ohne sich auf einen Anwendungsbereich festzulegen; vielmehr wird vom Anwendungsbereich abstrahiert, und auf der so geschaffenen abstrakten Ebene werden allgemeine, grundlegende Konzepte angeboten. Beispiele für Unidraw-Anwendungen sind einfache Zeichenwerkzeuge, ein Editor für elektronische Schaltkreise und ein Editor für Notensatz. Unidraw wird wegen seiner Architektur gerne als vorbildliches, objektorientiertes Framework angesehen, wohl nicht zuletzt aufgrund der vielen Entwurfsmuster, die dort zu finden sind.

Die von mir ursprünglich verwendete OGL verfolgt genau dieselbe Zielrichtung, ist aber vom Entwurf her konventioneller und weniger elegant als Unidraw. Zum Verzicht auf die OGL entschloß ich mich, nachdem ich in meiner Implementierung nur noch einige technische Funktionen der OGL benutzt hatte, ansonsten aber die Konzepte von Unidraw adaptiert hatte.

Als ich dem Framework den Namen Tool-Canvas-Framework gegeben habe, dienten die zwei zentralen Klassen `tTool` und `tCanvas` als Namensgeber. Im Laufe der „Evolution“ des Frameworks haben andere Klassen mehr und mehr an Bedeutung gewonnen, so daß letztlich andere Namen treffender wären. Dennoch habe ich es vorgezogen, die gewählte Bezeichnung beizubehalten.

7.5.3 Überblick

Die Elemente des Tool-Canvas-Frameworks sind:

- *Subjects* und *Views* (vgl. Abschnitt 7.4).

  Ein *Subject* beinhaltet einen semantischen Materialanteil, ein *View* den grafischen Anteil. Zu jedem *Subject* kann es mehrere *Views* geben, die dann in der Regel Bestandteile verschiedener Diagramme sind.

- Eine Diagramm-Klasse, die als Materialcontainer für *Views* dient und Operationen bereitstellt, die sich auf alle *Views* eines Diagramms beziehen.

- Ein *Canvas*, auf dem ein Diagramm dargestellt und bearbeitet werden.

- *Tools*, mit denen die angezeigten *Views* und die damit assozierten *Subjects* bearbeitet werden können.

- Manipulatoren, die von den *Tools* benutzt werden, um *View*- oder *Subject*-Eigenschaften durch den Benutzer interaktiv ändern zu lassen.

- *Commands*, mit denen die *Tools* eine Veränderung von *Subjects* und *Views* auslösen.

- Konnektoren, die mehrere *Views* miteinander verbinden.

- Factory-Klassen [Gamma et al. 95], die *View*-Objekte erzeugen.

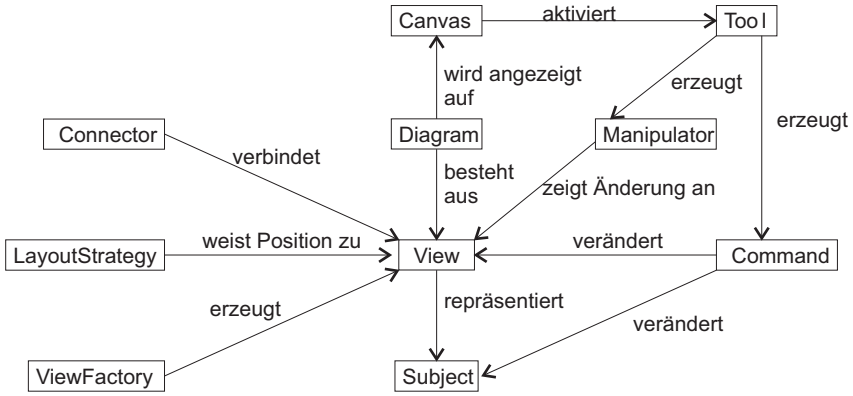


Abb. 22: Die wichtigsten Zusammenhänge des Frameworks

- Klassen, die ein bestimmtes Layout von Diagrammen implementieren.

Um mit dem Framework ein Werkzeug zu entwickeln, leitet man Unterklassen für *Subjects*, *Views* und *Commands* ab und implementiert eine spezielle *ViewFactory*. Die *Canvas*-Klasse kann i.a. instanziiert werden, ohne eine spezialisierte Klasse abzuleiten.

Für *Tools*, Manipulatoren und Konnektoren gibt es eine Auswahl existierender Klassen. Unter Umständen kommt eine Anwendung mit den vorgefertigten Klassen aus, ansonsten werden eigene Unterklassen abgeleitet.

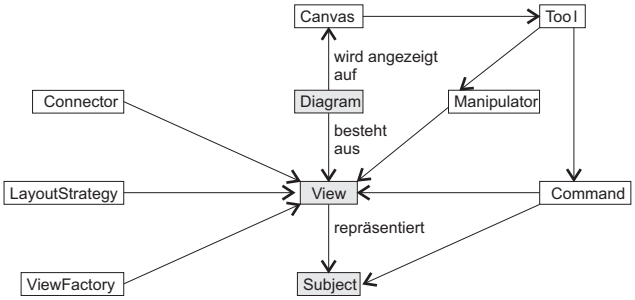
Die *ViewFactory*- und Layout-Klassen sind in Zusammenhang mit der Softwarevisualisierung für automatisch erzeugte Diagramme gedacht. Sie sollen dafür sorgen, daß zu den *Subjects*, die z.B. von einem Automaten generiert werden, die passenden *Views* erzeugt und sinnvoll im Diagramm plaziert werden.

Die wesentlichen Zusammenhänge zwischen den Klassen sind in Abbildung 22 zu sehen.

7.5.4 Die Klassen des Tool-Canvas-Frameworks

7.5.4.1 Material: Subject, View und Diagram

Für die Modellierung des Material gibt es zunächst zwei abstrakte Basisklassen, nämlich `tSubject` und `tView`. Eine von `tSubject` abgeleitete Klasse enthält semantische Informationen, während die Hierarchie der `tView`-Klassen für die grafische Repräsentation zuständig ist.



Die grundlegende grafische Funktionalität wird in der Klasse `tView` realisiert, deren Methoden die Anzeige von Diagrammelementen ermöglichen.

```
class tView
{
   void DrawOutline(Bool show);
   void DrawLabel(void);
   void Init(tCoord x, tCoord y, tCoord width, tCoord height);
   void Show(void);
   void Hide(void);
```

```
   void Move(tCoord x, tCoord y);
   void Resize(tCoord width, tCoord height);
   void Resize(float factor);
   void ShowOutline(void);
   void HideOutline(void);
   void HighLight(Bool SwitchOn);
   void Redraw(void);

   tCoord GetX(void);
   tCoord GetY(void);
   tCoord GetWidth(void);
   tCoord GetHeight(void);
   Bool IsShown(void);
   void SetSubject(tSubject* pSubject);
   tSubject* GetSubject(void);
   void SetLabel(String* pString);
   String* GetLabel(void);
};
```

Zu jedem *Subject* kann es eine beliebige Anzahl von *Views* geben, die dieses *Subject* darstellen. Dazu dienen Methoden `Attach` und `Detach`. `GetViews` liefert alle mit dem *Subject* verbundenen *Views*:

```
class Subject
{
   void Attach(View*);
   void Detach(View*);
   void SetLabel(String);
   String GetLabel();
   Set<View*> GetViews();
};
```

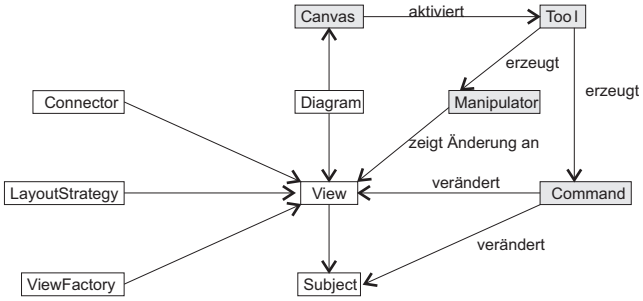
Ein Diagramm setzt sich aus einer Menge von `tView`-Objekten zusammen. Dazu gibt es die Klasse `tDiagram`, die als Materialcontainer für *Views* angesehen werden kann. Am Bildschirm kann ein Diagramm auf einer Zeichenfläche vom Typ `tCanvas` angezeigt werden. Die *Views* werden also immer in das Diagramm, nicht aber direkt auf den *Canvas* eingefügt.

Ein Diagramm kann gleichzeitig auf mehreren `tCanvas`-Objekten angezeigt und bearbeitet werden.[1] Mit jeder Manipulation muß dann eine Veränderung des Diagramms gleichzeitig in mehreren Fenstern sichtbar werden. Zu den Operationen, die von der Klasse `tDiagram` angeboten werden, zählen Methoden zum Laden und Speichern und zur Anordnung von *Views*.

1 noch nicht implementiert

7.5.4.2 Interaktivität: Canvas, Tool und Manipulator

Um Diagramme bearbeiten zu können, arbeiten `tTool`- und `tManipulator`-Objekte zusammen auf einem *Canvas*.[2] Auch wenn der *Canvas* zunächst ein Diagramm und erst indirekt *Views* anzeigt, interessieren im Zusammenhang mit den Benutzerinteraktionen vor allem die manipulierten *Views*; Aktionen, die sich auf das Diagramm als ganzes beziehen, sind die Ausnahme. Die Zusammenarbeit von *Canvas*, *Tool* und Manipulator muß also einen einfachen, direkten Zugriff auf einzelne *Views* zulassen.



Das grundlegende Zusammenspiel der Klassen soll an einem Beispiel erklärt werden. Ich gebe zunächst eine informelle Beschreibung und werde etwas später eine detaillierte Beschreibung in Form eines Interaktionsdiagramms nachreichen. Wir nehmen an, daß ein Benutzer eines Werkzeugs einen dargestellten *View* am Bildschirm verschieben möchte. Er tut dies mit der Maus, indem er das `tView`-Objekt an eine andere Position zieht (drag and drop). Im Inneren des Frameworks finden während dieser Aktion folgende Schritte statt:

- Der *Canvas* erhält ein Ereignis der Gestalt „Drag-and-drop an Position (x;y) beginnt“. Der *Canvas* ermittelt, daß an dieser Position ein `tView`–Objekt liegt, und folgert, daß ein Verschieben des *Views* erfolgen soll.
- Für Operationen wie das Erzeugen, Verändern und Löschen von *Views* sind eine Reihe von `tTool`-Objekten zuständig. In diesem Fall gibt es für das Verschieben ein *MoveTool*. Der *Canvas* bestimmt, daß das *MoveTool* zum aktiven *Tool* wird, und fragt beim *Tool* an, welcher Manipulator benutzt werden soll. Das *MoveTool* liefert einen *DragViewManipulator*.
- Solange der Benutzer die Maustaste gedrückt hält, bleibt der Manipulator aktiv und wird vom *Canvas* kontinuierlich mit der aktuellen Bildschirmposition versorgt. Der *DragViewManipulator* bewegt ein Abbild des *Views* an diese Position und zeigt damit den Effekt an, den der Benutzer durch seine Aktion erhält.
- Wenn die Maustaste losgelassen wird, teilt der *Canvas* dem Manipulator mit, daß die Aktion zu Ende ist. Der Manipulator merkt sich die letzte Position, die er erhalten hat, und entfernt das *View*-Abbild, das zur Anzeige der Aktion benutzt wurde, vom Bildschirm.
- Der *Canvas* weist nun das aktive *Tool* an, den Manipulator zu interpretieren. Das *MoveTool* holt sich also vom *DragViewManipulator* die letzte Mausposition und überprüft, ob das Verschieben des *Views* von der ursprünglichen an die neue Position zulässig ist.

Wenn das Verschieben zulässig ist, erzeugt das *MoveTool* nun ein *MoveCommand* und gibt dieses als Ergebnis seiner Manipulator-Interpretation an den *Canvas* zurück. Der *Canvas* gibt das Kommando an einen Kommandoprozessor weiter, der es ausführt (und speichert, um es bei Bedarf rückgängig zu machen).

[2] Der Begriff *Tool* nimmt den Sprachgebrauch von UniDraw auf und hat nichts mit dem in dieser Arbeit verwendeten Begriff *Werkzeug* zu tun.

