

Komponentenorientierte Entwicklung von Grafikapplikationen mit *BOOGA*

Inauguraldissertation
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von
Stephan Amann
von Interlaken, BE

Leiter der Arbeit: Prof. Dr. H. Bieri
Institut für Informatik
und angewandte Mathematik,
Universität Bern

Komponentenorientierte Entwicklung von Grafikapplikationen mit *BOOGA*

Inauguraldissertation
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von
Stephan Amann
von Interlaken, BE

Leiter der Arbeit: Prof. Dr. H. Bieri
Institut für Informatik
und angewandte Mathematik,
Universität Bern

Von der Philosophisch-naturwissenschaftlichen Fakultät angenommen.

Bern, 13. November 1997

Der Dekan:

Prof. Dr. H. Bunke

Zusammenfassung

Wiederverwendung von Software ist nicht nur für die Industrie ein immer noch zentrales Anliegen, auch eine universitäre Umgebung kann von einer konsequenten Umsetzung wiederverwendungsorientierter Konzepte profitieren. Die vorliegende Arbeit stellt die *Wiederverwendung* und die damit zusammenhängende Fragestellungen ins Zentrum des Interesses. Die bei der Entwicklung des *Komponentenframeworks BOOGA*¹ gemachten Erfahrungen dienen als praktische Grundlage der Ausführungen.

Wichtige Eigenschaften von *BOOGA* sind eine grosse *Flexibilität*, eine einfache *Erweiterbarkeit*, eine gute *Erlernbarkeit* und ein breites *Anwendungsbereich* innerhalb der Computergrafik. Die Umsetzung dieser Eigenschaften in *BOOGA*, kombiniert mit einem hohen Wiederverwendungspotential von Architektur *und* Implementierung, wird in dieser Arbeit diskutiert und mit Beispielen illustriert.

Neben der Realisierung einer *Forschungsplattform* sowie einer Diskussion der praktischen Erfahrungen, bildet vor allem auch die Einführung des Konzeptes des *Komponentenframeworks*, sowie eines an den Einsatz eines Komponentenframeworks angepassten *Vorgehensmodells* zur Anwendungsentwicklung einen wesentlichen Bestandteil der vorliegenden Arbeit.

¹Berne's Object-Oriented Graphics Architecture

Dank

Diese Dissertation ist am Institut für Informatik und angewandte Mathematik der Universität Bern unter der Leitung von *Prof. Dr. Hanspeter Bieri* entstanden. Ich möchte mich an dieser Stelle für seine wertvolle Unterstützung, für die gewährten Freiheiten, sowie für das entgegengebrachte Vertrauen bedanken.

Prof. Dr. O. Nierstrasz möchte ich für die Diskussionen und die Anregungen danken.

Die im folgenden dokumentierten Ergebnisse sind in enger Zusammenarbeit mit *Christoph Streit* entstanden. Seine Dissertation [Streit, 1997] und die vorliegende Arbeit sind eng miteinander verknüpft und als gegenseitige Ergänzung zu verstehen. Erst die Lektüre beider Arbeiten erlaubt, ein umfassendes Bild von **BOOGA** zu gewinnen. An dieser Stelle ein herzliches ‘Merci’ an Christoph für die vielen intensiven und anregenden Diskussionen und die gute Zusammenarbeit.

Die Entwicklung von **BOOGA** wurde von sehr vielen Personen mitgetragen. Dank gebührt allen, die zum heutigen System beigetragen und – früher oder später – unsere Begeisterung und Überzeugung teilten. Speziell erwähnen möchte ich *Bernhard Bühlmann*, den ersten Anwender von **BOOGA**. Ferner geht ein Dank an (in ‘Order of Appearance’): *Andrey Collison, Thierry Mathey, Thomas Teuscher, Pascal Habegger, Richard Bächler, Daniel Möri, Beat Liechti, Peter Sagara, Thomas Wenger, Thomas von Siebenthal, Lorenz Ammon, Roland Balmer*.

Ein herzlicher Dank auch allen, welche mir beim Korrekturlesen dieses Textes behilflich waren: *Silvia Michel, Dr. Theo-Dirk Meijler, Andrey Collison* und *Christoph Streit*.

Barbara Michel sei für den Zugriff auf ihre umfangreiche Kunsliteratur, der die Abbildungen zu Beginn der Kapitel entstammen, gedankt.

Ein spezieller Dank geht an *Dr. Igor Metz*, welcher mir vor vielen Jahren die Türen zur objektorientierten Denkweise aufschloss und in vielen Diskussionen wertvolle Anregungen vermittelte.

Als letztes ein sehr grosser Dank an alle Freunde, die mir auch in stressbeladenen Phasen viel Geduld entgegenbrachten und mir dadurch wieder neuen Antrieb vermittelten. *I'll be back!*

Inhaltsverzeichnis

1 Einleitung	1
1.1 Motivation	2
1.2 Wiederverwendung	3
1.3 Inhalt und Abgrenzung	6
2 Wiederverwendungskonzepte	9
2.1 Einleitung	10
2.2 Ein Klassifizierungsansatz	12
2.3 Implementationswiederverwendung	17
2.4 Architekturwiederverwendung	31
2.5 Wiederverwendung durch Komponenten	52
2.6 Abschliessende Betrachtungen	69
3 <i>BOOGA</i>: Grundlagen und Motivation	77
3.1 Einleitung	78
3.2 Die Teilgebiete der Computergrafik	79
3.3 Anwendungsgebiete von <i>BOOGA</i>	82
3.4 Anforderungen an <i>BOOGA</i>	84
4 Die Architektur von <i>BOOGA</i>	89
4.1 Einleitung	90
4.2 Die logische Sicht	91
4.3 Die physische Sicht: ein Komponentenframework	96
4.4 Hot-Spots	99
4.5 Allgemeine Konzepte	100
4.6 Die Architektur als Mittel zur Zielerfüllung	103
5 Die Applikationsentwicklung mit <i>BOOGA</i>	107
5.1 Einleitung	108
5.2 Bekannte Vorgehensmodelle	109
5.3 Der Applikationsbegriff von <i>BOOGA</i>	111
5.4 Vorgehensmodell von <i>BOOGA</i>	116
5.5 Projektorganisation	137
5.6 Zusammenfassung	142

6	Beispielanwendungen	145
6.1	Einleitung	146
6.2	Anwendung “Wireframe”	147
6.3	Anwendung “Radiosity”	164
6.4	Anwendung “Reverse Engineering”	173
7	Dokumentationskonzepte	179
7.1	Einleitung	180
7.2	Alternative Dokumentationskonzepte	183
7.3	Ein Dokumentationsansatz für <i>BOOGA</i>	186
7.4	Schlussbemerkungen	194
8	Schlussbemerkungen	197
8.1	Erreichte Ziele	198
8.2	Erfahrungen	200
8.3	Ausblick	206
Literaturverzeichnis		209
A	Notationen und Dateiformate	221
A.1	<i>BOOGA</i> -LEGOS	222
A.2	BSDL Sprachreferenz	227
A.3	Das <i>BOOGA</i> -Bildformat PIXI	231
B	Der <i>BOOGA</i>-Styleguide	235
B.1	Einleitung	236
B.2	Allgemeines	236
B.3	Aufbau einer Klasse	236
B.4	Namengebung	237
B.5	Codierkonventionen	240
C	Beispiel einer auf Patterns basierenden Dokumentation	243



Eduard Manet.
Le Déjeuner sur l'Herbe.
1863.

Kapitel 1

Einleitung

*Oh, lass es die Weisen doch verständlich
sagen, mir das Hirn nicht mit Erkenntnis
plagen.*

Christian Dietrich Grabbe,
1801 – 1836

1.1 Motivation

Für weitere Arbeiten auf diesem Gebiet scheint es mir wichtig, dass den Bearbeitern ein klarer Rahmen vorgegeben wird, und, viel wichtiger, dass auf ein bestehendes System aufgebaut werden kann und nicht bei 'Null' angefangen werden muss. So können wesentlich weiterführende Fragestellungen bearbeitet und beantwortet werden. Gerade auf dem Gebiet dieser Arbeit scheint es mir unmöglich, weiterführende Techniken als 'One-Man-Job' im Rahmen eines Diploms zu bearbeiten, ohne auf ein bestehendes System aufzubauen.

Diese Erkenntnis am Schluss meiner Diplomarbeit [Amann, 1993] war für mich persönlich die stärkste Motivation, die Arbeit an **BOOGA** aufzunehmen. Viele der im folgenden besprochenen Ideen und Lösungen finden ihren Ursprung in diesen Gedanken.

Anfang der 90er Jahre wurden in der Forschungsgruppe Computergeometrie und Grafik immer mehr Arbeiten aus dem Bereich der Computergrafik geschrieben, unter anderem auch meine Diplomarbeit. Teilweise übereinanderliegende Anforderungen und der Mangel eines gemeinsamen Standards haben dazu geführt, dass ähnliche Subsysteme mehrfach entwickelt wurden. Dies führte dazu, dass zuviel Aufwand in Standardproblemstellungen investiert werden musste. In der Folge stand oft zuwenig Zeit für tieferliegende Untersuchungen im eigentlichen Problembereich zur Verfügung. Anfang 1994 wurde der Entschluss gefasst, ein System zu entwickeln, welches fortan als Basis für die weiteren Entwicklungen dienen sollte. Das *Komponentenframework* **BOOGA** ist das Resultat dieser nunmehr über dreijährigen Anstrengung. Das Grundsystem wurde von Christoph Streit und mir entworfen und implementiert. Nach einer Vorlaufzeit von etwa einem Jahr wurden die ersten Projekte unter Einbezug weiterer Personen, aufbauend auf dem sich weiterentwickelnden Kern, gestartet.

BOOGA ist als Forschungsplattform für den Bereich Computergrafik konzipiert (vgl. Kapitel 4). Der Begriff Forschungsplattform subsumiert eine Anzahl von Anforderungen, die im wesentlichen eine grosse Erweiterbarkeit und Flexibilität beinhalten. Zentral ist die einfache, fast automatische Wiederverwendung bereits implementierter Bausteine. Es konnte beobachtet werden, dass Anwender von **BOOGA** ganz selbstverständlich existierende Bausteine verwenden.

In dieser Arbeit steht der Begriff *Wiederverwendung* im Zentrum des Interesses, wobei vor allem die technischen Probleme betrachtet werden. Neben einer Aufarbeitung einiger der wichtigsten Wiederverwendungsansätze wird **BOOGA** beschrieben, das von Anfang an konsequent mit dem Ziel entwickelt wurde, Wiederverwendung aktiv zu unterstützen.

1.2 Wiederverwendung

Wiederverwendung bereits erbrachter Leistung ist ein zentrales Anliegen jeder Ingenieurwissenschaft. Während sich in anderen Ingenieurdisziplinen das Prinzip der Wiederverwendung als selbstverständliche Prämisse etabliert hat, herrscht in der Informatik weitgehende Einigkeit bei der Ansicht, dass der Einsatz von Bauteilen noch nicht zum Standard geworden ist [Krueger, 1992; Udell, 1994]. In anderen Fachbereichen kein Diskussionsgegenstand – da offensichtlich und alltäglich –, wird in der Informatik sehr viel über dieses Thema gesprochen. Wiederverwendung ist zu einem eigentlichen Schlagwort geworden, das oft marktschreierisch eingesetzt wird, um eine neue Technologie oder ein neues Produkt zu verkaufen.

In der Informatik wird unter Wiederverwendung verstanden, bereits existierende Artefakte¹ bei der Konstruktion neuer Softwaresysteme wiederum einzusetzen. Dabei soll die Wiederverwendung nicht auf Codefragmente beschränkt sein, sondern alle Phasen der Softwarekonstruktion, also auch Analyse, Design, Dokumentation, etc. umfassen. Die Form der Wiederverwendung reicht dabei vom Kopieren einzelner Quelltextzeilen bis zum Konfigurieren von Standardapplikationen zur Laufzeit.

Geht man der Frage nach, warum wir Informatiker denn so viel über Wiederverwendung reden, so stellt sich heraus, dass uns deren bisher erreichter Grad im allgemeinen nicht zu befriedigen vermag. Durch den Einsatz unterschiedlichster Techniken (vgl. Kapitel 2) ist es gelungen, Wiederverwendung auf der *untersten Abstraktionsstufe* zu etablieren. Dies sogar so gut, dass wir uns dessen im Alltag oft nicht einmal bewusst sind. Das deutlichste Beispiel hierfür sind Hochsprachen, die in Abschnitt 2.3.1 besprochen werden.

Soll Wiederverwendung als natürliches Element in den Softwareentwicklungszyklus integriert werden, genügt es nicht, einfach eine Bibliothek wiederverwendbarer Softwareblöcke zur Verfügung zu stellen. Ähnlich, wie auf einer Schrotthalde nicht Neuwagen entstehen können, kann in einer solchen Umgebung der Wiederverwendung von bereits existierendem Code in neuen Applikationen nur wenig Erfolg beschieden sein.

Wesentliches Element einer erfolgreichen *Kultur der Wiederverwendung* ist deshalb die richtige Kombination von Wiederverwendungstechnologien auf den Stufen Design *und* Implementierung, um einerseits die nötige

¹In der Literatur über *Wiederverwendung von Software* und *Software Engineering* wird der Begriff *Artefakt* als Oberbegriff für ein Teilprodukt im Verlauf des Softwareerstellungsprozesses verwendet. In der Literatur über *Computergrafik* dagegen wird unter einem *Artefakt* ein sich in einem Bild manifestierender Diskretisierungsfehler oder eine Rechenun genauigkeit verstanden.

In dieser Arbeit wird Artefakt in der Bedeutung des Software Engineering verwendet. Teilweise wird der Begriff *Element* als Synonym zu Artefakt verwendet.

Flexibilität zu erreichen, andererseits aber die stets damit einhergehende Komplexität so weit wie möglich zu umgehen. Weitere, wichtige Elemente sind ein angepasstes Vorgehensmodell zur Entwicklung von Applikationen und geeignete Verfahren zur Dokumentation. Ebenfalls wichtig sind die Organisation des Entwicklungsteams sowie eine daran angepasste Entwicklungsumgebung. Alle diese Elemente wurden im Projekt **BOOGA** berücksichtigt und werden in der vorliegenden Arbeit ausführlich diskutiert. Viele Aussagen können dabei von der konkreten Anwendung, also **BOOGA**, abstrahiert und verallgemeinert werden.

Die Umsetzung einer Kultur der Wiederverwendung umfasst Problemstellungen aus den unterschiedlichen Wissenschaftsdisziplinen:

- *Technologische Probleme*

Der konsequente Einsatz fertiger Teillösungen bedarf entsprechend flexibler Architekturen und Hilfsmittel. Objektorientierte Technologien bieten hierfür gute Ansätze.

Sind erst einmal eine grosse Anzahl von wiederverwendbaren Bausteinen vorhanden, so müssen technische Voraussetzungen geschaffen werden, um diese Bausteine zu dokumentieren, zu katalogisieren und zu speichern. Weiter muss der Zugriff auf diese Bausteine einfach und schnell möglich sein.

Im Bereich des Software Engineerings gibt es zahlreiche Methoden, die sich unter anderem auch das Schlagwort der Wiederverwendung auf die Fahne schreiben. Die wenigsten dieser Methoden bieten aber auch tatsächlich ein Vorgehensmodell an, das bekannte Wiederverwendungstechnologien integriert.

Trotz all dieser Schwierigkeiten sind die technologischen Probleme heute nicht mehr die grössten Schwierigkeiten hinsichtlich der Wiederverwendung. Die folgenden Probleme dürften heute häufiger zum Misserfolg von Projekten mit einem grossen Wiederverwendungsanspruch beitragen.

- *Managementprobleme*

Die Einführung von Technologien, die breite Wiederverwendung ermöglichen, sind in einem ersten Schritt stets mit Kosten verbunden, da Investitionen in die Ausbildung der Mitarbeiter getätigt sowie bestehende Abläufe neu überdacht werden müssen. Die häufige Praxis, Entscheidungsträger jährlich am Gewinn zu beteiligen, kann dazu führen, dass solche Investitionen nicht rechtzeitig getätigt werden.

- *Psychologische Probleme*

Das *Not-invented-here* Syndrom ist ein ernstzunehmendes psycholo-

gisches Problem, wenn es darum geht, Wiederverwendungstechnologien auf breiter Basis einzuführen. Genügend Ressourcen vorausgesetzt, ist die Bereitschaft, Bausteine zur Wiederverwendung zur Verfügung zu stellen, oft grösser, als die Bereitschaft, diese Bausteine dann auch einzusetzen.

- *Wirtschaftliche Probleme*

Herstellung und Vertrieb wiederverwendbarer Bausteine können zu einem neuen Industriezweig führen. Es müssen Modelle gefunden werden, um den Einsatz und die Lizenzierung solcher Bausteine in Systemen anderer Hersteller zu verrechnen.

1.3 Inhalt und Abgrenzung

Im Zusammenhang mit **BOOGA** gibt es im wesentlichen zwei wichtige Themengebiete: die *Computergrafik*, welche den Anwendungsbereich darstellt und den umfangreichen Problemkreis der *Wiederverwendung*. Die beiden Themengebiete sind natürlich sehr stark miteinander verbunden. Sie wurden auf die beiden **BOOGA**-Dissertationen aufgeteilt: Die Arbeit von Christoph Streit [Streit, 1997] konzentriert sich auf die grafikrelevanten Teile, die vorliegende Arbeit auf die Aspekte des Software Engineering sowie der Wiederverwendung.

Christoph Streit behandelt in seiner Dissertation die generellen Anforderungen an Grafiksysteme und gibt einen Überblick bestehender Systeme. In einem zweiten Teil werden die wesentlichen Mechanismen und Abstraktionen von **BOOGA** erläutert.

Ein guter Eindruck des gesamten **BOOGA**-Projektes kann nur gewonnen werden, wenn beide Arbeiten gelesen werden.

In Kapitel 2 der vorliegenden Arbeit werden die Konzepte zur Wiederverwendung von Quellcode und Design diskutiert, die auch in **BOOGA** zur Anwendung kommen. Anhand eines Klassifizierungsschemas werden die verschiedenen Techniken einander gegenübergestellt und deren Vor- und Nachteile diskutiert.

Kapitel 3 versucht das Anwendungsgebiet von **BOOGA**, die Computergrafik zu strukturieren. Dazu werden einige Ansätze aus der Literatur besprochen und schliesslich das **BOOGA** zugrundeliegende Modell eingeführt.

Auf diesem Modell aufbauend wird, wie in Kapitel 4 erläutert, die Architektur von **BOOGA** aufgebaut.

BOOGA schreibt dem Anwendungsentwickler genau vor, wie die Struktur der Anwendung auszusehen hat. Der Einbezug existierender Softwarelemente ist zentral. Dies führt dazu, dass bestehende Vorgehensmodelle kaum angewendet können. In Kapitel 5 wird ein Modell für die Entwicklung von Anwendungen mit dem Komponentenframework **BOOGA** vorgestellt.

In Kapitel 6 wird dieses Vorgehensmodell mit einigen Beispielanwendungen illustriert. Die einzelnen Teilschritte des Modells werden anhand der Beispiele detaillierter besprochen.

Ein äusserst wichtiges Element der Wiederverwendung ist die Dokumentation. Kapitel 7 ist deshalb ganz diesem Thema gewidmet. Es wird auf die spezielle Probleme im Zusammenhang mit der Dokumentation wiederverwendbarer Software eingegangen und einige Lösungsansätze werden vorgestellt.

Kapitel 8 schliesst diese Arbeit mit einigen weiteren Betrachtungen bezüglich Projektorganisation ab und fasst die Erfahrungen, die während der Entwicklung von **BOOGA** gemacht wurden, zusammen.

Der Leser dieser Arbeit sollte grundlegende Kenntnisse der Computergrafik sowie der objektorientierten Technologien besitzen. Als Implementierungssprache für **BOOGA** wurde C++ gewählt. An einigen Stellen dieser Arbeit sind zur Illustration Codefragmente in C++ abgebildet. Mindestens passive Kenntnisse dieser Sprache sind nötig, um die entsprechenden Stellen nachvollziehen zu können.



Segesta.
Der Tempel von Osten.
Spätes 5. Jahrhundert v. Chr.

Kapitel 2

Wiederverwendungskonzepte

*Progress, far from consisting in change,
depends on retentiveness*

*...
Those who cannot remember the past
are condemned to fulfil it.*

Life of Reason,
George Santayana,
1863 - 1952

2.1 Einleitung

Wiederverwendung ist einer der ältesten Wünsche der Informatiker. Es ist nicht weiter erstaunlich, dass an derselben Konferenz, an der das heute noch immer aktuelle Schlagwort der *Software Krise* geprägt wurde, auch gleich ein möglicher Ausweg aufgezeigt wurde: *Software Komponenten* [McIlroy, 1968]. Diese Konferenz, die *NATO Software Engineering Conference*, wird heute als die Geburtsstunde des Software Engineering betrachtet.

Die Ziele, die mit Wiederverwendung erreicht werden wollen, sind vielfältig, doch lassen sie sich alle auf eine gemeinsame Basis zurückführen: die immensen Kosten, die bei der Entwicklung von Software anfallen, sollen massiv gesenkt werden. Teilziele, die direkt oder indirekt zur Kostensenkung beitragen und mittels Wiederverwendung erreicht werden können, sind:

- *Verbesserung der Qualität sowie der Zuverlässigkeit von Software*

Durch den wiederholten Einsatz derselben Softwareelemente werden diese immer stabiler und sicherer. Es wird auf bewährte Konzepte und Implementierungen gebaut, anstatt jedesmal eine neue Lösung zu erarbeiten.

- *Verkürzung der Entwicklungszeit*

Durch die bewusste Planung des Einsatzes bestehender Artefakte verringert sich der Aufwand in allen Entwicklungsphasen.

- *Verkleinerung des Testaufwandes*

Da Softwareelemente verwendet werden, die bereits in anderen, sich im Einsatz befindenden Systemen integriert sind, kann davon ausgegangen werden, dass die wiederverwendeten Artefakte korrekt funktionieren. Es muss somit lediglich das *Zusammenspiel* der neu verknüpften, bestehenden Elemente getestet werden.

- *Vereinfachung der Wartung*

Wird in einem wiederverwendeten Artefakt ein Fehler entdeckt, muss dieses idealerweise nur *einmal* korrigiert und dann in allen Systemen ersetzt werden, in denen es zum Einsatz kommt.

Es gibt eine breite Vielfalt von Techniken, die unter das Thema “Wiederverwendung” fallen. Im folgenden werden die im Rahmen von *BOOGA* verwendeten Techniken besprochen, namentlich *Hochsprachen* (Abschnitt 2.3.1), *Bibliotheken* (Abschnitt 2.3.2), *Design Patterns* (Abschnitt 2.4.1), *Frameworks* (Abschnitt 2.4.2) und *Komponenten* (Abschnitt 2.5).

Die objektorientierten Technologien haben heute weite Verbreitung gefunden. Es wird allgemein anerkannt, dass diese Verfahren bessere Abstraktionsmöglichkeiten bieten als die herkömmlichen prozeduralen Ansätze. Da auch **BOOGA** mittels objektorientierter Verfahren entwickelt wurde, liegt der Fokus in diesem Kapitel auf diesen Verfahren. Wie in Abschnitt 2.3.1 besprochen wird, sind aber alle in diesem Kapitel aufgeführten Technologien unabhängig vom verwendeten Paradigma einsetzbar. Die jeweiligen Aussagen sind in der Regel auf objektorientierte *und* prozedurale Ausprägungen der Technologien anwendbar.

Im folgenden soll das Klassifizierungsschema von Charles W. Krueger [Krueger, 1992] vorgestellt werden, welches die Grundlage für die restlichen Ausführungen dieses Kapitels bildet. Es soll dabei helfen, die verschiedenen besprochenen Ansätze möglichst aussagekräftig miteinander zu vergleichen. In Abschnitt 2.3 werden die bekanntesten Techniken zur *Wiederverwendung von Code* vorgestellt. Da die Implementation nur einen Teil des gesamten Softwareentwicklungszyklus ausmacht, ist der Nutzen dieser rein auf die Wiederverwendung der Implementation ausgerichteten Verfahren beschränkt. Die in Abschnitt 2.4 besprochenen Techniken berücksichtigen diese Tatsache durch die *Wiederverwendung von Design*.

In Abschnitt 2.5 werden schliesslich die Vorteile der zuvor besprochenen Ansätze, namentlich der einfache Einsatz von Implementationswiederverwendung und die Mächtigkeit von Architekturwiederverwendung kombiniert.

2.2 Ein Klassifizierungsansatz

Obwohl die einzelnen Wiederverwendungstechnologien sich sehr stark unterscheiden, haben sie doch einige Elemente gemein. Die *Abstraktion*, die auf zufällige Einzelheiten verzichtende, zusammenfassende Darstellung, als wohl wichtigster Elementarbaustein [Booch, 1994b; Krueger, 1992] der Informatikkonzepte, spielt auch bei der Wiederverwendung eine Schlüsselrolle:

*Abstraktion und Wiederverwendbarkeit sind
zwei Seiten derselben Medaille.*

[Wegner, 1983]

Neben der Wahl einer geeigneten Abstraktion sind weitere Kriterien wichtig:

Selektion Um ein wiederverwendbares Stück Software überhaupt einsetzen zu können, muss es zuerst *gefunden* werden. Der Vorgang der Selektion kann ohne geeignete Unterstützung so aufwendig werden, dass keine Wiederverwendung stattfindet.

Spezialisierung Bei vielen Wiederverwendungstechniken werden ähnliche Softwareelemente zusammengefasst und *generalisiert*. Um ein solchermassen verallgemeinertes Element einsetzen zu können, muss dieses zuerst *spezialisiert* werden. Spezialisierungsverfahren, die viel Wissen voraussetzen, beeinträchtigen die effiziente Wiederverwendung.

Integration Typischerweise beinhalten Wiederverwendungstechnologien ein *Integrationsframework*, um die einzelnen wiederverwendbaren Elemente zusammenzubauen und in die darunterliegende Abstraktionsschicht einzubetten.

Als Erweiterung gegenüber der Arbeit von Krueger soll ein weiteres Kriterium berücksichtigt werden:

Kombinierbarkeit Ein wesentliches Element einer guten Wiederverwendungstechnologie ist deren Fähigkeit, mit anderen Ansätzen derselben Gruppe kombiniert werden zu können. So ist es beispielsweise oft ein schwieriges Unterfangen, Funktionen, die in unterschiedlichen Hochsprachen entwickelt wurden, in derselben Anwendung

zu gebrauchen. Gerade diese Fähigkeit wird aber immer mehr zu einer Herausforderung, nicht zuletzt wenn es darum geht, bestehende Altlästen (*legacy applications*) weiterzuverwenden.

Charles W. Krueger unterteilt eine Abstraktion in zwei Schichten (vgl. Abbildung 2.1): die obere Schicht wird als Spezifikation bezeichnet, die untere als Realisierung.

Abstraktion

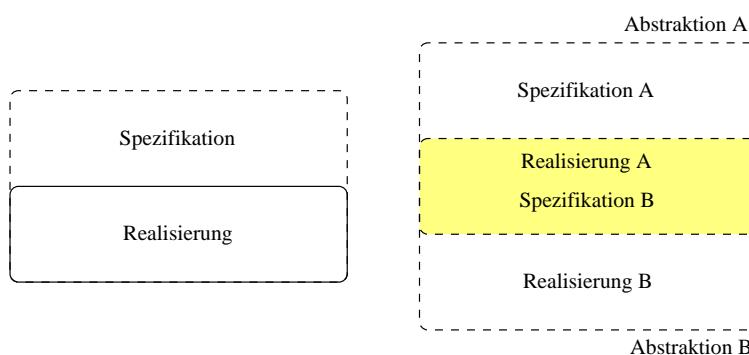


Abbildung 2.1

linke Abbildung:
Eine Abstraktion besteht aus den zwei Teilen Spezifikation und Realisierung.

rechte Abbildung:
Spezialisierung und Realisierung aufeinanderliegender Abstraktionsschichten greifen ineinander.

Normalerweise werden beim Problemlösungsprozess Abstraktionen aufeinandergeschichtet. So ist beispielsweise eine Programmiersprache eine Abstraktionsschicht über der Assemblersprache, die ihrerseits die Maschinensprache abstrahiert. Die Realisierungsschicht der höheren Programmiersprache übernimmt die Umsetzung auf die Assemblersprache und verwendet dazu die Spezifikation dieser Abstraktionsstufe.

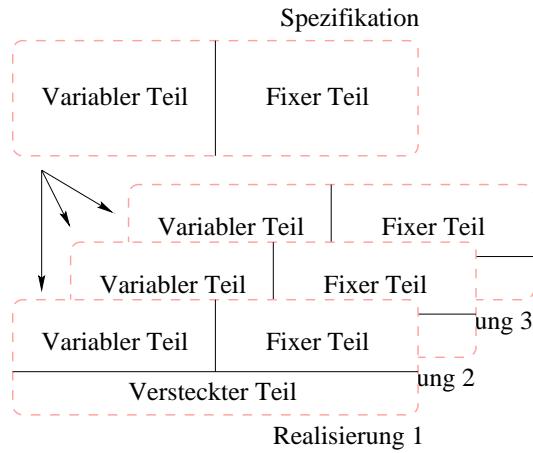
Während jede Abstraktion immer aus einer Spezifikation und einer Realisierung besteht, gibt es weitere, optionale Elemente innerhalb einer Abstraktion. Krueger unterscheidet hier zwischen *variablen*, *fixen* und *versteckten* Teilen. Abbildung 2.2 illustriert diese Aufteilung.

Betrachtet man beispielsweise einen Stack, so ist diese Abstraktion durch die möglichen Operationen darauf vollständig definiert. Somit gehören diese invarianten Charakteristiken des Stacks in den fixen Teil der Abstraktion. Der Datentyp, der im Stack abgelegt werden soll, beeinflusst diese Invarianten in keiner Weise und kann somit typunabhängig im variablen Teil der Abstraktion definiert werden. Wird der Stack mit Hilfe von C++-Templates generisch implementiert, so gelingt es, die gesamte Realisierung zu verstecken. Verwendet man die Technik "Copy and Modify", kopiert also Quellcode und modifiziert diesen anschliessend, um die richtige Typisierung des Stacks von Hand durchzuführen, so wird der versteckte Teil dieser Abstraktion leer sein.

Der Idealfall einer Abstraktion wäre eine vollständig versteckte Realisierung sowie eine völlig flexible Spezifikation. Manche Wiederverwen-

Abbildung 2.2

Eine Abstraktion besteht, orthogonal zu der Unterscheidung in Spezifikation und Realisierung, aus 3 Elementen: einem variablen Teil, einem fixem Teil und einem verstecktem Teil. Eine Spezifikation mit einem variablen Teil beschreibt eine ganze Familie von möglichen Realisationen der Abstraktion.



dungstechnologien kommen, wie noch gezeigt wird, recht nahe an dieses Ideal heran.

Bewertung

Das Klassifizierungsschema von Krueger soll den direkten Vergleich der einzelnen Technologien vereinfachen. Jede Technologie wird nach einer allgemeinen Besprechung bewertet. Da die Klassifizierung nach den fünf besprochenen Kriterien *Abstraktion*, *Selektion*, *Spezialisierung*, *Integration* und *Kombinierbarkeit* geschehen soll, liegt es nahe, auch die Bewertung nach diesen Kriterien vorzunehmen.

Das Hauptaugenmerk beim Einsatz von Wiederverwendungstechnologien hat in der Effizienzsteigerung zu liegen. Der verbesserte Einsatz der verfügbaren Ressourcen rechtfertigt schliesslich den oftmals nötigen zusätzlichen Initialaufwand. Ist der Anfang einmal gemacht, sollte sich eine solche Investition aber bezahlt machen:

“Binsenwahrheit” 1

Eine Wiederverwendungstechnologie muss, um effizient zu sein, die Wiederverwendung von Softwarelementen einfacher gestalten als deren komplette Neuentwicklung.

Der Selektion kommt eine sehr grosse Bedeutung zu, steht sie doch am Anfang jeglicher Wiederverwendung. Die Umsetzung der folgenden Aussage kann in der Praxis allerdings zu einem Problem werden:

“Binsenwahrheit” 2

Um ein Artefakt zur Wiederverwendung auswählen zu können, muss man zuerst wissen, was es tut.

Der Einsatz formalisierter Beschreibungen sowie geeigneter Hilfsmittel zum Suchen können hier helfen, den Aufwand gering zu halten. In Kapitel 8 werden einige Überlegungen hierzu angestellt.

Eng verknüpft mit der vorhergehenden Aussage ist auch die folgende:

“Binsenwahrheit” 3

Um ein Artefakt effizient wiederzuverwenden, muss man es schneller finden als neu schreiben können.

Das heisst, dass einerseits das Auffinden von Kandidaten zur Wiederverwendung einfach möglich sein muss, andererseits soll für diese Kandidaten eine gute Beschreibung vorliegen, die es wiederum innerhalb nützlicher Zeit erlaubt, eine endgültige Wahl zu treffen.

Während bei den Kriterien *Selektion*, *Spezialisierung*, *Integration* und *Kombinierbarkeit* die Bewertung mittels einer Abschätzung des nötigen Aufwandes erfolgen kann und somit direkt aus dem Gesamtziel der Aufwandminimierung ableitbar ist, erscheint es doch recht schwierig, ein objektives Kriterium für die Güte einer Abstraktion anzugeben. Charles W. Krueger schlägt hierfür den Begriff der *kognitiven Distanz* vor und definiert diesen wie folgt:

Definition 2.1 (Kognitive Distanz)

Grösse der intellektuellen Anstrengung, die vom Softwareentwickler erbracht werden muss, um das Softwaresystem von einer Entwicklungsstufe zu einer anderen zu bringen.

Es versteht sich von selbst, dass diese Definition lediglich eine informelle Grösse beschreibt, die eine auf Erfahrungswerten basierende Einschätzung und grobe Vergleichbarkeit der einzelnen Technologien erlaubt.

Das Ziel jeder Wiederverwendungstechnologie wird es sein, diese kognitive Distanz zu minimieren und zwar durch:

1. Verwendung von Abstraktionen mit knappen, aber doch mächtigen variablen und fixen Teilen
2. Maximierung des versteckten Teils einer Abstraktion
3. Automatisierung des Übergangs von der Spezifikation zur Realisierung, beispielsweise durch Verwendung von Compilern.

Dies kann wie folgt zusammengefasst werden:

“Binsenwahrheit” 4

Eine Wiederverwendungstechnologie muss, um effizient zu sein, die kognitive Distanz zwischen der Architektur eines Systems und dessen Implementierung verkleinern.

Diskutiert man heute über Wiederverwendung, so kommt die Sprache oft unmittelbar auf den Einsatz objektorientierter Technologien. Der Zusammenhang zwischen Wiederverwendung und Objektorientierung wird im folgenden Abschnitt besprochen.

2.3 Implementationswiederverwendung

Die älteste und am weitesten verbreitete Art von Wiederverwendung ist die direkte Wiederverwendung von während der Implementierungsphase anfallenden Softwareartefakten.

Ein sehr häufig verwendeter, einfacher Ansatz der Wiederverwendung besteht darin, Design- oder Codefragmente früherer Lösungen zu kopieren und zu modifizieren. [Krueger, 1992] nennt dieses Vorgehen in seiner Arbeit sehr treffend *scavenging*¹ (fleddern). Durch die direkte Wiederverwendung können frühere Aufwendungen ohne Zusatzaufwand für eine neue Aufgabe herangezogen werden. Bei der ‘Fledderei’ von Code werden einzelne Zeilen von Quellcode kopiert und angepasst. Jeder Entwickler dürfte dieses Verfahren sehr häufig einsetzen. Der Nachteil dieses Vorgehens ist aber einerseits die Fehleranfälligkeit², andererseits der geringe Nutzen auf die nachfolgende Testphase. Da der wiederverwendete Code modifiziert wurde, können sich wieder Fehler eingeschlichen haben, und so müssen alle Tests vollständig neu durchgeführt werden.

‘Fledderei’

Die Effizienz der ‘Fledderei’ ist durch den vollständig fehlenden Formalismus stark eingeschränkt. In der Regel kann auch nur Code gefleddert werden, den der betreffende Entwickler selber geschrieben und auf den er direkten Zugriff hat.

Im folgenden werden zwei formalere Methoden der Wiederverwendung ausführlicher besprochen und anhand der erweiterten Krueger-Klassifizierung bewertet.

2.3.1 Sprachen

Es mag überraschen, in einem Kapitel über Wiederverwendung einen Abschnitt über Sprachen zu finden. Andererseits sind gerade diese ein Beispiel für eine besonders effiziente Wiederverwendung. Es ist schwierig, mit einer anderen Technologie eine vergleichbare Effizienzsteigerung, wie sie durch den Einsatz von Hochsprachen im Vergleich zur Programmierung in Assembler erzielt wird, zu erreichen (manche Autoren [Brooks, 1975] sprechen von einem Faktor fünf).

Assemblerprogrammierer hatten das Wissen, wie bestimmte Konstrukte wie beispielsweise Schleifen, Subroutinen, etc. zu implementieren sind.

¹ *Scavenger*, engl.: Aasfresser

² Manche der am schwierigsten zu findenden Fehler in einem System können nachträglich auf diese Art der Wiederverwendung zurückgeführt werden. Nach dem Kopieren von Quellcodeteilen wurden nicht alle nötigen Anpassungen wie zum Beispiel Umbenennung von Variablennamen mit der nötigen Sorgfalt durchgeführt.

Diese Aufgaben mussten aber jedesmal neu gelöst werden, da die verwendete Abstraktion, die Assemblersprache, keinerlei Hilfe für eine Wiederverwendung bot. Durch die Entwicklung von Hochsprachen wurden die den Assemblerprogrammierern geläufigen Muster zu den zugrundeliegenden Sprachkonstrukten dieser Neuerung. Eine neue Abstraktionsebene war gefunden, welche die Assemblerschicht vollständig verbarg.

Dieses Anheben des Abstraktionsniveaus hat aber nicht nur beim Schritt von der Assemblerprogrammierung zur Programmierung in einer Hochsprache stattgefunden, sondern auch beim Schritt von den prozeduralen Sprachen zu den objektorientierten. So sind beispielsweise viele der Muster, die von geübten C-Programmierern eingesetzt wurden, um flexible und erweiterbare Systeme zu erzeugen, in die Sprache C++ eingeflossen und bilden hier allgemein verfügbare und dokumentierte Sprachelemente.

Die wiederverwendbaren Elemente einer Sprache auf einem bestimmten Abstraktionsniveau können die Implementierungsmuster der Sprachen auf der darunterliegenden Ebene sein. Die Konstrukte dieser Sprachen sind die Spezifikationen dieser Abstraktion, die entsprechenden Bausteine in Assembler sind die Realisierungen.

Der Befehl

```
if (Ausdruck) then
    Befehle 1
else
    Befehle 2
endif
```

ist ein wiederverwendbares Konstrukt, dessen Spezifikation sich direkt auf die Realisierung in der Assemblersprache abbilden lässt. Die variablen Teile in der Abstraktion sind der *Ausdruck* und die beiden *Befehlsblöcke*. Der fixe Teil ist die semantischen Beschreibung des *if-then-else*-Befehles:

Der *Ausdruck* wird ausgewertet. Ist sein Wert logisch wahr, so werden die Befehle unmittelbar nach dem *then* ausgeführt (*Befehle 1*), ansonsten die Befehle nach dem *else* (*Befehle 2*).

Der versteckte Teil der Abstraktion beinhaltet die gesamte Umsetzung in Assemblercode, bzw. in die zugrundeliegende Hochsprache.³

³Manche Sprachen werden von den entsprechenden Compilern nicht direkt in Assemblercode übersetzt, sondern zuerst in eine andere Sprache auf tieferem Abstraktionsniveau. Die ersten C++ Compiler (*C-Front* von AT&T) waren Compiler, die C++ nach C übersetzten. Anschliessend wurde ein Standard C-Compiler aufgerufen, um den Assemblercode zu erzeugen. Meist wird diese Art der Compiler, sobald genügend Erfahrung mit der Sprache existiert, durch *native* Compiler ersetzt, die direkt Maschinencode erzeugen.

Obwohl mit Simula bereits Ende der 60er Jahre der Grundstock für ein neues Paradigma gelegt wurde, haben die darauf aufbauenden objektorientierten Sprachen erst in den letzten 10 Jahren breite Beachtung gefunden. Heute hat sich die Erkenntnis durchgesetzt, dass die objektorientierten Technologien neben anderen Vorteilen vor allem auch die besseren Abstraktionsmöglichkeiten bieten. Aufgrund übertriebener Hoffnungen hat sich teilweise allerdings auch wieder Ernüchterung bezüglich dieser Vorteile verbreitet. Einige Aspekte dieser Problematik sollen in der Folge besprochen werden.

“Objekttechnologie hat das Versprechen der Wiederverwendbarkeit gebrochen” lautete der Untertitel eines Artikels, der 1994 im Magazin *Byte* [Udell, 1994] erschien. Auch sonst macht sich in letzter Zeit Ernüchterung in Bezug auf objektorientierte Technologien bemerkbar; Taligent⁴ musste seinen Versuch abbrechen, ein vollständig objektorientiertes Betriebssystem und eine Umgebung zur Anwendungsprogrammierung aufzubauen; Vorträge mit kritischen Titeln⁵ ziehen interessiertes Publikum an. Auch im beruflichen Umfeld des Autors sind immer wieder enttäuschte Stimmen zu vernehmen, die feststellen, die objektorientierten Technologien hätten versagt oder zuviel versprochen. Eine Umfrage unter deutschen Versicherungsgesellschaften [Schnur, 1996] hat ergeben, dass immerhin 15% der befragten Unternehmen objektorientierte Technologien *ablehnen* und 37% lediglich den Markt beobachten. Insgesamt setzen nur knapp 40% der befragten Unternehmen diese Technologie ein. Weiter zeigte die Umfrage, dass die Erwartung fast immer wesentlich höher liegen als die erreichten Ziele.

Die Gründe für dieses “Versagen” der Objekttechnologie sind vielschichtig. Einerseits wurde der Begriff *objektorientiert* aus Marketinggründen überstrapaziert; da mit *objektorientiert* positive Eigenschaften wie modern, erweiterbar, wiederverwendbar, etc. verbunden werden, lässt sich ein “objektorientiertes” Produkt immer besser verkaufen als eines ohne dieses Attribut. Andererseits scheint teilweise vergessen worden zu sein, dass mit den objektorientierten Technologien lediglich ein Satz von Werkzeugen bereitgestellt wird, der für gewisse Aufgaben etwas besser geeignet ist als die herkömmlichen Sprachen. Liest man die Ankündigungen zu neuen Produkten, die sich mit dem Zusatz *objektorientiert* schmücken, so stellt man teilweise fest, dass die dort gemachten Versprechen genau diese Haltung verstärken.

Der Paradigmen-Disput

⁴Taligent Inc. war ein Joint-Venture von Apple Computer Inc., IBM Corporation und Hewlett-Packard Company. Taligent Inc. hat seine Bestrebungen Anfang 1996 eingestellt.

⁵“Mythos der objektorientierten Programmierung” [Gamma und Weinand, 1996]; “Why Reuse has Failed” [Schmidt, 1996]; “Components: Another End to the Software Crisis?” [Weinand, 1996]

Alle im folgenden betrachteten Wiederverwendungstechnologien lassen sich unabhängig vom Paradigma der verwendeten Entwurfsmethode oder der Sprache einsetzen. Die vielversprechendsten Technologien sind, wie noch gezeigt wird, auf einer sehr hohen Abstraktionsstufe positioniert. Auf dieser Stufe sind gute Architekturen und integrierte Konzepte zur Wiederverwendung und Flexibilität aber wichtiger als ein bestimmtes Paradigma.

Auf der anderen Seite bemerkte bereits Niklaus Wirth, “... müssen wir den unleugbaren, starken Einfluss, den unsere Sprache auf unsere Art zu denken ausübt und den abstrakten Raum begrenzt, in dem wir unsere Gedanken formulieren” berücksichtigen. Die (Programmier-)Sprache und das verwendete Paradigma sind wesentliches Hilfsmittel⁶, um die Architektur und schliesslich die fertige Applikation zu beschreiben. Der Reichtum und die Ausdrucksmöglichkeiten dieser Sprache haben einen wesentlichen Einfluss auf die Flexibilität, Wartbarkeit und Wiederverwendbarkeit der damit erzielbaren Lösungen.

In experimentellen Studien, so beispielsweise [Lewis et al., 1991], konnte gezeigt werden, dass mit Hilfe des objektorientierten Paradigmas signifikante Steigerungen der Produktivität möglich sind. Dies wird in der erwähnten Studie vor allem durch die bessere Unterstützung von Wiederverwendungstechniken dieser Sprachen begründet.

Ein wesentliches Sprachelement im objektorientierten Ansatz ist die Vererbung. Aufgrund der Bedeutung der Vererbung für die Wiederverwendung soll hier genauer darauf eingegangen werden.

Vererbung

In [Johnson, 1996] werden verschiedene Motivationen für die Anwendung der Vererbung aufgeführt, im wesentlichen sind dies Implementations- und Schnittstellenvererbung:

- *Wiederverwendung von Code (Implementationsvererbung)*
Objektorientierte Sprachen bieten im Unterschied zu den modulorientierten Sprachen einen wesentlichen Vorteil: Bestehen bei einer modulorientierten Sprache geringfügig andere Anforderungen an ein Modul, so kann nicht das bereits bestehende verwendet oder erweitert werden, sondern es muss ein vollständig neues geschrieben werden. Bei der Verwendung von objektorientierten Sprachen kann hier die Vererbung eingesetzt werden, welche die inkrementelle Erweiterung von Klassen ermöglicht.

⁶Auch andere Autoren, so zum Beispiel [Booch, 1994a] stellen fest, dass der Entwurf einer Anwendung niemals vollständig unabhängig von der zu verwendenden Programmiersprache sein kann.

Mittels Vererbung können also bestehende Klassen erweitert und der Code in diesen Klassen direkt wiederverwendet werden. Die Vererbung und damit diese Art der Wiederverwendung führt aber zu einer starken Bindung zwischen Klassen einer Hierarchie und zu einer Verteilung der relevanten Funktionalität sowie des zugehörigen Quellcodes auf eine Vielzahl von Klassen. Nachträgliche Änderungen in einer Basisklasse können zu grossen Problemen in den abgeleiteten Klassen führen (dies wird auch das *fragile base class problem* genannt).

Der Einsatz der Vererbung mit dem Zweck, Code wiederzuverwenden, ist gut einsetzbar für Aufgaben im Bereich des *Rapid Prototyping*. Hier geht es darum, schnell eine lauffähige Applikation zu haben, ohne sich um Wiederverwendung, Wartbarkeit, etc. zu kümmern. Implementationsvererbung erschwert aber das Verständnis des Systems, führt damit unter anderem bei Applikationen mit langer Lebensdauer zu Wartungsproblemen und resultiert, gesamthaft betrachtet, in schlecht wiederverwendbarer Software! Diese Aussage ist nur auf den ersten Blick überraschend, besteht doch Software aus mehr als nur aus Code. Die Wiederverwendung der Architektur einer Software wird aber stark erschwert, wenn sich die *Struktur des Codes* nicht mit der *Struktur des Designs* deckt. Wird die Vererbung eingesetzt, um einzelne Codeteile wiederzuverwenden, so führt dies dazu, dass Vererbungshierarchien eingeführt werden, die sich nicht aus dem Anwendungsgebiet der Applikation, sondern aus Implementierungsdetails ergeben.

- *Polymorphismus (Schnittstellenvererbung)*

Polymorphismus ist die Eigenschaft einer Variable, Objekte unterschiedlichen Typs referenzieren zu können. In Sprachen mit statischer Typisierung müssen diese Objekte meist aus einer Hierarchie von Klassen erzeugt werden, wobei die polymorphe Variable vom Typ einer gemeinsamen Basisklasse sein muss.

Der Typ einer polymorphen Variable ist meist eine abstrakte Klasse, eine Klasse also, die lediglich die Schnittstelle definiert, aber noch keinerlei Implementierungen liefert. Mit Hilfe der Vererbung werden auf diese Weise Familien von Klassen mit identischen oder ähnlichen Schnittstellen definiert. Diese polymorphen Variablen können beispielsweise verwendet werden, um in Klassen parametrisierbare *Slots* zu definieren. Dies erlaubt, Objekte höchst dynamisch, im Extremfall sogar zur Laufzeit, zu kombinieren.

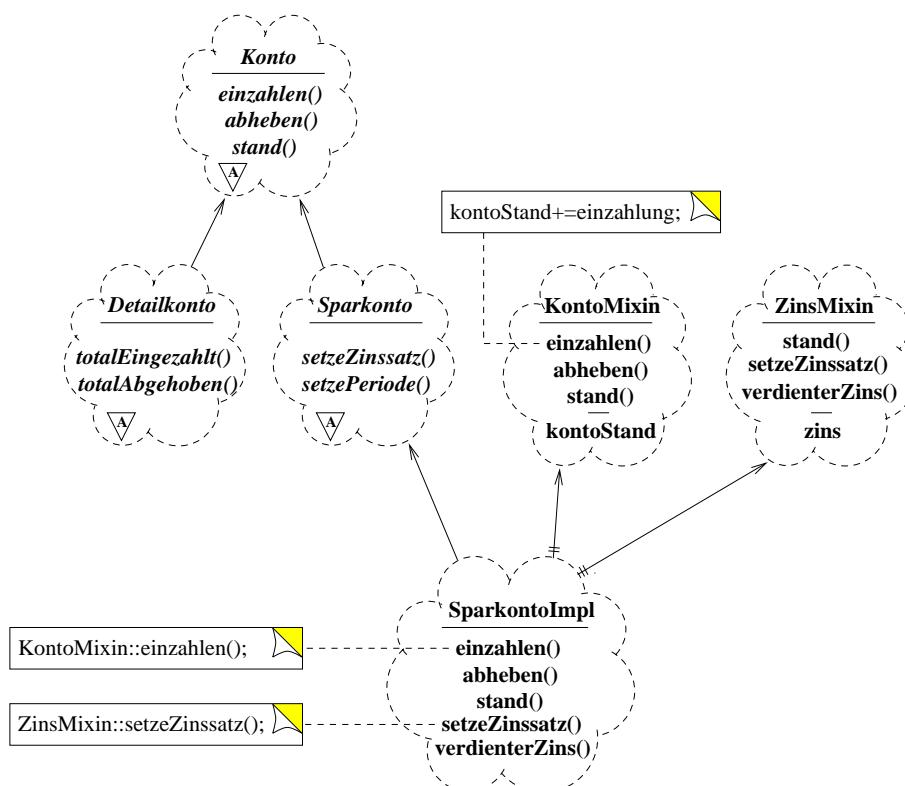
Diese Eigenschaft der Vererbung, den Polymorphismus handhabbar zu machen, nennt [Johnson, 1996] "den wahrscheinlich grössten Beitrag der Vererbung zur Wiederverwendung von Software." Diese

Aussage wird im Laufe der nächsten Abschnitte verdeutlicht werden, da viele der mächtigeren Wiederverwendungstechnologien auf dieser Anwendung der Vererbung aufbauen.

Diese beiden Arten der Vererbung werden in der Regel gleichzeitig in einer Mischform eingesetzt. Aus einer unkontrollierten Anwendung von Vererbung können Systeme entstehen, die äussert schwierig zu verstehen und zu warten sind. [Seidewitz, 1996] schlägt vor, dass, ähnlich wie die strukturierte Programmierung den Verzicht auf `goto's` proklamierte, eine *„strukturierte objektorientierte“* Programmierung die Vererbung nur in einer kontrollierten Form anwenden soll. Seidewitz empfiehlt, Vererbung entweder zur Schnittstellendefinition, zur Verwendung sogenannter *Mixins*⁷ oder zur Implementation anzuwenden. Die Basisklassen einer

Abbildung 2.3

Die abstrakten Klassen (*kursiver Klassename*) definieren die unterschiedlichen Typen von Konten. Mit Hilfe der Mixin-Klassen werden einzelne Methoden implementiert. Die Implementierungsklassen 'mischen' die benötigten Mixins zusammen und werden einem bestimmten Typ zugeordnet. In den Implementierungsklassen muss in der Regel nur noch eine korrekte Methode der entsprechenden Mixin-Klassen aufgerufen werden.



Vererbungshierarchie sollten in der Regel nur abstrakte Klassen sein. In den Blättern der Hierarchie befinden sich die Klassen, welche die weiter oben definierten Methoden implementieren. Mittels Mixin-Klassen und *Multiple Inheritance* kann mehrfach benötigte Funktionalität in einer Klasse definiert und wiederverwendet werden. Abbildung 2.3⁸ zeigt

⁷siehe auch [Brocha und Cook, 1990].

⁸Die Abbildung stellt ein Klassendiagramm in einer leicht angepassten Notation gemäss [Booch, 1994b] dar.

anhand eines einfachen Beispiels aus der Finanzwelt, wie einerseits eine Hierarchie von abstrakten Klassen (*Typen*) aufgebaut wird, in der lediglich Methoden deklariert, aber nicht implementiert werden, andererseits eine Anzahl von Mixin-Klassen zur Verfügung gestellt werden, die das spezielle Fachwissen implementieren. Zur Implementierung einer Klasse muss dann nur noch ein Typ sowie eine Anzahl von Mixins ausgewählt werden.

Der folgende Abschnitt bewertet den Einsatz von Hochsprachen bezüglich der in Abschnitt 2.2 besprochenen Klassifizierung.

Bewertung des Sprachenansatzes

Die wiederverwendbaren Elemente in Hochsprachen sind die Implementierungsmuster (Idioms) der Assemblersprachen oder allgemeiner, der Sprachen vorangehender Generationen. Sprachkonstrukte sind die Abstraktionsspezifikationen für die Muster der darunterliegenden Sprachen. Die folgende Abbildung 2.4 charakterisiert das Wesen der Abstraktion in

Abstraktion

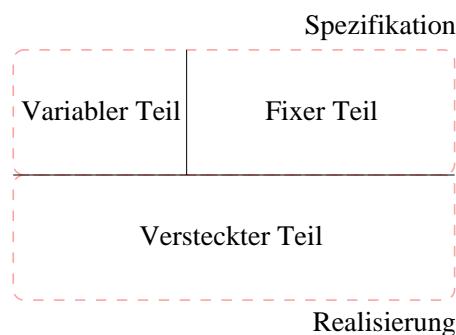


Abbildung 2.4

Die Abstraktionen bei Hochsprachen sind die einzelnen Konstrukte der Hochsprache. Die Realisierung der Abstraktion ist die Umsetzung des Sprachkonstruktes in die Assemblersprache und vollständig versteckt. Die Spezifikation beinhaltet die fixe Semantik sowie die variablen Teile, die mit weiteren Abstraktionen rekursiv gefüllt werden.

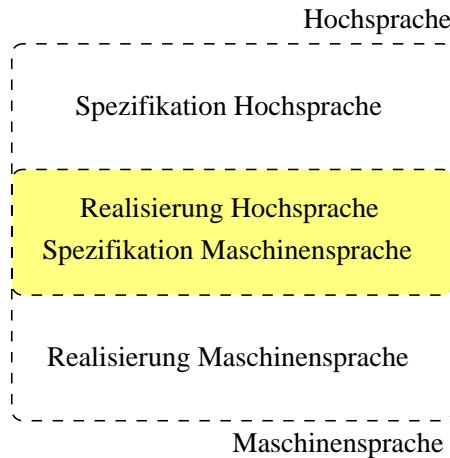
Sprachen. Hochsprachen verstecken die darunterliegenden Abstraktionschichten vollständig. Dies wird in Abbildung 2.5 verdeutlicht.

Die Semantik der einzelnen Konstrukte wird in einem Referenzhandbuch beschrieben. Da die Anzahl der Konstrukte im Allgemeinen sehr klein ist, kann ein Programmierer die Liste der Abstraktionen im Kopf haben, was die Auswahl stark beschleunigt und erleichtert. Ein Programmierer lernt eine neue Hochsprache typischerweise in einigen Tagen oder Wochen, vorausgesetzt, er ist mit dem der Sprache zugrundeliegenden Paradigma vertraut.

Selektion

Typischerweise beinhalten Sprachkonstrukte bestimmte parametrisierbare Positionen. Die Spezialisierung geschieht durch ein rekursives Einfüllen

Spezialisierung

**Abbildung 2.5**

Der Anwender einer Hochsprache wird vollständig von der darunterliegenden Abstraktionsschicht der Maschinensprachen abgeschottet.

von Sprachelementen des passenden Typs in die einzelnen Positionen. Diese Art der Spezialisierung bedarf zwar beim ersten Mal einer ausführlichen Schulung, ist aber im Ganzen betrachtet recht intuitiv, da sie an unsere natürliche Sprache angepasst und sehr mächtig ist.

Integration

Im Referenzhandbuch wird definiert, wie die einzelnen Sprachkonstrukte zusammenarbeiten und welche Auswirkungen sie zur Laufzeit auf Daten und Kontrollfluss haben. Die syntaktisch korrekte Integration wird vom Compiler kontrolliert. Der Compiler übernimmt vollständig die Umsetzung in den Assemblercode sowie die Überprüfung auf syntaktische sowie teilweise semantische Korrektheit. Diese maschinelle Integration stellt einen wesentlichen Vorteil des Konzeptes der Sprachen dar.

Kombinierbarkeit

Es ist in aller Regel kein leichtes Unterfangen, verschiedene Hochsprachen zu kombinieren. Wesentlich überraschender ist aber die Tatsache, dass es teilweise sogar unmöglich ist, Module zu kombinieren, die in derselben Sprache programmiert, aber mit Compilern unterschiedlicher Hersteller übersetzt wurden.

Trotz dieser technischen Schwierigkeiten sind die Hochsprachen das Fundament, auf dem die im folgenden besprochenen, weiterführenden Techniken basieren.

Bewertung

Sprachen können auf den unterschiedlichsten Abstraktionsebenen angetroffen werden. Die unterste Sprachebene bilden die maschinennahen Sprachen, darauf bauen die verschiedenen Sprachen der zweiten bis vierten Generation auf und schliesslich - bereits deutlich näher am Problemkreis - die objektorientierten Sprachen. Wie wir noch sehen werden, lässt sich diese Entwicklung der Sprachen durchaus noch weiter führen.

Höhere Programmiersprachen dienen heute oft als die unterste Abstraktionsebene, die im Laufe eines Softwareentwicklungszyklus betrachtet wird. Der grösste Nachteil von Hochsprachen im Hinblick auf Wiederverwendung ist die grosse kognitive Distanz zum eigentlichen Problem. Bevor die Hochsprache zum Einsatz kommt, muss zuerst ein grosser Aufwand in das Systemdesign investiert werden. Diese Ebene wird von Hochsprachen überhaupt nicht abgedeckt.

Abbildung 2.4 zeigt aber eigentlich das Bild einer idealen Abstraktion; die gesamte Realisierung bleibt versteckt, die Spezifikation unterteilt sich in einen flexiblen und einen variablen Teil. Ein weiterer äusserst positiver Punkt ist die automatisierte Umsetzung der Spezifikation in die Realisierung durch einen Compiler. Sprachen sind somit sehr mächtige Werkzeuge der Wiederverwendung.

Das Abstraktionsvermögen von Sprachen ist stark abhängig von der gewählten Sprache. Während die Sprachen der ersten Generationen eher wenig Abstraktionsmöglichkeiten bieten, beinhalten die objektorientierten Sprachen mit den Konzepten der Vererbung, dem Polymorphismus, der Kapselung und ihrer Fähigkeit, problemnahe Abstraktionen zu beschreiben, eine bereits deutlich verringerte kognitive Distanz. Der Aufwand für die Selektion, Spezialisierung und die Integration ist bei Sprachen generell sehr klein.

Seit der Arbeit von Krueger haben sich die Schwerpunkte im Software Engineering etwas verschoben; einige Technologien sind in den Vordergrund gerückt, andere sind weniger bedeutend geworden. Dieser Tatsache soll dadurch Rechnung getragen werden, dass in den folgenden Abschnitten von den Ausführungen von Krueger abgewichen wird. Die dort behandelten Technologien decken sich zwar nicht ganz mit den hier vorgestellten in Bezug auf Namengebung und Abgrenzung, doch lassen sich die beiden Klassierungen durchaus ineinander überführen.

2.3.2 Bibliotheken

Fast so alt wie Sprachen sind Bibliotheken, die meist für spezielle Anwendungsgebiete entwickelt wurden. Einfache Funktionsbibliotheken stellte sich denn auch [McIlroy, 1968] vor, als er von Komponenten⁹ sprach. Beim Design von Funktionsbibliotheken werden die einzelnen Funktionen ganz gezielt im Hinblick auf die einfache Wiederverwendbarkeit hin entwickelt. Oftmals beinhalten Funktionsbibliotheken Elemente, die nicht

⁹Sein Anliegen, das er an der NATO Konferenz vortrug, ging allerdings über die bloße Existenz solcher Funktionsbibliotheken hinaus. Für eine ausführlichere Besprechung dieser Arbeit sei auf Abschnitt 2.5 hingewiesen.

für sich selbst verwendet werden, sondern von den einzelnen Funktionen gemeinsam genutzt werden, beispielsweise spezielle Mechanismen zur Behandlung von Ausnahmefällen oder zur Speicherverwaltung.

Der Begriff *Bibliothek* wird, je nach Zusammenhang, unterschiedlich verwendet. Einerseits bezeichnet er auf *physischer* Stufe die Zusammenfassung mehrerer (Objekt- oder Quellcode-) Dateien zu einer Einheit. Eine andere oft verwendete Bezeichnung hierfür ist *Archiv*. Diese Sicht einer Bibliothek führt dazu, dass beispielsweise die Menge der Frameworks¹⁰ als Teilmenge der Bibliotheken betrachtet wird, da die zu einem Framework gehörenden Klassen physisch als Archiv zur Verfügung gestellt werden. Andererseits wird der Begriff verwendet, um auf *logischer* Stufe eine bestimmte Architektur (die Elemente einer Bibliothek hängen in der Regel nur sehr lose zusammen und spielen passive Rollen innerhalb der Applikation) zu bezeichnen (vgl. Abbildung 2.7). Im Rahmen dieser Arbeit soll der Begriff *Bibliothek*¹¹ ausschliesslich in dieser zweiten Bedeutung verwendet werden.

Bibliotheken haben sich als sehr nützlich in eng begrenzten Anwendungsbereichen erwiesen. Es gibt eine Vielzahl von Bibliotheken auf dem kommerziellen und dem *Public-Domain*-Markt, die sich beispielsweise auf mathematische Berechnungen, Dateihandhabung, Datenbanken, Benutzeroberflächen oder auch Grafikanwendungen konzentrieren.

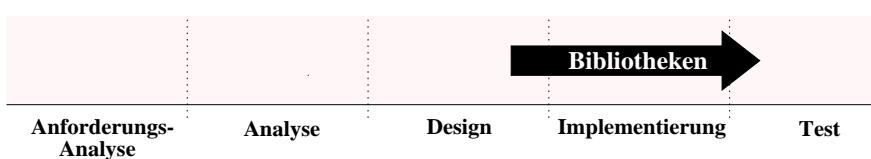
Bibliotheken werden mit Hilfe einer bestimmten Hochsprache realisiert und sind bei der Anwendung meist auch an diese gebunden. Das Zusammenfügen mehrerer Bibliotheken in einer Anwendung ist durchaus üblich, muss aber keinesfalls in jedem Fall gelingen!

Der Unterschied zwischen Klassen- und Funktionsbibliotheken ist aus struktureller Sicht klein. Er manifestiert sich in den verwendeten Abstraktionen (Klassen oder Funktionen) sowie den verwendeten Sprachparadigmen. Aufgrund des grösseren Abstraktionsvermögens von objektorientierten Sprachen sind allerdings auch die Klassenbibliotheken auf einer wesentlich höheren, d.h. problemnäheren Abstraktionsebene anzusiedeln.

Die Verwendung von Bibliotheken hat einen recht kleinen Einfluss auf den gesamten Entwicklungszeitraum einer Anwendung, kommt die Bibliothek doch frühestens in der späten Designphase zum Einsatz (Abbildung 2.6). Abhängig von der Granularität der Bibliothek können auch bereits gewisse Designüberlegung (auf sehr tiefer Stufe) in den in der Bibliothek enthaltenen Abstraktionen gekapselt sein.

¹⁰Für die Definition des Begriffes Framework sowie eine ausführliche Besprechung sei auf Abschnitt 2.4.2 verwiesen.

¹¹Andere Autoren verwenden Bibliothek als Oberbegriff; so wird zum Beispiel in [Metz, 1995; Gamma, 1992] zwischen Bausteinklassen und Frameworks (vgl. Abschnitt 2.4.2) unterschieden und diese beiden Begriffe unter dem Namen *Klassenbibliothek* subsumiert.

**Abbildung 2.6**

Bibliotheken können helfen, die Implementierungsphase drastisch zu verkürzen. Kleinere Auswirkungen sind auch auf die Design- sowie die Testphase zu erwarten.

Der späte Einsatz einer Bibliothek kommt auch zum Ausdruck, wenn die Architektur einer Anwendung analysiert wird, die einzig auf Bibliotheken als Wiederverwendungsbausteine aufbaut. Wie Abbildung 2.7 zu entnehmen ist, muss ein recht grosser Aufwand in die Entwicklung der Anwendung investiert werden. Die gesamte Ablauflogik ist in der Anwendung enthalten. Eine Bibliothek dient 'nur' dazu, in der Regel primitive und voneinander unabhängige Funktionen, Datentypen oder Klassen zur Verfügung zu stellen. Es wird keinerlei Hilfe für das Design der Anwendung gegeben.

**Abbildung 2.7**

Eine Bibliothek dient lediglich der Wiederverwendung von Programmcode. Die gesamte Ablauflogik muss in jeder Anwendung von Grund auf neu entwickelt werden.

Die folgende Bewertung gemäss Abschnitt 2.2 schliesst die Betrachtungen über die Bibliotheken als Wiederverwendungstechnologie ab.

Bewertung des Bibliotheksansatzes

Die wiederverwendbaren Elemente der Funktionsbibliotheken sind die eigens zu diesem Zweck entwickelten Funktionen. Oftmals müssen mehrere dieser Funktionen nacheinander verwendet werden, um eine Aufgabe zu erfüllen. Als Beispiel sei hier das Öffnen, Beschreiben und Schliessen einer Datei erwähnt. Hier müssen drei unterschiedliche Funktionen aufgerufen werden, wobei der Anwender für die korrekte Übergabe der internen Statusinformation (wie dem *Filehandle*) besorgt sein muss.

Abstraktion

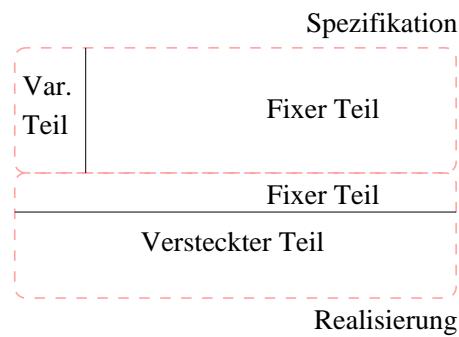
Klassenbibliotheken bieten in vielen Fällen bessere Abstraktionsmöglichkeiten an, da hier die internen Statusinformationen von zusammengehörigen Funktionen gemeinsam mit diesen in einem Objekt gekapselt werden können, so dass der Anwender von unnötigem Wissen entlastet wird.

