Universität–Gesamthochschule Siegen **Technische Informatik**
Prof. Dr.–Ing. H. Wojtkowiak Dr.–Ing. K. Quibeldey–Cirkel

Informatik–Proseminar im Sommersemester 1995

Entwurfsmuster

Software–Dokumentation mit Entwurfsmustern

Georg Odenthal

Geisweiderstraße 21
57078 Siegen

Siegen, 18.7.1995

1 Einleitung

Eine mittlerweile mehr als fünf Jahre alte Untersuchung, die von Coad/Yourdon in [Coad+91], S.157 zitiert wird und die die amerikanischen Software–Firmen in fünf Klassen unterteilt, will ich hier nochmals anführen, weil sie, so meine ich, erschreckend ist. Erschreckend, wenn man bedenkt, in welch hohem Maß unser Leben und unsere Zukunft von der Informationstechnologie abhängen und wie wenig sich in diesen fünf Jahren verändert hat.

> **Initial level (level 1)**: There is no formal method, no consistency, no standards on how systems should be built. Every software developer considers him– or herself an artist: anarchy prevails.
>
> **Repeatable level (level 2)**: There is a consensus within the organization about *„the way we do things around here“*, but it has not been formalized or written down. The systems development process is statistically stable through rigorous management of costs and schedules; however, success depends on the intuitive skills of project managers.
>
> **Managed level (level 3)**: There is a formal, documented process for developing systems. Software inspections are rigorously practiced, and configuration management is more advanced than at level 2. There is a '*software process group*' that constantly refines and updates the organization's methods.
>
> **Measured level (level 4)**: The organization has instituted formal process measurements – often referred to as '*software metrics*' to measure its process for building systems, as well as the resulting products.
>
> **Optimizing level (level 5)**: The organization uses the measurements from level 4 as a '*feedback*' mechanism to improve those parts of its process that are found to be weak or deficient.

Die höheren Ebenen zeichnen sich durch ein wachsendes Maß an Organisation in der Kontrolle und Dokumentation des Software–Entwicklungszyklus aus. Die Autoren bemerken, daß ein sinnvoller Einsatz von Werkzeugen, die den Entwicklungsprozeß unterstützen (Software–Entwicklungsumgebungen und Entwurfsmethoden), zumindest eine Einstufung des Unternehmens auf Ebene drei voraussetzt.
Nun befinden sich aber 87 % der großen US–Software–Firmen auf Ebene eins. 10 – 12 % sind auf Ebene zwei und ca. 1 % auf Ebene drei. Keine Firma befand sich Ende 1990 auf Ebene vier oder fünf.
An diesen Ergebnissen läßt sich abschätzen, warum es bei der Einführung von CASE in bestehende Unternehmensstrukturen so große Probleme gibt, warum die Software–Krise auch noch nicht einmal im Ansatz gelöst ist.
Unser Berufsstand hat hier hohe Verantwortung zu tragen, weil wir es sind, die die Entwicklung maßgeblich vorantreiben und weil wir zu den wenigen gehören, die sie einschätzen können . . . ?!
Die Beschäftigung mit der Theorie und der Praxis des Software–Entwicklungsprozesses ist eine der wichtigsten fachspezifischen Lerninhalte des Informatikstudiums. Auf diesem Gebiet ist noch sehr vieles im argen; innovative Kräfte sind gefragt.
Zu den neusten Entwicklungen im Bereich Software–Engineering zählen zweifellos die Entwurfsmuster. Im Verein mit einem frameworkbasierten Entwurfsprozeß stellen sie einen leistungsfähigen Ansatz dar, der das Objekt–Paradigma handhabbar werden läßt. An der rasanten Entwicklung im Bereich Frameworks kann man dies beobachten: Die meisten Software–Schnittstellen (API) graphischer Benutzeroberflächen werden dem Anwender durch Application Frameworks zugänglich gemacht. Das MVC–Entwurfsmuster [Kras+88], über Jahre in der Mottenkiste der Software–Theoretiker verschwunden, ist in aller Munde. Die Software–Industrie und die von ihr getragenen Konsortien, wie z.B. die OMG (*Object*

Management Group, http://www.omg.org) oder die CI Labs (*Component Integration Laboratories*, http://www.cilabs.org) entwickeln Standards, die in Form von Frameworks (CORBA mit SOM/DSOM, OpenDoc, Talingent Frameworks – siehe Glossar) umgesetzt werden.

2 Software–Dokumentation

Die Dokumentation von Entscheidungen und Abläufen innerhalb eines Software–Projekts stellt den wohl wichtigsten Arbeitsinhalt eines Software–Ingenieurs dar und ist trotzdem bis in heutige Zeit nicht in das Zentrum der Betrachtung gerückt. Weiterhin wird um Implementation (Sourcecode ist auch ein Software–Dokument, aber ein im Verhältnis zu den vorhergehenden weniger wichtiges) und Implementationssprachen gestritten. In den letzten Jahren kam zumindest die Diskussion um Entwurfs–Methoden auf – ein Schritt in die richtige Richtung. Des Pudels Kern wurde und wird aber weiterhin weitgehend verdrängt – vielleicht aus der Abneigung vieler gegen (vordergründige) Unschärfe? Oder weil die „Produktion von Fließtext“ schon für so manchen in der Schule eine unangenehme Tätigkeit war?
Die im Verlauf eines Projekts generierten Dokumente dienen als Grundlage der Kommunikation der Projektbeteiligten, zur Orientierung über den Projektstand für das Management und nicht zuletzt entscheidet sich an der Qualität der entstandenen Dokumente die Wiederverwendbarkeit von Entwürfen in anderen Projekten.

2.1 Qualitätsmerkmale für Software–Dokumentation

Abgesehen von subjektiven Beurteilungen der Dokumentationsqualität, die auch durch ganz unabhängige Parameter, wie DV–Erfahrung oder den allgemeinen Wissensstand über eine Problemdomäne oder das Bildungsniveau beeinflußt werden, gibt es doch eine Reihe von objektiven Qualitätsmerkmalen, die ich im folgenden hier aufführen möchte. (vgl. [Lehn94], S. 117ff)

- **Korrektheit**. Gemeint ist einerseits sprachliche Richtigkeit (Grammatik, Satzbau) und wichtiger, die sachliche Richtigkeit, die Übereinstimmung von beschriebenem und realem Sachverhalt.
- **Vollständigkeit**. Unterschieden in **formale** Vollständigkeit, die vorliegt wenn alle geforderten und/oder in Verzeichnissen aufgeführten Bestandteile vorhanden sind und in **sachliche** Vollständigkeit: Alle zum Gegenstandsbereich der Dokumentation gehörenden Sachverhalte und Komponenten sind beschrieben.
- **Konsistenz**. Wiederum zwei Bedeutungen: Konsistenz in einem lokalen Sinne, d.h. Widerspruchsfreiheit und Vermeidung von Mehrdeutigkeiten. Eher global bezüglich einer zusammengehörigen Dokumentation meint Konsistenz auch die Einhaltung einer einheitlichen Begriffswelt, einer einheitlichen Notation, sowie eines durchgängigen Layouts.
- **Verständlichkeit und Lesbarkeit**. Hier geht es um Forderungen an den in einem Dokument verwendeten Wortschatz – er sollte an das erwartete Wissen des Lesers angepaßt sein. Desweiteren sollten neue Begriffe vor ihrer Verwendung definiert werden. Der Satzbau sollte generell einfach sein. Wichtig ist die sinnvolle Reihenfolge der Informationsdarstellung, so daß z.B. Grundlagen zuerst dargestellt werden und der Schwierigkeitsgrad sich steigert. Die Verwendung von Beispielen zur Erläuterung von abstrakten Sachverhalten ist eine weitere Technik.

Die Deutsche Gesellschaft für Qualität (DGQ) hat 1986 folgende Liste von Qualitätsmerkmalen definiert, die ich ergänzend hier aufführen möchte, wobei ich auf oben schon erläuterte Begriffe nicht nochmals näher eingehe. (vgl. [Lehn94], S.119ff)

- **Änderbarkeit**. Erweiterungen und Änderungen an Software–Dokumenten sollen mit geringem Aufwand eingebracht werden können.

- **Aktualität**. Die Dokumentation soll die aktuelle Version eines Programmes beschreiben. Unter der Sichtweise, daß alle Produkte eines Software–Projektes Dokumente sind, ist diese Anforderung in der obigen Konsistenzforderung enthalten.
- **Eindeutigkeit und Einheitlichkeit**.
- **Identifizierbarkeit**. Eindeutige Ansprechbarkeit von Dokumenten oder Teilen davon, die Angaben zu einem festgelegten Sachverhalt enthalten. Hiermit ist wohl auch die Forderung nach vollständigen und ausreichend umfangreichen Inhaltsverzeichnissen und Indexen gemeint.
  Weiterhin die eindeutige Auszeichnung eines Dokumentes mit Namen des Autors, Erstellungs– und Änderungsdatum und Angaben zur Version.
- **Normenkonformität**. Einhaltung von (firmeninternen) Standards und allgemein üblichen Darstellungsmethoden.
- **Verständlichkeit.**
- **Vollständigkeit.**
- **Widerspruchsfreiheit.**

Diese Auflistungen erheben einerseits keinen Anspruch auf Vollständigkeit und – gravierender – definieren kein Verfahren zur Qualitätsprüfung, da sie nur eine Wunschliste darstellen und die Kriterien nicht formal überprüfbar sind. Der Faktor Mensch bleibt der wichtigste Garant für die Qualität der Software–Dokumentation. Anhand solcher Forderungskataloge können Vorgehensmodelle zur Software–Dokumentation allerdings abgeglichen werden.

2.2 Verfahren zur Software–Dokumentation

Es gibt eine Fülle von Darstellungsarten für die im Bereich Software–Entwicklung anfallenden Informationen. Diese Techniken lassen sich zunächst in *bildhafte* und *symbolische Darstellung* unterteilen (vgl. [Lehn94], S. 16f). Die bildhafte Darstellung durch zwei oder dreidimensionale Visualisierung von Informationen hat bisher keine große Verwendung im Bereich der Software–Entwicklung gefunden oder ist noch Gegenstand der Forschung.
Die symbolische Darstellung läßt sich weiter in *künstliche* und *natürliche Sprachen* unterteilen. Erstere faßt alle Spezifikationssprachen und mehr oder weniger graphische Notationen zusammen. Meist wird zur Software–Dokumentation eine Mischung aus natürlicher und künstlicher Sprache eingesetzt. Die Entwurfs–Methoden von Strukturierter Analyse (SADT) bis hin zur *Object Modeling Technique* (OMT) fassen jeweils eine Menge von symbolischen Darstellungstechniken zusammen. Eine Auswahl bekannter Techniken:

- Zustandsdiagramm und Petrinetz
- Datenflußdiagramm
- Entscheidungstabelle
- Struktogramm
- Entity–Relationship–Diagramm
- Klassendiagramme nach Coad/Yourdon, Booch oder Rumbaugh
- Objektdiagramme
- Szenarios bzw. Interaktionsdiagramme

Die OMT von Rumbaugh (vgl. [Rum+93]) beispielsweise faßt Klassen–, Zustands– und Datenflußdiagramm zusammen. In Software–Entwicklungsumgebungen wird ein Satz von Diagrammen dann rechnerunterstützt in Form von Werkzeugen angeboten. Die auf natürlicher

Sprache basierende Dokumentation wird nicht oder nur unzureichend und ausschließlich als optionale Annotation an Komponenten unterstützt.
Nach [Lehn94], S.18 werden die Verfahren der Software–Dokumentation oft nur im Kontext der Systemanalyse und der Spezifikation betrachtet. Die Betrachtung der Dokumentation als Grundlage des gesamten Entwicklungsprozesses tritt dabei stark in den Hintergrund. Erst mit der Verfügbarkeit leistungsfähigerer Software–Entwicklungsumgebungen, die den gesamten Software–Lebenszyklus unterstützen, ist eine umfassendere und integrierende Betrachtung der Software–Dokumentation zu beobachten: Durch die Werkzeugunterstützung wachsen die bisher getrennt betrachteten Bereiche Analyse/Entwurf mit formalen Methoden und die Dokumentation der Entwurfsergebnisse zusammen.

3 Hypertext

Der Begriff Hypertext taucht in vielen Veröffentlichungen über Entwurfsmuster auf (z.B. [Pree94], [John92], [Gam+95]). Dies liegt sicherlich an der den Mustern inhärenten Anforderung nach einer netzartigen Darstellung. Die Autoren sehen in diesem Mittel die im Moment adäquate Technik zur rechnergestützten Verwendung von Entwurfsmustern.
Ich möchte im folgenden auf Hypertext als Dokumentationsmedium genauer eingehen. Hierzu werde ich von einem kurzen Abriß über die Historie, der Einführung der wesentlichen Begriffe zu einer generellen Software–Architekturbeschreibung von Hypertextsystemen kommen. Zentral wird dann die Beleuchtung der Eignung von Hypertext für die Software–Dokumentation sein – einhergehend mit Beschreibungen zu Navigationstechniken im Hyperraum und der Referierung wissenschaftlicher Auswertungen zur Akzeptanz und Qualität von Hyperdokumenten.
Schließlich werde ich auch die neuere und meiner Meinung nach für die Softwaredokumentation wegweisende Erweiterung Hypermedia ansprechen.

3.1 Historie eines modernen Begriffes

Hypertext ist eine Idee, die auf eine 50–jährige Geschichte zurückblickt. Erstaunlich vor allem, wenn man die allgemeine Begriffsinflation verbunden mit der extremen Kurzlebigkeit im Computersektor bedenkt. Desweiteren ist der Bekanntheitsgrad des Begriffes unvergleichlich gering im Verhältnis zu seinem doch ehrwürdigen Alter. Das liegt natürlich auch daran, daß hypertextbasierte Werkzeuge vereint mit den in den letzten zehn Jahren aufgekommenen graphischen Benutzeroberflächen quasi durch die Hintertür und ohne große Erläuterungen das Licht einer größeren Öffentlichkeit erblickten: zuerst in Form von Online–Hilfen, im weiteren Verlauf dann elektronische Bücher und mittlerweile als vierteljährlich aktualisierte und weltweit vertriebene Software–Dokumentations–Bibliotheken (*Microsoft Developer Network*). In diesem Verlauf hat sich die Informationsmenge vervielfacht und die Benutzerschnittstelle grundlegend erweitert. Zunächst ein Blick zurück anhand [Niel93] und [Gloor90].

3.1.1 Memex

Roosevelts Präsidentenberater Vanevar Bush veröffentlichte in einem Artikel 1945 das mechanische System *Memex* – *„a sort of mechanized private file and library“*, *„a device in which an individual stores his books, records and communications“* ([Niel93], S.31f). Die Datenbasis sollte auf Mikrofilm gespeichert, der Zugriff über Photozellen parallel auf mehrere Dokumente realisiert und das ganze im Schreibtisch eingebaut werden. Informationen jeder Art sollten auf Mikrofilm erhältlich sein und eigene Dokumente durch einen Scanner integriert werden. Die Informationen sollten untereinander verknüpfbar sein. – Wie man weiß, das System wurde nie realisiert und ist mit heutiger Technik auch wohl noch nicht realisierbar.

3.1.2 Augment/NLS

Zwanzig Jahre später, die Computertechnik war noch zu teuer, um sie für andere Zwecke als zur numerischen Berechnung einzusetzen, befaßte sich Doug Engelbart im Rahmen des Augment/NLS–Projektes des Stanford Research Institute mit Büroautomatisierung und Textverarbeitung ([Niel93], S.34f), setzte die von ihm erfundene Maus als Eingabegerät ein und entwickelte Applikationen mit Fenstertechnik. Im *NLS–System* (oN Line System) wurde die Projektdokumentation der Arbeitsgruppe mit insgesamt mehr als 100.000 Elementen (items) verwaltet, die durch die Definition von Querverweisen vernetzt werden konnten.
Engelbart gilt als Initiator des *interactive computing*.

3.1.3 Xanadu und FRESS

Ted Nelson prägte 1965 durch sein *Xanadu*–Projekt das Wort Hypertext. Xanadu ist eine Vision von der Weltliteraturdatanbank, die alles was jemals jemand schrieb enthält und so einen universalen Hypertext darstellen würde ([Gloor90], S.10; [Niel93], S.35).
Andries van Dam hat 1967 an der Brown University das weltweit erste arbeitende Hypertext–System entwicklet, das dann zur Dokumentation von Apollo–Missionen am Houston Manned Spacecraft Center eingesetzt wurde.
An der Brown University wurde der Nachfolger *FRESS* (File Retrieval and Editing Sytsem) und das Intermedia System ([Niel93], S.102ff) entwickelt.