7.5.4.3 Canvas

Der *Canvas* ist die Zeichenfläche, auf der Diagramme angezeigt und mit Hilfe von *Tools* Veränderungen an *Views* vorgenommen werden. Technisch hat der *Canvas* drei wesentliche Aufgaben:

- *Events* (Maus- und Tastaturereignisse) entgegenzunehmen,
- die Intention eines *Events* zu bestimmen, indem ein zum *Event* passendes *Tool* aufgerufen wird,
- zu gegebenen Bildschirmkoordinaten ein *View*-Objekt zu lokalisieren (d.h. ein *Event* einem *View* zuzuordnen).

```
class tCanvas
{
   void RegisterTool (tTool*, tEventType, tClassInfo*);
   void UnregisterTool (tTool*);
   tTool* GetActiveTool (void);
   tManipulator* GetActiveManipulator (void);
   tTool* FindSuitableTool (tEventType, tClassInfo*);
   tView* FindView (tCoord x, tCoord y, tClassInfo*);

   // define the classes that are sensitive to mouse operations
   // and to tools - set sensitivity for a class
   // (which must be a subclass of tView)
   void SetSensitivity (tClassInfo* pClassInfo);
   void ResetSensitivity (void);
   Bool IsClassSensitive (tClassInfo* pClassInfo);

   tViewSelection* GetSelection(void);
   void Redraw(void);
};
```

Beim *Canvas* können sich *Tools* für bestimmte Ereignisse anmelden (Methode `RegisterTool`), so daß der *Canvas* bei Eintreten eines Ereignisses das passende *Tool* aktiviert. Optional kann bei der Registrierung eine Klasse angegeben werden, auf deren Exemplare die Aktivierung des *Tools* beschränkt werden soll. Das im Beispiel verwendete *MoveTool* würde man für das Ereignis *„LeftDrag“* und die Klasse `tView` anmelden. Der *Canvas* wird das *MoveTool* dann aufrufen, sobald ein Drag-Vorgang mit der linken Maustaste beginnt und gleichzeitig ein von `tView` abgeleitetes Objekt an der Startposition des Drag-Vorgangs gefunden wird. Für die Auswahl eines zu einem *Event* passenden *Tools* wird die Methode `FindSuitableTool` benutzt.

Ein Feature des *Canvas* ist das Hervorheben von `tView`-Objekten, sobald sich die Maus über dem Objekt befindet. Das Positionieren der Maus wird damit erheblich erleichtert. Diese Eigenschaft kann auf beliebige `tView`-Unterklassen beschränkt werden; diese Klassen werden als „sensitive“ bezeichnet. Um dieses Feature zu nutzen, stehen die Methoden `SetSensitivity`, `ResetSensitivity` und `IsClassSensitive` zur Verfügung.

Außerdem verfügt jeder *Canvas* über ein internes `tViewSelection`-Objekt, das eine Auswahl von *Views* enthält und über `GetSelection` abgefragt werden kann. Die Auswahl kann mit der Maus vorgenommen werden – was allerdings nicht mehr in die Zuständigkeit des *Canvas* fällt, sondern Aufgabe eines speziellen *Tools* ist.

7.5.4.4 Tool

Ein *Tool* hat den Zweck, für eine Veränderung des Diagramms einen passenden Manipulator zur Verfügung zu stellen und das Ergebnis der Manipulation in ein Kommando-Objekt zu überführen. Es gibt eine Reihe von bereits im Framework implementierten *Tools*, die auf alle *Views* anwendbar sind: Sie dienen dem Vergrößern (`ResizeTool`), Verschieben (`MoveTool`), Beschriften (`InputLabelTool`) und Selektieren (`SelectWithClickTool` und `SelectWithDragTool`). Jedes *Tool* ist ein Nachfahre der abstrakten Oberklasse `tTool`:

```
class tTool
{
   tManipulator* CreateManipulator (tEvent* pEvent);
   tCommand* InterpretManipulator (tManipulator* pManipulator);
   void Activate (Bool Active);
   Bool IsApplicable (tEvent* pEvent);
};
```

Wie bereits beschrieben wurde, können *Tools* beim *Canvas* für bestimmte Ereignisse registriert werden, so daß der *Canvas* bei Eintreten des Ereignisses das *Tool* aufruft (präziser: einen Manipulator durch die Methode `CreateManipulator` anfordert). Die Methode `Activate` kann das *Tool* deaktivieren und aktivieren, ohne daß eine `Unregister` bzw. erneutes `Register` beim *Canvas* notwendig ist. Durch Überschreiben der Methode `IsApplicable` lassen sich außerdem zusätzliche Vorbedingungen formulieren, die von `Canvas::FindSuitableTool` abgefragt werden.

Die Methode `InterpretManipulator` erhält den Manipulator, nachdem der Benutzer den Vorgang abgeschlossen hat, und muß daraufhin ein Kommando-Objekt (*Command*) erzeugen. `CreateManipulator` und `InterpretManipulator` müssen von jedem *Tool* überschrieben werden.

7.5.4.5 Manipulator

Manipulatoren haben die Aufgabe, während einer Benutzeraktion deutlich zu machen, welche Auswirkung die Aktion haben wird. Typischerweise sind Manipulatoren für Drag-Vorgänge erforderlich, um z.B. während des Verschiebens eines *Views* immer das *View* an seiner zwischenzeitlichen Position zu zeigen. Die Zusammenarbeit von *Tools* und Manipulatoren stammt aus Unidraw: Dort hatte man diese Unterscheidung ursprünglich nicht vorgenommen, dann aber festgestellt, daß viele *Tools* ein ähnliches Verhalten teilten, u.U. ein einziges *Tool* aber auch mehrere unterschiedliche Verhaltensweisen haben kann [Vlissides und Linton 89]. Ein Beispiel für einen nicht an ein einzelnes *Tool* gebundenen Manipulator ist die Klasse `tStretchRectangleManipulator`, die während eines Drag-Vorgangs ein „Gummirechteck“ anzeigt und für unterschiedliche *Tools* (Auswahl von *Views*, Größe eines *Views* ändern etc.) benutzt werden kann.

```
class tManipulator
{
   void Init(tCoord x, tCoord y);
   void ShowEffect (tCoord x, tCoord y);
   void Cancel(void);

   tCoord GetX(void);
   tCoord GetY(void);
```

```
    //coordinates where the manipulator was initialized:
    tCoord GetSourceX(void);
    tCoord GetSourceY(void);

    Protected:
    virtual void HideEffect (void);
};
```

Bei Beginn einer Benutzeraktion erzeugt ein *Tool* einen Manipulator und initialisiert diesen. Der initialiserte Manipulator wird dem *Canvas* übergeben, welcher im weiteren Verlauf immer wieder die aktuelle Mausposition an den Manipulator übermittelt, indem die Methode `ShowEffect` aufgerufen wird. Wenn die Manipulation abgeschlossen ist, läßt sich das Ergebnis durch Methoden wie `GetX` und `GetSourceX` abfragen.

Während der Entwicklung des IDTools ist die Klasse `tManipulatorConstraint` in das Framework aufgenommen worden, um denselben Manipulator mit unterschiedlichen Verhaltensweisen zu benutzen: So können im IDTool Objekte nur in horizontaler, Methodenaufrufe nur in vertikaler Richtung verschoben werden. Für beide Aktionen wird ein allgemeines `tDragViewManipulator`-Objekt benutzt, dem verschiedene `tManipulatorConstraint`-Objekte übergeben werden, um die Durchführung des Verschiebens entsprechend zu beschränken.

„Typische“ Manipulatoren werden während eines Drag-Vorgangs aktiviert und, entsprechend der Mausbewegung, kontinuierlich aufgefrischt. Manipulatoren können aber auch ganz andere Formen annehmen: So gibt es z.B. eine Klasse `tStringInputManipulator`, die einen (modalen) Dialog zur String-Eingabe zur Verfügung stellt. Eine vorgefertigte Klasse, die diesen Manipulator benutzt, ist `tInputLabelTool`, mit dem auf diese Art ein Label für ein `tView`-Objekt eingebeben werden kann.

Ebenso können mit Manipulatoren kontextsensitive Menüs erstellt werden, die sich für die verschiedenen Arten von *Views* aufrufen lassen. Ein entsprechendes *Tool* erhält auch hier die Aufgabe, aus dem Ergebnis des Manipulators Kommandos zu erzeugen, indem die ausgewählte Menüoption erfragt und ausgewertet wird.

7.5.4.6 Command

Die Aktionen, die mit *Tools* und Manipulatoren vorgenommen werden, führen nicht zu einer direkten Veränderung von Materialien; statt dessen werden Kommandoobjekte [Gamma et al. 95] erzeugt, die die intendierte Änderung beschreiben. Alle Kommandoobjekte erben von einer abstrakten Klasse `tCommand`. Diese Klasse sieht bereits die Möglichkeit vor, Aktionen rückgängig zu machen. Die einzigen Methoden, die durch `tCommand` vereinbart werden, sind `Do` und `Undo`. Spezifische Informationen, die für abgleitete Kommandos nötig sind, müssen als Parameter der Unterklassen-Konstruktoren übergeben werden.

```
class tCommand
{
    void Do(void);
    void Undo(void);
};
```

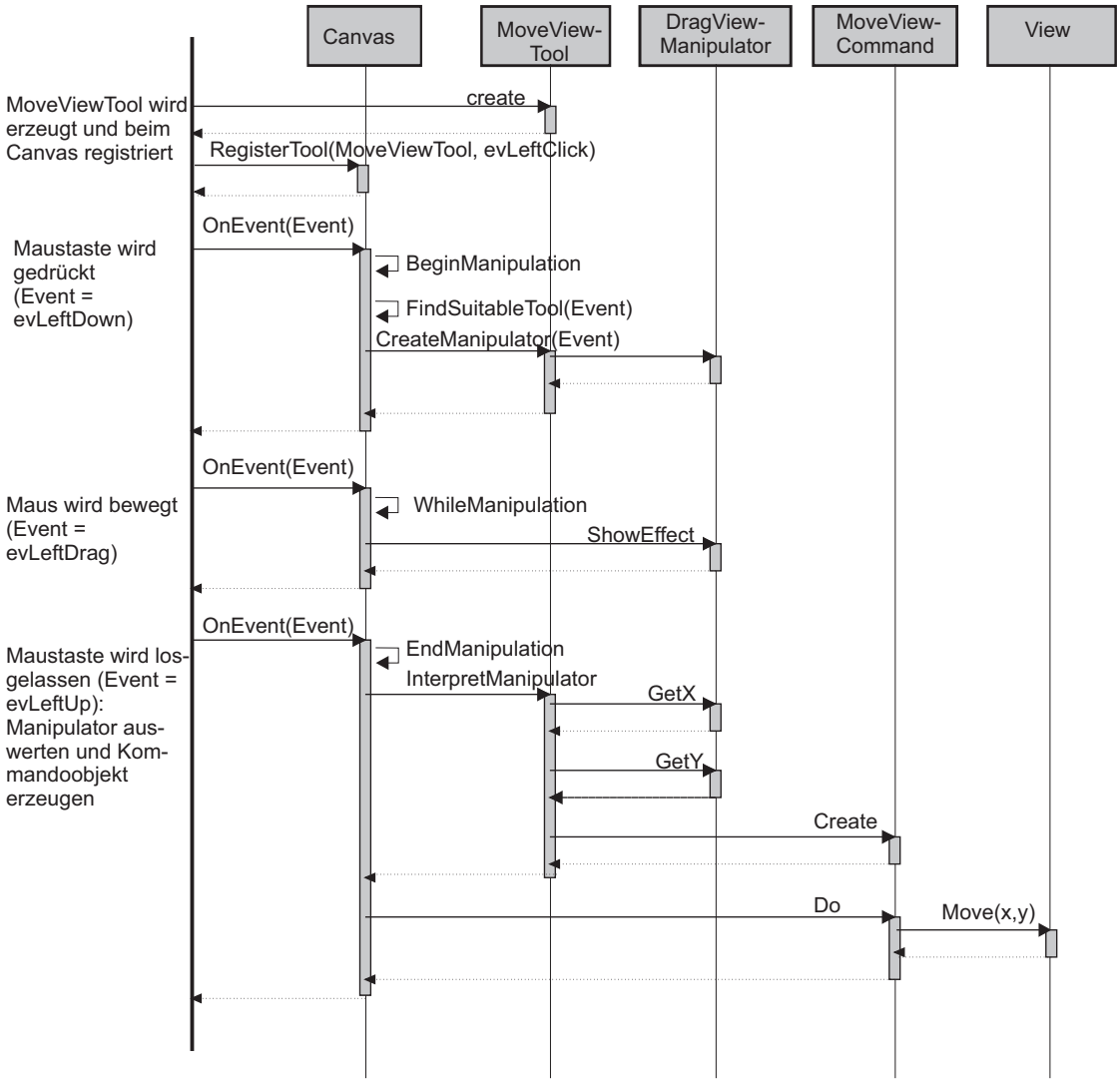


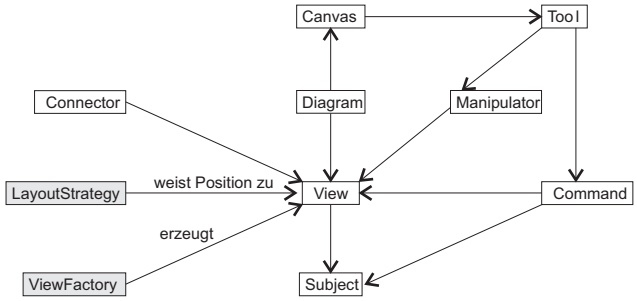
Abb. 23: Interaktives Verschieben eines View-Objekts

7.5.5 Beispiel

Nachdem die Schnittstellen der Klassen erläutert wurden, soll das Zusammenwirken der Methoden nochmals an dem Beispiel illustriert werden, das bereits einige Seiten zuvor erwähnt wurde (Abschnitt 7.5.4.2). Abbildung 23 zeigt das Interaktionsdiagramm für das Verschieben eines *Views* per Drag-and-Drop. Durch die vorangegangenen Erklärungen zu den beteiligten Klassen sollte die Abbildung verständlich sein.