Das Abstraktionsvermögen von Funktionsbibliotheken ist nicht sehr hoch, da Interna freigelegt werden müssen. Vergleicht man Abbildung 2.8

Abbildung 2.8

Die einzelnen Funktionen oder Klassen in der Bibliothek sind die Abstraktionen dieser Technologie. Die Realisation ist die Implementierung der einzelnen Elemente sowie deren Zusammenspiel.

Die Spezifikation beinhaltet die fixe Aufrufsyntax und die Semantik; der variable Teil ist auf die Übergabe von Parameter und Resultaten beschränkt.



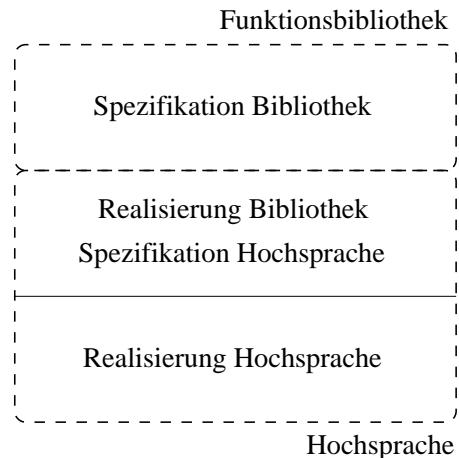
mit Abbildung 2.4, so sieht man, dass hier die Realisation nicht mehr vollständig versteckt werden kann. Außerdem ist der variable Teil einiges kleiner, er beschränkt sich bei den Funktionsbibliotheken im wesentlichen auf die einfache Übergabe von Parametern und Funktionsresultaten.

Mit Hilfe von Klassenbibliotheken lassen sich die Interna im Allgemeinen wesentlich besser verstecken als mit Funktionsbibliotheken. Klassenbibliotheken sind also auf einem höheren Abstraktionsniveau anzusiedeln, unterscheiden sich in ihrer Struktur aber nicht von Funktionsbibliotheken.

Bibliotheken können als Erweiterungen der Sprache betrachtet werden. Sie bilden allerdings eine durchlässige Abstraktionsschicht (Abbildung 2.9).

Abbildung 2.9

Funktionsbibliotheken werden in einer Hochsprache realisiert und erweitern scheinbar den Sprachumfang. Im Gegensatz zum vorangehenden Beispiel (vgl. Abbildung 2.5) wird die darunterliegende Abstraktionsstufe nicht gekapselt.



Selektion

Die einzelnen Elemente einer Bibliothek werden üblicherweise in Handbüchern festgehalten, die sehr umfangreich werden können. Der einzelne Entwickler ist normalerweise nicht mehr in der Lage, alle Funktionen einer umfangreichen Bibliothek auswendig zu lernen, so dass eine

Nachschlagemöglichkeit Voraussetzung zum Arbeiten mit Bibliotheken wird.

Die Spezialisierung beschränkt sich bei Bibliotheken darauf, die richtigen Parameter an die einzelnen Funktionen zu übergeben. Bei Klassenbibliotheken lässt sich ausserdem die Vererbung als Spezialisierungsinstrument einsetzen, indem von Bibliotheksklassen Unterklassen gebildet und bestehende Methoden überschrieben werden. Dabei muss allerdings beachtet werden, dass es in der Regel nicht gelingt, Unterklassen von Klassen zu bilden, die nicht im Hinblick auf diese Spezialisierungsart entwickelt wurden.

Spezialisierung

Das Referenzhandbuch definiert, welche Vorbedingungen erfüllt sein müssen, bevor eine Funktion aufgerufen werden kann. Beispielsweise muss eine Datei zuerst geöffnet werden, bevor Werte hineingeschrieben werden können. Während der Compiler der Hochsprache die Kontrolle der Aufrufsyntax übernimmt, gibt es in der Regel kein Hilfsmittel, um die Korrektheit der Vorbedingungen der einzelnen Funktionen zu überprüfen. Minimale Prüfungen können von den einzelnen Funktionen zur Laufzeit vorgenommen werden.

Integration

Gut implementierte *Klassenbibliotheken* können diesen Nachteil etwas ausgleichen, da die Möglichkeiten der Kapselung und der internen Statusinformation eines Objektes gewisse Plausibilitäts- und Korrektheitsprüfungen erlauben.

Normalerweise ist es nur selten möglich, Funktionsbibliotheken, die in unterschiedlichen Hochsprachen realisiert wurden, zu kombinieren. Andererseits gelingt es aber oft, Bibliotheken unterschiedlicher Hersteller zu kombinieren. Es ist sogar der Normalfall, dass in einem grösseren System mehrere auf bestimmte Aufgaben spezialisierte Bibliotheken in Kombination eingesetzt werden.

Kombinierbarkeit

Bei der Verwendung von Bibliotheken muss bei Sprachen, die keine anwenderdefinierbaren Namensräume zur Verfügung stellen, mittels geeigneter Vorkehrung (wie beispielsweise Präfixe für globale Namen) dafür gesorgt werden, dass keine *Namenskonflikte* auftreten. Haben Hersteller diese einfachen Grundregeln verletzt, kann dies den Einsatz einer bestimmten Kombination von Bibliotheken verunmöglichen.

Bibliotheken sind eine wichtige Wiederverwendungstechnologie, die direkt auf den Sprachen aufsetzt und deren Einsetzbarkeit wesentlich verbessert, da Lösungen zu Standardproblemen (File-I/O, Datenbanken, Graphische Benutzeroberflächen, etc.) direkt als Bibliotheken erworben werden können. Bibliotheken werden zudem oft mit einer Sprache zusammen

Bewertung

entwickelt (Standardbibliotheken), um den Sprachumfang so klein wie möglich zu halten. Solche Sprachen beinhalten dann nur noch Kontroll- und Deklarationselemente. Alle funktionalen Teile werden in die Bibliothek ausgelagert.

Die kognitive Distanz der Funktionsbibliotheken ist immer noch recht gross. Funktionsbibliotheken kommen, wie die Hochsprachen, erst in der letzten Phase der Systementwicklung zum Zuge und haben nur einen kleinen Einfluss auf die Designphase. Die Designphase kann verkürzt werden, wenn man weiss, dass eine Funktionsbibliothek zum Beispiel für den Datenbankzugriff verwendet werden soll, anstatt alles selbst neu zu entwickeln. Der Einsatz einer Funktionsbibliothek entbindet aber nur vom Entwerfen, Schreiben und Testen von *einzelnen Funktionen*, der Entwurf der gesamten Applikationslogik bleibt dem Entwickler der Applikation überlassen.

Wie bereits bei den Sprachen erwähnt, kann die kognitive Distanz durch den Einsatz des objektorientierten Paradigmas etwas verringert werden.

Mit der Besprechung von Hochsprachen und Bibliotheken wird hier die einfache Wiederverwendung von Implementation beendet. Der nächste Abschnitt ist der viel mächtigeren Wiederverwendung von Architektur oder Architekturbestandteilen gewidmet.

2.4 Architekturwiederverwendung

Mit zunehmender Grösse und Komplexität von Softwaresystemen wird der Entwurf und die Spezifikation der *Struktur* des gesamten Systems zu den zentralen Herausforderungen, während die Wahl der einzelnen Algorithmen und Datenstrukturen immer mehr in den Hintergrund rückt.

*“[Software Architecture is] what software
architects do”*

Kent Beck, in
[Gamma und Weinand, 1996]

Diese pointierte Formulierung ist natürlich keine sinnvolle Grundlage für eine Diskussion über die Wiederverwendung von Design (und vom Autor auch nicht als solche beabsichtigt). [Shaw und Garlan, 1996] beschreiben den Begriff ‘Software Architektur’ wie folgt:

Die Architektur von Software beinhaltet die Beschreibung der Elemente, aus denen Systeme aufgebaut werden, der Interaktionen zwischen diesen Elementen, der Muster, die deren Zusammenspiel koordinieren sowie allfällige Einschränkungen dieser Muster. Beispiele für strukturelle Fragestellungen sind: die Art, wie die einzelnen Systemteile organisiert werden, die globale Kontrollstruktur, die verwendeten Protokolle zur Kommunikation, Synchronisation und Datenzugriff, die Zuordnung von Funktionalität zu den einzelnen Entwurfselementen, die Zusammensetzung von Entwurfselementen, die physische Verteilung, Skalierbarkeit, Leistungsfähigkeit und Möglichkeiten der Weiterentwicklung.

Obwohl die im vorhergehenden Abschnitt besprochene Wiederverwendung von Produkten der Implementierungsphase unbestreitbar grosse Vorteile bringt, sind hier nur beschränkte Einsparungen an Aufwand möglich. Der Grund liegt in der Tatsache, dass die Implementierungsphase im gesamten Softwareentwicklungszyklus nur einen kleinen Anteil einnimmt. Eine massive Effizienzsteigerung kann erreicht werden, wenn eine erfolgreiche Wiederverwendung in *frühen* Phasen möglich wird.

Eine bereits seit längerer Zeit verwendete Form der Wiederverwendung sind Applikationsgeneratoren. Aufgrund einer abstrakten Problembeschreibung wird Quellcode erzeugt, der anschliessend nachbearbeitet oder ergänzt werden kann. Zwei sehr bekannte Vertreter dieser Technologie sind die Compilergeneratoren LEX und YACC

Generatoren

[Schreiner und Friedman, 1985]. Hier werden aufgrund von Regeln Module für die lexikalische Analyse und das Parsing von Sprachen erzeugt.

Applikationsgeneratoren sind eine sehr praktische und attraktive Variante, wenn eine automatische Übersetzung einer abstrakten Beschreibung in Quellcode möglich ist. In diesem Fall können viele der sonst nötigen Schritte des Softwareentwicklungsprozesses übersprungen werden, wodurch eine signifikante Beschleunigung möglich wird. Leider eignen sich nur wenige Gebiete für den Einsatz von Generatoren.

Eine weitere, spezielle Art der Wiederverwendung ist in der Wartungsphase zu finden [Johnson und Foote, 1988]. Wird eine bestehende Anwendung an neue Aufgaben angepasst, so kann dies als eine Wiederverwendung der ganzen Applikation betrachtet werden.

Die folgenden Abschnitte besprechen neuere Strömungen des Software Engineerings, die ebenfalls die Wiederverwendung in frühen Phasen der Softwareentwicklung zum Anliegen haben. Die in den vorausgegangenen Abschnitten besprochenen Techniken unterstützen die Wiederverwendung *architekturneutraler Softwareelemente*; der Einsatz einer bestimmten Sprache oder Bibliothek beeinflusst im Allgemeinen die Architektur eines Systems nicht unmittelbar. Im Folgenden werden Techniken vorgestellt, welche eine *architektspezifische* Wiederverwendung erlauben.

2.4.1 Design Patterns

In seiner Doktorarbeit (in [Gamma, 1992] veröffentlicht) an der Universität Zürich unter der Leitung von Prof. Dr. R. Marty, hat Erich Gamma den Grundstein zu einer Bewegung gelegt, die heute in einer wahren Flut von Konferenzen und Publikationen zu diesem Thema mündet, wie [Buschmann et al., 1996; Vlissides et al., 1996; Coplien und Schmidt, 1995; Coad et al., 1995; Pree, 1995], um nur einige zu nennen. Obwohl Erich Gamma Design Patterns¹² bereits in seiner Dissertation erwähnte und kurz darauf in einer Publikation [Gamma et al., 1993] beschrieb, wurde diesem Teilgebiet der objektorientierten Modellierung erst mit der Veröffentlichung von [Gamma et al., 1995] die gebührende breite Aufmerksamkeit¹³ zuteil.

Vor Gamma haben aber bereits andere Autoren ähnliche Problemkreise angesprochen. Während in [Shaw, 1990] die Forderung nach einem Katalog von bewährten Entwürfen formuliert wurde, erkannten

¹²Obwohl im deutschen Sprachraum der Begriff Entwurfsmuster immer mehr Verwendung findet, wird in dieser Arbeit der weiter verbreitete englische Begriff benutzt.

¹³Diese Aufmerksamkeit konnte nicht zuletzt durch eine von Gamma mitbegründete Strategie der frühen Verbreitung einer *Vorabversion* des Buches über das Internet stark erhöht werden.

[Johnson und Russo, 1991] *abstrakte Klassen* als wichtige Elemente flexibler Entwürfe. Die abstrakten Klassen werden dort als ideale Bausteine zum Entwurf von Frameworks bezeichnet. Aufgrund der Beschreibungen und der aufgeführten Beispiele (die sich teilweise direkt mit einzelnen Design Patterns aus [Gamma et al., 1995] decken) können die von Johnson und Russo beschriebenen abstrakten Klassen als direkte Vorläufer der Design Patterns betrachtet werden.

Worum geht es nun aber bei diesem neuen Schlagwort der 90er Jahre? Wie [Buschmann, 1996] ausführt, wird das Verständnis von Patterns erleichtert, wenn man sich die zugrundeliegenden Ideen dieser neuen Disziplin des Software Engineering vergegenwärtigt:

Definition

Wenn Experten ein bestimmtes Problem zu lösen versuchen, so wäre es eher ungewöhnlich, wenn sie eine neue Lösung erfinden würden, die sich völlig von allen existierenden Systemen unterscheidet. Im Normalfall werden sie sich an ähnliche, bereits gelöste Probleme erinnern, und versuchen, deren Quintessenz zur Lösungen heranzuziehen.

Dieses Expertenverhalten, das Denken in *Problem/Lösungs-Paaren*, ist keinesfalls nur in der Informatik anzutreffen, sondern vielmehr in vielen Disziplinen verbreitet. Ein besonders bekanntes Beispiel ist die Baukunst¹⁴. In diesem Gebiet wurden des öfteren Anstrengungen unternommen, Musterkataloge aufzustellen. Der Architekt Christopher Alexander [Alexander, 1979] ist das in diesem Zusammenhang am häufigsten zitierte Beispiel.

Experten im Bereich des Software Engineerings kennen eine Vielzahl an Mustern, die teilweise in unterschiedlichen Phasen des Softwarelebenszyklus zum Einsatz kommen. So werden beispielsweise in [Buschmann et al., 1996] Architektur, Design und Implementierungs Patterns beschrieben, wobei die letztgenannte Kategorie als *Idioms* bezeichnet wird. Idioms für C++ wurden bereits einige Zeit vor dem Design Patterns-Boom in [Coplien, 1992] beschrieben.

Design Patterns dienen dazu, dieses vorhandene Expertenwissen in einer Form festzuhalten, die es auch weniger erfahrenen Softwareentwicklern erlaubt, auf einen reichen Erfahrungsschatz zurückzugreifen. Die folgenden, positiven Eigenschaften untermauern diese Absicht:

Eigenschaften

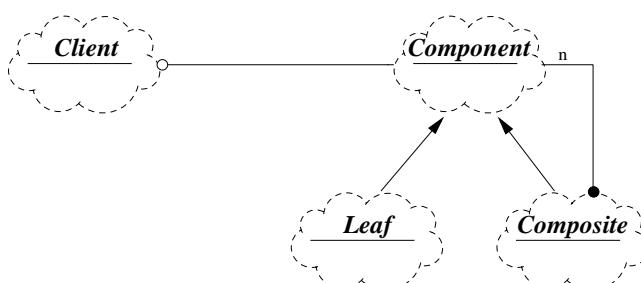
¹⁴Die Verwendung des üblichen Begriffes *Architektur* würde im gegebenen Kontext Anlass zu Verwirrung geben. Gemeint ist hier die althergebrachte Architektur, die sich mit dem Bau von Gebäuden beschäftigt, nicht die Softwarearchitektur.

1. Patterns dokumentieren existierende, bewährte Designerfahrung und tragen somit wesentlich zu einer Wiederverwendung bewährter Konzepte bei. Sie werden nicht erfunden oder künstlich erzeugt, sondern werden vielmehr „destilliert und ermöglichen die Wiederverwendung von Designkenntnissen, die von erfahrenen Praktikern gesammelt wurden“. Wer mit einem ausreichend grossen Satz von Patterns vertraut ist, „kann sie direkt auf Designprobleme anwenden, ohne sie erneut entdecken zu müssen“ [Gamma et al., 1993].
2. Patterns sind ein Mittel, um Softwarearchitekturen zu dokumentieren (vgl. auch Abschnitt 7.2.1). Sie beschreiben die Visionen des Architekten, der ein Softwaresystem entwirft. Müssen zu einem späteren Zeitpunkt Änderungen am System vorgenommen werden, sei dies zur Behebung von Fehlern oder zur Erweiterung der Funktionalität, so können die damit betrauten Entwickler diese ursprünglichen Visionen teilen und das System in einem in sich konsistenten Zustand behalten.
3. Patterns eröffnen den Softwarearchitekten ein gemeinsames Vokabular sowie ein breites Verständnis für Designprinzipien. Die Namen der Patterns werden, sofern sorgfältig gewählt, Teile einer weitverbreiteten Entwurfssprache und erleichtern somit die effiziente Diskussion über Designprobleme und deren Lösungen.

Abbildung 2.10 zeigt als Beispiel eines einfachen und oft eingesetzten Design Patterns das Klassendiagramm des *Composite-Patterns* aus [Gamma et al., 1995]. *Composites* werden verwendet, um Objekte in baumartige Strukturen zu verpacken. *Composites* erlauben es Anwenderklassen, atomare und zusammengesetzte Objekte gleich zu behandeln.

Abbildung 2.10

Design Patterns sind von einer konkreten Anwendung abstrahierte Problem/Lösungs-Paare, die vor einem konkreten Einsatz an die jeweiligen Anforderungen angepasst werden müssen. Diese Abbildung zeigt als Beispiel das *Composite*-Pattern [Gamma et al., 1995, Seite 163].



Design Patterns haben über diese besprochenen Punkte hinaus einen weiteren, nachhaltigen Einfluss auf den Entwurf objektorientierter Systeme. Eine Gemeinsamkeit der meisten Patterns ist die Flexibilität, die sie

in den Entwurf einbringen. Diese Flexibilität ist oft darauf ausgerichtet, gewisse Aspekte eines Systems unabhängig von anderen Systemteilen verändern zu können. Der Einsatz von Patterns erhöht somit die Flexibilität der Software. Dies setzt aber voraus, sich in der Designphase Gedanken darüber zu machen, welche Teile eines Systems variabel sein sollen. Diese Art des Designs wird in [Gamma et al., 1993] als *Variation-Oriented Design* bezeichnet. [Pree, 1995] verwendet den Begriff *Hot-Spots* für diese flexiblen Teile des Designs und *Hot-Spot Driven Design* für seinen Designansatz.

Die erwähnte Flexibilität wird in der Regel durch den intensiven Einsatz von Vererbung zusammen mit weiteren wichtigen Konzepten der objektorientierten Technologien erreicht. Eine Grundregel in diesem Zusammenhang lautet gemäss James Coplien:

All interesting problems in computer science may be reduced to “what’s in a name” and may be solved by one more level of indirection.

[Coplien, 1995]

Wenngleich diese Aussage etwas pointiert formuliert erscheint¹⁵, so ist sie für den Alltag eines Softwareentwicklers doch sehr zutreffend. Tatsächlich lassen sich die meisten Design Patterns auf diese einfache Grundregel reduzieren. Die Vererbung, Hand in Hand mit den Konzepten des Polymorphismus und der dynamischen Bindung¹⁶, bildet die technische Grundlage zur Umsetzung dieser Flexibilität. Die Reduktion auf die Frage *“what’s in a name”* weist auf die zentrale Rolle der richtigen Wahl der Abstraktion hin, die durch einen Namen bezeichnet wird. Die adäquate Wahl einer Abstraktion ist zentral für die Lösung von Informatikproblemen. Kann ein bestimmtes Problem mit einer getroffenen Auswahl von Abstraktionen nicht gelöst werden, so hilft es sehr oft, zusätzliche Abstraktionsschichten einzufügen (...*one more level of indirection*).

In einer weiteren Analogie zur Architektur (Abbildung 2.11) stellt [Gamma, 1996] die Schichtenarchitektur von Gebäuden vor. Gebäude sind in der Regel *‘anpassungsfähig’*, das heisst sie können im Laufe der Zeit oft erstaunlich einfach an neue Anforderungen angepasst werden. Dies ist möglich, da Elemente mit unterschiedlichen Änderungsraten voneinander

¹⁵Ein sehr interessantes Problem der Informatik ist beispielsweise das der NP-Vollständigkeit. Die Lösung dieses Problemes dürfte aber schwerlich in einer zusätzlichen Indirektionsstufe zu finden sein.

¹⁶Mit dynamischer Bindung wird die Fähigkeit objektorientierter Sprachen bezeichnet, die Bindung von Methodenname zum aufgerufenen Code nicht wie üblich zur *Compilierzeit*, sondern zur *Laufzeit* vorzunehmen. Dies erlaubt, unterschiedliche Implementierungen unter demselben Namen zur Verfügung zu stellen, wobei die Auswahl der Implementierung anhand des Objekttyps des Nachrichtenempfängers geschieht.

Die Schichten, von aussen nach innen:

<i>Hülle</i>	Verputz, Holzverkleidung
<i>Struktur</i>	Aussenmauern, tragende Wände
<i>Dienste</i>	Strom, Wasser, Gas
<i>Raumaufteilung</i>	Innenwände, Fenster, Decken
<i>Einrichtung</i>	Möbel, Lampen, Geräte

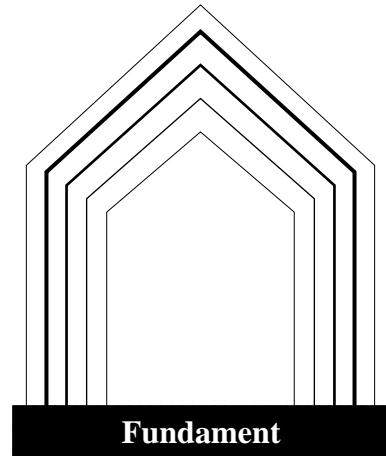


Abbildung 2.11

Ein Haus besteht aus mehreren Schichten, die *Schlupf* gegeneinander haben. Elemente mit unterschiedlichen Änderungsraten werden verschiedenen Schichten zugeordnet.

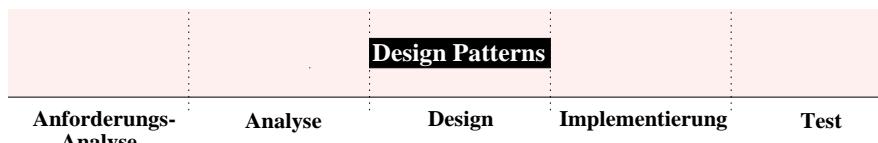
getrennten Schichten zugeordnet werden. Die Schichten müssen so aufeinander aufbauen, dass zwei benachbarte Schichten *Schlupf* haben, d.h. eine Schicht kann – bis zu einem gewissen Punkt – verändert werden, ohne die benachbarten Schichten modifizieren zu müssen.

In derselben Art können Softwarearchitekten Systeme entwickeln, die robust gegenüber sich ändernden Anforderungen sind. Einzelne Teile eines Systems sind für gewisse Teilaufgaben verantwortlich und sollten unempfindlich sein gegenüber Änderungen in anderen Systemteilen. Design Patterns können einen wesentlichen Teil zu der hierfür nötigen Flexibilität beitragen.

Die Auswirkung auf die anderen Phasen der Softwareentwicklung (Abbildung 2.12) bleibt gering. Design Patterns können aufgrund ihres generalisierten Charakters nicht in Funktions- oder Klassenbibliotheken

Abbildung 2.12

Design Patterns kommen ausschliesslich in der Design Phase zum Einsatz. Architektur Patterns, Idioms oder Dokumentationsmuster können in anderen Phasen zum Einsatz kommen.



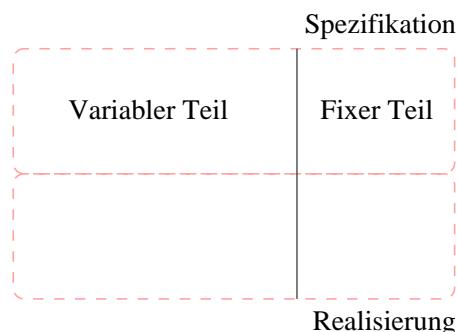
zur Verfügung gestellt werden. Abschnitt 2.4.2 wird eine weitere Wiederverwendungstechnologie betrachten, die durch Einschränkung auf ein bestimmtes Anwendungsgebiet den Einsatzbereich massiv ausdehnen kann.

Der folgende Abschnitt bespricht Design Patterns wiederum hinsichtlich des Klassifikationsschemas nach Krueger (vgl. Abschnitt 2.2).

Bewertung des Patternansatzes

Design Patterns sind von einem bestimmten Problem abstrahierte, immer wiederkehrende Arten des Zusammenspiels einzelner Klassen innerhalb von Systemen. Sie werden deshalb auch als *Mikroarchitekturen* (vgl. z. Bsp. [Gamma, 1996]) bezeichnet. Diese Mikroarchitekturen sind die wiederverwendbaren Elemente.

Design Patterns bieten den Vorteil eines gemeinsamen, mächtigen Vokabulars, das es Softwarearchitekten erlaubt, effizient über den Aufbau und das Design von Systemen zu diskutieren und dieses zu beschreiben. Andererseits bieten Design Patterns keinerlei Wiederverwendung bei der Implementierung. Die Realisierung dieser Abstraktion ist vollständig offen gelegt (vgl. Abbildung 2.13) und muss bei jeder Anwendung erneut angepasst werden.



Abstraktion

Abbildung 2.13

Die Abstraktionen sind die einzelnen Patterns. Die Realisation ist völlig offen gelegt; die Patterns müssen an den Problembereich angepasst werden. Der fixe Teil entspricht den zu unterstützenden Protokollen. Durch die nötige Anpassung wird der variable Teil sehr gross.

Es gibt mittlerweile eine Vielzahl von Publikationen, die allgemeine Patterns oder solche für spezielle Anwendungsgebiete beschreiben. Es ist fast unmöglich, hier den Überblick zu behalten. Es muss allerdings angemerkt werden, dass in einem bestimmten Anwendungsgebiet meist nur eine beschränkte Auswahl von Patterns zum Einsatz kommt. Diese Anzahl bewegt sich normalerweise im Bereich von einigen Dutzend Mustern, was durchaus im Rahmen des Zumutbaren liegt. Um ein Pattern einzusetzen zu können, muss man sich vorher intensiv damit auseinandergesetzt haben. Entweder hat man das Pattern selbst bereits einige Male verwendet oder eine Beschreibung des Patterns sehr aufmerksam studiert.

Design Patterns müssen jeweils an den Anwendungsbereich angepasst werden. Dies geschieht in der Regel durch Umbenennen der Elemente der Patterns (Klassen, Methoden) sowie durch eine Erweiterung der Klassen um anwendungsspezifische Protokolle. Des Weiteren gibt es für die meisten Patterns Variationen, die je nach den Erfordernissen der Anwendung hinsichtlich Laufzeit und Speicherplatz ausgewählt werden müssen. Viele Patterns sind sprachunabhängig formuliert. Somit müssen bei deren

Selektion

Spezialisierung

Umsetzung die Besonderheiten der gewählten Implementierungssprache berücksichtigt werden.

Integration

Design Patterns unterscheiden sich von den anderen in dieser Arbeit vorgestellten Wiederverwendungstechnologien darin, dass hier keinerlei Code wiederverwendet wird. Zudem ist eine automatisierte Umsetzung von Design Patterns in tieferliegende Abstraktionen, wie besprochen, nicht realisierbar. Somit verbleibt die Integration der Design Patterns in der Verantwortung des Entwicklers.

Kombinierbarkeit

Aufgrund der hohen Abstraktionsstufe, auf der Design Patterns beschrieben und angewendet werden, ergeben sich in der Regel keine Probleme bei der freien Kombination. Die Umsetzung in bestimmte Sprachen kann sich aber aufwendiger gestalten und bedarf in der Regel *überdurchschnittlicher Kenntnisse* der gewählten Implementierungssprache.

Bewertung

Insgesamt betrachtet stellen Design Patterns einen sehr wertvollen Beitrag zum Software Engineering dar. Das gemeinsame Vokabular sowie der Zugriff auf den Erfahrungsschatz unzähliger Experten bewirken oft eine angepasstere und flexiblere Architektur eines Systems. Obwohl mit Design Patterns keine unmittelbare Wiederverwendung von Code einhergeht, kann eine flexible Architektur dazu führen, dass dieselbe Anwendung mehr als nur einmal eingesetzt werden kann. Konfiguration tritt in diesem Fall an Stelle von Neuimplementierung. So betrachtet bergen Design Patterns ein beträchtliches Wiederverwendungspotential.

Andererseits ist aber auch eine gewisse Gefahr mit Patterns verbunden:

*Ich habe alle GOF¹⁷ Patterns bis auf eines
in meiner Applikation einbauen können; ich
bin gerade daran, auch noch für das letzte
eine Anwendung zu finden.*

Ein Usenet-Benutzer

Design Patterns erhöhen zwar die Flexibilität, gleichzeitig aber auch die Komplexität eines Systems. Wenn Design Patterns den grösstmöglichen Nutzen bringen sollen, so müssen sie sparsam dort eingesetzt werden, wo die Vorteile eines bestimmten Patterns dessen Nachteile aufwiegen. Einfachheit sollte stets das oberste Ziel beim Entwurf einer Anwendung sein, Flexibilität darf nur dort eingebracht werden, wo sie auch wirklich gebraucht wird!

¹⁷GOF ist die gängige Abkürzung für “Gang of Four” und bezeichnet die vier Autoren des Buches [Gamma et al., 1995].

Wie bereits in Abschnitt 2.3.1 zum Ausdruck gebracht wurde, kann die Weiterentwicklungen von Programmiersprachen durch ein gutes Verständnis und eine weite Verbreitung von entsprechenden *Coding Patterns* und Design Patterns für existierende Sprachen initiiert werden. Als Beispiel sei das Konzept der *Abstrakten Datentypen* erwähnt. Dieses Konzept kann als Entwurfsmuster für benutzerdefinierte Datentypen betrachtet werden und diente als Grundlage des Klassenkonstruktes in objektorientierten Sprachen. Auch andere, in objektorientierten Sprachen verfügbare Konstrukte wurden bereits unter dem prozeduralen Paradigma als wiederkehrendes Muster eingesetzt.

Patterns können somit als Hilfsmittel betrachtet werden, neue Konzepte ausführlich mit existierenden Sprachen und Paradigmen zu testen, bevor die erfolgversprechendsten in neue Ansätze auf höherer Abstraktionsstufe einfließen.

2.4.2 Frameworks

Während Bibliotheken eine gute Wiederverwendung von Code erlauben, dafür aber keinerlei Unterstützung auf der Stufe des Systementwurfs bieten, ist dies bei Design Patterns gerade umgekehrt. Was liegt also näher, als nach einem Weg zu suchen, diese beiden Technologien gewinnbringend zu vereinen? Objektorientierte *Frameworks* sind die Antwort auf diese Frage.

Der Begriff *Framework*^{18,19} wird von verschiedenen Autoren unterschiedlich eng gefasst. [Webster, 1995, Seite 25] beispielsweise definiert:

Definition

Definition 2.2 (Framework, Variante 1)

A framework is a collection of related objects that work together to provide a certain class of functionality.

Diese Definition macht eine Unterscheidung zwischen Klassenbibliotheken und Frameworks noch recht schwierig. Die Definition in [Johnson und Foote, 1988] ist da bereits wesentlich deutlicher:

¹⁸Frameworks existieren bereits seit geraumer Zeit. Durch den Einsatz des objektorientierten Paradigmas sowie Design Patterns haben Frameworks in letzter Zeit an Bedeutung zugenommen. Wenn im folgenden von *Framework* gesprochen wird, so sind stets, sofern nicht anders vermerkt, *objektorientierte* Frameworks gemeint.

¹⁹Oft wird in der deutschen Literatur hier auch der Begriff *Rahmenwerk* verwendet. Der englische Originalbegriff ist aber auch im deutschen Sprachraum noch weiter verbreitet und soll in dieser Arbeit verwendet werden.

Definition 2.3 (Framework, Variante 2)

A framework is a set of classes that embodies an abstract design for solutions to a family of related problems and supports reuses at a larger granularity than classes.

Hier wird also bereits darauf verwiesen, dass ein Framework mehr ist als eine Sammlung von Klassen, dass diese Klassen vielmehr eng miteinander verwoben sind und gemeinsam zur Lösung eines übergeordneten Problems beitragen. Die Art, wie diese Klassen miteinander verwoben sind, ist dabei nicht auf ein *bestimmtes Problem*, sondern auf eine ganze *Familie* von ähnlich gearteten Problemen ausgerichtet. Die Wiederverwendung findet nicht nur, wie bei Klassenbibliotheken üblich, auf der Stufe von einzelnen Klassen statt, sondern das im Framework enthaltene abstrakte Design wird für die einzelnen Probleme spezialisiert. In [Booch, 1994b, Seite 166] wird eine noch etwas genauere Definition eines Frameworks aufgeführt:

Definition 2.4 (Framework, Variante 3)

A framework is a collection of classes that provide a set of services for a particular domain; a framework thus exports a number of individual classes and mechanisms which clients can use or adapt.

Booch verdeutlicht mit dieser Definition, dass nicht die einzelnen Klassen alleine ein Framework ausmachen, sondern die *Mechanismen*, die Objekte dieser Klassen zusammenarbeiten lassen einen mindestens ebenso wichtigen Anteil haben. Mechanismen sind die Art und Weise, in welcher Klassen und Objekte zusammenarbeiten, um ein übergeordnetes Verhalten zu erreichen. Diese Mechanismen bilden zusammen mit den Klassen die *Struktur* eines Softwaresystems, welche auch als *Architektur* bezeichnet wird.

Ein Framework ist die Umsetzung einer bestimmten Architektur zur Lösung einer Familie von Problemen. „Es beschreibt“, wie [Metz, 1995] ausführt, „die globale Struktur, definiert die Zerlegung in Subsysteme, Klassen und Objekte, teilt den Beteiligten bestimmte Verantwortlichkeiten zu, diktiert, wer mit wem zusammenarbeiten muss und bestimmt auch den zeitlichen Ablauf dieser Zusammenarbeit.“

Eigenschaften

Die vorausgegangenen Definitionen decken aber einige wesentliche Eigenschaften von Frameworks noch nicht ab. So führt beispielsweise [Johnson, 1993] folgende zusätzlichen Anforderungen auf:

- *Ein Framework beeinflusst stark, wie ein Problem zerlegt werden muss.*

Ein Framework beinhaltet nicht nur ein abstraktes Design und eine Implementierung der Mechanismen und Basisklassen, sondern auch ein *Vorgehensmodell*, das festlegt, wie dieses Framework in den Prozess der Softwareerstellung einzugliedern ist und wie es diesen beeinflusst. Das Framework gibt dem Anwendungsentwickler das *Konzept* für seine Anwendung vor.

- Ein Framework besteht nicht nur aus einer Sammlung von Klassen, sondern beschreibt auch die Art, wie Instanzen dieser Klassen zusammenarbeiten.

Ein Framework stellt also die übergeordnete Architektur, die *Makroarchitektur* eines Systems zur Verfügung. Zur Realisierung dieser Architektur werden in der Regel eine Anzahl von *Mikroarchitekturen*, also Design Patterns verwendet. Diese erlauben, wie in Abschnitt 2.4.1 besprochen, die für ein Framework nötige Flexibilität zu erreichen. Trotzdem gehört aber auch die Sammlung von Klassen, also die *Infrastruktur* zu einem Framework.

- Bei der Anwendung eines Frameworks findet eine Umkehrung der Kontrolle statt, das sogenannte Hollywood-Prinzip²⁰.

Diese Umkehrung der Kontrolle wird in Abbildung 2.14 deutlich ge-

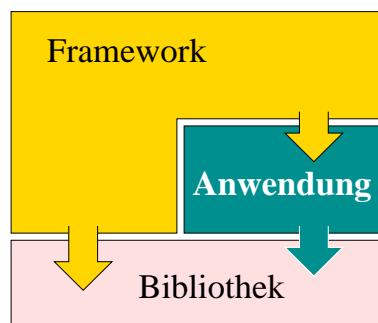


Abbildung 2.14

Bei der Verwendung von Frameworks kommt das *Hollywood-Prinzip* zur Anwendung. Die Anwendungsmethoden werden zum geeigneten Zeitpunkt vom Framework aufgerufen. Das Framework hat die Kontrolle über den zeitlichen Ablauf des Systems.

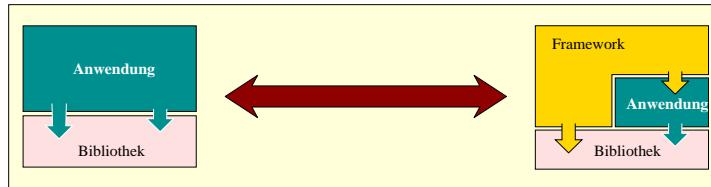
macht. Bei Verwendung von Bibliotheken (vgl. Abbildung 2.7, Seite 27) liegt die gesamte Kontrolle bei der Applikation. Anders bei Frameworks. Hier wird der Ablauf durch das Framework vorgegeben, der Applikationscode hat lediglich die Aufgabe, die anwendungsspezifischen Eigenheiten zu definieren. Dabei können natürlich sowohl das Framework, wie auch die Applikation auf Bibliotheken zurückgreifen um gewisse grundsätzliche Aufgaben zu realisieren.

²⁰Jungen Schauspielern und Schauspielerinnen, die sich bei einer Agentur um eine Rolle bei einem Film bewerben, wird in der Regel nach einem kurzen Gespräch die freundliche Abweisung “*don't call us, we call you*” mit auf den Weg gegeben.

Eine der Aufgaben eines Frameworks ist es, zum gegebenen Zeitpunkt die richtigen Anwendungsmethoden aufzurufen. Wie [Ohlsson, 1994] richtig bemerkt, hat dies zur Folge, dass eine Klasse, im Unterschied zum Ansatz mit Klassenbibliotheken, nicht mehr vollständig definiert werden muss, um wiederverwendet werden zu können. Beim Entwurf und der Implementierung des Frameworks verlässt man sich einfach darauf, dass die Anwendung die fehlenden Teile zu einem späteren Zeitpunkt zur Verfügung stellen wird. Dank den Möglichkeiten von objektorientierten Sprachen wie Polymorphismus und dynamischer Bindung können diese Erweiterungen physisch vollständig getrennt vom Frameworkcode vorgenommen werden. Wie aus den bisherigen Erläuterungen hervorgeht, gibt es klare Unterschiede zwischen ‘reinen’ Bibliotheken und ‘reinen’ Frameworks. Wie Abbildung 2.15 illustriert, sind die reinen Formen zwei einander ge-

Abbildung 2.15

‘Reine’ Bibliotheken und Frameworks stellen zwei gegenüberliegende Pole dar. Dazwischen sind aber auch Mischformen möglich.



genüberliegende Pole. Dazwischen gibt es auch alle möglichen Mischformen. Die *Standard Template Library* (STL) [Stepanov und Lee, 1995; Musser und Saini, 1996], die im neuen ANSI C++ Sprachstandard [Koenig, 1995] als Teil der C++ Standardbibliothek [Breymann, 1996] enthalten sein wird, weist beispielsweise sowohl Bibliotheks- wie auch Frameworkeigenschaften auf.

Klassifizierung

Bei der Betrachtung verschiedener Frameworks, wie beispielsweise ET++ [Weinand et al., 1988; Gamma, 1992], BINTREE LAB [Metz, 1995], CHOICES [Russo et al., 1988; Campell et al., 1993], CONDUIT+ [Hüni et al., 1995; Hüni und Keller, 1997] oder auch *BOOGA* [Amann et al., 1997], stellt man fest, dass beträchtliche Unterschiede hinsichtlich der *angesprochenen Problembereiche* sowie der *internen Struktur* bestehen. [Taligent Inc., 1994a] schlägt eine Klassifikation von Frameworks bezüglich dieser beiden orthogonalen Dimensionen vor.

Bezüglich der Dimension der Problembereiche wird eine Unterscheidung in *Support-Frameworks*, *Domain-Frameworks* und *Application Frameworks* vorgenommen, in teilweiser Anlehnung und Erweiterung von [Gamma, 1992]. Diese Kategorien werden im folgenden kurz vorgestellt:

- *Application Frameworks* kapseln für eine breite Vielfalt von Programmen anwendbares Expertenwissen. Diese Frameworkgrup-

pe umfasst eine *horizontale Schicht* von Funktionalität, die quer über verschiedene Anwendungsgebiete eingesetzt werden kann. Typische Vertreter sind Frameworks zur Programmierung grafischer Benutzeroberflächen (GUI²¹). Beispiele sind ET++ [Weinand et al., 1988; Gamma, 1992] oder die OBJECT WINDOWS LIBRARY (OWL) der Firma Borland, mit welcher grafische Oberflächen von Anwendungen für die Windows-Betriebssystemfamilie der Firma Microsoft erstellt werden können.

- *Domain-Frameworks* kapseln Expertenwissen in einem bestimmten Problembereich. Diese Frameworks umfassen eine *vertikale Schicht* von Funktionalität für eine bestimmte Gruppe von Anwendungen. Die meisten Frameworks, die in Firmen und Institutionen entwickelt werden, sind Vertreter dieses Typs. Beispiele dieser Gruppe sind das CONDUIT+-Framework [Hüni et al., 1995] für die Implementierung von Netzwerkprotokollen und *BOOGA*.
- *Support-Frameworks* stellen Dienste auf Systemstufe zur Verfügung, wie beispielsweise Dateizugriffe, Unterstützung für verteiltes Rechnen oder Gerätetreiber. Anwendungsentwickler verwenden Support-Frameworks meist über das *Client API*²² (vgl. Abbildung 2.16) während Modifikationen von den Systemvertreibern vorgenommen

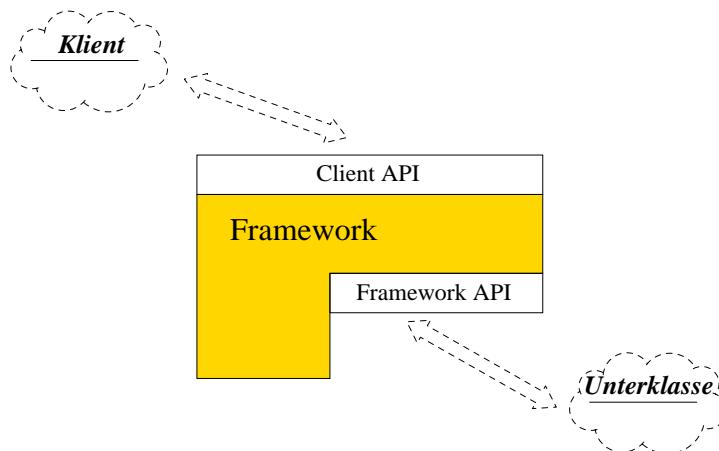


Abbildung 2.16

Im Allgemeinen stellt ein Framework zwei unterschiedliche Schnittstellen zur Verfügung. Während die Verwendung des *Framework API* meist durch Anwendung von Vererbungsbeziehungen entsteht, dient das *Client API* der Verwendung zur Komposition bestehender Frameworkbausteine.

werden. In Spezialfällen, wenn es beispielsweise darum geht, einen neuen Typ von Dateisystem oder einen neuen Gerätetreiber in das Betriebssystem zu integrieren, kann diese Framework aber auch über das *Framework API* angepasst werden. Das bereits erwähnte Framework CHOICES für die virtuelle Speicherverwaltung und

²¹Graphical User Interface

²²Application Programming Interface

Prozessverteilung [Russo et al., 1988] ist ein Vertreter dieser Frameworkklasse.

Orthogonal zu dieser Unterscheidung der Anwendungsebene kann ein Framework, wie erwähnt, auch aufgrund seiner Struktur und der Art, wie die Anwendung mit dem Framework interagiert, beschrieben werden. Wie bereits der Definition von Booch (vgl. Definition 2.4 auf Seite 40) entnommen werden kann, muss ein Framework nicht immer durch Anpassen von Klassen verwendet werden. Eine ganz andere Form der Benutzung ergibt sich, wenn fertige Bausteine des Frameworks zusammengesetzt werden. Dies drückt sich durch zwei unterschiedliche Programmierschnittstellen aus, die Frameworks im Allgemeinen zur Verfügung stellen: das *Framework API* und das *Client API*²³ (vgl. Abbildung 2.16).

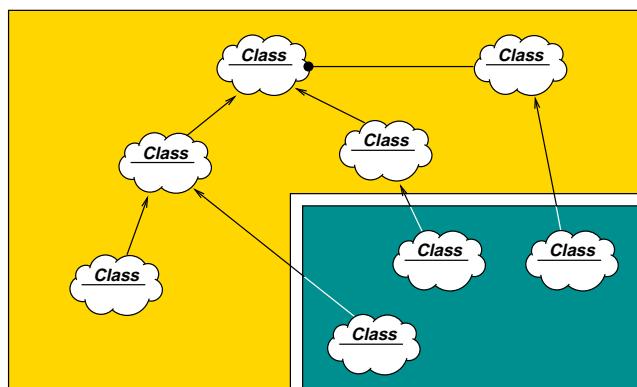
Der Standardisierungsgrad eines Anwendungsgebietes beeinflusst die Struktur eines Frameworks. Bei gut bekannten und standardisierten Anwendungsgebieten, wie Graphische Benutzeroberflächen oder Datenbanken, ist weniger Flexibilität erforderlich als bei relativ unbekannten Gebieten.

White-Box

Ist die Standardisierung nur sehr eingeschränkt möglich, ist also mit anderen Worten grösstmögliche Flexibilität gefordert, so wird das Framework das Implementieren der anwendungsspezifischen Methoden weitgehend der Applikation überlassen. Dies geschieht mittels Ableitung von den entsprechenden Frameworkklassen. Diese Art von Frameworks werden in

Abbildung 2.17

White-Box Frameworks werden mittels Vererbung an die jeweiligen applikationsspezifischen Gegebenheiten angepasst. Der Applikationsprogrammierer muss sehr viel über die Mechanismen des Frameworks wissen, um diese zu erweitern, hat dafür aber die volle Flexibilität des Frameworks zur Verfügung.



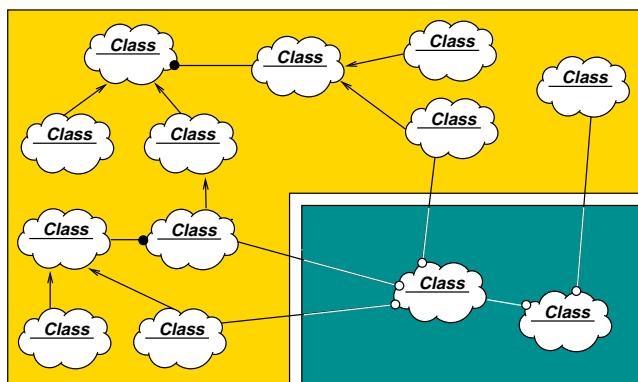
der Literatur als *White-Box* Frameworks bezeichnet (vgl. Abbildung 2.17); soll ein solches Framework verwendet werden, muss man dessen Aufbau und innere Mechanismen sehr genau verstehen.

²³In [Weyerhäuser, 1995] werden diese beiden Schnittstellen auch als *Calling API* (anstatt Client API) und *Subclassing API* (anstatt Framework API) bezeichnet.

[Taligent Inc., 1994a] bezeichnet White-Box Frameworks aufgrund der Wichtigkeit der Vererbung für die Spezialisierung auch als *inheritance-focused* oder *architecture-driven* Frameworks. Die Bezeichnung ‘*architecture-driven*’ ist dabei darauf zurückzuführen, dass Vererbung ein sehr statisches Mittel zur Wiederverwendung von Code ist, da sämtliche Entscheide während der Programmierung getroffen werden müssen.

Wird das Anwendungsgebiet andererseits sehr gut verstanden und ist eine Standardisierung zum mindest teilweise machbar, so kann zugunsten einer einfacheren Anwendbarkeit auf einen grossen Teil der Flexibilität des vorausgehenden Ansatzes verzichtet werden. Ein solches Framework wird verschiedene direkt verwendbare Klassen zur Verfügung stellen. Objekte dieser Klassen werden mittels Komposition oder Parametrisierung zusammengefügt. Dies erlaubt weniger Flexibilität zugunsten einer einfacheren Anwendbarkeit. Diese Frameworks werden als *Black-Box* Fra-

Black-Box

**Abbildung 2.18**

Black-Box Frameworks werden mittels Komposition und Parametrisierung an die applikationsspezifischen Gegebenheiten angepasst. Der Applikationsprogrammierer muss nur noch die Schnittstellen der einzelnen Klassen kennen, ohne die Mechanismen des Frameworks wirklich verstehen zu müssen.

meworks bezeichnet (vgl. Abbildung 2.18); die Klassen können alleine aufgrund einer genauen Kenntnis der Schnittstellen eingesetzt werden.

Diese Black-Box Frameworks werden von [Taligent Inc., 1994a] als *composition-focused* oder *data-driven* Frameworks bezeichnet. Es wird damit dem Umstand Rechnung getragen, dass die Komposition für Black-Box Frameworks viel zentraler ist als die Vererbung. Obwohl Vererbung - vor allem bei Sprachen mit statischer Typisierung - ebenfalls stark eingesetzt wird, geschieht dies hier aus anderen Motiven: an die Stelle von Codewiederverwendung tritt Polymorphismus. Die Bezeichnung ‘*data-driven*’ beschreibt die Art des Programmierens mit Black-Box Frameworks. Anstatt, wie bei White-Box Frameworks neue Klassen abzuleiten und zu implementieren, werden hier Objekte bestehender Klassen instanziiert und kombiniert.

Wie [Johnson und Foote, 1988] beobachteten, entsteht aus mehreren gleichartigen Applikationen über mehrere Iterationen ein erstes White-

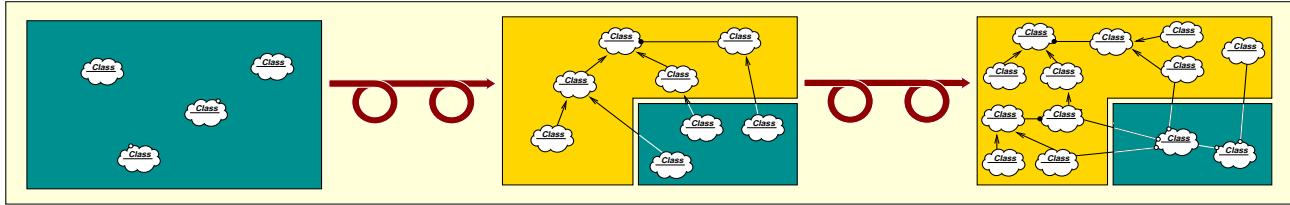


Abbildung 2.19

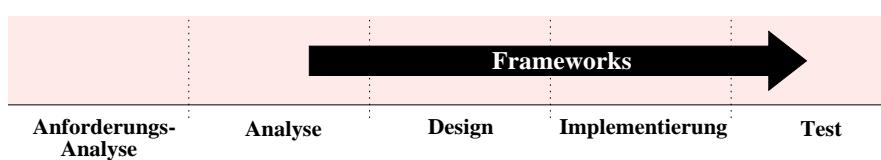
Auf der Grundlage von einigen Beispielapplikationen kann ein gutes White-Box Framework entwickelt werden, das sich dann mit zunehmender Erfahrung in Richtung eines Black-Box Frameworks entwickeln kann.

Box Framework (vgl. Abbildung 2.19). Mit zunehmendem Kenntnisstand entwickelt sich dieses dann in der Regel langsam in Richtung eines Black-Box Frameworks. Die innere Komplexität nimmt bei diesem Prozess laufend zu, ebenso die Wiederverwendbarkeit. Der Aufwand für die Applikationsprogrammierung wird hingegen in der Regel abnehmen.

Wie besprochen, beinhaltet ein Framework das abstrakte Design einer ganzen Familie von Anwendungen. Somit wird dieses Design wiederverwendet, was zu massiven Einsparungen in der Designphase beiträgt. Wie in Abbildung 2.20 gezeigt, ist ein existierendes Framework während bei-

Abbildung 2.20

Frameworks bieten Wiederverwendung von Implementierung und Design, durch das zugehörige Vorgehensmodell aber auch Hilfen bei der Problemanalyse.



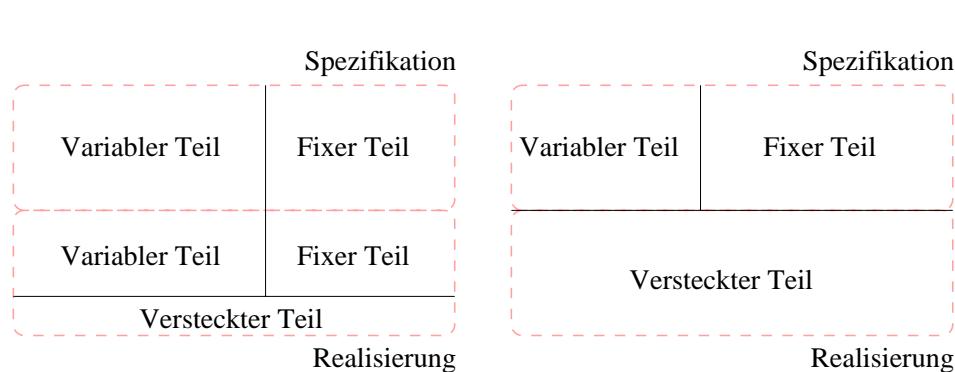
nahe dem gesamten Entwicklungszyklus von grossem Nutzen. Der mit der Analysephase sehr frühe Einsatzzeitpunkt von Frameworks ist darauf zurückzuführen, dass ein Framework, wie bereits erwähnt, neben den Klassen (Implementierungsphase) und dem abstrakten Design (Designphase) auch ein Vorgehensmodell zum Lösen des Problems beinhaltet.

Die folgenden Ausführungen betrachten Frameworks wiederum, wie in Abschnitt 2.2 besprochen, aus dem Blickwinkel von [Krueger, 1992]:

Bewertung von Frameworks

Abstraktion

Frameworks sind auf bestimmte Anwendungsgebiete oder Teile von Anwendungsgebieten beschränkte, vorbereitete *Makroarchitekturen*. Die kleinsten, sinnvoll verwendbaren Abstraktionen in Frameworks sind *Klassenteams*. Mit Hilfe der in Abschnitt 2.4.1 beschriebenen, durch Design Patterns ausgedrückten Mikroarchitekturen wird die für eine möglichst breite Anwendung nötige Flexibilität erzielt.

**Abbildung 2.21**

linke Abbildung:
Bei einem White-Box Framework muss ein beträchtlicher Teil der Realisierung offen gelegt werden, da zur Spezialisierung Vererbung verwendet wird.

rechte Abbildung:
Ein Black-Box Framework hat zwar einen kleineren variablen Teil, kann dafür die Realisierung aber weitgehend verstecken.

Je nach Struktur des Frameworks muss die Realisierung teilweise offen gelegt werden. Der variable Teil der Abstraktion ist bei den Black-Box hingegen in der Regel kleiner.

Die Selektion ist aufgrund der Komplexität eines Frameworks oft sehr schwierig. Sie beinhaltet bei einem White-Box Framework die Auswahl einer geeigneten Basisklasse und die Wahl der darin zu überschreibenden Methoden. Beim Überschreiben der Methoden müssen die jeweils gültigen Randbedingungen beachtet werden. Es muss bekannt sein, wann die Methode aufgerufen und welche Semantik von ihr erwartet wird.

Selektion

Bei Black-Box Frameworks hingegen gestaltet sich die Selektion ähnlich wie bei Bibliotheken.

Die Spezialisierung wird durch zwei Strategien erreicht, abhängig von der Struktur des Frameworks. Während White-Box Frameworks mittels Implementierung neuer, von Frameworkklassen abgeleiteter Klassen und Überschreiben bestimmter Methoden an ihre speziellen Aufgaben angepasst werden, geschieht dies bei reinen Black-Box Frameworks lediglich mittels Parametrisierung und Komposition.

Spezialisierung

Bei Frameworks muss die Fragestellung bezüglich der Integration umgedreht werden. Bisher wurde unter diesem Gesichtspunkt betrachtet, wie die einzelnen wiederverwendbaren Elemente kombiniert und in die spezifischen Teilen integriert werden. Ein wesentliches Merkmal von Frameworks ist aber die Inversion der Kontrolle, das heisst, dass nicht wiederverwendbare Elemente in die spezifischen, sondern die spezifischen in die wiederverwendbaren Elemente integriert werden.

Integration

Bei der Verwendung einer statisch typisierten Sprache kann wiederum der Compiler die syntaktische Überprüfung der Methodenaufrufe übernehmen, wie dies bereits bei den Bibliotheken bemerkt wurde. Zusätzlich

können nun aber die Methoden des Frameworks die anwendungsspezifischen Elemente ‘überwachen’. Diese Überwachung ist, da Bestandteil des Frameworks, wiederverwendbar.

Kombinierbarkeit Die bei den Bibliotheken bezüglich Kombinierbarkeit gemachten Aussagen lassen sich direkt auch auf Frameworks übertragen. In beiden Fällen ist es durchaus üblich, grosse Systeme nicht nur aus *einem* Framework, respektive *einer* Bibliothek aufzubauen, sondern aus mehreren.

Wie in [Johnson, 1996] ausgeführt, gibt es verschiedene Ansätze, um Frameworks zu kombinieren. Einige der Patterns in [Gamma et al., 1995] sind ebenfalls ausschliesslich dieser Problematik gewidmet. Diese Verfahren sind gut geeignet, um getrennte Frameworks zu entwickeln, die zusammenarbeiten können. Wie bereits bei der Besprechung der Kombinierbarkeit der Bibliotheken erwähnt, können aber - je nach verwendeter Sprache - auf der Stufe der Implementierung aufgrund von Namenskonflikten gravierende Probleme auftreten. Die Kombination von Frameworks, die in unterschiedlichen Sprachen implementiert wurden, ist eher noch schwieriger als dies bei Bibliotheken der Fall ist. Weitere Schwierigkeiten können sich bei der Kombination zweier Applikations-Frameworks ergeben. Diese sind oft so implementiert, dass sie einen sogenannten *Main-Loop* enthalten. Dieser Main-Loop ist eine Endlosschleife, in der immer wieder dieselben Funktionen aufgerufen werden. Diese Funktionen werden die Kontrolle zwischenzeitlich an die applikationsspezifischen Methoden abgeben. Die Kombination zweier solcher Frameworks wird in der Regel nicht gelingen, wenn sie nicht explizit im Design mindestens eines der beteiligten Frameworks vorgesehen ist.

Bewertung Wie gezeigt wurde, sind Frameworks ein sehr mächtiges, aber auch komplexes Mittel zur Wiederverwendung von Implementierung *und* Design. Die Entwicklung von guten Frameworks ist sehr aufwendig und erfolgt in der Regel in mehreren Teilschritten. Dabei werden entweder aus mehreren exemplarischen Anwendungen gemeinsame Teile extrahiert, oder die Entwickler des Frameworks haben bereits tiefgreifende Erfahrungen im Anwendungsgebiet.

[Taligent Inc., 1995] führt die folgenden generellen Vorteile von Frameworks auf:

- ☺ Ein Framework beschreibt wesentliche Teile der Architektur des zu lösenden Problems. Dadurch sind bereits die grössten Teile des Systemdesigns vollzogen. Durch die Wiederverwendung von Design und Implementierung sollte im Laufe der Zeit auch der gesamte Softwareerstellungsprozess an den Einsatz von Frameworks angepasst werden.

- ☺ Das Framework stellt eine durch die mehrfache Verwendung erprobte, zuverlässige Infrastruktur zur Lösung ähnlicher Probleme zur Verfügung.
- ☺ Ein Framework stellt einen Mechanismus für die *zuverlässige, kontrollierte* Funktionalitätserweiterung dar. Zusätzliche Funktionalität wird an genau festgelegten Stellen in das Gerüst eingefügt. Wichtig hierbei ist, dass anders als bei älteren Ansätzen die wiederverwendeten Elemente nicht direkt abgeändert und dadurch in ihrer Zuverlässigkeit nicht beeinträchtigt werden.
- ☺ Je grösser der Anteil an wiederverwendeten Teilen ist, desto kleiner wird der Aufwand zur Fehlerbehebung an der einzelnen Applikation werden. In Frameworkklassen entdeckte Fehler müssen nur einmal korrigiert werden und kommen unmittelbar allen Anwendungen zugute, die dieses Framework einsetzen.
- ☺ Ein weiterer, sehr wichtiger Punkt wird von [Birrer et al., 1995] erwähnt. Frameworks beinhalten nämlich nicht nur Design und Implementierung zu einem bestimmten Problemkreis aus der Sicht eines Informatikers, sondern sie kapseln auch Wissen über ein bestimmtes Anwendungsgebiet. Frameworks helfen somit, *Know-how* in den Kernkompetenzen eines Betriebes zu sichern, dies bezüglich Softwaretechnik *und* bearbeitetem Anwendungsgebiet.

Die besprochenen, bezüglich ihrer Struktur verschiedenen Frameworktypen haben jeweils unterschiedliche Vor- und Nachteile:

- *White-Box Framework*

- ☺ Ein White-Box Framework bietet eine grosse Flexibilität. Mit Hilfe der Vererbung können alle dafür vorgesehenen Methoden überschrieben werden. Mit Hilfe eines guten Frameworks sind auch Probleme lösbar, die sich stark von denen unterscheiden, für die es ursprünglich konzipiert wurde [Taligent Inc., 1994a].
- ☹ Bei einem White-Box Framework müssen, wie bereits erwähnt, die Mechanismen des Frameworks vom Anwendungsprogrammierer sehr genau verstanden werden. Dies führt dazu, dass dieser nicht nur genaue Kenntnisse des Anwendungsgebietes haben muss, sondern auch ein sehr tiefes Wissen über objektorientiertes Design und die verwendete Implementierungssprache im Allgemeinen sowie über das Design des Frameworks im Speziellen.

- :(Die Spezialisierung eines White-Box Frameworks geschieht mit Hilfe der Vererbung. Von den im Framework dafür vorgesehenen Klassen werden anwendungsspezifische Klassen abgeleitet und bestimmte Methoden des Frameworks überschrieben. Dies führt in der Regel zu einer grossen Anzahl von Klassen, die typischerweise wenig Komplexität aufweisen. Da aber die gesamte Funktionalität auf diese Weise auf viele Klassen und Hierarchiestufen verteilt wird, kann dies dazu führen, dass logisch zusammenhängende Methoden auf viele einzelne Klassen verstreut werden. Das so entstandene Softwaresystem ist in der Regel nur noch für Experten verständlich und nur von diesen wartbar.

- *Black-Box Framework*

- : Die ‘Wiederverwendung durch Vererbung’ wird durch ‘Wiederverwendung durch Objektkomposition’ ersetzt. Dies führt dazu, dass an die Stelle der aufwendigen und fehleranfälligen Implementierung neuer Klassen das ‘Zusammenstecken’ von Frameworkklassen tritt:

Frameworks should work out of the box.

[Johnson, 1996]

Um dies zu erreichen, stellt ein Black-Box Framework direkt benutzbare Klassen zur Instanzierung und Parametrisierung zur Verfügung.

- : Da bei Black-Box Frameworks die Objektkomposition in den Vordergrund des Interesses rückt, sind nicht mehr die internen Mechanismen wichtig, sondern nur noch die Schnittstellen der einzelnen Klassen. Diese sind schneller und einfacher lernbar, speziell wenn diesem Umstand bei der Entwicklung des Frameworks und der Vergebung der Namen bereits Rechnung getragen wurde. So sind die einzelnen Klassen in der Regel in Hierarchien angeordnet, um das Konzept des Polymorphismus ausnutzen zu können. Dies führt aber dazu, dass alle Klassen einer solchen Hierarchie identische Schnittstellen haben. Eine weitere Vereinfachung für den Anwender ergibt sich, wenn Methoden in unterschiedlichen Hierarchien, die vergleichbare Aufgaben erfüllen, auch denselben Namen erhalten.
- : Wie bereits [Johnson und Foote, 1988] beobachten konnten, nimmt die Wiederverwendbarkeit eines Frameworks zu, wenn die Beziehung zwischen den Teilen durch Protokolle und Komposition anstatt durch Vererbung ausgedrückt wird.

- ⌚ Die Entwicklung eines Black-Box Frameworks ist wesentlich aufwendiger und anspruchsvoller als die eines White-Box Frameworks. Fertige Klassen, die mittels Komposition und Parametrisierung an neue Aufgaben angepasst werden können, erfordern wesentlich genauere Kenntnisse des Anwendungsbereites und entsprechender Softwaretechniken.

Die im nächsten Abschnitt zur Diskussion stehenden Komponenten stellen eine Möglichkeit dar, viele der Vorteile von Frameworks zu erhalten, ohne den Nachteil der grossen Komplexität in Kauf nehmen zu müssen.

2.5 Wiederverwendung durch Komponenten

Der Trend in der modernen Softwareentwicklung geht zunehmend in Richtung offener Systeme. Bei offenen Systemen stehen Anforderungen bezüglich Erweiterbarkeit, Verteilung und Plattformunabhängigkeit im Vordergrund. Einmal erstellte Anwendungen sollen mit neuen, unabhängig davon entwickelten Anwendungen oder Teilanwendungen zusammen arbeiten. Die objektorientierten Technologien alleine können diese Anforderungen nur unzureichend unterstützen (vgl. hierzu auch Abbildung 2.22 auf Seite 60).

[McIlroy, 1968] führte den Begriff der Komponenten in die Softwaretechnologie ein. Seine Idee war, einen Industriezweig aufzubauen, der dafür zuständig ist, eine genügende Anzahl von “Komponenten ab Stange” zu produzieren. Diese könnten dann von anderen Firmen als Bauteile verwendet werden, um Applikationen daraus zusammenzubauen. Basierend auf einer genügend grossen Basis solcher Komponenten könnten sich Softwareentwickler auf die Frage “Welche Mechanismen sollen wir *verwenden*” konzentrieren, anstatt sich mit der Problemstellung “Welche Mechanismen sollen wir *entwerfen*” auseinandersetzen zu müssen.

McIlroy stellte sich Ende der 60er Jahre Komponenten auf einer rein funktionalen Basis und gebunden an eine bestimmte Programmiersprache vor. Als Beispiel führt er in seiner Publikation eine Sinusfunktion auf. Da nach seiner Ansicht Komponenten für jede Aufgabe in jeder denkbaren Ausführung und nach verschiedenen Effizienzkriterien implementiert erhältlich sein müssten, kommt er auf etwa 300 verschiedene nötige Sinusfunktionen.

Ebenfalls Ende der 60er, Anfang der 70er Jahre wurde mit der Entwicklung des Betriebssystems UNIX begonnen [Salus, 1994]. McIlroy war einer der beteiligten Entwickler und begründete das Konzept der Pipes, die das *Toolbox*-Prinzip von UNIX erst ermöglichen. Er legte damit den Grundstein der bis heute gültigen UNIX-Philosophie:

- Schreibe Programme, die *eine* Aufgabe *gut* erledigen.
- Schreibe Programme, die *zusammenarbeiten*.
- Schreibe Programme, die Textströme behandeln, da diese eine allgemein *gültige Schnittstelle* darstellen.

Diese Maxime bildet, wie noch gezeigt wird, den Grundstock für den Begriff der Komponente, wie er im Rahmen dieser Arbeit verwendet wird.