3.1.4 Aspen Movie Map

Das erste Hypermedia–System (siehe unten), die *Aspen Movie Map* wurde 1978 von Andrew Lippmann am MIT entwickelt ([Niel93], S.38f). Der Benutzer kann sich auf eine virtuelle Fahrt durch die Stadt Aspen begeben. Die Darstellung besteht aus einzelnen Photographien, die durch Links so miteinander verbunden sind, daß eine natürliche Navigation durch die Straßen der Stadt ermöglicht wird.
Natürlich stand der militärische Nutzen dieses Systems wieder mal im Vordergrund: Training von Soldaten in fremdem Terrain.

3.1.5 Symbolics Document Examiner

Ein weiterer wichtiger Meilenstein war 1985 der *Symbolics Document Examiner*, der auch eines der ersten Systeme war, die nicht im universitären oder im in–house–Rahmen einer Firma blieben, sondern unter kommerziellen Gesichtspunkten entstanden. Dieses Hypertext–System stellte das Interface zur Online–Dokumentation der Symbolics–Workstations dar. Die Anwender nutzten es, nicht weil es eine Hypertext–System war, sondern weil es der einfachste Weg war, auf die System–Dokumentation zuzugreifen. Die 8.000 Seiten mitgelieferter Papierdokumentation waren so alternativ in einem 10.000 Knoten und 23.000 Links umfassenden Hypertext gespeichert. Eine Auswertung unter 24 Nutzern der Workstation ergab, das 2 ausschließlich die Papierdokumentation benutzten, die Hälfte nur das Hypertext–Dokument und acht die Papierdokumentation noch nicht einmal ausgepackt hatten ([Niel93], S.42).

3.1.6 HyperCard

Den Durchbruch für Hypertext brachte zweifellos 1987 HyperCard von Apple, daß mit jedem Rechner ausgeliefert wurde und für das ein respektabler Markt von Anbietern entstand, die „Stacks“ (Dokumente und/oder Applikationen) für HyperCard entwickelten.

3.1.7 Und so weiter

Mittlerweile sind die Hyper–Technologien etabliert. Jeder Computernutzer wendet sie täglich intuitiv an. Durch das Internet und *World Wide Web* (WWW) rückt selbst Nelsons Vision in erreichbare Nähe, wenn es auch keine Weltliteraturdatenbank aber doch ein Weltmarktplatz werden wird – mit allen Vor– und Nachteilen.

3.2 Hypertext–Systeme: Definition und Anforderungen

Um Hypertext als Medium einschätzen zu können, ist es wichtig einige grundlegende Begriffe zu kennen. Ich möchte diese im folgenden anführen.
Ein konventioneller Text besteht aus einer Sequenz von Worten, Sätzen, Absätzen, Kapiteln, usw. und ist somit hierarchisch und streng sequentiell strukturiert. Diese Struktur resultiert natürlich auch aus der engen Verbindung zum Medium Papier. Die Zugriffsfunktionalität (ich wähle bewußt diesen Begriff, da es hier um die Anforderungsdefinition von Benutzerfunktio-

nalität gehen soll) besteht aus dem Inhaltsverzeichnis, dem Index und der manuellen sequentiellen Suche evtl. mit mehreren Startpunkten (Lesezeichen). Liegt ein sequentielles Dokument in elektronischer Form vor, so kann man auch noch die automatische Volltextsuche hinzuzählen.

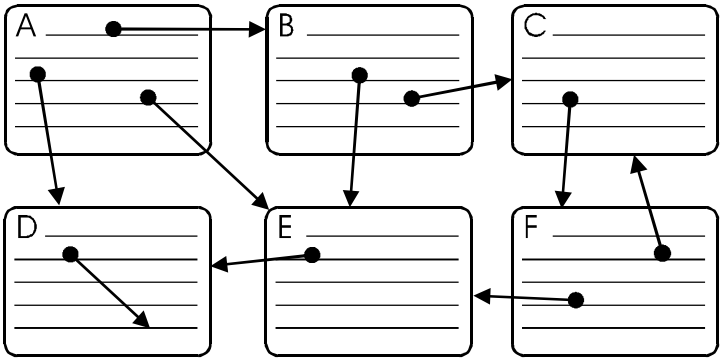

Abbildung 1 (vgl. [Niel93], S.1): Vereinfachte Sicht auf eine kleine Hypertext–Struktur.

Hypertext ist zumindest ab einer sinnvollen Hierarchieebene *nicht–sequentiell* strukturiert, d.h. ab der Satzebene (meist wohl erst ab der Absatzebene oder darüber) wird die sequentielle Abfolge aufgebrochen. Die entstehenden Textfragmente werden als **Knoten** bezeichnet und können durch **Links** untereinander verbunden werden. Es entsteht ein Graph aus Knoten und Kanten (Abbildung 1) wobei zumindest die Knoten, in die Links münden können, mit Namen versehen sein sollten (beispielsweise der Abschnittsüberschrift), da die meisten Navigationstechniken (siehe Abschnitt3.3) Knotennamen benötigen.
Aus der Nicht–Sequentialität von Hyperdokumenten muß nicht das Nicht–Vorhandensein einer übergreifenden Struktur folgen: Im Allgemeinen haben Hypertexte ein Inhaltsverzeichnis und einen Pfad, der das sequentielle Durchblättern der Information erlaubt. Dies insbesondere, wenn der Text ein eng zusammenhängendes Thema beschreibt (z.B. ein elektronisches Handbuch).
Bezüglich des Umfanges eines Knotens ist es wichtig, maximal die auf eine Bildschirmseite passende Information darzustellen (weiteres siehe Abschnitt3.4).

3.2.1 Linktypen

Verschiedene Linktypen sind gebräuchlich und noch mehr denkbar bzw. je nach Anwendungsfall sinnvoll, da auch verschiedene Knotentypen denkbar sind.

- Unidirektionaler Link von einem Knoten zu einem anderen.
- Link von einem Teil eines Knotens (ein Wort oder ein Satzfragment) zu einem anderen Knoten.
- Link zu einem ausführbaren Knoten.
- Implizite Links (Vorwärts/Rückwärtsblättern, Rückwärtsreferenz in das Inhaltsverzeichnis).
- Replacement Link: Das Ausgangsfragment wird durch das Zielfragment des Links ersetzt.
- Annotation Link: Anzeige einer weiterführenden Erläuterung in Form eines weiteren Knotens (Analogie: Fußnote) ohne den Kontext zu verlassen.
- Link, der nur einen Teil (z.B. nur einen Satz) eines größeren Knotens referenziert.
- Automatisch vom Hypertext–System vergebene Links die aufgrund einer vorgegebenen Liste von Wörtern und Zielen definiert werden.
- Zur Realisierung der Versionierung von Hypertexten die Linktypen Link–to–current, Link–to–specific, um auf die aktuellen und alten Versionen eines Knotens zu referenzieren ([Gloor90], S.72).

Eine andere Klassifizierung der Links in sieben Gruppen wird in [Gloor90], S.157 vorgestellt. Danach gibt es Links der Typen:

- *being* *A is–a B*,
- *showing* *A is an example of B* oder *A demonstrates B*,
- *causing* *A causes B* oder *A is a result of B*,
- *using* *A uses B*,
- *having* *A has B*,
- *including* *A includes B* und
- *similarity* *A is similar to B* oder *A can replace B*.

3.2.2 Zugriffsfunktionalität

Der Zugriff auf einen Hypertext kann ungleich vielfältiger gestaltet werden, als auf einen sequentiellen Text, da außer den oben aufgezählten Methoden für sequentielle Texte jetzt noch folgende hinzukommen können:

- Definition verschiedener vorgegebener Pfade durch ein Hypertext–Dokument.
- Anzeige der schon besuchten, bzw. noch nicht besuchten Knoten.
- Anzeige des Rückweges von einem Knoten über die Folge der schon besuchten Knoten.
- Ein– und Ausblenden von Knoten zur Verringerung der Komplexität, bzw. Vertiefung des Informationsgehaltes.
- Unterschiedliche Verknüpfung der Knoten um verschiedenen Nutzungsarten gerecht zu werden.

Im Abschnitt 3.3 gehe ich genauer auf verbreitete Navigationstechniken im Hyperraum ein.

3.2.3 Allgemeine Anforderungen an Hypertext–Systeme

Abschließend ergeben sich einige allgemeine Anforderungen an ein Hypertext–System:

- Realisierung einer für den Problembereich und den Umfang des Hypertextes adäquaten **Zugriffsfunktionalität**.
- Zumindest zwei **Benutzermodelle**: Autoren, die auf den Hypertext selbst und die Links modifizierenden Zugriff haben und Leser, die nur navigieren können. Wünschenswert ist weiterhin Leser–Funktionalität zur Annotation bestehender Hypertexte mit Anmerkungen und evtl. auch die Definition eigener Links.
- Unterstützung von **Gruppenarbeit** mit ausreichenden Locking–Mechanismen und Zugriffsschutz.
- **Versionsverwaltung** der Knoten und der Topologie.
- **Einheitliche Benutzerführung** und ausreichend schnelle Reaktion des Systems auf Anforderungen.

3.3 Navigation im Hyperraum

Meist werden zwei Kritikpunkte an Hypertextsystemen festgemacht: 1.) Das Lesen am Bildschirm ist unangenehm und nicht so effizient wie auf dem Papier und 2.) die Gefahr, sich in den vernetzten Informationen zu verirren (*„lost in Hyperspace“*, vgl. [Niel93], S.133) ohne die eigentlich wichtigen Informationen zu finden, da man sich nie sicher sein kann, alle Knoten besucht zu haben. Den zweiten Kritikpunkt möchte ich in diesem Abschnitt näher untersuchen, auf den ersten werde ich in Abschnitt3.4 eingehen.
Man muß hier zunächst anführen, daß die Orientierung in großen Dokumenten generell schwierig ist und es gerade bei Hypertext oft um sehr große Informationsmengen geht. Des-

weiteren liegt noch nicht allzuviel Erfahrung in der Gestaltung von Hypertexten (und erst recht von Hypermedia–Dokumenten) vor, als das man die gleiche Qualität, wie man sie bei anderen Medien gewohnt ist, erwarten kann. Umgekehrt werden nach über 500 Jahren Buchdruck immer noch strukturell schlecht aufgebaute Bücher herausgebracht (vom Inhaltlichen einmal abgesehen) und nach hundert Jahren bewegtem Bild immer noch schlechte Filme. Es muß hier auch die einzelne Produktion bewertet werden. Die Werkzeuge zur Erstellung von Hyperdokumenten werden sicherlich auch noch einige Evolutionsschritte erfahren.
Darüber hinaus ist ein Hypertext einfach aus anderen Anforderungen heraus als ein konventioneller Text entstanden. Es sind ja gerade die Möglichkeiten, die Papier nicht hat, die Hypertext so interessant machen. Und um diese Möglichkeiten zu nutzen, ist es natürlich erforderlich sich mit der grundlegenden Philosophie zu befassen und die Navigationstechniken zu kennen und anzuwenden.
Ich möchte im folgenden beschreiben, welche Navigationstechniken es für den Hyperraum gibt. Im wesentlichen mündet diese Auflistung in eine Anforderungsbeschreibung an die (Zugriffs–)Funktionalität der Benutzerschnittstelle eines Hypertext–Systems.
Intuitive und effiziente Bedienbarkeit, verschiedene Bedienungstechniken für unterschiedliche Benutzerprofile und unterschiedliche Anforderungen sind wünschenswerte Eigenschaften. Insgesamt betrachtet stellt die Qualität der Navigationstechniken das wesentliche Maß für die Bewertung eines Hypertext–Systems dar.

3.3.1 Inhaltsverzeichnis

Der Zugriff auf Informationen über ein Inhaltsverzeichnis ist die vertrauteste Art der Navigation in einem konventionellen Text und auch für Hypertexte – gerade auch wegen der Bekanntheit des Verfahrens – nicht minder geeignet.

3.3.2 Volltext– und Indexsuche

Der Benutzer eines Hypertext–Systems ist oft in der Situation, daß ihm der angebotene Informationsraum gänzlich oder in Teilen unbekannt ist und eine bestimmte Information gesucht wird. In diesem Fall kann die Navigation durch Volltext– oder Indexsuche sinnvoll sein.
Die von konventionellen Texten bekannten Indices werden auch in Hypertextsystemen unterstützt. Darüber hinaus sollte auch eine knotenübergreifende globale Suche nach beliebigen Textfragmenten mit den bekannten logischen Verknüpfungsoperatoren angeboten werden. Das Suchergebnis wird als Trefferliste angezeigt und der Benutzer kann nacheinander die entsprechenden Knoten besuchen.
Weitergehende Techniken, die mit unscharfer Suche arbeiten, einen Thesaurus zur Synonymsuche einsetzen oder Ähnlichkeiten zwischen Knoten gewichten sind denkbar und vom Umfang und Art des Hyperdokumentes abhängig.
Die verfügbare Rechnerleistung begrenzt diese Art der Navigation. Die bei *Microsoft Visual C++ V2.0* ([Micro94]) mitgelieferte Online–Dokumentation bietet unter *Windows NT* beispielsweise die Volltextsuche an. Die Indexinformationen sind dabei in einer ca. 60 MB großen Datei gespeichert – additiv zum eigentlichen Hilfetext von mehreren Megabyte Größe.

3.3.3 Guided Tours

Das einfachste Verfahren, dem Benutzer die Navigation zu erleichtern, besteht darin, sie aus seinen Händen zu nehmen und einen vorgegebenen Pfad anzubieten. Abhängig von der Art der Information und den voraussetzbaren Kenntnissen des Anwenders kann dies eine sinnvolle Funktion sein. Vor allem die Erstbenutzung des Dokumentes kann so stark vereinfacht werden. Die übrige Navigationsfunktionlität kann in diesem Modus ausgeblendet sein oder dem Benutzer optional zur Verfügung stehen. So kann er den Vorgabepfad auch verlassen und wieder dahin zurückkehren.

3.3.4 Backtracking

Eine wichtige Benutzerfunktion ist die Rückkehr auf dem gegangenen Pfad hin zum Ausgangspunkt. Diese Funktion sollte immer auf die gleiche Weise zur Verfügung stehen. Dabei ist es wichtig, daß der Benutzer alle besuchten Knoten genau so wieder vorfindet, wie er sie über einen Link verlassen hat. Diese Forderung bezieht sich auch auf die Stellung von Scrollbalken, die Position von Fenstern oder den Status selektierter Felder. So ist die Wiedererkennung gegeben und der Nutzer kann sich wesentlich besser orientieren.
Eine Erweiterung dieser Funktionalität stellen **History Lists** dar. Anhand derer kann der Nutzer direkt zu einem bestimmten Knoten zurückgehen, ohne alle auf dem zurückgelegten Pfad zwischenliegenden nochmals besuchen zu müssen.
Der sogenannte **Visual Cache** ist eine weitere Möglichkeit, das Umfeld eines Pfades darzustellen. Hierbei werden minimisierte oder ikonisierte Darstellungen der zuletzt besuchten Knoten in einem separaten Fenster angezeigt und der Anwender kann durch die Maus auswählen.

3.3.5 Lesezeichen

Diese konventionelle Technik wurde sinnvollerweise in den Hyperraum übertragen, mit der Erweiterung, daß dem Anwender eine **Bookmark List** zur Verfügung steht, um von dort aus zu den verschiedenen Lesezeichen zu verzweigen.

3.3.6 Annotationen und Anmerkungen

Der Benutzer sollte die Möglichkeit haben, das Hyperdokument mit Anmerkungen zu versehen, die möglichst intuitiv darstellbar sind. Hierzu sollten auch handschriftliche Anmerkungen in Form von Unterstreichung etc. zählen, die z.B. durch ein Digitalisiertablett eingegeben werden. Ferner die Definition neuer benutzerdefinierter Knoten und Links.
Diese Veränderungen am Hyperdokument sollten in einem Mehrbenutzersystem entsprechend in ihrer Sichtbarkeit einschränkbar sein und sich generell in der Darstellung von den ursprünglichen Knoten und Kanten unterscheiden.