7.5.6 Automatischer Aufbau: ViewFactory und LayoutStrategy

Die bisher vorgestellten Klassen eignen sich allgemein für Grafikeditoren, ohne besondere Rücksicht auf die Anforderungen im Rahmen der Softwarevisualisierung zu nehmen. Da das Tool-Canvas-Framework neben der interaktiven Manipulation von Diagrammen vor allem auch deren automatische Erstellung erlauben soll, gibt es zwei abstrakte Basisklassen, `tViewFactory` und `tLayoutStrategy`. Die beiden Klassen legen ein einfaches Protokoll fest, um zu einem automatisch erzeugten *Subject* ein passendes *View* zu generieren und diesem *View* den richtigen Platz im Diagramm zuzuweisen.



7.5.6.1 ViewFactory

`tViewFactory` stellt eine abstrakte Fabrik [Gamma et al. 95] dar, die zu einem gegebenen *Subject* ein *View*-Objekt liefert. Dazu ist nur eine einzige Methode, `CreateView`, nötig.

```
class tViewFactory
{
   tView* CreateView(tSubject* pSubject);
};
```

Für jede Art von Diagramm wird eine spezialisierte Klasse von `tViewFactory` abgeleitet, die in der Regel zusätzliche Methoden für die Erzeugung der einzelnen *View*-Objekte enthält (etwa `CreateObjectView`, `CreateMessageView` etc.). Die Methode `CreateView` muß so überschrieben werden, daß in Abhängigkeit vom *Subject*-Parameter das richtige *View* erzeugt wird.

Der Vorteil dieser `tViewFactory`-Schnittstelle liegt darin, daß die Erzeugung von *Views* stark verallgemeinert werden kann. Erst durch die durchgehende Verfügbarkeit der `CreateView`-Methode wird es möglich, ein allgemeines Protokoll zu schaffen, das *Views* in jede Art von Diagramm einfügt.[3]

7.5.6.2 LayoutStrategy

Nach der automatisierten Erzeugung von *Views* bleibt das Problem, die *Views* sinnvoll im Diagramm anzuordnen. Die abstrakte Klasse `tLayoutStrategy` enthält eine Schnittstelle, zu der unterschiedliche Layout-Verfahren implementiert werden können. Zu ihrem Namen gelangt diese Klasse, weil es sich um ein Strategie-Muster [Gamma et al. 95] handelt.

```
class tLayoutStrategy
{
   tLayoutStrategy(tDiagram* pDiagram);
```

[3] Diese Begründung kann noch präzisiert werden: Man könnte statt eines Fabrik-Create auch jedem Subject eine Methode `CreateView` verpassen. Damit wäre man aber auf eine bestimmte Repräsentation für ein *Subject* festgelegt. Der Vorteil der Fabrik ist also, die Abbildung von *Subject*s auf *Views* zu kapseln und austauschbar zu machen.

```
    void AssignPosition(tView* pNewView);
    tCommand* Recalc();
};
```

Im Konstruktor wird jeder *LayoutStrategy* bereits das Diagramm mitgeteilt, auf das sich die folgenden Operationen beziehen. Die Methode `Recalc` berechnet das Layout des gesamten Diagramms, während `AssignPosition` ein einzelnes, neu einzufügendes *View* positioniert. Falls `tManipulatorConstraint`-Klassen (s.o.) erstellt wurden, bietet es sich an, in `AssignPosition` die Zulässigkeit von Positionen mit Hilfe der `tManipulatorConstraint`-Objekte zu prüfen.

Durch diese Layout-Klasse wird die Anordnung eines Diagramms vom Diagramm selbst getrennt. Damit ist es möglich, für dasselbe Diagramm zwischen verschiedenen Layout-Strategien zu wechseln und für unterschiedliche Diagrammklassen dieselbe Layout-Implementierung zu verwenden. So ist es auch vorstellbar, eine allgemeine Klasse `tTreeLayout` zu erstellen, die ein Diagramm als Baum anordnet, ohne von spezialisierten *Diagram*- oder *View*-Klassen abhängig zu sein.

7.5.6.3 Beispiel

Abbildung 24 zeigt ein Interaktionsdiagramm für ein Szenario, in dem ein neues `tView`-Objekt erzeugt wird. Die Funktionskomponente einer Anwendung ruft eine Methode `GenerateNewView` der Interaktionskomponente auf und übergibt einen Verweis auf das *Subject*, zu dem ein *View* erstellt werden muß. Durch eine Instanz der Klasse `tNewViewCommand` (die bereits im Tool-Canvas-Framework vorhanden ist) wird nun ein neues *View* von einer spezialisierten *ViewFactory* angefordert und unter Benutzung einer bestimmten Layout-Strategie in das Diagramm eingefügt.

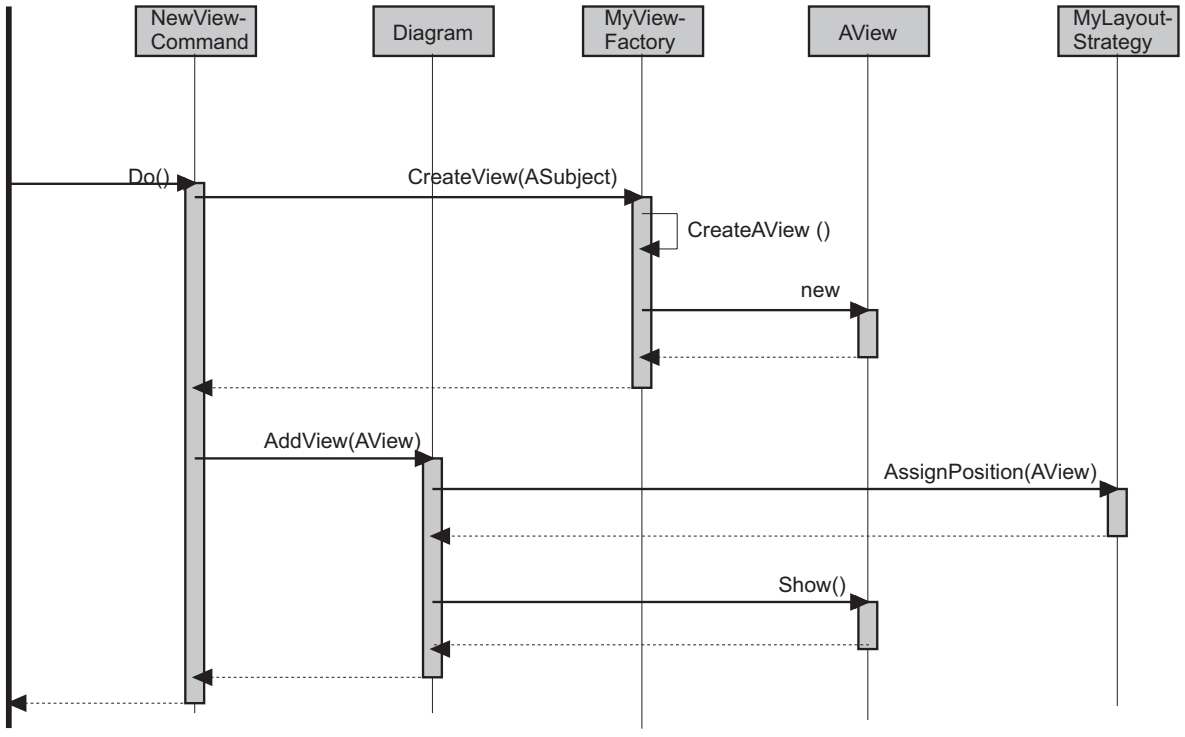


Abb. 24: Erzeugen neuer Views mit ViewFactory und LayoutStrategy

7.5.7 Nutzen

Bei der Erstellung von Grafik-Editoren entfällt in der Regel ein großer Teil des Aufwands auf die Programmierung der Benutzerschnittstelle. Das Tool-Canvas-Framework soll diesen Aufwand deutlich minimieren. Den wesentlichen Beitrag dazu liefern die Klassen `tCanvas`, `tTool` und `tManipulator`, durch die (a) ein klares Konzept für den Entwurf von Editoren vorgebeben und (b) bereits ein guter Teil der Interaktionsfunktionalität implementiert ist. Dieser Teil des Frameworks ist unabhängig von Einsatzkontexten, und es ist möglich, alleine aus den vorgegebenen *Tools* und Manipulatoren einen Editor zu erstellen.

Subjects und *Views* müssen anwendungsspezifisch modelliert werden. Dabei kann im Extremfall ganz auf *Subjects* verzichtet werden, falls in der Anwendung keine semantischen Eigenschaften zu den bearbeiteten Diagrammen abgebildet werden müssen. Konkrete *Views* enthalten im wesentlichen eine Beschreibung, wie die grafische Darstellungung am Bildschirm gezeichnet werden muß. Will man die *Views* mit einer *ViewFactory* erzeugen, muß eine Unterklasse von `tViewFactory` abgeleitet werden. Eine *LayoutStrategy*-Klasse muß ebenfalls, sofern sie erforderlich ist, abgeleitet werden.

Alle Veränderungen, die vom Benutzer an *Subjects* und *Views* vorgenommen werden, sind durch Kommando-Objekte zu realisieren. Dadurch entsteht zwar ein gewisser Aufwand, weil diverse Unterklassen von `tCommand` erstellt werden müssen; als Lohn dieser Arbeit erhält man eine durchgehende Undo/Redo-Funktionalität.

7.5.8 Diskussion der Framework-Eigenschaften

In [Gamma et al. 95] wird Framework definiert als „eine Menge kooperierender Klassen, welche die Elemente eines wiederverwendbaren Entwurfs für eine bestimmte Art von Software darstellen. Ein Framework bietet eine Architekturhilfe beim Aufteilen des Entwurfs in abstrakte Klassen und beim Definieren ihrer Zuständigkeiten und Interaktionen. Ein Entwickler paßt das Framework für eine bestimmte Anwendung an, indem er Unterklassen der Frameworkklassen bildet und ihre Objekte zusammensetzt."

Die vorgestellte Tool-Canvas-Architektur besitzt die Charakteristika, die in dieser Definition genannt werden; hervorzuheben ist insbesondere das Vorliegen *kooperierender* Klassen (sehr deutlich bei der Verzahnung von *Tools*, Manipulatoren und Kommandos) und die *Architekturhilfe*, die dem Entwickler eines Diagrammeditors durch das Tool-Canvas-Konzept gegeben wird.

Was jedoch nicht zu den vorgefertigten Klassen zählt, ist eine *Anwendungsklasse*, die Programme als höchste Einheit des Entwurfs erfaßt. Viele Frameworks bieten solche Klassen (meistens unter einem Namen wie `TApplication`), durch die für die gesamte Anwendung ein Rahmen vorgegeben wird. Sie erfüllen damit, was Walter Bischofberger als Framework beschreibt: „Frameworks gehen viel weiter, indem sie die gesamte Architektur einer Standardlösung kapseln. Der Benutzer muß sich nicht mehr darum kümmern, wie das Framework gesteuert wird." [Bischofberger 95]. Das gilt nicht für die Tool-Canvas-Architektur, in der der *Canvas* das umfassendste Konstrukt darstellt, das in einer vom Benutzer zu erstellenden Gesamtarchitektur verwendet wird.

Der Verzicht auf eine Anwendungsklasse geschieht bewußt: Der Benutzer ist zwar noch nicht von der Last befreit, einen eigenen Entwurf für die Anwendung selbst zu erstellen. Das bringt aber auf der anderen Seite den erheblichen Vorteil, daß sich die Tool-Canvas-Architektur um so leichter in unterschiedliche Systeme, besonders auch in andere Frameworks, integrieren läßt.

Bischofberger erklärt selbst, daß Frameworks auf verschiedenen Ebenen einer Anwendung verwendet werden können und daß ein Framework auf mehreren anderen basieren kann. Auf diese Art lassen sich unterschiedliche Frameworks in einem großen *Application Framework* vereinigen [Bischofberger 95].

Nach dieser Erklärung ist die Tool-Canvas-Architektur kein Application Framework, kann aber als untergeordnetes Framework in ein solches eingesetzt werden. So wird für das IDTool das Framework BibV30 [Lilienthal 97] als Application Framework verwendet.

7.5.9 Vergleich mit UniDraw

Mit dem Zusammenhang von *Tool* und Manipulator einerseits und der Trennung von *Subject* und *View* andererseits sind in das Tool-Canvas-Framework zwei zentrale Ideen von Unidraw eingegangen. Die entsprechenden Klassen haben in beiden Frameworks vergleichbare Schnittstellen.

Nahezu identisch ist die Verwendung von Kommando-Objekten, die die Auswirkung von Manipulationen kapseln. Die Kommandos des Tool-Canvas-Framework entsprechen dem Kommando-Muster in seiner üblichen Form, in der die Kommandos von einer Instanz des Werkzeugs (z.B. einem Kommandoprozessor) verarbeitet werden, während die Unidraw-Kommandos wie Nachrichten an die Materialien (*Subjects* und *Views*) geschickt und dort interpretiert werden.