Obwohl sich diese Arbeit vor allem mit den technischen Aspekten der Wiederverwendung auseinandersetzen will, wird gerade im Zusammenhang mit Komponenten in der Fachliteratur aber auch eine Vielzahl von anderen Problemen diskutiert. Sollen Komponenten in vielen Fachgebieten allgemein verfügbar sein, so ergeben sich Fragestellungen bezüglich Dokumentation, Auffinden, Qualität, Effizienz, allgemein gültigen Schnittstellen, Lizenzierung, etc. die hier nicht weiter betrachtet werden. Obwohl die allgemeine Verfügbarkeit von einfach integrierbaren, kostengünstigen und zuverlässigen Komponenten für alle denkbaren Anwendungsgebiete als Vision sehr geeignet ist, muss ein *erster Schritt* in diese Richtung darin bestehen, die Technologie der Komponenten allgemein bekannt zu machen, diese *lokal* einzusetzen und so auf breiter Basis Erfahrungen zu sammeln.

Beschäftigt man sich mit der Fülle von Publikationen, die zum Thema ‘Komponenten’ erschienen sind, so fällt auf, dass eigentlich nur auf einer fast philosophisch anmutenden Ebene Einigkeit herrscht: Komponenten werden von fast allen Autoren als ein Schritt aus der Softwarekrise hin in eine industrialisierte Softwaretechnologie betrachtet. Weitgehende Übereinstimmung besteht auch darin, dass die Herstellung von Software kein ausreichend industrialisierter Prozess ist, da das Wissen viel zu stark an einzelne Personen gekoppelt ist [Jacobson et al., 1992] und eine ungenügende Arbeitsteilung besteht. Die Entwicklung von Software ist immer noch eine *arbeitsintensive Handarbeit* anstelle einer *kapitalintensiven Ingenieurdisziplin* [Nierstrasz et al., 1991].

Bereits bei der Frage, was eine Komponente ist, herrscht Uneinigkeit. Die Bandbreite der Definitionen reicht von wenig nützlichen, sehr allgemeinen Ansätzen (Eine ‘Komponente’ ist ein Synonym zu ‘Ding’) bis hin zu sehr konkreten, produktbezogenen Vorstellungen (‘Komponenten sind das, was mit OLE²⁴ zusammengebaut wird’). Um einerseits die sehr unterschiedliche Auffassung von Komponenten zu dokumentieren, andererseits die Herkunft des für den Rest dieser Arbeit verwendeten Begriffes zu belegen, werden in der Folge einige Definitionen unterschiedlicher Personen aus dem Umfeld der objektorientierten Technologien vorgestellt und diskutiert. Die Definitionen sind teilweise sehr informell, manchmal gar etwas pointiert gehalten, da einige von den Autoren während Vorträgen an Konferenzen oder Seminarien teilweise ad-hoc aufgrund von Zuhörerfragen formuliert wurden.

Der Definitionsversuch von Andrew Watson (Systemarchitekt der

Definition

²⁴OLE: *Object Linkage and Embedding*, eine von Microsoft entwickelte Technologie zur Applikationsintegration und -kommunikation für die verschiedenen Windows-Betriebssysteme der Firma.

OMG²⁵), vorgestellt während der Panel Discussion der CUC 96²⁶ in München beschreibt den wohl grössten gemeinsamen Nenner aller Definitionen:

Definition 2.5

A Component is anything that is reused more than once.

Diese ‘Definition’ ist zwar sehr prägnant, hilft aber nur insofern weiter, als sie die Motivation für den Einsatz von Komponenten herausstreckt. Es stellt sich allerdings die Frage, ob die Wiederverwendung ein notwendiges Kriterium ist, um ein Softwareelement als Komponente bezeichnen zu können. Definition 2.11 wird diesen Aspekt etwas deutlicher beleuchten. Wie [Nierstrass und Meijler, 1995] ausführen, ist die Tatsache, dass ‘etwas’ wiederverwendet wird auch kein hinreichendes Kriterium, dieses Element als Komponente zu bezeichnen. Was eine Komponente wirklich auszeichnet, ist vielmehr die Tatsache, dass sie mit der *Absicht entwickelt* wurde, mit anderen Komponenten im Rahmen einer generischen Architektur zusammenzuarbeiten.

Eine ebenfalls sehr allgemeine Definition stammt von Bruce H. Cottman, Keynote Speaker an der CUC 96 (Chairman des *ComponentWare Consortium*²⁷):

Definition 2.6

Components: You can plug them together.

Dieser Definition kann man - “zwischen den Zeilen” - entnehmen, dass eine Komponente scheinbar eine klare Schnittstelle besitzen muss und mit anderen Komponenten kooperiert.

Auch [CWC, 1995a] legt das Schwergewicht der Definition auf die Eigenschaften der Schnittstelle:

Definition 2.7

Any discrete software module that exhibits a plug-and-play interface is called a component.

²⁵ OMG: *Object Management Group*, ein 1989 gegründeter Zusammenschluss von mittlerweile einigen Hundert der weltweit wichtigsten Herstellern und Anwendern von Informationstechnologien. Das Ziel der Organisation ist die Definition und Anwendung einer *Object Management Architecture* (OMA), deren heute bekanntestes Element die *Common Object Request Broker Architecture* (CORBA) ist [Ben-Natan, 1995].

²⁶ CUC: Component Users Conference, 15.-19. Juli 1996, Forum Hotel, München.

²⁷ Das *ComponentWare Consortium* (CWC) ist ein Zusammenschluss von Firmen mit dem Ziel, die Entwicklung und Verfügbarkeit von wiederverwendbarer, objekt-orientierter und verteilter Software zu fördern. Dabei sollen herstellerunabhängig Architekturen gefördert werden, die komponentenorientierte Software unterstützen. Das CWC arbeitet eng mit der OMG zusammen [CWC, 1995b; CWC, 1995c].

Was dadurch unter den Begriff Komponente fällt, wird erst klar²⁸, wenn die Definition von *plug-and-play* in derselben Publikation betrachtet wird:

Definition 2.8 (Plug-and-Play)

A type of software component that needs little or no modification for its use.

Diese Definition schliesst, im Gegensatz zu den meisten anderen Definitionen, Softwareelemente in den Begriff Komponenten mit ein, die vor dem Einsatz nicht nur konfiguriert, sondern sogar modifiziert werden müssen. Diese Ansicht steht in deutlichem Widerspruch zu den nachfolgend besprochenen Definitionen.

Die Definition von Douglas Schmidt:

Definition 2.9

A component is a cohesive unit of operations that can be reused solely upon its interface.

und jene von Manfred Broy:

Definition 2.10

A component is a physical encapsulation of related services according to a published interface.

(beide ebenfalls Keynote Speaker an der CUC 96 in München) sind sehr ähnlich. In beiden Definitionen wird betont, dass eine Komponente (1) eine Anzahl von *Operationen* oder *Diensten*, die - wie auch immer - zusammengehören und (2) eine Schnittstelle besitzt, die nach aussen bekannt gegeben wird, wobei die Art der Implementation keinen Einfluss auf die Wiederverwendung haben soll.

Diese beiden Definitionen haben zwei wesentliche Merkmale von Komponenten eingeführt, die in der abschliessenden Definition (vgl. Definition 2.15) ebenfalls enthalten sein werden. Es sind dies die klar definierten Schnittstellen von Komponenten sowie die Tatsache, dass eine Komponente als *Black-Box* zu betrachten ist. Die Definitionen von Schmidt (Definition 2.9) und Broy (Definition 2.10) unterscheiden sich von den Definitionen von Watson (Definition 2.5) und Cottman (Definition 2.6) darin, dass einzelne Funktionen oder Makros nicht als Komponenten betrachtet werden, da eine Komponente mehrere Dienste anbietet.

²⁸Diese beiden Definitionen stellen keine sauberen, formalen Definitionen dar, da sie sich gegenseitig referenzieren. Sie sind eine direkte Übersetzung aus [CWC, 1995a].

Der bei der Definition von Watson (Definition 2.5) diskutierten Einwand, dass die Anzahl der Wiederverwendungen nicht darüber entscheiden sollte, ob etwas eine Komponente ist oder nicht, wird in [Webster, 1995] verstrtkt:

Definition 2.11

A component is a unit of software designed to integrate and work with other units of software.

Nicht die zuflige sptere Wiederverwendung kennzeichnen also eine Komponente, sondern die Absicht des Entwicklers, die Komponente mit anderen zusammenarbeiten zu lassen.

Genauso betonen auch [Nierstrasz und Dami, 1996] in ihrer Definition, dass Wiederverwendung und Komponenten keine zuflichen Resultate sind:

Definition 2.12

A piece of software is a component because it has been designed to be used in a compositional way together with other components.

Diese beiden Definitionen fassen den Kerngedanken des Komponentenansatzes sehr prgnant zusammen. Fr den praktischen Einsatz bieten sie aber zuwenig Anhaltspunkte, welche Eigenschaften eine Komponente konkret aufweist.

[Nierstrasz und Dami, 1996] fhren neben der methodischen Definition 2.12 noch eine zweite, eher technisch orientierte Definition auf:

Definition 2.13 (Komponenten)

A component is a static abstraction with plugs.

‘Statisch’ soll in diesem Zusammenhang andeuten, dass es sich bei Komponenten um langlebige Elemente handelt, die unabhngig sind von den Applikationen, in denen sie verwendet wurden. Mit ‘Abstraktion’ wird betont, dass die Komponente eine - mehr oder weniger - undurchlssige Hlle um die gekapselte Software legt. Schliesslich muss eine Komponente auch mit anderen Komponenten zusammenarbeiten und kommunizieren knnen. Dies wird mit dem Zusatz ‘with plugs’ ausgedrkt.

Die folgende Definition von [Gamma, 1996] ist etwas weniger prgnant als die vorausgehenden, listet dafr aber explizit die Eigenschaften einer Komponente auf:

Definition 2.14

Die folgenden Eigenschaften werden als zentral betrachtet. Eine Komponente

- (1) *ist eine Black-Box mit einer wohldefinierten Schnittstelle,*
- (2) *kann mit anderen Komponenten kombiniert werden und interagieren und*
- (3) *unterstützt ‘Packaging’.*

Weitere, optionale Charakteristiken sind:

- (a) *Unabhängigkeit von der Implementationssprache sowie*
- (b) *Aufrufbarkeit über Adressräume hinweg.*

Mit Eigenschaft (1), der Forderung nach einer Kapselung der Implementationsdetails, wird der Tatsache Rechnung getragen, dass, wie im Abschnitt über Frameworks (siehe Abschnitt 2.4.2) besprochen, White-Box Wiederverwendung zwar mehr Flexibilität gewährleistet, dies aber zu Lasten der Verständlichkeit geht. Gleichzeitig wird die in Definition 2.7 und 2.8 ausdrücklich zugelassene Modifikation von Komponenten ausgeschlossen. Einige der in [Gamma et al., 1995] besprochenen Design Patterns können dabei helfen, flexible Black-Box Komponenten zu erstellen, indem die Komponenten beispielsweise mit bestimmten Verhaltensobjekten konfiguriert werden können. Die Forderung, dass eine Komponente eine Black-Box darstellt, kann, etwas pointiert formuliert, mit der folgenden Aussage motiviert werden:

*Adding new code is good,
changing old code is bad.*

[Liebermann, 1988]

Eigenschaft (2) ist eine grundlegende Forderung, die bereits in anderen besprochenen Definitionen erwähnt wurde. Eine Komponente wird stets mit der Absicht entwickelt, mit anderen Komponenten zusammenzuarbeiten. Dies bedeutet aber letztlich, dass eine geeignete Infrastruktur bereitgestellt werden muss, die eine einheitliche und einfache Kommunikation zwischen den Komponenten ermöglicht.

Die letzte zentrale Eigenschaft (3) der Definition von Gamma bedarf einer etwas ausführlicheren Erläuterung. Komponenten sollten idealerweise in binärer Form vertrieben werden können. Dies ist unter anderem aus lizenzrechtlichen Überlegungen wünschenswert, da so die zur Implementation der Komponente verwendeten Algorithmen als Gegenstand von Wettbewerbsvorteilen gegenüber Konkurrenten versteckt werden können.

Die Konsequenzen dieser Forderung sind beispielsweise ‘*release to release binary compatibility (RRBC)*’ oder das dynamische Nachladen von Komponenten in eine laufende Applikation.

Selbstverständlich sollte die Tatsache sein, dass die Menge der Komponenten, die zusammenarbeiten können, nicht im voraus festgelegt ist, sondern vielmehr laufend erweitert werden kann.

Die optionalen Eigenschaften schliesslich helfen dabei, die heutigen Anforderungen an moderne, grosse Applikationen zu erfüllen, bestehende Anwendungen zu kombinieren, die in unterschiedlichen Sprachen oder für verschiedenen Plattformen geschrieben wurden.

Die nachfolgende Definition definiert den Komponentenbegriff, wie er im Rahmen von *BOOGA* verwendet wird. Die Definition will nicht die gleiche Allgemeinheit aufweisen wie die ersten betrachteten Definitionen. Gleichzeitig sollen aber auch allzu visionäre Forderungen ausgeschlossen werden, um so einen handhabbaren Begriff zu erhalten, der es auch im kleineren Rahmen erlaubt, die Vorteile des Komponentenansatzes zu nutzen.

Definition 2.15 (Komponenten)

Eine Komponente

- (1) ist eine Black-Box,
- (2) kann mit anderen Komponenten kombiniert werden und interagieren, wobei die Menge der Komponenten nicht im voraus festgelegt ist,
- (3) erfüllt eine abgeschlossene Teilaufgabe und
- (4) stellt eine wohldefinierte Schnittstelle zur Verfügung.
- (5) Die Systemarchitektur ist ausserhalb der Komponenten definiert und durch ein Integrationsframework gegeben.

Die ersten beiden Eigenschaften decken sich mit den ersten beiden Forderungen aus der Definition 2.14 von Erich Gamma. (3) und (4) sind den Definitionen von Douglas Schmidt (Definition 2.9) und Manfred Broy (Definition 2.10) entnommen.

Die Methoden oder Funktionen der Schnittstelle (Eigenschaft (4)) können dabei gemäss ihrem Aufgabenbereich unterschieden werden:

Dienstschnittstelle: Dieser Teil der Schnittstelle ist für die eigentlichen Aufgaben der Komponente verantwortlich. Die von der Komponente zur Verfügung gestellten Dienste werden mit Hilfe dieser Schnittstelle in Anspruch genommen.

Verwaltungsschnittstelle: Dieser Teil der Schnittstelle erfüllt Verwaltungsaufgaben, wie beispielsweise die Abfrage nach den verfügbaren Diensten oder die Instanzierung von Komponenten, die alle Komponenten gleichermassen zur Verfügung stellen und oft von der gemeinsamen Infrastruktur verlangt werden.

Mit Eigenschaft (5) wird die bereits in Definition 2.14, Punkt (2) besprochene Forderung verstärkt, dass eine verbindende Infrastruktur grundlegend für den erfolgreichen Einsatz von Komponentenarchitekturen ist.

Ruft man sich nochmals die in der Einleitung zu diesem Abschnitt betrachtete UNIX-Philosophie (Seite 52) in Erinnerung, so werden viele Parallelen augenfällig:

- UNIX Programme sind ebenfalls Black-Boxes. Die Programme liegen in einer ausführbaren Form vor und können nicht mehr modifiziert werden.
- Die Menge der UNIX Programme ist selbstverständlich offen. Die einzelnen Programme können, sofern sie gemäss der UNIX Richtlinien entwickelt wurden, auf einfache Art kombiniert werden und zusammenarbeiten.
- Jedes UNIX Programm erfüllt eine abgeschlossene Teilaufgabe.
- Jedes UNIX Programm besitzt eine klar definierte Schnittstelle und erlaubt die Kommunikation basierend auf Textströmen. Die Schnittstelle eines UNIX Programmes kann auch in Dienstschnittstelle und Verwaltungsschnittstelle unterteilt werden: die Standardeingabe- und Standardausgabeströme stellen die Dienstschnittstelle dar. Über diese Ströme kommunizieren die unabhängigen Programme mittels Pipes miteinander. Jedes UNIX Programm erlaubt aber zusätzlich die Übergabe von Kommandozeilenparametern. Diese stellen die Verwaltungsschnittstelle eines Programmes dar, mit dessen Hilfe es für seine spezielle Aufgabe konfiguriert werden kann.
- Die Systemarchitektur ist ausserhalb der UNIX Programme im Betriebssystem festgelegt. Hier wird definiert, wie die einzelnen Programme gestartet und terminiert werden. Die Pipes als die eigentliche Kommunikationskanäle sind ebenfalls ausserhalb der einzelnen Programme definiert.

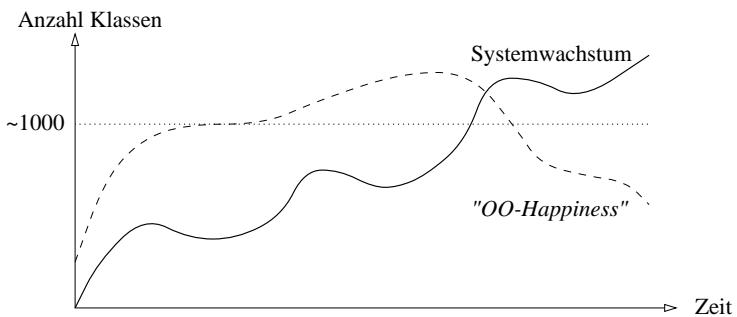
Der folgende Abschnitt bespricht die allgemeinen Eigenschaften von Komponenten, sowie deren Vor- und Nachteile.

Eigenschaften

Objektorientierte Systeme zeichnen sich in der Regel durch eine sehr grosse Anzahl von involvierten Objekten aus. Die Abhaengigkeiten zwischen diesen Objekten steigen, sofern keine speziellen Vorkehrungen getroffen werden, sogar noch schneller als die Anzahl der Objekte. Dieses spezielle Problem ist eine wichtige Ursache der in Abbildung 2.22 dargestellten, mit der Anzahl Klassen abnehmenden Zufriedenheit bei der Realisierung objektorientierter Systeme.

Abbildung 2.22

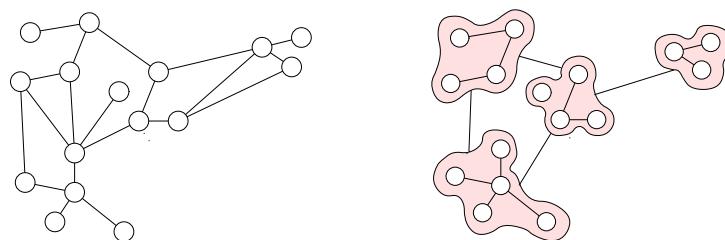
Mit zunehmender Systemgrösse überwiegen die Nachteile der durch den objektorientierten Ansatz erhöhten Komplexität die Vorteile [Gamma, 1996].



Das Ziel einer jeden guten Architektur muss sein, die Kopplung zwischen den Elementen eines Systems zu verringern; objektorientierte Technologien können aber diese Kopplung sogar noch erhöhen. Eine starke Kopplung führt jedoch fast zwangsläufig zu grossen, monolithischen Systemen. Diese werden durch Anpassungen und Änderungen immer stärker zu einer brüchigen, nicht mehr überschaubaren Ansammlung von dicht miteinander verwobenen Einzelteilen. Ein solches System ist nur schwer zu verstehen und sehr aufwendig in der Wartung. Frameworks (speziell White-Box Frameworks) können ebenfalls dazu beitragen, die Kopplung innerhalb eines Systems zu erhöhen.

Eine komponentenbasierte Vorgehensweise hingegen kann dabei helfen, das Problem der Kopplung zu verkleinern, indem das System in einzelne, voneinander weitgehend unabhängige Komponenten zerlegt wird (Abbildung 2.23). Wie diese Komponenten realisiert werden, ist dabei von sekundärer Bedeutung. Wichtig ist allerdings, dass die Anzahl der Komponenten sowie die Grösse einer einzelnen Komponente in einem sinnvollen Verhältnis zueinander stehen.

Die Kopplung zwischen Komponenten kann weiter reduziert werden, indem zur Kommunikation zwischen den Komponenten einfache Typen verwendet werden (vgl. UNIX, das auf Text als Datentyp zur Kommunikation zwischen den Komponenten basiert), sowie durch Einplanen von 'Schlupf', wie dies bereits in Abschnitt 2.4.1 erläutert wurde. Weiterhin sollte aufgrund der in Abschnitt 2.3.1 besprochenen Probleme auf Vererbung zur Definition der Komponentenschnittstelle verzichtet werden [Weinand, 1996].

**Abbildung 2.23**

Bei rein objektorientierten Ansätzen werden die Abhängigkeiten zwischen den Elementen eines Systems in der Regel sehr gross. Komponentenorientierte Architekturen helfen bei der Bewältigung dieser Komplexität [Weinand, 1996].

Es kann beobachtet werden, dass sich Anwendungen heute in zwei Richtungen entwickeln: einerseits sind da die monolithischen Systeme (auch als „*Fatware*“ oder „*Bloatware*“ bezeichnet), die immer mehr Ressourcen benötigen, um den Anforderungen aller Anwender gerecht zu werden. Auf der anderen Seite entstehen schlanken Grundsysteme, die mittels zusätzlicher *Plug-ins*, *Add-ons* oder *Komponenten* von jedem Anwender selber an seine speziellen Bedürfnisse angepasst werden können.

Der Problemlösungsansatz bei Verwendung von Komponententechnologien unterscheidet sich gemäss [Jacobson et al., 1992] von den bekannten objektorientierten Analyse- und Designmethoden. Während hier nach wie vor der traditionelle Top-Down Ansatz dominiert, muss bei der Verwendung von Komponenten ein Bottom-Up Verfahren bevorzugt werden. Der Grund hierfür wird unmittelbar einsichtig, wenn man sich vor Augen führt, dass das Ziel bei einem komponentenbasierten Ansatz darin besteht, ein Problem auf eine Anzahl grösstenteils bereits existierender Komponenten abzubilden. Mit einem reinen Top-Down Ansatz wird man nur mit sehr viel Glück eine Problemzerlegung finden, bei der die vorhandenen Komponenten von Nutzen sind.

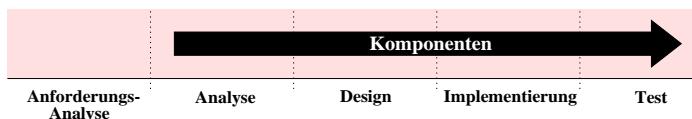
Wie bereits besprochen, muss zur Integration der Komponenten eine generische Architektur, beziehungsweise ein *Integrationsframework* zur Verfügung stehen, das die Kommunikation zwischen den Komponenten regelt. Grosse Teile der Systemanalyse sowie des Systemdesigns sind bereits durch diese generische Architektur definiert. Auch die Analyse sowie das Design der einzelnen Komponenten wird durch den Entscheid für ein bestimmtes, komponentenorientiertes Integrationsframework sehr stark beeinflusst.

Diese Überlegungen begründen den in Abbildung 2.24 dargestellten, sehr grossen Einfluss des Komponentengedankens auf den gesamten Entwicklungszyklus.

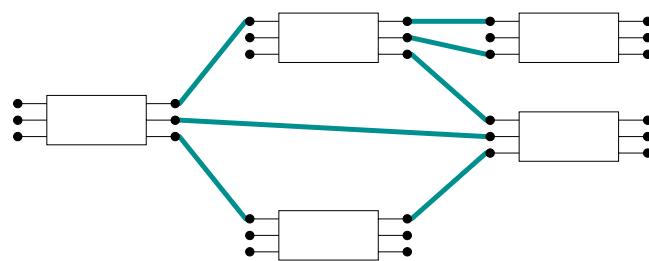
Der eigentliche Aufwand bei der Applikationsprogrammierung liegt – eine ausreichend grosse Basis an bestehenden Komponenten vorausgesetzt – theoretisch nur in der *Auswahl* sowie richtigen *Verknüpfung* der Komponenten (vgl. Abbildung 2.25). In der Realität wird es allerdings nur selten

Abbildung 2.24

Komponenten beeinflussen alle Entwicklungsstufen, mit Ausnahme der Anforderungsanalyse. Sämtliche Teilschritte können durch die Verwendung von Komponenten stark verkürzt werden.

**Abbildung 2.25**

Bei der Verwendung von Komponenten liegt die Applikationslogik einerseits in der richtigen Auswahl der Komponenten, andererseits in der Verknüpfung der einzelnen Schnittstellenelemente der Komponenten.



so weit kommen, dass für sämtliche Anforderungen bereits passende Komponenten bestehen. Fehlende Komponenten müssen entwickelt und in die *Komponentenbibliothek* integriert werden um späteren Anwendungen zur Verfügung zu stehen.

Welche Vorteile bringt nun aber der Einsatz von Komponenten konkret? Die folgende Auflistung aus [Meijler und Nierstrasz, 1997] soll die wesentlichen Pluspunkte des Komponentenansatzes kurz erläutern:

Geringe ‘time-to-market’: Anwendungen, die mit Hilfe von wieder verwendbaren Komponenten entwickelt werden, können schneller entwickelt werden. Es ergeben sich dadurch Zeit- und Preisvorteile gegenüber den Mitbewerbern.

Zuverlässigkeit: Komponenten, die bereits in vielen unterschiedlichen Anwendungen eingesetzt wurden, sind zuverlässiger als neu codierte. Die Testphase kann als Folge daraus, ohne Qualitätseinbusse, massiv verkürzt werden.

Arbeitsteilung: Komponenten sind aufgrund der nötigen, wohldefinierten Schnittstelle ideale Einheiten zur Arbeitsteilung zwischen Softwareteams. Die Projektorganisation, die Planung und die Erfolgskontrolle können dadurch stark vereinfacht werden.

Flexibilität: Verschiedene Anwendungen können nur dann auf eine gemeinsame Basis von wiederverwendbaren Elementen zurückgreifen, wenn diese eine genügende Flexibilität aufweisen. Komponenten

unterstützen Flexibilität durch Parametrisierung. Das Verhalten einer Komponente kann auf diese Weise vom Anwender an seine speziellen Bedürfnisse angepasst werden.

Anpassbarkeit: Jede Anwendung muss im Laufe ihrer Existenz an neue Anforderungen angepasst werden. Die meisten dieser Anforderungen können bei komponentenbasierten Systemen mittels Änderung der Komponentenparameter, Änderung der Kommunikationskanäle oder Austauschen einzelner Komponenten einfach und lokal erfüllt werden.

Flexibilität stellt eine Forderung an die einzelnen Komponenten, Anpassbarkeit eine Forderung an die gesamte Applikation dar. Dabei ist wichtig zu beachten, dass eine gewisse Konkurrenzsituation zwischen diesen beiden Forderungen besteht. Eine exzessive Flexibilität der einzelnen Komponenten führt in aller Regel aufgrund der dadurch stark ansteigenden Komplexität zu einer deutlich verringerten Anpassbarkeit der gesamten Applikation. Andererseits ist aber eine gewisse Flexibilität der Einzelteile eine grundlegende Voraussetzung um die Anwendung an neue Bedürfnisse anzupassen.

Die Black-Box-Eigenschaft des Komponentenansatzes erleichtert die Gratwanderung, die nötig ist, um die beiden Forderungen nach Flexibilität und Anpassbarkeit in einer ausgewogenen Balance zu halten.

Weitere Vorteile des Komponentenansatzes sind gemäss [Meijler und Nierstrasz, 1997]:

Verteilung: Die Verteilung von Anwendungen auf mehrere Hardwareressourcen wird immer wichtiger. Die Implementierung von verteilten Anwendungen ist inhärent komplex. Komponenten sind einerseits eine gute Basis zur Verteilung, andererseits kann die Komplexität der Kommunikation zwischen den verteilten Komponenten entweder in den betroffenen Komponenten oder in der verwendeten Infrastruktur gekapselt werden. Dies erleichtert die Entwicklung der gesamten Applikation, wie auch der einzelnen Komponenten.

Heterogenität: Moderne Anwendungen werden nicht nur auf mehrere Prozessoren verteilt, sondern auch auf unterschiedliche Plattformen mit unterschiedlichen Betriebssystemen. Zusätzlich muss '*Legacy Code*' weiterverwendet werden, oder die einzelnen Komponenten werden in unterschiedlichen Programmiersprachen implementiert. Komponenten müssen also unabhängig von der verwendeten Programmiersprache, der Plattform und des Betriebssystems kommunizieren können. Durch den Einsatz von Infrastrukturen wie CORBA kann diese Forderung erreicht werden.

Die Zerlegung einer Anwendung in Komponenten birgt aber noch einen weiteren Vorteil:

Iterative Entwicklung: Durch die nötige Unabhängigkeit der Komponenten von den Implementierungen der anderen wird eine iterative Entwicklung der gesamten Anwendung stark vereinfacht. Zu Beginn einer Entwicklung können entweder Komponenten verwendet werden, die gewisse Randbedingungen bezüglich Zeit- oder Speicherplatzkomplexität noch nicht erfüllen oder solche, die das gewünschte Verhalten erst simulieren. Diese Komponenten können dann schrittweise verbessert oder ersetzt werden.

Betrachtet man diese eindrückliche Liste der Vorteile von Komponenten, so stellt man sich unwillkürlich die Frage, warum der komponentenorientierte Ansatz nicht weitere Verbreitung gefunden hat. [Jacobson et al., 1992] beantwortet diese Frage mit einer Liste von Problemen:

Projektbindung: Projekte haben normalerweise ein knappes Budget und einen engen Zeitplan. Die Entwicklung guter, wiederverwendbarer Komponenten braucht zusätzliche Zeit, die in der Regel nicht zur Verfügung steht.

'Not-invented-here' Syndrom: Viele Entwickler fühlen sich immer noch unwohl, wenn sie dem Code von jemand anderem vertrauen müssen.

Fehlende Standards: Es gibt keine allgemein akzeptierten Standards für Komponenten. Obwohl sich gewisse Komponentenarchitekturen wie OPENDOC²⁹ oder OLE zu etablieren scheinen, sind diese nicht für alle Anwendungen gleichermassen geeignet. Es ist fraglich, ob der Wunsch nach *einem* einheitlichen Komponentenstandard überhaupt realisierbar ist; zu unterschiedlich scheinen die verschiedenen Gebiete der Informatik und deren Anforderungen zu sein.

Katalogisierung: Es kann durchaus sein, dass für ein bestimmtes Problem bereits eine Lösung in Form einer Komponente existiert, nur sind diese den Entscheidungsträgern des Projektes nicht bekannt und werden darum nicht berücksichtigt. Zusammen mit

²⁹OPENDOC ist ein Standard, der von Mitgliedern der *Component Integration Labs (CI Labs)* entwickelt wurde. Die *CI Labs* wurden 1993 von Apple, IBM, Novell, Oracle, Taligent, Wordperfect und XSoft (eine Abteilung von Xerox) gegründet.

OPENDOC stellt eine plattformunabhängige Architektur zur Entwicklung von Komponentensoftware dar [OPENDOC, 1994]. Der Fokus liegt dabei auf dokumentenbasierten Anwendungen aus dem Bereich der Büroautomation.

der Einführung einer Komponentenarchitektur in einem Betrieb müssen auch Lösungen zur Dokumentation und Katalogisierung der Komponenten erarbeitet werden.

Produktivität: Ein Entwickler empfindet seine Arbeit als wesentlich produktiver, wenn er Code schreiben kann, anstatt ‘nur’ bestehende Komponenten zusammenzufügen. Zudem werden Codezeilen leider immer noch als Mass zur Bestimmung der Produktivität von Entwicklern verwendet. Die Unsinnigkeit dieses Ansatzes wird in diesem Zusammenhang besonders deutlich.

Lizenziierung: Ein Komponentenmarkt muss sich vor dem Kopieren und Verteilen von Komponenten ohne Entgelt schützen. Gelingt die Lösung dieses Problems nicht, wird sich keine Firma finden, die Komponenten, in deren Entwicklung viel Aufwand gesteckt wurde, anderen zur Verfügung zu stellen. Neben der heute für ‘Fatware’ bekannten Form der Lizenzierung und Abrechnung drängt sich bei Komponenten ein ‘*pay-per-use*’ auf. Untersuchungen in dieser Richtung sind im Gange, die Lösungen sind aber noch nicht kommerziell verwertbar.

Je breiter die Anwendung der Komponenten erfolgen soll, um so schwerer wiegen diese Probleme. Will ein kleines Team damit beginnen, komponentenorientierte Anwendungen zu entwickeln und auf einen eigenen Komponentenstandard aufzubauen, so können die meisten der aufgezählten Problemkreise vernachlässigt werden.

In der Folge werden Komponenten wiederum gemäss der Klassifikation von [Krueger, 1992] besprochen.

Bewertung des Komponentenansatzes

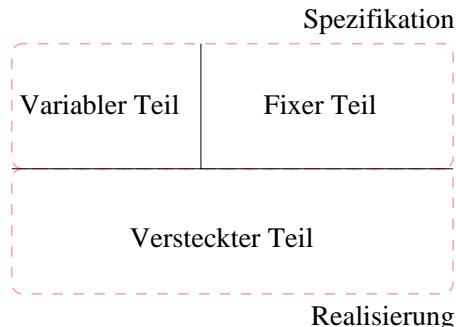
Komponenten als Black-Boxes stellen eigentlich ideale Elemente zur Wiederverwendung dar (vgl. Abbildung 2.26). Diese ist allerdings nur möglich, wenn die Komponenten basierend auf einem gemeinsamen Integrationsframework entwickelt wurden, das unter anderem festlegt, wie die Kommunikation zwischen den Komponenten abläuft, wie die Komponenten in ein System integriert werden, welche gemeinsame Funktionalität vorhanden ist etc.

Die Kapselung kann soweit gehen, dass zur Implementation einer einzelnen Komponente und zur Kombination von Komponenten zu einer Anwendung nicht die gleiche Sprache verwendet wird. Dies wurde bereits von [DeRemer und Kron, 1976] erkannt. Die Autoren nannten die Implementation einer Komponente ‘*Programming-in-the-Small*’, das Kombinieren zu ganzen Applikationen ‘*Programming-in-the-Large*’. Für die

Abstraktion

Abbildung 2.26

Da eine Komponente als Black-Box verstanden wird, ist die gesamte Realisierung versteckt. Die Spezifikation kann variable Teile enthalten, wenn die Komponente parametrisierbar ist.



Anwendungsprogrammierung wurde die Verwendung einer *Module Interconnection Language (MIL)* vorgeschlagen. Komponentenorientierte Ansätze haben gegenüber den objektorientierten Verfahren den grossen Vorteil, dass die Unterscheidung zwischen '*Computation*' und '*Composition*' explizit unterschieden wird [Nierstrasz und Meijler, 1995].

Selektion

Ähnlich wie bei Bibliotheken können auch Komponentensammlungen in Handbüchern dokumentiert werden. Gerade bei den visionären Absichten von standardisierten Komponentenarchitekturen ist aber abzusehen, dass die Anzahl der Komponenten so gross sein wird, dass die Katalogisierung zu einem der Hauptprobleme avanciert.

Eine Eigenheit des Komponentenansatzes ist die Tatsache, dass die Komponentenbibliothek von unterschiedlichen Programmierern laufend erweitert wird, da neue Komponenten sinnvollerweise anderen Anwendern so rasch wie möglich zur Verfügung gestellt werden sollte. Dies schliesst aber klassische Handbücher aus, da sich diese als zu statisch erweisen. In Kapitel 7 werden unter anderem Ansätze zur Dokumentation von Komponenten diskutiert.

Verglichen mit den in Abschnitt 2.3.2 besprochenen Bibliotheken sind Komponenten wesentlich aufwendiger zu beschreiben, als einzelne Funktionen. Ein einzelner Entwickler wird also, aufgrund der Anzahl der Komponenten sowie der Informationsmenge pro Komponente nicht mehr in der Lage sein, die relevante Information für alle wichtigen Komponenten präsent zu halten. Es müssen somit Verfahren gefunden werden, welche die Anforderungen an die Dokumentation bezüglich Flexibilität, Erweiterbarkeit, Zugänglichkeit aber auch Konsistenz mit dem Code erfüllen. Zur Dokumentation von **BOOGA** wurde ein Web-basierter Ansatz erprobt. Dieser wird in Abschnitt 7.3 besprochen.

Spezialisierung

Die Spezialisierung von Komponenten wird in der Regel durch Parametrisierung der einzelnen Komponenten mit elementaren Werten, Objekten oder weiteren Komponenten erfolgen. Durch die Anwendung von Design

Patterns wie *Strategy* [Gamma et al., 1995, Seite 315] kann eine ausreichende Flexibilität erzielt werden.

Die Integration der Komponenten erfolgt, wie bereits besprochen, mit Hilfe eines Integrationsframeworks. Bei einem komponentenbasierten Ansatz ist die Applikation im wesentlichen durch die *Verknüpfung* sowie die *Auswahl* der Komponenten gegeben.

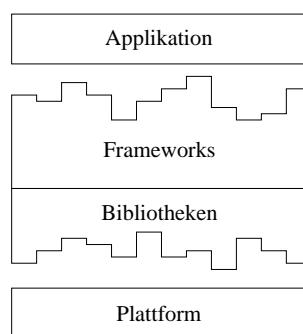
In der Regel werden die wiederverwendeten Komponenten und die anwendungsspezifischen, neuerstellten Komponenten in derselben Weise realisiert und integriert. Im Idealfall können die neu entstandenen Komponenten in die Komponentenbibliothek integriert werden.

Komponenten sind dafür entwickelt, miteinander kombiniert werden zu können. Dies ist allerdings nur möglich, wenn das gleiche Integrationsframework zugrundegelegt wird.

Sind Komponenten nicht dafür ausgelegt, miteinander zu kommunizieren, so kann eine Kombination trotzdem möglich sein, indem *Adapter*, also spezielle Schnittstellenkomponenten, geschrieben werden, die ein Komponentenprotokoll in das andere übersetzen.

[Weinand, 1996] vergleicht den objektorientierten mit dem komponentenorientierten Ansatz. Der grundlegende Aufbau einer objektorientierten Applikation wird dabei wie in Abbildung 2.27 skizziert. Obwohl sich mit Hilfe der objektorientierten Technologien ein guter Wiederverwen-

Integration



Kombinierbarkeit

Bewertung

Abbildung 2.27

Bei rein objektorientierten Ansätzen ergeben sich in der Regel zwei „semantische Lücken“, wohingegen die mittleren Bereiche sehr gut abgedeckt sind.

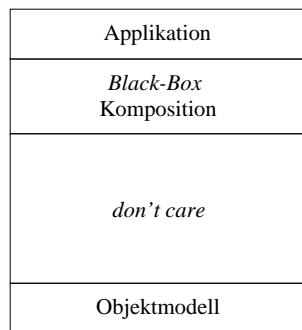
dungsgrad im mittleren Bereich erzielen lässt, stellen sich doch grosse Probleme: objektorientierte Ansätze weisen oft eine zu grosse Kopplung zwischen den Elementen eines Systems auf. Frameworks sind schwierig zu erstellen und zu erlernen. Die Portierung auf unterschiedliche Plattformen stellt ein weiteres Hindernis dar.

Der Komponentenansatz (Abbildung 2.28) zielt im Gegensatz dazu auf eine wesentlich gröbere Wiederverwendung ab. Wie die Komponenten

selbst implementiert sind, interessiert in der Regel weniger. Wichtig ist eine explizite Unterstützung von Black-Box Komposition sowie ein Modell, das spezifische Systemeigenschaften kapselt.

Abbildung 2.28

Der Komponentenansatz konzentriert sich sehr pragmatisch auf die semantischen Lücken:
Black-Box Komposition wird explizit unterstützt, wie aber die Komponenten intern aussehen ist ein Implementationsdetail.



Die längerfristige Vision des komponentenorientierten Ansatzes besteht darin, eine Arbeitsteilung zu erreichen. Eine Komponentenindustrie produziert allgemein einsetzbare Komponenten, die von den Applicationsentwicklern sehr einfach zusammengebaut werden können. Dabei sollen Implementationssprachen, Betriebssysteme oder zugrundeliegende Hardware idealerweise keine Rolle mehr spielen. Die Anforderungen an Komponenten steigen in einer solch visionären Umgebung allerdings erheblich.

Obwohl heute erste Ansätze zu solch ambitionierten Systemen sichtbar werden³⁰, ist der Aufwand, in eine solche Umgebung einzusteigen, für viele noch zu gross. Ein Komponentenansatz weist aber, wie betrachtet, trotzdem viele Vorteile auf. Um diese Vorteile nutzen zu können, ohne die Nachteile in Kauf nehmen zu müssen, kann eine Applikation durchaus auf Komponenten aufbauend entwickelt werden, ohne einen der sich entwickelnden Standards zu verwenden, wofür Erich Gamma an der CUC '96 in München den Begriff der *Component-Ready Architecture* prägte.

³⁰vgl. beispielsweise [CWC, 1995a; CWC, 1995b; CWC, 1995c; OPENDOC, 1994; Java Beans, 1996a; Java Beans, 1996b].

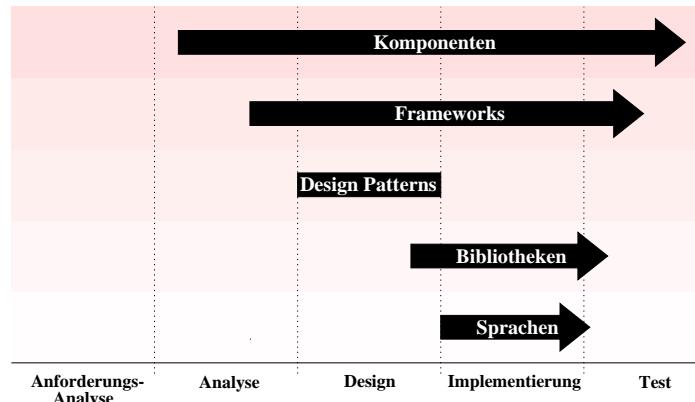
2.6 Abschliessende Betrachtungen

In diesem Kapitel wurden einige der gebräuchlichsten Wiederverwendungstechnologien betrachtet und auf ihr Abstraktionsvermögen sowie ihre Effizienz hin untersucht. Dabei wurden vor allem die technischen Aspekte berücksichtigt, ohne detailliert auf wirtschaftliche, oder psychologische Anliegen einzugehen.

Bei der Vielzahl der Wiederverwendungsansätze und der technischen Möglichkeiten, die sich in diesen verbergen, wird deutlich, dass das Problem der fehlenden Wiederverwendung nicht mehr nur in der ungenügenden Beherrschung der Technologie zu suchen ist, sondern auch in den Disziplinen, die rund um den Kern ‘Informatik’ angeordnet sind. Psychologische Hemmnisse, wie das ‘*not-invented-here*’ Syndrom, wirtschaftlicher Druck, der mittelfristige Investitionen verhindert und dadurch nicht selten auch die kurzfristigen Projekte in Frage stellt oder zu wenige, gut ausgebildete technische Mitarbeiter und Führungskräfte sind Probleme, denen man sich ebenfalls stellen muss, will man in der Informatik eine *Kultur der Wiederverwendung* etablieren.

Gebiete, in denen die Wiederverwendung zur Selbstverständlichkeit geworden ist weisen aber oft noch andere Eigenheiten auf. So gibt es von einem Hersteller der Unterhaltungselektronik eine Familie von Videorecordern, die sich durch zunehmende Funktionalität und somit auch durch die Preisgestaltung voneinander unterscheiden. Zu jedem Modell wird eine spezielle Fernsteuerung mitgeliefert, die genau die gemäss Datenblatt unterstützten Funktionen anbietet. Erstaunlicherweise kann man aber die Fernsteuerung eines teureren Modells zusammen mit einem billigeren Modell verwenden und hat die gesamte Funktionalität des *teureren* Modells zur Verfügung. Intern unterscheiden sich die Videorecorder also nicht, lediglich die externe Schnittstelle (also die Fernsteuerung) bietet unterschiedliche Dienste an. Dieselbe Vorgehensweise kann auch in vielen anderen Bereichen beobachtet werden. So wird beispielsweise in Uhren oft ein Mikroprozessor eingesetzt, der wesentlich mehr Funktionalität anbietet als die Uhr in ihrer Benutzerschnittstelle. Auch in der Informatik wird sich diese Tendenz weiter verbreiten. Da sich die Hardware immer noch sehr rasch weiterentwickelt und gleichzeitig immer günstiger wird, ist es durchaus vertretbar, einen Teil dieser zusätzlichen Leistung für eine zwar überdimensionierte, dafür aber günstigere - weil häufiger wiederverwendbare - Software zu opfern.

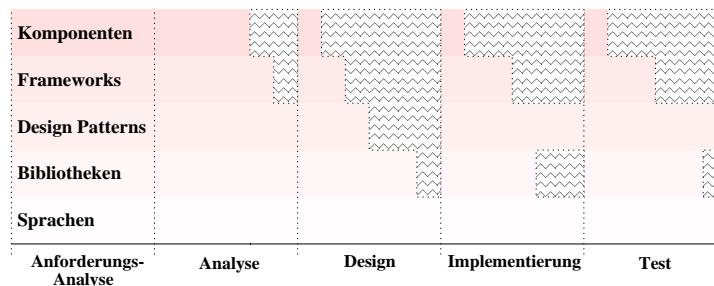
Abbildung 2.29 stellt die bereits bei den einzelnen Technologien besprochenen Einsatzzeitpunkte einander gegenüber. Dabei wird deutlich, dass die Framework- und Komponentenansätze aufgrund der Wiederverwendung von Design und Implementierung erwartungsgemäss die grössten Auswirkungen auf die Softwareentwicklung haben.

**Abbildung 2.29**

Rekapitulation und Gegenüberstellung der Abbildungen 2.6, 2.12, 2.20 und 2.24.

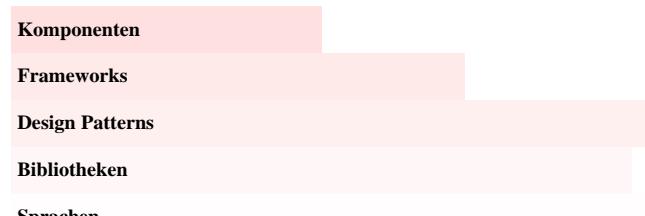
Die verschiedenen Wiederverwendungstechnologien kommen zu unterschiedlichen Zeitpunkten zum Einsatz.

Die folgende Grafik (Abbildung 2.30) versucht das Einsparungspotential der unterschiedlichen Wiederverwendungstechnologien qualitativ zu ver-

**Abbildung 2.30**

Das Einsparungspotential wird grösser, je früher im Softwarelebenszyklus eine Wiederverwendungstechnologie zum Einsatz kommt.

gleichen. Technologien, die bereits in frühen Softwareentwicklungsphasen zum Tragen kommen, weisen dabei natürlich erweise das grösste Potential

**Abbildung 2.31**

Die kumulierte Projektdauer aus Abbildung 2.30.

auf. Abbildung 2.31 zeigt, ebenfalls qualitativ, die kumulierte Projektdauer aus Abbildung 2.30.

Alle in diesem Kapitel besprochenen Wiederverwendungstechnologien können zusammenarbeiten. Sprachen bilden die gemeinsame Grundlage, auf die alle anderen Verfahren fundieren und gleichzeitig die wohl bisher beste Art der Wiederverwendung [Krueger, 1992]. Darauf aufbauend folgen Bibliotheken und Frameworks. Komponenten stellen die konsequente

Weiterentwicklung dieser Ansätze dar. Auch die nächste Zukunft lässt sich bereits skizzieren. Wie beispielsweise in [Roberts und Johnson, 1996] und [Czarnecki, 1996] angeregt wird, lässt sich durchaus auf ein Black-Box Framework eine *domainspezifische Sprache* als weitere Abstraktionschicht aufsetzen. Die nächsten Schritte könnten dann eine *grafische Sprache* sein, mit Hilfe derer Applikationen lediglich durch ‘*point-and-click*’ anstatt durch Programmierung erstellt werden können. Erste Ansätze in dieser Richtung werden heute für die Programmierung von Applikationen mit grafischen Benutzeroberflächen bereits von Umgebungen wie VISUAL BASIC von Microsoft oder DELPHI von Borland realisiert.

Erfolgreiche Wiederverwendungsprojekte werden immer mehrere der hier besprochenen Techniken kombinieren. Beispielsweise kann unter einem Generator ein Framework verborgen sein, welches auf mehrere Bibliotheken zurückgreift. Das Framework und die Bibliotheken sind eventuell in einer objektorientierten Sprache geschrieben. Der Benutzer wird nun zuerst versuchen, seine Probleme in der höchsten Abstraktionsebene - hier also mit dem Generator - zu lösen. Reicht die damit mögliche Flexibilität nicht aus, so kann auf die nächst tieferliegende Abstraktionsschicht zurückgegriffen werden. Wird diese Arbeitstechnik von allen Schichten ausreichend unterstützt, kann mit einem solchen System gleichzeitig ein Maximum an Wiederverwendbarkeit, Flexibilität und Effizienz erzielt werden.

Abbildung 2.32 illustriert ein *Schichtenmodell der Wiederverwendung*, welches darlegt, wie die verschiedenen Wiederverwendungstechnologien aufeinander aufbauen. Die untersten Schichten der Pyramide weisen in

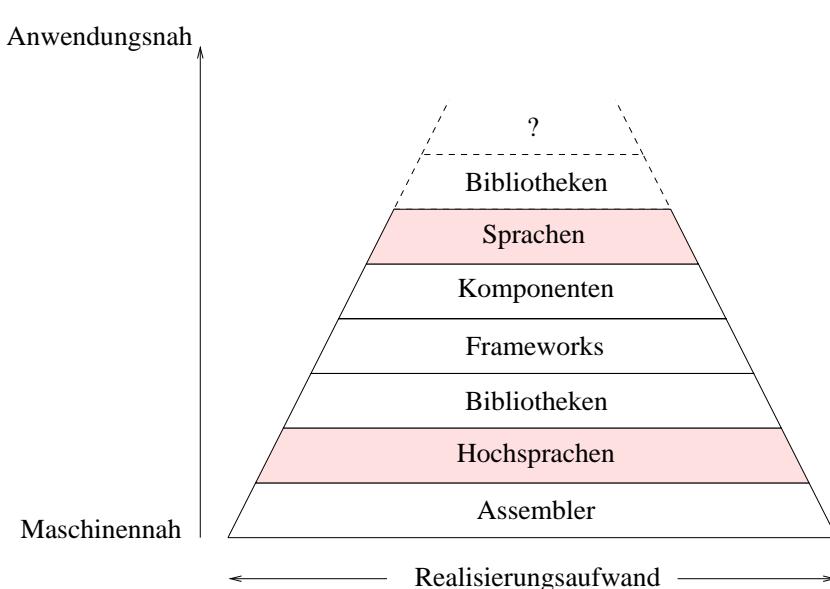


Abbildung 2.32

Das Schichtenmodell der Wiederverwendung

Durch Schichtung der unterschiedlichen Wiederverwendungstechnologien, die in diesem besprochen wurden, lässt sich die kognitive Distanz zwischen der Aufgabenstellung und der Lösung erheblich reduzieren.

Die Granularität der wiederverwendeten Elemente nimmt von unten nach oben zu.

Die grauen Schichten kapseln jeweils die darunter liegenden Ebenen vollständig ab.

Die Sprachen der oberen Schicht können Scripting-, Konfigurations- oder applikationsspezifische Sprachen sein.

der Regel die grössere Flexibilität auf. Der Aufwand, eine Applikation zu realisieren ist auf diesen untersten Schichten enorm, da auf einer sehr maschinennahen Abstraktionsebene gearbeitet werden muss. Zudem ist die Fehlerhäufigkeit tendentiell grösser, da wenig Hilfen zur Bewältigung der probleminhärenten Komplexität zur Verfügung stehen. Andererseits ist aber die Beherrschung der Basistechnologien auf den tiefen Stufen in der Regel wesentlich einfacher.

Bewegt man sich auf der Pyramide nach oben, wird in der Regel die Flexibilität abnehmen; der Entwickler wird in seinen Freiheiten eingeschränkt und muss sich den Regeln der gewählten Umgebung anpassen. Der Aufwand, der in die Schulung der Entwickler investiert werden muss, nimmt stark zu, da die Technologien auf höherer Stufe wesentlich schwieriger zu beherrschen sind. Die Fehleranfälligkeit nimmt aufgrund der zunehmenden Problemhäufigkeit ab, ebenso der Aufwand zur Realisierung einer Applikation.

Aus dem Schichtenmodell der Wiederverwendung lässt sich eine weitere Tendenz ablesen: während die unteren Schichten durch eine maschinennahe *Programmierung* repräsentiert werden, wird diese in den höheren Schichten durch eine problemnahe *Konfiguration* von Komponenten verdrängt (vgl. Abbildung 2.33). Mittels Konfiguration bereits existierender

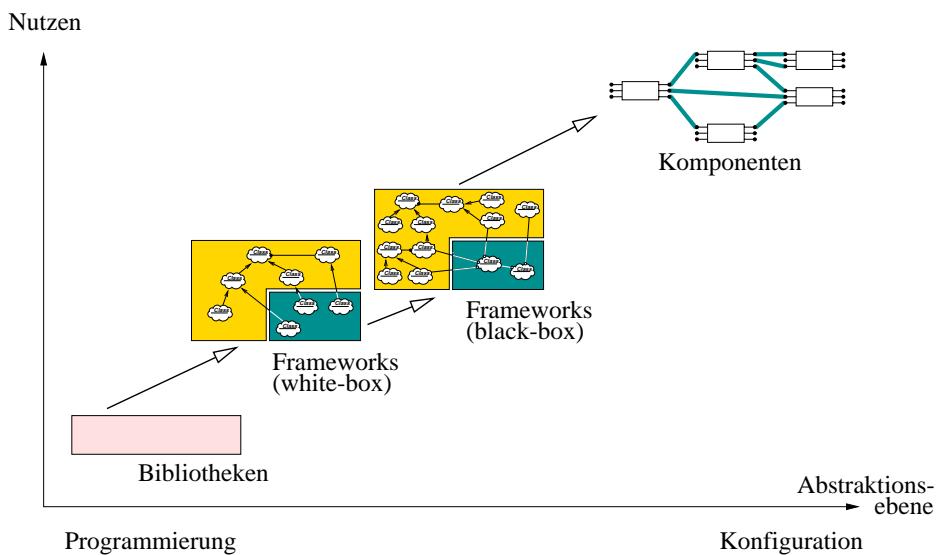


Abbildung 2.33

Die Konfiguration bestehender Komponenten erlaubt den effektiveren Einsatz betriebswirtschaftlicher Ressourcen.

Komponenten kann aber aus den bestehenden betriebswirtschaftlichen Ressourcen der wesentlich grössere Nutzen gezogen werden als durch Neuprogrammierung. Der Grund dafür liegt in der bereits besprochenen Verlagerung von der arbeitsintensiven Handarbeit zur kapitalintensiven, industrialisierten Fertigung von Software durch Technologien, welche die Wiederverwendung fördern.

Bei allen betrachteten Wiederverwendungstechnologien drängt sich eine Arbeitsteilung auf [Nierstrasz et al., 1992]. Es ist die Aufgabe des *Application Engineering*, das Wissen aus den unterschiedlichen Anwendungsgebieten zu abstrahieren und wiederverwendbare Softwareelemente zu entwickeln. Das spezifische Wissen über die einzelnen Anwendungsgebiete, das dabei gesammelt werden kann, wird in generischen Architekturen, Generatoren und spezialisierten Vorgehensweisen festgehalten. Die Aufgabe der zweiten Gruppe besteht im *Application Development*. Eine spezifische Applikation wird mit Hilfe der während des *Application Engineering* entwickelten Methoden und wiederverwendbaren Elementen realisiert.

Die heutigen Softwareentwicklungsmethoden unterstützen die hier besprochenen Technologien allerdings nicht direkt. So wird beispielsweise in den meisten Methoden eine strikte Trennung von Analyse und Design verlangt. Andererseits kann und soll in der Analysephase beispielsweise von der späteren Verwendung eines Frameworks profitiert werden können [Nierstrasz et al., 1992; Nierstrasz und Dami, 1996]. Die Entwicklung geeigneter Vorgehensweisen muss Gegenstand zukünftiger Forschung sein.

Der Einsatz geeigneter Wiederverwendungstechnologien kann nur erfolgreich sein, wenn damit einerseits eine länger dauernde Bereitschaft zur Investition einhergeht, andererseits eine *Kultur* aufgebaut werden kann, welche die Wiederverwendung gegenüber der Neuprogrammierung favorisiert [Nierstrasz et al., 1992].

Alle hier betrachteten Wiederverwendungstechnologien bergen neben den unbestreitbaren Vorteilen auch Nachteile und Risiken. So hat, wie bereits [Johnson, 1996] feststellte, jede Wiederverwendungstechnologie eine bestimmte, mehr oder weniger steile Lernkurve und führt neue Abhängigkeiten und zusätzlichen Overhead ein. Dies kann dazu führen, dass die Eintrittskosten in eine solche Technologie die zu erwartenden Ersparnisse übertreffen. So ist es in der Regel wenig sinnvoll, ein spezielles, domainspezifisches Framework zu entwickeln, wenn nicht mehr als zwei bis drei Anwendungen darauf aufbauend entwickelt werden sollen.

Stellt man bei einer sorgfältigen Kosten-Nutzen Analyse unter Einbezug aller Vor- und Nachteile fest, dass der Einsatz einer bestimmten Technologie schliesslich keinerlei Kostenersparnis mit sich bringt, so ist es angebracht, auf eine einfachere, mit tieferen Eintrittskosten verbundene Technologie auszuweichen.

Mit besonders hohen Kosten sind in der Regel Verfahren verbunden, die trotz erheblichem Wiederverwendungspotential eine grosse Flexibilität aufweisen. Die unweigerlich damit einhergehende Komplexität erhöht die Eintrittskosten enorm. Diesem Umstand wird durch den folgenden Rat-

schlag Rechnung getragen:

*Design as flexible as needed,
not as flexible as possible.*

Erich Gamma

Dieses Kapitel soll mit einem weiteren, pointierten Ratschlag schliessen,

Start stupid and evolve.

Kent Beck

der ganz allgemein für die iterative und inkrementelle Softwareentwicklung massgebend ist.



Salvador Dalí.
Erscheinung eines Gesichtes und einer Fruchtschale auf einer Kiste.
1938.

Kapitel 3

BOOGA: Grundlagen und Motivation

*Du siehst Dinge und sagst:
“Warum?”*

*Aber ich träume Dinge,
die es nie gab, und sage:
“Warum nicht?”*

George Bernard Shaw
1856-1950

3.1 Einleitung

Die in Kapitel 2 vorgestellten Verfahren zur Wiederverwendung von Design oder Implementation sind etablierte Ansätze zur Effizienzsteigerung. Im Rahmen des Projektes **BOOGA** ging es nun darum, eine geeignete Kombination und Gewichtung dieser Verfahren zu finden, um die *anwendungsspezifischen Aufgabenstellungen* im universitären Umfeld erfüllen zu können. **BOOGA** ermöglicht es, Arbeiten der Forschungsgruppe Computergrafik und Computergeometrie in einen grösseren Zusammenhang zu stellen und bietet auch fachgruppenübergreifende Lösungsansätze.

Dieses Kapitel leuchtet einerseits Aspekte des Begriffs Computergrafik aus, andererseits wird das Projekt **BOOGA** in einen Zusammenhang mit früheren Projekten der Fachgruppe Computergeometrie und Grafik gestellt. Dies ist insofern wichtig, als es zum Entwurf und zur Realisierung einer Forschungsplattform, an die sehr hohe Ansprüche (vgl. Abschnitt 3.4) gestellt werden, unabdingbar ist, auf einen grossen und breit abgestützten Erfahrungsschatz im Anwendungsgebiet zurückgreifen zu können. Abschliessend werden die Anforderungen an die Forschungsplattform **BOOGA** formuliert.

Es existieren unterschiedliche Definitionen zum Begriff *Computergrafik*. Die *International Standards Organization (ISO)* definiert Computergrafik recht allgemein als *Summe aller Methoden und Techniken, Daten für die grafische Ausgabe vorzubereiten*, während [Pavlidis, 1982] und [Meier, 1986] Computergrafik sehr spezifisch als den *Vorgang der Erzeugung von Bildern aufgrund von abstrakten Beschreibungen* betrachten.

Im Rahmen dieser Arbeit wird unter *Computergrafik* alles verstanden, was in irgendeiner Form mit der *Visualisierung und der Modellierung von Daten zur Visualisierung* zu tun hat.

Der folgende Abschnitt erläutert, nach einem kurzen Exkurs in die Literatur, die durch diese Definition der Computergrafik abgedeckten Teilgebiete.

3.2 Die Teilgebiete der Computergrafik

Verschiedene Autoren haben eine Aufteilung der Computergrafik beschrieben. Da sich die verschiedenen Teilgebiete zu unterschiedlichen Zeitpunkten entwickelten und je nach Anforderungen aus der Praxis sowie verfügbaren Basistechnologien (Rechnerleistung, grafische Ausgabegeräte, Interaktionsmöglichkeiten, etc.) verschieden gewichtet wurden, ergeben sich natürlicherweise unterschiedliche Schwerpunkte.

So unterteilt [Pavlidis, 1982] das Anwendungsgebiet ‘Computergrafik’ wie in Abbildung 3.1 dargestellt in drei Teilgebiete:

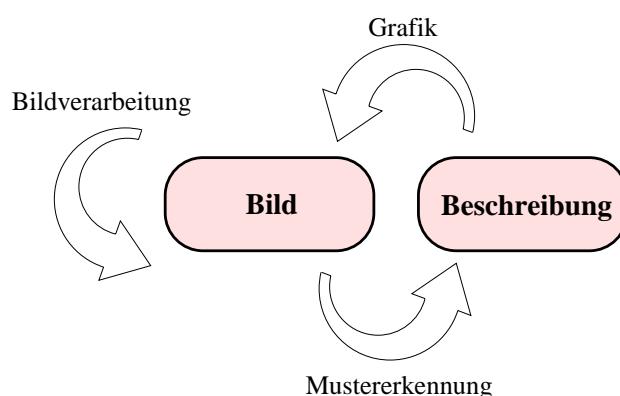


Abbildung 3.1

Die verschiedenen Teilgebiete des Anwendungsbereiches Computergrafik gemäss [Pavlidis, 1982].

Grafik (*Graphics*):

Die Grafik behandelt die Erzeugung von Bildern aufgrund nicht-bildlicher Information aus verschiedenen Anwendungsgebieten. Dies beinhaltet die grafische Darstellung von mathematischen Funktionen oder von Messdaten beispielsweise aus der Medizin, der Physik oder der Chemie. Hohe Anforderungen an Software und Hardware stellen Anwendungen aus den Gebieten der fotorealistischen Bildgenerierung, der Animation, der interaktiven Grafik (Grafikeditoren, *Virtual Reality*), der Simulatoren oder der Videospiele.

Bildverarbeitung (*Image Processing*):

Sind die Eingabe- sowie die Ausgabedaten einer Problemstellung Bilder, so werden die Techniken der Bildverarbeitung zum Einsatz kommen. Die Umwandlung unterschiedlicher Bildformate, die Aufbereitung von gestörten, verrauschten Bildern, die Korrektur von Über- oder Unterbelichtung, die Reduktion der Anzahl Farben, die künstlerische Verfälschung von Bildern¹ oder Kantendetektoren sind Beispiele dieses Teilgebietes der Grafik.

¹In vielen kommerziellen, teilweise auch in frei verfügbaren Programmen zur Bildbearbeitung sind heute eine Vielzahl von Filtern enthalten. Ein solcher Filter könnte beispielsweise dazu dienen, ein Bild impressionistisch zu verfremden.

Mustererkennung (*Pictorial Pattern Recognition*):

Soll eine Beschreibung eines Bildinhaltes erzeugt oder ein Bild einer bestimmten Klasse zugeordnet werden, allgemein also der Bildinhalt analysiert und klassifiziert werden, so spricht [Pavlidis, 1982] von Mustererkennung.

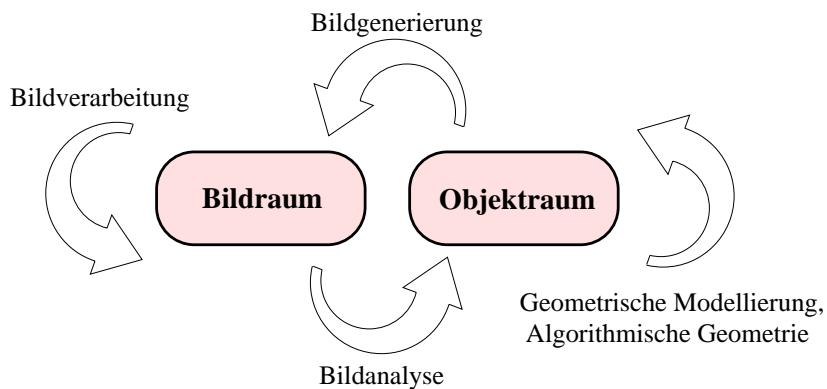
Unter Bilddaten versteht [Pavlidis, 1982] Rasterbilder unterschiedlicher Farbtiefen, diskrete Punkte oder ganze Polygone. Die Bilddaten können als Rasterbilder, Vektorbilder oder mittels Verfahren wie *Chain Codes*² gespeichert werden.

Betrachtet man Abbildung 3.1, so fällt bereits rein optisch ein Ungleichgewicht auf: in [Pavlidis, 1982] wird das Gebiet der Modellierung ausgeklammert. Bei einem Vergleich der Abbildung 3.2 mit Abbildung 3.1 fällt eine weitgehende Ähnlichkeit auf. Tatsächlich sind die beiden Sichtweisen nicht sehr weit voneinander entfernt. Im folgenden werden die Unterschiede kurz besprochen.

Die Unterteilung des Anwendungsgebietes Computergrafik, wie sie in Abbildung 3.2 dargestellt wird, lehnt sich weitgehend an [Meier, 1986] an.

Abbildung 3.2

Teilgebiete des Anwendungsbereiches Computergrafik nach [Meier, 1986]; die Bezeichnung der Teilgebiete richtet sich nach der in der vorliegenden Arbeit verwendeten Terminologie.



Einziger Unterschied ist die Anpassung der Bezeichnungen der Teilgebiete an die in der vorliegenden Arbeit verwendeten Begriffe. In der folgenden Auflistung wurden die in dieser Arbeit verwendeten Begriffe fett gedruckt. In Klammern dahinter steht der in der englischen Literatur gebräuchliche Fachbegriff sowie die Bezeichnung von [Meier, 1986], sofern sie sich von der hier verwendeten unterscheidet.

²Soll beispielsweise die Kontur eines Gegenstandes auf einem Bild beschrieben werden, so ist eine Folge von Tupeln, welche die x- und y-Koordinaten jedes einzelnen Punktes beschreiben, sehr ineffizient. Wesentlich besser geeignet ist das Verfahren der *Chain Codes*. Hier wird jeder möglichen (8-)Nachbarschaft eines Bildpunktes ein Wert zwischen 0 und 7 zugeordnet. Um die Kontur zu beschreiben genügt ein Anfangspunkt, sowie eine Folge von Werten zwischen 0 und 7, welche jeweils die Richtung des auf der Kontur nachfolgenden Punktes bezeichnen.

Die von [Meier, 1986] vorgeschlagene Gliederung beruht auf der Idee, die den unterschiedlichen Algorithmen zugrundegelegten Modelle in zwei Kategorien einzuteilen. Die Elemente des *Bildraums (Image Space)* sind, im Gegensatz zu den ‘Bildern’ in [Pavlidis, 1982], ausschliesslich Rasterbilder unterschiedlicher Farbtiefen. Der *Objektraum (Object Space)* umfasst alle anderen, zwei- oder dreidimensionalen Repräsentationen von geometrischen Objekten und ist häufig das Modell eines konkreten, physikalischen Raums. Zwischen diesen beiden Räumen sind Übergänge definiert, die den gängigen Teilgebieten der Computergrafik entsprechen:

Bildverarbeitung (Image Processing):

Dieser Bereich deckt sich mit der Bildverarbeitung, wie sie bei der Einteilung nach [Pavlidis, 1982] besprochen wurde.