3.3.7 Maps

Um eine individuelle Navigation im Gegensatz zu den oben erwähnten *Guided Tours* zu ermöglichen, können dem Anwender Landkarten, die (normalerweise nur) einen Ausschnitt aus der Topologie des Netzwerks aus Knoten und Links darstellen, angezeigt werden. Der bisher zurückgelegte Weg sollte der Benutzer darstellen können.
Liegt dem Hypertext eine **Hierarchie** zugrunde, kann die Darstellung des Netzausschnittes auch noch in unterschiedlicher Granularität erfolgen. Hierdurch verringert sich die Anzahl der gleichzeitig sichtbaren Knoten, da nur die auf einer bestimmten Hierarchieebene befindlichen (Meta–) Knoten angezeigt werden und auch nur die dann sichtbaren Links. Eine weitere Hilfe erhält der Anwender durch **Größenangaben** (z.B. anhand der Anzahl der Wörter eines Textes) von Knoten und übergeordneten Metaknoten.
Die Visualisierung muß nicht in Form eines Graphen, sondern kann auch durch **dynamische Inhaltsverzeichnisse** (Verzeichnisebenen lassen sich beliebig ein– und ausblenden) erfolgen. Letztere sind kompakter und schneller in der Darstellung, da so auf eine zweidimensionale graphische Anzeige verzichtet werden kann. Gute Beispiele kann man in einigen Online–Hilfen von Windows und OS/2 finden.
Zukünftig wird eine immer besser animierte Darstellung bis hin zu synthetischen Navigationsräumen entwickelt werden; diese erlauben dann wieder eine auf realen Metaphern basierende Suche, beispielsweise das virtuelle Buch, die virtuelle Bibliothek und andere dreidimensionale Darstellungen.

3.3.8 Animation

Durch Animationstechniken, die mit einer graphischen Benutzerschnittstelle realisierbar sind, kann zum Beispiel ein unterschiedliches Verhalten beim Blättern über die Seiten des Textes eines Knotens und beim Wechsel von einem Knoten zu einem anderen durch einen Link dargestellt werden.
Verschiedene Schriftattribute und Fensterhintergründe können darüber hinaus die intuitive Orientierung für den Benutzer nochmals verbessern.

3.3.9 Navigation in Hypermedia–Systemen

Stehen außer dem Medium Text noch Graphik und Bewegtbilder zur Darstellung von Information zur Verfügung, kommen noch weitere und meist wesentlich intuitivere Navigationstechniken in Frage: Selektierbare Bereiche in Graphiken und auch Bewegtbildern (**Hotspots** genannt und nicht mit denen von Pree [Pree94] zu verwechseln) dienen als Ausgangspunkte für Links oder zur Steuerung einer Animation. Diese Techniken haben sicherlich ihre Grenzen, vor allem was die dadurch explorierbare Menge von Knoten angeht. Sie besitzen aber durch die Intuitivität immense Vorteile in bestimmten Anwendungsfällen, in denen keine Voraussetzungen an das Benutzerprofil angestellt werden können oder die Bedienung beispielsweise nur durch einen Touchscreen erfolgen kann.

3.4 Ein Vergleich von Dokumentationsmedien

Einer der größten Kritikpunkte am Medium Hypertext (vgl. Abschnitt 3.3) ist sicherlich die ausschließliche Repräsentation auf einem Computersystem, da sie sich zumindest nicht vollständig auf ein sequentielles Medium übertragen lassen. Daraus entsteht die Notwendigkeit, die Texte auf dem Bildschirm zu lesen.
Bevor ich einige Auswertungen aus der Literatur hierzu beschreibe, möchte ich bemerken, daß meiner Meinung nach die Diskussion am Wesentlichen vorbeigeht. Die Vorteile, die in der Repräsentation von Informationen als Hyperdokument bestehen, wiegen die Nachteile der Lesbarkeit am Bildschirm mehr als auf. Gut entworfene Hyperdokumente überfordern den Leser nicht mit (Bildschirm–)seitenlangen Texten; nicht umsonst wird von der Verwendung von Scrollbalken für Hypertextsysteme abgeraten oder empfohlen, die maximale kognitive Größe eines Hypertext–Knotens auf 10 Sätze Text zu beschränken ([Gloor90], S.216). Informationssammlungen eignen sich mehr oder weniger gut für die Umsetzung in ein Hyperdokument, bzw. hängt es von der Art der Verwendung ab, ob die eine oder andere Repräsentation vorzuziehen ist. Es gibt z.B. mittlerweile das Manuskript eines Kafka–Romanes als Hypertext auf CD–ROM. Für den Literaturwissenschaftler sicherlich eine völlig neue Dimension und große Erleichterung bei der Analyse eines solchen Textes. Den Roman in dieser Form zu lesen ist nur Technikfreaks anzuraten.

3.4.1 Testergebnisse

Zunächst Ergebnisse von Tests, die die Lesegeschwindigkeit untersucht haben. Laut [Gloor90], S.123ff ist sie beim Lesen auf dem Bildschirm um ca. 30 % geringer als beim Lesen von Papier. Beim Einsatz von besseren Bildschirmen mit 91 dpi Auflösung (das war 1987, heute sind mindestens 96 dpi Standard) und mit *anti–aliasing* (Technik zur Verringerung der Treppenstruktur, die durch die Rasterung in Pixel entsteht) dargestellten Fonts verringert sich der Abstand auf nahezu Null. Analoges gilt für wachsende Bildschirmdiagonalen. Die schnellere Ermüdung vor dem Bildschirm bleibt allerdings bestehen.
Untersuchungen, die Nielsen in [Niel93], S.156ff zitiert, und die sich auf den Vergleich von Hypertext und konventionellem Blättern in einer Datei und den Vergleich beim Repetieren von Informationen aus einem Text, der in Papierform und als Hypertext vorlag, lassen auf keinen

Vorteil von Hypertext schließen. Die Zeit, in der die Rezipienten eine vorgegebene Aufgabe lösten, war immer in etwa gleich oder sogar schlechter für den Hypertext. Allerdings unterstelle ich diesen Untersuchungen, daß sie nicht mit hinreichend großen Datenmengen angestellt wurden. Einmal waren es Zeitungsartikel, ein anderes Mal ein immerhin 138 Seiten langer Text.
Leider hatte ich nicht die Möglichkeit, nach weiteren Untersuchungen zu recherchieren.
Ein Gegenbeispiel ist z.B. eine zitierte Umfrage ([Niel93], S.160) unter 16 Studenten, die eine Enzyklopädie (also ausreichend umfangreich) mit äquivalentem Inhalt als Hypertext und als Papierversion benutzten. Nach einem Vergleich befragt sagte die Hälfte, daß die elektronische Version schneller sei, drei, daß mehr Informationen enthalten wären und einer, daß die Informationen aktueller wären.
Zuguterletzt noch eine Untersuchung ([Niel93], S.161), die die Qualität der Benutzerschnittstelle mit einbezog: Die Hypertext–Version einer Sherlock Holmes Enzyklopädie wurde mit der Papierversion verglichen. Die Anwender kamen mit letzterer besser und schneller zurecht. Nachdem die Benutzerschnittstelle verbessert wurde, kehrte sich dieser Zustand um.

3.4.2 Online–Dokumentation

Die von Lehner ([Lehn94], S.54) bemerkte oberflächliche Einschätzung, daß eine Online–Dokumentation prinzipiell nur ein anderes Medium als Papier darstellt, erweist sich bei genauerer Betrachtung als falsch, wenn nicht grundlegende strukturelle Änderungen des in Papierform vorliegenden Textes vor der Transformation vorgenommen werden.
Ein gutes Negativbeispiel hierfür ist der *Adobe Acrobat Reader*, wie er unter vielen graphischen Benutzerschnittstellen verfügbar ist. In verschiedenen Applikationen wird dieses Werkzeug zur einfachen Transformation der gedruckten Benutzerdokumentationen in elektronische Handbücher verwendet. Das Layout der Bücher bleibt dabei annähernd gleich, einzig die Volltextsuche erweist sich als Vorteil. Das System bietet die Möglichkeit Links zu definieren, diese Mühe machen sich wohl aber die meisten Anbieter nicht (zu verifizieren auf der *Adobe Acrobat CD Sampler* [Adob94]). Die Lesbarkeit der am Bildschirm dargestellten Seiten ist unbefriedigend; der Eindruck einer aufgeschlagenen Doppelseite eines Buches ist nicht darstellbar. Fazit: Vor allem für umfangreichere Dokumentationen ist dieses Werkzeug nicht geeignet.
Lehner stellt in [Lehn94], S.55ff Qualitätsmerkmale von Online–Benutzerdokumentationen dar. Ich möchte diese Aufstellung an dieser Stelle gerafft wiedergeben, da sie eine gute Ergänzung zu den oben angeführten Anforderungen an Hypertext–Systeme sind:

- **„Visuelle Erfaßbarkeit**: Im allgemeinen sind Darstellungen am Bildschirm für den Benutzer schlechter und langsamer als ein gedrucktes Handbuch. Daher muß die visuelle Erfaßbarkeit für die Bildschirminhalte **optimiert** werden [...].
- **Informationsgehalt**: Der Informationsgehalt von Bildschirminhalten muß auf Grund der begrenzten Darstellungskapazität dieser darauf abgestimmt werden. Man kann davon ausgehen, daß eine gedruckte Seite etwa dem Umfang von drei Bildschirmseiten entspricht. Das bedeutet, daß z.B. die **Redundanz**, welche ein wichtiges Kriterium für ein Benutzerhandbuch darstellt, in diesem Falle weitgehend reduziert werden sollte, um die begrenzt verfügbare Kapazität optimal ausnutzen zu können.
- **Zugriffsmöglichkeit**: Durch die Möglichkeit des Online–Zugriffs auf gewünschte Informationen in **komfortabler** und **schneller** Form steigt der Nutzen eines Hilfesystems.
- **Art der Aktivierung der Hilfeleistung**: Hierbei sind zwei verschiedene Möglichkeiten gegeben: Bei der ersten Variante wird die Hilfeleistung explizit vom Benutzer gefordert; in diesem Falle spricht man von **passiver Hilfe**. Bei der zweiten Variante wird ein Hilfetext

automatisch ohne Aktivierung durch den Benutzer angezeigt (z.B. bei fehlerhafter Benutzereingabe); in diesem Fall spricht man von **aktiver Hilfe**. Das Problem bei der aktiven Hilfeleistung ist es, das Wissensdefizit des Anwenders zu erkennen und einen geeigneten Zeitpunkt zur Darstellung der Hilfeinformation zu finden, ohne den Benutzer bei der laufenden Arbeit zu stören [...].

- **Benutzerbezogenheit**: Auch bei diesem Merkmal werden zwei Varianten unterschieden: Bei der ersten spricht man von einer **standardisierten Hilfeleistung**, weil die angezeigte Hilfeinformation für alle Anwender die gleiche ist. Bei der **individuellen Hilfeleistung** wird auf die Fähigkeit des Anwenders eingegangen und eine dem Benutzer angepaßte Hilfeinformation am Bildschirm angezeigt [...].
- **Kontextsensitivität**: Ist die angezeigte Hilfeinformation unabhängig von der derzeitigen Dialogsituation immer gleich, so spricht man von einer **statischen Hilfeleistung**, währenddessen eine **dynamische Hilfeleistung** (kontextsensitive Hilfe) sich in Abhängigkeit zur aktuellen Dialogsituation ändert [...]“.

Aus dieser Aufstellung werden indirekt auch die unterschiedlichen Möglichkeiten eines qualitativ hochwertigen Hypertext–Systems gegenüber der konventionellen Papierform deutlich.

3.5 HyperHyper . . . Hypermedia

Wenn man von Hypermedia spricht wird meist Multimedia damit in Verbindung gebracht. Dieser inflationär verwendete Begriff, der wohl nur deshalb noch nicht Unwort des Jahres wurde, weil er (vordergründig) zu unpolitisch ist, steht aber nur sehr eingeschränkt mit Hypermedia in Beziehung.
Hypermedia ist als Erweiterung von Hypertext um weitere Medien zu sehen: Bewegtbild, Graphik und Ton können sinnvolle neue Möglichkeiten bieten, Informationen darzustellen, wenn sie richtig eingesetzt werden. Multimedia hingegen umschreibt nur eine spezielle Facette von Informationssystemen, die auf Hypermedia–Techniken basieren, aber nur eine sehr eingeschränkte Funktionalität bieten, da sie vornehmlich für den Privatanwender und auf leichte intuitive Bedienbarkeit ausgelegt sind. Zumindest alle Aspekte, die sich auf die Modifikation von Hyperdokumenten beziehen, befinden sich nicht im Kontext von Multimedia. Desweiteren versammelt sich unter diesem Begriff auch der gesamte Kanon der technischen Mittel, um Multimedia zu realisieren: die Datenautobahn, alle nötige Computer–Peripherie.
Bezüglich des Einsatzes von Hypermedia–Systemen im Bereich Software–Entwicklung sehe ich die Entwicklung erst am Anfang. Simulation von Abläufen und generell die Visualisierung sind Techniken, die gerade in diesem Bereich von größtem Interesse sind.

3.6 Hypertext und CASE

Die Anwender von Hypertextsystemen im Bereich der Software–Entwicklung können als mit der Bedienung von Programmen bestens vertraut angesehen werden und es ist somit möglich, ein umfangreiches und gut ausgestattetes Hypertext–System für diesen Bereich zu fordern.
Meiner Meinung nach ist es wichtig den iterativen Charakter der Entwurfstätigkeit möglichst gut in einem Dokumentationswerkzeug zu implementieren. Kein Mensch kann ein komplexes Problem durch die sequentielle und wohlstrukturierte Niederschrift der nötigen Entwurfsideen und Abstraktionen lösen. Deshalb ist die durch Hypertext mögliche schrittweise Verfeinerung eines Dokumentes und die durch entsprechende Editoren erreichbare Flexibilität in der Darstellung der Informationen ideal für den Prozeß der Software–Entwicklung geeignet.

3.6.1 Software–Architektur von Hypertextsystemen

In [Gloor90], S.107ff wird eine Drei–Schichten–Architektur eines Hypertext–Systems beschrieben:

- **Präsentationsschicht**: Die Darstellung der Informationen mit einer Benutzerschnittstelle. Hier werden die von der unterliegenden Schicht gelieferten Daten interpretiert. Ein wesentlicher Aspekt ist die unterschiedliche Funktionalität des Systems für Autoren und Benutzer (zwei Bedienmodi).
- **Hypertext Abstract Machine** (HAM): Verwaltung der Knoten und Kanten. Dies umfaßt sicherlich eine Aufbereitung der Daten in eine Hauptspeicherrepräsentation um die Zugriffsgeschwindigkeit zu steigern. Weiterhin werden hier die bei Mehrbenutzersystemen und Versionsverwaltung hinzukommenden Daten kontrolliert.
  Nielsen ([Niel93], S.108ff) beschreibt Bestrebungen auf dieser Ebene eine Standardisierung zum Datenaustausch zwischen verschiedenen Hypertextsystemen zu implementieren.
- **Datenbankschicht**: Realisierung der Datenspeicherung und evtl. der Verteilung der Daten auf einem Netzwerk. Auf dieser Ebene ist noch kein Aspekt der logischen Datenanordnung als Hypertext implementiert. Generell entspricht diese Ebene der unteren Ebene im ANSI/Sparc Schichtenmodell (vgl. [Vos94], S.24) für eine Datenbank–Architektur.

Meiner Meinung nach wäre es sinnvoll, die beschriebene Datenbankschicht im ANSI/Sparc–Modell auf die konzeptionelle Ebene zu verlagern, d.h. ein Datenmodell für ein Hypertext–System zu entwickeln, das es ermöglicht, ein konventionelles oder objektorientiertes DBS zur Speicherung der Daten heranzuziehen.
Die HAM wäre dann zu einem Teil eine Datenbankapplikation, die Benutzeranfragen (hier z.B. *„gehe zu Knoten k“*, *„definiere Link l vom Linktyp t von Konten k1 nach Knoten k2“*) und auch Teile der Systemsteuerung (z.B. *„Zeige nur Knoten und Kanten von Benutzer b“*, *„Zeige nur Kanten die nach dem 5.5.95 definiert wurden“*) in Datenbankabfragen umsetzt (siehe nächster Abschnitt).