Unidraw bezieht die Semantik von Diagrammen ein, wozu Zustandsvariablen eingeführt werden. Mit der grafischen Verbindung zweier Komponenten werden deren Zustandsvariablen aneinander gebunden, was einen „Datenfluß“ durch das Diagramm ermöglicht. Durch Transferfunktionen wird die Funktionalität von Komponenten im Rahmen dieses Datenflusses modelliert. Ein Anwendungsbeispiel ist ein Schaltnetzeditor, in dem die logischen Funktionen elektronischer Bauteile durch solche Transferfunktionen beschrieben werden. Das Tool-Canvas-Framework bietet dagegen keinen solchen Ansatz; statt dessen muß der Benutzer alle semantischen Aspekte selbst in den *Subjects* realisieren.

Die `tCanvas`-Klasse des Tool-Canvas-Frameworks entspricht im wesentlichen der Unidraw-Klasse `Editor`, wobei es dort aber zusätzlich eine `Viewer`-Klasse gibt, die für die rein darstellenden Aufgaben zuständig ist; erst ein Editor-Objekt (in das ein *Viewer* eingebettet ist) ermöglicht das Bearbeiten von grafischen Komponenten.

Im Gegensatz zum Tool-Canvas-Framework sieht Unidraw keine Schnittstelle für Layoutalgorithmen vor. In [Vlissides 90] wird eine derartige Erweiterung aber als zukünftige Arbeit vorgeschlagen: „The architecture could include an object that ... positions components according to a specification. Such an object could use established layout algorithms and heuristics to produce pleasing component layouts.“

Bei Unidraw handelt es sich um ein *Application Framework*. Die Anwendungsklasse trägt wie das Framework den Namen Unidraw und stellt u.a. Methoden wie `Run`, `Open` und `Close`, eine Main Loop und einen Kommandoprozessor bereit. Genau diese Aufgaben werden beim Tool-Canvas-Framework der einbettenden Anwendung bzw. dem einbettenden Application Framework überlassen.

7.6 Architektur des IDTool

Das IDTool benutzt zwei Frameworks, nämlich das *Application Framework* BibV30 und das gerade besprochene Tool-Canvas-Framework. Auf die Verwendung des Tool-Canvas-Frameworks werde ich erst im nächsten Abschnitt eingehen. Vorher soll die Architektur des IDTool im Überblick vorgestellt werden.

7.6.1 Architektur

Das BibV30-Framework legt die Struktur der Anwendungen fest: Die zentralen Klassen für Interaktions- und Funktionskomponente werden von den BibV30-Klassen `tFKBase` und `tIAKBase` abgeleitet; im IDTool entstehen so die spezialisierten Klassen `tIDToolFP` und `tIDToolIP`. Sie stellen den funktionalen bzw. interaktiven Kern des Werkzeugentwurfs dar.

Ich habe bereits argumentiert, daß bei den Diagrammen, die mit dem IDTool bearbeitet werden, semantische und grafische Anteile voneinander getrennt werden sollen. Das Tool-Canvas-Framework gibt dafür die Oberklassen `tSubject` und `tView` vor. Das IDTool verwaltet *Subjects* und *Views* als eigene Materialien. Als Materialcontainer werden dafür zwei Klassen eingeführt: `tScenario` für die semantischen Anteile (*Subjects*) und `tInteractionDiagram` für die grafischen Anteile (*Views*).

Das dritte Material, das vom IDTool benutzt wird, ist das Programmodell. Über Änderungen des Programmodells wird die Funktionskomponente `tIDToolFP` von der ControlEngine benachrichtigt. Das IDTool erfragt dann die nötigen Informationen, um das Szenario (`tScenario`) fortzuschreiben und das Diagramm (`tInteractionDiagram`) zu ergänzen.

Abbildung 25 zeigt den Zusammenhang der genannten Klassen (die nur einen Teil des IDTool ausmachen). Die mit dem Tool-Canvas-Framework zusammenhängenden Klassen werden im folgenden Abschnitt behandelt.

7.6.2 IDTool und Tool-Canvas-Framework

7.6.2.1 Verwendung des Frameworks

Die Oberfläche des IDTool verfügt über eine Zeichenfläche, auf der die Interaktionsdiagramme angezeigt werden. Für diese Zeichenfläche wird ein Objekt der Klasse `tCanvas` aus dem

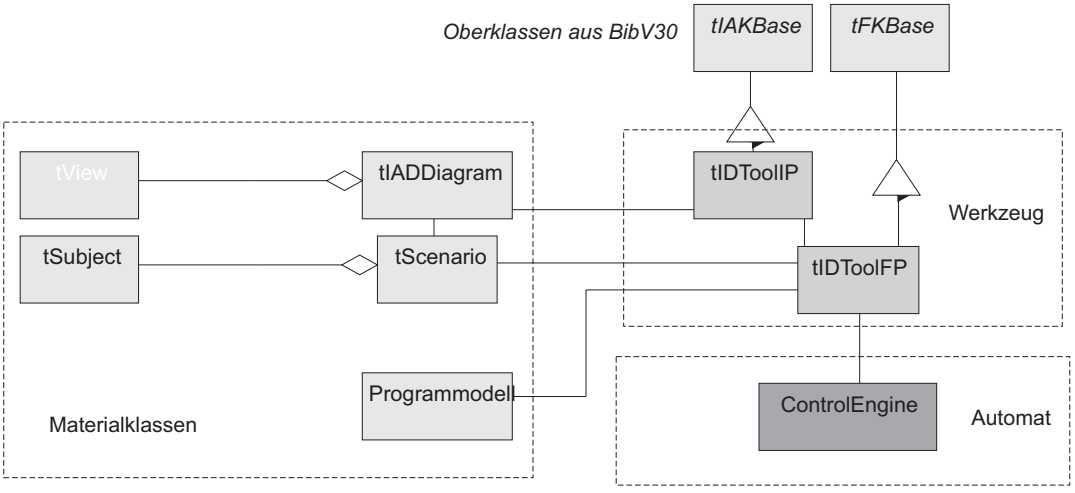


Abb. 25: Architektur des IDTool im Überblick

Tool-Canvas-Framework benutzt. Die Klasse arbeitet auf *Diagram*-Objekten, wofür im IDTool eine Klasse `tInteractionDiagram` abgeleitet wird, welche die spezifischen grafischen Umgangsformen von Interaktionsdiagrammen enthält. Eine solche spezifische Umgangsform ist z.B. das „Auseinanderziehen“ eines bestehenden Diagramms, um Platz für einen einzufügenden Methodenaufruf zu schaffen. Die eigentlichen (grafischen) Diagrammelemente aber werden in eigenen *View*-Objekten gespeichert. Aufgabe von `tInteractionDiagram` ist es, die *View*-Objekte zu verwalten.

Für das IDTool ist eine Anzahl verschiedener *View*-Klassen erstellt worden. Sie alle erben indirekt von der abstrakten Oberklasse `tView` und repräsentieren folgende Elemente von Interaktionsdiagrammen:

- Objekte (`tObjectView`) und deren Objektlinien (`tObjectLineView`)
- unterschiedliche Darstellungen von Interaktion zwischen den Objekten: Aufrufe von Methoden an anderen Objekten (`tExternMessageView`), am selben Objekt (`tReflexive-MessageView`) und die Rückkehr von Methodenaufrufen (`tReturnView`)
- die Bereiche, in denen Objekte aktiv sind (`tControlRegionView`)
- die Systemabgrenzung (`tSystemBorderView`)

Eine wichtige Eigenschaft der Interaktionsdiagramme besteht darin, daß alle Nachrichten an einem *ObjectView* oder *SystemBorderView* beginnen und enden müssen. Außerdem sollen sie, sobald eines dieser beiden *Views* bewegt wird, ebenfalls ihre Position anpassen. Das Verhalten, das für die Nachrichten benötigt wird, stellt die allgemeine Klasse `tEdgeView` zur Verfügung. Um `tObjectView` und `tSystemBorderView` in dieser Hinsicht einheitlich behandeln zu können, wird für beide die gemeinsame Oberklasse `tIADNodeView` eingeführt.

Mit den *Views* korrespondieren vier *Subject*-Klassen, die die semantischen Anteile von Objekten, Methodenaufrufen, Rücksprüngen und Systemabgrenzung modellieren: Im wesentlichen

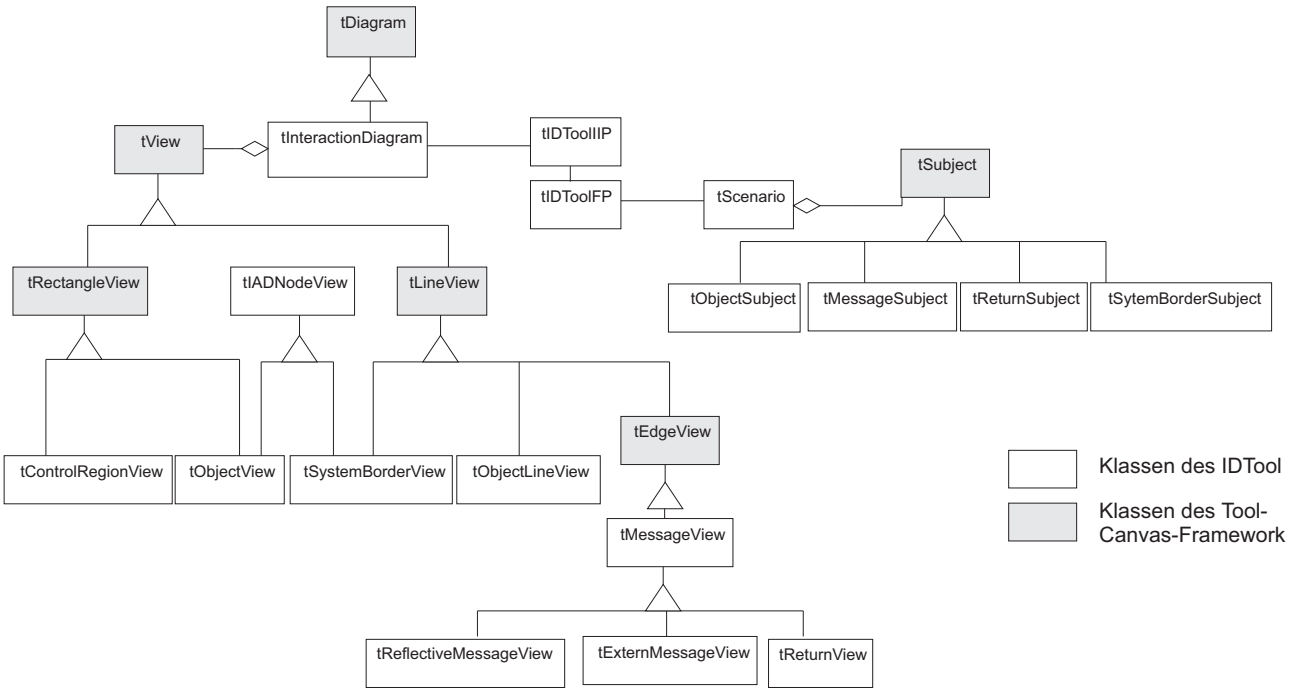


Abb. 26: Klassen des IDTool und des Tool-Canvas-Framework

handelt es sich um die Angaben über Objekt- und Methodennamen und Aufrufparameter. In den *Subjects* wird nur der Teil der Informationen gespeichert, der für das IDTool wirklich wichtig ist; zusätzliche Informationen muß das IDTool aus dem Programmodell erfragen. Zu diesem Zweck liefert das Programmodell mit jedem seiner Elemente eine eindeutige Identifizierungsnummer (object identifier, OID), die das IDTool bei späteren Anfragen angibt.[4] Die Klasse `tScenario` verwaltet alle *Subjects*; in ihr ist auch durch die Reihenfolge der `tMessageSubject-` und der `tReturnSubject`-Objekte die zeitliche Abfolge der Methodenaufrufe und -rücksprünge gespeichert. Alle *Subject*- und *View*-Klassen sowie `tScenario` und `tInteractionDiagram` zählen zu den Materialklassen des IDTool. Abbildung 26 gibt einen Überblick über die Klassen.

Die Bearbeitung des Diagramms nimmt der Benutzer mit einer Reihe von Nachfahren der Klasse `tTool` aus dem Tool-Canvas-Framework vor. Die Klassen stammen teilweise aus dem Framework (z.B. `tInputLabelTool` zur Eingabe von Beschriftungen), teilweise wurden sie speziell für das IDTool implementiert (z.B. `tNewObjectTool` und `tMoveMessagesTool`, deren Namen für sich sprechen). Die Manipulatoren, die die Wirkung einer Benutzeraktion anzeigen, bevor diese Aktion beendet ist, konnten alle aus dem Tool-Canvas-Framework übernommen werden; zusätzliche Manipulatoren waren nicht notwendig.