Bildanalyse (Image Analysis; Computervision [Meier, 1986]):

Gegenstand der Bildanalyse ist die Erzeugung von Beschreibungen der Inhalte von Bildern. So können beispielsweise aufgrund von zwei oder mehreren Bildern Rückschlüsse auf Form und Position der abgebildeten Gegenstände gezogen und ein Modell der abgebildeten Szene im dreidimensionalen Raum berechnet werden.

Der von [Pavlidis, 1982] hierfür verwendete Begriff der *Mustererkennung* wird heute für ein Teilgebiet der Bildanalyse verwendet. Gegenstand dieses Faches ist die Entdeckung und Beschreibung von Mustern in Bildern. Diese Muster helfen dann, neben weiteren Informationen, bei der folgenden Analyse des Bildes Rückschlüsse auf dessen Inhalt zu ziehen.

Bildgenerierung (Image Synthesis; Computergrafik [Meier, 1986]):

Werden aufgrund von Modellen Rasterbilder berechnet, so spricht man von Bildgenerierung. Die Bildgenerierung deckt sich wiederum mit dem von [Pavlidis, 1982] als ‘Grafik’ umschriebenen Teilgebiet.

Modellierung und Algorithmische Geometrie (Geometric Modeling, Computational Geometry):

Gegenstand dieses Teilgebietes ist die Modellierung, Speicherung und Verarbeitung geometrischer Informationen und beschränkt sich dabei auf den Objektraum. Wichtig sind numerische und mathematische Verfahren zur Beschreibung der Gestalt von zwei- und dreidimensionalen Objekten sowie Algorithmen zur Berechnung der geometrischen und topologischen Eigenschaften.

In Abschnitt 4.2 wird diese Gliederung weiterentwickelt und dann als Grundlage von **BOOGA** dienen.

3.3 Anwendungsgebiete von **BOOGA**

Gerade im Gebiet der Computergrafik existieren viele Systeme, einerseits kommerziellen Ursprungs oder *Public Domain*, doch sind diese Systeme häufig sehr stark auf einzelne Teilgebiete spezialisiert. Für eine detailliertere Besprechung einer Auswahl von Grafiksystemen sei auf [Streit, 1997] verwiesen.

Der Leitgedanke beim Entwurf von **BOOGA** war, ein System zu schaffen, das nicht auf ein einzelnes der in Abschnitt 3.2 besprochenen Teilgebiete der Computergrafik beschränkt ist, sondern das Anwendungsgebiet umfassend abdeckt. Aufgrund der zum Zeitpunkt der ersten Entwurfsüberlegungen zu **BOOGA** bereits bearbeiteten Themen³, sowie der damals geplanten oder sich in Arbeit befindenden, weiteren Forschungsarbeiten⁴, bestand die Anforderung nach einem umfassenden und softwaretechnisch ausgereiften System, das als integrierende Plattform zukünftige Arbeiten kanalisiert und deren softwarebezogene Resultate in einem einheitlichen Format für weiterführende Arbeiten einfach verwendbar machen kann.

Die im folgenden aufgeführten Arbeiten sollen die Breite des Anwendungsgebietes aufzeigen und damit die aus Sicht der Computergrafik wesentlichste Anforderung an **BOOGA** untermauern: ein allgemeines Modell für die ganze Breite dieses Gebietes zu finden.

Bildgenerierung

Bis zum Projektstart von **BOOGA** wurden viele der heute wichtigen Bildgenerierungsalgorithmen in verschiedenen Arbeiten intensiv untersucht und so ein breit abgestütztes Fachwissen aufgebaut.

- [Amann, 1991; Bebie, 1994; Stainhauser und Spiess, 1993] betrachtete Spezialitäten des bereits seit längerer Zeit bekannten Z-Buffer-Verfahrens, das bei Anwendungen mit besonders grossen Ansprüchen an die Darstellungsgeschwindigkeit eingesetzt wird, also beispielsweise bei allen interaktiven Grafikanwendungen.
- [Collison, 1990; Fabregas, 1990; Habegger, 1994] untersuchten das Verfahren Raytracing, das bereits sehr gute, fotorealistische Bilder bei einer für Standbilder in nicht-interaktiven Anwendungen akzeptablen Geschwindigkeit liefert.

³vgl. [Amann, 1991; Amann, 1993; Bühlmann, 1991; Bühlmann, 1992; Collison, 1990; Collison, 1994; Dubuis, 1991; Fabregas, 1990; Stainhauser und Spiess, 1993; Streit, 1993]

⁴vgl. [Bebie, 1994; Bühlmann, 1994; Dubuis, 1995; Frei, 1996; Habegger, 1994; Hofer, 1994; Mani, 1994]

- Die Technik, welche die realistischsten Bilder liefert, leider aber sowohl von der Programmierung her, wie auch bezüglich Rechenzeit mit grossem Abstand am aufwendigsten ist, heisst Radiosity und wurde in [Dubuis, 1991; Dubuis, 1995] bearbeitet.
- Eine Kombination der beiden letztgenannten Verfahren wurde schliesslich in [Amann, 1993] vorgenommen.

Verschiedene Arbeiten im Bereiche der Modellierung erleichterten die Erstellung komplexer Modelle und erhöhten die Attraktivität der resultierenden Bilder durch die damit verbundene grössere Realitätsnähe.

Geometrische Modellierung

- [Streit, 1993] behandelt die Modellierung komplexer Objekte mittels *Lindenmayer-Systemen (L-Systeme)*. L-Systeme erlauben die Modellierung von Pflanzen und deren Wachstum.
- [Mani, 1994] erweiterte einen bestehenden Raytracer um die Möglichkeit, Flammen in einem Bild oder einer Animation zu verwenden.
- [Bühlmann, 1991] realisierte einen grafischen Editor zum interaktiven Erstellen von Rotationskörpern.
- In [Bühlmann, 1992] wurden digitale Geländemodelle sowie Satellitenbilder dazu verwendet, Modelle *realer* Landschaften zu erzeugen.
- [Bühlmann, 1994] setzte sich detailliert mit dem Problembereich der *Virtual Reality* und der speziellen Interaktionstechniken auseinander.
- Vertiefende Erfahrungen der Fachgruppe im Bereich von interaktiven Editoren zum Erstellen von Szenen wurden in [Hofer, 1994] gewonnen. Neben den technischen Grundlagen wurden hier auch die gerade in diesem Bereich wichtigen psychologischen Aspekte behandelt.

Obwohl die Gebiete der Bildverarbeitung und der Bildanalyse am Institut für Informatik (IAM) durch die Fachgruppe für künstliche Intelligenz abgedeckt werden, entstanden auch in der Fachgruppe Computergeometrie und Grafik Ideen für Projekte, die mehrere Gebiete miteinander vereinen. Beim Projektstart von **BOOGA** war abzusehen, dass diese Kombinationen in der nahen Zukunft an Gewicht gewinnen würden.

Bildverarbeitung, Bildanalyse

3.4 Anforderungen an **BOOGA**

BOOGA wird je nach Kontext als Forschungsplattform oder als Grafiksystem bezeichnet. Während die Bezeichnung *Plattform* den *Anwendungszweck* von **BOOGA** umreisst, nämlich als Grundlage für Forschungsarbeiten eingesetzt zu werden, wird durch die Einordnung als *Grafiksystem* die *Struktur* von **BOOGA** beschrieben.

Ein *Grafiksystem* stellt keine eigenständige Anwendung dar, sondern ist ein Hilfsmittel zu deren Erstellung, spezialisiert auf den Bereich der Computergrafik. Grafiksysteme werden mit Hilfe einer der in Kapitel 2 besprochenen Wiederverwendungstechnologien aufgebaut, in der Regel handelt es sich um Bibliotheken oder Frameworks.

Grafiksysteme sind meist auf einzelne Anwendungsbereiche innerhalb der in Abschnitt 3.2 aufgeführten Teilgebiete der Computergrafik spezialisiert. Sollen gebietsübergreifende Lösungen entwickelt werden, so müssen unterschiedliche Systeme kombiniert werden. Dies ist aus mehreren Gründen ein schwieriges Unterfangen. Einerseits stellen sich technische Probleme bei der Kombination verschiedener Bibliotheken oder Frameworks (vgl. Kapitel 2), andererseits definiert jedes System seine eigenen Abstraktionen und baut auf unterschiedlichen architektonischen Visionen auf. In einer Forschungsumgebung, in der teilweise unkonventionelle Ansätze verfolgt und Problemstellungen definiert werden, welche mehrere Gebiete der Computergrafik umfassen, ist es ein naheliegender Wunsch, ein **einheitliches, umfassendes Modell** für alle in Abschnitt 3.2 aufgeführten Teilgebiete zu finden. Ein System, das auf einem solchen Modell basierend implementiert wird, kann für eine Vielzahl von Anwendungen innerhalb des Anwendungsbereiches der Computergrafik eingesetzt werden.

Ein gemeinsames, einheitliches Modell stellt die Grundlage für die **Wiederverwendung von Konzepten** dar. Diese ist wiederum die Basis, um ein Softwaresystem zu realisieren, das auch die **Wiederverwendung von Implementationen** ermöglicht. Diese beiden Wiederverwendungsarten bilden die Grundlage für **BOOGA** als gemeinsame Forschungsplattform.

Das nächste wichtige Ziel leitet sich ebenfalls aus dem Bedürfnis nach einer einheitlichen Forschungsplattform ab: stärker noch als in produktorientierten Umgebungen besteht in Forschungsgruppen der Wunsch nach **grösstmöglicher Flexibilität und Erweiterbarkeit**.

Wie bereits in Kapitel 2 erwähnt, bringt zunehmende Flexibilität auch eine stark erhöhte Komplexität mit sich, sofern keine besonderen Vorkehrungen getroffen werden. Ein System, dessen Anwendung sehr aufwendig

ist, wird nicht die nötige Akzeptanz finden, um breit eingesetzt zu werden. Daraus ergibt sich unmittelbar die nächste wichtige Anforderung an **BOOGA**, die in direkter Konkurrenz zu der verlangten Flexibilität steht: die **Einarbeitung** in und die **Anwendung** von **BOOGA** soll so **einfach** wie nur möglich sein.

Es ist wichtig, an dieser Stelle anzumerken, dass dieser Konflikt zwischen Flexibilität und Komplexität durchaus nicht nur im Zusammenhang mit der Ausrichtung von **BOOGA** als Forschungsplattform zu sehen ist, wenngleich er dadurch verschärft wird. Im Folgenden werden die Richtlinien erläutert, die bei der Entwicklung von **BOOGA** dazu geführt haben, dass Flexibilität und einfache Anwendung innerhalb desselben Systems möglich sind.

Lokale Erweiterbarkeit

Ein System zu erweitern, dessen Komponenten eine enge Kopplung aufweisen, ist offensichtlich ein mit der Grösse des Systems zunehmend schwierigeres Unterfangen. Wie bereits in Abschnitt 2.4.1 (vgl. Abbildung 2.11 auf Seite 36) besprochen wurde, weisen auch andere Autoren auf die Notwendigkeit einer Entkopplung voneinander nur lose abhängiger Systemteile hin. In **BOOGA** wurde strikt darauf geachtet, dass Änderungen oder Erweiterungen in einem Teil des Systems keine Auswirkungen auf das restliche System haben.⁵

Ein Beispiel, das die Forderung nach lokaler Erweiterbarkeit besonders gut unterstreicht, ist die unabhängige Erweiterbarkeit von Daten und Operationen innerhalb von **BOOGA**. Die lokale Erweiterbarkeit führt dazu, dass ein Entwickler Probleme lokal in einem Teil des Systems lösen kann und sich nicht um die Integration in das Gesamtsystem zu kümmern braucht; er muss nur sehr wenige über seine direkte Aufgabe hinausgehende Kenntnisse des Systems haben.

Der 95%-Fall

Wird ein System so entwickelt, dass die wiederverwendbaren Softwareelemente in den am häufigsten auftretenden Fällen sehr einfach eingesetzt werden können und zusätzliche Parameter oder Methodenaufrufe nur nötig sind, wenn Aufgaben gelöst werden müssen, die nicht dem Standard entsprechen, so wird auf elegante Art grösstmögliche Flexibilität (falls nötig) mit einfacher Anwendbarkeit kombiniert.

⁵ Ausnahmen von dieser Forderung waren allerdings nötig, wenn das System im Verlauf einer Konsolidierungsphase von Altlasten befreit wurde, die sich bedingt durch den iterativen Entwicklungsprozess ansammelten. Trotzdem konnten in den meisten Fällen die auf dem Grafiksystem aufbauenden Anwendungen von grösseren Anpassungen verschont werden.

Mit Hilfe dieser Regel lässt sich das Problem der zunehmenden Komplexität elegant umgehen, indem der Anwender nur dann damit konfrontiert wird, wenn er auf Flexibilität angewiesen ist. In diesem Fall wird er aber auch bereit sein, den damit verbundenen, höheren Preis zu bezahlen.

Flexibilität wo nötig

Die 95%-Regel löst nur die Komplexitätsprobleme des *Anwenders* bereits existierender, flexibler Softwareelemente. Deren *Entwickler* hat aber ebenfalls mit der zunehmenden Komplexität zu kämpfen. Wird das Gesamtsystem in jedem Detail vollständig flexibel gehalten, so kann auch die vorausgehend besprochene Regel nicht mehr helfen, die daraus resultierende Komplexität im Griff zu behalten.

Flexibilität muss dort zur Verfügung gestellt werden, wo sie *benötigt* wird, nicht dort, wo sie *möglich* ist. Den richtigen Umfang an Flexibilität zu finden ist ein sehr schwieriges Unterfangen. Hilfe kann dabei das bereits in Abschnitt 2.4.1 erwähnte *Hot-Spot Driven Design* leisten. Die Teile des Systems, die mit hoher Wahrscheinlichkeit Änderungen unterworfen sind, müssen identifiziert und anschließend nur *diese* flexibel gestaltet werden.

Bei einer Forschungsumgebung wie **BOOGA**, die tendentiell einer grösseren Dynamik unterworfen sein wird als beispielsweise ein Framework zur Unterstützung der Programmierung graphischer Benutzeroberflächen, können sich diese *Hot-Spots* im Lauf der Zeit ändern. Das anfängliche Design war nicht darauf ausgerichtet, alle möglichen zukünftigen Anwendungsbereiche abzudecken, sondern nur dort Flexibilität anzubieten, wo die Wahrscheinlichkeit zusätzlicher Anforderungen ausreichend gross war. Dieser Ansatz verlangt fast zwangsläufig nach einem iterativen Entwicklungsansatz.

Iterative Entwicklung

A complex system that works is invariably found to have evolved from a simple system that worked.

[Gall, 1986]

Die Entwicklung von **BOOGA** basiert auf einem iterativen Ansatz. Neue Anwendungen werden einzelne, neue Anforderungen an das System stellen und somit in Änderungen desselben resultieren. Obwohl die ersten Anwendung naturgemäß mehr Erweiterungen zur Folge hatten als spätere, sind bei neuen Projekten Anpassungen von **BOOGA** nie ganz auszuschliessen. Ein System, das mittels eines

iterativen Prozesses entstanden ist und diesen konsequent einplant, hilft die Komplexität zu bewältigen, indem in jedem Iterationsschritt nur gerade die aktuellen Probleme zu bewältigen sind. Es ist nicht nötig, den Überblick über alle Details zu behalten. Wichtig ist hierbei aber die Feststellung, dass Iterationen über das Design und die Implementierung zu erfolgen haben, *nicht aber über die Visionen und grundlegenden Konzepte!*

Wenig unterschiedliche Konzepte

Jedes neue Konzept das eingeführt wird, ist eine weitere Hürde für aktuelle und neue Anwender des Systems und macht die Lernkurve immer steiler. Es ist massgebend für den Erfolg eines Systems, die Anzahl der unterschiedlichen Konzepte so klein wie möglich zu halten. Der Begriff *Konzept* soll in diesem Zusammenhang sehr allgemein verstanden werden; auf allen Abstraktionsstufen werden gewisse Konzepte verwendet, angefangen von Architektur- und Entwurfsmustern, über sprachspezifische Idiome, bis hin zu Namenskonventionen und Implementierungsrichtlinien.

Nachdem in diesem Kapitel die grundlegenden Anforderungen aus Sicht der Computergrafik und des Softwareengineering eingeführt wurden, beschreibt das nächste Kapitel die Architektur von **BOOGA**.



Thomas Cole.
Der Traum des Architekten (Ausschnitt).
1840.

Kapitel 4

Die Architektur von *BOOGA*

Architecture

*Ah, to build, to build!
That is the noblest art of all the arts.
Painting and sculpture are but images,
Are merely shadows cast by outward things
On stone or canvas, having in themselves
No separate existence. Architecture,
Existing in itself, and not in seeming
A something it is not, surpasses them
As substance shadow.*

Henry Wadsworth Longfellow
1807-82

4.1 Einleitung

In Abschnitt 2.4 wurde besprochen, was unter einer Architektur zu verstehen ist:

Die Architektur von Software beinhaltet die Beschreibung der Elemente, aus denen Systeme aufgebaut werden, der Interaktionen zwischen diesen Elementen, der Muster, die deren Zusammenspiel koordinieren sowie allfällige Einschränkungen dieser Muster. Beispiele für strukturelle Fragestellungen sind die Art, wie die einzelnen Systemteile organisiert werden, die globale Kontrollstruktur, die verwendeten Protokolle zur Kommunikation, Synchronisation und Datenzugriff, die Zuordnung von Funktionalität zu den einzelnen Entwurfselementen, die Zusammensetzung von Entwurfselementen, die physische Verteilung, Skalierbarkeit, Leistungsfähigkeit und Möglichkeiten der Weiterentwicklung.

Dieses Kapitel gibt einen Überblick über die Architektur von **BOOGA**. Die in Abschnitt 4.2 vorgestellte *logische Sicht* von **BOOGA** beschreibt die elementaren Entwurfselemente sowie die globale Kontrollstruktur. Die physische Schicht, welche in Abschnitt 4.3 besprochen wird, zeigt, wie die verschiedenen Abstraktionen mit Hilfe eines Schichtenmodells strukturiert werden.

Nicht alle Elemente in einem Komponentenframework sind den gleichen Anforderungen bezüglich Flexibilität ausgesetzt. Folglich ist ein wichtiger Teil einer Architekturbeschreibung eines auf Wiederverwendung und Flexibilität ausgerichteten Systems die Beschreibung der Stellen, die eine hohe Flexibilität aufweisen müssen. Diese *Hot-Spots* werden in Abschnitt 4.4 besprochen.

Gewisse weitere, allgemeine Architekturbestandteile werden in Abschnitt 4.5 zusammengefasst.

Abschnitt 4.6 wird aufzeigen, wie die in Abschnitt 3.4 aufgestellten Anforderungen mit den in diesem Kapitel vorgestellten Konzepten erreicht werden können.

Dieses Kapitel beschreibt die Architektur von **BOOGA** nur in groben Zügen. Das Ziel ist, dem Leser die nötigen Grundlagen zum Verständnis der folgenden Ausführungen mitzugeben. Eine wesentlich detailliertere Besprechung dieses Themas erfolgt in [Streit, 1997].

4.2 Die logische Sicht

Die in Abbildung 4.1 dargestellte Sicht ist zentral für **BOOGA**. Damit lassen sich nicht nur die wesentlichsten Konzepte erläutern, sie dient auch als Grundlage zur Klassifizierung von neuen Softwareelementen, die im Rahmen von **BOOGA** entwickelt werden (vgl. Kapitel 6) sowie als Hilfsmittel zur Dokumentation dieser Elemente (vgl. Kapitel 7).

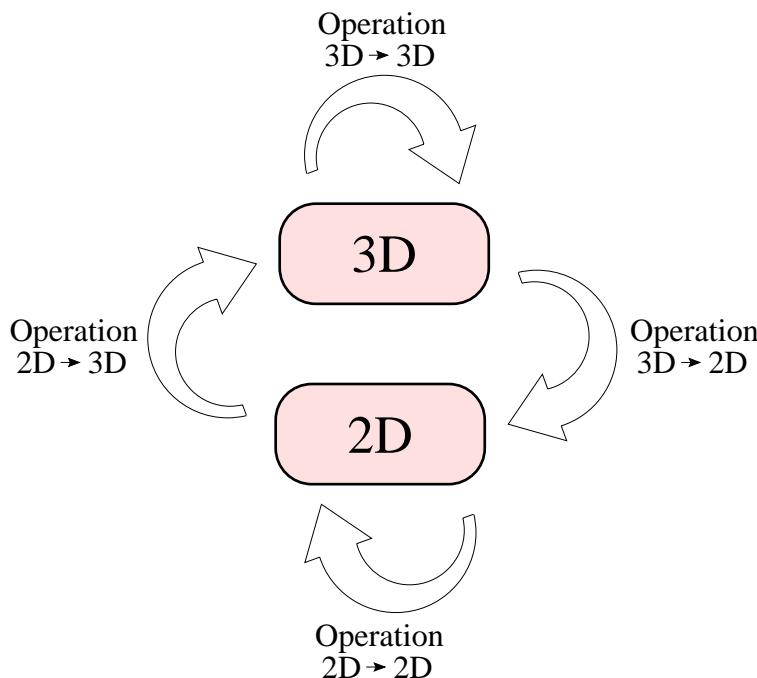


Abbildung 4.1

Die logische Sicht von **BOOGA** basiert im wesentlichen auf der in Abschnitt 3.2, Abbildung 3.2 vorgestellten Gliederung des Anwendungsgebietes.

Vergleicht man die beiden Abbildungen 4.1 und 3.2 miteinander, so fällt eine ähnliche Struktur auf, obwohl sich die Bezeichnungen unterscheiden. Tatsächlich ist die logische Sicht von **BOOGA** unter anderem von der Gliederung des Anwendungsgebietes Grafik nach [Meier, 1986] inspiriert. Weitere Grundlagen unseres Modells sind das Prinzip der *Renderingpipeline* [Foley et al., 1990] und die *Datenflussmodellierung* [Martin und McClure, 1985; McClure, 1989].

Als unschön an der Gliederung nach [Meier, 1986] als Grundlage für die Definition eines allgemeinen Modells der Computergrafik wird die ungleiche Verteilung der grafikrelevanten Datenstrukturen empfunden. Während der Bildraum ausschliesslich als Matrix zur Speicherung von Pixeldaten definiert wird, umfasst der Objektraum ‘alles andere’. Dieses Ungleichgewicht störte anfänglich vor allem aus ‘ästhetischen’ Gründen, führte bei genauerer Analyse aber zu schwerwiegenderen Problemen, sollte die Gliederung nach [Meier, 1986] als Grundlage für unser System Verwendung finden.

Betrachtet man die Übergänge zwischen beiden Räumen etwas genauer, so fällt auf, dass gewisse Operationen auf Daten der zweidimensionalen Computergrafik ausgeführt werden, andere auf Daten der dreidimensionalen Computergrafik. Diese explizite Unterscheidung tritt aber in Abbildung 3.2 (Gliederung nach Meier) nicht zu Tage.

2D

Der Raum der zweidimensionalen Datenstrukturen umfasst in **BOOGA** nicht nur Rastergrafik, sondern auch alle anderen Arten von Datenstrukturen zur Modellierung zweidimensionaler Grafiken. So können beispielsweise zweidimensionale Grafiken als Vektorgrafiken abgelegt und erst zur Ausgabe auf den Bildschirm ins Rasterformat umgewandelt werden.

3D

Analog gibt es auch für die dreidimensionale Computergrafik die unterschiedlichsten Datenstrukturen: manche grafischen Objekte lassen sich analytisch, d.h. mittels mathematischer Formeln beschreiben, andere nur als ein Netz aus Polygonen oder als Menge von Voxels¹.

Zwischen diesen beiden Räumen, dem Raum der Datenstrukturen für zweidimensionale Grafik und dem Raum der dreidimensionalen Grafik lassen sich Übergänge definieren. Ein erster Versuch, diesen Übergängen, ähnlich wie [Meier, 1986] Namen aus dem Anwendungsgebiet der Grafik zu geben scheiterte, da jede solche Benennung schliesslich eine Einschränkung bedeutet hätte. Für jeden Übergang lassen sich Anwendungen² finden, welche sich nicht einer solchen Klassifizierung unterordnen lassen. Aus diesem Grunde wurden die Übergänge sehr allgemein als *Operation3D→3D*, *Operation3D→2D*, *Operation2D→2D* und *Operation2D→3D*³ bezeichnet.

Tabelle 4.1 zeigt, wie sich die unterschiedlichen Teilgebiete der Computergrafik auf die verschiedenen **BOOGA**-Operationen verteilen lassen. Die aufgeführten Teilgebiete der Computergrafik decken das Anwendungsspektrum der jeweiligen Operationen nicht ab, sondern illustrieren nur deren Möglichkeiten.

Wie in Kapitel 5 noch genauer besprochen wird, setzt sich jede **BOOGA**-Anwendung aus einer Anzahl von Komponenten zusammen, die jeweils klar definierte Teilaufgaben erfüllen. Jede dieser Komponenten kann dabei genau einem der Übergänge zugeordnet werden. Die den **BOOGA**-Komponenten zugrundeliegende Definition wurde in Abschnitt

¹Mit *Voxel* wird das räumliche Äquivalent von Pixeln bezeichnet (*Pixel* = Picture Element, *Voxel* = Volume Element).

²In Abschnitt 6.4 wird eine Anwendung vorgestellt, der eine für ein Grafiksystem eher untypische Problemstellung zugrundeliegt.

³In Quellcodedateien wurde der Pfeil durch ein ‘To’ ersetzt, so dass die Übergänge dort die Namen *Operation3D To 3D*, *Operation3D To 2D*, *Operation2D To 2D* und *Operation2D To 3D* tragen.

BOOGA-Operation	Teilgebiet der Computergrafik
<i>Operation3D→3D</i>	Modellierung (in 3D) Geometrische Datenverarbeitung
<i>Operation3D→2D</i>	Bildgenerierung 'Computergrafik' nach [Meier, 1986]
<i>Operation2D→2D</i>	Modellierung (in 2D) Bildverarbeitung Geometrische Datenverarbeitung
<i>Operation2D→3D</i>	Bildanalyse Computervision

Tabelle 4.1

Teilgebiete der Computergrafik, welche durch die jeweiligen spezifischen **BOOGA**-Operationen abgedeckt werden. Die aufgeführten Teilgebiete illustrieren die Anwendungsbereiche für die Operationen, es sind aber durchaus auch andere Anwendungsbereiche möglich.

2.5 ausführlich besprochen (vgl. Definition 2.15). In dieser Arbeit werden die Begriffe **BOOGA-Komponente** und **BOOGA-Operation** synonym verwendet. Wenn der Kontext eindeutig ist, so wird in der Regel auf den Zusatz '**BOOGA**' verzichtet.

Im folgenden werden einige Beispiele für solche Komponenten aufgeführt. Das Schwergewicht liegt dabei weder auf einer vollständigen Aufzählung der realisierten Komponenten, noch auf den Algorithmen, auf denen diese aufbauen. Die Beispiele sollen lediglich ein Gefühl dafür vermitteln, was eine **BOOGA**-Operation darstellen kann.

In diese Gruppe von Operationen gehören all jene, welche Daten des dreidimensionalen Raumes erzeugen oder modifizieren, ohne dabei deren Dimension zu verändern:

Operation3D→3D

Einlesen von Szenen: Szenenbeschreibungen müssen gespeichert und zu einem späteren Zeitpunkt wieder eingelesen werden können. Mit Hilfe einer Leseoperation, die beispielsweise in einem grafischen Editor Verwendung findet, kann eine Datenstruktur angelegt werden, welche die Szene in einem zur Weiterverarbeitung geeigneten Format im Rechnerspeicher beschreibt.

Speichern von Szenen: Manuell oder automatisch modifizierte Szenen sollen in der Regel für eine spätere Verwendung konserviert werden können. Spezielle **BOOGA**-Komponenten übernehmen das Speichern von Szenen in unterschiedlichen Formaten.

Modellierung: Während einfache Objekte wie Würfel, Kugeln oder Polygone in der Regel vordefiniert sind und ohne weiteren Aufwand in eigenen Szenen- oder Objektbeschreibungen Verwendung finden können, müssen für komplexere Objekte, wie beispielsweise Pflanzen, andere Wege gefunden werden. So kann eine spezielle Komponente die Beschreibung einer Pflanze in Form einer *L-System Gram-*

matik in einfache, für jeden Bildberechnungsalgorithmus verständliche grafische Objekte zerlegen.

Weitere Modellierungsaufgaben können ganze Szenen betreffen. So ist beispielsweise die bekannte Clippingoperation (vgl. [Foley et al., 1990]) als eine **BOOGA**-Komponente zu realisieren. Denkbar ist auch eine ‘*Dalí-Operation*’, welche die in einer Szene enthaltenen Objekte zerfliessen lässt.

Editoren: Obschon gerade zu Beginn der Entwicklung von **BOOGA** der einzige Weg, eine Szene zu definieren, der einer textuellen Beschreibung der in ihr enthaltenen grafischen Objekte war, ist diese Art von grafischer Modellierung natürlich alles andere als benutzerfreundlich. Interaktive, grafische Editoren sind weitere Beispiele von **BOOGA**-Komponenten.

Operation3D→2D Die Operationen dieser Gruppe führen einen Übergang vom dreidimensionalen Raum in den zweidimensionalen Raum durch. Neben dem offensichtlich dieser Gruppe zugehörigen *Rendering* (Bildberechnung) wurden auch andere Komponenten realisiert:

Rendering: Die Berechnung einer zweidimensionalen Ansicht einer dreidimensionalen Szene unter Berücksichtigung von Eigenschaften der *virtuellen Kamera* und Lichtquellen kann mit unterschiedlichen Verfahren, einerseits zum Bestimmen der Sichtbarkeiten, andererseits zum Berechnen der Schattierungen erfolgen. Verschiedene Komponenten realisieren Algorithmen wie *Wireframe-Rendering*, *Z-Buffer* oder *Raytracing*.

Antialiasing: Um die sichtbaren Auswirkungen von Abtastfehlern (*Aliasing*) zu verringern, existieren unterschiedliche *Antialiasing*-Verfahren. Ein einfaches Verfahren basiert auf der Idee, ein Bild einfach in einer grösseren Auflösung zu berechnen und anschliessend auf das ursprünglich gewünschte Format zu verkleinern. Dabei werden mehrere Bildpunkte des berechneten Bildes auf einen Bildpunkt des korrigierten Bildes abgebildet. Die Abtastfehler werden durch diese Mittelwertbildung deutlich reduziert.

Strukturvisualisierung: Wird eine Szene in einem grafischen Editor erstellt, so kann der ‘Autor’ wohl die visuelle Wirkung überprüfen, selten aber die Struktur, den Aufbau der Szene. Diese Struktur ist aber gerade in der Computergrafik wichtig, da sie die nötige Rechenzeit für ein Bild massiv beeinflusst. Spezielle Komponenten können diese Struktur visualisieren und dem Benutzer so helfen, die Szene hinsichtlich der Rechenzeit zu optimieren.

Diese Operationen sind vergleichbar mit Operation $3D \rightarrow 3D$. Hier sind Eingabedaten *und* Ausgabedaten allerdings zweidimensional. Grundsätzlich sind Komponenten mit sehr ähnlichen Aufgaben wie im dreidimensionalen Fall sinnvoll. Die folgende Liste zeigt deshalb nur einige zusätzliche Operationen auf.

Operation $2D \rightarrow 2D$

Rasterkonvertierung: Wie bereits erwähnt, sind im **BOOGA**-Modell nicht nur Bilder im zweidimensionalen Raum enthalten, sondern auch alle anderen Datenstrukturen der zweidimensionalen Computergrafik. Wird beispielsweise eine Wireframekomponente (Operation $3D \rightarrow 2D$) ausgeführt, so ist das Resultat folgerichtig eine Menge von Strecken. Sollen diese auf einem Bildschirm ausgegeben werden, so muss zuerst eine *Rasterkonvertierung* stattfinden. Dies ist die Aufgabe einer speziellen Komponente.

Bildraumoperationen: Auf Rasterbildern sind eine Vielzahl von Operationen möglich. Zu erwähnen sind beispielsweise die Umwandlung zwischen unterschiedlichen Farbmodellen, die nachträgliche Farbkorrektur, Helligkeitsanpassungen, Kantendetektionen oder gar so komplexe Aufgaben wie die bereits in Abschnitt 3.2 erwähnten Filter zur Erzeugung von Kunsteffekten.

Diese Operationen gehen von Daten des zweidimensionalen Raumes aus und berechnen daraus Informationen des dreidimensionalen Raumes. Dies kann beispielsweise dazu verwendet werden, um die Kameraparameter eines Bildes zu bestimmen, wenn die dargestellte Szene, respektive deren grundsätzliche Struktur bekannt ist.

Operation $2D \rightarrow 3D$

Rekonstruktion: Anhand von einem oder mehreren Bildern und allfälligen Zusatzwissen können Daten über die dargestellten dreidimensionalen Objekte extrahiert werden. Dies erlaubt beispielsweise die automatisierte Erfassung von Häusern, was Voraussetzung für die effiziente und kostengünstige Modellierung von Strassenzügen oder ganzen Städten ist.

Editoren: Mit Hilfe spezieller Editoren, die Zusatzwissen über den Modellierungsbereich besitzen, können dreidimensionale Szenen mittels zweidimensionaler Skizzen eingegeben werden. Dies erlaubt eine wesentlich einfachere Bedienung als die oft recht komplexen und vom Benutzer viel Abstraktionsvermögen fordernden bekannten Ansätze für 3D-Editoren.

Diese exemplarischen Betrachtungen der **BOOGA**-Operationen schliessen die Besprechung der logischen Sicht auf **BOOGA**. In den folgenden Kapiteln wird jedoch noch vermehrt Bezug auf Abbildung 4.1 und diesen Abschnitt genommen.

4.3 Die physische Sicht: ein Komponentenframework

Mit der im letzten Abschnitt vorgestellten logischen Sicht auf *BOOGA* wird der grundlegende Forderung (vgl. Abschnitt 3.4) nach einem umfassenden, einheitlichen Modell für den gesamten Anwendungsbereich der Computergrafik Rechnung getragen. Auch für die Wiederverwendung von Konzepten sowie für einen einheitlichen Applikationsbegriff wird hierdurch das Fundament bereitet. Viele Anforderungen, wie beispielsweise die verlangte Flexibilität, Erweiterbarkeit und Einfachheit, sind aber dadurch noch nicht abgedeckt.

BOOGA bietet Wiederverwendung auf unterschiedlichen Abstraktionsstufen. In Abschnitt 2.6 wurde besprochen, wie die verschiedenen Wiederverwendungstechnologien aufeinander aufbauen und sich so gegenseitig ergänzen können. Beim Design von *BOOGA* wurde genau diesen Überlegungen Rechnung getragen und eine *3-Schichtenarchitektur* realisiert (vgl. Abbildung 4.2). Während die unterste Schicht, die *Bibliotheksschicht*, ‘lediglich’ Wiederverwendung von Implementierungen gewisser Basisfunktionalitäten bietet, beinhaltet die darauf aufbauende *Frameworkschicht* bereits Wiederverwendung von Design und Implementierung. Die oberste Schicht schliesslich, die *Komponentenschicht*, ermöglicht, die im vorausgehenden Abschnitt besprochene logische, komponentenbasierte Architektur zu realisieren.

Abbildung 4.2

BOOGA baut auf einem Schichtenmodell auf. Die Schichten repräsentieren verschiedene Abstraktionsebenen (vgl. Abschnitt 2.6, Abbildung 2.32).



Für ein System, das wie *BOOGA* die Vorteile von Komponenten, Frameworks und Bibliotheken in der genannten Art kombiniert, wird in der Folge der Begriff *Komponentenframework* verwendet.

Im Folgenden wird kurz auf die einzelnen Schichten und deren Aufgaben eingegangen. Eine ausführlichere Besprechung der in den ein-

zernen Schichten enthaltenen Mechanismen und Abstraktionen ist in [Streit, 1997] zu finden.

Die Bibliotheksschicht enthält Klassen und Funktionen für allgemeine Aufgaben, wie beispielsweise die Behandlung von Zeichenketten, mathematische Funktionen, allgemeine Datenstrukturen oder Kapselungen des Betriebssystems.

Die Frameworkschicht führt geometrische Objekte für zwei- und dreidimensionale Computergrafik ein. Diese geometrischen Objekte sind die Daten, auf denen die **BOOGA**-Komponenten operieren und werden als gerichteter, azyklischer Graph organisiert.

Besteht die Aufgabe eines Anwenders darin, neue geometrische Objekte zu realisieren, so wird er sich innerhalb dieser Schicht bewegen und muss sein Objekt konform zu den vom Framework vorgeschriebenen Protokollen realisieren. Diese Aufgabe wird durch die bereits vorhandenen Basisklassen unterstützt und entspricht dem in Abschnitt 2.4.2 besprochenen Ansatz des *White-Box Reuse*.

BOOGA-Anwendungen sind immer aus einer Anzahl von Komponenten aufgebaut. Diese Komponenten akzeptieren Objekte der Frameworkschicht als Eingabe und erzeugen, je nach Aufgabe, allenfalls neue Objekte dieser Schicht. Werden neue Komponenten realisiert, so entspricht dies dem Ansatz des *Black-Box Reuse* bezüglich der Frameworkschicht.

Welche Vorteile bringt nun diese Schichtenarchitektur, bzw. ein Komponentenframework? Wie in Abschnitt 2.6 besprochen, wird erst eine Kombination unterschiedlicher, einander ergänzender Wiederverwendungstechnologien erfolgreiche Wiederverwendung ermöglichen und somit auch die in Abschnitt 3.4 umrissenen, ambitionierten Ziele zu erreichen helfen.

Wie in Abschnitt 2.3.2 besprochen, eignen sich *Bibliotheken* sehr gut, um isolierte Klassen oder Funktionen zur Wiederverwendung der Implementation zur Verfügung zu stellen, bieten aber selber keinerlei Hilfestellung auf höherer Abstraktionsstufe an. Bibliotheken sind einfach in der Anwendung und in der Regel rasch erlernt. All dies prädestiniert Bibliotheken dazu, auf unterster Stufe die grundlegenden, oft problemneutralen, systemnahen oder sprachspezifischen Abstraktionen⁴ zur Verfügung zu stellen.

Bibliotheksschicht

Frameworkschicht

Komponentenschicht

Vorteile der Schichtenarchitektur

⁴Beispiele für Abstraktionen der Bibliotheksschicht in **BOOGA** sind Zeichenketten (*String*), Vektoren (*Vektor2D*, *Vektor3D*), Matrizen (*TransMatrix2D*, *TransMatrix3D*), Listen (*List*), Stacks, Symboltabellen (*SymTable*).

Bibliotheken alleine würden die Wiederverwendung von Konzepten nur ungenügend unterstützen. Gleichzeitig würde es auf diese Weise auch schwerfallen, die Kompatibilität von in unterschiedlichen Arbeiten entwickelten Lösungen sicherzustellen. Dies führt zur nächsten, auf die Bibliotheken aufbauenden Schicht, der *Frameworkschicht*. Frameworks bieten, wie in Abschnitt 2.4.2 diskutiert, eine sehr grosse Flexibilität und Erweiterbarkeit und können durch die Umkehrung der Kontrolle (das *Hollywood-Prinzip*) zu einer einheitlichen, standardisierten Form der Applikationen führen, was die Wiederverwendung auf dieser höheren Stufe wesentlich vereinfacht. Frameworks haben allerdings auch gewichtige Nachteile. An dieser Stelle wesentlich ist die grosse Einarbeitungszeit, die nötig ist, um ein Framework einzusetzen oder gar zu erweitern. Ein Framework ist deshalb speziell in Bereichen von grossem Nutzen, in denen Anwender mit profunden Kenntnissen des Anwendungsgebietes, des spezifischen Frameworks, der Objekttechnologie sowie der verwendeten Sprache und Idiome Teillösungen für andere Anwender erstellen, welche diese dann als *Blackboxes* verwenden können.

Um die Nachteile von Frameworks auszugleichen, bieten sich die in Abschnitt 2.5 eingeführten Komponenten an. Vorteile von Komponenten in diesem Zusammenhang sind Reduktion der Kopplung zwischen Systemteilen, Wiederverwendung im Grossen und die Einfachheit der Anwendung.

Durch die Kopplung dieser drei Schichten zu einem Komponentenframework können die Vorteile der einzelnen Ansätze kombiniert werden, während sich die Nachteile durch diese Überlagerung stark reduzieren.

Ein Entwickler, der mit Hilfe eines Komponentenframeworks Anwendungen erstellt, wird sich solange wie möglich auf der höchsten Abstraktionsebene bewegen, welche durch die Komponentenschicht realisiert wird. Reichen die dort vorhandenen Abstraktionen nicht aus, kann auf der nächsttieferen Schicht, der Frameworkschicht, das Fehlende ergänzt werden. Dasselbe Vorgehen wiederholt sich beim Übergang zwischen der Framework- und der Bibliotheksschicht.

Durch die freie Erweiterbarkeit aller Schichten und klare Schnittstellen zwischen den Schichten ist eine Erweiterung um zusätzliche Funktionalität jederzeit möglich. Wird diese Flexibilität jedoch nicht benötigt, kann mit wenig Detailwissen über das Gesamtsystem, durch Komposition und Konfiguration, auf einfache Art eine Applikation aus bestehenden Elementen aufgebaut werden.

Ein Einsteiger kann sich somit schrittweise in ein solches System einarbeiten. Er kann schnell erste, motivierende Resultate erreichen und mit zunehmendem Wissen komplexere Anwendungen erstellen.

4.4 Hot-Spots

Im Projekt **BOOGA** wurden, neben dem anwendungsgebietbedingten Anspruch der Allgemeinheit des Modelles, die teilweise widersprüchlichen softwaretechnischen Anforderungen *Wiederverwendung*, *Flexibilität*, *Erweiterbarkeit* und *Einfachheit* zu den zentralen Anliegen erhoben. Sollen diese Anforderungen in einem im Anwendungsgebiet derart breit abgestützten Projekt umfassend realisiert werden, so wird die daraus resultierende Komplexität das Projekt fast zwangsläufig scheitern lassen. Sollte es jedoch wider Erwarten zu einem erfolgreichen Abschluss kommen, wird das resultierende System so komplex in der Anwendung und im Unterhalt, dass die Bereitschaft, dieses System tatsächlich einzusetzen, äusserst klein sein dürfte.

In der Regel ist es nicht nötig, ein Framework in allen Aspekten flexibel und erweiterbar zu halten, da diese Eigenschaften meist nur in Bezug auf einige wenige Bereiche benötigt werden. Diese Überlegung hat zum Begriff der *Hot-Spots* geführt, der in Abschnitt 2.4.1 auf Seite 35 eingeführt wurde. Basierend auf einer Analyse der bisher realisierten und geplanten Grafikanwendungen (vgl. Abschnitt 3.3) wurde festgestellt, dass die wesentlichen Erweiterungen in der Implementation neuer grafischer Objekte und neuer Operationen liegt. Das Augenmerk bezüglich Flexibilität, Erweiterbarkeit und Wiederverwendbarkeit konzentrierte sich daher auf diese Bereiche.

Als zusätzliche Anforderung wurde die *unabhängige* Erweiterbarkeit von grafischen Objekten und Operationen aufgestellt. Sobald jemand ein neues grafisches Objekt implementiert, müssen alle bereits bestehenden Operationen dieses Objekt sogleich verwenden können. In einem höchst dynamischen System, wie dies eine Forschungsplattform darstellt, ist es nicht annehmbar, dass bereits bestehende Komponenten angepasst werden müssen, sobald ein neues grafisches Objekt in den Katalog der wiederverwendbaren Elemente aufgenommen wird.

Im Bereich der dreidimensionalen Grafik wurde als weiterer *Hot-Spot* die flexible Behandlung von Texturen identifiziert. Mit Texturen lässt sich bei der Bildgenerierung eine deutlich erhöhte Realitätsnähe erreichen. In [Teuscher, 1996] konnte mittels des sehr allgemein gehaltenen Texturmodells von **BOOGA** (vgl. [Streit, 1997]) eine äusserst mächtige *Shading Language* realisiert werden.

4.5 Allgemeine Konzepte

Neben den bisher besprochenen Architekturelementen existieren weitere, allgemeine Fragestellungen, die vor dem Beginn der ersten Implementierungsphase abgeklärt werden müssen und teilweise auch als Elemente der Architektur betrachtet werden können. Beispiele hierfür sind:

- *Wie soll die Ausgabe von Fehlermeldungen erfolgen?*
- *Wie soll auf Fehler reagiert werden?*
- *Wie können Komponenten einheitlich konfiguriert werden?*
- *Wie sollen die Namen von Klassen oder Methoden vergeben werden?*
- *Wie sieht der grundlegende Aufbau einer Schnittstellenbeschreibung (C++-Headerdatei) aus?*
- *Wie können einzelne Komponenten in einheitlicher Form statistische Angaben ausgeben?*
- *Welches Format haben Ein- und Ausgabedateien zum Datenaustausch zwischen Anwendungen und der Speicherung von Daten?*

Es würde zu weit führen, auf all diese und ähnliche Fragestellungen an dieser Stelle detailliert einzugehen. Drei wichtige Problemkreise sollen dennoch genauer betrachtet werden, da diese ganz wesentlich die in Abschnitt 3.4 aufgeführten Anforderungen an **BOOGA** beeinflussen.

Komponentenkonfiguration

Wie bereits besprochen wurde, erhöht eine grösitere Flexibilität in aller Regel die Komplexität eines Systems. Dies ist vertretbar in Situationen, wo diese Flexibilität auch tatsächlich benötigt wird. Oft will ein Anwender aber möglichst viel Software wiederverwenden und damit lediglich eine Standardaufgabe lösen. Muss in diesem Fall ebenfalls die gesamte Komplexität beherrscht werden, so vermindert dies die Akzeptanz des Systems, da ein Anwender auch für einfachste Aufgaben bereits ein Experte sein muss. Wird hingegen die 95% Regel befolgt, die besagt, dass ein wiederverwendbares Softwareelement in 95% aller Fälle ohne Änderungen und ohne aufwendige Konfiguration verwendet werden können soll, so wird ein inkrementelles Lernen des Systems möglich. Ein neuer Anwender des Systems kann dieses zuerst einsetzen, um einfache Standardaufgaben zu lösen und muss sich dazu lediglich mit einigen grundsätzlichen Regeln auseinandersetzen. Steigen die Ansprüche des Anwenders, so wird er sich das zusätzlich nötige Wissen Stück für Stück an Hand seiner Anwendung

aneignen. Die Motivation des Anwenders, ein solches System einzusetzen, das mit seinen Bedürfnissen mitzuwachsen scheint, ist ungleich grösser als bei einem System, das dieses Verhalten nicht aufweist.

In **BOOGA** wurde Wert darauf gelegt, die 95% Regel zu befolgen. So können beispielsweise alle Komponenten ohne weitere Konfiguration in einer einheitlichen Art und Weise angewendet werden. Sollen komplexere Aufgaben damit erfüllt werden so gibt es mehrere Möglichkeiten, die Komponente zu konfigurieren.

Jede Komponente weist ein bestimmtes Standardverhalten auf. Dieses kann entweder mittels spezieller Methoden konfiguriert werden oder durch Änderung von Werten in der in **BOOGA** realisierten Konfigurationsdatenbank. Einige Beispiele sollen dies verdeutlichen:

- Es gibt verschiedene Komponenten, welche *Pixmaps* erzeugen. Es sind grundsätzlich unterschiedliche Typen von Pixmaps möglich, so können die Bilder beispielsweise in Graustufen umgerechnet oder mit 16,7 Millionen Farbabstufungen abgelegt werden. Da davon ausgegangen wird, dass in einer Standardanwendung meistens mit demselben Typ Bild gearbeitet wird, kann in der Konfigurationsdatenbank der Standardtyp für einePixmap gesetzt werden. jede Komponente, die eine neuePixmap erzeugen muss, wird in dieser Konfigurationsdatenbank den korrekten Typ vonPixmap abfragen. Soll eine bestimmte Komponente ausnahmsweise einen anderen Typ verwenden, so kann diese Einstellung bei dieser Komponente direkt mit Hilfe einer entsprechenden Methode verändert werden.
- Verschiedene Operationen sind darauf angewiesen, eine bestimmte Farbe anwenden zu können. So wird ein Renderer, sofern bei einem grafischen Objekt keine Oberflächenbeschreibung (Textur) mitgegeben wird, dieses Objekt mit einer Standardfarbe einfärben. Auch diese Farbe kann aus der Konfigurationsdatenbank ausgelesen werden.

Ein weiterer, wichtiger Aspekt, der die Verwendung eines Systems vereinfacht, betrifft die Namensgebung von Methoden. Es kommt in einem Framework öfter vor, dass Klassen in verschiedenen Hierarchien Methoden enthalten, die ähnliche Funktionalitäten erfüllen. Durch eine gleiche Benennung dieser Methoden muss sich ein Anwender weniger Namen merken, was die Anwendung ebenfalls vereinfacht. Es ist wichtig für ein System, an dem mehrere Entwickler mitarbeiten, dass eine einheitliche Form für die Schreibweise von Namen, für die Benennung von Methoden und Klassen, für die Ausgabe von Fehlermeldungen, etc. gefunden wird. Eine solche Form wurde für **BOOGA** in Form eines *Style Guides* (vgl.

Styleguidelines

Anhang B) schriftlich festgehalten und für alle Entwickler als verbindlich erklärt. Um zu erreichen, dass diese Richtlinie auch eingehalten wird, wurde sie bewusst sehr knapp gehalten.

Ein- und Ausgabeformate

Unterschiedliche, teilweise *BOOGA*-spezifische Dateiformate sind nötig, um den Datenaustausch mit anderen Anwendungen zu gewährleisten oder Ein- und Ausgaben zu speichern. Zwei wichtige *BOOGA*-eigene Dateiformate, die *BOOGA Scene Description Language* (BSDL) und das Format zum Speichern von Bildern (*Pixi*), werden in Anhang A beschrieben.

4.6 Die Architektur als Mittel zur Zielerfüllung

Dieses Kapitel hat die wesentlichen Elemente der Architektur von **BOOGA** aufgezeigt. Im Folgenden werden die in Kapitel 3 formulierten Ziele nochmals aufgenommen und besprochen, wie die vorgestellte Architektur zur Erfüllung dieser Ziele beiträgt.

Einheitliches und umfassendes Modell der Computergrafik

In Abschnitt 4.2 wurde dargelegt, wie das in Abbildung 4.1 dargestellte Modell den gesamten Bereich der Computergrafik abzudecken vermag. Komponenten, welche die unterschiedlichsten Aufgaben erfüllen, werden auf eine uniforme Art aktiviert und miteinander verknüpft.

Wiederverwendung von Konzepten und Implementation

Das einheitliche Modell erleichtert wesentlich die Definition wiederverwendbarer Konzepte. Durch ein einfaches, klares Komponentenmodell sowie ein Framework, welches das abstrakte Design sowie weite Teile der Implementierung beinhaltet, wird die Wiederverwendung von Konzepten und Implementation konsequent umgesetzt.

Einfachheit in der Anwendung

Durch die Konzentration der Flexibilität auf die definierten *Hot-Spots*, die Anwendung der 95%-Regel und die Umsetzung des *Style Guides* (vgl. Anhang B) konnte die Komplexität in weiten Teilen vor dem Anwender versteckt werden. Die unterschiedlichen Schichten des Komponentenframeworks tragen dazu bei, dass für die Realisierung bestimmter Aufgaben nötige Wissen auf bestimmte Bereiche des Gesamtsystems einzuschränken.

Flexibilität bezüglich Einsatzgebiet und Erweiterungen

Die logische Sicht beschreibt ein sehr einfaches, allgemeines Modell, das wenig Einschränkungen beinhaltet. Auch die durch das Framework definierten Protokolle (vgl. [Streit, 1997]) sind sehr einfach und allgemein gehalten.

Lokale Erweiterbarkeit

Das Komponentenframework **BOOGA** ist im wesentlichen für die Erweiterung an zwei unterschiedlichen Stellen vorgesehen:

1. Neue Operationen werden als Komponenten realisiert und in die Komponentenschicht von **BOOGA** integriert werden. Diese neuen Komponenten stehen fortan neuen Anwendungen zur Verfügung.

2. Neue Objekttypen für den zwei- oder dreidimensionalen Raum werden implementiert und in die Frameworkschicht eingefügt. Auch diese Objekte stehen neuen Anwendungen uneingeschränkt zur Verfügung.

Unter lokaler Erweiterbarkeit wird nun verstanden, dass *bestehende* Komponenten nicht angepasst werden müssen, wenn sie Objekttypen verarbeiten sollen, die zum Zeitpunkt der Komponentenprogrammierung noch gar nicht zur Verfügung standen. Umgekehrt sollen bestehende Objekttypen natürlich auch von zukünftigen, neuen Komponenten verarbeitet werden können. [Streit, 1997] beschreibt die in *BOOGA* implementierten Mechanismen, welche diese Flexibilität ermöglichen.



Andrea Mantegna.
Kampf der Seegötter.

Kapitel 5

Die Applikationsentwicklung mit *BOOGA*

*I cannot imagine any condition which
would cause a ship to founder*

*...
Modern shipbuilding has gone beyond that.*

Kapitän Edward J. Smith, sechs
Jahre vor seinem Kommando auf
der Jungfernfahrt der *Titanic*.

5.1 Einleitung

BOOGA mit seiner speziellen Architektur als Komponentenframework sowie den Mechanismen und Abstraktionen der verschiedenen Schichten beeinflusst die Architektur der darauf aufbauenden Anwendungen sowie deren Entstehung. Die entsprechenden Schlüsselemente wurden in den beiden vorangegangenen Kapiteln besprochen. Sobald konkrete Aufgaben gelöst werden sollen, bedarf es zusätzlich eines angepassten *Vorgehensmodelles (Methode)* zur Softwareentwicklung, welche aufzeigt, wie man unter Verwendung der Architekturvorgaben von einer Problemstellung zur fertigen Anwendung kommt.

Eine solche Methode beschreibt den Ablauf eines Softwareprojektes, beginnend mit der Problemstellung und endend mit einer Wartungsphase nach Abschluss der eigentlichen Entwicklung.

Das im folgenden diskutierte Vorgehensmodell behandelt sehr ausführlich den Prozess der Anwendungserstellung mit einem Komponentenframework wie **BOOGA**. In der Praxis wird die vorgeschlagene Methode selten so formal wie hier beschrieben eingehalten werden. Manche Teilschritte werden zusammengefasst, andere implizit, aufgrund der Erfahrung des Entwicklers durchgeführt. Das in diesem Kapitel beschriebene Vorgehen soll denn auch eher dem, mit der Anwendungsentwicklung mit Hilfe von Komponentenframeworks weniger erfahrenen Entwickler, einen Leitfaden in die Hand geben, der mit zunehmender Praxis verinnerlicht und weniger formal wird.

In der Geschichte des Software Engineering wurde eine Vielzahl von Methoden entwickelt, um datenzentriert, funktional oder objektorientiert Software zu entwickeln. Im folgenden Abschnitt wird aufgezeigt, warum bekannte Ansätze für die Entwicklung von Anwendungen, aufbauend auf einem System wie **BOOGA**, wenig geeignet sind. Anschliessend wird ein einfaches Vorgehensmodell skizziert, das bei einem komponentenzentrierten Ansatz zum Einsatz kommen kann.

Neben der Entwicklung von Anwendungen muss auch der Pflege des Frameworks sowie der Synchronisation dieser unterschiedlichen Aufgaben ein angemessenes Gewicht zugeordnet werden. Die Abschnitte 5.4.6 und 5.5 sind dieser Problematik gewidmet.

Im letzten Abschnitt dieses Kapitels wird diskutiert, wie sich die vorgestellte Methode auf andere Komponentenframeworks und darauf aufbauende Anwendungsentwicklungsprojekte verallgemeinern lässt.

Das nächste Kapitel (Kapitel 6) wird das hier besprochene Vorgehen anhand einiger Beispielanwendungen illustrieren.

5.2 Bekannte Vorgehensmodelle

Es existiert eine grosse Zahl an unterschiedlichen Methoden, welche ein Vorgehen für die Erstellung von Software, von der Problemstellung bis hin zur fertigen Anwendung, beschreiben. Diese Methoden beinhalten einerseits eine Sammlung von grafischen *Notationen* zur strukturierten Dokumentation von Analyse- oder Designresultaten, andererseits aber auch ein *Vorgehensmodell* (z.Bsp. Wasserfallmodell oder Spiralmodell). Dieses Vorgehensmodell zeigt unter anderem auf, welche Notation in einem bestimmten Stadium zur Anwendung kommen soll, welche Teilschritte nötig sind und wie die einzelnen Resultate dieser Teilschritte aussehen. Die am meisten verbreiteten Vertreter aus dem Bereich der objektorientierten Methoden sind [Booch, 1994b; Rumbaugh et al., 1991; Jacobson et al., 1992].¹ Diese Methodiker betonen, wie dies bei einer objektorientierten Methode zu erwarten ist, die Wichtigkeit der Wiederverwendung. Am deutlichsten drückt sich hierbei Jacobson aus, der die gesamte Softwareentwicklung in vier Prozesse aufteilt (vgl. Abbildung 5.1). Die drei Prozesse *Analyse*, *Konstruktion* und *Test* werden dabei für jedes

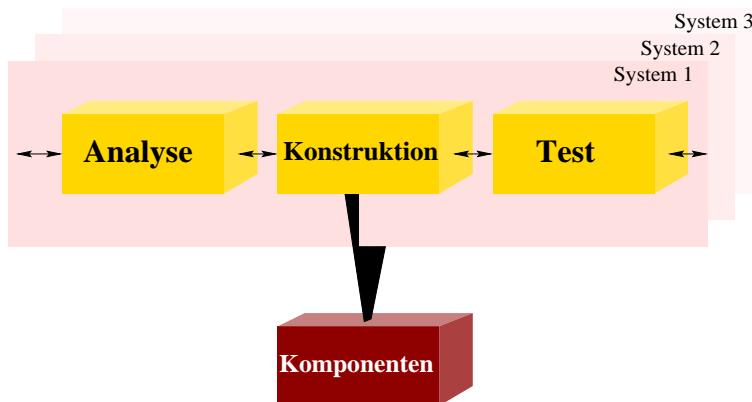


Abbildung 5.1

[Jacobson et al., 1992] unterteilt die Softwareentwicklung in vier voneinander abhängige Prozesse, wobei die drei oberen für jedes zu entwickelnde System initiiert werden, der untere hingegen idealerweise nur einmal pro Organisation.

neue System gestartet, der *Komponenten*-Prozess existiert idealerweise nur einmal pro Organisation (z. Bsp. Abteilung, Firma, Konzern) und dient der Wiederverwendung von einzelnen Softwarelementen während der Konstruktionsphase über Projektgrenzen hinweg.

Die bekannten Methoden unterstützen die Wiederverwendung von *architekturneutralen Softwareelementen*, im wesentlichen also von einzelnen Funktionen oder Klassen. Sobald komplexere Elemente aber Anforderungen an die Architektur eines Systems stellen, wie beispielsweise Kom-

¹Diese drei Methoden sind gegenwärtig einer Integrationsphase unterworfen. In einem ersten Schritt sind Grady Booch, James Rumbaugh und Ivar Jacobson übereinkommen, die *Notationen* in der *Unified Modeling Language* (UML) zu vereinen [Booch et al., 1997]. Die einzelnen Vorgehensmodelle, die grössere Unterschiede aufweisen als die jeweiligen Notationen, werden vorerst noch getrennt weiterentwickelt.

ponenten (Abschnitt 2.5) oder Frameworks (Abschnitt 2.4.2), in diesem Sinne also *architekturnspezifisch* sind, können diese Methoden nur noch bedingt eingesetzt werden, da die Wiederverwendung sich nur auf Softwareprodukte in der Konstruktionsphase² beschränkt.

Die drei erwähnten objektorientierten Methoden beschreiben im wesentlichen die bereits von den strukturierten Vorgehensweisen bekannten Teilschritte *Analyse*, *Design*, *Implementierung*, *Test* sowie *Integration*. Gemeinsam ist ebenfalls, dass in der Analysephase ein Modell des Anwendungsbereiches erstellt wird. Hierfür werden die Anwendungsobjekte anhand der Problemstellung identifiziert, diese gefundenen Objekte in Klassen eingeordnet, Relationen zwischen den Klassen hergestellt sowie Methoden definiert. Existieren bereits Frameworks oder Komponenten (im Sinne von Abschnitt 2.5), welche wiederverwendet werden sollen, so macht ein solches Vorgehen wenig Sinn, da die Architektur, sowie in der Regel die im Anwendungsgebiet gebräuchlichen Abstraktionen, bereits definiert wurden oder in ausführbarer Form vorhanden sind.

Die meisten Methoden sind auf die Entwicklung vollständig neuer Systeme ausgerichtet. Unter dieser Voraussetzung können im allgemeinen nur während der Konstruktionsphase einzelne architekturneutrale Softwarelemente wiederverwendet werden. Sollen hingegen, wie bei **BOOGA**, aufbauend auf einer gemeinsamen Plattform ähnliche Anwendungen erstellt werden, so muss eine angepasste Vorgehensmethode definiert werden. Diese sichert eine einheitliche Architektur der Anwendungen und damit einerseits die Möglichkeit zur Wiederverwendung existierender Komponenten, andererseits die Bereitstellung im Rahmen einer Anwendung neu erstellter Komponenten zur späteren Wiederverwendung.

Bevor die an **BOOGA** angepasste Vorgehensmethode beschrieben wird, muss zuerst Klarheit über die speziellen Eigenschaften einer **BOOGA**-Applikation geschaffen werden.

²Die Terminologie entspricht hier Jacobson; die vergleichbare Phase heisst bei Rumbaugh Implementierung und bei Booch Evolution.

5.3 Der Applikationsbegriff von *BOOGA*

Stellt man unterschiedlichen Entwicklerteams dieselbe Aufgabe, so werden die einzelnen Lösungen sehr unterschiedlich ausfallen. Jedes Team wird eine eigene Architektur definieren und unterschiedliche Abstraktionen implementieren. *BOOGA* als Forschungsplattform und Komponentenframework schränkt die Freiheiten der Applikationsentwickler ein, indem eine Architektur sowie eine Anzahl von Komponenten vorgegeben werden. Erst durch diese Einschränkung wird die Grundlage zur Wiederverwendung von Komponenten geschaffen.

Das Ziel bei der Applikationsprogrammierung mit Hilfe von *BOOGA* ist, wie dies Abbildung 5.2 anschaulich illustriert, die Verwendung möglichst



Abbildung 5.2

Eine *BOOGA*-Applikation soll aus so viel bereits existierenden Komponenten zusammengesetzt werden wie möglich.

(Diese Abbildung stammt mit freundlicher Genehmigung des Autors aus [Booch, 1994b].)

vieler Standardkomponenten. Die Applikationsprogrammierung wird so zur *Applikationskomposition*, wie dies auch durch Definition 5.1 ausgedrückt wird:

Definition 5.1

Eine *BOOGA*-Applikation besteht aus einer Menge von Komponenten, welche instanziert, konfiguriert, verknüpft und aktiviert werden. Die Verknüpfung der Komponenten bestimmt zusammen mit deren Funktionalität die Eigenschaften der Applikation.

Bei der Verknüpfung der Komponenten sind syntaktische³ und

³Bei der Verknüpfung von Komponenten sind die Typen der Parameter, bzw. Resultate zu berücksichtigen.

semantische⁴ Restriktionen zu beachten.

Eine Applikation kann zusätzlich Initialisierungs- und Terminierungscode beinhalten.

```

int main(int argc, char* argv[])
{
    // Initialisierungcode:
    RCString name(argv[1]);           // Kommandozeilenparameter behandeln
    World3D* world = new Word3D;      // Daten für implizite Verknüpfung

    BSDLParser3D parse;               // Instanzierung
    parse.setFilename(name + ".bsdl"); // Konfiguration
    parse.execute(world);             // Aktivierung
                                      // Implizite Verknüpfung

    RayshadeWriter writer;            // Instanzierung
    writer.setFilename(name + ".ray"); // Konfiguration
    writer.execute(world);            // Aktivierung

    // Terminierungscode:
    delete world;                   // Daten freigeben
    return 1;                        // Rückgabewert
}

```

Abbildung 5.3

Eine kleine **BOOGA**-Applikation, welche eine Szenenbeschreibung in der **BOOGA** Scene Description Language (BSDL) einliest und im RAYSHADE [Kolb, 1992] Format wieder ausgibt. Der Dateiname der Eingabedatei wird ohne die Endung (.bsdl) als erster und einziger Parameter der Anwendung übergeben. Die Überprüfung der Korrektheit der Kommandozeilenparameter wurde zur Vereinfachung unterlassen.

Die Verknüpfung kann entweder durch direkte Konfiguration einer Komponente mit einer oder mehreren anderen Komponenten explizit geschehen oder durch die Aktivierungsreihenfolge implizit gegeben sein. Instanzierung und Aktivierung erfolgen stets in derselben Art und werden durch Dienste der Verwaltungsschnittstelle der Komponenten abgedeckt. Die Konfiguration ist komponentenspezifisch und somit Teil der Dienstschnittstelle der Komponenten (vgl. Abschnitt 2.5).

Abbildung 5.3 zeigt eine einfache Applikation, in der zwei Komponenten instanziert und implizit, d.h. durch sequentielle Aktivierung miteinander verknüpft werden. Aufwendigere Anwendungen beinhalten viel mehr Komponenten, zudem können mittels Schleifen und Bedingungen die Verknüpfungen zwischen den Komponenten komplexer gestaltet werden.

Natürlich können nicht schon alle denkbaren Komponenten in der Komponentenbibliothek enthalten sein. Neue Anwendungen bedingen in der

⁴So kann einePixmap beispielsweise erst dargestellt werden, *nachdem* sie eingelesen wurde.

Regel auch neue Komponenten. Es gibt also bei der Applikationserstellung zwei unterschiedliche Aufgaben: die *Komponentenentwicklung* und die *Applikationskomposition*.

Neue Komponenten werden konform zu den vorgegebenen Komponentenprotokollen und aufbauend auf der Bibliotheks- und Frameworkschicht realisiert. Die Frameworkschicht kann bei dieser Aufgabe als Black-Box betrachtet werden. Änderungen oder Ergänzungen dieser Schicht sind nur in den seltensten Fällen nötig.

Komponentenentwicklung

Die Entwicklung neuer Komponenten erfordert wesentlich mehr Kenntnisse von **BOOGA** als die blosse Komposition bestehender Komponenten. Die Protokolle der Verwaltungsschnittstelle müssen korrekt implementiert werden, um ein reibungsloses Zusammenspiel mit den anderen Komponenten zu gewährleisten. Kenntnisse der Framework- und Bibliotheksschicht sind hierfür nötig und die Anforderungen an die C++- und Objekttechnologiekenntnisse sind wesentlich höher.

Eine zusätzliche Schwierigkeit bei der Komponentenentwicklung liegt in der Abgrenzung der Aufgaben einer Komponente. Sie sollte stets eine *atomare Teilaufgabe* (vgl. Abschnitt 5.4.3) erfüllen und mittels der Dienstschnittstelle konfigurierbar und damit in einem bestimmten Rahmen flexibel einsetzbar sein. Die Erfüllung dieser Forderungen erhöht die Wahrscheinlichkeit, dass die neue Komponente in zukünftigen Anwendungen wiederverwendet werden kann.