3.6.2 Ein Vorschlag für die Grobarchitektur einer Software–Entwicklungsumgebung

Folgend ein Vorschlag für die Grobarchitektur eines Basissystems einer Software–Entwicklungsumgebung, die Hyperdokumente als integralen Bestandteil einsetzt:

- **Objektbank** zur konsistenzerhaltenden Verwaltung aller Knoten und Beziehungen, die zusammengenommen alle Dokumente eines klassischen Projekts ergeben, die sich jetzt als ein **Hyper–Verbunddokument** darstellen. Die verschiedenen Dokumenttypen (Texte, Sourcecode, Graphiken, Simulationen, ausführbarer Code, Testdaten etc.) aber auch feingranularere Einheiten, wie z.B. Klassen, Attribute, Methoden, Relationen, beliebige Annotationen etc. werden alle als unterschiedliche Knotentypen realisiert. Der Zugriff auf die Daten in der Objektbank muß mit **variabler Granularität** möglich sein. Diese Anforderung läßt sich durch **rekursiv schachtelbare Knoten** erfüllen.
  Versionsverwaltung und Mehrbenutzerzugriff sind integraler Bestandteil der Objektbank.
- Ein **Softwarebus**, der um die für die Generierung und Verwaltung von und den Zugriff auf Hyperdokumente nötigen Funktionalität erweitert ist und desweiteren die zur Integration von Werkzeugen nötigen Mechanismen und Protokolle bietet. Eine wichtige Forderung ist die flexible Integration neuer Dokumentationstechniken und Entwurfsmethoden in die Umgebung. Hierzu ist die Bereitstellung von weitreichend konfigurierbaren Editoren (generative Applikationen) und einer leistungsfähigen Applikationsbeschreibungssprache nötig.

- Alle wünschenswerten **Werkzeuge** für die Unterstützung eines Vorgehensmodells, für die Navigation im Hyperdokument und für die Dokumentation. Werkzeuge arbeiten nur auf den für sie zuständigen Knotentypen, ein Übersetzer also z.B. nur auf den für ihn zuständigen Sourcecodes. Durch die Steuerung des Mehrbenutzerzugriffs kann ein Anwender nur die für ihn freigegeben Knoten modifizieren. Die Versionsverwaltung erlaubt auf der Werkzeugebene die schrittweise Verfeinerung eines Dokuments und die Erzeugung von älteren Zuständen des Hyperdokuments.

Ein Beispiel ist das in [Gloor90], S.25f erwähnte Datenbanksystem *Neptune* von Tektronix, das für die Verwaltung von CAD– und CASE–Dokumenten entwickelt wurde und die Hypertext–Funktionalität hierzu in Form der Hypertext Abstract Machine (HAM) enthält. Das System bietet Versionsverwaltung und Mehrbenutzerzugriff. Alle in einem Softwareprojekt anfallenden Dokumente werden verwaltet.
Ein System, das den Einsatz von Hypertext in der Analysephase umsetzt ist *gIBIS* (graphical Issue Based Information System) der amerikanischen Firma MCC, ein Mehrbenutzer–Hypertext–Sytem zur Dokumentation von Positionen/Meinungen und Argumenten zu Designfragen im Software–Entwicklungsprozeß (vgl. [Niel93], S.49f).

3.7 Cooperative Authoring: Distributed Hypertext

Eine unbedingte Voraussetzung für den Einsatz eines Hypertext–Systems zur Software–Dokumentation ist die Unterstützung des Mehrbenutzerzugriffes mit einem auf Knotenebene installierten Locking–Mechanismus, einer Attributierung von Knoten und Links mit Benutzerrechten und einer Versionsverwaltung.
Hypertext besitzt eine inhärente Eignung für die Unterstützung der Synchronisation von Mehrbenutzerzugriffen, da die Partitionierung der Daten auf die Knoten keine aufwendigeren Mechanismen zum Sperren von Bereichen einer Datei erfordert. Auch die Abstraktion von einer physischen Verteilung der Daten in heterogenen Netzwerken läßt sich mit Hyperdokumenten einfacher Realisieren.
Ein Problem stellt die stetige Veränderung der Topologie des Informationsraumes für den einzelnen Anwender aufgrund der Modifikationen von Links durch andere Anwender dar.
Hier müssen geeignete Techniken und/oder Arbeitsorganisationsverfahren eingesetzt werden, z.B. die oben erwähnte Attributierung mit Benutzerrechten oder die Einschränkung des sichtbaren Bereiches für den einzelnen Anwender.
Die Versionsverwaltung wird schon in Single–user Systemen relevant, da durch Veränderung von Knoteninhalten oder die Aufspaltung eines Knotens in mehrere Knoten die Zuordnung der eintreffenden Links inkonsistent werden kann. Der Benutzer muß in solchen Fällen vom System auf die Erfordernis einer Kontrolle der Links hingewiesen werden (vgl. [Niel93], S.178f).

4 Anwendungen

Nach der Anforderungsbeschreibung an Software–Dokumentation generell und die nähere Betrachtung von Hypertext als Dokumentationsmedium untersuche ich nun konkrete Anwendungen: Die Software–Entwicklungsumgebung objectiF (Microtool, Berlin) bezüglich ihrer Verwendbarkeit zur Software–Dokumentation und das Autorensystem ToolBook (Asymetrix, München) auf die Erweiterbarkeit hin zu einem Hypertext–System. Danach werde ich Vorgehensmodelle angeben, die zumindet aufzeigen, was für die Software–Dokumentation mittels Hypertext zu beachten ist.

4.1 Die Integration eines Dokumentationswerkzeuges in eine Software–Entwicklungsumgebung am Beispiel objectiF

Viele bekannte und aktuelle Software–Entwicklungsumgebungen (SEU) unterstützen die Software–Dokumentation nicht oder nur sehr unzureichend. Dem Anwender werden eine oder mehrere Entwurfsmethoden in Form von Grapheneditoren angeboten. Automatische Code–Generierung und Reverse–Engineering sind oft integriert.
Software–Dokumentation, so scheint es, wird durch rechnerunterstützte Entwurfsmethoden überflüssig. Dies suggerieren auch die bekannten Methodenpäpste, da sie in ihren Büchern nichts oder nur an untergeordneter Stelle über Software–Dokumentation schreiben. Beispielsweise wird in dem Buch von Rumbaugh zu seiner OMT–Methode [Rum+93] auf knapp einer von insgesamt 601 Seiten explizit über den Sinn von Software–Dokumentation berichtet.
An der SEU objectiF der Berliner Firma *Microtool* möchte ich exemplarisch die Funktionalität zur Software–Dokumentation darstellen und Erweiterungsvorschläge im Bereich Design–Dokumentation mit Patterns (Abschnitt 4.1) vorstellen.

4.1.1 objectiF aus der Nähe betrachtet

objectiF, im Prospekt als objektorientierte Software–Entwicklungsumgebung bezeichnet, ist eine mehrbenutzerfähige, unter Windows laufende SEU, die (momentan noch) auf der OOA/D–Methode von Coad/Yourdon basiert und den Anwender von der Analyse bis zur Implementation in C++ unterstützt. Microtool setzt dabei umfassend auf die Unterstützung der Microsoft–Produkte. So enthält objectiF in seiner Objektbank die Microsoft–eigene Klassenbibliothek MFC als Thema (engl. *subject*: Menge von Klassendefinitionen), so daß der Entwickler in der Designphase sofort darauf aufsetzen kann, unterstützt explizit den Microsoft C++–Compiler und verwendet WinWord für die Dokumentation.
Die wesentlichen Komponenten sind:

- Die Objektbank, ein objektorientiertes Datenbankmanagementsystem (OODBMS) aus Eigenentwicklung, das alle Daten eines Projekts speichert.
- Der Klassenmodell–Editor zur Modellierung statischer Klassenhierarchien.
- Ein Editor für Zustandsdiagramme zur Modellierung des dynamischen Objektverhaltens.
- Die Nutzung von Microsoft Word als OLE–Server zur Generierung von Dokumenten.
- Für die Generierung von C++–Code ein Codegenerator.
- Ein syntaxgesteuerter Sourcecode–Editor, der intensiven Gebrauch von den Informationen aus der Objektbank macht, um Informationen über Bezeichner zu liefern, über die korrekte Verwendung von Variablen zu wachen und Implementationsvorschläge anzubringen.

Die Konsistenz wird dank der Objektbank auch im Mehrbenutzerbetrieb zwischen den verschiedenen Dokumenten automatisch erhalten. Näherer Betrachtung bedarf es der Anbindung von objectiF an Word, da hier versucht wird, eine leistungsfähige Textverarbeitung als Do-

kumentationswerkzeug in die SEU zu integrieren. Auf eine Eigenentwicklung (wie bei der Objektbank) haben die Entwickler hier verzichtet.
Die folgende Abbildung 2 ist ein Screenshot des Klassenmodell–Editors mit dem *Observer–Pattern* von Gamma ([Gam+95], S. 293ff) als Beispiel:

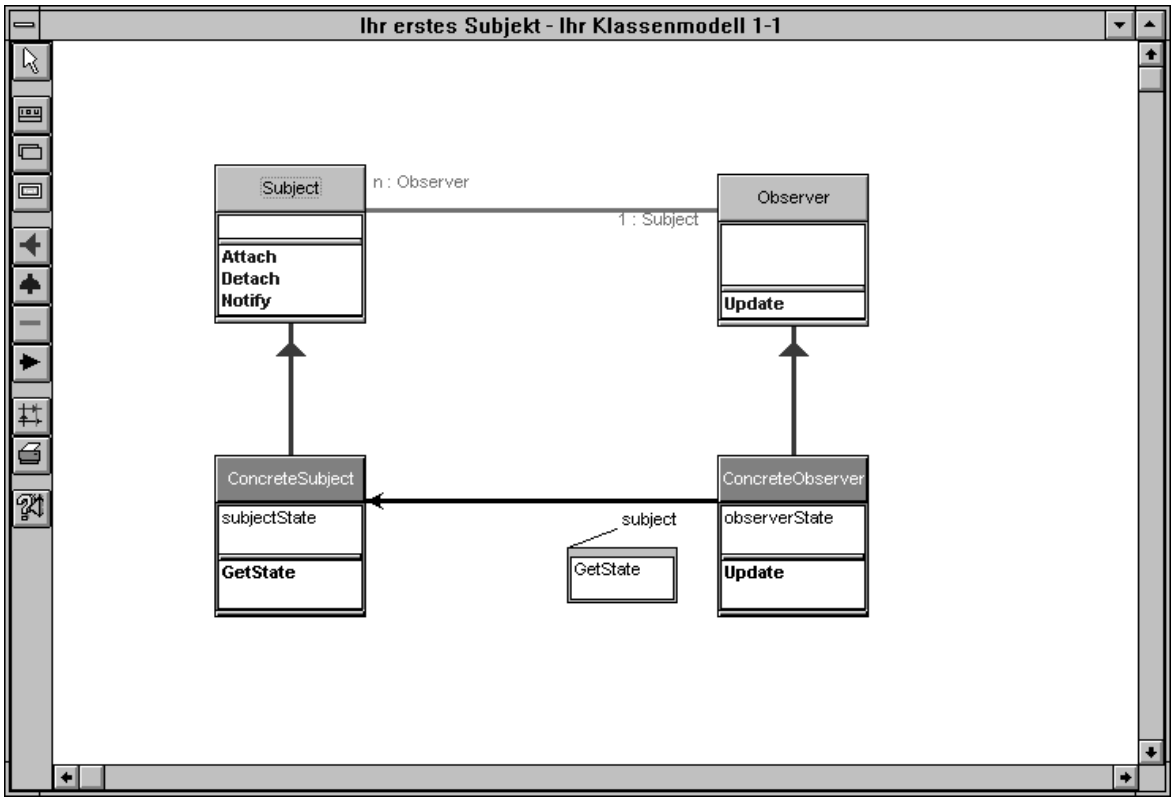

Abbildung 2: Screenshot des Klassenmodell–Editors von objektiF

4.1.2 Die Interoperabilitätsschnittstelle von objectiF

Zur Anbindung von Fremdprodukten wurde objectiF mit einer Schnittstelle versehen, die den dynamischen Datenaustausch (*Dynamic Data Exchange* – DDE) und Objektkommunikation und –interaktion (*Object Linking and Embedding* – OLE) unterstützt. Word wird mit diesen Techniken als OLE– und DDE–Server von objectiF aus angesprochen.
Aus Sicht des Benutzers äußern sich diese Verbindungen wie folgt:
Zum einen können die im Klassenmodell–Editor angezeigten Klassen (in der Abbildung 2 z.B. *ConcreteSubject*) Methoden (*GetState*) und Attribute (*subjectState*) enthalten. Diese Komponenten lassen sich mit textuellen Beschreibungen annotieren. Diese Aktion löst der Anwender über ein Kontextmenü aus. Word wird aktiviert und die Beschreibung dort eingegeben. Bereits existierende Beschreibungen können auf diesem Wege wieder geladen und verändert oder angesehen werden. Für die Realisierung dieser Funktionalität ist die OLE–Verknüpfung zwischen objectiF und Word zuständig.
Die andere Funktion besteht darin, in Word ein Dokument mit einer speziellen Dokumentvorlage zu eröffnen und durch die dann in Form eines objectiF–Menüs angebotenen Erweiterungen von Word aus auf die in der Objektbank von objectiF gespeicherten Daten zuzugreifen. Diese Funktionalität wird durch eine Erweiterung von Word um ein Menü mit den Anweisungen für die automatische Formatierung der Textblöcke und die Steuerung von objectiF durch DDE implementiert. Der Anwender kann alle in textueller Form zu einer Klasse vorliegenden Daten nach Word importieren und die Datenmenge selektiv steuern, also z.B. nur die Klassen, Methoden und Attributnamen und additiv die Parameterlisten. Auch die oben

beschriebenen textuellen Annotationen sind hier abfragbar und zuguterletzt auch noch die Klassendiagramme als Graphik.
Gleicht man die beschriebenen Techniken an der wichtigen Anforderung der Konsistenzerhaltung ab, ist folgendes zu bemerken:
Die Beschreibungen einzelner Komponenten (Klassen, Methoden, Attribute) werden in der Objektbank und nicht als Word–Dokument gespeichert – die Konsistenzerhaltung ist also gewährleistet; erneute Änderungen des Textes werden gespeichert, der Mehrbenutzerzugriff wird durch die Objektbank kontrolliert. Die Konsistenzerhaltung *innerhalb* eines Beschreibungstextes ist *nicht* gegeben, d.h. wird ein Bezeichner für eine Komponente in einem Text verwandt und das Objekt, das durch diesen Bezeichner benannt ist anschließend geändert oder gelöscht, wirkt sich diese Änderung des Klassenmodells nicht auf den Beschreibungstext aus.
Bezüglich des zweiten Verfahrens (OLE) ist zu sagen, daß die Konsistenzerhaltung hier von objectiF nicht mehr kontrolliert werden kann, da ein externes Word–Dokument erzeugt wird. Die Modifikation und Erweiterung des halbautomatisch erzeugten Dokumentes ist so zwar komfortabel in Word möglich, sollten sich aber die Daten des Klassenmodells ändern, werden diese Änderungen nicht automatisch an dem Word–Dokument vorgenommen.

4.1.3 Resümee

Abschließend kann man feststellen, daß die angebotenen Dokumentationstechniken nur von relativ geringer Qualität sind. Der Anwender kann konsistente Dokumentationen erst zu einem sehr späten Zeitpunkt des Entwicklungsprozesses erstellen; Änderungen am Modell erfordern das manuelle Nachvollziehen dieser Änderungen an der Dokumentation.
Zur Design–Dokumentation mit Entwurfsmustern eignet sich der hier beschriebene Ansatz trotz allem, da diese Art von Software–Dokument erst in einer Entwicklungsphase erstellt wird, der schon mehrere Iterationen an Änderungen am Entwurf vorangegangen waren. Der Entwurf ist also hinreichend stabil; es sind keine größeren Änderungen mehr zu erwarten.

4.2 Das Autorensystem ToolBook und die Anforderungen an ein Hypertext–System

Der Einsatz eines Autorensystems zur Generierung eines Hypertext–Systems ist meiner Meinung nach ein pragmatischer Ansatz, da in heute erhältliche Autorensysteme schon einige Entwicklungsarbeit investiert wurde und man somit flexible und erweiterbare Umgebungen erwarten kann.
Ich möchte zunächst die wesentlichen Eigenschaften des Autorensystems ToolBook beschreiben.