Die Veränderungen, die aufgrund einer vom Benutzer durchgeführten Aktion vorgenommen werden müssen, sind in einer Reihe von Kommandoobjekten (Nachfahren von `tCommand`) gekapselt. Die Erstellung neuer Kommandoklassen ist bei vielen Aktionen allein aus dem Grund erforderlich, daß im Kommandoobjekt auch Informationen und Anweisungen für ein eventuelles Rückgängigmachen der Aktion gespeichert werden müssen und sich dafür kein einheitliches Vorgehen anwenden läßt.

7.6.2.2 Bewertung des Tool-Canvas-Framework

Bei der Programmierung des IDTool hat sich das Tool-Canvas-Framework bewährt (immerhin wurden Programm und Framework zusammen entworfen, und die Entwicklung des Frameworks erfolgte im wesentlichen nach den Anforderungen des IDTool). Gleichzeitig wurde das Framework auch in einem einfachen Zeichenenditor getestet. Nach den Erfahrungen bei der Implementierung des IDTool zeigt sich:

- Die Konzepte, nach denen *Tools*, Manipulatoren und Kommandos zusammenarbeiten, haben sich gut bewährt. *Tools* und Manipulatoren sind gut miteinander kombinierbar, und mit zunehmender Reife des Frameworks lassen sich zusätzliche Interaktionsmöglichkeiten unter Verwendung der vorhanden *Tool*- bzw. Manipulator-Klassen immer leichter realisieren. Es zeigt sich aber auch, daß die Kommandos dem Programmierer einige Disziplin und guten Stil abverlangen, weil sie dazu verleiten, zu viel Funktionalität in den Kommando-Klassen zu implementieren und unbedarft auf die Materialien zuzugreifen, anstatt hier den sauberen Weg über Anfragen an die Funktionskomponente zu gehen.
- `tLayoutStrategy` und `tViewFactory` wurden für das IDTool doch nicht benutzt. Die beiden Klassen versprechen den theoretischen Nutzen, unabhängig von dem zugrunde liegenden Diagramm mit zwei einheitlichen Aufrufen, `CreateView` und `AssignPosition`, neue *Views* in das Diagramm einzufügen. Letztlich hat sich gezeigt, daß die

4 Andere Kriterien, etwa Speicheradressen oder Namen der repräsentierten Objekte, sind nicht eindeutig, während jede Identifizierungsnummer nur einmal während einer Programmausführung vergeben wird.

erforderlichen Operationen sehr stark von der Diagrammart (hier: vom Interaktionsdiagramm) abhängig sind, so daß z.B. in einer *ViewFactory*-Klasse sehr viel mehr Aufwand entsteht, als es für eine *Factory*-Klasse angemessen wäre. Stattdessen ließen sich die Operationen zum Erzeugen von *Views* sehr viel besser in entsprechenden Kommandos kapseln. Methoden, die das Layout bestimmen, sind beim IDTool in der Klasse `tInteraction-Diagram` implementiert worden.

- In der Beschreibung des IDTool wurden die Klassen `tEdgeView` und `tIADNodeView` erwähnt. Mit diesen Klassen wird ein Schritt in Richtung eines graphentheoretischen Ansatzes gegangen – eine sinnvolle Erweiterung des Tool-Canvas-Frameworks könnte darin bestehen, ein vollständiges Konzept von Knoten und Kanten zu realisieren (also etwa `tNodeView` und `tEdgeView`) und von `tDiagram` eine *Graph*-Klasse abzuleiten. Mit einem solchen Modell könnte die Erstellung weiterer Diagramme für VOOP2 zusätzlich erleichtert werden.

Die Verwendung des Tool-Canvas-Frameworks im IDTool entspricht der Intention des Frameworks; andere Werkzeuge, für die das Framework ineressant erscheint, haben sehr ähnliche Anforderungen, so daß man davon ausgehen kann, daß das Framework sich an anderen Stellen ähnlich gut bewähren wird wie in dieser Anwendung.

7.6.3 Funktionskomponente

Die Funktionskomponente des IDTool, *IDToolFP*, enthält Methoden, um mit Materialien umzugehen und um die ControlEngine zu steuern. Ein Ausschnitt des Interface der Klasse:

```
class tIDToolFP : public tFKBase
{
   public:
   tIDToolFP (tString pName);
   ~tIDToolFP(void);

   tScenario* GetScenario();
   tInteractionDiagram* GetDiagram();

   void NewMaterial();
   void LoadMaterial();
   void StoreMaterial();

   String& GetFilename(); // fuer Load-/StoreMaterial
   void SetFilename(const String& Filename);

   void LoadProgram(const tString& Filename);
   void StartExecution();
   void ContinueExecution();
   void InterruptExecution();

   protected:
   void NewMessageFunctor(const tObject*, const tFunction*,
      const tObject*, const tFunction*);
   void NewReturnFlowFunctor(const tObject*, const tFunction*,
      const tObject*, const tFunction*);
};
```

Die Methoden `NewMaterial`, `LoadMaterial` und `StoreMaterial` legen ein *Scenario*- und ein *InteractionDiagram*-Objekt neu an und speichern bzw. laden die beiden Objekte. Die Methoden `LoadProgram`, `StartExecution` etc. steuern die ControlEngine. Die beiden *Functor*-Objekte am Schluß der Deklaration beschreiben das Ziel von Callback-Aufrufen, die von der ControlEngine beim Eintreten und beim Verlassen von Methoden ausgehen.

7.6.4 Portabilität

Das IDTool ist portabel, z.Zt. läuft es unter Unix (Solaris 2.6 mit Motif 1.2) und Microsoft Windows (NT oder 95). Die Portabilität wird durch die GUI-Bibliothek wxWindows gewährleistet, auf der auch das Tool-Canvas-Framework basiert. Da bisher nicht alle Teile des BibV30-Frameworks von Unix auf MS Windows portiert worden sind, werden beim Compilieren des IDTools für NT einige abstrakte Oberklassen, die aus BibV30 stammen, ignoriert; weitere Einschränkungen für die Architektur oder Funktionalität des IDTool entstehen dabei nicht. Zu den relevanten BibV30-Teilen, die für das IDTool auf MS Windows übertragen wurden, gehören Containerklassen und Klassen zur Serialisierung von Objekten, die es ermöglichen, Objekte persistent in Dateien zu speichern und später wieder zu laden.

Weil ControlEngine und Programmodell (bisher) nur für Unix implementiert wurden, steht das IDTool unter MS Windows nicht als Visualisierungswerkzeug, sondern nur als Diagrammeditor zur Verfügung. Auch unter Unix kann das IDTool als reiner Editor compiliert werden. Bei allen Varianten (Unix oder Windows, Visualierungs- oder ausschließlich Editierwerkzeug) ist der Quelltext derselbe, dem Compiler müssen vor der Übersetzung lediglich die entsprechenden Parameter mitgeteilt werden.

7.7 Integration von Elementen 2. Ordnung

Bisher sind im IDTool nur Elemente 1. Ordnung realisiert. In diesem Abschnitt werde ich anhand eines Beispiels diskutieren, wie Elemente 2. Ordnung im IDTool berücksichtigt werden könnten. Diese Frage betrifft sowohl die Interaktion als auch die Auswirkungen auf die unterschiedlichen Materialien, mit denen das Werkzeug IDTool umgeht.

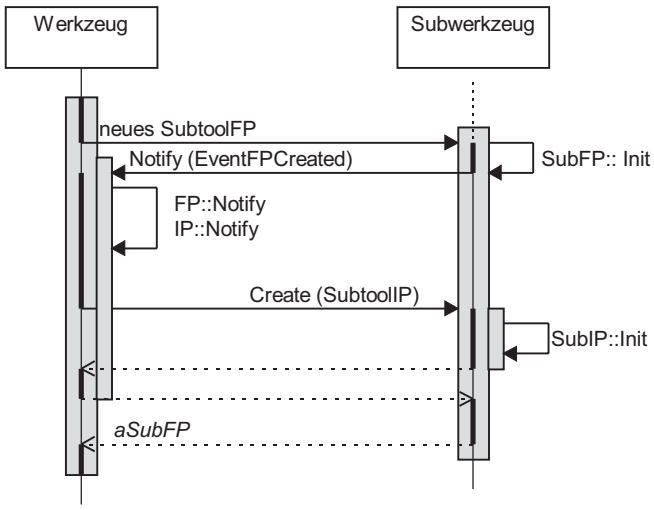


Abb. 27: Interaktionsdiagramm mit zusammengefaßten Objekten

7.7.1 Beispiel

7.7.1.1 Gruppierung von Objekten

In [Lilienthal und Strunk 96] wird das Beispiel eines erweiterten Interaktionsdiagramms vorgestellt, in dem eine Interaktionssequenz zwischen vier Objekten zu sehen ist (Abbildung 14 auf S. 65). Bei den Objekten handelt es sich um je zwei zusammengehörende Funktions- und Interaktionskomponenten. Eine vereinfachte Darstellung derselben Sequenz zeigt Abbildung 27: Für jedes Paar von Funktions- und Interaktionskomponente wird nur noch eine Komponente angezeigt. Zu sehen ist also nicht mehr ein Objekt im programmiersprachlichen Sinn, sondern eine Einheit auf Entwurfsebene. Darüber hinaus erscheinen in der zweiten Abbildung nicht mehr alle Methodenaufrufe, die innerhalb eines Objekts (nach dem erweiterten Verständnis) erfolgen. Der Nutzen der vereinfachten Darstellung liegt in der verbesserten Verständlichkeit[5]. Die Autoren bezeichnen die zusammengefaßten Objekte als „cluster of objects“. Ich werde hier den Begriff *Objektgruppierung* benutzen.

Eine Objektgruppierung ist ein Element 2. Ordnung (vgl. Definition auf S. 35). Der Begriff ist denkbar allgemein gewählt; über Sinn oder Inhalt der Gruppierung sagt er noch nichts aus. Im Unterschied dazu könnten Spezialisierungen von Objektgruppierungen angegeben werden, die eine bestimmte Semantik beinhalten: Die auf S. 36 genannten Beispiele für Elemente 2. Ordnung „Klassenbibliothek“ und „Prozeß“ sind eine solche Spezialisierung, und die in Abbildung 27 vorgenommenen Gruppierungen könnte man mit dem Namen „Werkzeug“ belegen. Auf diese Art erhält man einen fachlichen Entwurf, wie ihn Abbildung 28 zeigt.

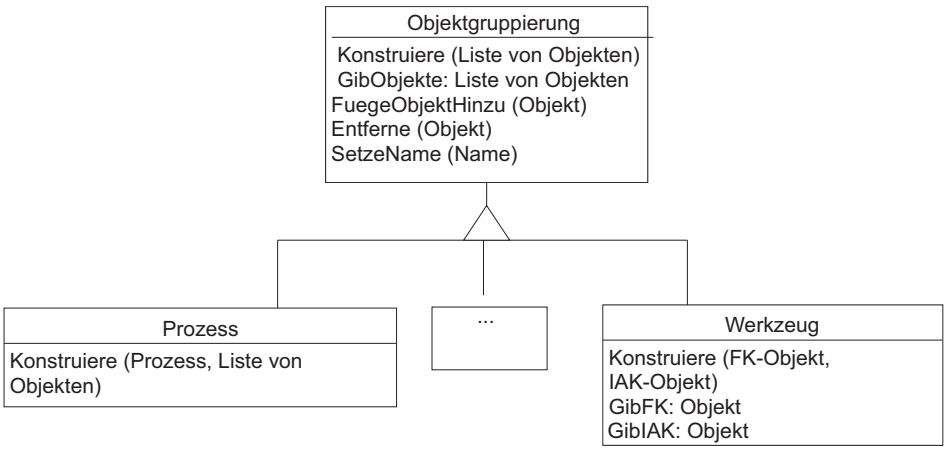


Abb. 28: Fachliche Hierarchie von Objektgruppierungen

7.7.1.2 Handhabungsvision

Aus der Sicht des Benutzers stellt sich nun die Frage, wie mehrere Objekte zu einer Gruppierung zusammengefaßt werden können. Die folgende Handhabungsvision gibt einen möglichen Weg an:

5 "... immensely improved the students' understanding of the interaction diagrams.The use of enhanced interaction diagrams in industrial projects achieved similar results."

„Der Benutzer hat mit dem IDTool eine Programmausführung visualisiert. Es ist ein Diagramm entstanden, das je ein Objekt der Klassen tSubFP und tSubIP enthält; die beiden Objekte sind mit den Klassennamen beschriftet.

Der Benutzer zieht per Drag-and-Drop das tSubFP-Objekt mit der Maus über das Diagramm, bis sich der Mauszeiger über dem tSubIP-Objekt befindet, und läßt das tSubFP-Objekt dort 'fallen'. Das tSubFP-Objekt ist nun aus dem Diagramm verschwunden. An der Stelle des tSubIP-Objekts erscheint ein Objekt, das mit tSubIP überschrieben ist und mit dem alle Pfeile verbunden sind, die zuvor zwischen einem der beiden Objekte und anderen Instanzen im Diagramm verliefen. Damit ist eine Objektgruppierung entstanden. Um nun aus der Gruppierung ein 'Werkzeug' zu machen, selektiert der Benutzer die Gruppierung durch einfachen Mausklick und wählt aus dem Menü das Kommando 'In Werkzeug umwandeln'. Daraufhin erscheint das abgebildete Werkzeugfenster (Abb. 29), in dem der Benutzer angibt, welches der beiden Objekte die Funktionskomponente des Werkzeugs darstellt und welches die Interaktionskomponente. Anschließend schließt er den Dialog und ändert den Namen der Gruppierung von 'tSubIP' auf 'tSubTool', indem er den Namen im Diagramm anklickt und dort editiert."