Bei der Realisierung einer neuen Komponente kann diese oft auch aus bereits bestehenden zusammengesetzt werden (Abbildung 5.4). Dies kann

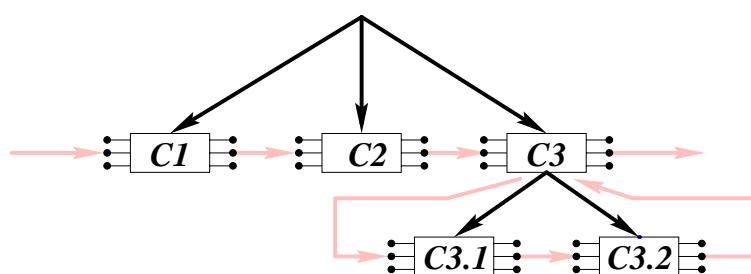


Abbildung 5.4

Eine **BOOGA**-Applikation setzt sich aus einer Anzahl von Komponenten zusammen, die selber wieder aus weiteren Komponenten zusammengesetzt sein können. Die grauen Pfeile symbolisieren den Kontrollfluss, die schwarzen Pfeile zeigen den strukturellen Aufbau der Komponentenhierarchie.

grundsätzlich auf zwei Arten erfolgen, die im folgenden anhand konkreter Beispiele beschrieben werden:

1. Verwendung einer konkreten Komponente

Bei der Komposition wird eine konkrete Komponente direkt instanziert und bei der Ausführung entsprechend aktiviert. Zur Laufzeit ist an dieser Stelle keinerlei Konfiguration möglich.

Ein Beispiel soll dies verdeutlichen:

In der Komponentenbibliothek von **BOOGA** existieren verschiedene Renderingkomponenten, welche ein bestimmtes Verfahren zur Berechnung von Bildern aufgrund von Szenenbeschreibungen ermöglichen. Da alle Renderingverfahren gewisse gleiche Aufgaben lösen müssen, können diese in einer gemeinsamen Basisklasse **Renderer** zusammengefasst werden. Eine der Aufgaben dieser Basisklasse ist das “Sammeln” aller in der Szene vorhandenen Lichtquellen sowie Kameras. Diese werden dann von den spezifischen Renderingkomponenten zur Bildberechnung verwendet. Dieses Sammeln von bestimmten grafischen Objekten in der Szenenbeschreibung wird aber auch noch von anderen Komponenten benötigt. Es liegt daher nahe, eine spezielle Komponente zu realisieren, die nach bestimmten Objekten suchen kann. Alle Rendererkomponenten verwenden die Komponente **CollectorFor**, um die besprochene Aufgaben zu erfüllen. Diese Komponente ist fest in der Implementierung der Rendererkomponente enthalten.

2. Verwendung einer Basisabstraktion

Bei dieser Variante wird mittels einer polymorphen Variablen lediglich eine Referenz auf eine abstrakte Basiskomponente gespeichert. Die konkrete Komponente wird erst zur Laufzeit instanziert und der Referenz zugewiesen.

Auch hier soll wieder ein konkretes Beispiel aus **BOOGA** als Illustration dienen:

Bei der Bildberechnung treten aufgrund von Rechenun genauigkeiten und Diskretisierungs- sowie Quantisierungsfehlern gewisse Bildstörungen auf, die durch *Antialiasingverfahren* ausgeglichen werden können. Diese Verfahren beruhen auf der Idee, durch Erhöhung der Anzahl Abtastpunkte und anschliessende Mittelwertbildung diese Fehler zu verringern. In der Komponentenbibliothek von **BOOGA** existiert eine Antialiasingkomponente **OversamplingAntialiaser**, welche durch eine Vervielfachung der Grösse des gesamten zu berechnenden Bildes, sowie anschliessende Reduktion der Bildgrösse durch Mittelwertbildung ein sehr einfaches und universell für alle Renderingarten anwendbares Verfahren realisiert. Die Antialiasingkomponente wird mittels der spezifischen Dienstschnittstelle mit dem zu verwendenden Renderer konfiguriert. Im Gegensatz zu der vorher aufgezählten Variante muss hier bei der Applikationskomposition also eine explizite Verknüpfung zwischen der zu verwendenden Rendererkomponente und der Antialiasingkomponente hergestellt werden.

Dieser hierarchische Aufbau erlaubt eine komponentenbasierte Wieder-verwendung auf unterschiedlichen Abstraktionsniveaus und erleichtert so-mit die einzelnen Schritte des in Abschnitt 5.4 beschriebenen Vorgehens-modells.

Die Komposition von Anwendungen aus Komponenten erfordert im Ge-gensatz zur Komponentenentwicklung kaum Kenntnisse über die tiefer-liegenden Schichten des Komponentenframeworks (vgl. Abschnitt 4.3). Grundlegende Kenntnisse der Komponentenschicht, deren Protokolle so-wie eine gute Übersicht über die bereits existierenden Komponenten sind die Anforderungen an einen Applikationsentwickler.

Seine Aufgaben könnten durch geeignete Hilfsmittel weiter vereinfacht werden. Ein elektronischer Komponentenkatalog, der auf einfache Weise erlaubt, rasch auf eine bestimmte Komponente zuzugreifen⁵ ist ein Bei-spiel hierfür. Denkbar ist auch, die Anwendung anstelle durch Program-mierung mittels *visueller Komposition* zusammenzubauen. Eine mögliche, hierfür nötige grafische Notation wird in Abschnitt 6 verwendet und in Anhang A.1 eingeführt.

Applikationskom-position

⁵Abschnitt 7.3 skizziert einen solchen, speziell auf *BOOGA* zugeschnittenen Komponentenkatalog.

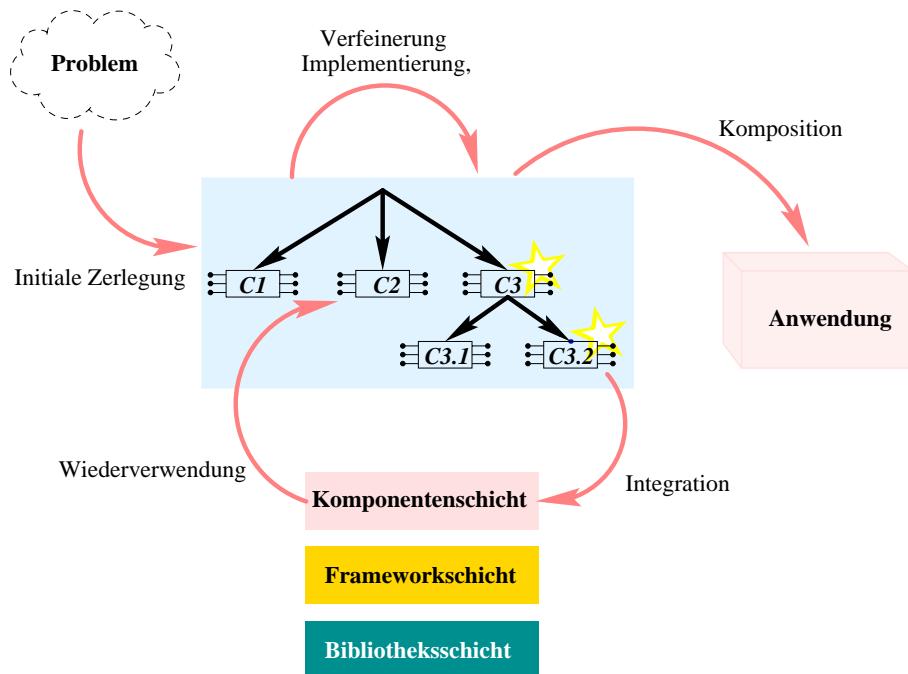
5.4 Vorgehensmodell von **BOOGA**

Dieser Abschnitt beschreibt ein Vorgehensmodell, das bei der Applikationsentwicklung mit **BOOGA** angewendet werden kann. Dieses Modell basiert auf den bereits erwähnten objektorientierten Methoden. Speziell der Gedanke der iterativen, inkrementellen Entwicklung wurde aufgenommen und verstärkt zum Ausdruck gebracht. Im folgenden wird das Modell zuerst in einer Übersicht erläutert. Anschliessend wird im Detail auf die einzelnen Teilschritte des Entwicklungszyklus eingegangen. Wo direkte Anleihen aus den erwähnten Methoden vorgenommen wurden, wird dies im Text explizit erwähnt.

Aufgrund der bereits im vorhergehenden Abschnitt besprochenen Anforderungen, die sich aus den Besonderheiten des in **BOOGA** verwendeten Komponenten- und Applikationsbegriffs ergeben, lässt sich ein grober Ablauf wie in Abbildung 5.5 skizziert festhalten. Während diese Abbildung

Abbildung 5.5

Der Weg von einer Problemstellung zur fertigen Anwendung führt – unter Einbezug des Komponentenframeworks – über mehrere Iterationen, in denen die initiale Zerlegung verfeinert und Komponenten implementiert und getestet werden. Die mittels Sternen gekennzeichneten Komponenten C3 und C3.2 sind neue Komponenten, die für die vorliegende Problemstellung realisiert und anschliessend in die Komponentenbibliothek integriert wurden.



lediglich eine Übersicht darstellt, wird in Abbildung 5.6 das Vorgehen als systematischer Ablauf dargestellt.

Eine *initiale Zerlegung* der Aufgabe führt zu einer ersten Aufteilung in Komponenten. Diese Aufteilung könnte aus unterschiedlichen Gründen noch nicht optimal sein:

- Die Komponentenbibliothek bietet Komponenten mit einer Funktionalität, ähnlich der gewünschten an, aber keine, welche die An-

forderungen vollständig erfüllen. In diesem Fall kann eine andere Zerlegung zu einem höheren Wiederverwendungsgrad für die vorliegende Problemstellung führen.

- Die erzielte Zerlegung resultiert in sehr problemspezifischen Komponenten, die wenig Wiederverwendungsmöglichkeiten in späteren Projekten bieten. Eine modifizierte Zerlegung oder etwas allgemeiner formulierte Komponenten führen zu künftig besser wiederverwendbaren Komponenten.
- Erfahrungen mit dem traditionellen Wasserfallmodell zeigen, dass es sehr schwierig ist, von Anfang an alle Anforderungen an ein System vollständig zu erfassen. Mit zunehmendem Wissen, das durch die Realisierung des Systems, beziehungsweise der einzelnen Komponenten gewonnen wird, werden sich die Systemanforderungen ändern, wodurch auch die initiale Zerlegung in Komponenten Änderungen unterworfen werden kann.

In der *Verfeinerungs- und Implementierungsphase* wird einerseits die bestehende Zerlegung angepasst, andererseits bereits als definitiv akzeptierte Komponenten realisiert. Diese Phase ist stark iterativ. Die einzelnen Komponenten werden schrittweise bis zu ihrer gewünschten Funktionalität erweitert und angepasst.

Bereits bestehende Komponenten werden aus der Komponentenbibliothek entnommen (*Wiederverwendung*). Neu erstellte Komponenten werden in die Komponentenbibliothek *integriert*.

Die gewünschte Anwendung wird schliesslich aus den einzelnen Komponenten in der bereits besprochenen Weise zusammengesetzt (*komponiert*).

5.4.1 Übersicht

Abbildung 5.6 zeigt den gesamten Entwicklungsprozess für **BOOGA**-Anwendungen im Überblick. Wesentlich an diesem Vorgehensmodell ist die starke Betonung des iterativen Gedankens, dessen Wichtigkeit für die Entwicklung flexibler, objektorientierter Software bereits in vielen Publikationen erwähnt wurde, so beispielsweise in [Booch, 1991; Booch, 1994b; Gossain und Anderson, 1990; Jacobson et al., 1992; Rumbaugh et al., 1991]. Die hier vorgestellte Vorgehensweise stellt die Iteration noch stärker ins Zentrum, als dies in den bekannten Methoden bereits der Fall ist.

Gegenüber Abbildung 5.6 wurde hier die Frameworkpflege als eigenständiger Prozess vereinfachend weggelassen. Trotzdem ist diese Aufgabe aber ein wesentlicher Bestandteil der gesamten Methode und wird in Abschnitt 5.4.6 genauer diskutiert.

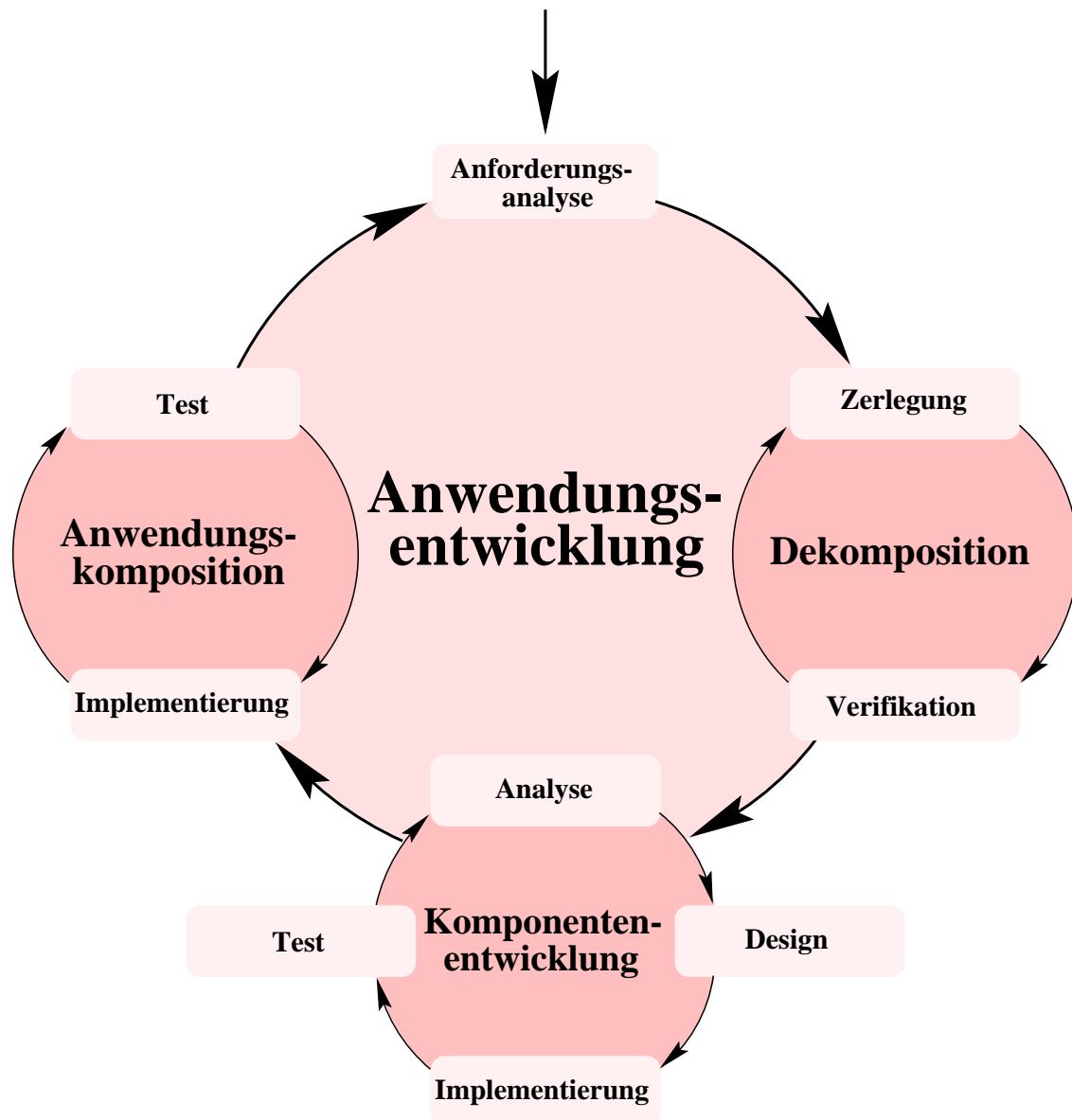


Abbildung 5.6

Der hochgradig iterative Softwareentwicklungsprozess für **BOOGA**-Anwendungen.

Das Ziel besteht darin, möglichst schnell eine lauffähige Applikation zusammenzubauen. Die dazu verwendeten Komponenten können im Verlaufe der Iteration durch passendere ersetzt oder neue Komponenten schrittweise bis zur gewünschten Funktionalität erweitert werden.

Der gesamte Prozess besteht aus vier *Phasen*, nämlich der *Anforderungsanalyse*, der *Dekomposition*, der *Komponentenentwicklung* sowie der *Anwendungskomposition*. Die einzelnen Phasen sind in der Regel in *Teilschritte* untergliedert. Jede Phase stellt, ebenso wie der gesamte Ablauf, für sich selber einen iterativen Prozess dar. In den nachfolgenden Abschnitten werden die einzelnen Phasen und Teilschritte des Entwicklungszyklus im Detail erklärt.

Auf den ersten Blick erstaunlich scheint, dass der Prozess zwar einen Eingang, jedoch keinen Ausgang kennt. Normalerweise schliesst sich an die Entwicklung eines Systems eine Wartungsphase an, die zwar in den

Entwicklungsmethoden erwähnt, in der Regel aber nicht weiter erläutert wird. Diese Phase wird oft nicht mehr von den Entwicklern der Software durchgeführt, sondern von einer speziell auf die Wartung von Software ausgerichteten Organisationseinheit. Betrachtet man aber die Aufgaben, die während der Wartung anfallen (Fehlerbehebung, Erweiterungen), so wird man feststellen, dass auch die Wartungsphase durch den im folgenden beschriebenen Prozess optimal abgedeckt wird. Die Entwicklungs- und die Wartungsphase unterscheiden sich in diesem Modell nur durch die *Iterationsfrequenz*; mit zunehmender Reife werden die Iterationen immer seltener (vgl. Abbildung 5.7).

Es ist wichtig anzumerken, dass nicht bei jeder Iteration durch den Prozess jeder einzelne Teilschritt durchgeführt werden muss. So kann beispielsweise bei der Lösung eines Problems, für das bei einer optimalen Zerlegung bereits alle Komponenten in der Komponentenbibliothek vorhanden sind, die Phase der Komponentenentwicklung übersprungen werden.

Das Vorgehensmodell von **BOOGA** stellt zusammenfassend eine Ausprägung eines Spiralmodells dar, wie dies in Abbildung 5.7 veranschau-

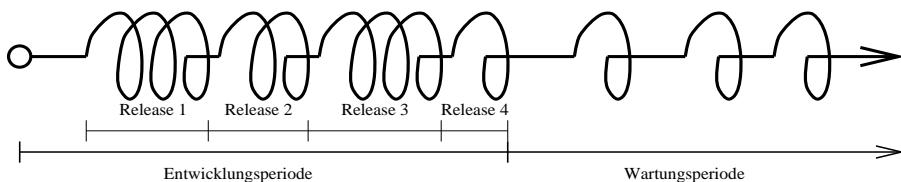


Abbildung 5.7

Der Lebenszyklus einer Applikation lässt sich grob in zwei Perioden unterteilen: die Entwicklungsperiode und die Wartungsperiode.

licht wird. Der Lebenszyklus einer Applikation wird dabei in zwei Perioden unterteilt: die Entwicklungsperiode, in welcher der gesamte Prozess sehr intensiv eingesetzt und oft durchlaufen wird und die Wartungsperiode, wo der Prozess nur bei Bedarf Anwendung findet. Während die Entwicklungsperiode klare Anfangs- und Endpunkte hat, beginnt die Wartungsperiode mit dem Ende der Entwicklung und dauert an, solange die Anwendung sich im Einsatz befindet. Bereits aus diesem Grund drängt sich eine unterschiedliche Organisationsform zur Bewältigung der Aufgaben der beiden Perioden auf. Üblicherweise findet eine Form der *Projektorganisation* in der Entwicklung Anwendung, während die Wartung, in der Regel von mehreren Anwendungen gleichzeitig, von einer speziellen Abteilung übernommen wird.

Die Bedeutung der *Releases* und der damit verbundenen *Releaseplanung* wird im folgenden Abschnitt behandelt.

Anforderungs-analyse

5.4.2 Anforderungsanalyse

Der erste Schritt jeder Problemlösung muss eine Analyse der vorliegenden Problemstellung sein. In manchen Fällen ist die Aufgabe bereits klar genug formuliert und sauber strukturiert, so dass dieser erste Teilschritt bereits als gegeben betrachtet werden kann. In den meisten Fällen wird aber eine mehr oder weniger vage Idee am Anfang eines Projektes stehen. Diese muss konkretisiert und ausformuliert werden, so dass der Umfang der Problemstellung ersichtlich wird.

In den folgenden Schritten sollte berücksichtigt werden, dass die Resultate der Anforderungsanalyse nicht ‘in Stein gehauen’ sind, sondern sich im Verlaufe des Projektes noch ändern können. Der Ansatz des Komponentenframeworks (vgl. Abschnitt 4.3) kommt dieser Forderung sehr entgegen, da die Grundlagen der hierzu nötigen Flexibilität bereitgestellt werden.

Ziel

Das Ziel der Anforderungsanalyse ist ein Katalog von Anforderungen (engl. *Requirements*), welche die zu entwickelnde Anwendung erfüllen muss. Die Grenzen des Modells müssen definiert und die vom System anzubietende Funktionalität festgelegt werden. Manche dieser Anforderungen müssen eventuell durch einen Prototypen verifiziert werden (vgl. *Konzeptualisierungsphase* in [Booch, 1994b]).

Vorgehen

Da die Anforderungsanalyse der erste Schritt ist, der ausgehend von einer manchmal vagen Idee zu einer konkreten Beschreibung eines Produktes führen soll, ist es sehr schwierig, hierfür ein formales Vorgehen zu empfehlen.

Konzeptualisierung ist von Natur aus eine sehr kreative Aktivität, deshalb sollte sie nicht durch strenge Entwicklungsregeln eingeengt werden.

[Booch, 1994b]

Trotzdem können Hilfsmittel, wie die in [Jacobson et al., 1992] vorgestellte *Use-Case-Analysis*, dazu dienen, die Anforderungen an ein System systematisch zu erfassen und zu dokumentieren. Sie bietet auch eine ausgezeichnete Möglichkeit, die Anforderungen zusammen mit Fachleuten aus anderen Bereichen oder Informatikern, welche nicht mit **BOOGA** vertraut sind, zu diskutieren.

Resultat

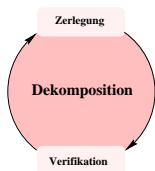
Die Resultate dieses ersten Schrittes sind abhängig vom gewählten Vorgehen. Hat sich der Entwickler entschieden, die Use-Cases von Jacobson als Hilfsmittel zu verwenden, so wird ein Use-Case Diagramm zusammen mit einer genauen Beschreibung der einzelnen Use-Cases diesen ersten Schritt abschliessen. Andernfalls sollte mindestens eine einfache Liste von Anforderungen vorliegen.

Bei komplexeren Aufgaben ist es darüber hinaus sinnvoll, einen *Releaseplan*⁶ zu erstellen, der den Funktionsumfang der iterativ zu erweiternden Anwendung beschreibt. Aus dem Releaseplan der Gesamtanwendung lassen sich die Releasepläne für die einzelnen Komponenten ableiten. Ziel dieses Vorgehens ist, möglichst früh eine lauffähige Version mit eingeschränkter Funktionalität der zu erstellenden Applikation vorliegen zu haben und diese dann schrittweise bis zur fertigen Anwendung weiter zu entwickeln.

5.4.3 Dekomposition

Während der Dekompositionsphase wird das Gesamtproblem in einzelne Teilaufgaben zerlegt, welche dann von Komponenten – bereits vorhandenen oder neu zu entwickelnden – gelöst werden. Die Zerlegung muss dabei so erfolgen, dass die Komponenten einer der vier in Abschnitt 4.2 vorgestellten **BOOGA**-Komponentenkategorien zugeordnet werden können.

Es ist wichtig zu beachten, dass einzelne Anforderungen unter Umständen nicht durch reine Komponentenentwicklungen erfüllt werden können, sondern eine Erweiterung der Frameworkschicht nötig sein kann. Dieses Vorgehen wird in Abschnitt 5.4.6 besprochen und anhand einiger Beispiele in Kapitel 6 illustriert.



Ziel

Beim ersten Durchlauf dieser Phase geht es darum, eine initiale Zerlegung zu finden. In späteren Durchläufen soll die Dekomposition wenn nötig verbessert werden. Die zur Beurteilung der Güte nötigen Kriterien sind dabei:

⁶Der Begriff *Version* wird in der Regel verwendet, um ein aus der Sicht des jeweiligen Anwenders homogenes Gebilde (z. Bsp. ein *Programm* für einen Benutzer, *Datei* für den Programmierer einer Anwendung) zu bezeichnen. Demgegenüber steht der Begriff *Release* (auch *Konfiguration*), der für heterogene Gebilde verwendet wird. So ist eine Anwendung aus der Sicht des Programmierers ein heterogenes Gebilde, welches aus verschiedenen Quelltextdateien zusammengesetzt wird.

Der Release einer Anwendung besteht also aus Entwicklersicht aus einer Anzahl Quelltextdateien in unterschiedlichen Versionen.

- die Erreichung der Anforderungen gemäss der vorausgegangenen Anforderungsanalyse,
- der Grad der Wiederverwendung existierender Komponenten,
- die Anzahl der neu zu erstellenden Komponenten und
- die Allgemeinheit (und somit Wahrscheinlichkeit der späteren Wiederverwendung) der neuen Komponenten.

Vorgehen

Die beiden Teilschritte *Zerlegung* und *Verifikation* werden solange durchlaufen, bis eine ausreichend⁷ korrekte Zerlegung gefunden worden ist.

Resultat

Das Resultat dieser Phase sollte eine ausreichend korrekte Zerlegung in Teilaufgaben sein. Diese Zerlegung wird im folgenden als Basis dazu dienen, erste Versionen der neuen Komponenten sowie der Anwendung zu erstellen. Sollte sich zu einem späteren Zeitpunkt herausstellen, dass die als Resultat dieser Phase in einer bestimmten Iteration erzielte Zerlegung nicht passend ist, so können in einer weiteren Iteration die nötigen Anpassungen vorgenommen werden.

Manche Komponenten der Komponentenbibliothek vermögen unterschiedliche Aufgaben zu erfüllen und bedürfen hierzu einer Konfiguration, d.h. einer Anpassung an die entsprechende Aufgabe. Diese Konfiguration kann unterschiedlich aufwendig sein. Oft müssen nur einzelne Parameter korrekt gesetzt werden. In anderen Fällen, in denen Komponenten durch die Verwendung des *Strategy Patterns* [Gamma et al., 1995] sehr flexibel an neue Aufgaben angepasst werden können, ist die Konfiguration eine eigene kleine Entwicklungsaufgabe. Einfache Parametrisierungen können durchaus bereits in der Dekompositionsphase definiert werden, die aufwendigeren Entwicklungen können als Spezialfall einer Modifikation einer bestehenden Komponente betrachtet und in der nächsten Phase vorgenommen werden.

5.4.3.1 Zerlegung

Dieser Teilschritt ist das zentrale Element des gesamten Entwicklungsprozesses und kann nur sehr schwer durch konkrete Empfehlungen oder

⁷Sowohl der gesamte Prozess wie auch jede einzelne Phase werden iterativ durchgeführt. In frühen Iterationsschritten können durchaus noch Änderungen der im vorausgegangenen Schritt vorgenommenen Zerlegung erfolgen. Mit zunehmender Reife des Systems wird sich die Zerlegung stabilisieren.

Vorgehensweisen gestützt werden. Massgebend für die erreichte Güte einer Zerlegung ist hier die *Erfahrung* mit **BOOGA**. Ein wenig erfahrener Anwender, der die **BOOGA** zugrundeliegende Philosophie noch nicht verinnerlicht und keinen Überblick über die bereits vorhandenen Komponenten hat, wird viele Iterationen benötigen, um zu einer guten Lösung zu kommen, währenddessen ein erfahrener **BOOGA**-Entwickler oft bereits auf Anhieb eine optimale Lösung findet.

Ziel

Ziel dieses Teilschrittes ist, eine mögliche Zerlegung des Gesamtproblems in Teilaufgaben zu finden.

Vorgehen

Wie bereits erwähnt kann dieser Schritt nicht durch ein ‘*Kochrezept*’ beschrieben werden. Ein erfahrener Entwickler weiss aufgrund seiner Erfahrung, wie ein Problem idealerweise zu zerlegen ist. Trotzdem können einige Hinweise gegeben werden, die dabei helfen, gute oder schlechte Komponenten als solche zu erkennen. Bei der Besprechung des nächsten Teilschrittes (*Verifikation*) werden diese Kriterien im Detail besprochen.

Resultat

Das Resultat dieses Teilschrittes ist eine Liste von Komponentenkandidaten, die im nachfolgenden Teilschritt aufgrund unterschiedlicher Kriterien verifiziert werden. Jede einzelne Komponente sollte dabei gemäss der in Abschnitt 4.2 vorgestellten Klassierung in eine der vier Kategorien eingeteilt werden können und eine *atomare* Teilaufgabe erfüllen.

5.4.3.2 Verifikation

Dieser Teilschritt ist vor allem für Entwickler wichtig, welche noch wenig Erfahrung mit der Anwendungsentwicklung mit **BOOGA** aufweisen. Mit zunehmender Erfahrung wird die Verifikation als *intuitiver Filter* im vorausgegangenen Teilschritt zur Wirkung kommen und somit als selbständige Aktion mehr und mehr an Bedeutung verlieren.

Ziel

Die Komponentenkandidaten des vorausgegangenen Teilschrittes erfüllen in der Regel noch nicht alle Anforderungen an ‘gute’ Komponenten. Dieser Teilschritt hilft bei einer Beurteilung der Kandidaten und ermöglicht somit, für die Gesamtlösung ungünstige Komponenten auszuscheiden.

Vorgehen

Jeder einzelne Kandidat sollte gemäss dem folgenden Kriterienkatalog kritisch beurteilt werden:

- **Existiert bereits eine passende Komponente?**

Sollte ein Anforderungskatalog für einen Komponentenkandidaten bereits auf eine existierende Komponente in der Komponentenbibliothek passen, so ist dies ein gutes Indiz dafür, dass die Zerlegung bezüglich dieses Kandidaten gut gelungen ist.

Existiert in der Komponentenbibliothek eine ähnliche Komponente, die beinahe der Funktionalität des Kandidaten entspricht, so muss zwischen zwei Varianten abgewägt werden: (a) die Zerlegung wird so modifiziert, dass die Komponente aus der Bibliothek entnommen werden kann oder (b), es wird eine neue Komponente realisiert, die eventuell auf der bereits existierenden Komponente aufbaut.

Variante (a) kann Einfluss auf die gesamte Problemzerlegung und somit auf die anderen Komponentenkandidaten haben. Sollte dies zu einer insgesamt schlechteren Zerlegung führen, so ist Variante (b) der Vorzug zu geben.

Bei Variante (b) gibt es verschiedene Implementierungsmöglichkeiten, die in der Besprechung der Komponentenentwicklungsphase genauer diskutiert werden.

Um die Frage nach der Existenz einer bestimmten Komponente beantworten zu können, ist entweder eine gute Kenntnis des Systems nötig, oder es muss auf ein Dokumentationssystem (vgl. Kapitel 7) zurückgegriffen werden.

- **Erfüllen die Komponenten atomare Teilaufgaben?**

Eine Komponente erfüllt eine atomare Teilaufgabe, falls sie nicht weiter in sinnvolle Komponenten zerlegt werden kann, oder selber eine Komposition aus atomaren Komponenten darstellt.

Bei atomaren Komponenten ist die Wahrscheinlichkeit wesentlich grösser, dass eine passende Komponente bereits in der Komponentenbibliothek existiert oder, falls sie neu erstellt werden muss, zu einem späteren Zeitpunkt wiederverwendet werden kann.

- **Ist die Problemkomplexität der Komponente gering?**

Diese Fragestellung ist eng mit dem vorhergehenden Kriterium verknüpft. Eine Komponente sollte eine nicht zu grosse Komplexität aufweisen. Dies erleichtert die Implementierung, das Testen sowie das Verständnis der Komponente, wenn sich

zu einem späteren Zeitpunkt die Frage stellt, ob sie wiederverwendet werden soll.

- **Sind die Komponenten allgemein verwendbar?**

Ist eine Komponente sehr stark auf eine bestimmte Aufgabe spezialisiert, so ist die Chance einer Wiederverwendung äußerst klein. Wenn möglich, sollte eine Komponente von der spezifischen Problemstellung abstrahiert realisiert werden. Dies kann beispielsweise erreicht werden, indem die Komponente mit speziellen Komponenten konfiguriert werden kann, wie dies bereits in Abschnitt 5.3 anhand von Beispielen erläutert wurde.

An dieser Stelle soll allerdings eine Warnung ausgesprochen werden: Es ist sehr wünschenswert, dass eine Komponente allgemein eingesetzt werden kann, diese Verallgemeinerung soll jedoch im Rahmen der iterativen Entwicklung geschehen, d.h. eine Komponente darf zu Beginn durchaus sehr spezifisch sein und erst mit der Zeit und bei Bedarf verallgemeinert werden. Es wurde in den vorausgegangenen Kapiteln bereits ausführlich diskutiert, dass Flexibilität stets mit einer erhöhten Komplexität einhergeht.

Zusätzlich zu einer isolierten Betrachtung der einzelnen Komponenten bedarf aber auch das Gesamtsystem einer kritische Beurachtung. So muss sichergestellt werden, dass alle Anforderungen der vorausgegangenen Phase erfüllt werden können. Dieser bei der herkömmlichen Softwareentwicklung offensichtliche Schritt darf selbstverständlich auch bei dem hier vorgestellten Ansatz, ob der Betrachtung der Güte einzelner Komponenten, nicht in Vergessenheit geraten.

Oftmals sind mehrere Zerlegungen möglich. Sollte eine dieser Zerlegungen zu einer insgesamt besseren Liste von Komponentenkandidaten führen, so ist dieser Variante der Vorzug zu geben.

Resultat

Das Resultat dieses Teilschrittes ist eine kritische Bewertung der im vorausgegangenen Teilschritt gefundenen Komponenten, beispielsweise in Form einer Bewertungsmatrix. Diese Matrix dient als Feedback für den nächsten Iterationszyklus innerhalb der Dekompositionsphase. Werden mehrere Zerlegungen einander gegenübergestellt, so kann eine Bewertungsmatrix als Kriterium für die Güte einer Zerlegung verwendet werden.



5.4.4 Komponentenentwicklung

Die Phase der Komponentenentwicklung sollte mit zunehmender Reife und Umfang der Komponentenbibliothek idealerweise immer seltener durchlaufen werden müssen. Komponenten die im Teilschritt der *Verifikation* der vorausgegangenen *Dekompositionsphase* als bereits vorhanden klassiert wurden, können direkt der Komponentenbibliothek entnommen werden und in die nächste Phase, der *Anwendungskomposition*, einfließen.

Die Phase der Komponentenentwicklung deckt sich weitgehend mit der in [Booch, 1994b] vorgestellten Methode zur Anwendungsentwicklung. Die Teilschritte werden aber gegenüber Booch soweit abgekürzt, als bereits die äussere Struktur der Komponente sowie das Zusammenspiel mit den restlichen Elementen gegeben ist.

Ziel

Fehlende Komponenten müssen entsprechend den vorgegebenen Schnittstellen und aufbauend auf der Framework- und Bibliotheks- schicht schrittweise realisiert werden. Ein iteratives Vorgehen ergibt sich fast zwangsläufig aus der Forderung nach einem raschen Übergang zur nächsten Phase, so dass sehr früh in der gesamten Entwicklung bereits eine lauffähige Anwendung realisiert werden kann.

Vorgehen

Die einzelnen Komponenten werden in den vier Teilschritten *Analyse*, *Design*, *Implementierung* und *Test* realisiert. Dabei ist besonders Wert darauf zu legen, in ersten Durchläufen der Komponentenentwicklungsphase möglichst rasch die Kernfunktionalität zu verwirklichen und diese dann schrittweise zu erweitern.

Resultat

Das Resultat dieser Phase sind schrittweise vollständigere Komponenten, welche einerseits in der folgenden Phase der *Anwendungskomposition* benötigt werden, andererseits in der Komponentenbibliothek abgelegt und so weiteren Entwicklern für folgende Projekte zur Verfügung gestellt werden können.

5.4.4.1 Analyse

Je nach Komplexität der Aufgabe, die eine Komponente zu erfüllen hat, wird dieser Schritt sehr kurz ausfallen oder direkt mit dem nächsten Teilschritt, dem *Design* zusammen behandelt werden. Da Komponenten

ja idealerweise stets atomare Teilaufgaben erfüllen und somit in der Regel eher wenig Komplexität aufweisen, wird dies fast immer der Fall sein. In Abschnitt 6.3 wird aber ein Beispiel besprochen, welches eine recht komplexe Komponente benötigt wodurch eine eigenständige Analysephase nötig und sinnvoll wird.

Die Analyse einer einzelnen Komponente sollte nicht unabhängig von der Umgebung geschehen. So muss hier selbstverständlich die spezielle Form einer **BOOGA**-Komponente berücksichtigt, sowie alle notwendigen Schnittstellen vorgesehen werden. Daneben ist aber hier bereits darauf zu achten, die Umsetzung der Komponentenanforderungen in ein Modell **BOOGA**-konform zu gestalten. Dazu gehört insbesondere auch die Berücksichtigung bereits existierender Komponenten, Frameworkstrukturen und Klassen. Weiter ist es auch wichtig, die allgemeinen Mechanismen und Konventionen wie Errorhandling, Konfigurationsmechanismen oder Namensgebung von **BOOGA** zu berücksichtigen. Diese Anmerkung treffen natürlich auch auf die folgenden Teilschritte zu.

Ziel

[...] der Zweck einer Analyse [ist], eine Beschreibung eines Problems zu erstellen.

Die Beschreibung muss vollständig, konsistent, lesbar und überprüfbar für verschiedene interessierte Parteien sein, ausserdem testbar für die Realität.”

Mellor in [Booch, 1994b]

Gemäß diesem Grundsatz sollte im Analyseteilschritt eine Beschreibung der Aufgabenstellung an die Komponente entstehen. Im Sinne des iterativen Grundsatzes wird aber die Forderung nach *Vollständigkeit* insofern abgeschwächt, als die Beschreibung nur *möglichst vollständig* sein muss. Aufgrund von sich ändernden Anforderungen, neuen Erkenntnissen während der Design- und Realisierungsphase und oft unvollständigen Informationen kann die Analyse einer Problemstellung eigentlich nie *vollständig* abgeschlossen werden.

Die unter der Komponentenschicht von **BOOGA** liegenden Framework- und Bibliotheksschichten können selbstverständlich nicht alle denkbaren Anforderungen erfüllen. Ein weiteres Ziel des Analyseteilschrittes ist somit auch das Aufdecken allfälliger Lücken.

Vorgehen

Der Analyseteilschritt der Komponentenentwicklungsphase weist

gewisse Ähnlichkeiten mit der Anforderungsanalysephase für die gesamte Anwendung auf. An dieser Stelle sollten die Anforderungen an eine einzelne Komponente konkretisiert und in ein erstes Modell umformuliert werden. Auch hier können, je nach Umfang und Komplexität der Komponente, Hilfsmittel wie die bereits erwähnten *Use-Cases* eingesetzt werden.

Aufwendigere Komponenten, die oft eine explizite eigene Analysephase bedingen, können zu einer Erweiterung der Framework- oder Bibliotheksschicht führen. Diese Erweiterung wird in der Regel von einem erfahrenen Komponentenentwickler selber vorgenommen und spezifisch für die Komponente entwickelt, welche diese Erweiterung benötigt. Wiederum im Sinne eines iterativen Vorgehens und einer grösstmöglichen Wiederverwendung kann diese Erweiterung zu einem späteren Zeitpunkt von einem Frameworkentwickler in das Komponentenframework **BOOGA** integriert und so zukünftigen Anwendungen zugänglich gemacht werden (vgl. Abschnitt 5.4.6).

Wie [Booch, 1994b] ausführt, ist die *Szenario-Planung* die zentrale Aktivität der Analysephase. Szenarios sind Beschreibungen des Verhaltens eines Systems, in unserem Fall einer Komponente, bezogen auf einen speziellen Funktionspunkt. Ein Funktionspunkt ist die Aktion eines Systems als Reaktion auf ein Ereignis. Booch unterscheidet zwischen primären Szenarios, welche das Schlüsselverhalten beschreiben, und sekundären Szenarios, die das Verhalten unter besonderen Bedingungen zeigen.

Die für eine Komponente wichtigen Szenarios können mit Hilfe unterschiedlicher Hilfsmittel wie beispielsweise CRC-Karten⁸ oder, in einem weiteren Schritt, mit Objekt- oder Klassendiagrammen entwickelt und dokumentiert werden.

Unter professionellen Entwicklern von Software anerkannt und doch nicht immer praktiziert, sind der Einbezug von weiteren Entwicklern in die Problemlösung sowie das Vorstellen und kritische Besprechen (beispielsweise mit der Technik des *Walkthrough*) eines Lösungskonzeptes. Durch ein solches Vorgehen werden Erfahrungen anderer Entwickler mit den eigenen Ideen synergetisch verknüpft. In der Regel zahlt sich der investierte zusätzliche Zeitaufwand durch eine flexiblere und manchmal einfachere Lösung aus.

Resultat

⁸Das Akronym CRC steht für *Class – Responsibility – Collaboration*. CRC Karten wurden von [Beck und Cunningham, 1989] ursprünglich für die Schulung des objektorientierten Gedankengutes verwendet. Es eignet sich aber auch ausgezeichnet als Hilfsmittel während einer Kreativitätsphase am Anfang eines Problemlösungsprozesses.

Das Resultat dieses Teilschrittes – sofern er explizit durchgeführt wird – sind eine Anzahl von Szenarios sowie Diagrammen, welche die Aufgaben der neu zu entwickelnden Komponenten in einer klaren, umsetzbaren Form umreissen.

Je nach Grösse und Umfang der Komponente kann ein weiteres Resultat des Analyseteilschrittes auch ein *Releaseplan* sein. Dieser beschreibt die Folge der iterativ zu erstellenden und erweiternden Versionen der Komponente sowie der allfällig zugehörigen weiteren Klassen. Der Releaseplan einer einzelnen Komponente muss sich aus dem Releaseplan der ganzen Anwendung ableiten, der in der Phase der Anforderungsanalyse erstellt wurde.

5.4.4.2 Design

Der Designteilschritt ist die unmittelbare Vorbereitung für die nachfolgende Implementierung. Die in der Analyse klar umrissene Aufgabenstellung muss nun in einen realisierbaren Entwurf umgesetzt werden. Dabei kann oft auf bereits existierende Komponenten zurückgegriffen werden. Neben dem Entwurf der nötigen Klassen ist die Entscheidung, wie diese Wiederverwendung erfolgen soll ein zentraler Punkt des Designteilschrittes. In Abschnitt 5.3 wurde bereits die Möglichkeit zusammengesetzter Komponenten diskutiert und die beiden grundsätzlich möglichen Varianten anhand von Beispielen eingeführt. Da jede **BOOGA**-Komponente grundsätzlich auch immer eine Klasse ist, kann neben den dort besprochenen Kompositionsvarianten natürlich auch die Vererbung eingesetzt werden, um Funktionalität und damit Quellcode mit den Implementierungen anderer Komponenten zu teilen.

Ein weiterer wichtiger Punkt ist der Entwurf oder die Wiederverwendung allfällig nötiger Basisabstraktionen oder geometrischer Objekte der Bibliotheks- oder Frameworkschicht. Auch hier muss eine Entscheidung über Wiederverwendung, Weiterentwicklung oder Neuentwurf der entsprechenden Klassen gefällt werden. Wenn immer möglich, ist dabei der Wiederverwendung der Vorzug über die Weiterentwicklung und der Weiterentwicklung der Vorzug über den Neuentwurf zu geben.

Ziel

Das Ziel dieses Teilschrittes ist ein Entwurf der neu zu entwickelnden Komponente. Im Zentrum sollte dabei die grösstmögliche Wiederverwendung stehen. Neu zu entwickelnde Klassen sollten so entworfen werden, dass sie sich einfach in das Komponentenframework integrieren lassen um späteren Anwendern zur Verfügung zu stehen.

Vorgehen

Je nach Komplexität der zu lösenden Aufgabe kann sich der Entwurf lediglich darauf beschränken, die Basisklasse auszuwählen, von der die neue Komponente abgeleitet werden soll. Ist diese allerdings sehr komplex, so werden eine ganze Anzahl von Klassen zu entwerfen und neue Protokolle zu definieren sein. Hierzu können die bekannten Verfahren und Notationen aus den unterschiedlichen Methoden herangezogen werden.

Die zentralen Mechanismen die beim Entwurf und der nachfolgenden Implementierung einzuhalten sind, werden in [Streit, 1997] ausführlich beschrieben.

Resultat

Die Resultate dieses Teilschrittes sind – je nach Komplexität mehr oder weniger ausführliche – Klassen-, Objekt- und Interaktionsdiagramme⁹. Weiter muss festgelegt werden, wo sich die neu zu erstellende Komponente in die Klassenhierarchie der **BOOGA**-Komponenten einfügt.

5.4.4.3 Implementierung

Die Implementierung ist die Umsetzung des vorausgegangenen Entwurfes in C++¹⁰. Die Realisierung erfolgt iterativ in den durch den Releaseplan vorgegebenen Einzelschritten.

Ziel

Das Ziel dieses Teilschrittes ist die Erstellung ausführbarer Komponenten und der dazugehörigen Klassen.

Vorgehen

Für die Implementierung empfiehlt sich die Verwendung der für **BOOGA** bereitgestellten Entwicklungsumgebung (vgl. Abschnitt 5.5), welche im wesentlichen auf einem integrierten Entwicklungswerkzeug inklusive grafischen Hilfsmitteln, unter anderem zur Visualisierung der Klassenhierarchie, beruht.

Resultat

Die Resultate dieses Teilschrittes werden, nach einer kurzen Testphase im nächsten Teilschritt, in die Phase der Anwendungskomposition einfließen.

⁹Diese Namensgebung bezieht sich auf die Verwendung der Notation von Booch.

¹⁰Soll bestehender C-Code übernommen werden, so ist dies natürlich durch die Verwendung eines *Wrappers* möglich.

5.4.4.4 Test

Der Umfang der Testphase hängt wiederum stark von der Komplexität der erstellten Komponente ab. Bei umfangreicherer Aufgaben werden die Komponenten ausführlicheren Tests unterzogen werden müssen, bei einfachen Komponenten können die Tests oft vereinfacht werden.

Ziel

Die im vorderen Schritt erstellten Komponenten müssen compiliert und mit den restlichen Komponenten und dem Komponentenframework ‘gelinkt’ werden können.

Vorgehen

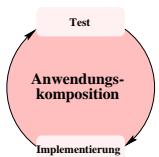
Um aufwendigere Komponenten ausführlich zu testen, kann es sinnvoll sein, eine eigene einfache Testapplikation zu definieren, welche entweder nur auf die neue Komponente aufbaut oder, falls dies nicht möglich ist, zusätzlich bereits gründlich getestete Komponenten einschliesst. Oftmals kann auf eine selbständige Testapplikation verzichtet werden und statt dessen die endgültige Anwendung schrittweise erweitert werden, wodurch der Testteilschritt in der Komponentenentwicklungsphase ohne Nachteile minimiert wird.

Resultat

Die Funktionalität der Komponenten soll soweit sichergestellt sein, dass diese in die nächste Phase einfließen können. Dies bedeutet noch nicht eine vollständig abgeschlossene Testphase, aber die ‘offensichtlichsten’ und grössten Fehler sollten entfernt sein.

5.4.5 Anwendungskomposition

In der Phase der Anwendungskomposition geht es darum, aus den neu erstellten und wiederverwendeten Komponenten eine lauffähige Anwendung zusammenzubauen. Je nach Art und Komplexität der Anwendung wird das eigentliche Programm nur, wie in Abschnitt 5.3 besprochen, aus den impliziten oder expliziten Verknüpfungen der Komponenten sowie deren Instanzierung und Konfiguration bestehen. Wird die zu lösende Aufgabe komplexer und beinhaltet beispielsweise eine aufwendige grafische Benutzerschnittstelle, so werden die Definitionen der Dialoge und die Realisierung der *Callback Funktionen* einen wesentlichen zusätzlichen Aufwand in dieser Phase bedeuten.¹¹



¹¹Das Schwergewicht von **BOOGA** liegt nicht in der Erstellung von grafischen Benutzeroberflächen (*Graphical User Interfaces, GUI*), obwohl einige Anwendungen mit **BOOGA** realisiert wurden, die über recht aufwendige Benutzerschnittstellen verfügen

Aus Sicht des Vorgehensmodells können diese zusätzlichen Aktivitäten wie spezialisierte Komponenten betrachtet werden und in einer oder mehreren Iterationen analog zur Komponentenentwicklungsphase entworfen und realisiert werden.

Ziel

Das Ziel dieser Phase ist das Zusammensetzen der in den vorausgegangenen Phasen erstellten Komponenten zu einer lauffähigen und korrekten Version der Anwendung.

Vorgehen

Bei der iterativen Umsetzung ist gemäss der während der Phase der Anforderungsanalyse aufgestellten Releaseplanung vorzugehen. Grundsätzlich sind zwei Stufen der Iteration ineinander verschachtelt: zu Beginn werden in der Anwendungskompositionssphase noch nicht alle für die endgültige Funktionalität nötigen Komponenten verwendet. Des Weiteren können die einzelnen Komponenten noch in einer nicht den definitiven Anforderungen genügenden Form aus der vorangegangenen Phase der Komponentenentwicklung hervorgegangen sein.

In den folgenden Iterationen des gesamten Prozesses werden die einzelnen Komponenten vervollständigt und die noch fehlenden Komponenten schrittweise erstellt. In den Iterationen der Anwendungskompositionssphase wird die Anwendung an sich schrittweise erweitert und vervollständigt.

Im Idealfall, d.h. wenn die Anwendung ausschliesslich aus Komponenten besteht, ist die Phase der Anwendungskomposition eine reine Konfigurationsarbeit.

Resultat

Die Resultate dieser Phase sind schrittweise in der Funktionalität zunehmende Versionen der Anwendung.

5.4.5.1 Implementierung

Im Teilschritt ‘Implementierung’ erfolgt die Realisierung des Hauptprogrammes, das nach einer Initialisierungsphase die nötigen Komponenten

(z.Bsp. [Habegger, 1996]). Ein speziell auf die Realisierung von GUI's ausgerichtetes Framework (z. Bsp. ET++, vgl. [Gamma, 1992]) kann in diesem Bereich zu besseren Resultaten bezüglich Realisierungsaufwand führen. Die **BOOGA**-Anwendungen mit grafischen Benutzerschnittstellen verwenden das Toolkit wxWINDOWS [Smart, 1997] um die Programmierung der Oberfläche zu vereinfachen. Da wxWINDOWS für unterschiedliche Betriebssysteme und Benutzeroberflächen angeboten wird, wird gleichzeitig die gute Portabilität von **BOOGA** gewahrt.

instanziert und wie bereits besprochen zur gewünschten Funktionalität verknüpft.

BOOGA bietet in seiner jetzigen Form vor allem Unterstützung für die Dekompositionsphase (durch die allgemeine, wiederverwendbare Architektur, die einheitliche Form der Komponenten sowie die Komponentenbibliothek) und die Komponentenentwicklungsphase (durch die Framework- und Bibliotheksschichten, sowie die bestehende Komponentenhierarchie). In späteren Erweiterungen von **BOOGA** könnte auch die Phase der Anwendungskomposition und hier speziell der Teilschritt Implementierung profitieren, indem auf eine C++-Programmierung verzichtet wird und eine *Scriptingsprache*¹² Verwendung findet. Darauf aufbauend wären auch *Visual Programming* Ansätze denkbar. Auf diese Weise könnte eine Brücke zwischen der Dekomposition- und der Kompositionsphase geschlagen werden.

Ziel

Das Ziel dieses Teilschrittes ist die Realisierung des Hauptprogrammes, welches die einzelnen Komponenten instanziert, konfiguriert und miteinander verknüpft.

Vorgehen

Die während der Dekompositionsphase gefundene Zerlegung ist die Basis für diesen Teilschritt. Auch hier spielt die Releaseplanung wiederum eine wichtige Rolle. Ist vorgesehen, in einem frühen Release nur eine bestimmte Teilfunktionalität zu realisieren, so muss dies hier selbstverständlich berücksichtigt werden.

Die einzelnen Komponenten müssen instanziert und mit den korrekten Parametern konfiguriert werden, um die ihnen zugesetzte Aufgabe zu erfüllen.

Die Implementierung des Hauptprogrammes ist – abgesehen von Anwendungen mit einer grafischen Benutzeroberfläche – wesentlich einfacher als die Entwicklung neuer Komponenten.

Resultat

Nach Abschluss dieses Teilschrittes sollte ein gegenüber der vorausgegangenen Iteration erweitertes oder verbessertes Hauptprogramm vorliegen, das mit den bereits vorhandenen Komponenten zu einer vollständigen Anwendung kombiniert werden kann.

¹²Beispielsweise Tcl/Tk oder Perl.

5.4.5.2 Test

In diesem Teilschritt werden unterschiedliche Ziele verfolgt: einerseits soll das im vorausgegangenen Teilschritt erstellte Hauptprogramm getestet werden, andererseits werden die neu erstellten Komponenten wahrscheinlich zum ersten Mal produktiv, die aus der Komponentenbibliothek entnommenen teilweise zum ersten Mal in der entsprechenden Kombination getestet. Es können also bei den Tests Fehler auftreten, die in einem erneuten Durchlauf des Implementierungsschrittes der Anwendungskompositionssphase oder solche, die in einer erneuten Anwendung der Komponentenentwicklungsphase korrigiert werden müssen.

Ziel

Das Ziel dieses Teilschrittes ist, Fehler des Implementationsteilschrittes oder der Komponentenentwicklungsphase zu finden. Die Behebung der Fehler kann in diesem Teilschritt oder – bei umfangreicher Änderungen – in den Implementierungsteilschritten der entsprechenden Phasen erfolgen.

Vorgehen

Die Anwendung wird im vorausgegangenen Teilschritt zusammengesetzt. Dies sollte schrittweise erfolgen, so dass während des Testens nicht die gesamte Anwendung in einem Schritt nach Fehlern abgesucht werden muss.

Resultat

Die Anwendung sollte mit jedem Durchlauf des Testteilschrittes weniger Fehler aufweisen. Ist die gemäss Releaseplan nötige Funktionalität eines Releases realisiert und getestet, kann der nächste begonnen werden. Sind alle geplanten Releases beendet und getestet, so ist die Entwicklungsperiode der Anwendung beendet.

5.4.6 Die Frameworkpflege

In diesem Kapitel lag das Schwergewicht bisher in der Entwicklung von neuen Anwendungen. An verschiedenen Stellen wurde aber bereits angetönt, dass das Komponentenframework **BOOGA** mit einer neuen Aufgabe oftmals Erweiterungen erfährt. Diese Erweiterungen können einerseits sehr anwendungsspezifisch, andererseits aber auch allgemeiner Natur und somit für zukünftige Anwendungen von grossem Nutzen sein.

Die Erfahrung hat gezeigt, dass Entwickler, welche sich intensiv mit **BOOGA** auseinandergesetzt haben, die in den unterschiedlichen Schichten des Komponentenframeworks enthaltenen wiederverwendbaren Abstraktionen in den eigenen Anwendungen bereitwillig einsetzen, sofern

sie den Anforderungen gerecht werden. Bei jeder bisher erstellten, nicht-trivialen Anwendung mussten jedoch neue Abstraktionen eingeführt oder bestehende Klassen erweitert oder angepasst werden. Diese Erweiterungen und Ergänzungen müssen auf die eine oder andere Weise wieder in **BOOGA** zurückfliessen. Nur so kann gewährleistet werden, dass nicht unterschiedliche Entwickler dasselbe Problem auf unterschiedliche, inkompatible Art lösen. Würde dies in erheblichem Masse auftreten, entstünden wieder Insellösungen, welche nicht mehr integriert werden könnten. Dies würde dem Plattformgedanken von **BOOGA** klar widersprechen.

Aufgrund der unterschiedlichen Kenntnisstände der Anwendungsentwickler¹³ sollte es diesen nicht überlassen werden, die Elemente des Komponentenframeworks in eigener Regie abzuändern. Der folgende Abschnitt bespricht die Organisation des gesamten **BOOGA**-Teams detaillierter. An dieser Stelle soll vorweggenommen werden, dass das Team, um die aufgeführten Problemstellungen zu lösen, in zwei Gruppen mit unterschiedlichen Aufgaben aufgeteilt wurde. Das Frameworkteam ist für die Pflege und Weiterentwicklung von **BOOGA** zuständig. Das Anwendungsteam erstellt Anwendungen oder bestimmte Erweiterungen von **BOOGA**, dies allerdings immer in Absprache mit Mitgliedern des Frameworkteams. Die Resultate des Anwendungsteams werden nach erfolgreicher Implementierung, koordiniert vom Frameworkteam, in das Komponentenframework integriert.

Das Framework selbst wird also ebenfalls in einem iterativen Prozess (vgl. Abbildung 5.8) weiterentwickelt, vergleichbar mit demjenigen der Komponentenentwicklungsphase. Neue Anforderungen werden analysiert (Teilschritt 1), anschliessend ein Entwurf aufgestellt (Teilschritt 2) und implementiert (Teilschritt 3). Tests (Teilschritt 4) beenden schliesslich eine Iteration der Frameworkpflege.

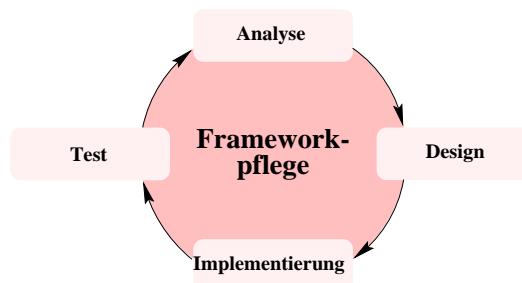


Abbildung 5.8

Die Frameworkpflege ist ein andauernder, iterativer Prozess. Solange das Framework verwendet werden soll, muss ein bestimmter Aufwand in die Pflege und Weiterentwicklung gesteckt werden.

Besonders wichtig ist die Schnittstelle zwischen den beiden iterativen Prozessen der Anwendungsentwicklung und der Frameworkpflege, sowie

¹³Die meisten der Studenten, die mit **BOOGA** arbeiteten, kamen zum ersten Mal mit einem der Gebiete Computergrafik, objektorientierte Technologien, Programmierung in C++, Arbeiten in einem grösseren Team, etc. in Berührung.

deren Synchronisation. Eine fehlerhafte Änderung im Framework wird wohl mehr als einen Anwendungsentwickler in Bedrängnis bringen. Die Schnittstelle sowie der Informationsfluss sind dadurch sichergestellt, dass jede Anwendungsentwicklung von einem Mitglied des Frameworkteams betreut wird. Die technischen Vorkehrungen, die ergriffen wurden, um die Synchronisation zwischen den Prozessen sicherzustellen, werden ebenfalls im nächsten Abschnitt besprochen.

5.5 Projektorganisation

Ein wichtiges Element für erfolgreiche Wiederverwendung, das bisher noch nicht angesprochen wurde, ist die Organisation der Projektteams. Eine Organisationsform, die sich sehr bewährt hat und die auch in der Literatur [Booch, 1994b; Gamma und Weinand, 1996; Jacobson et al., 1992] zu finden ist, ist die Aufteilung in ein Frameworkteam und ein oder mehrere Anwendungsteams.

Obwohl die im folgenden besprochene Projektorganisation nicht zwangsläufig mit dem in diesem Kapitel diskutierten Vorgehensmodell verknüpft ist, stellt sie doch ein wichtiges Hilfsmittel dar, um die im vorausgegangenen Abschnitt behandelte Frameworkpflege mit der Anwendungsentwicklung zu koordinieren.

Die Aufgaben des *Frameworkteams* sind die Wartung und Weiterentwicklung des Frameworks sowie die Beratung der Anwendungsteams. Die Mitglieder dieses Teams benötigen genaue Kenntnisse des Frameworks und dessen Mechanismen. Sehr wichtig ist auch, dass die *Vision*, der architektonische Gedanke des Frameworks allen bekannt ist und mitgetragen wird.

Teams

Das *Anwendungsteam*¹⁴ verwendet das Framework um damit Anwendungen zu entwickeln. Es wird, vor allem in der Anfangsphase der Anwendungsentwicklung oder wenn die Teammitglieder noch wenig Erfahrung mit dem Framework haben, eng mit dem Frameworkteam zusammenarbeiten. Dieses berät sie bezüglich der Dekomposition (vgl. Abschnitt 5.4.3) und allfällige nötiger Erweiterungen des Frameworks (vgl. Abschnitt 5.4.6).

Frameworkerweiterungen sollten in der Regel mit dem Frameworkteam abgesprochen werden. Je nach dessen Beurteilung ergeben sich daraufhin grundsätzlich zwei unterschiedliche Arten, um die Erweiterungen zu realisieren. Erkennt das Frameworkteam aufgrund des Überblicks über die Aktivitäten der Anwendungsteams oder durch die profunde Kenntnis des Frameworks ein allgemeines Interesse an der Erweiterung, so wird sie vom Frameworkteam vorgenommen oder zumindest koordiniert und kontrolliert. Ist die Erweiterung sehr anwendungsspezifisch, so wird das Frameworkteam allenfalls beratend tätig und die Neuerungen werden nicht in das Framework integriert.

¹⁴In der Folge wird stets nur im Singular von *einem* Anwendungsteam gesprochen. Die Meinung ist dabei, dass *ein* Anwendungsteam für *eine* Anwendung verantwortlich ist. Werden mehrere Anwendungen parallel entwickelt, so werden dafür mehrere Teams eingesetzt. Im Fall von **BOOCH** wurden die Anwendungen in der Regel von jeweils nur einer Person entwickelt. Es wird trotzdem der Begriff *Team* verwendet.

Aufgaben

Das Frameworkteam hat somit im Detail die folgenden Aufgaben:

- Die Mitglieder des Frameworkteams setzen sich mit den Problemstellungen der Anwendungsteams auseinander und wirken beratend bei der Anwendungserstellung mit. Das Frameworkteam ist somit ein Pool von ‘Chef-Architekten’. Dies hat mehrere Vorteile:
 - Die Mitglieder des Frameworkteams lernen durch ihre Mitarbeit die Probleme der Anwendungsentwickler kennen: welche Problemstellungen müssen gelöst werden, wo sind Lücken im Framework, wo gibt es Inkonsistenzen, welche den Lernaufwand vergrössern, wo liegen Probleme bezüglich Laufzeit- oder Speichereffizienz, etc.
 - Die Anwendungsentwickler fühlen sich ernstgenommen und können Einfluss auf die Gestaltung des Frameworks nehmen. Die Akzeptanz des Frameworks als ganzes steigt dadurch.
 - Die Anwendungen weisen eine einheitliche Struktur auf und neu entwickelte Softwareelemente können später bei Bedarf einfacher übernommen werden.
- Neue Softwareartefakte werden ausschliesslich durch das Frameworkteam ins Framework integriert. Die Neuerungen werden getestet und allenfalls an die für das Framework gültigen Konventionen bezüglich Namensgebung, Dokumentation und Stil angepasst.
- Durch Erweiterungen und Korrekturen wird das Framework instabiler. Änderungen an bestehendem Quellcode führen früher oder später zu Inkonsistenzen, die behoben werden müssen. Dies kann dazu führen, dass neue Basisklassen eingeführt und bestehende Klassen verschwinden oder angepasst werden müssen. Die Arbeit an einem Framework, das eingesetzt wird, ist niemals abgeschlossen.
- Die Behebung von Fehlern im Framework ist schliesslich ebenfalls Aufgabe des Frameworkteams.

Vorteile

Die Aufteilung in zwei unterschiedliche Gruppen drängt sich aus mehreren Gründen auf:

- Das Framework stellt die Basis für viele Arbeiten dar. Ein einzelner Anwendungsentwickler hat aber – mit gutem Grund – vor allem das Gelingen *seines* Projektes zum Ziel. Muss jemand unter Zeitdruck eine Anwendung schreiben und hat gleichzeitig die Kompetenz das

Framework anzupassen, so ist die Gefahr gross, dass seine Änderungen mit den Anforderungen anderer Teams kollidieren, da die Zeit nicht zur Verfügung steht, sich ausführlich genug mit anderen Teams abzusprechen.

- Die Entwicklung von Anwendungen und die Weiterentwicklung des Frameworks stellen, wie in Abbildung 5.9 illustriert, Zyklen dar,

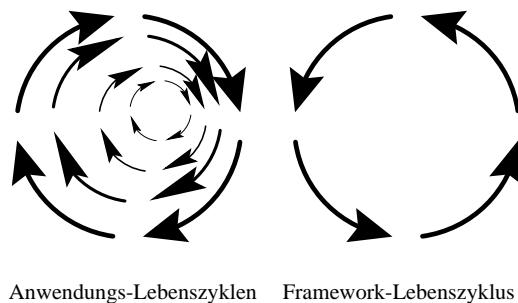


Abbildung 5.9

Die Lebenszyklen mehrerer Applikationen müssen mit dem Lebenszyklus des Frameworks synchronisiert werden.

welche synchronisiert werden müssen. Dazu eignen sich (vgl. auch [Gamma und Weinand, 1996]) folgende Lösungsansätze:

- Änderungen werden in kleinen Schritten durchgeführt und in kurzen Intervallen auf ihre Korrektheit überprüft. Im Idealfall ist das Framework jeden Abend in einer stabilen Lage verfügbar.
- Die Anwendungsentwickler werden nicht mit zu häufigen Versionswechseln des Frameworks überlastet. Es werden nur stabile, getestete Versionen des Frameworks freigegeben.
- Bei Frameworkänderungen sollte nach Möglichkeit darauf geachtet werden, dass die von den Anwendern verwendeten Schnittstellen unverändert bleiben. Sollte dies nicht möglich sein, so können *Erweiterungen* (zusätzliche Methoden) in der Regel ohne weitere Probleme eingeführt werden. Sind Änderungen bestehender Methoden unumgänglich, so ist es oft sinnvoll, eine Zeitlang sowohl die alte wie auch die neue Schnittstelle anzubieten, um so durch Verlängerung der Transitionsphase das Risiko zu minimieren, bei einem Anwendungsprojekt in einer kritischen Phase zusätzliche Probleme und Verzögerungen einzubringen. Die Verwendung einer älteren Version des Frameworks sollte zudem für jeden Entwickler immer möglich sein.
- Sind grössere Änderungen unumgänglich, so müssen diese geplant und mit den Teams abgesprochen werden. Eine genaue

Anleitung durch das Frameworkteam, welche Änderungen vorzunehmen sind, hilft die Aufwände zu minimieren.

Die saubere Trennung der beiden Zyklen wird durch die Aufteilung in einzelne Teams gefördert. Es wird so sichergestellt, dass der Pflege des Frameworks die nötige Bedeutung beigemessen wird und nicht die unter Termindruck stehenden Anwendungen übergewichtet werden.

- Speziell in einer industriellen Umgebung muss berücksichtigt werden, dass an Anwendungsentwickler und Frameworkentwickler andere Anforderungen gestellt werden. Während die Frameworkentwickler ein fundiertes Wissen der objektorientierten Technologien aufweisen müssen, ist es für die Anwendungsentwickler wichtiger, gute Kenntnisse der Abläufe im Anwendungsbereich zu haben.