4.2.1 ToolBook aus der Nähe betrachtet

Zunächst stellt ToolBook eine Umgebung zur interaktiven Erstellung von (Windows–) Applikationen dar. Der Schwerpunkt liegt dabei auf Hypermedia–Anwendungen, auf interaktiven Schulungsanwendungen, auf Datenbankfrontends und Spielen.
Die für den Anwender von ToolBook sichtbaren Objekte sind anhand der Buch–Metapher vorstrukturiert. Ein ToolBook–Buch besteht wie ein reales Buch aus Seiten. Eine Seite entspricht einem Fensterinhalt in der Anwendung. Auf einer Seite werden Objekte angeordnet. Die anhand der Buch–Metapher bis zur Seitenebene fest implementierte Objekt–Hierarchie bietet den Vorteil, daß sich der Anwender nicht erst in einer neuen abstrakten Denkwelt zurechtfinden muß.
Das Objektparadigma ist darüber hinaus insoweit umgesetzt, als das Objekte Attribute und Methoden haben. Das Geheimnisprinzip läßt sich durch lokale Variablen eingeschränkt umsetzen, wird in der Dokumentation aber nicht erläutert. Message–passing ist insoweit realisiert,

als daß die Objekte eine gemeinsame Untermenge von Nachrichten haben; Botschaften werden anhand der Objekthierarchie solange nach oben weitergereicht, bis sich ein Objekt mit einer entsprechenden Methode findet.
Es wird nicht in Klassen und Objekte unterschieden (somit gibt es auch keine Vererbung). Statt dessen ist die Struktur der Objekthierarchie ab oberhalb der Seitenebene benutzerdefiniert: Der Anwender positioniert Objekte auf einer Seite und kann sie in Gruppen zusammenfassen. Gruppen können wiederum als Objekte behandelt (und so auch rekursiv geschachtelt) werden; als Konsequenz gibt es also einen Satz von Methoden die auf alle Objekte einer Gruppe zugreifen.

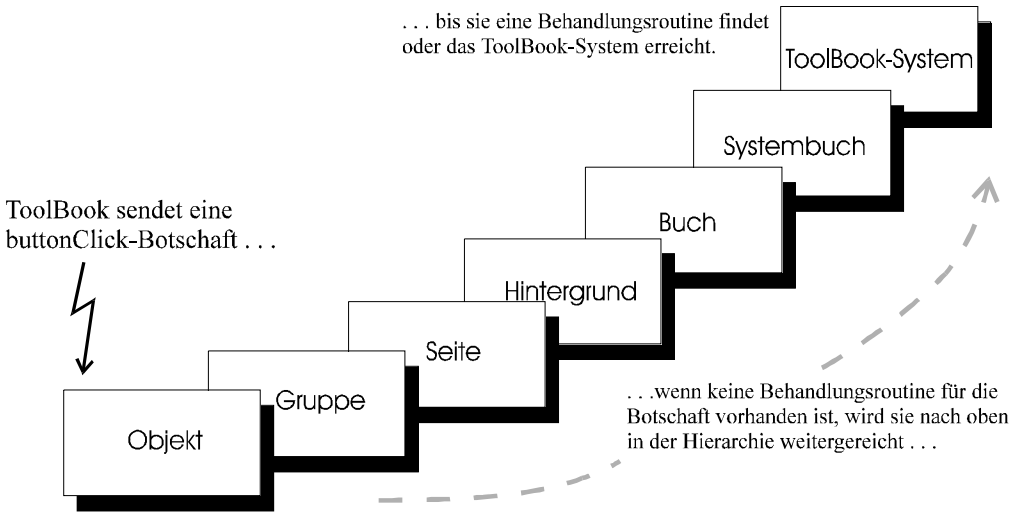

Abbildung 3 (vgl. [Asym94], S.1–12): Die Objekthierarchie von ToolBook

Beispiele für Objekte sind Menüs, Knöpfe und Eingabefelder verschiedener Art, Textfelder, Objekte, die den Zugriff auf Datenbanken erlauben, OLE–Objekte zur Interaktion mit anderen Applikationen (ToolBook kann als OLE–Client fungieren), Graphikobjekte und Objekte, die Videosequenzen verwalten, sowie ganz wichtig Aktionswörter, mit denen Links generiert werden können. In Abbildung 4, Seite 23 ist der Screenshot einer mit ToolBook generierten Hypertext–Applikation zu sehen.
Objekte besitzen eine Menge von vordefinierten Methoden (beispielsweise hat ein Texteingabefeld schon die nötige Funktionalität, um Tastatureingaben zu verarbeiten und Textattribute zu verwenden) und eine Menge von Ereignisbehandlungsroutinen (vergleichbar den *Application–Callbacks*, die in allen APIs graphischer Benutzeroberflächen anzutreffen sind), die durch Benutzer– oder Systemereignisse ausgelöst werden. Das Verhalten eines Objektes kann dadurch konfiguriert werden, daß eine Behandlungsroutine mit einem in der ToolBook–eigenen Programmiersprache *OpenScript* verfaßten Programmfragment versehen wird. Dieses Skript wird beim Empfang der entsprechenden Botschaft abgearbeitet. Zum Beispiel hat ein Button–Objekt eine Methode, die beim Klicken mit der Maus auf dieses Objekt ausgelöst wird. In dieser Methode kann nun die Reaktion der Applikation auf dieses Ereignis spezifiziert werden. Ist die Methode für das Objekt nicht definiert, wird das Ereignis in der Objekthierarchie an das nächst übergeordnete Objekt (eine Gruppe oder die aktuelle Seite) weitergereicht. Dies solange, bis die Nachricht im ToolBook–System (vgl. Abbildung 3) angelangt ist und dort eine Fehlermeldung auslöst.
Zusammengenommen bietet sich dem Anwender eine gut ausgewählte Untermenge des Objektparadigmas, die durch eine leistungsfähige Werkzeugunterstützung ergänzt wird.
Die Hauptapplikation bietet zunächst eine an einen GUI–Builder (Editor zur Erstellung graphischer Benutzeroberflächen) erinnernde Funktionalität zur Anordnung der Objekte auf den Seiten. Durch Kontextmenüs kann der Benutzer auf einzelne Objekte zugreifen und mit weite-

ren Werkzeugen die objektspezifischen Attribute bearbeiten. Das kann z.B. ein Skript sein, das in einem Texteditor bearbeitet wird oder eine Videosequenz in einem Schnitteditor.
Um eine Applikation ablaufen zu lassen, schaltet der Anwender vom Autoren– in den Lesermodus um. Die gesamte ToolBook–Funktionalität zum Modifizieren eines Buches ist dann nicht mehr verfügbar; statt dessen die durch die Objekthierarchie und die Skripte spezifizierte Funktionalität der Applikation.

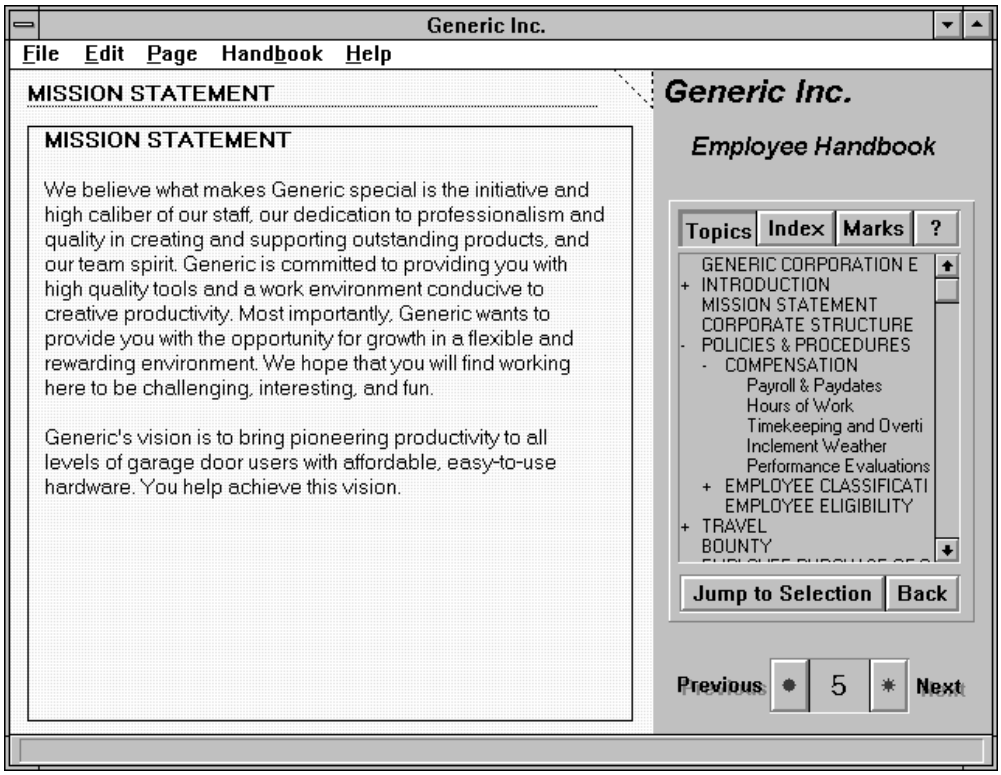

Abbildung 4: Screenshot einer ToolBook–Applikation

4.2.2 ToolBook als Hypertext–System

ToolBook bietet von den im Kapitel 3 beschriebenen Anforderungen an Hypertext–Systeme folgende Merkmale:

- Die **Software–Architektur** (Abschnitt 3.6.1) eines Hypertext–Systems kann mit ToolBook realisiert werden, da eine umfangreiche Funktionalität zum Zugriff auf Datenbanken enthalten ist.
- **Mehrbenutzerzugriff** und **Versionsverwaltung** werden nicht direkt von der in ToolBook integrierten Datenverwaltung unterstützt. Diese Funktionalität sollte in einer Systemarchitektur, die aus einer Datenbank und ToolBook als Datenbankapplikations–Generator besteht auch von der Datenbank bereitgestellt werden (siehe Abschnitt3.6).
- Alle wichtigen **Linktypen** (Abschnitt 3.2) sind mit der von ToolBook bereitgestellten Funktionalität generierbar, einige davon nur durch manuelle Erweiterung.
- Die in den Abschnitten 3.2 und 3.3 beschriebene **Zugriffsfunktionalität** zur Navigation in Hyperdokumenten wird dem ToolBoook–Anwender ebenfalls nicht vordefiniert beigestellt und muß auch manuell durch Skripte generiert werden.

Aus dieser Aufstellung läßt sich ein offensichtlicher aber behebbarer Nachteil von ToolBook herauslesen: Die mangelnde Unterstützung des Anwenders durch vorgefertigte Komponenten für die Erstellung von Hypertext–Systemen. Verglichen mit anderen Alternativen ist ToolBook

aber gut für die Erstellung solcher Systeme geeignet. Im Unterschied zu einem konventionellen Textverarbeitungsprogramm stehen dem Benutzer eines Autorensystems Links als integraler Bestandteil zur Verfügung. Durch die Erweiterung moderner Systeme wie ToolBook zur Integration von Videosequenzen und Ton sind weitere Vorteile gegeben. Die andere Alternative ist die vollständige Neuentwicklung eines Hypertext–Systems, was mit einem ungleich höheren Aufwand verbunden wäre, dem Zuschnitt auf spezielle Anforderungen und Effizienzaspekten aber am besten gerecht werden könnte.

4.2.3 Resümee

Die Integration eines Autorensystems in eine SEU ist über die eigentliche Adaption des Sytems hinaus eine zu leistende Arbeit. Wichtig ist meiner Meinung nach hierbei der Bezug auf einen Interoperabilitätsstandard, der über die Funktionalität von OLE hinausgeht. Soweit hier keine Unterstützung seitens der SEU und des Autorensystems vorliegt, ist von einer Kopplung abzuraten, da die Konsistenzerhaltungsproblematik auf anderem Wege nicht realisierbar ist. Die oben beschriebenen Nachteile bei der Kopplung sind die Folgen.
Momentan sind meines Wissens kommerziell noch keine Software–Systeme (SEU wie auch Autorensysteme) verfügbar, die einen ausreichend weit entwickelten Standard für Verbunddokumente unterstützen. Die Integration von leistungsfähigen Textverarbeitungs– oder Hypertext–Systemen in SEU's ist also momentan nur unzureichend und mit erheblichem Aufwand realisierbar; die Entwicklung proprietärer Lösungen durch Anbieter von SEU's nicht (mehr) zu erwarten und auch nicht sinnvoll, da momentan einiges im Bereich Verbunddokumente (CORBA, OpenDoc) seitens der durch die Software–Firmen getragenen Organisationen (OMG, CILabs) marktreif wird.

4.3 Vorgehensmodelle zur Generierung von Hypertext–Software–Dokumenten

Die Produktion eines qualitativ hochwertigen Hypertextes stellt an den Autor wesentlich höhere Anforderungen, als die eines konventionellen Textes. Das gilt unabhängig von der Qualität der Werkzeugunterstützung, da eine sinnvolle Partitionierung der Information vorgenommen werden soll, die Vernetzung der Knoten durch Links einiger Überlegung bedarf und es sich meist um große Datenmengen handelt.
Hyperdokumente enthalten nicht nur Informationen, sondern sind auch untrennbar mit der nötigen Funktionalität zur Navigation verbunden. Diese Funktionalität ist oft spezifisch auf einen bestimmten Anwendungsfall zugeschnitten. Man kann die Produktion von Hyperdokumenten somit als Software–Entwicklung bezeichnen. Dieses Vorgehen ist natürlich als inadäquat bezüglich des *Einsatzes* von Hypermedia in der Software–Entwicklung selbst zu bezeichnen; der Projektteilnehmer kann nicht auch noch nebenher ein Softwareprojekt zur Erstellung einer Hyper–Applikation durchführen. Somit kommt für den Einsatz in diesem Bereich nur ein vorkonfiguriertes Hypertext–System in Frage, daß sämtliche Benutzerfunktionalität zur Eingabe und die nötigen Komponenten zur Navigation in den Daten schon enthält.

4.3.1 Vorstrukturierung von Software–Dokumenten

Um den Anwender auch noch von der Tätigkeit der Strukturierung des Hyperdokumentes zumindest teilweise zu entlasten, kommt folgendes in Betracht:

- Die Wahl einer festen **globalen Vorstrukturierung**, wie sie durch ein Vorgehensmodell zur Software–Entwicklung gegeben ist und desweiteren durch die Verwaltung aller Dokumente in einem gemeinsamen Informationsraum entsteht,

- eine **lokale Vorstrukturierung** durch halbautomatische Erzeugung von Teilen des Hyperdokumentes auf Anforderung des Benutzers,
- hierdurch **Trennung von Struktur und Inhalt**: der Anwender orientiert sich im vorstrukturierten Dokument und kann sich auf die inhaltlichen Aspekte konzentrieren. Die lokal–strukturellen Aspekte werden in den Methodeneditoren behandelt, z.B. im Klassenmodell– oder im Zustandsdiagramm–Editor.
- Optional **automatische Generierung von Links.** Beispielsweise bei der Eingabe von Texten, wenn ein Bezeichner als Begriff verwendet wird, Erzeugung einer Referenz hin zur Definition.

Die Tätigkeit der Erstellung eines Hyperdokumentes kann sich so zumindest in weiten Teilen auf die Vervollständigung vorgegebener Informationen reduzieren.
Die weiter oben beschriebene Erzeugung eines Word–Dokumentes durch die Abfrage von objectiF, stellt ein solches Vorgehen ansatzweise dar, ist aber mit den Nachteilen einer sequentiellen Dokumentation und folglich dem Verlust (oder zumindest dem Verlust der Sichtbarkeit) der im Klassendiagrammen enthaltenen Strukturinformation verbunden.

4.3.2 Objektorientierte Dekomposition

Als praktisches Vorgehensmodell für die Erzeugung von Hypertexten läßt sich z.B. die in [Gloor90], S.216f zitierte *objektorientierte Dekomposition* anführen (der Begriff objektorientiert ist hier (mal wieder) überladen worden. Offensichtlich lassen sich keine Komponenten des Objekt–Paradigmas in diesem Vorgehen identifizieren . . .):

(1) Inhaltszusammenfassung in einem Satz.
(2) Problembeschreibung hierzu in fünf bis neun Sätzen.
(3) Identifikation der grundlegenden Konzepte des Problems und Beschreibung der Unterprobleme in hierarchisch weiter unten gelegenen Knoten: Top–Down–Vorgehen.
(4) Definition der Beziehungen und Schnittstellen.
(5) Review und Revise. Alle Informationen zu einem Hypertext–Knoten und den unterliegenden Knoten sind zusammengetragen und werden nochmals einer Überprüfung unterzogen und evtl. revidiert.
(6) Wiederholung der Schritte 1 bis 5 für jedes neue Konzept.
(7) Zusammenfassung verwandter Konzepte. Nach Beschreibung aller Konzepte in Hypertext–Knotenhierarchien werden nun inhaltlich verwandte Konzepte in hierarchisch übergelagerte Knoten zusammengefaßt: Bottom–Up–Verfahren.
(8) Implementierung der Links. Die Links können erst nach vollständiger Erzeugung aller Knoten und der Bottom–Up–Revision anhand der in Schritt 4 ermittelten Beziehungen eingeführt werden.