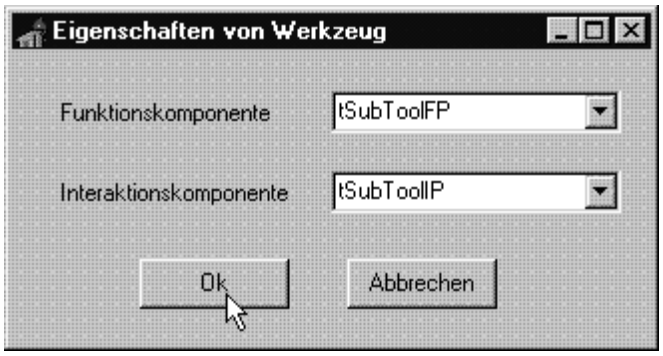


Abb. 29: Fenster für „Werkzeug"-Eigenschaften

Die Gruppierung soll selbstverständlich auch wieder rückgängig gemacht werden können (indem der Benutzer etwa die Objektgruppierung per Mausklick selektiert und dann im Menü einen Eintrag „Gruppierung auflösen" wählt).

Es stellt sich nun die Frage, wie der Entwurf des IDTool erweitert werden muß, um die geschilderte Vision umzusetzen. Als erstes werde ich auf die nötigen Anpassungen eingehen, die für die Benutzerinteraktion nötig sind.

7.7.1.3 Anpassung auf IAK-Seite

In der Handhabungsvision wird die Aktion per Drag-und-Drop ausgelöst. Was passiert bisher, wenn mit der Maus eine Drag-Aktion durchgeführt wird? Für das Drag-Ereignis haben sich beim Programmstart mehrere Nachfahren der Klasse `tTool` (aus dem Tool-Canvas-Framework) angemeldet. Darunter ist das *MoveObjectTool*, das aktiviert wird, sobald ein Objekt mit der Maus über den Bildschirm gezogen wird. Die Bewegung des Objekts zeigt ein *DragViewManipulator* an. Hier ergibt sich nach der Vision das Problem, das bei Beginn der Drag-Aktion noch nicht bekannt ist, ob das Objekt an einer freien Stelle des Diagramms losgelassen (und damit wie bisher verschoben) wird, oder ob die Aktion über einem anderen Objekt endet und somit zu einer Objektgruppierung führen soll. Bei Beginn der Aktion darf also nicht mehr das bisherige *MoveObjectTool* aufgerufen werden, sondern ein allgemeineres *DragObjectTool*. Dies verhält sich zunächst genauso und erzeugt einen *DragViewManipulator*, muß aber, wenn der Drag-Vorgang beendet ist, feststellen, ob ein Verschieben oder ein Gruppieren von Objekten durchgeführt werden muß. Im ersten Fall erzeugt es, wie das *MoveObjectTool*,

ein Kommandoobjekt vom Typ `tMoveViewCommand`. Andernfalls muß es ein Kommando eines neuen Typs `tGroupViewCommand` erzeugen.

Die Klasse `tGroupViewCommand` wäre dann dafür zuständig, daß die Objektgruppierung in den Materialien, auf denen das Werkzeug arbeitet, berücksichtigt wird; daher werde ich im folgenden die Auswirkungen auf die Materialien *Subjects* (in `tScenario`), *Views* (in `tInteractionDiagram` verwaltet) und das Metamodell erörtern.

Wenn über das Menü schließlich die Umwandlung der Objektgruppierung in ein „Werkzeug" erfolgt, wird eine Kommando-Klasse `tCreateToolCommand` benötigt. Sie muß dafür sorgen, daß das Werkzeug aus Abbildung 29 erscheint, und muß anschließend die Umwandlung von „Objektgruppierung" nach „Werkzeug" an die Materialien weitergeben.

7.7.1.4 Subjects

In Abbildung 28 wurde bereits eine fachliche Hierarchie von Objektgruppierungen skizziert. Die fachlichen Klassen müssen als *Subject*-Klassen im IDTool implementiert werden, d.h., die abstrakte Klasse `tSubject` des Tool-Canvas-Frameworks wird zu den Klassen `tObjectClusterSubject`, `tProcessSubject`, `tToolSubject` usw. erweitert. Die Klasse `tToolSubject` wird etwa so aussehen:

```
class tToolSubject: public tObjectClusterSubject
{
   public:
   tToolSubject(tObjectSubject* pFP, tObjectSubject* pIP);
   tObjectSubject* GetFP();
   tObjectSubject* GetIP();

   protected:
   tObjectSubject *pIP, *pFP;
};
```

Zusätzlich zu den allgemeinen Eigenschaften, die Objektgruppierungen besitzen und die in der Oberklasse `tObjectClusterSubject` definiert werden, verweist `tToolSubject` also explizit auf zwei *Subjects*, die die Funktions- und die Interaktionskomponente des Werkzeugs, für das das *ToolSubject* steht, beschreiben. Diese beiden *Subjects* existierten bereits und bleiben weiter bestehen, auch wenn sie im Diagramm keine Repräsentationen (*Views*) mehr haben. Das `tScenario`-Element, das als Container für die *Subjects* dient, enthält nun also neben den beiden bisherigen *Subjects* ein zusätzliches Element vom Typ `tToolSubject`.

7.7.1.5 Views

Nachdem nun neue *Subject*-Klassen für die Objektgruppierungen eingeführt worden sind, liegt es nahe, dazu korrespondierend auch neue *View*-Klassen vorzusehen. Das ist aber nicht unbedingt erforderlich, denn der angegebenen Vision ist nicht zu entnehmen, daß eine Objektgruppierung anders aussieht als die bisher benutzten Objekte vom Typ `tObjectView`. Wenn neben dem Aussehen auch die Handhabung im Sinne von direkter Manipulation[6] übereinstimmt, kann die bereits bestehende *View*-Klasse benutzt werden.

[6] Wenn eine Objektgruppierung wie oben beschrieben mittels einer Menüanweisung aufgelöst wird, handelt es sich nicht um direkte Manipulation.

Wenn Objektgruppierungen mit einem eigenen Symbol versehen werden, wie es in Abschnitt 7.2.2 vorgeschlagen wurde (vgl. Abbildung 18a), kann von der allgemeinen Klasse `tObject-View` eine entsprechende Spezialisierung gebildet werden. Da jedes *View*-Objekt andere *Views* aggregieren kann, besteht aber auch die Möglichkeit, es bei der existierenden Klasse zu belassen und den erzeugten Instanzen durch hinzugefügte *Views* ein anderes Aussehen zu geben.

Die *View*-Objekte, die in der Vision zusammengefaßt werden, können aus dem Interaktionsdiagramm entfernt und gelöscht werden; wenn die Gruppierung später wieder aufgelöst werden sollte, können neue *Views* angelegt werden – die notwendigen Informationen bleiben in den *Subjects* erhalten. Aus dem *InteractionDiagram* werden durch das *GroupViewCommand* also zwei *Views* entfernt, und ein neues *View*-Objekt wird eingefügt. Jedes Objekt der *View*-Hierarchie verweist auf ein *Subject*, und so hält das neue *View*-Objekt einen Verweis auf das ebenfalls neu angelegte *ToolSubject*.

7.7.1.6 Metamodell

Solange die Objektgruppierung nur innerhalb des IDTool bekannt sein soll, sind *Subjects* und *Views* die einzigen Materialien, die angepaßt werden müssen. Wenn VOOP2 aber mehrere Werkzeuge umfaßt und die Gruppierung, die der Benutzer im IDTool vornimmt, auch in den anderen Werkzeugen wirksam werden soll, muß diese Information zusätzlich im Programmodell bekannt gemacht werden.

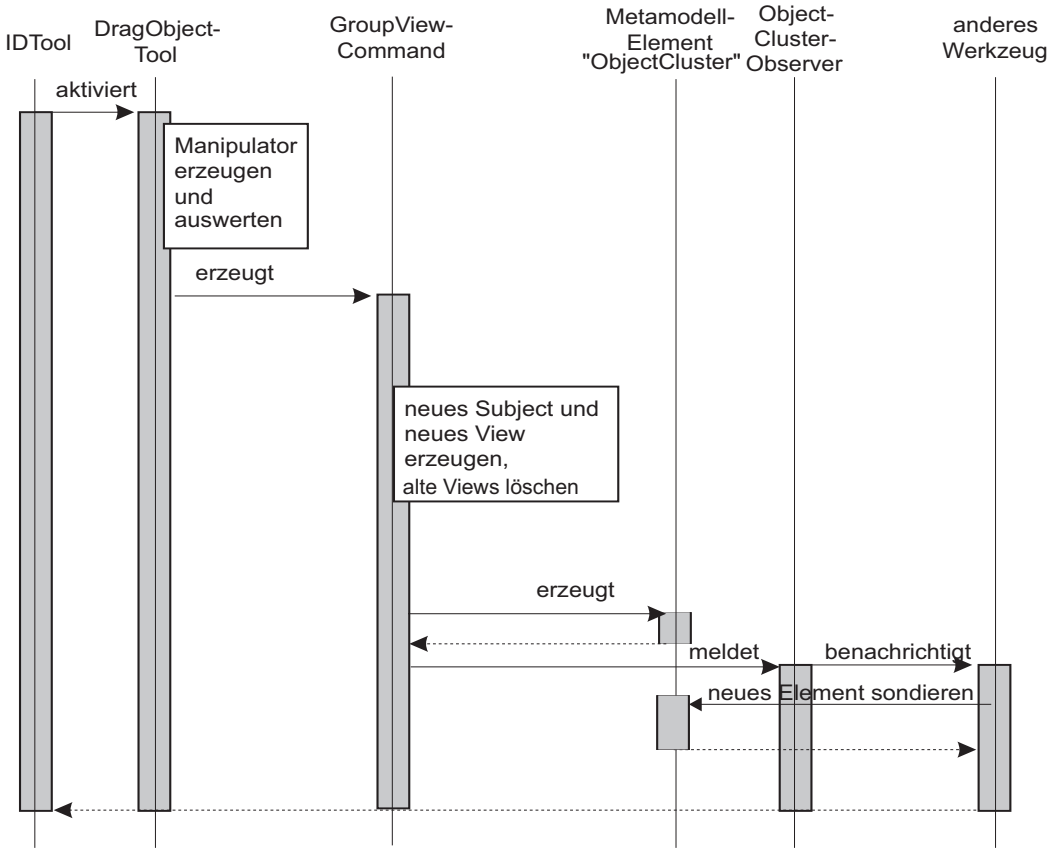


Abb. 30: Erzeugen eines Elements 2. Ordnung („Tool“)

Die erste Konsequenz ist, daß das Metamodell um eine oder mehrere Klassen erweitert werden muß, die die Objektgruppierungen darstellen; hier kommen also Elemente 2. Ordnung im Metamodell hinzu. Das geschieht vom Prinzip her analog zu den zusätzlichen *Subject*-Klassen. Die neuen Typen verweisen auf andere Elemente des Programmodells, die zur Metamodell-Klasse `tObject` (S. 61) gehören. Während aber die *Subjects* relativ schmal ausfallen, muß in den neuen Metamodell-Klassen jede Anforderung berücksichtigt werden, die von einem der Werkzeuge benutzt wird; u.U. enthalten die zusätzlichen Metamodell-Klassen damit deutlich größere Schnittstellen als die entsprechenden *Subject*-Klassen des IDTool.

Die zusätzlichen Elemente, die nun im Programmodell auftreten können, bringen eine zweite Konsequenz mit sich: Es muß eine Instanz geben, die die Werkzeuge über neue Elemente 2. Ordnung informiert. Für die Elemente 1. Ordnung übernimmt die ControlEngine diese Aufgabe, also ist es nur konsequent, der ControlEngine auch die Verantwortung für das Bekanntmachen der Elemente 2. Ordnung zu überlassen.

So wie die ControlEngine bisher über zwei Schnittstellenklassen `tClassObserver` (Abschnitt 6.3.3) und `tFunctionObserver` (Abschnitt 6.3.4) verfügt, kommt nun noch eine Schnittstelle wie `tObjectClusterObserver` hinzu. Alle Werkzeuge, die das Konzept der Objektgruppierungen beherrschen, melden sich bei dieser Klasse an, übergeben eine Callback-Methode und werden fortan benachrichtigt, wenn eine Objektgruppierung erzeugt oder aufgelöst wird. `tObjectClusterObserver` benötigt weiterhin eine Methode, über die ihm das IDTool mitteilt, daß eine neue Gruppierung bekanntgegeben werden muß.

Den schematisierten Ablauf für das Anlegen des Elements 2. Ordnung zeigt das Interaktionsdiagramm in Abbildung 30.