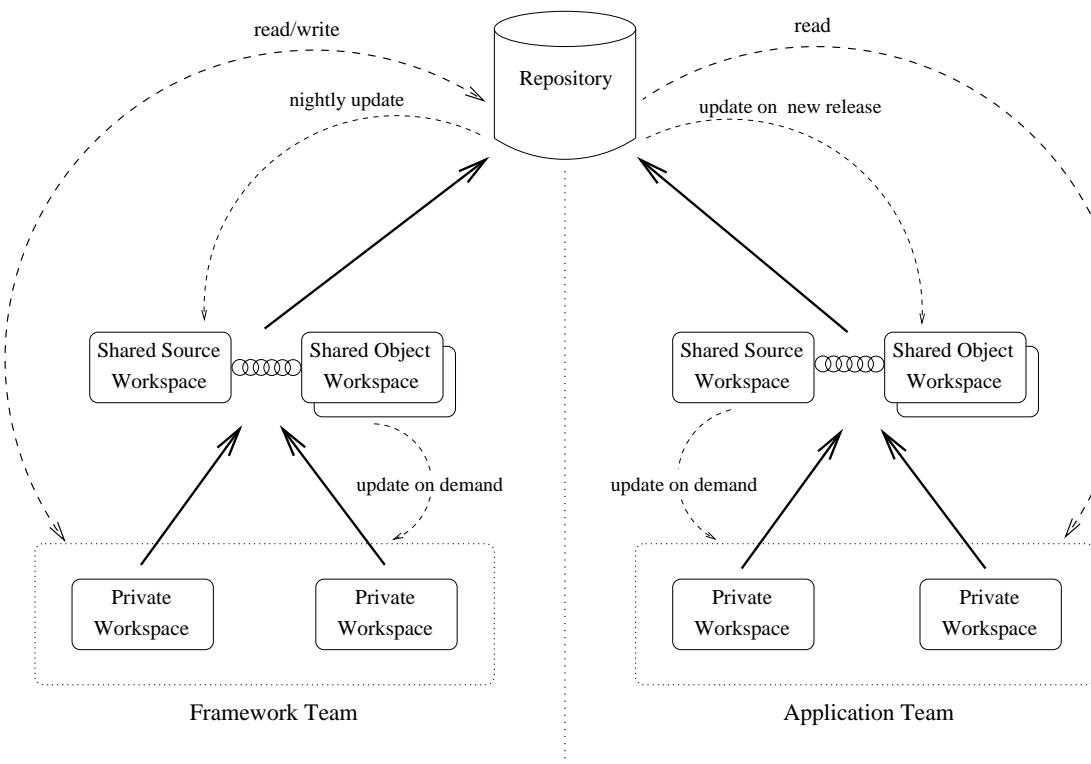
Umsetzung

Bei der Arbeit an **BOOGA** wurden diese Ansätze in die Praxis umgesetzt. Christoph Streit und der Autor dieser Arbeit stellten das Frameworkteam dar, Studenten, welche eine Semester- oder Diplomarbeit schrieben, die Anwendungsteams. Die einzelnen Arbeiten wurden durch das Frameworkteam betreut. Änderungen am Framework wurden ausschliesslich durch das Frameworkteam vorgenommen.

Die angesprochene Trennung in zwei getrennte Zyklen wurde durch die Gestaltung der Entwicklungsumgebung unterstrichen (vgl. Abbildung 5.10). Alle **BOOGA**-Mitarbeiter verwenden dieselbe Entwicklungsumgebung basierend auf SNiFF+. Der Quellcode wird mit Hilfe eines Versionskontrollsystems verwaltet. Dies erlaubt, jederzeit auf ältere Versionen einer Datei zurückzugreifen.

Jeder Entwickler hat seinen eigenen, privaten Arbeitsbereich (*Private Workspaces*). Dies ist eine Verzeichnisstruktur, in der im wesentlichen nur die Dateien enthalten sind, an denen der betreffende Entwickler gerade arbeitet. Alles andere ist zentral für alle verwendbar abgelegt (*Shared Workspaces*). Bei der Entwicklung von **BOOGA** wurde diese zentrale Ablage doppelt geführt: die Anwendungsentwickler bauen auf einer stabilen Frameworkversion auf, die Frameworkentwickler ändern das Framework in einem getrennten Bereich. Trotzdem haben (was zum Beispiel für die rasche Fehlerbehebung hilfreich sein kann) alle über das Versionkontrollsystem ständig Zugriff auf alle Versionen der einzelnen Dateien, also auch auf die neueste.

Um die Konvention, dass nur die Frameworkentwickler das Framework ändern dürfen, durchzusetzen, haben die Anwendungsentwicklungsteams ausschliesslich lesenden Zugriff auf die Frameworkdateien im Versionkontrollsystem.

**Abbildung 5.10**

Der folgende Abschnitt fasst dieses Kapitel zusammen. Es wird erläutert, dass das vorgestellte Modell auch für andere Komponentenframeworks Anwendung finden kann.

Die Struktur der verschiedenen Arbeitsbereiche sowie deren Abhängigkeiten.

Das *Repository* enthält die Daten des Versionkontrollsystems.

5.6 Zusammenfassung

In diesem Kapitel wurde ein Vorgehensmodell zur Erstellung von Anwendungen mit dem Komponentenframework **BOOGA** eingeführt. Die Ausführungen fassen die Erfahrungen des Autors und anderer Entwickler zusammen, beruhend auf vielen Anwendungen, die mit **BOOGA** entwickelt wurden. Die hier vorgestellten iterativen Phasen und Teilschritte des gesamten Prozesses haben sich in der praktischen Arbeit, der Entwicklung von Anwendungen, der Weiterentwicklung von **BOOGA** sowie der Betreuung von studentischen Arbeiten herauskristallisiert und bewährt. Sie beruhen im weiteren zu wesentlichen Teilen auf Ideen, die auf den erwähnten, anerkannten, objektorientierten Methoden aufbauen.

Beim Studium der vorgeschlagenen Methode lässt sich leicht erkennen, dass keinerlei Bezug zu dem Anwendungsgebiet von **BOOGA**, der Computergrafik, besteht. Tatsächlich ist das Vorgehen lediglich auf die *Struktur* von **BOOGA**, d.h. den recht grobkörnigen Komponentenbegriff (vgl. Definition 2.15 auf Seite 58) und die spezielle Struktur des Komponentenframeworks (vgl. Abschnitt 4.3) abgestimmt und bietet somit ideale Voraussetzungen, um auf ähnliche Systeme übertragen zu werden.

Soll die Methode in die Praxis übergeführt und in Industrieprojekten eingesetzt werden, so werden sich aufgrund des stark iterativen Vorgehens Schwierigkeiten bei der Planung und Fortschrittskontrolle ergeben. Dies kann aber dadurch vermindert werden, dass sich die Anzahl sowie der Aufwand der Iterationen und somit der Ressourcenbedarf aus der Releaseplanung ableiten lässt. Die Releaseplanung stellt somit die Grundlage des Projektmanagements dar, falls auf einem stark iterativen Ansatz, wie dem hier vorgestellten, aufgebaut wird.

Als Zusammenfassung der in dieser Arbeit bisher diskutierten, wiederverwendungsbezogenen Themen lässt sich festhalten, dass eine erfolgreiche Umsetzung des Gedankens der Wiederverwendung im wesentlichen auf drei Pfeilern beruht:

1. *Eine Sammlung von Hilfsmitteln, welche die Wiederverwendung direkt unterstützen.*

BOOGA stellt ein solches Hilfsmittel dar, die in Abschnitt 5.5 vorgestellten Elemente der Entwicklungsumgebung sind weitere, wichtige Grundlagen der täglichen Arbeit. Darüber hinausgehend sind aber ab einer bestimmten Teamgrösse und einer gewissen Anzahl von Komponenten auch Hilfsmittel nötig, welche Informationen über die Komponenten speichern und die Komponenten katalogisieren. Ein möglicher Ansatz in diese Richtung wird in Kapitel 7 vorgestellt. Zusätzlich könnte sich auch ein Hilfsmittel, das die bereits

erwähnte visuelle Komposition von **BOOGA**-Anwendungen zulässt, als sehr hilfreich erweisen.

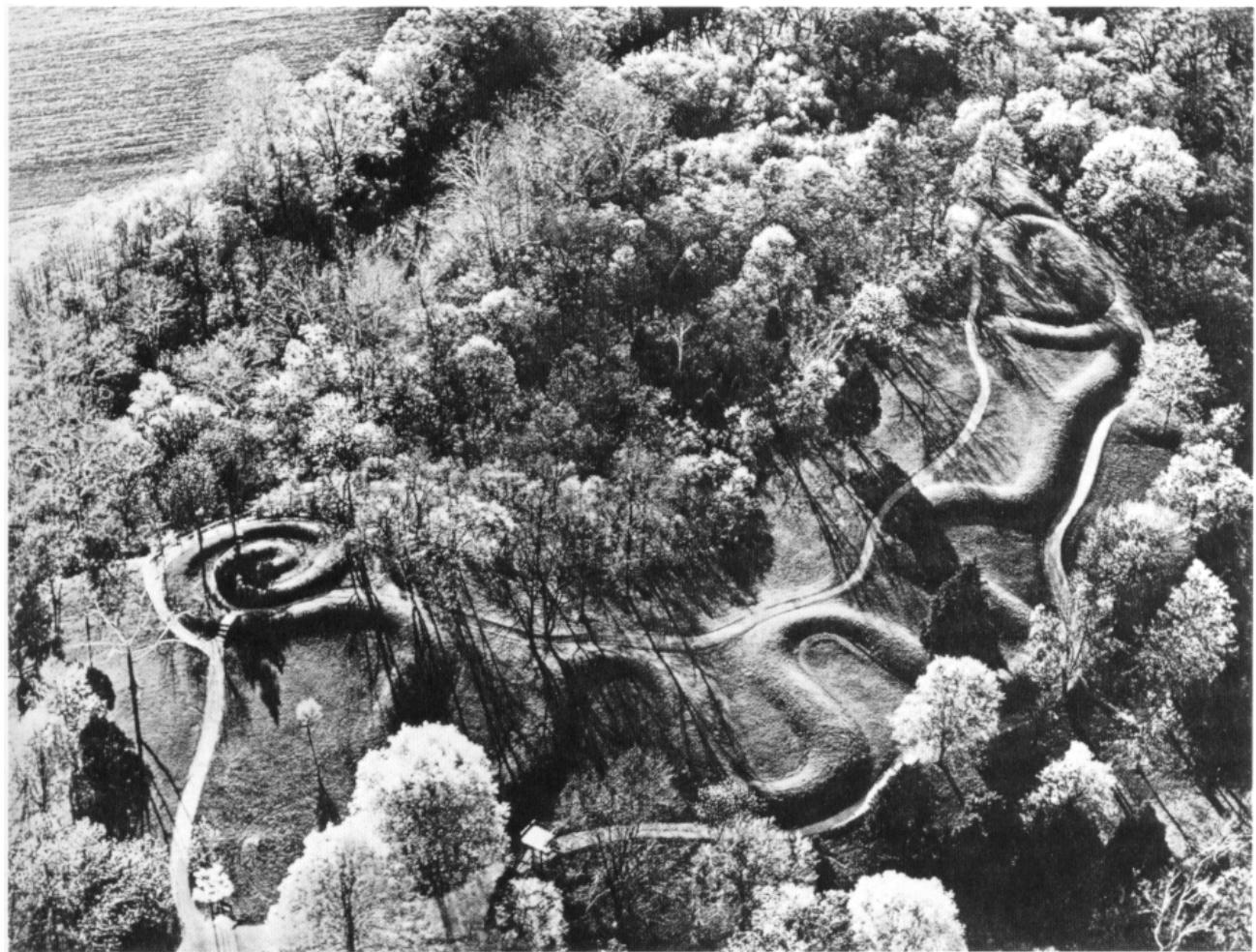
2. *Ein Vorgehensmodell, das den einzuhaltenden Prozess beschreibt.*

Der Prozess der Anwendungsentwicklung, derjenige der Frameworkpflege sowie das Zusammenspiel der beiden Prozesse muss beschrieben und allen Beteiligten bewusst sein. Erst durch ein Vorgehensmodell können die Stärken eines Komponentenframeworks voll ausgeschöpft werden. Es bietet den weniger erfahrenen Anwendern und Entwicklern einen Rahmen, in dem sie sich bewegen können und der es ihnen erlaubt, auch mit weniger Erfahrung zu Resultaten zu gelangen, die sich in die zugrundeliegende Philosophie einfügen.

3. *Eine Kultur, welche Wiederverwendung direkt fördert.*

Eine Organisation, welche aktive Wiederverwendung betreiben möchte, muss die Eigenverantwortung sowie die Motivation der Mitarbeiter fördern, in deren Ausbildung investieren sowie direkte Anreize schaffen, um einerseits wiederverwendbare Komponenten zu realisieren, andererseits existierende Komponenten einzusetzen.

Es sind Ansätze denkbar, welche die wiederverwendbaren Komponenten sogar innerhalb der Firma wie Produkte behandeln, mit denen gehandelt wird. Die Entwickler von besonders oft wiederverwendeten Komponenten oder die Entwickler von Anwendungen mit einem besonders kleinen Anteil von Neuentwicklungen sollten mit materiellen oder imateriellen Belohnungen ausgezeichnet werden.



Grosser Schlangen-Mound.
Adams County, Ohio.
Um 350 v. Chr. – 400 n. Chr.

Kapitel 6

Beispielanwendungen

*Wege entstehen dadurch,
dass wir sie gehen.*

Franz Kafka,
1883 - 1924

6.1 Einleitung

Das im vorangegangenen Kapitel vorgestellte Vorgehensmodell wird im Folgenden anhand einiger Beispiele demonstriert. Das Schwergewicht liegt dabei nicht darauf, die Beispiele möglichst vollständig zu entwickeln, d.h. mit allen nötigen Entwurfsdiagrammen oder sogar bis hin zum C++-Quelltext. Es wird vielmehr aufgezeigt, wie die verschiedenen Probleme auf unterschiedliche Art mit Hilfe von **BOOGA** gelöst werden können und wie das Vorgehensmodell eingesetzt wird. Zudem wird aufgezeigt, an welchen Stellen allenfalls Erweiterungen in den verschiedenen Schichten von **BOOGA** vorgenommen werden müssen.

Die vorgestellten Problemstellungen entstammen, bis auf eine Ausnahme, der Computergrafik. Die Ausnahme dient dazu, einen Eindruck der Anwendungsvielfalt von **BOOGA** zu geben. Zu Beginn der Entwicklung wurde nicht geplant, Anwendungen wie das in Abschnitt 6.4 vorgestellte CASETOOL zu realisieren. Die Stärken eines Systems zeigen sich aber oft bei unvorhergesehenen Anwendungen oder, wie sich [Johnson, 1996] ausdrückte:

*Good frameworks can be used for things,
the designers never dreamed of.*

Weitere Beispielanwendungen sind in [Streit, 1997] zu finden, wobei dort allerdings weniger Gewicht auf den Prozess gelegt wird als in dieser Arbeit.

Bei den folgenden Erläuterungen wird in der Regel darauf verzichtet, Irrwege zu beschreiben und iterativ zu korrigieren, wie dies bei ersten Anwendungsentwicklungen mit **BOOGA** der Fall wäre. Statt dessen sollen die Lösungen so präsentiert werden, wie sie von einem erfahrenen Entwickler entworfen würden. Dieses Vorgehen soll dabei helfen, einem Einsteiger das Vorgehen für die Entwicklung von **BOOGA**-Anwendungen zu vermitteln.

6.2 Anwendung “Wireframe”

Dieses Beispiel dient als einfache Einführung in die Denkweise und das Vorgehen der Anwendungsentwicklung mit Hilfe von **BOOGA**. Die resultierende Anwendung wird ebenfalls in [Amann et al., 1997] und [Streit, 1997] vorgestellt. An dieser Stelle liegt das Gewicht auf der Beschreibung des Lösungsprozesses. Dieses Beispiel eignet sich ausgezeichnet als Einstieg und kann hier aufgrund seiner Einfachheit sehr ausführlich behandelt werden.

6.2.1 Problembeschreibung

Eines der zentralen Anwendungsgebiete von **BOOGA** ist die Berechnung von Bildern anhand von Szenenbeschreibungen, also die Implementierung von Rendering-Verfahren. An dieser Stelle soll ein sehr einfaches Verfahren realisiert werden. Die Aufgabe besteht darin, die Elemente einer dreidimensionalen Szene als einfache Drahtgittermodelle darzustellen. Dabei brauchen für dieses Beispiel keine Verdeckungen und keine Beleuchtungsberechnungen berücksichtigt zu werden.

6.2.2 Anforderungsanalyse

Anforderungs-analyse

Um das in der – bewusst grob gehaltenen – Aufgabenstellung beschriebene Problem lösen zu können, ist einiges an Hintergrundwissen nötig, einerseits aus dem Bereich der Computergrafik¹, andererseits von **BOOGA**². Dieses Zusatzwissen wird in die im Folgenden zu formulierenden Anforderungen einfließen und dabei teilweise genauer erläutert werden.

Eine erste Aufgabe einer jeden Bildberechnungsanwendung besteht darin, die Beschreibung der darzustellenden Szene einzulesen. In **BOOGA** ist hierzu die **BOOGA Scene Description Language** (BSDL, vgl. Anhang A.2) definiert worden. Jede **BOOGA**-Anwendung, die Szenen einliest oder ausgibt, sollte diese Sprache verwenden.

Anforderung 1

Das Einlesen einer Szenenbeschreibung hat im BSDL-Format zu erfolgen.

In der Aufgabe ist nicht klar festgehalten, wie das Resultat ausgegeben werden soll. Bei **BOOGA**-Anwendungen ist es üblich, dass die Ausgabe auf verschiedene Arten und in unterschiedlichen Formaten erfolgen

¹vgl. hierzu [Foley et al., 1990].

²vgl. hierzu [Streit, 1997].

kann. Dies wird in der Aufgabenstellung nicht erwähnt, da es für einen **BOOGA**-Anwendungsentwickler selbstverständlich ist, solche Verallgemeinerungen einzubauen. In der Regel sind diese mit keinem zusätzlichen Aufwand verbunden. Ausgaben auf den Bildschirm, in eine Datei im **BOOGA** eigenen PIXI-Format (vgl. Anhang A.3) oder anwendungsspezifisch weiteren, speziellen Formaten sind für die vorliegende Aufgabenstellung sinnvoll. Bei einem Drahtgittermodell ist es sehr einfach möglich und für gewisse Aufgaben äußerst wünschenswert, das Resultat statt in einem Rasterformat in Vektorform auszugeben. Naheliegend ist beispielsweise die Ausgabe als *Encapsulated Postscript (EPS)*, welches einen von vielen Anwendungen unterstützten Standard zum Austausch von Grafiken darstellt.

Anforderung 2

Die Ausgabe des Resultates soll auf den Bildschirm oder im Postscript oder PIXI-Format erfolgen.

Zwischen dem Einlesen der Szene und der Ausgabe des Bildes – in welcher Form auch immer – muss selbstverständlich noch die eigentliche Berechnung erfolgen. Der Tatsache, dass sowohl Raster- wie auch Vektorformate für die Ausgabe unterstützt werden sollen, können weitere Anforderungen entnommen werden:

Anforderung 3

Das Drahtgittermodell muss in einem Vektorformat berechnet werden.

Anforderung 4

Die Vektdarstellung des Drahtgittermodells muss in ein Rasterbild konvertiert werden können.

Denkbar wäre auch eine Umkehrung der Anforderungen 3 und 4, so dass die Rasterdarstellung berechnet wird und bei Bedarf in eine Vektorform umgerechnet wird. Allerdings sind diese Verfahren algorithmisch wesentlich aufwendiger als das hier vorgeschlagene Vorgehen.³

Je nach grafischem Objekt muss die Gitterauflösung der Approximation passend gewählt werden. Eine Kugel braucht beispielsweise ein feineres Gitter um die Umrisse erkennen zu lassen als ein Würfel.

³Eigentlich wird hier nicht mehr eine saubere Anforderungsanalyse vorgenommen, da bereits Lösungsansätze und Wissen über die Realisierung vorweggenommen werden. Andererseits macht es auch keinen Sinn, an dieser Stelle so zu tun, als ob man sich noch nicht für einen bestimmten Weg entscheiden kann, wenn aufgrund des Fachwissens oder der Erfahrung bereits eine der Varianten klar gegenüber den anderen dominiert.

Anforderung 5

Die Objekte der Szene sollen als Drahtgittermodelle in 'ausreichender' Gitterauflösung dargestellt werden.

Bereits in der Problemstellung erwähnt wurde die folgende Anforderung:

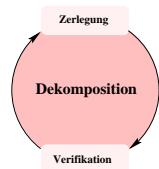
Anforderung 6

Vereinfachend brauchen keine Verdeckungen und Beleuchtungswerte berechnet zu werden.

Mit diesen Überlegungen kann die Anforderungsanalyse vorerst beendet werden. Sollten bei der Beschäftigung mit der Aufgabe weitere Anforderungen hinzukommen, können diese in einer späteren Iteration des Prozesses einfließen.

6.2.3 Dekomposition

In dieser Phase des Entwicklungsprozesses geht es darum, die Zerlegung in einzelne Komponenten vorzunehmen. Im Vordergrund steht dabei die Zerlegung in **BOOGA**-Komponenten. Auf einer höheren Abstraktionsstufe ist aber ebenfalls eine Zerlegung möglich. **BOOGA** wurde aufbauend auf einem Komponentenbegriff entwickelt, welcher der UNIX-Philosophie sehr nahesteht. Obwohl **BOOGA** mittlerweile auf andere Plattformen (Microsoft's Windows 95 und Windows NT 4.0) portiert wurde, sind die meisten Anwendungen unter UNIX entwickelt worden und folgen der dort üblichen Philosophie der Verknüpfung von Anwendungen mittels *Pipes*, die Ein- und Ausgabeströme der Prozesse miteinander verbinden.



Bei der Zerlegung einer Aufgabenstellung in Komponenten kann dem Umstand Rechnung getragen werden, dass UNIX-Anwendungen als eigenständige Elemente in die Dekomposition einfließen können. So wird für diese Aufgabenstellung davon ausgegangen, dass die folgenden Anwendungen (teilweise mit **BOOGA** erstellt) bereits existieren:

1. Eine ganze Anzahl von Programmen um Rasterbilder auf dem Bildschirm anzuzeigen ist entweder entgeltlich oder frei erhältlich . In einem ersten Schritt ist es nicht nötig, diese Aufgabe mit einer eigenen Komponente oder Bibliotheksfunktion zu lösen. Leider unterstützt aber keines dieser Programme das **BOOGA**-eigene Format **Pixi**. Es gibt jedoch eine ganze Anzahl von Standardformaten, die unterstützt werden.

2. Eine kleine **BOOGA**-Anwendung, **PIXI2PPM**, wandelt Bilder, welche im **PIXI**-Format vorliegen, ins **PPM**-Format um. Dieses Format wird von einer grossen Anzahl von Darstellungs- oder Konvertierungsprogrammen unterstützt.

Die eigentliche Anwendung **Wireframe** braucht also die Anforderung 2 nur noch zu einem Teil abzudecken. Lediglich die Ausgabe im **PIXI**- und **Postscript**-Format verbleibt noch als Forderung; die Ausgabe auf den Bildschirm wird bereits durch existierende Programme erfüllt. Dieser verbleibende Teil von Anforderung 2 wird im folgenden mit 2' bezeichnet.

Zerlegung

Die Anforderungen 5 und 6 werden vorerst nicht beachtet, da sie eher die Implementierung einer Komponente betreffen als eine eigene Teilaufgabe beschreiben. Sie werden demzufolge in der nächsten Phase behandelt. Die vorerst wichtigen Anforderungen lassen sich direkt in getrennte Komponenten umsetzen:

Anforderung 1: Einlesen der Szene

Diese Komponente muss eine Datei mit einer Szenenbeschreibung im **BSDL**-Format einlesen. Aufgrund dieser Beschreibung werden Instanzen der den geometrischen Objekten entsprechenden Klassen erzeugt.

Bei der gestellten Aufgabe geht es darum, eine dreidimensionale Szene einzulesen. Der Typ dieser Komponente ist **Operation3DTo3D**. Sie bekommt als Eingabe eine **World3D** und wird diese durch die in der Szenenbeschreibung gefundenen Elemente ergänzen.

Wie bereits bei der Formulierung der Anforderung 1 erwähnt wurde, gibt es sehr viele Aufgaben, welche ein Einlesen einer Szenenbeschreibung erfordern. Aus diesem Grund war eine Komponente, welche diese Aufgabe erfüllt, auch eine der ersten, die entwickelt und in der Komponentenbibliothek zur Verfügung gestellt wurde. Die erste Teilaufgabe ist also bereits durch die existierende Komponente **Parser3D** abgedeckt.

Anforderung 3 : Drahtgitterdarstellung berechnen

Dieses Teilproblem stellt die eigentliche Aufgabe dar. Da die Ausgangsszene drei-, die zu berechnende Drahtgitterdarstellung aber zweidimensional ist, handelt es sich bei der Komponente **Wireframe** um eine **Operation3DTo2D**.

Eine Komponente, welche diese Aufgabe erfüllt, existiert noch

nicht⁴ in der Komponentenbibliothek von **BOOGA** und muss in der folgenden Phase der Komponentenentwicklung entwickelt werden.

Anforderung 4 : Rasterkonvertierung vornehmen

Die vorgängig beschriebene Komponente **Wireframe** erzeugt eine Menge von Streckenobjekten im zweidimensionalen Raum. Diese müssen, soll eine Ausgabe auf den Bildschirm erfolgen, in eine Rasterdarstellung konvertiert werden. Da **BOOGA** auch zweidimensionale Objekte unterstützt, existiert bereits eine Komponente **Rasterize**, welche diese häufige Aufgabe erfüllt.

Anforderung 2' : Ausgabe im PIXI- und Postscript-Format

Die Ausgabe soll in zwei unterschiedlichen Formaten erfolgen:

- Für die Ausgabe in Postscript braucht die Rasterkonvertierung nicht durchgeführt zu werden. Stattdessen wird das Resultat der Komponente **Wireframe** der Komponente **PSWriter2D** übergeben, die aus einer zweidimensionalen Szene Postscript erzeugt.
- Soll ein Rasterbild ausgegeben werden, so muss die Komponente **Rasterize** ausgeführt werden, anschliessend muss das so erzeugte Bild in das PIXI-Format umgewandelt und ausgegeben werden. Diese Aufgabe wird von der Komponente **PixiWriter** wahrgenommen.

Die hier besprochene Zerlegung soll im folgenden Teilschritt der Verifikation einer kritischen Betrachtung unterzogen werden.

In Abschnitt 5.4.3.2 wurden folgende Kriterien zur Beurteilung von Komponentenkandidaten aufgestellt:

Verifikation

- Existiert bereits eine passende Komponente?
- Erfüllen die Komponenten atomare Teilaufgaben?
- Ist die Komplexität der Komponente gering?
- Sind die Komponenten allgemein verwendbar?

Im folgenden sollen die im vorausgegangenen Teilschritt identifizierten Komponenten gemäss dieser Kriterien bewertet und anschliessend die gesamte Zerlegung beurteilt werden. Zusätzlich werden bei den einzelnen

⁴Dies ist die Übungsannahme für dieses Beispiel. Tatsächlich wurde die Komponente **Wireframe** natürlich realisiert.

Komponenten jeweils die Zuordnung zu einer der vier Komponentenklassen (vgl. Abschnitt 4.2) aufgeführt.

Die erste Komponente liest eine Szenenbeschreibung im Format BSDL ein und baut eine `World3D` auf⁵, welche die Instanzen der den geometrischen Objekten entsprechenden Klassen enthält.

Parser3D

<i>Vorhanden</i>	Ja
<i>Atomar</i>	Ja
<i>Komplexität</i>	Gross ⁶
<i>Allgemeinheit</i>	Ja
<i>Typ</i>	<code>Operation3DTo3D</code>

Die `Wireframe`-Komponente muss neu erstellt werden und ist eigentlich spezifisch für diese Aufgabe. Die Komponente ist aber nicht nur für diese Anwendung brauchbar, sondern kann immer eingesetzt werden, wenn in einer Anwendung die Berechnung einer Drahtgitterdarstellung benötigt wird.

Wireframe

<i>Vorhanden</i>	Nein
<i>Atomar</i>	Ja
<i>Komplexität</i>	Gering
<i>Allgemeinheit</i>	Ja
<i>Typ</i>	<code>Operation3DTo2D</code>

Ebenfalls vorhanden und sehr allgemein einsetzbar sind die Komponenten `Rasterize`, `PSWriter2D`, und `PixiWriter`.

PSWriter2D

<i>Vorhanden</i>	Ja
<i>Atomar</i>	Ja
<i>Komplexität</i>	Gering
<i>Allgemeinheit</i>	Ja
<i>Typ</i>	<code>Operation2DTo2D</code>

⁵Eine analoge Komponente wird verwendet um Beschreibungen einer zweidimensionalen Szene einzulesen. Beide Implementierungen sind lediglich Adapter (vgl. [Gamma et al., 1995, Seite 139]) einer allgemeinen `Parser`-Klasse.

⁶Der Parser musste aus Gründen der Erweiterbarkeit so realisiert werden, dass neue geometrische Objekte in die Sprache integriert werden können, ohne den Parser an sich zu erweitern. Die *interne* Komplexität des Parsers ist recht gross. Trotzdem gelang es, die Verwendung des Parsers einfach zu halten. Die Erweiterung des Parsers um neue Szenenelemente ist ebenfalls recht einfach möglich.

Rasterize

<i>Vorhanden</i>	Ja
<i>Atomar</i>	Ja
<i>Komplexität</i>	Gering ⁷
<i>Allgemeinheit</i>	Ja
<i>Typ</i>	Operation2DTo2D

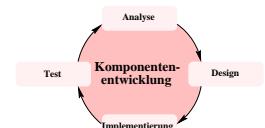
PixiWriter

<i>Vorhanden</i>	Ja
<i>Atomar</i>	Ja
<i>Komplexität</i>	Gering
<i>Allgemeinheit</i>	Ja
<i>Typ</i>	Operation2DTo2D

Die ganze Zerlegung kann als gut beurteilt werden, da lediglich *eine* neue Komponente erstellt werden muss. Die Komplexität dieser neuen Komponente ist äusserst gering und sie ist auch in anderen Anwendung einsetzbar. Der nächste Abschnitt bespricht die Entwicklung dieser neuen Komponente.

6.2.4 Komponentenentwicklung

Die Phase der Komponentenentwicklung kann bei dieser Aufgabe sehr kurz gehalten werden, da nur eine recht einfache Komponente zu entwickeln ist. Wie in der Einleitung erwähnt, liegt das Schwergewicht in diesem Kapitel auf der Analyse und dem Design. Ausnahmsweise soll aber bei diesem Beispiel der C++-Quellcode der neuen Komponente sowie der fertigen Anwendung (vgl. Abschnitt 6.2.5) vorgestellt werden, um zu zeigen, wie gering der Aufwand ist, diese zu realisieren.



Wie bereits erwähnt, fallen die Teilschritte Analyse und Design bei einfachen Komponenten oft zusammen. Die Komponente **Wireframe** ist sehr einfach, so dass hier keine explizite Unterscheidung zwischen den beiden Schritten gemacht wird.

Aufgrund der mächtigen Mechanismen in **BOOGA** kann die Komponente **Wireframe** sehr einfach realisiert werden. Normalerweise kommen in einer Szenenbeschreibung eine grosse Anzahl unterschiedlicher Typen von geometrischen Objekten vor: Dreiecke, Kugeln, Zylinder, Kegel, Würfel, Freiformflächen, etc. Für jedes dieser Objekte müsste normalerweise ein

Analyse, Design

⁷Die Realisierung der Rasterkonvertierungen für einzelne geometrische Objekte wie Strecken, Ellipsen, etc. unter Anwendung beliebiger Transformationen ist zwar algorithmisch anspruchsvoll, doch existieren hierfür ausreichend Algorithmen in der Literatur (z. Bsp. [Foley et al., 1990]).

Algorithmus implementiert werden, um die Drahtgitterdarstellung zu berechnen.

Jedes geometrische Objekt in der Frameworkschicht von **BOOGA** ist bereits dafür vorgesehen, sich selbst in eine andere Darstellung zerlegen zu können. Eine häufig gewählte, vereinfachende Darstellungsform ist beispielsweise die Zerlegung in Dreiecke. Beim Entwurf einer Komponente wird festgelegt, welche geometrischen Objekte erkannt werden können. Kann eine Komponente ein bestimmtes Objekt nicht verarbeiten, so wird dieses in einer anderen Darstellungsform der Komponente erneut präsentiert.⁸ Es leuchtet ein, dass dieses Vorgehen nicht zu einer besonders laufzeiteffizienten Implementierung führt. Wird Wert auf kurze Laufzeiten gelegt, so kann die Komponente mit zusätzlichen Verfahren ergänzt werden, um direkt Polygone, Würfel, Kugeln, etc. darzustellen.

Bei der Entwicklung dieser Komponente wird man sich somit in einem ersten Schritt darauf beschränken können, nur einzelne Dreiecke als Drahtgitter darzustellen. Diese Aufgabe ist äußerst einfach; es müssen lediglich die drei Eckpunkte des Dreieckes in Bildschirmkoordinaten umgerechnet⁹ und anschliessend drei zweidimensionale Strecken in der ‘Resultatwelt’ abgelegt werden. Da keine Beleuchtungsberechnungen nötig sind (vgl. Anforderung 6), werden die Strecken in einer Standardfarbe gezeichnet. Die Beschränkung auf Dreiecke erfüllt gleichzeitig auf elegante Art auch die Anforderung 5: die Zerlegung der Objekte in Dreiecke erfolgt stets in dem Objekttyp angepasster Genauigkeit: so wird eine Kugel von den in **BOOGA** implementierten Mechanismen automatisch in wesentlich mehr Dreiecke zerlegt, als beispielsweise ein Würfel.

Implementierung

Der folgende C++-Quellcode stellt die Schnittstellendeklaration der Klasse **Wireframe** dar. Sie wird von der Basisklasse **Renderer** abgeleitet, welche einige, allen Renderingverfahren gemeinsame Methoden zur Verfügung stellt. Ein Beispiel hierfür ist **getCamera**, die in der Implementierung der Methode **visit** verwendet wird, um die aktive Kamera, von der aus die Szene berechnet werden soll, zu ermitteln.

Der erste Teil (Zeilen 1 bis 16) ist bei vielen Komponenten ähnlich und stellt die ‘C++-Klasseninfrastruktur’ bereit. Interessanter ist der nächste Abschnitt (Zeilen 18 bis 23), der für jedes geometrische Objekt, welches die Komponente verarbeiten kann, eine spezielle **visit**-Methode zur Verfügung stellt. Die Auswahl der korrekten Methode erfolgt in der auf Zeile 34 deklarierten **dispatch**-Methode. Mit **postprocessing** auf Zeile 28 wird eine Methode der Wurzelklasse der Komponentenhierarchie überschrieben, die es erlaubt, Aktionen nach dem Durchlaufen des gesamten Szenengraphen auszuführen.

⁸Dieser Mechanismus wird detailliert in [Streit, 1997] beschrieben.

⁹Diese Aufgabe wird von der Klasse **Viewing3D** wahrgenommen.

```

1 class Wireframe : public Renderer {
2     declareRTTI(Wireframe);           // Enable RTTI support.
3
4     //~~~~~ Constructors, destructors, assignment ~~~~~
5     // Constructors, destructors, assignment
6     //~~~~~ Constructors, destructors, assignment ~~~~~
7     public:
8         Wireframe() {}
9     private:
10        Wireframe(const Wireframe&);           // No copies.
11
12    public:
13        // virtual ~Wireframe();           // Use default version.
14
15    private:
16        Wireframe& operator=(const Wireframe&); // No assignments.
17
18     //~~~~~ New methods of class Wireframe ~~~~~
19     // New methods of class Wireframe
20     //~~~~~ New methods of class Wireframe ~~~~~
21    private:
22        Traversal::Result visit(Triangle3D* obj);
23
24     //~~~~~ From class Component<> ~~~~~
25     // From class Component<>
26     //~~~~~ From class Component<> ~~~~~
27    protected:
28        virtual bool postprocessing();
29
30     //~~~~~ From class Visitor ~~~~~
31     // From class Visitor
32     //~~~~~ From class Visitor ~~~~~
33    public:
34        virtual Traversal::Result dispatch(Makeable* obj);
35    };

```

Die im folgenden aufgelistete Implementierung der Methode `visit` lässt sich grob in drei Teile unterteilen (eine detaillierte Beschreibung folgt nach dem Quellcode):

Zeilen 3 bis 13 : Initialisierung

Zeilen 14 bis 31 : Transformation der Dreieckspunkte von Weltkoordinaten in Bildschirmkoordinaten

Zeilen 32 bis 46 : Erzeugen der Streckenobjekte und Ablegen in der ‘Resultatwelt’

Die Methode schliesst mit dem Resultat `Traversal::CONTINUE` ab, was dem Traversierungsalgorithmus signalisiert, mit der Traversierung des Szenengraphen weiterzufahren.

```

1 Traversal::Result Wireframe::visit(Triangle3D* obj)
2 {
3     //
4     // Get current viewing parameters
5     //
6     const Viewing3D* viewing = getCamera()->getViewing();
7     //
8     // Get accumulated transformation of object
9     //
10    const TransMatrix3D& tm =
11        getTraversal()->getPath()->getLastTransform();
12
13    //
14    // Perform viewing transformation
15    //
16    Vector3D v3D[3]; Vector2D v2D[3];
17    for (int i=0; i<3; i++) {
18        v3D[i] = viewing->transformWorld2Screen(
19            tm.transformAsPoint(obj->getVertex(i)));
20
21        //
22        // Clip triangles behind the eye.
23        //
24        if (v3D[i].z() < EPSILON)
25            return Traversal::CONTINUE;
26
27        v2D[i].x() = v3D[i].x();
28        v2D[i].y() = v3D[i].y();
29    }
30
31    //
32    // Create Line2D objects in result world
33    //
34    Line2D* line;
35    ConstantTexture2D* texture;
36
37    for (int i=0; i<3; i++) {
38        texture = new ConstantTexture2D();
39        texture->setColor(Color::getDefault());
40
41        line = new Line2D(v2D[i], v2D[(i+1)%3]);
42        line->appendTexture(texture);
43
44        getResult()->getObjects()->adoptObject(line);
45    }
46
47    //
48    // Method finished successfully
49    //
50    return Traversal::CONTINUE;
51 }
52 }
```

Zeilen 3 bis 13 : Initialisierung

Zur Vereinfachung werden häufig benötigte Werte einmal bestimmt und zwischengespeichert.

Zeilen 14 bis 31 : Transformation der Dreieckspunkte von Weltkoordinaten in Bildschirmkoordinaten

Die einzelnen Dreieckspunkte werden zuerst (Zeile 20) vom Objektkoordinatensystem ins Weltkoordinatensystem und anschliessend (Zeile 19) ins Bildschirmkoordinatensystem umgerechnet. Sollte sich ein Eckpunkt des Dreiecks hinter dem Augpunkt befinden, so wird das ganze Dreieck ignoriert (Zeilen 22 bis 26).

Auf den ersten Blick erstaunt es, dass die Methode `transformWorld2Screen` einen `Vector3D` als Resultat liefert und nicht einen `Vector2D`. Bei der Transformation ins Bildschirmkoordinatensystem fällt neben den x- und y-Koordinaten noch die Distanz zum Augpunkt als Zusatzinformation ab. Diese kann verwendet werden, um Verdeckungen zu ermitteln. Auf Zeilen 28 und 29 werden diese Werte in `Vector2D`-Werte umgerechnet.

Zeilen 32 bis 46 : Erzeugen der Streckenobjekte und Ablegen in der 'Resultatwelt'

Soll in `BOOGA` ein Objekt eine konstante Farbe erhalten, so muss ihm eine `ConstantTexture2D` (respektive eine `ConstantTexture3D` im dreidimensionalen Fall) zugeordnet werden. Diese Textur wird auf Zeile 39 erzeugt, auf Zeile 40 initialisiert und auf Zeile 43 dem zuvor (Zeile 42) erzeugten `Line2D`-Objekt zugeordnet. Schliesslich (Zeile 45) wird das neue Objekt in die 'Resultatwelt' eingefügt.

Wie bereits erwähnt, erlaubt die Methode `postprocessing` Aktionen zu definieren, die nach dem Durchlaufen des gesamten Szenengraphen ausgeführt werden. Im Falle der `Wireframe`-Komponente muss in die resultierende `World2D` eine Kamera (`Camera2D`) eingefügt werden, welche anschliessend das Betrachten der neuen Szene erlaubt. Die Parameter der `Camera2D` werden dabei aus den Parametern der `Camera3D` der Eingabeszene bestimmt. Abschliessend wird auf Zeile 18 die Berechnung der *Bounding Rectangles* initiiert.

```

1  bool Wireframe::postprocessing()
2  {
3      //
4      // Create a new Camera2D to watch the newly created world
5      //
6      Camera2D* camera = new Camera2D;

```

```

7   camera->getViewing()->setResolution(
8       getCamera()->getViewing()->getResolutionX(),
9       getCamera()->getViewing()->getResolutionY());
10  camera->getViewing()->setWindow(
11      Vector2D(0,0),
12      Vector2D(
13          getCamera()->getViewing()->getResolutionX(),
14          getCamera()->getViewing()->getResolutionY()));
15  camera->setBackground(getCamera()->getBackground());
16
17  getResult()->getObjects()->adoptObject(camera);
18  getResult()->getObjects()->computeBounds();
19
20  return Renderer::postprocessing();
21 }

```

Als letztes muss noch die Methode `dispatch` implementiert werden. Diese ist dafür verantwortlich, die korrekten `visit`-Methoden der Komponente aufzurufen. Auf Zeile 3 wird festgelegt, dass die `Wireframe`-Komponente den Objekttyp `Triangle3D` unterstützt. Falls ein anderes Objekt übergeben wurde, wird dem Traversierungsalgorithmus mittels `Traversal::UNKNOWN` signalisiert, dass es sich bei dem übergebenen Objekt um einen der Komponente unbekannten Objekttypus handelt. Der Traversierungsalgorithmus wird daraufhin dieses Objekt zerlegen lassen und die Teile wiederum der `dispatch`-Methode übergeben.

```

1  Traversal::Result Wireframe::dispatch(Makeable* obj)
2  {
3      tryConcrete(Triangle3D, obj);
4
5      //
6      // Create decomposition for objects other than triangles.
7      //
8      return Traversal::UNKNOWN;
9  }

```

In diesem Abschnitt wurde der *gesamte* Code, der zur Implementierung der `Wireframe`-Komponente nötig ist, vorgestellt.

Test

Auf dieser Stufe wird sich der Test der `Wireframe`-Komponente darauf beschränken, diese ohne Fehler oder Warnungen zu compilieren. Die Funktionalität wird am besten getestet, wenn nach der nächsten Phase die eigentliche Anwendung zur Verfügung steht. Dies ist ein durchaus zulässiges Vorgehen, da diese Komponente das einzige neue Element ist, dass in die Anwendung einfließen wird.



6.2.5 Anwendungskomposition

Ziel dieser Phase ist, die Anwendung aus den bestehenden Komponenten

sowie der neuen Komponente `Wireframe` zusammenzusetzen.

Um einen Überblick über die Struktur der Anwendung zu bekommen, sind in Abbildung 6.1 die nötigen Komponenten in der in Anhang A.1 beschriebenen Notation zur Anwendung zusammengefügt.

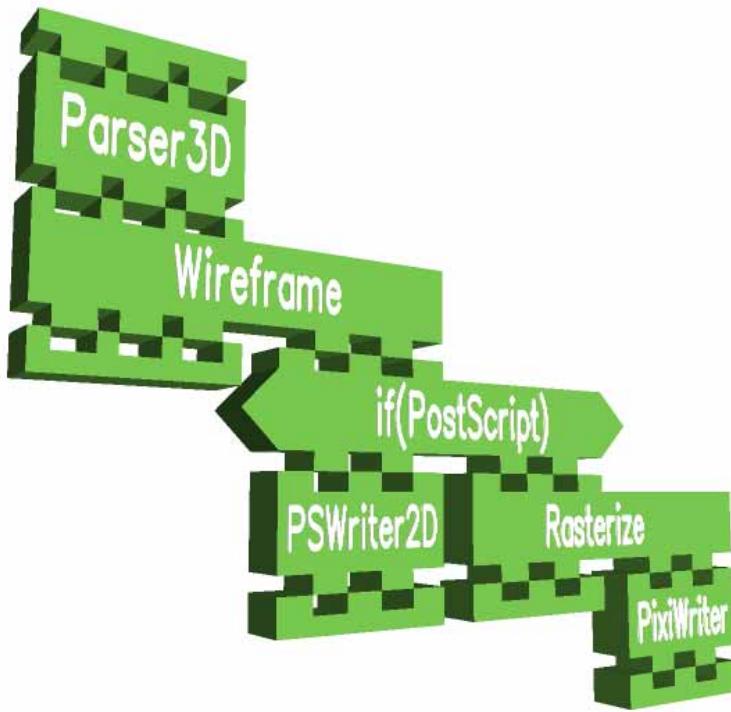


Abbildung 6.1

Die Komposition der Komponenten in der in Anhang A.1 vorgestellten Notation der `BOOGA`-LEGOs zu einer vollständigen Anwendung.

Die in Abbildung 6.1 visualisierte Komposition lässt sich direkt in C++-Code umsetzen. Der folgende Quellcode lässt sich grob in vier Gruppen gliedern (eine detaillierte Beschreibung folgt nach dem Quellcode):

Implementierung

Zeilen 3 bis 16 : Initialisierungsphase

Zeilen 17 bis 24 : Lesen der Szenenbeschreibung

Zeilen 25 bis 32 : Berechnen der Drahtgitterdarstellung

Zeilen 33 bis 50 : Ausgeben des Resultates im gewünschten Format

```

1 int main(int argc, char* argv[])
2 {
3     //
4     // Initialize everything ...
5     //
  
```

```

6   Configuration::setOption(Name("Report.ErrorStream"),
7                           Name("cerr"));
8
9   initBSDLParserGlobalNS();
10  initBSDLParser3DNS();
11
12  RCString in, out;
13  bool ps = false;
14
15  parseCmdLine(argc, argv, in, out, ps);
16
17  //
18  // Read scene
19  //
20  World3D* world3D = new World3D;
21  Parser3D parser;
22  parser.setFilename(in);
23  parser.execute(world3D);
24
25  //
26  // Process scene
27  //
28  Wireframe wireframe;
29  World2D* world2D = wireframe.execute(world3D);
30
31  delete world3D;
32
33  //
34  // Generate Postscript or Pixi, depending on flag
35  //
36  if (ps) {
37      PSWriter2D psWriter;
38      psWriter.execute(world2D);
39
40      delete world2D;
41  } else {
42      Rasterizer rasterizer;
43      World2D* rasterized = rasterizer.execute(world2D);
44
45      PixiWriter writer;
46      writer.execute(rasterized);
47
48      delete rasterized;
49      delete world2D;
50  }
51
52  return 0;
53 }
```

Zeilen 3 bis 16 : Initialisierungsphase

Auf Zeile 6 wird festgelegt, dass die Ausgabe von Fehlermeldungen, die durch die Klasse `Report` geschieht, auf den Standardfehlerkanal von UNIX erfolgen soll. Auf den Zeilen 9 und 10 wird der Parser mit den Befehlen konfiguriert, die er verstehen soll. Die Befehl-

le sind in *Namespaces* (vgl. Anhang A.2) angeordnet. Im vorliegenden Beispiel werden dem Parser einerseits der globale Namespace, andererseits der Namespace für dreidimensionale Szenen bereitgestellt.

Auf den Zeilen 12 und 13 werden schliesslich die Variablen definiert, die auf Zeile 15 gebraucht werden. Die Funktion `parseCmdLine` liest die Kommandozeile und interpretiert die der Anwendung übergebenen Parameter. Die Implementierung dieser Funktion folgt etwas später.

Zeilen 17 bis 24 : Lesen der Szenenbeschreibung

Zuerst wird eine neue, leere Welt (`World3D`) erzeugt (Zeile 20), anschliessend eine Instanz der Komponente `Parser3D`. Der Parser wird mit dem Namen der Eingabedatei konfiguriert, die zuvor von der Kommandozeile gelesen wurde. Schliesslich wird die Komponente ausgeführt (Zeile 23).

Zeilen 25 bis 32 : Berechnen der Drahtgitterdarstellung

Auf Zeile 28 wird eine Instanz der `Wireframe`-Komponente erzeugt und in Zeile 29 aufgerufen. Das Resultat dieser Komponente ist eine `World2D`. Schliesslich kann die `World3D` vernichtet werden, da sie im weiteren nicht mehr benötigt wird.

Zeilen 33 bis 50 : Ausgeben des Resultates im gewünschten Format

Je nach Option der Kommandozeile wird entweder eine Postscriptausgabe erzeugt (Zeilen 37 bis 40) oder ein PIXI-Bild (Zeilen 42 bis 49).

Die Funktion `parseCmdLine` erlaubt ein einfaches Interpretieren der Kommandozeile. Bisher wurde ins `BOOGA`-Framework noch keine generische Funktion zur Kommandozeileninterpretation eingebaut, da die meisten Anwendungen nur sehr wenige, einfache Optionen unterstützen. In der Regel kann die folgende Funktion als Vorlage für eigene Entwicklungen dienen.

```

1 void parseCmdLine(int argc, char* argv[],
2                   RCString& in, RCString& out, bool& ps)
3 {
4     if ((argc == 2 && !strcmp(argv[1], "-h")) || argc>5) {
5         usage(argv[0]);
6         exit(0);
7     }
8 }
```

```

9     int cur = 1;
10    if (!strcmp(argv[cur],"--ps")) {
11        ps = true;
12        cur++;
13    }
14
15    if (cur < argc)
16        in = argv[cur++];
17    if (cur < argc)
18        out = argv[cur];
19 }
```

Die `usage`-Funktion ist in allen **BOOGA**-Anwendungen enthalten und ermöglicht eine einfache Hilfestellung für den Benutzer. Bei Fehleingaben oder der Eingabe '*Programmname -h*' wird eine kurze Anleitung auf dem Bildschirm ausgegeben.

```

1 void usage(const RCString& name)
2 {
3     cerr << "Usage: " << name
4         << " [--ps] [in-file [out-file]]\n";
5     cerr << " where:\n";
6     cerr << " --ps           : (optional) "
7         << "write output as Postscript instead of Pixi\n";
8     cerr << " in-file       : (optional) "
9         << "filename of input\n";
10    cerr << " out-file      : (optional) "
11        << "filename of output\n";
12 }
```

Test

Die vorliegende Anwendung muss nun anhand genügend vieler unterschiedlicher Testszenen systematisch getestet werden. Dies wird hier nicht weiter ausgeführt, da sich dieser Teilschritt des **BOOGA**-Vorgehensmodells nicht wesentlich von anderen Methoden unterscheidet.



6.2.6 Iteration

Sobald die Anwendung eingesetzt wird, werden einerseits neue Anforderungen auftauchen, die zusätzlich erfüllt werden müssen, andererseits Fehler, die in der Testphase unentdeckt geblieben sind. Diese Wartungsaufgaben können durch teilweises, erneutes Durchlaufen des Prozess erfüllt werden. So könnte es, je nach Einsatzgebiet der Komponente, nötig werden, zusätzliche geometrische Objekte direkt zu unterstützen, um die Ausführungsgeschwindigkeit zu erhöhen. Dies kann einfach durch das Hinzufügen weiterer `visit`-Methoden geschehen.

6.2.7 Illustration

Abbildung 6.2 zeigt die Ausgabe einer Szene in Form einer Wireframedarstellung, berechnet durch die beschriebene Anwendung. Dieselbe Szene ist in Anhang A.2 als BSDL-Szenenbeschreibung aufgeführt und mittels einer Raytracing-Komponente berechnet in Abbildung A.1 wiedergegeben.

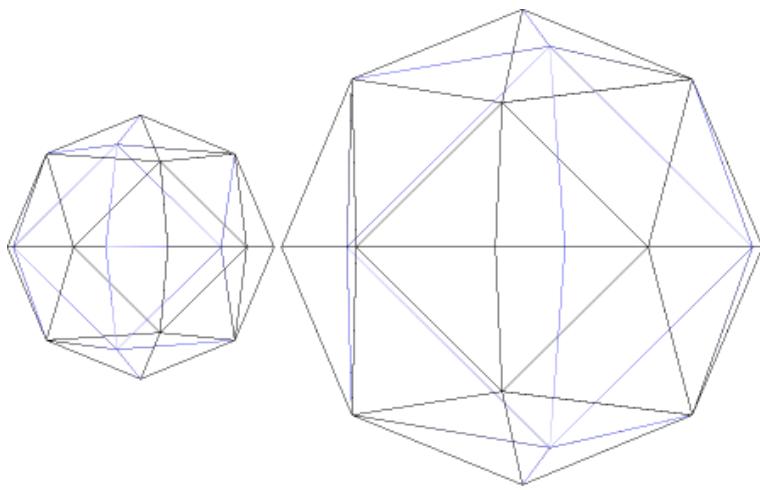


Abbildung 6.2

Die Wireframedarstellung einer einfachen Szene bestehend aus zwei unterschiedlich grossen Kugeln.

6.3 Anwendung “Radiosity”

Radiosity wurde ursprünglich in der Thermodynamik eingesetzt, um den Wärmeaustausch zwischen Objekten zu berechnen. Dasselbe Vorgehen kann aber angewendet werden, um den Austausch von sichtbarem Licht zu berechnen. So wird Radiosity heute vor allem auch zur Erzeugung photorealistischer Bilder eingesetzt. Einzelheiten zum Verfahren können beispielsweise [Amann, 1993] oder [Silion und Puech, 1994] entnommen werden.

Das Radiosityverfahren wurde noch *nicht* für **BOOGA** realisiert. An dieser Stelle wird aber skizziert, wie es sich nahtlos in **BOOGA** integrieren lässt. Gleichzeitig wird illustriert, dass es manchmal nicht genügt, nur neue Komponenten zu erstellen, sondern dass in gewissen Fällen auch die anderen Schichten des Komponentenframeworks erweitert werden müssen.

An dieser Stelle kann keine ausführliche Beschreibung des Verfahrens erfolgen. Die entsprechenden Grundlagen werden für das Verständnis dieses Abschnittes vorausgesetzt.

6.3.1 Problembeschreibung

Das Radiosityverfahren ist ein zweistufiges Bildberechnungsverfahren. In einem ersten Schritt werden die Helligkeitswerte in einzelnen Punkten eines Dreieck- oder Vierecknetzes (engl. *Mesh*) blickpunktunabhängig berechnet. Im zweiten Schritt wird mittels eines beliebigen Bildberechnungsverfahrens (beispielsweise z-Buffer oder Raytracing) ein Bild von einem bestimmten Kamerastandpunkt aus erzeugt.

Anforderungs-analyse

6.3.2 Anforderungsanalyse

Auch bei diesem Beispiel fliesst an dieser Stelle wieder einiges an Fachwissen in die Problemanalyse mit ein. Im Folgenden werden einige Anforderungen formuliert, die aufgrund der Aufgabenstellung alleine nicht einsichtig sind, sich aber aus den Eigenheiten des Radiosityverfahrens ergeben.

Wie im vorausgegangenen Beispiel muss natürlich auch hier zuerst die Szenenbeschreibung eingelesen werden:

Anforderung 1

Die Szenenbeschreibung muss im BSDL-Format eingelesen werden.

Auch die Ausgabe des Resultates erfolgt wiederum wie bereits besprochen:

Anforderung 2

Die Ausgabe des Resultates soll auf den Bildschirm oder im PIXI-Format erfolgen.

Die Voraussetzung für die Anwendung eines Radiosityverfahrens ist die Aufteilung der Körper in geeignete Drei- und Vierecke (sogenannte *Patches*). Diese Aufteilung muss angepasst an die Eigenheiten der Szene erfolgen.

Anforderung 3

Die geometrischen Objekte müssen in Patches zerlegt werden.

Nachdem die Zerlegung erfolgt ist, kann die Berechnung der eigentlichen Radiositywerte erfolgen. Für diese Berechnung gibt es eine Anzahl unterschiedlicher Verfahren. An dieser Stelle ist nicht weiter von Interesse, welches Verfahren angewendet wird, wichtig ist lediglich, dass nach dessen Beendigung die Helligkeitswerte für die nachfolgende Bildberechnung zur Verfügung stehen.

Anforderung 4

Die Helligkeitswerte für die Eckpunkte der Patches sollen gemäss einem geeigneten Radiosityverfahren berechnet werden.

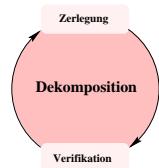
Eine besondere Schwierigkeit liegt darin, die bestehenden Bildberechnungskomponenten für den nachfolgenden Schritt verwenden zu können. Die Helligkeitswerte des Radiosityschrittes müssen für diese Komponenten völlig transparent zur Verfügung gestellt werden.

Anforderung 5

Die Bildberechnung muss mit den existierenden Bildberechnungskomponenten ohne Modifikationen durchführbar sein.

6.3.3 Dekomposition

In diesem Schritt geht es wiederum darum, die Anwendung in einzelne Komponenten zu zerlegen. Auch bei diesem Beispiel wird wieder der Umstand ausgenutzt, dass einige Teile der Aufgabe bereits als eigenständige UNIX-Anwendungen existieren. Die Darstellung auf dem Bildschirm kann also wieder aus der Aufgabenstellung ausgeklammert werden.



Zerlegung

Auch hier können die einzelnen Anforderungen wieder direkt in Komponenten umgesetzt werden. Auf den ersten Blick scheint es vielleicht nicht klar zu sein, wie die Probleme tatsächlich durch einzelne Komponenten gelöst werden können; dies wird in der Phase der Komponentenentwicklung verdeutlicht.

Anforderung 1: Einlesen der Szene

Wiederum findet die **Parser3D**-Komponente Verwendung, um eine Szenenbeschreibung im BSDL-Format einzulesen.

Anforderung 3: Erzeugung der Patches

Jedem geometrischen Objekt der Szene muss ein Mesh zugeordnet werden, das im folgenden Schritt als Grundlage zur Berechnung der Radiositywerte dient. Da es unterschiedliche Ansätze gibt, wie die Objekte in Patches zerlegt werden können¹⁰, werden mit der Zeit voraussichtlich mehrere solcher Komponenten realisiert, die dieses Problem auf verschiedene Arten lösen.

Anforderung 4: Berechnung der Radiositywerte

Auch für diesen Teilschritt gibt es eine ganze Anzahl unterschiedlicher Verfahren. Die im vorangegangenen Schritt erzeugten Meshes müssen auf das verwendete Radiosityverfahren abgestimmt werden.

Anforderung 5: Bildberechnung mit einer Standardkomponente

Für die Bildberechnung kann beispielsweise die **ZBuffer**- oder die **Raytracing**-Komponente eingesetzt werden.

Anforderung 2': Ausgabe im PIXI-Format

Die Ausgabe erfolgt mit der **PixiWriter**-Komponente.

Der nächste Teilschritt verifiziert die hier vorgeschlagene Aufteilung in Komponenten.

Verifikation

Gleich wie im vorausgegangenen Beispiel werden auch hier die einzelnen Komponentenkandidaten betrachtet und beurteilt.

Wie vorher wird die **Parser3D**-Komponente verwendet, um die Szenenbeschreibung einzulesen und den Szenengraphen, bestehend aus den Instanzen der unterschiedlichen Klassen von geometrischen Objekten, aufzubauen.

¹⁰Eine Diskussion der damit verbundenen Problemstellungen ist in [Amann, 1993] zu finden.

Parser3D

<i>Vorhanden</i>	Ja
<i>Atomar</i>	Ja
<i>Komplexität</i>	Gross
<i>Allgemeinheit</i>	Ja
<i>Typ</i>	<code>Operation3DTo3D</code>

Das nächste Teilproblem besteht darin, die Objekte in Patches zu zerlegen. Die Komponente `CreatePatches` existiert noch nicht. Zudem stellt sie neue Anforderungen an die Framework- und Bibliotheksschichten (vgl. Abschnitt 6.3.4.1).

CreatePatches

<i>Vorhanden</i>	Nein
<i>Atomar</i>	Ja
<i>Komplexität</i>	Gross ¹¹
<i>Allgemeinheit</i>	Bedingt ¹²
<i>Typ</i>	<code>Operation3DTo3D</code>

Die Komponente `CreatePatches` muss eng mit der ebenfalls neuen Komponente `Radiosity` zusammen entwickelt werden.

Radiosity

<i>Vorhanden</i>	Ja
<i>Atomar</i>	Ja
<i>Komplexität</i>	Gross
<i>Allgemeinheit</i>	Bedingt
<i>Typ</i>	<code>Operation3DTo3D</code>

Gemäss Anforderung 5 soll im zweiten Berechnungsschritt ein Standardverfahren ohne Modifikation zur Bildberechnung eingesetzt werden können.

ZBuffer, Raytracing

<i>Vorhanden</i>	Ja
<i>Atomar</i>	Ja
<i>Komplexität</i>	Mittel
<i>Allgemeinheit</i>	Ja
<i>Typ</i>	<code>Operation3DTo2D</code>

Das Resultat kann am Schluss mit der bereits besprochenen Komponente `PixiWriter` in eine Datei ausgegeben werden.

¹¹Wie später noch diskutiert wird, muss für diese Komponente die Frameworkschicht erweitert werden. Ausserdem ist je nach verwendetem Verfahren auch die Realisierung der eigentlichen Komponente sehr anspruchsvoll.

¹²Die Allgemeinheit wird dadurch etwas eingeschränkt, dass gewisse Radiosityverfahren auf spezielle Datenstrukturen der Patches angewiesen sind. Ein Beispiel dafür ist das *Hierarchische Radiosityverfahren* (vgl. [Silion und Puech, 1994]).

PixiWriter

Vorhanden	Ja
Atomar	Ja
Komplexität	Gering
Allgemeinheit	Ja
Typ	Operation2DTo2D

Abschliessend betrachtet müssen also zwei neue Komponenten realisiert werden, die spezifisch für das Radiosityverfahren sind. Die restlichen Teilprobleme können durch existierende Komponenten abgedeckt werden. Durch die im folgenden skizzierte Lösung wird es möglich, Standardbildberechnungskomponenten einzusetzen sowie unterschiedliche Radiosity- und Patchgenerierungsverfahren einfach auszutesten und zu kombinieren.

Aufgrund der bisher aufgestellten Anforderungen ist es noch nicht möglich, sich für ein bestimmtes Radiosityverfahren zu entscheiden. Diese Entscheidung beeinflusst die Entwicklung der beiden neuen Komponenten und muss spätestens an dieser Stelle getroffen werden. In der folgenden Besprechung wird jedoch nicht weiter auf die Algorithmik und die realisierten Verfahren eingegangen.



6.3.4 Komponentenentwicklung

Für alle **BOOGA**-Anwendungen muss ein Ziel darin bestehen, nicht nur die bestehenden Komponenten zu verwenden, sondern auch die in den verschiedenen Schichten des Komponentenframeworks implementierten Mechanismen so weit wie möglich auszunutzen. So ist in **BOOGA** ein sehr mächtiges Texturkonzept realisiert worden (vgl. [Streit, 1997]). Radiosity wird bei der hier skizzierten Lösung als *Textur* auf dem Körper betrachtet, die in einem ersten Schritt initialisiert wird (**CreatePatches**). Anschliessend werden in einem zweiten Schritt die Helligkeitswerte in den Eckpunkten der Meshes berechnet (**Radiosity**). Für einen Bildberechnungsalgorithmus präsentiert sich diese Radiositytextur wie eine ‘normale’ Textur.

In diesem Beispiel werden also zwei neue Komponenten benötigt, welche im folgenden getrennt entworfen werden. Die wesentlichen Frameworkerweiterungen werden in der folgenden Besprechung ebenfalls erwähnt.

6.3.4.1 CreatePatches

Analyse

Viele der geometrischen **BOOGA**-Objekte haben gekrümmte Oberflächen. Für das Radiosityverfahren können aber nur Netze aus planaren Patches eingesetzt werden. Diese Netze weisen aber beim zweiten Schritt, der eigentlichen Berechnung des Bildes, den Nachteil auf, dass

ihre approximierte Form die Konturen der geometrischen Objekte im Bild verfälscht. In [Amann, 1993] wird ein *heteromorpher*¹³ Ansatz eingeführt, der analytische Körper mit Patches kombiniert, um so die Vorteile der beiden Ansätze zu vereinen. Dies kam der hier verwendeten Idee der *Radiositytextur* bereits sehr nahe. Abbildung 6.3 visualisiert diese Idee. Für die Bildberechnung wird die analytische Beschreibung des Körpers ver-

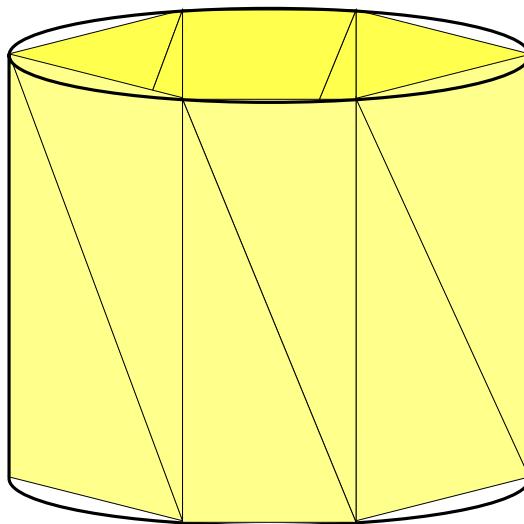


Abbildung 6.3

Der in [Amann, 1993] vorgestellte heteromorphe Ansatz kombiniert eine triangularisierte und die analytische Sicht eines Körpers um die unterschiedlichen Bedürfnisse des Radiosity und des anschliessenden Bildberechnungsschrittes optimal zu kombinieren.

wendet, für die Berechnung der Helligkeitswerte das approximierte Mesh. In der hier vorgestellten Lösung wird dieses Mesh als spezielle Textur realisiert.

Aufgrund des bisher Besprochenen kristallisieren sich die folgenden neuen Anforderungen an die Frameworkschicht von **BOOGA** heraus:

Anforderung 6

Es muss eine Klasse realisiert werden, die ein Mesh, bestehend aus Dreiecken oder Vierecken (je nach implementiertem Verfahren) speichern und verwalten kann.

Anforderung 7

Es muss eine spezielle Texturklasse implementiert werden, welche ein Mesh enthält.

Verschiedene Radiosityverfahren stellen unterschiedliche Anforderungen an die Funktionalität einer Meshstruktur. Demzufolge werden unterschiedliche Meshklassen benötigt werden.

¹³Jeder Körper wird sowohl analytische beschrieben als auch als Mesh approximiert. Die beiden Darstellungsformen werden so kombiniert, dass für jede Teilaufgabe die passende Repräsentation verwendet wird. Ein Körper wird somit durch unterschiedliche ‘Gestalten’ (*heteromorph: verschiedengestaltig*) gleichzeitig beschrieben.

Anforderung 8

Dieselbe Texturklasse sollte mit unterschiedlichen Typen von Meshes verwendet werden können.