5 Entwurfsdokumentation mit Mustern

Durch Entwurfsmuster erfährt der Bereich der Software–Dokumentation eine wesentliche Bereicherung des Spektrums der Beschreibungsmittel. Es ist eine Sprache für die Diskussion von Software–Design entstanden. Die Erzeugung wiederverwendbarer Entwürfe wird mit Mustern erleichtert.
Da im Rahmen des Seminars die Entwicklung des Entwurfsmusterbegriffes, die Begriffsdefinitionen und die verschiedenen Strömungen in anderen Arbeiten dargestellt werden, möchte ich hier nicht unnötige Redundanzen erzeugen und verzichte auf eine umfassende Beschreibung. Die im Anhang aufgeführte Literatur, insbesondere [Pree94], S.60ff; [Gam+93]; [Gam+95], S.351ff und [Alex77] enthalten darüber hinaus die nötigen einführenden Informationen. Statt dessen will ich die Verwendung von Entwurfsmustern zur Dokumentation von Frameworks beschreiben. Hierzu werde ich zunächst den Begriff Framework beleuchten und anschließend zwei verschiedene Hypertext–Dokumentationstechniken, die für unterschiedliche Anwendungsbereiche eines Frameworks gedacht sind, beschreiben.
Zuletzt möchte ich noch auf die in [Pree94], S.223ff gemachten Feststellungen zur Eignung von modernen OOA/D–Methoden für eine Framework– und Entwurfsmusterbasierte Software–Entwicklung eingehen.

5.1 Frameworks

Um wiederverwendbares Design zu produzieren, bedarf es einiger Anstrengungen. Frameworks (die deutsche Übersetzung Rahmenwerk ist nicht gebräuchlich) sind Produkte solcher Entwicklungen, die zunächst in das Objekt–Paradigma eingeordnet werden können.
Die in objektorientierten Programmiersprachen durch Klassen definierten abstrakten Datentypen (ADTs) sind durch die Vererbungsrelation in einem *Baum* angeordnet (wenn man die eigentlich überflüssige Mehrfachvererbung einmal außer acht läßt). Es können mehrere unabhängige Wurzeln (Basisklassen) existieren, die dann mehrere Bäume definieren. Zusammengenommen stellen sie einen Wald dar, der als *Klassenbibliothek* bezeichnet wird.
Als Frameworks werden solche Klassenbibliotheken oder Teile von Klassenbibliotheken bezeichnet, die bestimmten Anforderungen genügen. Frameworks können konzeptionell als Erweiterung der abstrakten Klasse aufgefaßt werden (vgl. [Gam92], S.11f): Ein *System* aus teils konkreten und teils vom Anwender des Frameworks noch für seinen Anwendungsfall zu konkretisierenden abstrakten Klassen definiert einen Entwurf zur Lösung einer (Problem–) Klasse von verwandten Aufgabenstellungen. Wichtig ist, daß Komponenten eines Frameworks nicht isoliert verwendet werden können, ein Framework ist ein System und unterscheidet sich so von Klassenbibliotheken, die Sammlungen von *Bausteinklassen*, wie z.B. Datenstruktur–Klassen (Listen, Bäume, Hash–Tabellen), anbieten.
Die Klassen eines Frameworks haben festgelegte Verantwortlichkeiten; die Interaktion zwischen den Klassen ist durch *Protokolle* ebenso fest definiert. Diese Protokolle stellen ein weiteres wichtiges Merkmal von Frameworks dar. Der Anwender benutzt diese Protokolle in seinen konkreten Klassen und erweitert diese gegebenenfalls.
Ein weiteres Merkmal stellt die Robustheit dar. Die Verwendung eines Frameworks erfordert nicht dessen Veränderung. Der Benutzer konkretisiert nur die *Hot Spots* (vgl. [Pree94], S.106) in abgeleiteten Klassen. Strukturelle Änderungen des Systems sind so natürlich nicht machbar; eine Änderung oder Erweiterung des Frameworks ist nur auf der Grundlage genauer Kenntnisse über die internen Abläufe möglich. Daher ist im Vorfeld einer Entscheidung über den Einsatz eines Frameworks die Überprüfung auf die Eignung für den Problembereich notwendig. Die Wahl eines bestimmten (Application–)Frameworks oder *Architectural Framework*

(vgl. [Busch95]) zu Beginn des Grobentwurfs stellt eine grundlegende Designentscheidung dar, da die fundamentale Struktur eines Software–Sytems hierdurch festgelegt ist.

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method (107)	Adapter (class) (139)	Interpreter (243) Template Method (325)
	Object	Abstract Factory (87) Builder (97) Prototype (117) Singleton (127)	Adapter (object) (139) Bridge (151) Composite(163) Decorator (175) Facade (185) Flyweight (195) Proxy (207)	Chain of Responsibility (223) Command (233) Iterator (257) Mediator (273) Memento (283) Observer (293) State (305) Strategy (315) Visitor (331)

Abbildung 5 ([Gam+95], S.10): *Design Pattern Space*

5.1.1 Application Frameworks

Ein *Application Framework* stellt eine spezielle Art von Framework dar. Es definiert das Grundgerüst einer vollständigen Applikation. Solche generischen Applikationen enthalten alle grundlegenden Verwaltungsfunktionen der Problemdomäne für die sie entwickelt wurden; der globale Kontrollfluß und das *Look and Feel* der Applikation sind vordefiniert.
Ein Application Framework für Editoren enthält so z.B. die teilweise noch zu adaptierende, aber im wesentlichen festgelegte Funktionalität zum Speichern und Laden von Dokumenten, Anzeige, Copy&Paste sowie Ausdruck, etc. Allerdings ist noch keine applikationsspezifische Funktionalität realisiert, wie z.B. das konkrete Format eines Dokuments der Applikation.
Der Anwender eines Application Frameworks wird stark in seiner Arbeit unterstützt, da sich seine Tätigkeit ganz auf die applikationsspezifischen Komponenten konzentrieren kann; durch die Vorgaben wird auch die Einhaltung eines qualitativ hochwertigen Entwurfs erleichtert. Der Benutzer erhält z.B. bei den *Microsoft Foundation Classes* (MFC) (vgl. [Micro94]) nach einem interaktiven Konstruktionsprozeß der initialen Applikation mit einer *AppWizzard* genannten Komponente der Entwicklungsumgebung eine Menge von Dateien mit Klassendefinitionen, die sich im zu konkretisierenden Teil in die Klassen *CDocument* und *CView* aufteilen, von denen abgeleitet wird, um die applikationsspezifische Funktionalität zu implementieren. Microsoft hat hier seinem Application Framework MFC eine Variation des MVC–Entwurfsmusters zugrundegelegt; der Anwender kann sich in dieser Denkwelt (meine Daten und deren Visualisierung) aufhalten und wird nur minimal von betriebssystemspezifischen Fragen tangiert. Auch so komplexe Konzepte wie OLE werden dem Anwender auf einer handhabbaren Abstraktionsebene angeboten (vgl. [Bisch95]).
Generell läßt sich sagen, daß Frameworks den momentan leistungsfähigsten Ansatz zur Bereitstellung von wiederverwendbarem Design darstellen. Was bisher fehlt sind Techniken (und vor allem Werkzeuge, die diese Techniken umsetzen) zur qualitativ hochwertigen Dokumentation und Exploration von Frameworks. Pionierarbeit ist hier von Gamma mit ET++ und den in [Gam92] beschriebenen Tools und formal–graphischen Notationen geleistet worden.

5.2 Hypertext–Dokumentations–Muster für Frameworks

Entwurfsmuster (engl. *Patterns* oder *Design Patterns*) sind hierarchisch nach Abstraktionsebenen oder nach der Enthaltenseinsbeziehung der beschriebenen Objekte in einem Netz

angeordnet. Diese Forderung wird schon von Alexander in [Alex77], S.xii gestellt und in seinen Patterns auch angewandt, indem er sie in die hierarchisch angeordneten Subnetze *Towns*, *Buildings* und *Constructions* unterteilt. Gamma et al. verwenden die in Abbildung 5 dargestellte Unterteilung der Muster in *Creational*, *Structural* und *Behavioral* und die weitere Einteilung in Muster, deren Komponenten eher durch statische (*Class*) oder durch dynamische Beziehungen (*Object*) ausgezeichnet sind; für erstere gilt, daß die (statische) Vererbungsrelation die für sie bestimmende Beziehung ist, letztere machen vor allem Gebrauch von der (dynamischen) Komposition von Objekten zur Laufzeit. Einer Hypertext–Übersicht ähnelt der in Abbildung 6 dargestellte Graph. Die Beziehungen der Gamma–Pattern untereinander werden so deutlich.

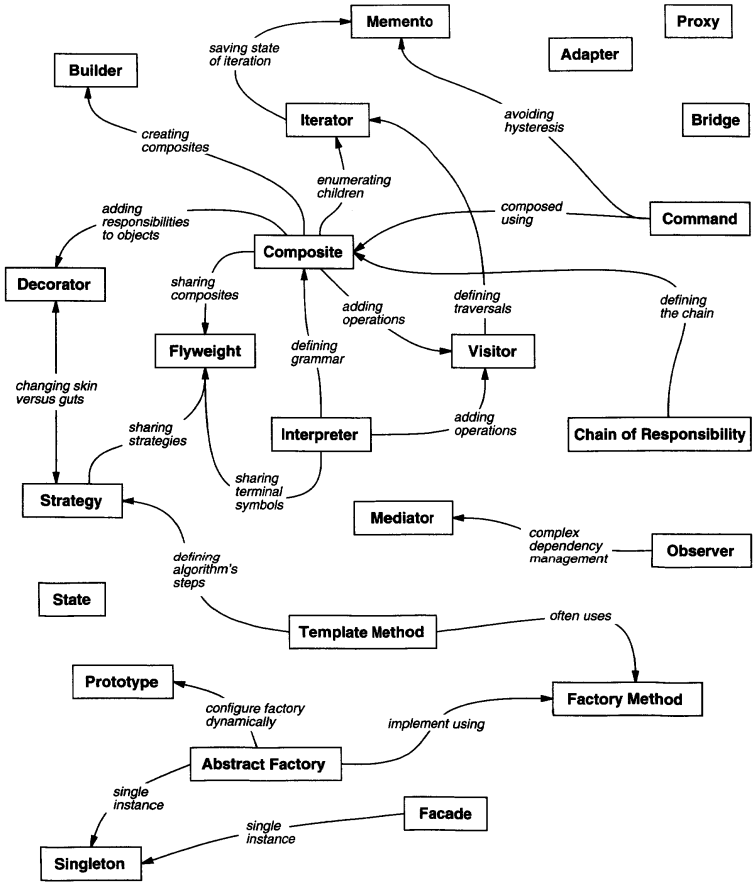

Abbildung 6 ([Gam+95], S. 12): *Design pattern relationships*

Buschmann (Abbildung 7, Seite 29) schließlich gruppiert Muster (weiter übergeordnet) in *Architectural Frameworks*, *Design Patterns* und *Idioms*. Erstere unterstützen bei der Wahl der generellen Systemarchitektur, die Design Patterns befassen sich mit Fragen des Entwurfs, Idioms werden an der Schwelle vom Entwurf zur Implementierung eingesetzt.

5.2.1 Anforderungen an Entwurfsmuster unter dem Blickwinkel der Software–Dokumentation

Zunächst einige grundlegende Forderungen an Entwurfsmuster aus Sicht der Software–Dokumentation:

- Entwurfsmuster sollen zur Produktion von **wiederverwendbarer** Design–Dokumentation beitragen. Sonst ergibt sich gegenüber konventioneller Dokumentation kein Vorteil aus ihrer Anwendung.

- Entwurfsmuster sollen **erweiterbar** sein in zweierlei Hinsicht: Zum einen müssen sich neue Muster integrieren lassen und zum anderen die bestehenden veränderbar sein.
- Zur Dokumentation von Entwurf und Programmcode ist es von großem Vorteil eine **paradigmatisch verwandte** Dokumentationstechnik einzusetzen.

Überblick über die Gruppierung von Entwurfsmustern

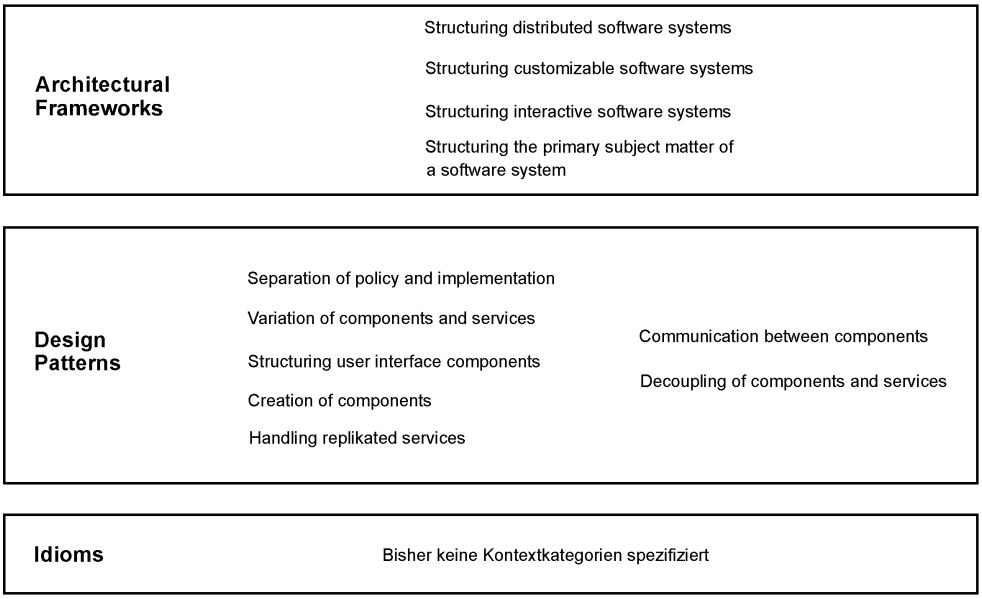

Abbildung 7 ([Busch95], Organis. von Mustern(4)): Gruppierung von Entwurfsmustern

Die Forderung nach Wiederverwendbarkeit führt zu folgenden Anforderungen: Die Dokumentation mittels Patterns erfordert ein Dokumentationsmuster (im folgenden als *Template* bezeichnet, nicht zuletzt, um die Mehrfachbelegung des Musterbegriffes nicht weiter zu strapazieren). Dieses Template strukturiert wie ein Formular das Dokument (hier das Entwurfsmuster) vor (vgl. z.B. [Gam+95], S.6ff). Es wird so einerseits die Lesbarkeit erhöht und andererseits erhält der Autor eine Hilfestellung. Anhand der **hierarchischen Gliederung** sollte der Anwender eines Musterkataloges entlang der vom Problembereich vorgegebenen Anordnung geleitet werden; desweiteren eine Einteilung in Abstraktionsebenen vorgenommen werden.

Die Forderung nach Erweiterbarkeit impliziert wegen des sonst nicht vertretbaren Aufwandes eine elektronische Verwaltung der Pattern. Die Integration neuer Muster erfordert in aller Regel eine Vernetzung mit den bestehenden in Form von Verweisen. Die Veränderung bestehender Patterns wirft die Frage auf, ob die alte Version des Entwurfsmusters bestehen bleiben soll. Ist dies der Fall (z.B. Unterstützung mehrerer Versionen eines Frameworks), muß das Dokumentationssystem eine **Versionsverwaltung** enthalten. Die klassischen Werkzeuge zur Versionsverwaltung arbeiten durchweg auf der Dateiebene. Ein für den Objekt–Kontext verwendbares Tool muß mit einer feineren und einstellbaren Granularität (z.B. Versionierung von Klassen) aufwarten. Versionsverwaltung sollte daher in die unterliegende Datenbank integriert werden.