7.7.2 Fazit

Das Beispiel der Handhabungsvision zeigt, daß sich Elemente 2. Ordnung in das IDTool und in die Gesamtarchitektur von VOOP2 integrieren lassen. Der Aufwand dafür bleibt vertretbar; es fällt jedoch auf, daß für ein Element 2. Ordnung bis zu drei neue Materialklassen erstellt werden, nämlich für *Subject* und *View* im IDTool und eine weitere im Metamodell. Für jedes weitere Werkzeug können zusätzliche Spezialisierungen von `tSubject` und `tView` nötig werden. Alle diese Klassen benötigen zwar nur wenig Code, aber ihre Anzahl wird relativ hoch, was ein gewisses Manko darstellt.

Weiterhin muß die ControlEngine um eine Klasse erweitert werden, die als Schnittstelle für die neuen Elemente im Programmodell dient. Hier ist es vorstellbar, daß es nur eine einzige Schnittstelle gibt, über die die Elemente 2. Ordnung bekanntgemacht werden und in der der jeweilige Elementtyp als Parameter angegeben wird.

Damit kann die ControlEngine für weitere hinzukommende Elementtypen unverändert bleiben, und es muß, abgesehen von den Werkzeugen, nur noch das Metamodell erweitert werden. Wenn man dann den Mehrprozeßentwurf aus Abschnitt 6.4 nutzt, kann die Erweiterung des Metamodells dynamisch erfolgen. Das bedeutet, daß für spätere Werkzeuge zusätzliche Elemente 2. Ordnung eingeführt werden können und im Metamodell bekannt sind, daß aber weder die ControlEngine noch das Metamodell erneut compiliert werden müssen. Der Entwurf im Mehrprozeßraum kann auf diese Art seine Flexibilität und Erweiterbarkeit gut ausspielen.

Es bleiben aber auch einige offene Probleme. Für das genannte Beispiel sind etwa folgende Fragen zu stellen:

- Was passiert, wenn eine Objektgruppierung gebildet wird und im weiteren Verlauf der Programmausführung einige der gruppierten Objekte gelöscht werden? Soll der Benutzer darüber informiert werden? Wenn ja, wie? Wie wird das in der Gruppierung sichtbar?
- Wenn ein Diagramm vom Benutzer editiert und dabei ein Methodenaufruf an eine Objektgruppierung eingefügt wird: auf welches Objekt bezieht sich der Aufruf? Die Frage stellt sich spätestens, wenn die Gruppierung aufgelöst wird.

Ein grundsätzliches Problem wird immer darin bestehen, die Menge der Elemente 1. und 2. Ordnung konsistent zu halten. Diese Anforderung ist mit dem vorhandenen Wissen und der vorliegenden Architektur immerhin zu bewältigen; wie Benutzer mit Elementen zweiter Ordnung umgehen und wie ein Werkzeug für die damit erschlossene Abstraktionsebene Unterstützung leistet, sind tiefergehende, konzeptuelle Fragen, die in dieser Arbeit unbeantwortet bleiben.

7.8 Zusammenfassung

Das Visualisierungssytem VOOP2 ist so konzipiert, daß mehrere Werkzeuge unterschiedliche Diagramme aus dem Programmodell erzeugen. Der erste Prototyp eines solchen Werkzeugs ist das IDTool, das in diesem Kapitel beschrieben wurde. Das IDTool erzeugt Visualisierungen in Form von Interaktionsdiagrammen und fungiert gleichzeitig als Diagrammeditor. Das Werkzeug läuft unter Unix und Microsoft Windows. Da ControlEngine und Programmodell derzeit nur unter Unix zur Verfügung stehen, beschränkt sich die Windows-Version auf reine Editor-Funktionalität.

Zu den Vorarbeiten, die für das IDTool nötig waren, gehörte die Definition einer Notation für Interaktionsdiagramme. Die Notation, die ich in Abschnitt 7.2 vorgestellt habe, entspricht der allgemeinen Form von Interaktionsdiagrammen und nimmt gleichzeitig einige besondere Merkmale der in der Literatur vorhandenen Varianten auf.

Das IDTool arbeitet auf mehreren Arten von Materialien: Zunächst einmal nutzt es das allgemein verfügbare Programmodell. Darüber hinaus benutzt es *Subjects* und *Views*. Die *Subjects* beinhalten diejenigen Anteile, die die Semantik eines Diagramms beeinflussen, während die *Views* grafische Materialeigenschaften ausdrücken.

Die Trennung von *Subjects* und *Views* ist Bestandteil eines Frameworks, das ich erstellt und im Abschnitt 7.5 unter dem Namen *Tool-Canvas-Framework* vorgestellt habe. Es stellt eine Architektur und eine Anzahl von Klassen zur Verfügung, mit deren Hilfe sich Editoren für strukturierte Grafik realisieren lassen. Auf der Basis des Frameworks entstand das IDTool.

Die Diagramme, die mit dem IDTool beim derzeitigen Stand der Implementierung erstellt werden können, enthalten noch keine Elemente zweiter Ordnung. Wie eine solche Erweiterung erfolgen kann, habe ich im letzten Abschnitt dieses Kapitels anhand eines Beispiels beschrieben. Das Beispiel ging von einem Szenario aus, in dem ein Benutzer mehrere angezeigte Objekte interaktiv zu einer Komponente zusammenfaßt. Die notwendigen Änderungen bzw. Ergänzungen betreffen im ersten Schritt die Klassen innerhalb des Werkzeugs IDTool. Wenn die Aktion auch für andere Werkzeuge bekanntgemacht werden soll, müssen darüber hinaus ControlEngine und Metamodell erweitert werden.

Kapitel 8
Zusammenfassung und Ausblick

In dieser Arbeit wurde eine erweiterbare Architektur für Visualisierungssysteme entwickelt, um das Verhalten objektorientierter Programme zu veranschaulichen. Basis des Systems ist ein Programmodell, in dem der Zustand des untersuchten Programms nachgebildet wird. Ein Metamodell stellt die Typen zur Verfügung, aus deren Elementen das Programmodell besteht. Das Metamodell umfaßt zunächst nur Elemente 1. Ordnung, die der programmiersprachlichen Ebene entsprechen und Klassen, Objekte und Methoden repräsentieren. Darüber hinaus kann es um Elemente 2. Ordnung erweitert werden, mit denen auf einer höheren Abstraktionsebene Architekturkonzepte des untersuchten Programms modelliert werden können. Um das Programmodell zu erstellen und parallel zur Programmausführung zu aktualisieren, wurde ein Automat (die ControlEngine) vorgestellt.

Es erfolgte ein Überblick über grafische Notationen, mit denen Struktur und Verhalten von objektorientierten Programmen dargestellt werden. Ein Werkzeug, das auf die Architektur des Visualisierungssystems aufbaut und Visualisierungen in Form von Interaktionsdiagrammen erzeugt, ist das IDTool. Der Vorteil der Interaktionsdiagramme liegt gegenüber anderen Darstellungstechniken darin, daß das Verhalten von Objekten im Zeitverlauf abgebildet wird, ohne die Anzeige animieren zu müssen. Damit eignen sich die Diagramme auch für gedruckte Medien, z.B. für Dokumentationen von Anwendungen und Frameworks. Vor diesem Hintergrund wurde eine Notation für erweiterte Interaktionsdiagramme vorgestellt.

Um Diagrammen während der Visualisierung eine verständlichere Anordnung zu geben und sie für spätere Benutzung (z.B. in Dokumentationen) anpassen zu können, ist das IDTool nicht nur Visualisierungswerkzeug, sondern auch ein Diagrammeditor. Zu diesem Zweck wurde ein Grafik-Framework vorgestellt, mit dem die Entwicklung von derartigen Editoren erleichtert wird. Andere Visualisierungswerkzeuge, die noch entwickelt werden sollen, werden von dem Framework profitieren.

Behandelt wurde ebenfalls die Benutzerschnittstelle des IDTool und der Umgang mit diesem Werkzeug. Der Entwurf und die Implementierung des Werkzeugs wurden im Überblick vorgestellt. Schließlich folgte ein Beispiel, wie Elemente 2. Ordnung in der Benutzerschnittstelle des Werkzeugs berücksichtigt werden könnten und welche Erweiterungen dadurch in der Implementierung nötig würden.

Das in dieser Arbeit beschriebene Visualisierungssystem zeichnet sich durch mehrere Vorteile aus:

- Es ist keine Instrumentierung der Quelldateien und damit kein Recompilieren der untersuchten Programme erforderlich.

- Die Visualisierung erfolgt interaktiv, und das erzeugte Diagramm kann sowohl während als auch nach der Visualisierung editiert werden.
- Das System ist darauf ausgerichtet, um Elemente 2. Ordnung erweitert zu werden, so daß die Architektur untersuchter Programme adäquat dargestellt werden kann.

Erste Erfahrungen mit dem System zeigen bestehende Probleme auf. Diagramme werden schnell zu groß und damit zu unübersichtlich, und die Anzeige der Diagramme wird mit zunehmender Größe langsamer. Hier dürfte sich die Performanz der Anzeige spürbar verbessern, wenn die interne Verwaltung der Diagrammelemente effizienter erfolgt.

Um die Größe der Diagramme von vornherein zu beschränken, sollten zusätzliche Filter für die Informationserfassung zur Verfügung gestellt werden. Bisher ist nur eine Auswahl aus einer Liste aller Klassen möglich; statische Filter könnten vor Beginn der Visualisierung nach anderen Kriterien, z.B. nach Frameworks, selektieren, und dynamische Filter könnten während des Ablaufs verhindern, daß häufig auftretende, aber nicht interessierende Ereignisse im Diagramm erscheinen.

Unvollständig ist bisher die Implementierung des IDTool, da die Diagramme noch nicht alle vorgesehenen Darstellungselemente enthalten. Zur zukünftigen Arbeit wird daher die Weiterentwicklung dieses Werkzeugs zählen, dazu aber auch die Entwicklung zusätzlicher Werkzeuge, die mit anderen Darstellungsmitteln zusätzliche Sichten auf die Programmausführung bieten.

Viele offene Fragen ergeben sich in Zusammenhang mit den Elementen 2. Ordnung. Einige Beispiele für solche Elemente wurden in dieser Arbeit genannt; als nächstes sollte versucht werden, eine vollständige (falls dies überhaupt möglich ist) Hierarchie von Elementen 2. Ordnung aufzustellen, mit der sich die Architektur von beliebigen objektorientierten Programmen auf verschiedenen Ebenen modellieren läßt.

Wenn es eine umfassende Sammlung von Elementen 2. Ordnung gibt, bleibt das Problem, wie diese für ein konkretes Programm erkannt werden. Es ist zu untersuchen, ob ein Visualisierungssystem bestimmte Elemente automatisch erkennen kann. Sofern das möglich ist, sollte es geschehen; für andere Elemente könnte das System zumindest Vorschläge machen und die Entscheidung dem Benutzer überlassen.

Damit werden dann auch Werkzeuge benötigt, die die Konzepte der Elemente 2. Ordnung in passenden Diagrammen darstellen und so miteinander verknüpft sind, daß sich der Benutzer einfach durch eine Reihe verschiedener Ansichten bewegen kann. Er könnte so von einer übersichtlichen Gesamtdarstellung bis zu einzelnen Details gelangen oder umgekehrt bei einzelnen Objekten beginnen und sich zu den architektonischen Konzepten hinauf bewegen, in denen die Objekte stehen. Wenn solche Werkzeuge zur Verfügung stehen und die Potentiale der Softwarevisualisierung ausnutzen, sollte damit das Verstehen von großen Anwendungen und Frameworks erheblich erleichtert werden.