Design

Die neuen Meshklassen können unterschiedlich entworfen werden. Eine Variante wäre, diese als spezielle, geometrische Objekte zu realisieren, eventuell sogar als neuen Aggregattyp (vgl. [Streit, 1997]).

Für die Implementierung der Patches könnte auf die bereits existierenden Klassen `Triangle3D` oder `Polygon3D` zurückgegriffen werden. Aufgrund der spezifischen Bedürfnisse¹⁴ müssten aber spezielle Methoden zur Verfügung gestellt werden, die ausschliesslich für die Implementierung von Radiosityverfahren benötigt würden. Dies widerspricht dem Gedanken der Entwicklung wiederverwendbarer Klassen, da somit die Schnittstellen überladen und dadurch die Wartung sowie das Verständnis des Systems erschwert wird. Daher ist es für das vorliegende Problem besser, die Meshklassen als eine eigenständige Klassenhierarchie zu realisieren. Dies umso mehr, als sich im Moment kein wesentlicher Vorteil aus der zuerst skizzierten Variante abzeichnet.

Die neue Komponente `CreatePatches` muss jedem geometrischen Objekt eine neue Instanz der Texturklasse `RadiosityTexture` zuordnen. Dieser Textur wird ein Mesh zugeordnet, das aufgrund des Typs des geometrischen Objektes erzeugt wurde.

Implementierung

Dieser Teilschritt wird hier nicht weiter betrachtet.

Test

Da das korrekte Erzeugen der Meshes nicht einfach, aber zentrale Voraussetzung für die folgenden Schritte ist, wird es bei diesem Beispiel nötig sein, eine spezielle Testkomponente zu schreiben, welche die Meshes, eventuell zusammen mit den analytischen Repräsentationen, visualisiert. Dies führt zu einer neuen Anforderung, welche in einer weiteren Komponente mündet. Diese Komponente wird an dieser Stelle allerdings nicht genauer betrachtet.

Anforderung 9

Zur Kontrolle der Korrektheit von `CreatePatches` müssen Bilder der erzeugten Meshes berechnet werden.

Zur Realisierung dieser Komponente kann die im vorderen Beispiel betrachtete Komponente `Wireframe` als Basis herangezogen werden.

¹⁴So müssen in den Eckpunkten sowie den Patches zusätzliche Radiositywerte und Zwischenresultate abgelegt werden können.

6.3.4.2 Radiosity

Die Komponente **Radiosity** muss die Energieverteilung innerhalb der Szene bestimmen. Abhängig vom gewählten Verfahren müssen an dieser Stelle allfällige weitere Anforderungen an die Frameworkschicht formuliert werden.

Bei der Realisierung der Komponente **Radiosity** kann ein weiterer Mechanismus des Komponentenprotokolles verwendet werden. Zuerst wird beim Durchlaufen des Szenengraphs – mittels der bereits besprochenen **visit**-Methode – jedes Objekt einmal ‘besucht’. Dabei wird eine Referenz auf das Mesh für die folgende Verarbeitung in einer geeigneten Datenstruktur der **Radiosity**-Komponente zwischengespeichert. Nachdem der ganze Graph durchlaufen wurde, kann mit Hilfe der **postProcessing**-Methode das eigentliche Radiosityverfahren gestartet werden. Mit den mittels **CreatePatches** erzeugten und durch die **visit**-Methode der **Radiosity**-Komponente gesammelten Meshes wird der Energieaustausch berechnet, und die resultierenden Energiewerte werden in den Meshes gespeichert.

Nach Beendigung dieser Komponente muss die Bildberechnungskomponente unter Verwendung der bereits erwähnten **RadiosityTexture** für einen bestimmten Oberflächenpunkt den Farbwert bestimmen können. Dazu greift die **RadiosityTexture** auf die zuvor berechneten Energiewerte in den Meshpunkten zu. Daraus ergibt sich eine weitere Anforderung an die neue Textur:

Anforderung 10

Die Radiositytextur muss aufgrund des Meshes einen Farbwert an einer bestimmten Stelle auf der Oberfläche des Körpers berechnen können.

Auch hier wird dieser Teilschritte nicht genauer betrachtet.

Analyse

Design

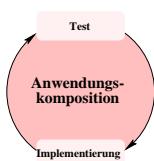
Implementierung

Bei dieser Komponente kann auf die Implementierung einer spezielle Testkomponente verzichtet werden, da – korrekte Funktion von **CreatePatches** vorausgesetzt – sie nun die einzige ungetestete Komponente ist und die eigentliche Anwendung gleichzeitig die Testumgebung darstellt.

6.3.5 Anwendungskomposition

Die Anwendung selbst ist auch in diesem Beispiel wieder sehr einfach. Abbildung 6.4 vermittelt einen Eindruck über ihren Aufbau.

Test



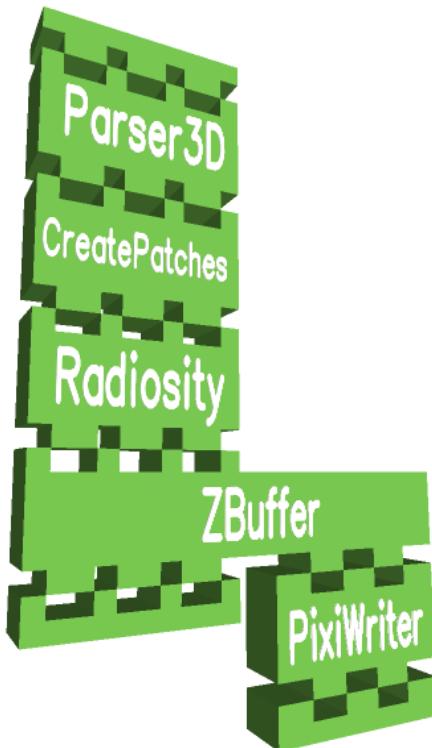


Abbildung 6.4

Die Komposition der Komponenten in der in Anhang A.1 vorgestellten Notation der **BOOGA**-LEGOS zur Radiosity Anwendung.

Implementierung In diesem Beispiel wird die Implementierung der Anwendung selbst nicht mehr angegeben. Sie ist aber direkt aus Abbildung 6.4 ableitbar.

Test Auch hier müssen wieder viele unterschiedliche Szenen verwendet werden, um die neuen Komponenten und die Anwendung zu testen.



6.3.6 Iteration

In den Iterationsphasen können nun Erweiterungen und Verbesserungen an den implementierten Komponenten vorgenommen werden. Weitere Mesherzeugungs- und Radiosityverfahren können analog implementiert werden und erlauben so einen raschen und einfachen Vergleich der unterschiedlichen Methoden.

6.4 Anwendung “Reverse Engineering”

In diesem Beispiel wird eine Problemstellung betrachtet, die auf den ersten Blick überhaupt nichts mit Computergrafik zu tun hat. Die folgenden Ausführungen werden sehr knapp gehalten. Detaillierte Angaben zum Entwurf und zur Realisierung dieser Anwendung können in [von Siebenthal und Wenger, 1996] nachgelesen werden.

6.4.1 Problembeschreibung

Die Anwendung CASETOOL soll C++-Quellcode einlesen und daraus Klassendeklarationen, Vererbungsbeziehungen und Aggregationsbeziehungen extrahieren. Diese Daten sollen anschliessend in einer gebräuchlichen Notation visualisiert werden.

6.4.2 Anforderungsanalyse

Anforderungs-analyse

Entgegen den bisherigen Beispielen steht hier nicht das Einlesen einer Szenenbeschreibung am Anfang, sondern die Verarbeitung des zu visualisierenden C++-Quellcodes.

Anforderung 1

Der C++-Quellcode muss eingelesen werden.

Für die folgende Aufgabe wird nicht die gesamte Information dieses Quellcodes benötigt. Zur Erstellung von Klassendiagrammen genügt es, nur die *Klassendefinitionen* zu betrachten.

Anforderung 2

Die gefundene Information muss in geeigneter Form abgelegt werden.

Diese Anforderung hat wiederum eine Erweiterung der Frameworkschicht von **BOOGA** zur Folge. Liegen die zur Erstellung eines Klassendiagrammes nötigen Angaben einmal in Form einer Szenenbeschreibung vor, muss ein Diagramm erstellt werden.

Anforderung 3

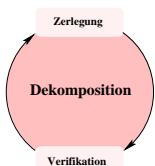
Die fertig aufgebaute Datenstruktur muss in Form eines Diagrammes ausgegeben werden.

Da die verschiedenen Methodiker unterschiedliche Notationen zur Beschreibung der Entwurfsresultate eingeführt haben, wäre es wünschenswert, wenn die Ausgabe des Diagrammes nicht fest an eine bestimmte Notation gebunden wäre.

Anforderung 4

Unterschiedliche Notationen sollen unterstützt werden.

Die Ausgabe der Resultate sollte natürlich wieder auf möglichst viele Arten möglich sein.



6.4.3 Dekomposition

Die Dekompositionsphase wird bei diesem Beispiel nicht mehr so ausführlich wie bisher besprochen. Abbildung 6.5 zeigt die in [von Siebenthal und Wenger, 1996] detailliert beschriebene Zerlegung. Die schwarzen Pfeile bezeichnen gelesene oder geschriebene Dateien als Seiteneffekte bestimmter Komponenten, die grauen Pfeile den Ablauf sowie mögliche Alternativen. Die Sterne kennzeichnen die im folgenden genauer beschriebenen, neu zu erstellende Komponenten.

CTParser

Diese Komponente liest C++-Quellcodedateien ein und erstellt daraus einen Szenengraphen, bestehend aus Instanzen eines neuen Typs von geometrischen Objekten. Diese Objekte beinhalten alle nötigen Informationen für die folgenden Schritte, allerdings noch ohne Zuordnung einer grafischen Repräsentation.

UMLRenderere

Der UMLRenderere stellt die Klassen in der *Unified Modelling Language* dar (vgl. [Booch et al., 1997]). Dazu wird der Szenengraph durchlaufen und bei jeder zuvor erzeugten Instanz des neuen Typs eine grafische Repräsentation angefügt. Die nachfolgend angewendeten Komponenten haben auf diese methodentypische Repräsentation dann mittels der Verfeinerungsmethode (*decompose*) Zugriff.

CPPRenderere

Die Komponente CPPRenderere ist eine spezielle Komponente, welche als Seiteneffekt wiederum eine C++-Quellcodedatei erzeugt, in der aktuellen Version von CASETOOL allerdings nur die Klassendefinitionen.

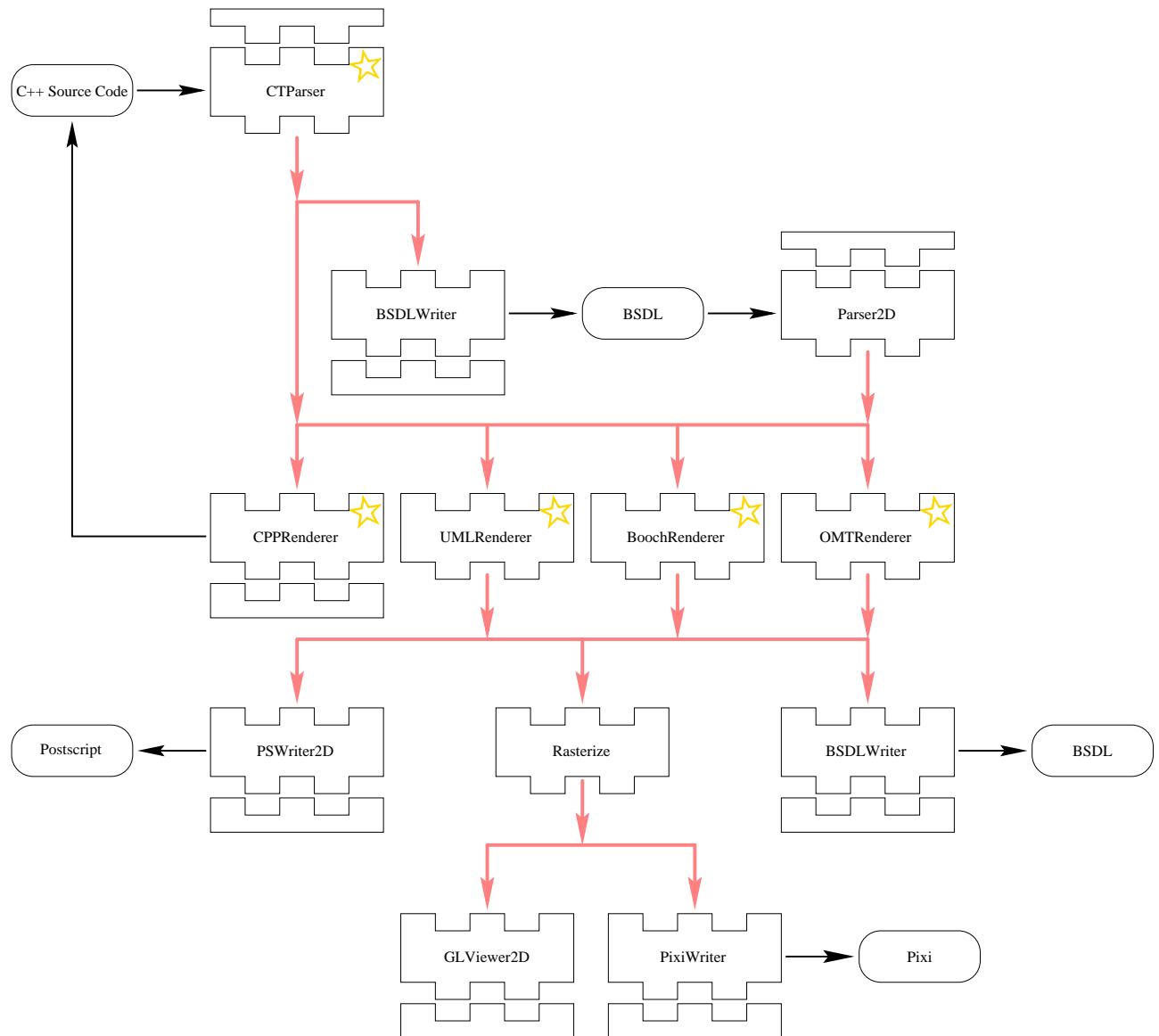


Abbildung 6.5

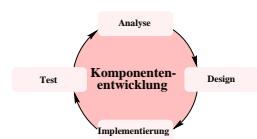
Die weiteren Komponenten **BoochRenderer** und **OMTRenderer** sind vorgesehene Erweiterungen, ebenso wie allfällig weitere, methodenspezifische Renderer.

Mit den vorhandenen und neu erstellten Komponenten lassen sich viele unterschiedliche Anwendungsfälle lösen.

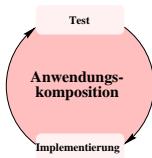
Alle in diesem Beispiel benötigten Komponenten sind vom Typ **Operation2DTo2D**, da ausschliesslich Informationen des zweidimensionalen Raumes verarbeitet werden.

6.4.4 Komponentenentwicklung

In [von Siebenthal und Wenger, 1996] werden die neuen Komponenten so-



wie die nötigen Frameworkerweiterungen ausführlich dokumentiert und sollen an dieser Stelle nicht wiederholt werden.



6.4.5 Anwendungskomposition

In Abbildung 6.6 wird eine mögliche Komposition (also ein Weg durch die in Abbildung 6.5 aufgezeigten Varianten) wie üblich in der **BOOGA**-LEGO-Notation dargestellt.



Abbildung 6.6

Eine mögliche *Reverse Engineering*-Anwendung in der in Anhang A.1 vorgestellten Notation der **BOOGA**-LEGOS.

Die in diesem Kapitel aufgeführten Beispiele haben zum einen das im vorausgegangenen Kapitel eingeführte Vorgehensmodell illustriert, zum anderen einen Einblick in die Vielfalt der Anwendungsmöglichkeiten von **BOOGA** gegeben. Weitere **BOOGA**-Anwendungen sind in [Streit, 1997] und in den diversen Arbeiten von Studenten [Ammon, 1996; Bächler, 1995; Balmer, 1996; Habegger, 1996; Liechti, 1996; Matthey, 1996; Matthey, 1997; Matthey, 1997;

Sagara, 1996; von Siebenthal und Wenger, 1996; Teuscher, 1996] zu finden.

In den bisherigen Ausführungen wurde des öfteren auf Schwierigkeiten und Probleme im Zusammenhang mit der Dokumentation von Frameworks und Komponentensystemen im allgemeinen und **BOOGA** im speziellen hingewiesen. Das nun folgende Kapitel behandelt die damit verbundenen Fragestellungen.



Jan Gossaert, genannt Mabuse.
Der Sündenfall.

Kapitel 7

Dokumentationskonzepte für wiederverwendungsorientierte Systeme

*And finally, I cannot tell you all the things
whereby we can commit sin;*

*for there are divers ways and means, even
so many that I cannot number them.*

Mosiah 4:29,
The Book of Mormon

7.1 Einleitung

Fragen der Wiederverwendung spielen eine zentrale Rolle in der vorliegenden Arbeit. Das Augenmerk wurde bisher auf die technischen Aspekte gerichtet: die Form und Organisation der wiederverwendbaren Elemente und die Architekturen, die deren Zusammenarbeit definieren. Einige der nichttechnischen Problemstellungen wurden ebenfalls kurz erwähnt.

Eine sehr wichtige Voraussetzung für die Wiederverwendung ist bisher allerdings unerwähnt geblieben:

“Binsenwahrheit” 5

Software muss verstanden werden, um sie zu gebrauchen.

Eine gute Dokumentation ist ein wesentliches Hilfsmittel, um die Wiederverwendbarkeit von Software zu erhöhen; schliesslich nützt die beste Bibliothek, das mächtigste Framework nur wenigen, wenn die zur Verwendung nötige Information nicht zugänglich ist.

Als Dokumentation einer Bibliothek genügt es oft, ein Handbuch - sei es in traditioneller Papierform oder elektronisch - mit der Auflistung und Beschreibung der vorhandenen Funktionen anzulegen. Diese Wiederverwendungsform zeichnet sich dadurch aus, dass die wiederverwendeten Elemente oft nur lose miteinander verknüpft sind und bei einer allfälligen Erweiterung der Bibliothek die Beschreibung einfach ergänzt werden kann.

Ein Komponentenframework beinhaltet dagegen, speziell in der Frameworkschicht, stark miteinander verknüpfte Klassen und Klassenhierarchien. Klassen, die dem Framework hinzugefügt werden, in manchen Phasen der Frameworkentwicklung aber auch Änderungen an bestehenden Klassen, führen dazu, dass die herkömmliche, statische Dokumentation allenfalls geeignet ist, die grundlegenden Mechanismen und Protokolle festzuhalten, nicht aber die wiederverwendbaren Elemente zu beschreiben.

Wie den beiden vorausgegangenen Kapiteln entnommen werden konnte, beschreibt die bei der Arbeit mit *BOOGA* anzuwendende Methode ein stark iteratives Vorgehen. Dies bringt Vorteile vor allem hinsichtlich unvollständiger oder sich ändernder Anforderungen. Gleichzeitig wird aber die Aufgabe, Quellcode und Dokumentation konsistent zu halten, wesentlich aufwendiger. Wünschenswert wären hier Hilfsmittel, die es erlauben, die Redundanz zwischen Code und Dokumentation zu verringern. Abschnitt 7.2.4 wird kurz einige Möglichkeiten hierzu aufzeigen.

Beim Schreiben von Dokumentation stellt sich grundsätzlich immer die Frage nach dem Zielpublikum. Im Falle eines Komponentenframeworks kommen hierfür in Frage:

- Der *Benutzer des Komponentenframeworks* möchte gerne eine einfache verständliche Anleitung, wie er mit Hilfe des Komponentenframeworks neue Anwendungen erstellen kann.

Die Benutzerdokumentation muss auch auf die Erweiterungsmöglichkeiten des Frameworks eingehen, kann allerdings für die Details auf die Dokumentation für die Frameworkentwickler verweisen.

- Der *Frameworkentwickler*, hat die Aufgabe, das Framework zu pflegen und weiterzuentwickeln. Er braucht ein viel tieferes Verständnis der Protokolle und Mechanismen als der Anwendungsentwickler. Für ihn ist es auch hilfreich, die Gründe für eine bestimmte Implementierungsvariante zu erfahren. Kennt er diese, so kann er zusätzliche Erweiterungen im “Sinn und Geist” der ursprünglichen Architekten des Frameworks vornehmen und somit die Konsistenz des gesamten Systems gewährleisten.
- *Entwickler, Projektleiter oder Vorgesetzte* können daran interessiert sein, die Geschichte eines Systems nachzulesen, um herauszufinden, warum ein Projekt erfolgreich war oder nicht.

Ausser den Personen der letzten Kategorie ist niemand an den leidvollen Erfahrungen und Irrwegen interessiert, die speziell Frameworkentwickler in Kauf zu nehmen haben. [Parnas und Clements, 1986] verlangten denn auch von einer guten Dokumentation, dass sie ein Top-Down Design vortäuschen soll, da dies der am einfachsten zu verstehende Weg ist. Als Analogie wird auf das Vorgehen von Mathematikern hingewiesen, die selten den zuerst gefundenen Beweis eines Satzes veröffentlichen, sondern eine verbesserte, von allen Umwegen befreite und elegantere Version.

Neben dem Zielpublikum lässt sich Dokumentation auch nach dem Einsatzzweck klassifizieren. Die folgende Auflistung ist auf die Beschreibung von Frameworks ausgerichtet und entstammt teilweise [Johnson, 1996].

- Das *Einsatzgebiet* eines Frameworks (*wofür kann es verwendet werden?*) kann durch exemplarische Anwendungen sehr gut dokumentiert werden (vgl. Abschnitt 7.2.2).
- Die *Anwendung* eines Frameworks (*wie wird es verwendet?*) kann durch eine patternbasierte Dokumentation (vgl. Abschnitt 7.2.1) oder wiederum durch Beispiele beschrieben werden.
- Das Wissen, um *Erweiterungen* eines Frameworks (*wie wurde es implementiert?*) vorzunehmen, wird durch Spezifikationen, Design Patterns und ebenfalls durch Beispiele weitergegeben.

- Die *Geschichte* eines Frameworks (*wie ist es entstanden?*), welche es erlaubt, festzustellen, *warum* gewisse Dinge so und nicht anders realisiert wurden, wird durch das Führen eines *Logbuches* ermöglicht, in das alle wichtigen Entscheide eingetragen werden.

Die folgende Tabelle 7.1 zeigt auf, welches Zielpublikum an welchen Dokumenten interessiert sein wird.

Benutzer	Dokumentationszweck			
	Einsatzgebiet	Anwendung	Erweiterung	Geschichte
Anwendungsentwickler	×	×		
Frameworkentwickler	×	(×)	×	(×)
Vorgesetzte	×			×

Tabelle 7.1

Dokumentationszweck und Zielpublikum: Für jeden Typ von Dokumentation gilt es die geeignete Form zu finden.

× bedeutet von grossem, (×) von bedingtem Interesse für die jeweilige Person.

Im Folgenden sollen einige Dokumentationskonzepte diskutiert werden, die über das reine Auflisten von isolierten Elementen hinausgehen. Schliesslich wird das für die Dokumentation von *BOOGA* vorgesehene Konzept vorgestellt (Abschnitt 7.3). Wie in Kapitel 8.2 noch diskutiert wird, wurde bei der Entwicklung von *BOOGA* dem Dokumentationskonzept zu wenig Rechnung getragen. Da dieses erst recht spät aufgestellt wurde, stellt zum gegenwärtigen Standpunkt die Dokumentation eindeutig die Achillesferse von *BOOGA* dar.

7.2 Alternative Dokumentationskonzepte

Dieser Abschnitt soll als kurzer Überblick und ohne Anspruch auf Vollständigkeit einige Ideen hinsichtlich der Dokumentation von wiederverwendbaren Softwaresystemen aufzeigen. Einige der dargelegten Ideen flossen in das in Abschnitt 7.3 vorgestellte Dokumentationskonzept von **BOOGA** ein. Die im Folgenden besprochenen Konzepte sind als Alternative oder Ergänzung zu den traditionellen Handbüchern zu verstehen.

7.2.1 Dokumentation basierend auf Patterns

In Abschnitt 2.4.1 wurden Design Patterns als Mittel beschrieben, Expertenwissen und verallgemeinerte Lösungen zu wiederkehrenden Problemen im Bereich Softwaredesign festzuhalten.

[Johnson, 1992] zeigt auf, wie die vielfältigen Anforderungen an die Dokumentation von Frameworks erfüllt werden können, indem diese als eine Mustersprache¹ (*Pattern Language*) aufgebaut wird. Die Funktionsweise eines Frameworks wird mit *Design Pattern* beschrieben. Mit einer Dokumentation in Form einer Mustersprache wird dagegen festgehalten, wie ein Framework *verwendet* werden kann. Im ersten Fall ist das hauptsächliche Zielpublikum der Frameworkentwickler, im zweiten Fall der Anwendungsentwickler.

Auch in dem hier diskutierten Zusammenhang beschreibt ein Pattern die Lösung eines immer wiederkehrenden Problems, hier bei der Verwendung des zu dokumentierenden Frameworks. Die meisten Patterns werden voraussetzen, dass der Leser mit anderen Patterns der Mustersprache vertraut ist. Die einzelnen Muster sind untereinander durch Querverweise verknüpft.

Ein Anwendungsentwickler wird in der Regel immer nur gerade so viel vom verwendeten Framework wissen wollen, wie nötig ist, um sein Problem zu lösen. Diesem Wunsch kann mit einer Mustersprache ideal nachgekommen werden.

Das Beispiel in Anhang C zeigt, wie *ein* Pattern einer solchen Mustersprache für das **BOOGA**-Komponentenframework aussehen könnte. In der Beschreibung kommen Verweise auf andere Muster vor, wie dies bei einer Mustersprache notwendig ist. Da es an dieser Stelle nicht möglich ist, eine vollständige Mustersprache zu entwickeln, können diese Verweise nicht aufgelöst werden. Weiter würden die einzelnen Patterns zudem

¹Eine Mustersprache beinhaltet eine Menge vernetzter Muster zu einem bestimmten Bereich. Die einzelnen Muster zeigen wiederum in Form von Problem-Lösungspaaren bewährte Lösungen zu häufigen Problemen auf.

mit Nummern zur einfacheren Identifikation versehen; dies wird beim skizzierten Beispiel ebenfalls weggelassen.

7.2.2 Dokumentation durch Beispielanwendungen

Eine immer noch oft vertretene Meinung manifestiert sich in der Aussage:

Quellcode ist die beste Dokumentation.

In dieser generellen Form kann diese Aussage sicherlich nicht akzeptiert werden, da es gerade bei grossen Systemen unmöglich ist, nur über den Quellcode effizient auf die Funktionalität zu schliessen. Auch die Fehler-suche wird erschwert, da nur ersichtlich ist, was der Entwickler gemacht hat, nicht was er machen *wollte*. Trotzdem kann jeder, der diese Mei-nung vertritt, auch einen Pluspunkt für sich verbuchen: es ist ein oft sehr schwieriges und unter Zeitdruck leider nicht selten vernachlässigtes Unterfangen, Dokumentation und Quellcode gleichzeitig zu warten und konsistent zu halten. Somit ist der Quellcode tatsächlich die einzige zu-verlässige Quelle um in Zweifelsfällen abschliessend über strittige Punkte zu entscheiden.

Wie in der Einleitung zu diesem Kapitel besprochen wurde, gibt es ver-schieden Zielgruppen und Zwecke von Dokumentation. Quellcode kann nicht jedem Leser zugemutet werden und ist nicht für jeden Zweck ge-eignet. So ist es beispielsweise nahezu unmöglich, sich ausschliesslich an-hand des Quellcodes in **BOOGA** einzuarbeiten. Andererseits können aber überschaubare *Beispielanwendungen* eine gute Möglichkeit sein, um die Funktionsweise von einzelnen Mechanismen zu demonstrieren. Diese Bei-spielanwendungen können von einem neuen Anwender eines Frameworks auch gleichzeitig als Vorlage für seine eigene Anwendung eingesetzt wer-den.

Die Beispiele sollten von unterschiedlichem Schwierigkeitsgrad sein. Die ersten Beispiele, mit denen der Anwender in Berührung kommt, erläutern die typischen Anwendungsgebiete des Frameworks, die folgenden Beispiele führen dann schrittweise in komplexere Gebiete ein. Die Beispielanwen-dungen sollen dabei nicht durch einen möglichst eleganten Implementie-rungsstil bestechen, oder das Framework bis zu den letzten Möglichkeiten ausreizen, sondern jeweils einzelne Aspekte illustrieren und die gestellten Aufgaben mit möglichst einfachen Mitteln realisieren.

In Kapitel 6 wurden einige Beispiele für **BOOGA** besprochen. Jede mit **BOOGA** geschriebene Anwendung kann gleichzeitig wieder als ein Bei-spiel für anderen Anwender dienen. Dies wird auch durch die in Abschnitt 5.5 besprochene Organisationsform zusätzlich vereinfacht.

7.2.3 Ablauorientierte Dokumentation

Objektorientierte Systeme verhalten sich oft sehr dynamisch. Es werden Objekte erzeugt und vernichtet, Nachrichten fliessen hin und her und Referenzen auf Objekte werden verändert. All dies ist sehr aufwendig mit traditionellen Mitteln im Detail festzuhalten.

Alternativ zu den herkömmlich verwendeten statischen Diagrammen würden sich hier multimediale Mittel hervorragend eignen, um Entwickler mit den Abläufen innerhalb des Frameworks vertraut zu machen.

‘LOOK!’² ist ein Produkt, das einige Ansätze in dieser Richtung realisiert.

7.2.4 Automatisierbare Dokumentation

Die bereits im Abschnitt 7.2.2 erwähnten Schwierigkeiten, Dokumentation und Quellcode ständig konsistent zu halten, haben noch zu anderen Lösungsansätzen geführt. Deren Grundidee ist, Quellcode und Dokumentation nicht zu trennen, sondern zusammenzuhalten. Dadurch müssen Änderungen idealerweise nur noch an einem Ort, sicherlich aber nur noch in einer Datei vorgenommen werden.

Dabei gibt es grundsätzlich zwei unterschiedliche Verfahren. Das häufiger angewandte Verfahren (`C++2MAN`³, `JAVADOC`⁴) basiert auf der Idee von strukturiertem Kommentar. In der Syntax der verwendeten Programmiersprache wird die Dokumentation als normaler Kommentar, der Steuerbefehle enthalten kann, eingeflochten. Der Quelltext kann mit dem Compiler normal übersetzt werden, die Dokumentation wird mit einem speziellen Hilfsprogramm extrahiert und formatiert, bzw. zur Formatierung durch einen Textformatierer (`LATEX`, `TEX`, `TROFF`) vorbereitet.

Ein anderer Ansatz wurde von Donald E. Knuth 1984 vorgeschlagen [Knuth, 1984; Lindner, 1990]: das `WEB`⁵-System, welches das Gebiet des *Literate Programming* begründete. Quellcode und Dokumentation werden auch hier in einer Datei abgelegt. Mit Hilfe von zwei verschiedenen Programmen wird aus dieser Mischung der Quellcode und die Dokumentation extrahiert. Der Quellcode kann anschliessend einem Compiler übergeben werden, die Dokumentation auch hier wieder einem Textformatierer.

²LOOK! ist ein Produkt der Firma Objective Software Technology.

³`C++2MAN` ist ein frei erhältliches Programm, das aus `C++ Header`-Dateien strukturierten Kommentar extrahiert und in Form von `UNIX-man Pages` formatiert.

⁴`JAVADOC` ist ein zusammen mit `JAVA` definiertes Verfahren zur Dokumentation von `JAVA`-Applets.

⁵Knuth’s `WEB` ist nicht zu verwechseln mit dem World Wide Web.

7.3 Ein Dokumentationsansatz für *BOOGA*

In den vorausgegangenen Abschnitten wurden einige alternative Varianten zur Dokumentation besprochen. Bei dem im folgenden skizzierten Ansatz handelt es sich um eine *Idee* zur Dokumentation von *BOOGA*. Aus den verschiedenen besprochenen Ansätzen sollen die zur Dokumentation eines Komponentenframeworks geeigneten Elemente extrahiert und zu einem einheitlichen Konzept kombiniert werden.

Im Folgenden werden zuerst einige Anforderungen an die Dokumentation eines Komponentenframeworks formuliert. Anschliessend werden die wichtigsten Ideen zusammengefasst und in einem Prototyp präsentiert.

7.3.1 Die Anforderungen

Anforderung 1

Die Dokumentation muss einfach erweiterbar sein.

Wie bereits diskutiert wurde, muss die Beschreibung mit dem zugehörigen Komponentenframework mitwachsen können. Eine neue Anwendung erfordert in der Regel neue Komponenten, oft auch neue Abstraktionen in der Frameworkschicht.

Anforderung 2

Jeder Entwickler muss die Dokumentation erweitern können.

Alle an der Anwendungs- und Frameworkentwicklung Beteiligten müssen auf die Dokumentation neu verfügbarer Komponenten zugreifen können, um zeitraubende Missverständnisse aufgrund unterschiedlicher Versionen von Dokumentation und System zu vermeiden.

Anforderung 3

Jeder muss stets auf die aktuelle Version der Dokumentation zugreifen.

Eine Dokumentation, die laufend erweitert wird verursacht beträchtliche organisatorische Umtriebe, wenn ein traditioneller Ansatz gewählt wird.

Anforderung 4

Informationen sollen nicht redundant in mehreren Dokumenten vorkommen.

Wird Information redundant gehalten, so ergibt sich wiederum das Problem, diese bei Änderungen konsistent zu halten.

Anforderung 5

Der Zugriff auf eine einzelne Information muss auf vielen unterschiedlichen Wegen möglich sein.

Oft ist es nicht einfach, aus einer grossen Menge von Information die richtige herauszufinden. Dies wird noch schwieriger, wenn keine Redundanz erlaubt ist. Als Ausweg bietet sich eine möglichst starke Verknüpfung der Einzelinformationen in einem Informationsnetz an.

Anforderung 6

Verschiedene Dokumentationstypen müssen kombiniert werden können.

Je nach Aufgabe und Typ der Person, die auf eine bestimmte Information zugreifen will, passt ein anderes der in diesem Kapitel besprochenen Dokumentationskonzepte. Trotzdem soll die Gesamtdokumentation einheitlich verwendet und verwaltet werden können. Bei der Informationssuche kann es auch vorkommen, dass mehrere der Dokumentationskonzepte abwechslungsweise benötigt werden.

Anforderung 7

Multimediale Dokumentation muss integriert werden können.

Das verwendete Dokumentationsmedium muss die Einbindung multimedialer Dokumentation erlauben, um so beispielsweise die Abläufe im Framework zu dokumentieren.

Anforderung 8

Die Dokumentation muss interaktiv gestaltet werden können.

Es ist hilfreich, wenn direkt aus der Dokumentation zu den entsprechenden *Wizards*⁶ verzweigt werden kann.

Anforderung 9

Im Aufbau der Dokumentation sollte sich die Architektur des Komponentenframeworks widerspiegeln.

Das Auffinden von Information wird wesentlich vereinfacht, wenn sich die architektonischen Grundlagen des Frameworks auch in der zugehörigen Dokumentation widerspiegeln.

⁶ Wizards sind Hilfsmittel, die nach einem interaktiven Dialog Teile der Anwendung generieren.

7.3.2 Ein Prototyp

Die Techniken des *World Wide Web (WWW)* sind gut geeignet, um viele der aufgezählten Anforderungen direkt zu erfüllen. So ist beispielsweise die Integration von interaktiven Hilfsmitteln und Multimedia problemlos möglich.

Die Dokumentation enthält die Auflistung, bzw. Beschreibung der folgenden Elemente:

Komponenten

Die einzelnen Komponenten müssen aufgeführt und deren Verwendung und Konfiguration beschrieben werden. Wählt der Anwender eine bestehende Komponente aus, so soll er gemäss dem in Abschnitt 7.2.2 besprochenen Prinzip direkt Zugriff auf allenfalls vorhandene Anwendungen erhalten, die diese Komponente einsetzen.

Wichtig für den Einsatz oder die Weiterentwicklung bestehender Komponenten könnte auch der Verweis auf bestehende Testanwendungen und Testdaten sein, um die Komponente verifizieren zu können.

Elemente der Frameworkschicht

Die Elemente der Frameworkschicht und deren Zusammenspiel sind ideal mit einer Mustersprache beschreibbar. Ein erfahrener Anwender ist eventuell nicht an einem Muster mit einer sehr ausführlichen Beschreibung interessiert, sondern möchte nur eine Übersicht der verfügbaren Klassen und deren Methoden erhalten. Diese Information kann wie in Abschnitt 7.2.4 beschrieben, direkt aus den Quelltextdateien extrahiert werden.

Die Frameworkschicht enthält auch die Klassen, die zur Repräsentation der geometrischen Objekte benötigt werden. Die Beschreibung der BSDL-Befehle, welche zur Beschreibung dieser Objekte in einer Szene benötigt werden, müssen ebenfalls dokumentiert werden.

Elemente der Bibliotheksschicht

Die im vorausgegangenen Abschnitt besprochenen Ausführungen gelten auch hier.

Anwendungen

Zu jeder Anwendung, die mit *BOOGA* realisiert wurde, wird ein kurzer Steckbrief abgelegt. Er enthält mindestens die Aufgabe der Anwendung und eine Liste der verwendeten Komponenten. Idealerweise wird allenfalls vorhandene zusätzliche Dokumentation ebenfalls von hier aus referenziert. Diese Anwendungen dienen wie in

Abschnitt 7.2.2 wiederum als Beispiele und Vorlagen für zukünftige Anwendungen.

Zusammen mit den Anwendungen müssen auch die nötigen Testdaten abgelegt werden, um die Anwendung bei Modifikationen wieder auf Korrektheit testen zu können.

Mustersprache

Die Standardaufgaben eines Anwendungsentwicklers werden mit einer Mustersprache, wie in Abschnitt 7.2.1 beschrieben, dokumentiert. Die einzelnen Muster sind über *Hyperlinks* untereinander und mit der restlichen Dokumentation verbunden.

Wizards

Wizards helfen dabei Standardaufgaben zu automatisieren. Abbildung 7.6 zeigt ein Beispiel eines in das Dokumentationssystem integrierbaren Wizards.

Dokumente zu/über *BOOGA*

Trotz der vernetzten Dokumentation ist es aus unterschiedlichen Gründen immer noch nötig, herkömmliche Dokumente zu schreiben. Dies können Publikationen sein, Semester- oder Diplomarbeiten von Studenten oder Dokumente, die grundlegende Mechanismen erläutern.

Selbstverständlich sollte auch auf diese Dokumente in elektronischer Form zugegriffen werden können.

Im Folgenden werden anhand einiger *Screenshots* die grundlegenden Ideen der *BOOGA*-Dokumentation erläutert.

Der Einstieg in das *Dokumentationszentrum* erfolgt durch die in Abbildung 7.1 gezeigte Seite. Auf allen Seiten wird stets der grosse *BOOGA*-Schriftzug am oberen Seitenrand erscheinen. Durch Anklicken dieses Schriftzuges wird der Anwender immer wieder auf die Eingangsseite zurückgeführt.

Auf der linken Seite befinden sich vier *Icons*, die eine erste Vorselektion der gewünschten Information erlauben:

Info

In diesem Abschnitt der Dokumentation befinden sich einige allgemeine Angaben zu *BOOGA*. Wird dieses Icon gewählt, so wird die in Abbildung 7.2 gezeigte Seite angezeigt.

In der Mitte des unteren Teils wird eine neue Übersicht angezeigt, welche die Auswahl von allgemeinen Informationen, Informationen über das *BOOGA*-Team oder rechtliche

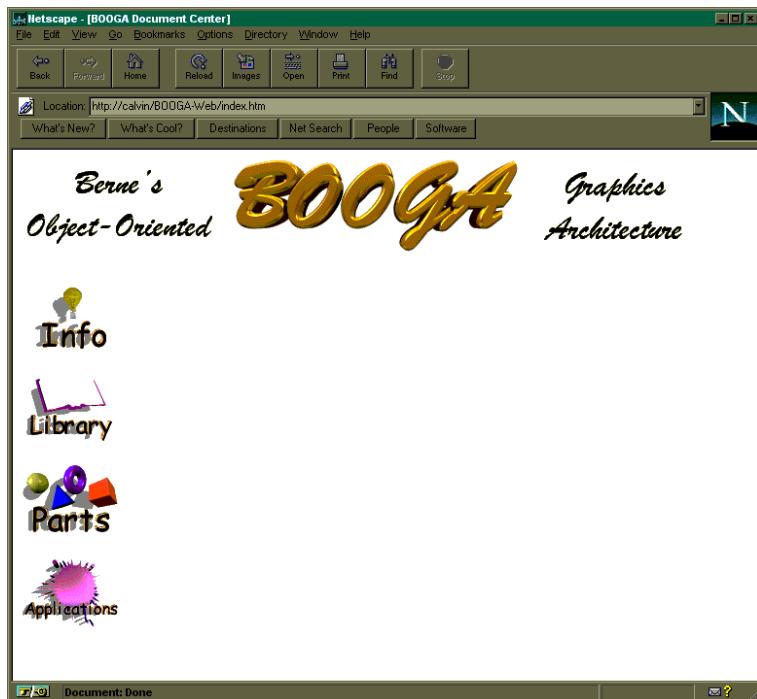


Abbildung 7.1

Der Einstieg in das **BOOGA**-Dokumentationssystem. Durch Anklicken der Icons links wird der Bereich gewählt, der von Interesse ist.

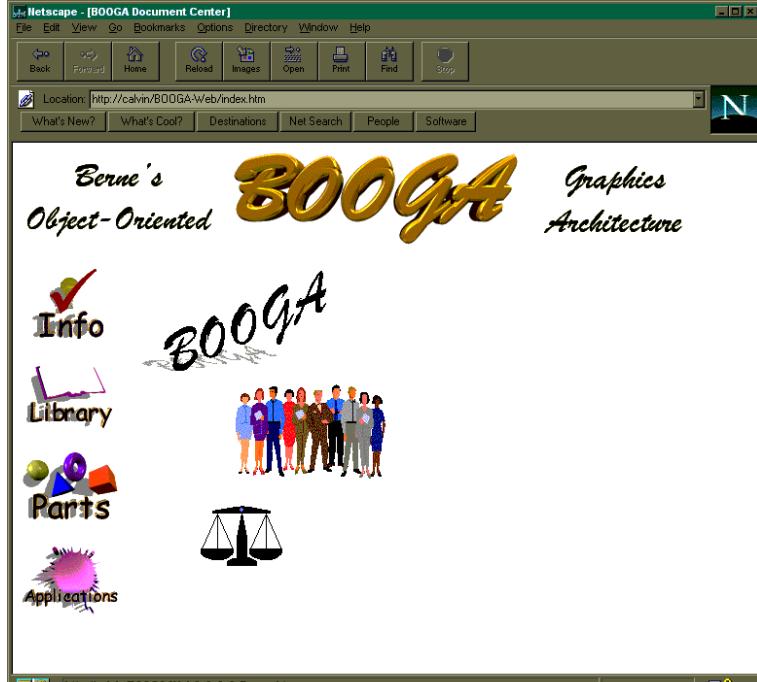


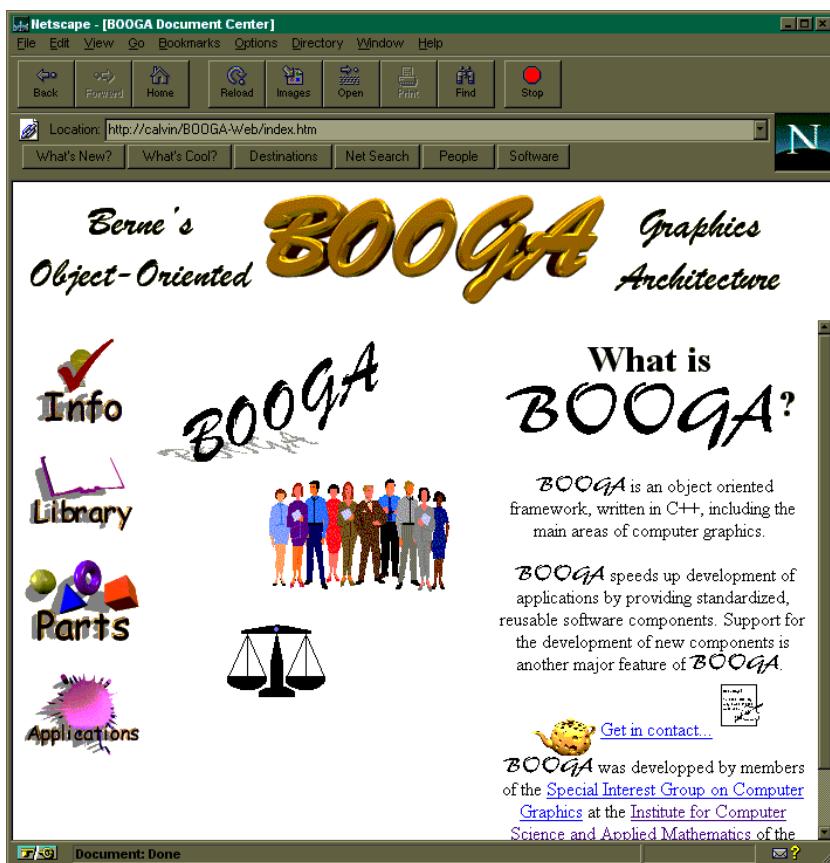
Abbildung 7.2

Das Icon *Info* wurde selektiert, erkennbar am überlagerten ✓-Zeichen.

Angaben zulässt. Abbildung 7.3 zeigt schliesslich ein *About BOOGA* an.

Library

Der Abschnitt *Library* enthält die gesammelten Dokumente

**Abbildung 7.3**

Einige allgemeine Angaben zu *BOOGA* auf der rechten unteren Seite.

über *BOOGA*: Publikationen, studentische Arbeiten oder Dokumente über das gesamte System.

Parts

Dieser Abschnitt ist einer der wichtigsten. Hier ist der Einstiegspunkt zu den Beschreibungen sowohl der einzelnen Komponenten als auch der Framework- und Bibliotheksschicht. Das Konzept der Komponenten, welche Operationen auf zwei- oder dreidimensionalen Daten sind, wird hier verwendet, um auf die Beschreibungen der Komponenten zugreifen zu können. Abbildung 7.4 zeigt die *Parts*-Seite. Durch Anklicken eines bestimmten Übergangs wird eine Liste der verfügbaren Komponenten des entsprechenden Typs angezeigt (vgl. Abbildung 7.5). Die Beschreibung von Klassen der Framework- und Bibliotheksschichten können durch Anwählen der mit '2D' beziehungsweise '3D' bezeichneten Kreise abgerufen werden.

Abbildung 7.5 zeigt auf der rechten Seite den Beginn der Auflistung von Komponenten. Die Komponenten des Typs *Operation3DTo3D* sind in vier Kategorien eingeteilt: *I/O*

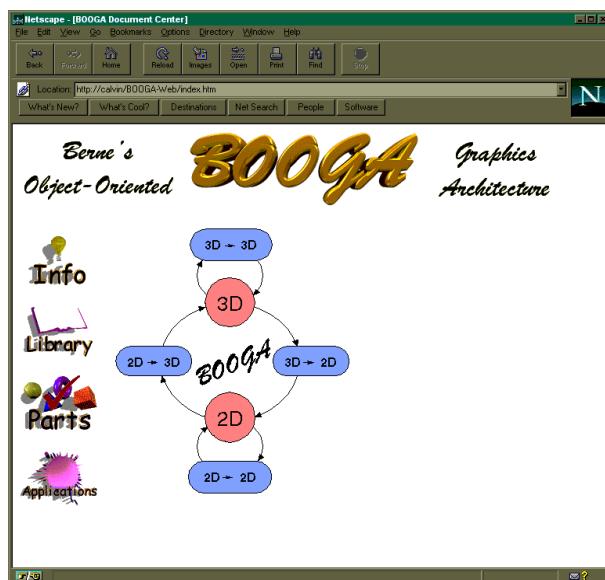


Abbildung 7.4

Die in Abschnitt 4.2 besprochene logische Sicht von **BOOGA** hilft auch bei der Gliederung der Dokumentation.

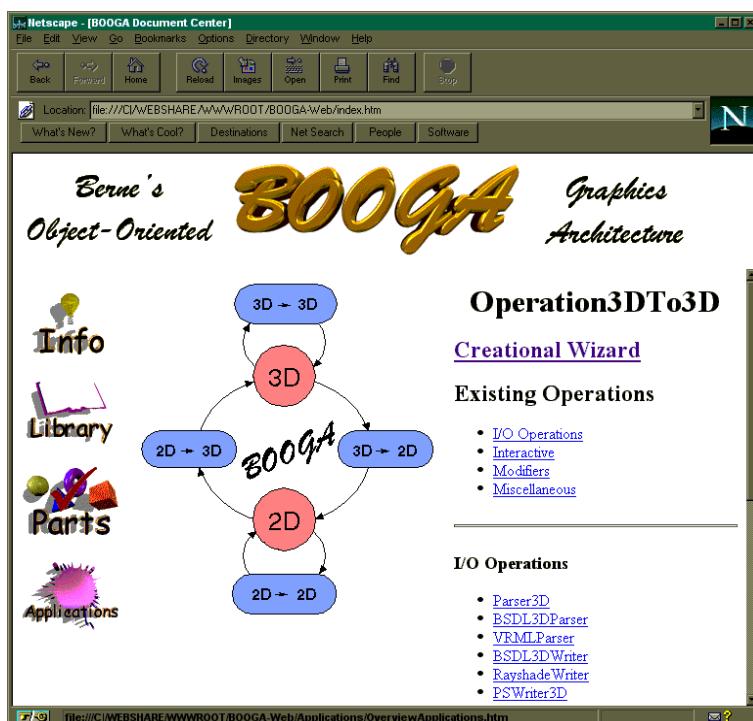


Abbildung 7.5

Durch Wählen des oberen Überganges in Abbildung 7.4 wird die Liste der existierenden Komponenten dieses Typs angezeigt, sortiert nach weiteren Kriterien.

Operations, Interactive, Modifiers und Miscellaneous. Außerdem befindet sich im Kopf der Liste ein Hyperlink, der auf einen *Class Wizard* (vgl. Abbildung 7.6) verweist.

Applications In diesem Abschnitt werden die bereits mit **BOOGA** erstellten Anwendungen aufgelistet und beschrieben.

**Abbildung 7.6**

Wenige Angaben genügen bereits, um den Rumpf einer Klassendeklaration und der zugehörigen Klassenimplementierung zu erzeugen.

7.4 Schlussbemerkungen

Aufgrund der in *BOOGA* gemachten Erfahrungen und der Einblicke in verschiedene Industrieprojekte lässt sich zusammenfassend festhalten, dass es gerade bei wiederverwendungszentrierten Systemen äusserst wichtig ist, nicht nur die Software, sondern auch die zugehörige Dokumentation mit aller Sorgfalt zu planen und zu entwickeln. Während es bei der Softwareerstellung selbstverständlich ist, unterschiedliche Methoden und Hilfsmittel einzusetzen sowie eine Architektur und ein Vorgehen zu planen, wird die Dokumentation oft einfach ‘nur’ geschrieben.

Bei Systemen wie *BOOGA*, die als Grundlage für Anwendungen dienen, genügen Dokumente, welche die Resultate der einzelnen Softwareentwicklungsphasen festhalten, nicht mehr. Vielmehr müssen eigentliche *Lehrmittel* erstellt werden, die in die oft hochkomplexe Materie einführen, *Nachschlagewerke*, welche dem bereits erfahrenen Anwender ein rasches Auffinden jedwelcher benötigter Information ermöglicht und *Richtlinien*, welche die Konsistenz des Systems bei Erweiterungen sicherstellen.

Alle diese Dokumente können rasch einen Umfang erreichen, der sehr abschreckend auf einen Entwickler wirkt. Eine gute Dokumentation sollte daher dem Anwender immer nur soviel Information zur Verfügung stellen, wie er gerade benötigt, nicht weniger, aber auch nicht mehr. Darüber hinaus sollte die Dokumentation den Anwender motivieren und zur weiten Auseinandersetzung mit dem System veranlassen.

Sind die dokumentierten Systeme einer dauernden Erweiterung sowie Änderungen unterworfen, wird die Sicherstellung der Konsistenz der Dokumentation in sich selbst sowie gegenüber dem System zu einer grossen Herausforderung.

Der Widerspruch, dass einerseits zur Sicherstellung der Konsistenz Redundanz weitgehend vermieden werden muss, andererseits aber dem Benutzer dieselbe Information unter verschiedenen Aspekten zugänglich sein sollte, kann wohl nur durch ein hypertextähnliches System aufgelöst werden.

Die in Abschnitt 7.2 vorgestellten alternativen Dokumentationskonzepte können dabei helfen, ein Dokumentationssystem zu entwickeln, da sich den aufgestellten Anforderungen gewachsen zeigt, wie dies im vorausgegangenen Abschnitt anhand eines auf dem Web basierenden Systems skizziert wurde.

Das nächste Kapitel schliesst die vorliegende Arbeit mit einer Besprechung der gemachten Erfahrungen sowie einem Ausblick in die Zukunft von *BOOGA* ab.



Walt Disney Productions.

Kapitel 8

Schlussbemerkungen

*Ich kann die Geister rufen aus
unermesslichen Tiefen.*

*Aber das kann ich, und so kann es
jedermann; nur, werden sie auch kommen,
wenn du sie rufst?*

Shakespeare, King Henry IV

8.1 Erreichte Ziele

Rückblickend kann festgehalten werden, dass bezüglich des Hauptziels, nämlich einer *vielseitig verwendbare Forschungsplattform* zu schaffen, die Erwartungen erfüllt, wenn nicht gar übertroffen wurden. Dies kommt durch die Anzahl und die Breite der abgedeckten Themen in den bereits abgeschlossenen [Ammon, 1996; Bächler, 1995; Balmer, 1996; Habegger, 1996; Liechti, 1996; Matthey, 1996; Matthey, 1997; Sagara, 1996; von Siebenthal und Wenger, 1996; Teuscher, 1996], sowie weiteren, zum Zeitpunkt der Drucklegung dieser Dissertation noch hängigen Arbeiten zum Ausdruck. Durch die Wiederverwendung von Konzepten, Architektur und Quellcode von *BOOGA* konnten in diesen Arbeiten die gestellten Aufgaben in wesentlich grösserer Tiefe behandelt werden, als dies ohne *BOOGA* der Fall gewesen wäre.

Ein weiteres Ziel und gleichzeitig wichtige Grundlage für *BOOGA* war die *Gliederung des Gebietes der Computergrafik* (vgl. Abschnitt 4.2). Die möglichen Operationen der Computergrafik werden als Abbildungen von einem Wertebereich in einen Definitionsbereich aufgefasst, wobei als Werte- und Definitionsbereich der zwei- und der dreidimensionale Raum möglich sind. Diese Gliederung führte schliesslich zur Entwicklung eines Softwaresystems, das dem in Abschnitt 4.3 eingeführten Begriff des *Komponentenframeworks* entspricht.

Wenn in dieser Arbeit und im Projekt *BOOGA* auch nicht allen mit der Wiederverwendung verbundenen Problemen die nötige Aufmerksamkeit zuteil werden konnte, so ist es doch gelungen, in einem abgegrenzten Bereich ein System zu erstellen und ein Klima zu schaffen, welche die Wiederverwendung als natürliches Element in den Entwicklungszyklus einbinden. Aus technischer Sicht ist dies stark auf dem in Abschnitt 4.3 besprochenen *Komponentenframework* begründet. Kennzeichnend aus Sicht der Softwarearchitektur sind die drei aufeinander aufbauenden Schichten der Wiederverwendungstechniken Bibliothek, Framework und Komponenten. Die Schichtung erlaubt, die in Kapitel 2 aufgeführten Vorteile der einzelnen Techniken zu kombinieren und die Nachteile weitgehend auszuschliessen. Als ergänzende Elemente zu Architektur und Software wurde ein *Vorgehensmodell* (vgl. Kapitel 5) eingeführt, sowie ein *Dokumentationsansatz* (vgl. Kapitel 7) besprochen. Weiter wurde eine Organisationsform erläutert, welche die Koordination von paralleler Framework- und Anwendungsentwicklung erlaubt (vgl. Abschnitt 5.5). Dieses Konzept des Komponentenframeworks wird von uns neben den praktischen Erfolgen als das wichtigste Resultat des Projektes *BOOGA* sowie der vorliegenden Arbeit betrachtet.

Ein weiteres Ziel war die einfache Einarbeitung in *BOOGA*. Oft erlebten Studenten den ersten Kontakt mit *BOOGA* als äusserst schwierig und

erdrückend. Wie nur soll so ein gigantisches Ungetüm verstanden und für die eigene Arbeit verwendet werden? Nach kurzer Zeit der intensiven Auseinandersetzung (aber auch Betreuung durch erfahrene **BOOGA**-Anwender) setzte sich bei den meisten die Erkenntnis durch, dass eben gerade dieses Verständnis gar nicht nötig ist. Meist muss nur ein kleiner Ausschnitt des ganzen Systems wirklich verstanden werden, für den Rest genügt ein oberflächliches Verständnis. In mehr als einem Fall führte der erste Kontakt mit **BOOGA** zu dem Wunsch, sich auch weiterhin mit dem System zu befassen. Dies war für uns als Frameworkentwickler natürlich jeweils eine sehr erfreuliche und ermutigende Erfahrung. Auch das Ziel, eine verhältnismässig *leicht erlernbare Umgebung* zu schaffen, kann somit als erfüllt betrachtet werden.

8.2 Erfahrungen

Dieser Abschnitt erläutert die wesentlichen Erfahrungen aus der Entwicklung von **BOOGA**.

Frameworkentwicklung

Die Entwicklung eines Frameworks erfolgt stets iterativ. Die Gründe dafür werden in [Johnson und Russo, 1991] erläutert: ein Entwurf muss selbstverständlich nur deshalb mehrmals überarbeitet werden, weil die Autoren ihn nicht auf Anhieb richtig machen konnten. Möglicherweise ist dies der Fehler der Entwickler: entweder hätten sie nur lange genug das Problem analysieren müssen oder sie waren nicht gut genug.

Tatsächlich liegt der Grund natürlich darin, dass es – aus unterschiedlichen Gründen – meist nicht möglich ist, alle Anforderungen von Anfang an korrekt und umfassend zu erfassen. Da diese Anforderungen aber die Grundlage für den gesamten nachfolgenden Entwicklungsprozess darstellen, kann auch eine sehr lange Analysephase, durchgeführt von einem sehr erfahrenen Analytiker kaum zum gewünschten Erfolg führen.

Bei der Frameworkentwicklung kommt erschwerend die erwünschte Eigenschaft der Wiederverwendbarkeit hinzu. Software kann erst als wiederverwendbar bezeichnet werden, wenn sie wiederverwendet wurde. Die Wiederverwendung ist somit der Test für die Eigenschaft ‘wiederverwendbar’. Wie bei jedem Test können aber Fehler und Probleme auftauchen, die Änderungen am Framework bewirken. Dieser Ablauf wird in Abbildung 8.1 veranschaulicht.

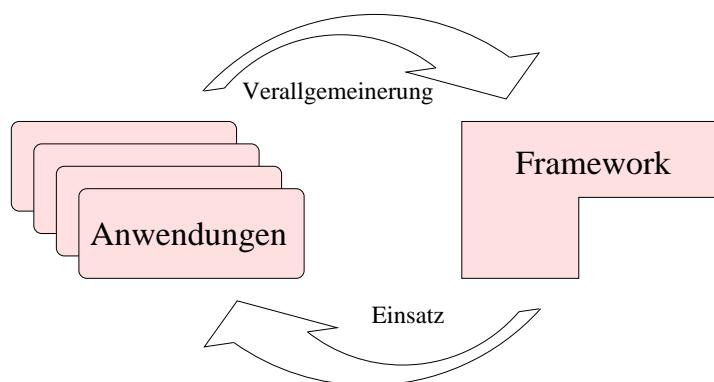


Abbildung 8.1

Die Entwicklung eines Frameworks erfolgt stets iterativ. Anwendungen, die mit dem Framework erstellt wurden, sind eine wichtige Quelle für Erweiterungen.

Abbildung 8.1 macht einen weiteren Grund für die Notwendigkeit der Iteration deutlich: Es kann im Allgemeinen nur ungenügend anhand theoretischer Überlegungen entschieden werden, wie wiederverwendbare Software realisiert werden soll. Meist wird man vielmehr entweder aufgrund von Erfahrung im Anwendungsbereich oder besser anhand von genügend Beispielen die Gemeinsamkeiten der zukünftigen Anwendungen extrahieren.

Kennzeichnend für ein Framework ist, dass Teile, welche sich durch eine hohe Änderungswahrscheinlichkeit über mehrere Anwendungen hinweg auszeichnen, flexibel gestaltet werden. Dadurch können Erweiterungen ohne Modifikationen vorgenommen werden. Erst die Erfahrung beim Einsatz des Frameworks zeigt, wo diese Flexibilität wirklich nötig ist und wo nicht.

Die Voraussetzungen zur Entwicklung eines eigenen Frameworks lassen sich wie folgt zusammenfassen:

- Die Entwickler des Frameworks brauchen ein sehr gutes Wissen über das Anwendungsgebiet. Idealerweise haben sie bereits mehrere Anwendungen in diesem Gebiet ohne den Einsatz eines Frameworks entwickelt.

Im Falle von **BOOGA** hatten die ersten Entwickler eine langjährige Erfahrung in der Entwicklung von Grafikanwendungen und in objektorientierten Technologien.

- Um zu entscheiden, ob eine eigene Frameworkentwicklung Sinn macht, sollte die Anzahl der mit Hilfe des Frameworks zu realisierenden Anwendungen abgeschätzt werden: die Investitionen werden sich frühestens nach drei bis fünf Anwendungen bezahlt machen.

BOOGA ist bis zum Zeitpunkt dieser Arbeit in mehr als einem Dutzend Semester- und Diplomarbeiten eingesetzt worden und Erweiterungen des Komponentenframeworks sind Gegenstand von weiteren Dissertationen. Rückblickend kann heute gesagt werden, dass ohne **BOOGA** die meisten Arbeiten in der gleichen Zeit deutlich weniger Resultate erzielt hätten.

- Iteration ist, wie gerade diskutiert, eine unabdingbare Begleitscheinung bei der Frameworkentwicklung. Es ist aber äußerst wichtig zu beachten, worüber iteriert wird. Änderungen der Implementierungsdetails sind selten problembehaftet. Das Design kann sich ebenfalls in recht weiten Grenzen ändern, ohne Schwierigkeiten zu bereiten. Schnittstellenänderungen ziehen zwar oft aufwendige Konsequenzen nach sich, doch können diese mit organisatorischen Massnahmen (vgl. Abschnitt 5.5) gehandhabt werden.

Die Vision des Frameworks, dessen Ausrichtung und architektonische Leitbilder müssen hingegen sehr früh stabilisiert werden und sollten sich nicht mehr ändern.

So wurde die logische Architektur von **BOOGA** (vgl. Abbildung 4.1 in Abschnitt 4.2) in den ersten drei Monaten der Projektlaufzeit entwickelt und anhand von Entwürfen von Prototypen verifiziert.

Während die meisten Klassenhierarchien, Klassen und Methoden geändert wurden, blieb diese logische Architektur stets stabil.

- Als Folge der iterativen Entwicklung muss abgeklärt werden, ob und wie bestehende Anwendungen an neue Versionen des Frameworks angepasst werden sollen. Eine fertige Anwendung kann ohne Anpassung an neue Versionen auskommen, solange keine Fehler in der Frameworkversion zutage treten. Sollte dies der Fall sein, oder wird für eine neue Version der Anwendung eine Eigenschaft der neuen Frameworkversion benötigt, muss die Anwendung angepasst werden. Die kostspielige Alternative ist der parallele Unterhalt mehrerer Versionen des Frameworks.

Bei der Entwicklung von **BOOGA** wurde Wert darauf gelegt, dass stets alle Anwendungen auf der neuesten stabilen Version des Frameworks aufbauten. Während der Laufzeit des Anwendungsprojektes wurde es dem Bearbeiter überlassen, den Zeitpunkt der Umstellung auf die neueste Version zu bestimmen. Nach Abschluss des Projektes wurde die Aufgabe der Anpassung in der Regel von den Mitgliedern des Frameworkteams übernommen.

Diese Überlegungen zur Frameworkentwicklung belegen die Tatsache, dass die Arbeit an einem Framework nie als abgeschlossen betrachtet werden kann. Anwendungen, welche mit einem Framework erstellt werden, üben stets gewisse Kräfte auf ein Framework aus. Werden Anwendungen erstellt, für welche das Framework noch nicht optimal ist, muss sich entweder das Framework anpassen können oder die Anwendungen können nicht mehr den grösstmöglichen Nutzen daraus ziehen. Dies hat mit der Zeit zur Folge, dass Alternativen zum Einsatz des Frameworks gesucht werden und das Framework schliesslich ‘stirbt’.

Eine weitere Konsequenz der besprochenen Spezialitäten der Frameworkentwicklung ist die Schwierigkeit, diese Arbeit zu planen. Folglich sollten Frameworks nie auf dem kritischen Pfad eines Projektes liegen.

Bei der Realisierung eines Frameworks sollten weiter die folgenden Empfehlungen beachtet werden:

Verwende eine kleine Anzahl unterschiedlicher Schnittstellen.

Je weniger unterschiedliche Schnittstellen ein Framework aufweist, desto einfacher ist die Einarbeitung und der Umgang damit. Eine kleine Anzahl unterschiedlicher Schnittstellen kann beispielsweise dadurch erreicht werden, dass in unterschiedlichen Klassen ähnliche Methoden mit demselben Namen bezeichnet werden.

Verwende gute Defaults für die häufigsten Anwendungsfälle.

Ein gutes Framework sollte, ohne dass neue Klassen erstellt werden, direkt verwendet werden können, um damit Standardaufgaben zu lösen.

Verwende Komposition statt Vererbung.

Vererbung kann bei einer typgebundenen Sprache wie C++ ganz generell für zwei unterschiedliche Zwecke eingesetzt werden:

- Zur Definition gemeinsamer Schnittstellen.
- Zur Wiederverwendung von Implementationsdetails.

Die Wiederverwendung von Implementationsdetails ist allerdings oft wesentlich flexibler realisierbar, wenn hierfür Komposition anstelle von Vererbung eingesetzt wird.

Der Entwicklungsumgebung und der richtigen Auswahl von Hilsmitteln kommt eine zentrale Bedeutung zu. Das Konzept für die Entwicklungsumgebung sollte so früh wie möglich erstellt werden. Sind einmal grosse Mengen von Quellcode erstellt worden und viele Personen beteiligt, so ist der nötige Aufwand wesentlich grösser.

Der bereits im Abschnitt 5.5 besprochenen Entwicklungsumgebung von **BOOGA** kommt ein bedeutender Anteil an der effizienten Abwicklung der Anwendungsprojekte zu.

Die Bedeutung von Beispielanwendung ist in dieser Arbeit bereits mehrfach zum Ausdruck gekommen. Beispiele sind ein wesentlicher Bestandteil einer Frameworkdokumentation und dienen als Test für den Einsatz und Beweis für die Wiederverwendbarkeit des Frameworks.

Bei **BOOGA** wurde jede Anwendung, die von einem Anwendungsteam erstellt wurde, allen anderen zur Verfügung gestellt. So wächst die Anzahl der Beispielanwendungen laufend.

Die Dokumentation ist, wie bereits in Kapitel 7 besprochen wurde, eine wesentliche Voraussetzung zur Wiederverwendung.