Eine zum objektorientierten Entwurf und zu einer objektorientierten Programmiersprache passende Dokumentationstechnik äußert sich durch eine **objektorientierte Struktur**. Entwurfsmuster eignen sich zur Dokumentation von objektorientiertem Entwurf, da sich die Kapselung, die Vererbungsrelation und die Komposition gut auf die Netzwerkanordnung aus Knoten (Entwurfsmustern) und Kanten (die Referenzen zwischen den Mustern) adaptieren läßt. Die Gamma–Pattern und vor allem die Metapattern [Pree94] bieten diese Struktur inhärent, da sie mit abstrahierten Frameworks, bzw. allgemeinen Strukturprinzipien der Objektorientierung

arbeiten. Durch die Trennung von dieser *Struktur* und dem eigentlichen *Inhalt* der Dokumentation (siehe Abschnitt 4.3.1) kann sich der Software–Ingenieur bei der Tätigkeit des Dokumentierens voll auf die inhaltlichen Aspekte konzentrieren. Die Vorstrukturierung durch halbautomatisches Generieren von Teilen der Dokumentation und die Bereitstellung von Dokumentations–Templates verhindern den Paradigmawechsel zwischen objektorientiertem Entwurf und Dokumentation.
Fazit: Es treten bei der Werkzeugunterstützung von Software–Dokumentation mit Mustern prinzipiell die gleichen Fragestellungen auf, wie bei der Werkzeugunterstützung der Dokumentation von objektorientiertem Programmcode allgemein: Zur adäquaten Dokumentation müssen sich die, das Paradigma auszeichnenden Mittel in der Dokumentation fortsetzen – objektorientierte Dokumentation.

5.2.2 Zwei Templates für unterschiedliche Software–Dokumente

Johnson beschreibt in [John92] ein Modell zur Dokumentation von Frameworks mit Patterns. Verschiedene Typen von Patterns sind in einem hierarchischen Netz angeordnet. Es gibt Patterns, die den generellen Zweck des Frameworks erläutern, andere, die das „wie“ der Benutzung des Frameworks in Form von Cookbooks (s.u.) beschreiben und schließlich jene, die das detaillierte Design des Frameworks darstellen. Anhand des Entwurfs eines graphischen Editors (*HotDraw*) stellt er die Pattern und deren Anordnung vor. In [Beck+94] dokumentieren Beck und Johnson das Application Framework Hot Draw mittels Mustern.
Die elektronische Verwaltung von Entwurfsmustern bietet sich an, da so eine Integration dieser Dokumentationstechnik in eine Software–Entwicklungsumgebung erst möglich wird. Desweiteren bietet sich Hypertext an, da Hypertext exakt auf das Anforderungsprofil von Patterns paßt.
Ich möchte im folgenden zwei Dokumentations–Templates vorstellen, die, wie ich meine besonders gut für zwei wichtige Aspekte der Dokumentation von Frameworks geeignet sind:
Die **Benutzung von Frameworks** durch Anwender, die sich nicht für die Interna eines Frameworks interessieren, sondern die es möglichst schnell produktiv für die Entwicklung einer eigenen Applikation einsetzen wollen. Hierzu stelle ich den CookBook–Ansatz vor.
Desweiteren die **Design–Dokumentation von Frameworks** für Benutzer, die ein Framework verstehen wollen, um es zu erweitern oder zu verändern. Dieser Benutzergruppe gehören z.B. auch die Entwickler des Frameworks an. Hierzu gehe ich auf die Metapatterns von Pree ein.
Der Begriff Dokumentations–Template wird hierzu erweitert: Nicht nur die in allen Publikationen zu Pattern beschriebenen textuellen Gliederungen (vgl. [Alex77], S.x und [Gam+95], S.6ff) sollen ein Template sein, sondern auch die Metapatterns mit ihrer sich aus objektorientierten Entwurfsprinzipien ergebenden Strukturierung. Ein Template im Sinne dieser Arbeit ist somit ein Denkmodell, eine Vorstrukturierung, die es dem Autor und dem Leser erleichtert, qualitativ hochwertige Dokumente zu erstellen, bzw. zu explorieren.

5.2.3 Cookbooks

Cookbooks, das intuitive „man nehme...“ werden von Johnson in [John92] beschrieben, der Klassiker [Kras+88] ist ein gutes Beispiel für ein Cookbook. Pree kategorisiert in seinem Buch [Pree94] die verschiedenen Design Pattern Methoden (*design pattern approaches*); eine davon ist das *framework cookbook*. Ich beziehe mich im folgenden auf seine Beschreibung.
Für Pree sind Cookbooks gut dazu geeignet, das *how to use* eines Frameworks zu dokumentieren und nicht geeignet zur Entwurfsdokumentation. Ein Cookbook besteht aus mehreren Rezepten (man kann sie als eine mögliche Variante eines Pattern ansehen), die in eher informeller Sprache den Zweck (*purpose*), das Prozedere zur Nutzung (*procedure*) und anhand von Sourcecode–Beispielen (*source code examples*) den Einsatz des Frameworks beschreiben. Die Rezepte stellen verschiedene Nutzungsaspekte dar und referenzieren sich untereinander. Eine

Hierarchie ist denkbar und sinnvoll, beispielsweise angelehnt an Anfänger (Ein Rezept zur Erläuterung der grundlegenden Philosophie mit einem Hello–World–Beispiel) und fortgeschrittene Nutzer (Ein Rezept zur Integration von benutzerdefinierten Kontextmenüs). Besonders wichtig bei der Dokumentation ist die Auswahl guter Sourcecode–Beispiele, da so der praktische Einsatz des Frameworks am direktesten dargestellt werden kann (Abbildung 8, Seite 31).

Purpose

A `Document` object manages the data set of an application independently of how it is displayed or printed. It also provides methods to save the data in files and read it back (for details see the recipe „Saving and restoring data in documents“).
ET++ asks your application to create a new application–specific document when the user starts the application or chooses the New or Open menu commands.

Steps how to do it

(1) Declare the file type of your document as an instance of `Symbol`. This is usually done by an `extern` declaration in the header file where your `Document` class is defined:

```
extern Symbol cYourDocType;
```

The actual declaration is done in the implementation file of your `Document` class:

```
Symbol cYourDocType(„YourTypeName“);
```

If you use a file format that is predefined in ET++, use the predefined file type. For example, a file made up of ASCII characters is of type `cDocTypeAscii`.
The file type has to be passed to the constructor of class `Application`. It becomes the principal file type of your application. If your application must deal with various file types, see the recipe „Handling different file types in an application“.

(2) Implement `YourApplication::DoMakeManager` in the subclass `YourApplication` of `Application`. ET++ calls this method every time a new document needs to be created. You need the following declaration as part of the definition of `YourApplication`:

```
Manager *YourApplication::DoMakeManager(Symbol managerType);
```

The Parameter `managerType` is the principal file type. Usually this file type is passed on to the created document.
Recall that `Document` is a subclass of `Manager`. Thus references to `Document` objects are type compatible with the required return type of `DoMakeManager`.
A sample implementation is given in the source code example for this recipe.

(3) If you have menu commands other than the standard File menu commands, that is, New, Open, Save, Save As and Revert, that apply to the document or ist contents, override the methods `DoMakeMenuBar`, `DoMenuCommand` and `DoSetUpMenu` of class `Document`. See the „Creating menus and handling menu commands“ recipe for details on these methods.

Source code example

```
. . .
#include “YourApplication.h“
#include “YourDocument.h“

YourApplication::YourApplication(...) : Application(..., cYourDocType)
{
    . . .
}
Manager *YourApplication::DoMakeManager(Symbol  managerType)
{
    YourDocument *aDoc;
    aDoc = new YourDocument(managerType, ...);
    return aDoc;
}
. . .
```

Abbildung 8 : Ein Cookbook–Rezept aus [Pree94], S.81f

Pree schlägt als adäquates Dokumentationswerkzeug für Cookbooks ein Hypertext–System vor. Die Fähigkeit, Informationen durch die Darstellung als Hypertext in der Breite (Menge der

gleichzeitig angebotenen Informationen) und Tiefe (Hierarchisierung über die Detaillierung) zu begrenzen kann hier gut ausgenutzt werden. Als Beispiel dient der Hypertext–ET++–Cookbook–Editor (vgl. [Pree94], S.84ff).

5.2.4 Metapatterns

Wird ein Hypertext–System zur Dokumentation mit Patterns des Design Pattern Katalogs von Gamma et al. [Gam+95] verwendet, kann auch das Design eines Frameworks auf diese Weise beschrieben werden. Pree bezeichnet die Gamma–Patterns als *framework example design pattern* (vgl. [Pree94], S.105) — aus der Warte, daß durch Abstraktion von konkreten Frameworks Entwurfsentscheidungen beschrieben werden. Durch eine Umbenennung der Klassen, Attribute, Methoden und Beziehungen lassen sich aus Gamma–Pattern konkrete Mikroframeworks generieren. Entwerfer von Musterkatalogen versuchen, daß die Framework–Beispiele nicht zu bereichsspezifisch sind, um die Flexibilität aufzuzeigen und eine gute Adaptierbarkeit auf andere Bereiche zu gewährleisten.
Trotzdem ist es wichtig zu erkennen, daß Framework–Beispielkataloge ihre Grenzen haben. Es gibt zum einen nicht viele allgemeine, bzw. nicht bereichsspezifische Framework–Beispiele. Es besteht die Gefahr einer „Inflation" von Patterns minderer Qualität mit einem hohen Maß von Redundanz. Ein Beispiel hierfür bilden die Patterns von Coad [Coad95], die sich auf ein geringe Zahl wirklich verschiedener Pattern reduzieren lassen. Die Folge ist der Verlust der durch die kompakte Darstellung der Gamma–Pattern gewonnenen Übersicht. Zum Zweiten bietet der Gamma–Pattern–Ansatz keine Methodik, um Design unabhängig von einem mehr oder weniger spezifischen Framework–Beispiel zu extrahieren (vgl. [Lehn94], S.159). Wie schon erwähnt entstehen Gamma–Pattern „nur" durch eine Umbenennung der Klassen–, Beziehungs– und Methodennamen spezieller Frameworks, bzw. werden konkrete Frameworks durch Umbenennung der Bezeichner der Gamma–Patterns in die Begriffswelt der konkreten Anwendung generiert.
Zur Dokumentation von beliebigen, also allgemeinen und domänenspezifischen Frameworks und aus der Warte, daß eine Werkzeugunterstützung für den Dokumentationsprozeß selbst und für die Exploration von Frameworks grundlegend wichtig ist, beschreibt Pree in seinem Buch [Pree94] den Metapattern–Ansatz. Auf einer weiteren Abstraktionsebene über den Gamma–Pattern installiert und für die Werkzeugunterstützung bestens geeignet, sind Metapattern ein weiterer Schritt hin zu qualitativ hochwertiger Software–Dokumentation.
Der Unterschied zwischen Entwurfsmuster–Katalogen und Metapatterns resultiert in einer gegenseitigen Ergänzung. Der in [Gam+95] niedergeschriebene Entwurfsmuster–Katalog wird zur Bibel des Entwurfsingenieurs werden. Unterstützung bei der *Konstruktion* von Frameworks und der Entwicklung gutem objektorientiertem Designs stehen im Mittelpunkt dieses Ansatzes. Desweiteren hilft dieser Katalog bei der Diskussion von Designfragen, da endlich eine durchdachte und verbreitete Begriffswelt geschaffen wurde. Der Ansatz von Pree dient zunächst der *Dokumentation* und somit der Benutzer–Unterstützung bei der *Exploration* von Frameworks. Ich vernachlässige in meinen Ausführungen bewußt den Konstruktionsaspekt der Metapattern, da er für die Entwurfs–Dokumentation keine Rolle spielt (Abschnitt 5.3). Die Metapattern kommen am Ende des Entwicklungsprozesses eines Frameworks zum Einsatz. Erst wenn ein Framework *mature and well designed* (vgl. [Pree94], S.106 und S.170) ist, also die *hot spots* identifiziert sind, bzw. die Zuordnung zu Template– und Hook–Klassen (s.u.) erfolgt ist, lohnt es sich, es mit Metapatterns zu dokumentieren.
Bei der Dokumentation von hinreichend bereichsunabhängigen Frameworks oder Teilen von Frameworks sind die zwei Techniken Gamma– und Metapattern gleichmächtig. Für die Dokumentation von domänenspezifischen Frameworks ist der Metapattern–Ansatz flexibler, da immer vom Allgemeinen auf das Besondere geschlossen werden kann und die Exploration eines Frameworks in den spezifischen Komponenten sich so einfacher darstellt. Der Anwender

ist nicht auf das Verstehen von immer neuen Design Patterns angewiesen, sondern kann nur auf Basis der Kenntnis von grundlegenden Strukturbeziehungen von Klassen und Objekten, formuliert in den sieben Metapatterns, diese Strukturen in ihm unbekannten domänenspezifischen Komponenten eines Frameworks identifizieren.
Die Metapattern von Pree sind sicher durch die Analyse von Design Patterns entstanden. Pree hat die Gemeinsamkeiten der Design Patterns erkannt und versucht, diese auf einer höheren Abstraktionsebene zu beschreiben. Nebenbei entstand so eine Notation zur Darstellung der dynamischen Struktur von Objekten zur Analyse der Abläufe beim Versenden von Nachrichten an Gruppen von Objekten ([Pree94], S.127ff). Das *Forwarding* von Botschaften und die Benachrichtigung bei *Updates* kann mit diesen Mechanismen mit entsprechenden Werkzeugen visualisiert und getestet werden.
Pree greift die von anderen Autoren schon beschriebene Unterscheidung zwischen zwei Methoden–, bzw. zwei Klassentypen auf: *Template*– und *Hook*–Methoden, bzw. Klassen ([Pree94], S.106ff). In Template–Methoden sind die *frozen spots* eines Frameworks implementiert. Das ist die (meist in abstrakten Basisklassen) vordefinierte Grundstruktur des Frameworks. Die Hook–Methoden werden von Template–Methoden aufgerufen und stellen die adaptierbaren Komponenten des Frameworks dar, von Pree als *hot spots* bezeichnet. Eine Template–Methode kann wiederum als Hook–Methode verwendet werden; so können die unten beschriebenen Hierarchien aus Template– und Hook–Objekten erzeugt werden. Die Erweiterung des Konzepts von der Methoden– auf die Klassenebene ist evident. Durch die Trennung von Template– und Hook–Methoden in verschiedene Klassen wird die flexible Komposition von Objekten mit zur Laufzeit austauschbaren Hook–Objekten erst möglich. Aber auch die „Wiedervereinigung“ in Template–Hook–Klassen macht Sinn: in den unten beschriebenen Unification– und Recursive Unification Metapattern wird mit diesem Typ gearbeitet.
Zur Einschätzung der Verwendungsfähigkeit der Metapattern möchte ich folgend die sieben Metapatterns und ihre Eignung für bestimmte Entwurfsprobleme sowie die Zuordnung von Metapatterns zu Frameworks zusammenfassen, soweit dazu Informationen in dem Buch von Pree zu finden waren (die Seitenangaben beziehen sich alle auf [Pree94]).

Unification metapattern
Ausgezeichnet durch: Template– und Hook–Klasse in einer Klasse vereinigt. Keine dynamische Veränderung der Template–Hook–Beziehung möglich; Veränderung des Verhaltens statisch durch Vererbung.
Framework–Beispiele: Flexible Objekterzeugung

Connection metapattern
Ausgezeichnet durch: Austausch des Hook–Objekts zur Laufzeit durch Trennung in Template– und Hook–Klasse.
Framework–Beispiele: Spieler mit austauschbaren Rollen

1:N Connection metapattern
Ausgezeichnet durch: Austausch der Hook–Objekte selektiv zur Laufzeit durch Trennung in Template– und Hook–Klasse.
Framework–Beispiele: Publisher–Subscriber–Framework

1:1 Recursive Connection metapattern
Ausgezeichnet durch: (Temporäres) Hinzufügen von Verhalten zu Hook–Klassen, ohne die Basisklasse zu verändern (S.139ff). Listen von Objekten erzeugen, Message forwarding.
Framework–Beispiele: Wrapper (S.164ff)

1:N Recursive Connection metapattern
Ausgezeichnet durch: Baumstrukturen von Objekten erzeugen, Message forwarding. Hook–Objekte können nur Blätter im Baum sein; Template–Objekte können innere Knoten und Blätter sein. Hierarchien mit Manager– und Worker–Objekten (S.133ff).
Framework–Beispiele: Item–Container–Framework

1:1 Recursive Unification metapattern
Ausgezeichnet durch: Listen von Objekten erzeugen, Message forwarding. Die Template/Hook–Objekte können beliebige Plätze in der Liste einnehmen.
Framework–Beispiele: Konverter–Framework (S.166f)

1:N Recursive Unification metapattern
Ausgezeichnet durch: Baumstrukturen von Objekten erzeugen, Message forwarding. Template/Hook–Objekte können beliebige Plätze im Baum einnehmen. Die entstehenden Hierarchien haben keine Manager–Worker–Auszeichnung der Objekte. Diese Unterscheidung wird also entweder nicht benötigt oder ist an dieser Stelle nicht sinnvoll (S. 133ff).
Framework–Beispiele: Grouped–Item–Framework (S.135f, S.164)

5.2.5 Anwendung der Metapatterns am Beispiel ET++

Anhand der konkreten Adaption des ET++–Frameworks zur Konstruktion eines einfachen Hypertext–Editors stellt Pree den Einsatz der Metapattern dar. Diese Darstellung läßt meiner Meinung nach etwas zu wünschen übrig, da sie unvollständig und zu unzusammenhängend ist. Bei der nachfolgenden Beschreibung der konkreten Implementation wird der Bezug zu den mit Metapatterns dokumentierten Komponenten nicht hergestellt. Dies liegt sicherlich aber auch an der schlichten Größe und Komplexität auch schon eines kleineren Beispiels – immerhin wird hier der Entwurf eines vollständigen Hypertext–Editors inklusive Sourcecode dargestellt. Die Dokumentation mittels eines vollständigen Hypertextes und eines entsprechenden Editors würde sicherlich einen besseren Eindruck hinterlassen. So ist schlußendlich eine Einschätzung der Qualität des Metapattern–Ansatzes nicht möglich, da momentan auch keine aktuellere Literatur verfügbar ist.