Anhang A
Literatur

[Alexander et al. 77] Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, Shlomo Angel: A Pattern Language. Oxford University Press, Oxford, 1977

[Anderson 95] Eddie Anderson: Dynamic Visualization of Object Programs written in C++. Objective Software Technology Ltd., 1995

[Baecker et al. 97] Ron Baecker, Chris DiGiano, Aaron Marcus: Software Visualization for Debugging. In: Communications of the ACM, Jg. 40 Nr. 4, April 1997

[Bäumer et al. 95] Dirk Bäumer, Reinhard Budde, Karl-Heinz Sylla, Guido Gryczan, Heinz Züllighoven: Objektorientierte Konstruktion von Software-Werkzeugen und -Materialien. In: Informatik-Spektrum, Jg. 18, Nr. 4, Aug. 1995

[Bertin 81] J. Bertin: Graphics and Graphic Information Processing. Walter de Gruyter, Berlin, 1981

[Biggerstaff et al. 94] 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

[Bischofberger 95] Walter R. Bischofberger: Frameworkbasierte Softwareentwicklung. Proceedings of the OOP Munich, 1995

[Blackwell 96] Alan F. Blackwell: Metacognitive Theories of Visual Programming: What do we think we are doing? In: Proceedings IEEE Symposium on Visual Languages, S. 240-246, 1996

[Booch 94] Grady Booch: Object-Oriented Analysis and Design with Applications. 2nd ed., Benjamin Cummings, Redwood City, Calif., 1994

[Bruegge et al. 93] Bernd Bruegge, Tim Gottschalk, Bin Luo: A Framework for Dynamic Program Analyzers. In: Proceedings of OOPSLA ‘93, ACM Press, New York, 1993

[Burnett et al. 95] Margaret Burnett, Adele Goldberg, Ted Lewis (Hrsg.): Visual Object-Oriented Programming. Prentice Hall, Englewood Cliffs, New Jersey, 1995

[Cheng und Gray 92] G. Cheng, N. A. B. Gray: A Program Visualisation Tool. In: Proceedings of TOOLS Pacific ´92, 1992

[Churchill 95] Steve Churchill: Programming Fresco. A hands-on tutorial for the Fresco user interface system. Fujitsu Ltd. 1995

[Coleman et al. 94] Derek Coleman, Patrick Arnold, Stephanie Bodoff, Chris Dollin, Helena Gilchrist, Fiona Hayes: Object-Oriented Development: The Fusion Method. Prentice Hall, Englewood Cliffs, New Jersey, 1994

[Coplien 92] James O. Coplien: Advanced C++. Addison-Wesley, Reading, Mass., 1992

[Corbi 89] Thomas A. Corbi: Program Understanding: Challenge for the 1990's. IBM Research Report, 1989

[Cunningham und Beck 86] Ward Cunningham, Kent Beck: A Diagram for Object-Oriented Programs. In: Object-Oriented Programming Systems, Languages, and Applications Conference 1986, ACM Press, Washington, DC, 1986

[De Pauw et al. 93] 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 Press, Washington, DC, 1993

[De Pauw et al. 94] Wim De Pauw, Doug Kimelman, John Vlissides: Modeling Object-Oriented Program Execution. In: Proceedings of ECOOP, Springer Verlag, Berlin/Heidelberg, 1994

[Doebele 97] Markus Doebele: Erkenntnis als Voraussetzung erfolgreichen Wandels. In: OBJEKTspektrum 4/97, S. 41-51, 1997

[Ellis und Stroustrup 90] Margaret A.Ellis, Bjarne Stroustrup: The Annotated C++ Reference Manual. Addison-Wesley, Reading, Mass., 1990

[Floyd 95] Christiane Floyd: Software Engineering. Kritik und Perspektiven. In: Jürgen Friedrich, Thomas Herrmann, Max Peschek, Arno Rolf (Hrsg.): Informatik und Gesellschaft. Spektrum Akad. Verlag, Heidelberg, 1995

[Floyd und Züllighoven 97] Christiane Floyd, Heinz Züllighoven: Softwaretechnik. In: Peter Rechenberg, Gustav Pomberger (Hrsg.): Informatik-Handbuch. Hanser, München/Wien, 1997

[Fowler 97] Martin Fowler: Analysis Patterns. Addison-Wesley, Reading, Mass., 1997

[Fröhlich und Stranzinger 97] Joachim Hans Fröhlich, Thomas Stranzinger: Extracting Object Architectures from C++-Programs. In: Proceeding of PLDI, 1997

[Gamma et al. 95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: Design Patterns: Elements of Reuseable Object-Oriented Software. Addison-Wesley, Reading, Mass., 1995

[Goldberg 84] Adele Goldberg: Smalltalk-80: The Interactive Programming Environment. Addison-Wesley, Reading, Mass., 1984

[Goldberg et al. 95] Adele Goldberg, Margaret Burnett, Ted Lewis: What is Visual Object-Oriented Programming? In: [Burnett et al. 95], S. 11 ff., 1995

[Hamer und Friedrich 96] Keno Hamer, Stephen Friedrich: Visualisierung des Objektverhaltens in C++-Programmen. Studienarbeit, Universität Hamburg, 1996

[Ingalls 81] Daniel H. H. Ingalls: Design Principles behind Smalltalk. In: Byte, Jg. 6 Nr. 8, 1981

[Jacobson et al. 92] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, Gunnar Overgaard: Object-Oriented Software Engineering - A Use Case Driven Approach. Addison-Wesley, Workingham, England, 1992

[Jerding et al. 96] Dean F. Jerding, John T. Stasko, Thomas Ball: Visualizing Message Patterns in Object-Oriented Program Executions. In: Proceeding of OOPSLA '96. ACM Press, New York, 1996

[Jerding und Stasko 94] Dean F. Jerding, John T. Stasko: Using Visualization to Foster Object-Oriented Program Understanding. Technical Report, GVU Georgia Tech, 1994

[Jerding und Stasko 95] Dean F. Jerding, John T. Stasko: The Information Mural: A Technique for Displaying and Navigating Large Information Systems. In: IEEE Proceedings Information Visualization, Atlanta, 1995

[Kilberth et al. 94] Klaus Kilberth, Guido Gryczan, Heinz Züllighoven: Objektorientierte Anwendungsentwicklung. 2. Aufl., Vieweg, Braunschweig/Wiesbaden, 1994

[Kleyn und Gingrich 88] Michael F. Kleyn, Paul C. Gingrich: Graphtrace - Understanding Object-Oriented Systems Using Concurrently Animated Views. In: Proceedings of OOPSLA ´88. ACM Press, New York, 1988

[Koskimies und Mössenböck 96] Kai Koskimies, Hanspeter Mössenböck: Scene: Using Scenario Diagrams and Active Text for Illustrating Object-Oriented Programs. Proceedings of International Conference on Software Engeneering (ICSE'96), Berlin, 1996

[Laffra und Malhotra 94] Chris Laffra, Ashok Malhotra: HotWire - A Visual Debugger for C++. In: Proceedings of USENIX C++ Technical Conference, S. 109-122. Usenix Assn, Berkley, Calif., 1994

[Laffra 97] Chris Laffra: Advanced Java. Prentice Hall, Englewood Cliffs, New Jersey, 1997

[Lange und Nakamura 95] Danny B. Lange, Yuichi Nakamura: Interactive Visualization of Design Patterns Can Help in Framework Understanding. In: Proceedings of OOPSLA ’95, S. 39-54, ACM Press, New York, 1995

[Lange und Nakamura 97] Danny B. Lange, Yuichi Nakamura: Object-Oriented Program Tracing and Visualization. In: IEEE Computer, Jg. 30 Nr. 5, S. 63-70, Mai 1997

[Larkin und Simon 87] Jill H. Larkin, Herbert A. Simon: Why a Diagram is (Sometimes) Worth Ten Thousand Words. In: Cognitive Science, Jg. 11 Nr. 1, S. 65-99, Januar 1987

[Lilienthal und Strunk 96] Carola Lilienthal, Wolfgang Strunk: Documenting Frameworks by Visualizing Dynamics. Proccedings of Technology of Object-Oriented Languages and Systems (TOOLS USA), 1996

[Lilienthal 97] Carola Lilienthal: Dokumentation zur BibV30.
URL: http://swt-www.informatik.uni-hamburg.de/Software/BibV30

[Linton et al. 92] Mark A. Linton, Paul R. Clader, John A. Interrante, Steven Tang, John Vlissides. InterViews Reference Manual. Ausgabe 3.1. Computer Solutions Laboratory, Stanford, 1992

[Lippman 96] Stanley B. Lippman: Inside the C Plusplus Object Model. Addison-Wesley, Reading, Mass., 1996

[Littman et al. 86] David C. Littman, Jeannine Pinto, Stanley Letovsky, and Elliot Soloway : Mental Models and Software Maintenance. In: Elliot Soloway, Sitharama Iyengar: Empirical Studies Of Programmers. First Workshop on Empirical Studies of Programmers, Washington D.C., 1986

[Menapace et al. 93] Julia Menapace, Jim Kingdon, David MacKenzie: The „stabs“ Debug Format. Cygnus Support, 1993

[Meyer 88] Bertrand Meyer: Object-Oriented Software Construction. Prentice Hall, Englewood Cliffs, New Jersey, 1988

[Myers 90] Brad A. Myers: Taxonomies of Visual Programming and Program Visualization. In: Journal of Visual Languages and Computing, Jg. 1 Nr. 1, S. 97-123, 1990

[Müller 97] Bernd Müller: Reengineering – eine Einführung. Teubner, Stuttgart, 1997

[Ortigosa und Campo 96] Alvaro Ortigosa, Marcelo R. Campo: Architectural-Driven Analysis of C++ Program Behavior Using Meta-Objects. In: Proccedings of the 20th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS USA 96), 1996

[Petre et al. 97] M. Petre, A. F. Blackwell and T. R. G. Green: Cognitive Questions in Software Visualisation. In: [Stasko et al. 97]

[Price et al. 93] Blaine A. Price, Ronald M. Baecker, Ian S. Small: A Principled Taxonomy of Software Visualization. In: Journal of Visual Languages and Computing, Jg. 4, S. 211-266, 1993

[Rational 97] Grady Booch, Ivar Jacobson, James Rumbaugh: The Unified Modeling Language for Object-Oriented Development. Version 1.1, Rational Software Corp., 1997

[Riehle und Züllighoven 94] Dirk Riehle, Heinz Züllighoven: Späte Erzeugung. 39. Internationales Wissenschaftliches Kolloquium. Thüringen: Technische Universität Ilmenau (Thüringen), 1994.

[Rogers 97] Gregory F. Rogers: Framework-based Software Development in C++. Prentice Hall, Englewood Cliffs, New Jersey, 1997

[Roman und Cox 93] Gruia-Catalin Roman, Kenneth C. Cox: A Taxonomy of Program Visualization Systems. In: IEEE Computer, Jg. 26, Nr. 12, S. 11-24, 1993

[Rumbaugh et al. 91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, William Lorensen: Object-Oriented Modeling and Design. Prentice Hall, Englewood Cliffs, New Jersey, 1991

[Schäfer 94] Steffen Schäfer: Objektorientierte Entwurfsmethoden. Addison-Wesley, Bonn, 1994

[Schiffer 96] Stefan Schiffer: Visuelle Programmierung – Potential und Grenzen. In: H. C. Mayr (Hrsg.): Beherrschung von Informationssystemen. Oldenbourg, Wien, 1996

[Schiffer 97] Stefan Schiffer:Visuelle Programmierung. In: Peter Rechenberg, Gustav Pomberger (Hrsg.): Informatik-Handbuch. Hanser, München/Wien, 1997

[Sefika et al. 96] Mohlalefi Sefika, Aamod Sane, Roy H. Campbell: Architecture-Oriented Visualization. In: Proceedings of ACM OOPSLA'96, ACM Press, New York, 1996

[Shilling und Stasko 92] John J. Shilling, John T. Stasko: Using Animation to Design, Document and Trace Object-Oriented Systems. Technical Report, Georgia Institute of Technolgy, 1992

[Shilling und Stasko 94] John J. Shilling, John T. Stasko: Using Animation to design object-oriented systems. In: Object Oriented Systems, Jg. 1 Nr. 1, 1994

[Shneiderman 96] Ben Shneiderman: The Eyes Have It: A Task by Data Type Taxonomy for Information Visualizations. In: IEEE Symposium on Visual Languages, 1996

[Smart 95] Julian Smart: User Manual for wxWindows 1.63. Edinburgh 1995

[Smart 95a] Julian Smart: Manual for Object Graphics Library 2.0. Edinburgh 1995

[Stallmann und Pesch 94] Richard M. Stallmann, Roland H. Pesch: Debugging with GDB. Ed. 4.12, Free Software Foundation, Cambridge, Mass., 1994

[Stasko et al. 97] J. Stasko, J. Domingue, B. Price & M. Brown (Hrsg.): Software Visualization: Programming as a Multi-Media Experience. MIT Press, 1997

[Stroustrup 92] Bjarne Stroustrup: The C++ Programming Language. 2nd ed. Addison-Wesley, Reading, Mass., 1992

[Stroustrup 94] Bjarne Stroustrup: Design und Entwicklung von C++. Addison-Wesley, Bonn, 1994

[Sutherland 63] I. E. Sutherland: Sketchpad. A Man-Machine Graphical Communication System. In: AFIPS Conference Proceedings, Spring Joint Computer Conference, S. 2-19, 1963

[Troll 97] Troll Tech AS: Qt Documentation. URL: http://www.troll.no

[Tilley und Smith 96] Scott R. Tilley, Dennis B. Smith: Coming Attractions in Program Understanding. Technical Report CMU/SEI-96-TR-019, ESC-TR-96-019, Pittsburgh 1996

[Vlissides 90] John M. Vlissides: Generalized Graphical Object Editing. Technical Report, Computer Systems Laboratory, Stanford University 1990

[Vlissides und Linton 89] John M. Vlissides, Mark A. Linton: Unidraw: A Framework for Building Domain-Specific Graphical Editors. In: Proceedings of ACM SIGGRAPH/SIGCHI User Interface Software and Technologies '89 Conference, 1989

[West 93] Alan West: Animating C++ Programs. Dynamic C++ Animation White Paper. Objective Software Technology Ltd., 1993

[West 94] Alan West. Making a Case for Animating C++ Programs. Dr Dobbs Journal, Jg. 19 Nr. 11, Okt. 1994

[Yourdon et al. 95] Ed Yourdon, Katharine Whitehead, Jim Thomann, Karin Oppel, Peter Nevermann: Mainstream Objects. An Analysis and Design Approach for Business . Prentice Hall, Upper Saddle River, New Jersey, 1995