Rückblickend muss leider festgestellt werden, dass der Dokumentation bei der Entwicklung von **BOOGA** zuwenig Beachtung geschenkt wurde. Unterschiedliche Gründe haben zu diesem Mangel geführt:

- Zu Beginn des **BOOGA**-Projektes war noch keinesfalls sicher, dass **BOOGA** die heutige Grösse annehmen und den eingetretenen Erfolg haben wird. Für einen Forschungsprototyp, der fast nur von den

**Entwicklungs-
umgebung**

**Beispielan-
wendungen**

Dokumentation

Autoren verwendet wird, wäre eine detaillierte Dokumentation wie beispielsweise in Abschnitt 7.2.1 besprochen, ein unverhältnismässiger Aufwand gewesen. Die zukünftige Bedeutung von **BOOGA** und somit der Dokumentation wurde anfänglich unterschätzt.

- Trotzdem wurde rasch erkannt, dass herkömmliche Dokumentationsansätze in einem rasch ändernden Umfeld nicht zum Erfolg führen können. Versuche mit einem Hilfsmittel, das aus C++-Klassendeklarationen strukturierten Kommentar extrahiert und mittels L^AT_EX formatiert, wurden wieder eingestellt, als SNiFF+ als Entwicklungsumgebung ausgewählt wurde. Der rasche Zugriff auf die Quelltexte und die guten Such- und Visualisierungshilfsmittel machten diese Form der Dokumentation bald überflüssig.
- Die Dokumentation hätte vom Frameworkteam geschrieben werden müssen. Wir waren aber während der ganzen Entwicklungszeit vollständig mit den nötigen Erweiterung, der Fehlerbehebung und der Betreuung der Anwendungsteams ausgelastet, so dass die Dokumentation vernachlässigt werden musste. Ein Frameworkteam von nur zwei Personen¹ ist für ein Framework in der Grösse² von **BOOGA** klar zu klein.

Trotz dem weitgehenden Fehlen expliziter Dokumentation für **BOOGA** sind – bei genauerer Betrachtung – doch einige Elemente der nötigen Dokumentation vorhanden:

- Beispielanwendungen wurden bereits als wichtiger Teil der Dokumentation bezeichnet. Aufbauend auf **BOOGA** existieren heute viele Anwendungen unterschiedlicher Komplexität.
- Dank SNiFF als zentralem Hilfsmittel in der Entwicklungsumgebung ist der Zugriff auf den Quelltext und den Kommentar im Quelltext rasch und einfach möglich.
- Aufbauend auf **BOOGA** sind viele Anwendungsprojekte entstanden. Jedes solche Projekt beinhaltet eine eigene Dokumentation, so dass die Summe der vorhandenen Projektdokumentation ein ungefähres Bild vom Framework zeichnet.

¹Erschwerend kam hinzu, dass wir nicht die volle Arbeitszeit für **BOOGA** einsetzen konnten, sondern weitere Aufgaben im Bereich Lehre und Unterhalt zugunsten des Institutes zu erbringen hatten.

²**BOOGA** umfasste Ende 1996 über 500 Klassen und mehr als 200'000 Zeilen C++-Quelltext.

- Die vorliegende Arbeit und [Streit, 1997] zeigen schliesslich die Visionen, Konzepte und Mechanismen des Komponentenframeworks auf.

Abschliessend muss festgehalten werden, dass der Dokumentation gleich viel Bedeutung wie der Architektur, der Entwicklungsumgebung und der Organisation zukommen muss. Tatsächlich scheinen diese vier Themenbereiche die kritischen Elemente zu sein, deren Erfüllungsgrad über Gelingen oder Scheitern von Projekten mit dem zentralen Anliegen der Wiederverwendung entscheidet.

8.3 Ausblick

Die vorliegende Arbeit markiert nicht das Ende der Entwicklung von **BOOGA**. Ein Framework ist niemals fertig entwickelt. Obwohl sich einige stabile Elemente im Komponentenframework herauskristallisiert haben, sind noch viele Teile nicht oder nur unvollständig umgesetzt worden.

- Der gesamte Problemkreis ‘Radiosity’ wurde bisher lediglich konzeptionell untersucht (vgl. Abschnitt 6.3) aber noch nicht realisiert.
- Es gibt bisher kaum Anwendungen aus dem Bereich Bildanalyse. Komponenten aus diesem Problembereich könnten dazu beitragen, die Komponentenbibliothek zu vervollständigen und die Konzepte von **BOOGA** auch in diesem Bereich unterstützen.
- Bei vielen Anwendungen ist der noch zu schreibende Code sehr einfach. Es wäre wünschenswert, eine Scriptingsprache für **BOOGA** zu entwickeln, um somit auf einfache Art und Weise, im Idealfall ganz ohne C++-Code schreiben zu müssen, neue Anwendungen erstellen zu können.

Mit der zunehmenden Breite an existierenden Anwendungen steigt auch die Attraktivität von **BOOGA** für den Einsatz in der Lehre. Auch Dienstleistungen für andere Universitätsinstitute oder externe Interessenten sind denkbar, ist doch in der letzten Zeit die Computergrafik immer stärker in das Zentrum des allgemeinen Interesses gerückt.

Notwendige Voraussetzung für jeden zukünftigen Einsatz von **BOOGA** ist allerdings eine ständige Pflege und Weiterentwicklung des Frameworks unter Beachtung der im vorausgegangenen Abschnitt besprochenen Erfahrungen. Alle Elemente des Frameworks bedürfen der Pflege:

- *Die Architektur muss neuen Bedürfnissen angepasst werden.*
Diese Anpassung muss derart vorgenommen werden, dass sich keine Widersprüche zur bereits etablierten Architektur ergeben.
- *Der Quellcode verschiedener Autoren muss konsolidiert werden.*
Ein altes Sprichwort lautet: “Viele Köche verderben den Brei”. Arbeiten viele Entwickler autonom und ohne Vorschriften, so werden die entstehenden Lösungen mit der Zeit immer stärker von einer gemeinsamen Grundidee abweichen. Dies wiederum erschwert, ja verunmöglicht mit der Zeit die erfolgreiche Wiederverwendung.

- *Eine stabile, aktuelle Entwicklungsumgebung.*

Die Wiederverwendung hat sich in den von uns betreuten studentischen Projekten als erfolgreich erwiesen, weil jederzeit alle Beteiligten Entwickler auf alle bestehenden und sich in Entwicklung befindlichen Softwareelemente Zugriff hatten. Jeder Entwickler konnte also abschätzen, ob sich für ihn eine Neuentwicklung einer Komponente lohnt.

Es bleibt zu hoffen, dass sich auch in Zukunft genügend engagierte Forscher in der Fachgruppe Computergeometrie und Grafik finden, welche die in dieser Arbeit skizzierten Visionen weiter verfolgen und **BOOGA** als sich weiterentwickelndes System zu betreuen bereit sind.

Literaturverzeichnis

- [Alexander, 1979] Alexander C. (1979). *The Timeless Way of Building*. Oxford University Press.
- [Amann, 1991] Amann S. (1991). Theoretische Grundlagen eines CSG Shaders basierend auf z-Buffering. Informatikprojekt IAM-PR-91396, Institut für Informatik und angewandte Mathematik, Universität Bern.
- [Amann, 1993] Amann S. (1993). RADSHADE – Ein globales Beleuchtungsmodell basierend auf Raytracing und Radiosity. Diplomarbeit, Institut für Informatik und angewandte Mathematik, Universität Bern.
- [Amann et al., 1997] Amann S., Streit C., und Bieri H. (1997). *BOOGA – A Component-Oriented Framework for Computer Graphics*. In *GraphiCon '97 Proceedings, Moskau*, Seiten 193–200.
- [Ammon, 1996] Ammon L. (1996). Magische Bilder entzaubert – SIRDS Algorithmen im Vergleich. Informatikprojekt IAM-FCG-9602, Institut für Informatik und angewandte Mathematik, Universität Bern.
- [Bächler, 1995] Bächler R. (1995). Entwurf und Implementierung einer NURBS-Library. Diplomarbeit, Institut für Informatik und angewandte Mathematik, Universität Bern.
- [Balmer, 1996] Balmer R. (1996). Sketching – Generierung von Szeneninformationen aus einer Skizze. Informatikprojekt IAM-FCG-9604, Institut für Informatik und angewandte Mathematik, Universität Bern.
- [Bebie, 1994] Bebie T. (1994). Texturen, verbesserte Z-Buffer-Algorithmen und eine interaktive 3D-Testumgebung. Diplomarbeit, Institut für Informatik und angewandte Mathematik, Universität Bern.
- [Beck und Cunningham, 1989] Beck K. und Cunningham W. (1989). A Laboratory for Teaching Object-Oriented Thinking. In *SIGPLAN Notices*, Ausgabe 24.
- [Ben-Natan, 1995] Ben-Natan R. (1995). *CORBA : A Guide to Common Object Request Broker Architecture*. McGraw-Hill.

- [Birrer et al., 1995] Birrer A., Bischofberger W. R., und Eggenschwiler T. (1995). Wiederverwendung durch Framework-Technik – vom Mythen zur Realität. *OBJECTspektrum*, September/Oktober, Nummer 5, Seiten 18–26.
- [Booch, 1991] Booch G. (1991). *Object Oriented Design with Applications*. The Benjamin/Cummings Publishing Company.
- [Booch, 1994a] Booch G. (1994). Designing an Application Framework. *Dr. Dobbs Journal*, February, Seiten 24–31.
- [Booch, 1994b] Booch G. (1994). *Object-Oriented Analysis and Design. With Applications*. The Benjamin/Cummings Publishing Company, second edition.
- [Booch et al., 1997] Booch G., Rumbaugh J., und Jacobson I. (1997). Unified Modeling Language, Version 1.1. URL: <http://www.rational.com/uml>.
- [Breymann, 1996] Breymann U. (1996). Die C++ Standard Template Library – Grundlagen und Konzepte. *OBJECTspektrum*, Januar/Februar, Nummer 1, Seiten 82–85.
- [Brocha und Cook, 1990] Brocha G. und Cook W. (1990). Mixin-based Inheritance. In *ECOOP/OOPSLA '90 Proceedings*, Seiten 303–311.
- [Brooks, 1975] Brooks F. P. (1975). *The Mythical Man-Month*. Addison Wesley.
- [Bühlmann, 1991] Bühlmann B. (1991). Graphischer Editor zum Erstellen von Rotationskörpern. Informatikprojekt IAM-PR-91386, Institut für Informatik und angewandte Mathematik, Universität Bern.
- [Bühlmann, 1992] Bühlmann B. (1992). Verknüpfung von digitalen Geländemodellen und Satellitendaten zur 3D-Darstellung. Informatikprojekt IAM-PR-92429, Institut für Informatik und angewandte Mathematik, Universität Bern.
- [Bühlmann, 1994] Bühlmann B. (1994). Ein Framework für Virtual Reality-Applikationen. Diplomarbeit, Institut für Informatik und angewandte Mathematik, Universität Bern.
- [Buschmann, 1996] Buschmann F. (1996). What is a Pattern? *Journal of Object-Oriented Programming (JOOP)*, March/April, Seiten 17–18.
- [Buschmann et al., 1996] Buschmann F., Meunier R., Rohnert H., Sommerlad P., und Stal M. (1996). *A System of Patterns — Pattern-Oriented Software Architecture*. John Wiley & Sons.

- [Campell et al., 1993] Campell R. H., Islam M., Raila D., und Madany P. (1993). Designing and Implementing CHOICES: an Object-Oriented System in C++. *Communications of the ACM*, September, Ausgabe 36, Nummer 9, Seiten 117–126.
- [Coad et al., 1995] Coad P., North D., und Mayfield M. (1995). *Object Models: Strategies, Patterns and Applications*. Yourdon Press.
- [Collison, 1990] Collison A. (1990). Effizienzsteigerung von Ray Tracer. Informatikprojekt IAM-PR-90355, Institut für Informatik und angewandte Mathematik, Universität Bern.
- [Collison, 1994] Collison A. (1994). Eine physikalisch basierte und eine heuristische Technik zur Generierung von 2D-Haufen. Diplomarbeit, Institut für Informatik und angewandte Mathematik, Universität Bern.
- [Coplien, 1992] Coplien J. O. (1992). *Advanced C++, Programming Styles and Idioms*. Addison Wesley.
- [Coplien, 1995] Coplien J. O. (1995). Advanced C++ Programming Styles. CHOOSE Tutorial, University of Berne.
- [Coplien und Schmidt, 1995] Coplien J. O. und Schmidt D. C., Editoren (1995). *Pattern Languages of Program Design*. Addison-Wesley.
- [CWC, 1995a] ComponentWare Consortium (CWC) (1995). ComponentWare Architecture: A Technical Product Description. URL: http://www.componentware.com/arch_wp.htm.
- [CWC, 1995b] ComponentWare Consortium (CWC) (1995). ComponentWare Consortium Mission Statement. URL: http://www.componentware.com/cwc_mi~1.htm.
- [CWC, 1995c] ComponentWare Consortium (CWC) (1995). ComponentWare Consortium Technology Plan - Statement of Work. URL: <http://www.componentware.com/techrmwp.htm>.
- [Czarnecki, 1996] Czarnecki K. (1996). „Separation of Concerns“ – Objektorientierte Frameworks und das generative Paradigma. *OBJECT-spectrum*, November/Dezember, Nummer 6, Seiten 35–40.
- [DeRemer und Kron, 1976] DeRemer F. und Kron H. H. (1976). Programming-in-the-Large Versus Programming-in-the-Small. *IEEE Transactions on Software Engineering*, June, Ausgabe SE-2, Nummer 2, Seiten 80–86.

- [Dubuis, 1991] Dubuis E. (1991). Objektorientierte Implementation der Radiosity-Methode. Diplomarbeit, Institut für Informatik und angewandte Mathematik, Universität Bern. Lizentiatsarbeit.
- [Dubuis, 1995] Dubuis E. (1995). *Acceleration Techniques for the Radiosity Method*. Inauguraldissertation, Institut für Informatik und angewandte Mathematik, Universität Bern.
- [Fabregas, 1990] Fabregas X. (1990). Implementierung und Anwendung eines gutstrukturierten Ray Tracers. Diplomarbeit, Institut für Informatik und angewandte Mathematik, Universität Bern.
- [Foley et al., 1990] Foley J., van Dam A., Feiner S., und Hughes J. (1990). *Computer Graphics, Principles and Practice*. Addison-Wesley, second edition.
- [Frei, 1996] Frei B. (1996). Eine Erweiterung von Radiosity mit Texturen. Diplomarbeit, Institut für Informatik und angewandte Mathematik, Universität Bern.
- [Gall, 1986] Gall J. (1986). *Semantics: How Systems Really Work and How They Fail*. Ann Arbor, MI: The General Systemantics Press, second edition.
- [Gamma, 1992] Gamma E. (1992). *Objektorientierte Software-Entwicklung am Beispiel von ET++*. Springer Verlag.
- [Gamma, 1996] Gamma E. (1996). Design Patterns, Frameworks, Components - Elements of modern Application Architectures. Vortrag an der Component User's Conference '96, München.
- [Gamma et al., 1993] Gamma E., Helm R., Johnson R., und Vlissides J. (1993). Design Patterns: Abstraction and Reuse of Object-Oriented Design. In Nierstrasz O., Editor, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. Springer-Verlag.
- [Gamma et al., 1995] Gamma E., Helm R., Johnson R., und Vlissides J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley.
- [Gamma und Weinand, 1996] Gamma E. und Weinand A. (1996). Mythen der objektorientierten Programmierung. Seminar IFA OT Consulting, Zürich.
- [Gossain und Anderson, 1990] Gossain S. und Anderson B. (1990). An Iterative-Design Model for reusable Object-Oriented Software. In *EC-COP/OOPSLA '90 Proceedings*, Seiten 12–27. ACM.

- [Habegger, 1994] Habegger P. (1994). Visualisierung der Funktionsweise eines Ray-Tracers. Informatikprojekt IAM-PR-93514, Institut für Informatik und angewandte Mathematik, Universität Bern.
- [Habegger, 1996] Habegger P. (1996). Ein grafischer Strukturbrowser. Diplomarbeit, Institut für Informatik und angewandte Mathematik, Universität Bern.
- [Hofer, 1994] Hofer A. (1994). EasyDo - Interaktive Erstellung direkt manipulierbarer 3D-Szenen. Diplomarbeit, Institut für Informatik und angewandte Mathematik, Universität Bern.
- [Hüni et al., 1995] Hüni H., Johnson R. E., und Engel R. (1995). A Framework for Network Protocol Software. In *OOPSLA '95 Conference Proceedings*.
- [Hüni und Keller, 1997] Hüni H. und Keller B. (1997). Ein OO-Framework für Netzwerkprotokoll-Software. *OBJECTspectrum*, Januar/Februar, Nummer 1, Seiten 51–56.
- [Jacobson et al., 1992] Jacobson I., Christerson M., Jonsson P., und Övergaard G. (1992). *Object-Oriented Software Engineering*. Addison-Wesley.
- [Java Beans, 1996a] Java Beans (1996). *JavaTM Beans 1.0 API Specification*. Sun Microsystems. URL: <http://java.sun.com/beans/spec.html>.
- [Java Beans, 1996b] Java Beans (1996). JavaTM Beans: A Component Architecture for Java. URL: <http://java.sun.com/docs/white/index.html>.
- [Johnson, 1992] Johnson R. E. (1992). Documenting Frameworks using Patterns. In *Proceedings of the OOPSLA '92 Conference on Object-oriented Programming Systems, Languages and Applications*, Seiten 63–76. Published as ACM SIGPLAN Notices, volume 27, number 10.
- [Johnson, 1993] Johnson R. E. (1993). How to Design Frameworks. Notes for the OOPSLA '93 Conference on Object-oriented Programming Systems, Languages and Applications.
- [Johnson, 1996] Johnson R. E. (1996). Frameworks. Seminar Zühlke Informatik, Zürich.
- [Johnson und Foote, 1988] Johnson R. E. und Foote B. (1988). Designing Reusable Classes. *Journal of Object-Oriented Programming (JOOP)*, June/July.

- [Johnson und Russo, 1991] Johnson R. E. und Russo V. E. (1991). Reusing Object-Oriented Design. Technical Report UIUCDCS 91-1696, University of Illinois.
- [Knuth, 1984] Knuth D. E. (1984). Literate programming. *The Computer Journal*, Nummer 2.
- [Koenig, 1995] Koenig A. (1995). *Working Paper for Draft Proposed International Standard for Information Systems – Programming Language C++*. Doc. No. X3J16/95-0087 WG21/N0687.
- [Kolb, 1992] Kolb C. (1992). *RAYSHADE Users Guide and Reference Manual*. URL: <http://www-graphics.stanford.edu/~cek/rayshade/doc/guide/guide.html>, siehe auch <http://www-graphics.stanford.edu/~cek/rayshade/rayshade.html>.
- [Krueger, 1992] Krueger C. W. (1992). Software Reuse. *ACM Computing Surveys*, Juni, Ausgabe 24, Nummer 2, Seiten 131–183.
- [Lewis et al., 1991] Lewis J. A., Henry S. M., Kafura D. G., und Schulman R. S. (1991). An Empirical Study of the Object-Oriented Paradigm and Software Reuse. In *OOPSLA'91 Proceedings, ACM SIGPLAN Notices*, Ausgabe 26, Seiten 184–196.
- [Liebermann, 1988] Liebermann H. (1988). Position Statement in the Panel on Varieties of Inheritance. In *Addendum to the OOPSLA '87 Proceedings*, Ausgabe 23, Seite 35. Published as SIGPLAN Notices.
- [Liechti, 1996] Liechti B. (1996). Virtual Reality Language Facility für Berne's Object-Oriented Graphics Architecture. Diplomarbeit, Ingenieurschule Bern, Höhere Technische Lehranstalt HTL, Informatik-Abteilung.
- [Lindner, 1990] Lindner S. (1990). Literarisches Programmieren : Donald E. Knuths WEB-System. *c't*, Ausgabe 10.
- [Mani, 1994] Mani R. (1994). Darstellung von Feuer. Informatikprojekt IAM-PR-93517, Institut für Informatik und angewandte Mathematik, Universität Bern.
- [Martin und McClure, 1985] Martin J. und McClure C. (1985). *Diagramming Techniques for Analysts and Programmers*. Prentice-Hall.
- [Matthey, 1996] Matthey T. (1996). Computer Animation für **BOOGA**. Informatikprojekt IAM-FCG-9601, Institut für Informatik und angewandte Mathematik, Universität Bern.

- [Matthey, 1997] Matthey T. (1997). Objektorientierte gebäudemodellierung. Diplomarbeit, Institut für Informatik und angewandte Mathematik, Universität Bern.
- [McClure, 1989] McClure C. (1989). *CASE is Software Automation*. Prentice-Hall.
- [McIlroy, 1968] McIlroy M. D. (1968). Mass Produced Software Components. In Naur P. und Randell B., Editoren, *Software Engineering; Report on a conference by the NATO Science Committee*, Seiten 138–150, Garmisch, Germany. NATO Science Affairs Division, Brussels, Belgium.
- [Meier, 1986] Meier A. (1986). *Methoden der grafischen und geometrischen Datenverarbeitung*. Teubner.
- [Meijler und Nierstrasz, 1997] Meijler T. D. und Nierstrasz O. (To appear 1997). Beyond Objects: Components. In Papazoglou M. P. und Schlageter G., Editoren, *Cooperative Information Systems: Current Trends & Directions*. Academic Press.
- [Metz, 1995] Metz I. (1995). *Bintree Lab: Ein Framework von Datenstrukturen und Algorithmen für Bintrees*. Inauguraldissertation, Institut für Informatik und angewandte Mathematik, Universität Bern.
- [Musser und Saini, 1996] Musser D. R. und Saini A. (1996). *STL Tutorial and Reference Guide - C++ Programming with the Standard Template Library*. Addison-Wesley.
- [Nierstrasz et al., 1991] Nierstrasz O., Tsichritzis D., de Mey V., und Stadelmann M. (1991). Objects + Scripts = Applications. In *Esprit 1991 Conference*, Seiten 534–552. Kluwer Academic Publishers.
- [Nierstrasz et al., 1992] Nierstrasz O., Gibbs S., und Tsichritzis D. (1992). Component-Oriented Software Development. *Communications of the ACM*, September, Ausgabe 35, Nummer 9, Seiten 160–165.
- [Nierstrasz und Dami, 1996] Nierstrasz O. und Dami L. (1996). Component-Oriented Software Technology. In Nierstrasz O. und Tsichritzis D., Editoren, *Object-Oriented Software Composition*, Seiten 3–28. Prentice-Hall.
- [Nierstrasz und Meijler, 1995] Nierstrasz O. und Meijler T. (1995). Requirements for a Composition Language. In Ciancarini P., Nierstrasz O., und Yonezawa A., Editoren, *Object-Based Models and Languages for Concurrent Systems*, Ausgabe 924 von *Lecture Notes in Computer Science*, Seiten 147–161. Springer-Verlag, Berlin.

- [Ohlsson, 1994] Ohlsson L. (1994). Framework and OO Method Evolution. URL: <http://www.pt.hk-r.se/~michaelm/lofwandmethod.html>. Position Paper Framework Research Group, University of Karlskrona/Ronneby.
- [OPENDOC, 1994] OPENDOC (1994). *OPENDOC for Macintosh - An Overview for Developers*. Apple Computer, Inc.
- [Parnas und Clements, 1986] Parnas D. L. und Clements P. C. (1986). A Rational Design Process: How and Why to Fake It. *IEEE Transactions on Software Engineering*, February, Ausgabe SE-12, Nummer 2, Seiten 251–257.
- [Pavlidis, 1982] Pavlidis T. (1982). *Algorithms for Graphics and Image Processing*. Computer Science Press.
- [Pree, 1995] Pree W. (1995). *Design Patterns for Object-Oriented Software Development*. Addison-Wesley.
- [Roberts und Johnson, 1996] Roberts D. und Johnson R. E. (1996). Evolve Frameworks into Domain-Specific Languages. URL: <http://www.cs.wustl.edu/~schmidt/PLoP-96/roberts.ps>. Submitted to PLoP '96 Writters Workshop.
- [Rumbaugh et al., 1991] Rumbaugh J., Blaha M., Premerlani W., Eddy F., und Lorensen W. (1991). *Object-Oriented Modeling and Design*. Prentice-Hall, Inc.
- [Russo et al., 1988] Russo V., Johnston G., und Campell R. (1988). Process Management and Exception Handling in Multiprocessor Operating Systems Using Object-Oriented Techniques. In *OOPSLA '88 Conference Proceedings*, Ausgabe 23, Seiten 248–258. Published as OOPSLA '88, Special Issue of SIGPLAN Notices.
- [Sagara, 1996] Sagara P. (1996). *BOOGA's new House*. Diplomarbeit, Ingenieurschule Bern, Höhere Technische Lehranstalt HTL, Informatik-Abteilung.
- [Salus, 1994] Salus P. H. (1994). *A Quarter Century of UNIX*. Addison-Wesley.
- [Schmidt, 1996] Schmidt D. C. (1996). Why reuse has failed... and how you can make it work for you. Seminar ascom Corporate Training, Bern.
- [Schnur, 1996] Schnur B. (1996). Objektorientierung in der Versicherungsbranche. *OBJECTspectrum*, November/Dezember, Nummer 6, Seiten 41–45.

[Schreiner und Friedman, 1985] Schreiner A. T. und Friedman G. (1985). *Comiler bauen mit UNIX. Eine Einführung.* Carl Hanser Verlag.

[Seidewitz, 1996] Seidewitz E. (1996). Controlling Inheritance. *Journal of Object-Oriented Programming (JOOP)*, January, Seiten 36–42.

[Shaw, 1990] Shaw M. (1990). Prospects for an Engineering Discipline of Software. *IEEE Software*, November, Ausgabe 7, Nummer 6, Seiten 15–24.

[Shaw und Garlan, 1996] Shaw M. und Garlan D. (1996). *Software Architecture - Perspectives on an Emerging Discipline.* Prentice-Hall.

[Silion und Puech, 1994] Silion F. und Puech C. (1994). *Global Illumination.* Morgan Kaufmann Publishers.

[Smart, 1997] Smart J. (1997). *User Manual for wxWINDOWS 1.67: A Portable C++ GUI Toolkit.* Artificial Intelligence Applications Institute, University of Edinburgh. EH1 1HN, URL: <http://web.ukonline.co.uk/julian.smart/wxwin>.

[Stainhauser und Spiess, 1993] Stainhauser D. und Spiess L. (1993). Pre-viewer für den Raytracer Rayshade und Implementation eines z-Buffer-Algorithmus für CSG. Informatikprojekt IAM-PR-92436/92437, Institut für Informatik und angewandte Mathematik, Universität Bern.

[Stepanov und Lee, 1995] Stepanov A. und Lee M. (1995). The standard template library. URL: <ftp://butler.hpl.hp.com/stl/doc.ps>. Hewlett Packard.

[Streit, 1993] Streit C. (1993). Modellierung mit Lindenmayer-Systemen in der Computergrafik. Diplomarbeit, Institut für Informatik und angewandte Mathematik, Universität Bern.

[Streit, 1997] Streit C. (1997). *BOOGA — Ein Komponentenframework für Grafikanwendungen.* Inauguraldissertation, Institut für Informatik und angewandte Mathematik, Universität Bern.

[Telligent Inc., 1994a] Telligent Inc. (1994). *Building Object-Oriented Frameworks.* URL: <http://www.telligent.com/buildingoofw.html>. A Telligent White Paper.

[Telligent Inc., 1994b] Telligent Inc. (1994). *Leveraging Object-Oriented Frameworks.* URL: <http://www.telligent.com/leveraging-oofw.html>. A Telligent White Paper.

- [Telligent Inc., 1995] Telligent Inc. (1995). Vorteile objektorientierter Frameworks. *OBJECTspektrum*, September/Oktober, Nummer 5, Seiten 10–16. Übersetzter Teilauszug aus [Telligent Inc., 1994b].
- [Teuscher, 1996] Teuscher T. (1996). Shading Languages. Diplomarbeit, Institut für Informatik und angewandte Mathematik, Universität Bern.
- [Udell, 1994] Udell J. (1994). Componentware. *Byte*, Mai, Seiten 46–56.
- [Vlissides et al., 1996] Vlissides J. M., Coplien J. O., und Kerth N. L., Editoren (1996). *Pattern Languages of Program Design*. Addison-Wesley.
- [von Siebenthal und Wenger, 1996] von Siebenthal T. und Wenger T. (1996). Anfertigen von Klassendiagrammen aus Sourcecode. Informatikprojekt IAM-PR-FCG-03, Institut für Informatik und angewandte Mathematik, Universität Bern.
- [Webster, 1995] Webster B. F. (1995). *Pitfalls of Object-Oriented Development*. M&T Books.
- [Wegner, 1983] Wegner P. (1983). Varieties of Reusability. In *Workshop on Reusability in Programming*, Seiten 30–44. ITT Programming.
- [Weinand, 1996] Weinand A. (1996). Components: Another End to the Software Crisis? In *Components User Conference (CUC)*.
- [Weinand et al., 1988] Weinand A., Gamma E., und Marty R. (1988). ET++ – an Object-Oriented Application Framework in C++. In *OOPSLA '88 Conference Proceedings*, Seiten 46–57.
- [Weyerhäuser, 1995] Weyerhäuser M. (1995). Taligens CommonPoint-Architektur: Frameworks - mehr als nur Objekte. *OBJECTspektrum*, September/Oktober, Nummer 5, Seiten 60–67.

Anhang A

Notationen und Dateiformate

A.1 ***BOOGA**-LEGOS¹*

Dieser Abschnitt beschreibt die grafische Notation, die für die Visualisierung von ***BOOGA***-Applikationen verwendet wird. Beispiele für deren Anwendung, sind in Kapitel 6, in [Streit, 1997], sowie in [Amann et al., 1997] zu finden. Die Notation wurde mit dem Ziel entwickelt, den Aufbau von ***BOOGA***-Applikationen auf einfache Art und Weise zu illustrieren. Sie ist besonders für die Darstellung von Applikationen mit einem deterministischen Verhalten geeignet. Anwendungen, deren Ablauf durch den Benutzer in weiten Grenzen beeinflusst werden können, eignen sich hierfür wesentlich weniger gut, da keine Symbole für die Darstellung von Verzweigungen und Iterationen existieren.

Jeder Komponententyp wird durch ein eigenes Symbol repräsentiert. Die Dimension der Eingabewelt wird durch die Anzahl von Vertiefungen dargestellt und diejenige des Resultates mit der entsprechenden Zahl von Erhebungen. Zum Beispiel besitzt der Komponententyp **Operation3DTo2D** auf der oberen Seite drei Vertiefungen und auf der unteren zwei Erhebungen (siehe weiter unten). Dadurch werden Objekte vom Typ **World3D** und **World2D** als Eingabe- bzw. Ausgabedatenstrukturen repräsentiert. Die zur Verfügung stehenden Symbole werden in der Folge besprochen.

A.1.1 Operationen

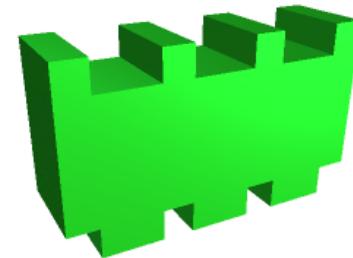
Operation3DTo3D

Die **Operation3DTo3D**-Komponente verarbeitet 3D-Welten. Das Resultat ist die transformierte Eingabe und/oder ein neu erzeugtes Objekt vom Typ **World3D**. Beispiele für solche Komponententypen sind Parser oder Editoren.

¹Dieser Abschnitt ist eine Übersetzung aus [Amann et al., 1997], ergänzt um Abschnitt A.1.2.

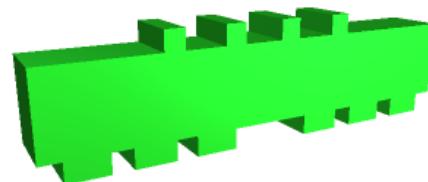
Eingabewelt : 3D
 Resultatwelt(en) : 3D

Die Eingabewelt durchläuft die Komponente und wird von dieser verarbeitet. Im Verarbeitungsschritt hat die Komponente lesenden und schreibenden Zugriff auf die Elemente der Eingabewelt.



Eingabewelt : 3D
 Resultatwelt(en) : 2 × 3D

Eine neue 3D-Welt wird erzeugt. Die Komponente hat aber auch lesenden und schreibenden Zugriff auf die Eingabewelt.

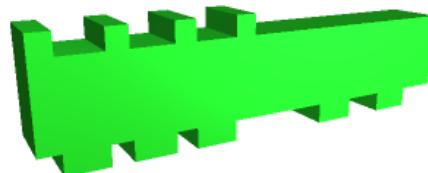


Operation3DTo2D

Die **Operation3DTo2D**-Komponente erzeugt eine 2D-Welt basierend auf einer 3D-Eingabe. Beispiele für diesen Komponententyp sind die verschiedenen Rendering-Verfahren.

Eingabewelt : 3D
 Resultatwelt(en) : 3D+2D

Eine 2D-Welt wird erzeugt. Die Komponente hat aber auch lesenden und schreibenden Zugriff auf die 3D-Eingabewelt.

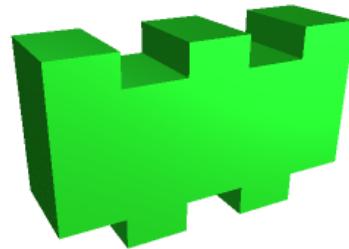


Operation2DTo2D

Die **Operation2DTo2D**-Komponente verarbeitet 2D-Welten. Das Resultat ist die transformierte Eingabe und/oder ein neu erzeugtes Objekt vom Typ **World2D**. Beispiele für solche Komponententypen sind Bildbearbeitungsfunktionen oder Editoroperationen.

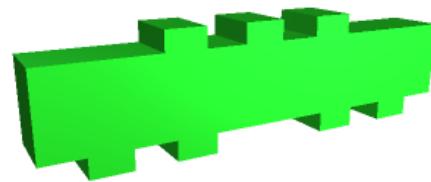
Eingabewelt : 2D
 Resultatwelt(en) : 2D

Die Eingabewelt durchläuft die Komponente und wird von dieser verarbeitet. Im Verarbeitungsschritt hat die Komponente lesenden und schreibenden Zugriff auf die Elemente der Eingabewelt.



Eingabewelt : 2D
 Resultatwelt(en) : $2 \times 2D$

Eine neue 2D-Welt wird erzeugt. Die Komponente hat aber auch lesenden und schreibenden Zugriff auf die Eingabewelt.

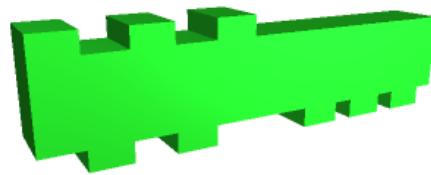


Operation2DTo3D

Die Operation2DTo3D-Komponente erzeugt eine 3D-Welt basierend auf einer 2D-Eingabe. Beispiele für diesen Komponententyp sind die verschiedenen Verfahren der Bildanalyse, die aus 2D-Datenstrukturen 3D-Modell rekonstruieren.

Eingabewelt : 2D
 Resultatwelt(en) : $2D+3D$

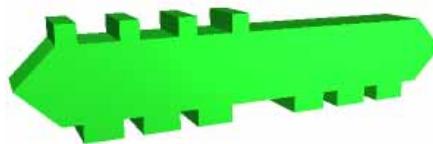
Eine 3D-Welt wird erzeugt. Die Komponente hat aber auch lesenden und schreibenden Zugriff auf die 2D-Eingabewelt.



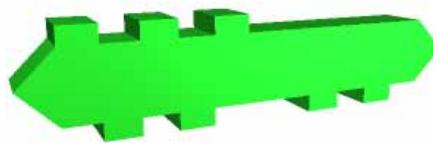
A.1.2 Bedingte Ausführung

Mit den **BOOGA**-LEGOs dieses Abschnittes können einfache bedingte Abläufe in einem Diagramm ausgedrückt werden. Die Bedingung wird als Text in das LEGO geschrieben. Trifft die Bedingung zu, wird die Eingabewelt auf der linken Seite weitergegeben, ansonsten rechts.

Eingabewelt : 3D
 Resultatwelt(en) : 3D



Eingabewelt : 2D
 Resultatwelt(en) : 2D



A.1.3 Erzeugung von ‘Welten’

Die beiden folgenden Symbole symbolisieren die Erzeugung leerer 2D-
 bez. 3D-Welten (Objekte vom Typ `World2D` und `World3D`).

Eingabewelt : —
 Resultatwelt(en) : 3D



Eingabewelt : —
 Resultatwelt(en) : 2D



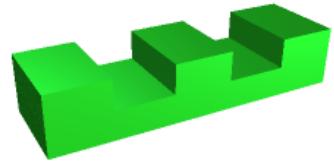
A.1.4 Löschen von ‘Welten’

Die beiden letzten Symbole werden für die Entfernung von zuvor er-
 zeugten Weltobjekten verwendet. Diese entstehen durch eine *Creation*-
 Operation oder bei der Aktivierung einer Komponenten.

Eingabewelt : 3D
 Resultatwelt(en) : —



Eingabewelt : 2D
Resultatwelt(en) : —



A.2 BSDL Sprachreferenz²

BOOGA besitzt eine eigene Szenenbeschreibungssprache (BSDL, **BOOGA** Scene Description Language), die sowohl für den 2D- wie für den 3D-Fall zum Einsatz kommt. Die Sprache zeichnet sich durch folgende Eigenschaften aus:

- *Wenige Schlüsselwörter*

Im Unterschied zu vergleichbaren Sprachen (vgl. zum Beispiel RAY-SHADE) kennt BSDL lediglich die vier Schlüsselwörter `define`, `const`, `using` und `namespace`. Dadurch hat die Sprache eine sehr einfache Struktur (siehe auch den nächsten Abschnitt).

- *Dynamische Konfiguration*

Der in **BOOGA** enthaltene Parser für BSDL lässt sich dynamisch, d.h. zur Laufzeit konfigurieren (Komponenten `Parser2D` und `Parser3D`). Erst danach ist er beispielsweise in der Lage, die Definition einer Kugel zu verarbeiten. Das zugehörige Schlüsselwort (`sphere` in unserem Beispiel) ist vor dem Konfigurationsprozess nicht im Sprachumfang von BSDL enthalten. Auch mathematische Funktionen werden auf diese Weise dem Parser bekanntgegeben.

Das Konzept erlaubt die beliebige Erweiterung des Sprachumfangs, was eine Grundbedingung für einen Parser in einem Framework darstellt.

- *Mathematische Ausdrücke*

BSDL kann einfache arithmetische Ausdrücke verarbeiten, die sich aus den vier Grundoperationen zusammensetzen. Der Sprachumfang lässt sich allerdings dynamisch um beliebige mathematische Funktionen erweitern. Auf diese Weise werden zum Beispiel alle trigonometrischen Funktionen zur Verfügung gestellt.

- *Polymorphe Variablen*

In Berechnungen oder als Parameter für Objektdefinitionen können einfache Zahlenwerte, Vektoren (2D und 3D), Matrizen (2D und 3D) und Zeichenketten verwendet werden. Die verschiedenen Datentypen können frei miteinander kombiniert werden. Gegebenenfalls werden automatische Konversionen vom System durchgeführt.

²Dieser Anhang wurde mit freundlicher Genehmigung aus [Streit, 1997] übernommen.

A.2.1 Sprachdefinition

BSDL ist sehr einfach aufgebaut und lässt sich mit Hilfe der erweiterten Backus Nauer Form (EBNF-Notation) wie folgt beschreiben:

Eine Welt besteht aus einer beliebigen Anzahl von Definitionen und Objekten.

```
World ::= { <Definition> | <Object> }+
```

Eine Definition erlaubt die Erzeugung von Objekten, Namenräumen und Konstanten, die für den Szenenaufbau verwendet werden können. Anstelle von IDENTIFIER können zur Laufzeit beliebige neue Schlüsselwörter konfiguriert werden.

```
Definition ::= define IDENTIFIER <Specifier>
            | define IDENTIFIER namespace `;`
            | const IDENTIFIER <Value> `;`
```

Die nachstehende Regel spezifiziert das Aussehen von Objektdefinitionen. Sie werden durch ein Schlüsselwort eingeleitet und können mit einer beliebigen Anzahl von Parametern attributiert sein. Zudem ist eine beliebige Verschachtelung der Definitionen möglich.

Mit Hilfe der using Direktive kann ein benutzerdefinierter Namenraum aktiviert werden (siehe das Beispiel im nächsten Abschnitt).

```
Specifier ::= IDENTIFIER <ValueList> <OptSpecifiers>
           | using IDENTIFIER `;`
```

```
OptSpecifiers ::= `;`
                 | `(` {<Specifier>}+ `)` [ `;` ]
```

```
ValueList ::= [ <Value> ]
            | `(` <Values> `)`
```

Parameter für Objektdefinitionen oder Argumente von mathematischen Ausdrücken können sich aus Zahlwerten, Vektoren, Matrizen und Zeichenketten zusammensetzen. Im Unterschied zum IDENTIFIER ist die Zeichenkette eines STRING durch Anführungsstriche begrenzt.

```
Values ::= { <Value> ` , ` } <Value>
```

```
Value ::= NUMBER
        | VECTOR2D | VECTOR3D
        | MATRIX2D | MATRIX3D
        | STRING | IDENTIFIER
        | <Expression>
```

Mathematische Ausdrücke setzen sich aus den vier Grundoperationen zusammen. Eine beliebige Klammerung ist möglich. Zusätzlich werden Funktionen unterstützt, die dynamisch in den Sprachumfang integriert werden können.

```
Expression ::= `(` <Value> `)`
| <Value> `+` <Value>
| <Value> ` - ` <Value>
| <Value> `*` <Value>
| <Value> `/` <Value>
| ` - ` <Value>
| IDENTIFIER `(` <Values> `)`
```

A.2.2 Beispiel

Die folgende einfache Beispielszene besteht aus zwei Kugeln. Die Szene verwendet zur Illustration alle Schlüsselwörter von BSDL. Abbildung A.1 zeigt das Resultat, nachdem die Szenenbeschreibung mit Hilfe der Raytracer-Applikation verarbeitet wurde.

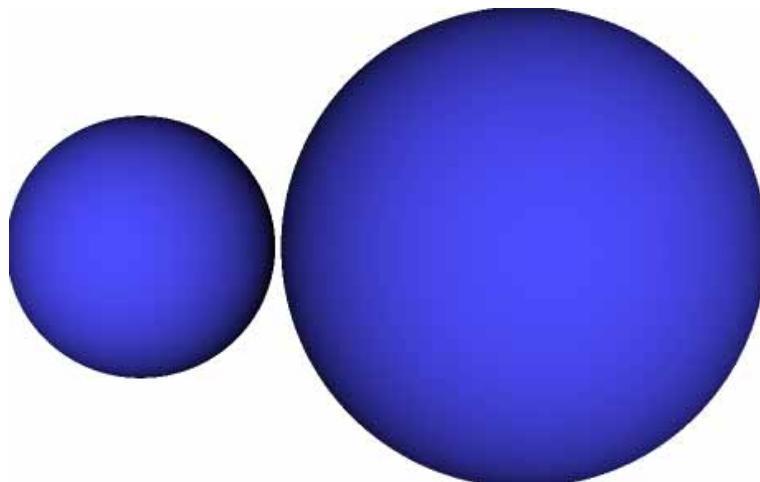


Abbildung A.1

Die einfache Beispielszene mit Hilfe eines Raytracers visualisiert.

```
using 3D; // Wir bewegen uns im 3D Namenraum.

camera { // Eine perspektivische Kamera.
perspective {
    eye [1, 1.5, 11]; // Position,
    lookat [1, 1.5, 0 ]; // Fokussierungsort und
    up [0, 1, 0 ]; // Orientierung der Kamera.
}
background [1, 1, 1];
```

```
}

// Ein eigener Namespace definieren.
define MyNS namespace;

// Texturdefinition.
define MyNS::kugelTextur phong {
    ambient [.1, .1, .3];
    diffuse [.3, .3, 1.];
}

// Der Kugelradius wird als Konstante definiert.
const RADIUS 1.4;

// Definition einer Punktlichtquelle mit Intensitaet 1
// und weisser Farbe.
pointLight (1, [1, 1, 1]) {
    position [0, 0, 100];
}

// 1. Kugel
sphere (RADIUS, [-1.5, 1.5, 0]) {
    MyNS::kugelTextur; // Definierte Textur verwenden.
}

// 2. Kugel mit Transformation.
sphere (RADIUS*1.8, [-1.5, 1.5, 0]) {
    translate [4, 0, 0];
    MyNS::kugelTextur;
}
```

A.3 Das **BOOGA**-Bildformat PIXI

PIXI, das Bildformat von **BOOGA** wird verwendet, um Rasterbilder zu speichern oder zwischen verschiedenen **BOOGA**-Anwendungen über Dateien oder UNIX-Pipes auszutauschen. Es ist eng an das frei erhältliche PPM-Format (Portable Pixmap Format) von Jef Poskanzer angelehnt.

Der wesentliche Unterschied zwischen den beiden Bildformaten ist der Umfang des beschreibbaren Farbraumes. Während PPM Farbbilder mit je 8 Bit für die Grundfarben Rot, Grün und Blau ablegt, verwendet PIXI hierfür je eine Fliesskommazahl der Grösse 32 Bit. Ausserdem können nicht nur Farbkanäle abgespeichert werden, sondern zusätzlich die vordefinierten Kanäle *Alpha*³ und *Depth*⁴ sowie weitere, benutzerspezifische Bildkanäle.

Bei auf 8 Bit begrenzten Bildformaten stellt sich das Problem der *Normierung* auf den möglichen Wertebereich: die Farbinformationen werden intern üblicherweise in wesentlich höherer Genauigkeit berechnet als für die Ausgabegeräte darstellbar. Zudem können die Werte ausserhalb des Intervalls [0, 1[(beziehungsweise des Ganzzahlbereichs zwischen 0 und 255) liegen.

Werden die Zwischenresultate in das für die Ausgabe notwendige Format umgerechnet, gehen Informationen verloren, die möglicherweise für die Weiterverarbeitung wichtig gewesen wären. Diese Überlegungen waren der Hauptgrund für die Entwicklung eines eigenen Dateiformates.

Die folgenden Ausführungen beschreiben das PIXI-dateiformat wiederum in der erweiterten Backus Naur Form (EBNF).

Eine PIXI-Datei besteht grundsätzlich aus einem Kopf und den eigentlichen Bildinformationen.

PixiFile ::= Header Image .

Der Kopf ist zeilenweise aufgebaut. Jede Teilinformation schliesst mit einem Zeilenendezeichen ab.

Header ::= MagicNumber Comment Size Range Channels .

Die Magic Number wird dazu verwendet, den Dateityp zu erkennen. Stimmen beim Einlesen diese ersten Byte der Datei nicht, wird eine Fehlermeldung ausgegeben und der Lesevorgang unterbrochen.

³Der *Alpha*-Kanal wird beispielsweise für *Blendingeffekte* verwendet. Er beschreibt den Anteil des sichtbaren Hintergrundes pro Pixel.

⁴Im *Depth*-Kanal kann die Tiefeninformation, also der Abstand der dem Pixel entsprechenden Objektoberfläche in der Szene. Diese Information kann beispielsweise für einen SIRDS-Renderer (vgl. [Ammon, 1996]) verwendet werden.

```
MagicNumber ::= "BOOGA Pixmap" NL.
```

Der Kommentar kann dazu verwendet werden, die 'Entstehungsgeschichte' des Bildes festzuhalten: welche Anwendungen wurden mit welchen Parametern verwendet.

```
Comment ::= { '#' Text NL }*.
Text    ::= { Char | Digit | White }*.
Char    ::= { 'a' | ... | 'z' | 'A' | ... | 'Z' }.
Digit   ::= { '0' ... '9' }.
White   ::= { ' ' | '\t' }.
```

Die Grösse des Bildes Size wird als Ganzzahl in ASCII Form ausgegeben.

```
Size    ::= Width ' ' Height NL.
Width   ::= { Digit }+.
Height  ::= { Digit }+.
```

Range bezeichnet den Wertebereich der Bildinformationen und dient einer allfälligen Skalierung beim Einlesen des Bildes.

```
Range  ::= Min Max NL.
Min    ::= ( { Digit }+ [ '.' ] { Digit }* ) | ( '.' { Digit }+ )
          [ ( 'e' | 'E' ) { Digit }+ ].
```

Der Header wird mit einer Liste der belegten Bildkanäle abgeschlossen. Die Kanäle 0 und 1 sind für den Alpha- und den Tiefenkanal reserviert.

```
Channels ::= { { Digit }+ }*
```

Das Bild selbst wird in Form von 4 Byte Fliesskommazahlen abgelegt. Zuerst kommen die Bilddaten in der Form Rot – Grün – Blau zeilenweise, anschliessend die verwendeten zusätzlichen Kanäle, ebenfalls zeilenweise.

Anhang B

Der *BOOGA*-Styleguide¹

¹Dieses Kapitel entspricht – mit kleinen Änderungen – dem während dem Projekt eingesetzten Dokument.

B.1 Einleitung

Ein Framework ist eine generische Softwarearchitektur und bestimmt als solche Analyse, Design und Implementierung der mit dem Framework erstellten Applikationen.

Der Softwareentwicklungsprozess wird vom Framework wesentlich beeinflusst. Die Softwareerstellung mit Frameworks unterscheidet sich somit stark von der Softwareerstellung mit herkömmlichen Software Engineering-Ansätzen.

Ein Framework beinhaltet nicht nur die Bibliotheken mit den Softwarezeugnissen, sondern auch eine Vorgehensweise zur Applikationsentwicklung und somit auch Richtlinien auf jeder Ebene des Applikationslebenszyklus.

Damit ein Framework einfach zu verwenden ist (und somit die Grundvoraussetzung erfüllt, um *überhaupt* verwendet zu werden), muss nicht nur die Anzahl der verschiedenen Protokolle klein gehalten werden (was vor allem die Analyse- und Designphase betrifft), sondern es müssen auch Richtlinien zum Programmierstil aufgestellt werden, die oft kleinlich anmuten, in ihrer Gesamtheit aber letztlich dem Anwender und Entwickler eines Frameworks und der mit diesem erstellten Applikationen entgegenkommen.

Das vorliegende Dokument beinhaltet vor allem die Richtlinien und Überlegungen der *Implementierungsphase*.

B.2 Allgemeines

Verstöße gegen eine Regel sind im Code zu kennzeichnen und zu begründen.

B.3 Aufbau einer Klasse

Die Klassendeklaration ist wie folgt aufgebaut:

1. `public`-Teil
2. `protected`-Teil
3. `private`-Teil

Es können mehrere `public`, `protected` und `private` Teile verwendet werden, um Methoden, Attribute oder verscheidene Aspekte der Funktionalität zu unterscheiden. Die Gruppierung von Attributen oder Methoden erfolgt durch Leerzeilen oder neue `public`, `protected` oder `private` Blöcke.

Attribute dürfen nicht `public` sein.

Ausnahmen von dieser Regel bilden Konstanten.

Nach Möglichkeit sollten Attribute im `private`-Teil deklariert werden, ausnahmsweise im `protected`-Teil.

```
class Application {  
public:  
    Application();  
    ~Application();  
  
    void run();  
  
protected:  
    void eventLoop();  
  
private:  
    EventRecord myLastMouseEvent;  
};
```

B.4 Namengebung

Eine konsistente und allgemein verbindliche Namengebung erlaubt eine wesentlich schnellere Einarbeitung in ein Framework. Neue Teammitglieder können schnell Code von verschiedenen Autoren verstehen.

Die folgenden Regeln sind
verbindlich für **BOOGA** Anwendungen.

Die Regeln werden im Abschnitt B.4.8 nochmals zusammengefasst.

B.4.1 Dateinamen

Folgende Suffixe sind für Dateinamen zu verwenden:

.c	C++-Source
.h	C++-Header
.l	Lex-Source
.y	Yacc-Source

Normalerweise sollten pro Klasse ein gleichnamiges Source-/Header-Dateipaar erzeugt werden.

B.4.2 Typen

Namen von Typen beginnen mit einem Grossbuchstaben. Neue Wörter innerhalb des Namens beginnen mit einem Grossbuchstaben und werden *nicht* mittels `_` abgetrennt.

Folgende Präfixes können Typnamen vorangestellt werden:

M	Mixins
E	Enumerations
T	Template Parameter

B.4.3 Templates

Namen von Templates sollten im allgemeinen mit einer Präposition enden:
Beispiele:

- `ListOf<Object3D*>;`
- `CommandOn<World>;`

B.4.4 Methoden

Methodennamen beginnen mit einem Kleinbuchstaben. Neue Wörter innerhalb des Namens beginnen mit einem Grossbuchstaben und werden *nicht* mittels `_` abgetrennt.

„Getters“ – Methoden, die Werte (von Attributen oder berechnete) liefern – und „Setters“ – Methoden, die Werte setzen – werden mit Präfixes versehen.

B.4.4.1 „Getters“

- create** Methoden, die ein neues Objekt erzeugen, das der Aufrufer selbst löschen muss (mittels `delete`).
- copy** Methoden, die ein existierendes Objekt kopieren. Der Aufrufer ist dafür verantwortlich, dass die erzeugte Kopie gelöscht wird (mittels `delete`).
- orphan** Methoden, die die Verantwortlichkeit für ein alloziertes Objekt *abgeben*. Der Aufrufer ist dafür verantwortlich, dass die das Objekt gelöscht wird (mittels `delete`).
- get** Alle anderen Fälle von „Getters“.
- is** Methoden, die einen boole'schen Rückgabewert haben.

B.4.4.2 „Setters“

Der Rückgabewert von „Setters“ ist `void`.

- adopt** Methoden, die die Verantwortung für ein vom Aufrufer alloziertes Objekt übernehmen. Bei Konstruktoren sollten solche *Parameter* mit dem Präfix `adopt` versehen werden.
- set** Alle anderen Fälle von „Setters“.

Es ist fehlerhaft, wenn in einer `set`-Methode der Parameter kopiert und die Kopie einem Attribut zugewiesen wird. In diesem Fall ist die Verwendung einer `adopt`-Methode korrekt.

B.4.5 Attribute

Alle Attribute werden mit dem Präfix `my` versehen, `static` Attribute mit dem Präfix `our`.

```
class Foo {
public:
    Foo();

private:
    int myData;
    static float ourData;
};
```

B.4.6 Parameter, Variablen

Variablennamen beginnen mit einem Kleinbuchstaben. Neue Wörter innerhalb des Namens beginnen mit einem Grossbuchstaben und werden *nicht* mittels `_` abgetrennt.

B.4.7 Konstanten

Konstantennamen werden ausschliesslich mit Grossbuchstaben geschrieben. Neue Wörter innerhalb des Namens werden mittels `_` abgetrennt.

B.4.8 Übersicht

Identifier	Convention	Example
Typen	Beginnen mit einem Grossbuchstaben	ConfigurationHandler
Mixin Klassen	Präfix M	MNotifiable
Enumerations	Präfix E	ESharedFlag
Methoden	Beginnen mit einem Kleinbuchstaben	transformAsPoint()
Attribute	Präfix my	myRefractionIndex
Statische Attribute	Präfix our	ourErrorStream
Variablen	Beginnen mit einem Kleinbuchstaben	returnValue
Konstanten	Alles Grossbuchstaben	MAGIC_NUMBER
„Getters“	Präfixe create, copy, orphan, get, is	copyCamera(), isDone()
„Setters“	Präfixe adopt, set	adoptMatrix()

Tabelle B.1

Übersicht der Konventionen
zur Namengebung in
BOOGA

B.5 Codierkonventionen

B.5.1 Globale Variablen

Globale Variablen sollten nicht verwendet werden.

B.5.2 Parameter

- Arrayparameter sollten immer mit der Arrayschreibweise deklariert werden.

Beispiel: `int foo(int a[]);`

B.5.3 Datentypen

`bool`

Der (in neueren Compilern eingebaute) Datentyp `bool` mit den vordefinierten Konstanten `true` und `false` ist der Verwendung von `int` oder Enumerations bei boole'schen Werten vorzuziehen.

`Real`

Für Fliesskomma-Arithmetik sollte der Datentyp `Real` verwendet werden.

Standard Datentypen Es ist besser, eigene Datentypen zu deklarieren, als die eingebauten Datentypen direkt zu verwenden.

Beispiele:

Vermeiden	Besser
<code>long time;</code>	<code>typedef long TimeStamp;</code> <code>TimeStamp time;</code>
<code>short mouseX;</code>	<code>typedef short Coordinate;</code> <code>Coordinate mouseX;</code>

Anhang C

Beispiel einer auf Patterns basierenden Dokumentation

Bearbeiten von Objekttypen durch Komponenten

Absicht

Definieren der Objekttypen (konkret oder abstrakt), die von einer Komponente bearbeitet werden.

Motivation

Eine Instanz einer speziellen Traversierungsklasse (vgl. ‘**Implementierung einer Traversierungsklasse**’) ist für das korrekte Durchlaufen des Szenengraphen zuständig. Dabei wird jedes Objekt, das angetroffen wird, der Komponente übergeben und dort bearbeitet. Diese muss nun entscheiden, was damit zu geschehen hat und wie die Traversierung weitergehen soll. Soll das Objekt verarbeitet werden, muss eine entsprechende Methode implementiert werden.

Umsetzung

Grundsätzlich findet für jede Komponente eine Aufgabenteilung statt: die Methode `dispatch` entscheidet, ob das aktuelle Objekt von der Komponente bearbeitet wird. Für jeden Objekttyp, der von der Komponente unterstützt wird, existiert eine `visit`-Methode, die den Algorithmus enthält, wie das Objekt verarbeitet wird.

Die `dispatch`-Methode

Die `dispatch`-Methode wird stets wie folgt *deklariert*:

```
public:  
    virtual Traversal::Result dispatch(Makeable* obj);
```

Für die Implementierung dieser Methode stehen zwei *Präprozessormakros* zur Verfügung. Je nachdem, ob es sich bei dem zu verarbeitenden Typ um ein konkretes geometrisches Objekt wie beispielsweise `Line3D` oder `Sphere3D` handelt oder um eine abstrakte Oberklasse wie `Object3D` oder `Primitive3D` wird entweder `tryConcrete` oder `tryAbstract` verwendet.

Die Reihenfolge der Makroaufrufe ist äusserst wichtig, da jedes geometrische Objekt durch *einen* konkreten Typ und *mehrere* abstrakte Typen beschrieben wird. Die Makroaufrufe müssen stets vom speziellen zum allgemeineren Typ erfolgen. Trifft eine Bedingung zu, so wird die entsprechende `visit`-Methode aufgerufen und deren Rückgabewert als Resultatwert von `dispatch` verwendet. Nach einer erfolgreichen `try`-Bedingung wird die Bearbeitung der `dispatch`-Methode abgebrochen. Trifft keine der Bedingungen zu, gibt die Methode den Wert `Traversal::UNKNOWN` zurück, um dem Traversierungsalgorithmus so mitzuteilen, dass die Komponente das übergebene Objekt nicht hat bearbeiten können.

```
Traversal::Result ComponentName::dispatch(Makeable* obj)  
{  
    tryConcrete(Concrete1, obj);
```

```

    tryConcrete(Concrete2, obj);
    tryAbstract(Abstract1, obj);
    tryAbstract(Abstract2, obj);

    return Traversal::UNKNOWN;
}

```

Die visit-Methode

Die Deklaration einer visit-Methode ist abhängig vom Typ (konkret oder abstrakt) des geometrischen Objektes, das verarbeitet werden soll:

```

private:
    Traversal::Result visit(ObjectType* obj);

```

Die Implementierung von visit entspricht stets dem folgenden Schema:

```

Traversal::Result ComponentName::visit(ObjectType* obj)
{
    // ...

    return Traversal::CONTINUE;
}

```

In der Regel wird als Resultatwert Traversal::CONTINUE zurückgegeben. Dies führt dazu, dass die Traversierung des Szenengraphen gemäss dem verwendeten Traversierungsalgorithmus normal weitergeführt wird. Weitere Alternativen werden in ‘**Die Traversierung des Szenengraphs**’ beschrieben.

Beispiel

Als Beispiel soll eine Komponente realisiert werden, die nur Dreiecke (**Triangle3D**) verarbeiten kann. Die Aufgabe liegt darin, die Eckpunkte eines Dreiecks sowie die zugehörigen Normalen in Weltkoordinaten umgerechnet auszugeben, zudem eine Meldung, wenn ein Objekt zerlegt werden muss.

```

1 class Triangulator : public Operation3D {
2 declareRTTI(Triangulator);           // enable RTTI support
3
4 //~~~~~ Constructors, destructors, assignment ~~~~~
5 // Constructors, destructors, assignment
6 //~~~~~ Constructors, destructors, assignment ~~~~~
7 public:
8     Triangulator(ostream& os = cout);
9 private:
10    Triangulator(const Triangulator&);          // No copies.
11
12 public:
13     // virtual ~Triangulator();                 // Use default version.
14
15 private:

```

```

16     Triangulator& operator=(const Triangulator&);    // No assign.
17
18 //~~~~~ New methods of class Triangulator ~~~~~
19 //~~~~~ From class Visitor ~~~~~
20 //~~~~~ My data members ~~~~~
21 private:
22     Traversal::Result visit(Triangle3D* obj);
23     Traversal::Result visit(Object3D*   obj);
24
25 //~~~~~ From class Visitor ~~~~~
26 //~~~~~ My data members ~~~~~
27
28 public:
29     virtual Traversal::Result dispatch(Makeable* obj);
30
31 //~~~~~ My data members ~~~~~
32
33
34 private:
35     ostream& myStream;
36 };

```

Der erste Teil (Zeilen 1 bis 17) stellt die Klasseninfrastruktur zur Verfügung. Interessanter sind die beiden darauffolgenden Teile (Zeilen 18 bis 24 und Zeilen 25 bis 30). Auf Zeile 29 wird die `dispatch`-Methode wie besprochen deklariert, auf den Zeilen 22 und 23 die nötigen `visit`-Methoden.

Im folgenden werden nur die Implementierungen der `dispatch` und `visit`-Methoden genauer beschrieben.

`dispatch` wird wie zuvor beschrieben implementiert. Die neue Komponente versteht genau einen konkreten Objekttyp (`Triangle3D`). Instanzen einer anderen, von der abstrakten Basisklasse `Object3D` abgeleiteten Klasse, werden durch den Aufruf `tryAbstract(Object3D, obj)` behandelt.

```

1 Traversal::Result Triangulator::dispatch(Makeable* obj)
2 {
3     tryConcrete(Triangle3D, obj);
4     tryAbstract(Object3D,   obj);
5
6     //
7     // Create decomposition for objects other than triangles.
8     //
9     return Traversal::UNKNOWN;
10 }

```

Als nächstes wird die `visit`-Methode für Dreiecke implementiert. Diese Methode ist recht einfach zu realisieren: auf Zeile 3 wird die aktuelle Transformationsmatrix bestimmt, anschliessend das Dreieck in der Form

`triangle($v_1, n_1, v_2, n_2, v_3, n_3$)`

ausgegeben, wobei v_i einen Eckpunkt (*Vertex*) und n_i eine Normale bezeichnet. Eckpunkt und Normale werden als Vektoren in der Form

[x, y, z]

geschrieben.

```
1 Traversal::Result Triangulator::visit(Triangle3D* obj)
2 {
3     Transform3D transMatrix = getTraversal()->getPath()-> getLastTransform();
4
5     myStream << "triangle (" 
6         << transMatrix.transformAsPoint(obj->getVertex(0)) << ", "
7         << transMatrix.transformAsNormal(obj->getNormal(0)) << ", "
8         << transMatrix.transformAsPoint(obj->getVertex(1)) << ", "
9         << transMatrix.transformAsNormal(obj->getNormal(1)) << ", "
10        << transMatrix.transformAsPoint(obj->getVertex(2)) << ", "
11        << transMatrix.transformAsNormal(obj->getNormal(2))
12        << ");" << endl;
13
14     return Traversal::CONTINUE;
15 }
```

Schliesslich wird noch die `visit`-Methode für ein `Object3D` realisiert. Der Typ des unbekannten Objektes wird ausgegeben und anschliessend durch den Rückgabewert `Traversal::UNKNOWN` eine Zerlegung in andere Objekte ausgelöst.

```
1 Traversal::Result Triangulator::visit(Object3D* obj)
2 {
3     myStream << "// Decomposing " << obj->getTypeId_().name() << endl;
4
5     return Traversal::UNKNOWN;
6 }
```


Ironic

*An old man turned ninety-eight
He won the lottery and died the next day
It's a black fly in your Chardonnay
It's a death row pardon two minutes too late
Isn't it ironic . . . don't you think*

*It's like rain on your wedding day
It's a free ride when you've already paid
It's the good advice that you just didn't take
Who would've thought . . . it figures*

*Mr. Play It Safe was afraid of fly
He packed his suitcase and kissed his kids good-bye
He waited his whole damn life to take that flight
And as the plane crashed down he thought
"Well isn't this nice . . ."
And isn't it ironic . . . don't you think*

It's like rain on your wedding day . . .

*Well life has a funny way of sneaking up on you
When you think everything's okay and everything's going right
And life has a funny way of helping you out when
You think everything's gone wrong and everything blows up
In your face*

*A traffic jam when you're already late
A no-smoking sign on your cigarette break
It's like ten thousand spoons when all you need is a knife
It's meeting the man of my dreams
And then meeting his beautiful wife
And isn't it ironic . . . don't you think
A little too ironic . . . and yeah I really do think . . .*

It's like rain on your wedding day . . .

*Life has a funny way of sneaking up on you
Life has a funny, funny way of helping you out
Helping you out*

Alanis Morissette

Curriculum Vitae

Am 28. Oktober 1967 wurde ich als Sohn der Gisela Amann, geborene Hautmann und des Günther Amann in Regensburg (BRD) geboren. Aufgewachsen bin ich in Interlaken (BE).

Nach Beendigung der obligatorischen Schulzeit in Interlaken trat ich 1983 ins dortige Gymnasium ein, welches ich 1986 mit der Maturitätsprüfung, Typus C abschloss. Es folgte ein Zwischenjahr, in dem ich erste Erfahrungen im Erwerbsleben machte sowie die Rekrutenschule absolvierte.

Im Herbst 1987 begann ich meine universitäre Ausbildung. Das Informatikstudium mit den Nebenfächern Mathematik und Betriebswirtschaftslehre schloss ich 1994 mit einer Diplomarbeit bei Herrn Prof. Dr. H. Bieri ab. Der Titel meiner Diplomarbeit lautet "*RADSHADE - Implementation eines globalen Beleuchtungsmodells basierend auf Radiosity und Raytracing*".

Seit Anfang 1993 bin ich als selbständiger Berater und Dozent für objektorientierte Technologien in Industrie und Dienstleistungsunternehmen tätig.

Im Anschluss an meine Diplomarbeit begann ich mein Doktorstudium in Informatik. Vom Frühjahr 1994 bis im Sommer 1995 war ich als Assistent am Institut für Informatik und angewandte Mathematik der Universität Bern bei Herrn Prof. Dr. H. Bieri angestellt. In dieser Zeit erhielt ich einen Lehrauftrag der Universität Bern für die Spezialvorlesung "Einführung in die objektorientierte Programmierung mit C++".

Seit dem Sommer 1995 bin ich bei der ASCOM Tech AG in Bern als Informatiker mit Aufgaben im Bereich objektorientierter Softwareentwicklung betraut.

Ich lebe zur Zeit am Holzikofenweg in Bern.