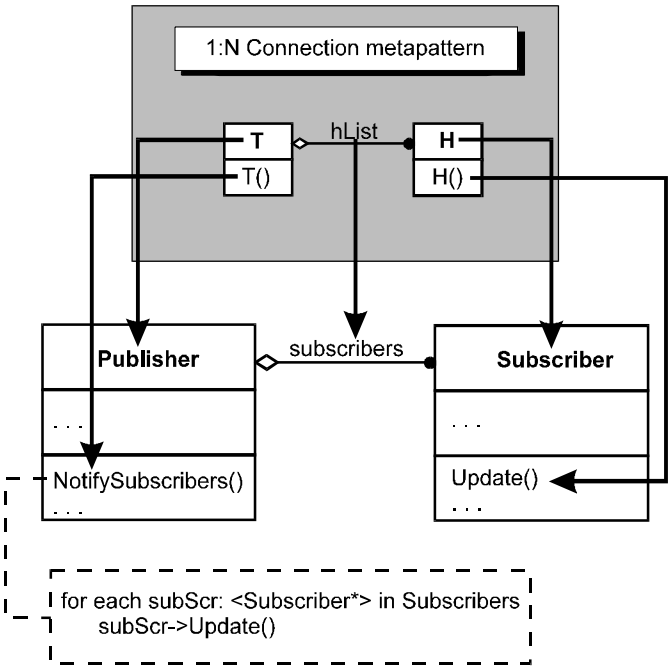

Abbildung 9 ([Pree94], S.162): Metapattern annotiert am Publisher–Subscriber–Pattern

5.2.6 Werzeugunterstützung zur Dokumentation mit Metapatterns

Das adäquate Werkzeug, um Frameworks zur Dokumentation mit Metapatterns zu annotieren, ist ein Hypertext–System (vgl. [Pree94], S.167f). Die Dokumentation und das hierfür konfigurierte Hypertext–System (*metapattern browser*, vgl. [Pree94], S.170) stellt eine Einheit dar. Der Autor sollte die halbautomatische Generierung des Dokumentations–Templates (graphische oder textuelle Notation des Metapattern) auslösen können, nachdem er die entsprechenden Klassen und Methoden selektiert hat. Pree beschreibt informell einige Anforderungen an einen Metapattern–Editor: Die Metapatterns werden durch Links (Abbildung 9) mit den korrespondierenden Klassen, Methoden und Beziehungen verbunden. Die selben Metapatterns werden so sicherlich mehrmals mit verschiedenen Komponenten des Frameworks verbunden und kategorisieren hierdurch gleichzeitig den Entwurf eines Frameworks. Der Benutzer des Editors kann die zu einem Metapattern annotierten Framework–Komponenten durch ein Menü auswählen. Die Annotationen der Links können auf verschiedene Repräsentationen des Frameworks verweisen. So ist es denkbar, das als Klassendiagramm in Booch–Notation vorliegenden Framework mit Metapatterns zu annotieren, genauso wie den Sourcecode. Auch Annotationen in die Objektdiagramme zur Visualisierung des Forwarding und des Message–passing sind denkbar.

5.3 Resümee: Entwicklungsmethoden und Entwurfsmuster

Entwurfsmuster sollten nicht überbewertet werden. Sie sind keine Alternative zu einer Entwurfsmethode, sondern als Ergänzung zu begreifen. In Verbindung mit einem Framework–zentrierten Entwurf kann eine auf maximale Wiederverwendbarkeit der Ergebnisse ausgerichtete Software–Entwicklung umgesetzt werden. Diese Auffassung vertritt Pree in seinem Buch [Pree94]. Ich möchte in diesem abschließenden Abschnitt auf die Integration des Entwurfsmuster–Ansatzes in eine Entwurfsmethode und damit verbunden in eine Software–Entwicklungsumgebung, eingehen.

Pree zitiert Weinand [Wei92] mit der Aussage, daß die Verwendung eines Application Frameworks wie ET++ gegenüber des Einsatzes einer Graphik Toolbox bei der Entwicklung einer Applikation dazu führt, daß die Menge des zu schreibenden Sourcecodes um mehr als 80 %

reduziert werden kann. Aus dieser Sicht ist die Framework–basierte Software–Entwicklung ein Durchbruch.
Zunächst noch einmal die Unterscheidung in:

- **Verwendung** eines Frameworks zur Adaption für eine konkrete Applikation – hier ist die Dokumentation des Frameworks mit z.B. Cookbooks ein praktikabler Ansatz.
- Die **Neuentwicklung**, bzw. **strukturelle Veränderung** eines Frameworks wird meist durch Extraktion von Komponenten und sicherlich durch Exploration bestehender Frameworks eingeleitet. Die Entwicklungsarbeit kann hierbei effektiv durch Verwendung einer Dokumentation durch Gamma–Pattern und Metapatterns mit einer entsprechenden Werkzeugunterstützung vereinfacht werden.

Das generelle Vorgehen bei der Neu–Entwicklung eines Frameworks und der anschließenden Dokumentation mit Patterns nach Pree besteht darin, daß zunächst die initialen *hot spots* identifiziert werden müssen, also die Komponenten des Frameworks, die adaptierbar sein sollen. Zur Identifikation ist domänenspezifisches Wissen erforderlich – Entwurfserfahrung allein reicht nicht aus. Die weitere Entwicklung gestaltet sich iterativ in einem Zyklus, der sämtliche Phasen des Software–Entwicklungs–Prozesses und auch den praktischen Einsatz des Frameworks zur Verifikation des Entwurfs einschließt. In jedem Zyklus entsteht ein flexibleres Framework, das unterschiedlichen Anforderungen besser gewachsen ist. Neue hot spots werden hierbei identifiziert, eventuell auch grundlegende Änderungen der Architektur nötig. Es besteht somit also ein Unterschied zum klassischen Entwurfsprozeß. Die bekannten Entwurfsmethoden sind für diese Aufgabe nicht ausreichend geeignet. Pree stellt das an der OMT von Rumbaugh [Rum+93] exemplarisch dar (vgl. [Pree94], S.225f). Er sieht hier einen Ansatzpunkt für die Metapattern. Mit deren Hilfe soll die Identifikation geeigneter Abstraktionen beschleunigt und so die Anzahl der Redesign–Zyklen verringert werden.
Wesentliche Inspiration hat sich Pree hier nach eigener Aussage bei B. Meyer [Mey90] geholt, der kontrastierend zum klassischen *Software–Lifecycle* sein Cluster–Modell setzt. Cluster sind die von Coad in seiner OOA–Notation als *subjects* (Themen) bezeichneten Gruppen zusammengehöriger Klassen. Das Cluster–Modell beinhaltet darüber hinaus eine Entwurfs-systematik: Cluster werden iterativ von der Spezifikation über Entwurf/Implementation und Validierung bis zur erneuten Generalisierung in Zyklen entwickelt. Dabei legt Meyer den Schwerpunkt auf das Testen der Pre– und Postconditions, mit dem Ziel, die Stabilität der Cluster zu verbessern. Die auch in Linz tätigen J. Sametinger und A. Stritzinger stellen in ihrem Artikel [Sam+95] einen ähnlichen Ansatz dar, der auch die Anfordernugen von Frameworks berücksichtigt.
Die Metapattern kommen außer zur reinen Dokumentation auch beim Redesign eines Frameworks zum Einsatz. Der Software–Ingenieur benutzt sie einerseits, um die hot spots zu dokumentieren und so die sich anschließenden Adaptionstests zu unterstützen. Aber auch die Analyse des Klassendiagrammes hinsichtlich der Fragestellung, ob Klassen weiter aufgeteilt oder getrennte Klassen zusammengeführt werden, kann durch die annotierten Metapatterns vereinfacht werden: Sind mehrere Abstraktionen in einer Klasse vereinigt – daran erkennbar, daß mehrere Metapatterns annotiert wurden, empfiehlt sich eventuell eine Aufteilung. Im umgekehrten Fall, daß eine Klasse für sich allein keine „Lebensberechtigung“ hat, kann eine Vereinigung (*Unification*) mit einer anderen Klasse sinnvoll sein.
Die vorgestellte Klassifizierung von Entwurfsentscheidungen nach dem Maß der gewünschten Flexibilität der hot spots (vgl. [Pree94], S.228ff) ist ein im Gegensatz zu den Gamma–Patterns unabhängig von konkreten Problemdomänen anwendbares Konzept: Je nach Anwendung von *Unification*, *Connection* und/oder *Recursive Unification/Connection Metapatterns* entstehen unterschiedliche Entwürfe.

Die durch Wahl eines bestimmten Metapattern festgelegte Entwurfsentscheidung macht die Eignung der Metapattern zur **Konstruktion** von Frameworks deutlich. Ich habe diesen Aspekt der Metapattern in meinen Ausführungen nicht weiter behandelt, da er für die Dokumentation mittels Metapattern keine Rolle spielt.
Zusammenfassend läßt sich sagen, daß die (Qualität der) Integration eines Metapattern–Editors in eine Software–Entwicklungsumgebung von grundlegender Bedeutung für die Verwendbarkeit des Metapattern–Ansatzes ist. Ich sehe in den im Abschnitt 3.6 (Hypertext und CASE) gemachten Aussagen zu einem Basissystem für eine Software–Entwicklungsumgebung den praktikablen Ansatzpunkt. Ein Basissystem, das auf die Dokumentenverwaltung als zentrale Aufgabe ausgerichtet ist und eine flexible Werkzeugkonstruktion und –integration ermöglicht, sollte als Grundlage für eine mit allen gewünschten Methoden–Editoren ausgestatteten Umgebung dienen. Entwurfsmuster–Editoren sind dann eine Komponente aus der Menge der zur Generierung der Dokumente vorhandenen Werkzeuge. Wichtig hierbei, so meine ich, ist dabei die in Abschnitt 4.3 vorgestellte Sichtweise der durch Vorgehensmodelle **statisch** und während des Entwurfsprozesses **halbautomatisch dynamischen Vorstrukturierung** der Dokumente.

6 Glossar

Außer den hier als Auswahl aufgeführten Begriffsdefinitionen sind einige weitere in den Folien von F. Buschmann zur OOP'95 zu finden [Busch95]. So z.B. die Begriffe Komponente, Beziehung, funktionale und nichtfunktionale Eigenschaften, Software–Architektur etc.

Abstrakte Klasse Der primäre Zweck einer abstrakten Klasse ist die Definition eines Interface oder Protokolls (vgl. [Gam+95]). Einige oder alle Methodenimplementierungen werden Subklassen überlassen (z.B. in C++ durch die Deklaration einer Methode als *pure virtual*). Somit können keine Instanzen von einer abstrakten Klasse erzeugt werden.

CORBA Der *Common Object Request Broker* ist ein von der OMG (*Object Management Group*, http://www.omg.org) definierter Standard für verteilte Objekte. Die OMG ist ein 1989 gegründetes Konsortium der Softwareindustrie, der mittlerweile mehr als 470 Firmen angehören. 1990 erschien die erste gemeinsame Spezifikation, die OMA (*Object Management Architecture*), deren Basis ein erweiterbarer Software–Bus, der ORB (*Object Request Broker*) ist. Der ORB wurde 1991 in der Spezifikation CORBA 1.1 standardisiert. Darin werden unter anderem gemeinsame *object services* definiert, die Standardaufgaben, wie z.B. Namensgebung, Lebenszyklus von Objekten, Persistenz, Ereignisse, gemeinsame Fähigkeiten wie Verbunddokumente (siehe OpenDoc) oder Email etc. regeln.
Die Objekte kommunizieren hierbei lokal oder über den Software–Bus transparent im Netz. Der ORB leitet dabei Anfragen an ein Objekt weiter, überträgt Parameter, ruft entsprechende Objektmethoden auf und leitet das Ergebnis an den Aufrufer zurück.
Kommerzielle Implementierungen, die auf CORBA 1.1 basieren sind z.B. das SOM/DSOM–Framework von IBM [Troy+95] oder Distributed Smalltalk von Hewelett–Packard [Eis+94].
Ende 1994 wurde CORBA 2.0 verabschiedet. In dieser Erweiterung des Standards geht es hauptsächlich um die Kommunikation zwischen verschiedenen ORB's unterschiedlicher Hersteller auf heterogenen Plattformen (auch über das Internet). Die *object facilities* wurden und werden erweitert, so z.B. um Concurrency, Transaktionsverwaltung, Zeitsynchronisation in verteilten Umgebungen, Lizensierung, Abfragesprachen, Replikation etc. (siehe z.B. [Stal95]). Weitere Literatur zu CORBA ist im Literaturverzeichnis aufgeführt.

DDE (Dynamic Data Exchange) Ist ein Windows–Kommunikationsprotokoll, mit dem Daten zwischen DDE–fähigen Applikationen ausgetauscht und Befehle in anderen Windows–Anwendungen ausgeführt werden können. Die Rollen der Applikationen bei der Kommunikation teilen sich in die Client–Anwendung, die die Kommandos versendet und die Server–Applikation, die auf die empfangenen Kommandos reagiert, auf. Eine Applikation kann auch ausschließlich Client–Funktionalität beinhalten, genauso wie Applikationen, die beide Rollen einnehmen können möglich sind. Letztere können auch eine wechselseitige DDE–Verbindung initiieren.
ToolBook ist eine reine Client–Applikation. Es lassen sich mit entsprechenden Skripten z.B. Daten aus einer Tabellenkalkulation oder aus einer Datenbank abrufen.

Entwurfsmuster / Designpattern / Pattern Vorgefertigtes (*ready–to–use*) Design in einem extrem wiederverwendbaren Zustand dargestellt einerseits und andererseits (aus der Sicht derer, die Entwurfsmuster „fertigen") Ergebnisdokumentation einer durch mehrere Designzyklen gereiften Softwarearchitektur.

Framework Siehe den Abschnitt Frameworks auf Seite26.

Frozen spot Komponente eines Frameworks, die nicht zur Adaption entworfen wurde.

Forwarding / Propagierung von Botschaften Meist durch die Ausnutzung der Vererbungsrelation werden Methodenaufrufe an andere Objekte weitergereicht. Dieser Mechanismus wird in den 1:1 und 1:N Recursive Unification und Recursive Connection Metapatterns benutzt ([Pree94], S.128ff), der Observer von Gamma ([Gam+95], S.293) basiert darauf, ebenso das MVC–Framework [Kras+88]. (Siehe auch [Gam92], S. 128f)

Hot spot Diese Komponenten eines Frameworks müssen zur Verwendung adaptiert werden.

Message Passing Wird meist als ein anderer Begriff für den Aufruf einer Methode einer Klasse verstanden. Der Begriff dient so zur Assoziation mit der Vorstellung, daß Objekte untereinander mittels Nachrichten kommunizieren (vgl. [Pree94], S.253).
Bei genauerem Hinsehen muß zwischen dem dynamischen und dem statischen Aufruf einer Methode unterschieden werden.
Das Versenden einer Nachricht (der Methodenaufruf) an ein Objekt kann von diesem Objekt auch *nicht* verstanden werden. Dies gilt für den Aufruf von überladenen Methoden. Ist eine Methode gerade nicht überladen worden, so tritt dieser Fall ein und die Nachricht wird entlang der Vererbungshierarchie weitergereicht.
Im statischen Fall bezieht sich ein Methoden*aufruf* explizit auf eine vorhandene Methode (dieser Sachverhalt ist auch vom Compiler überprüfbar). In diesem Fall trifft die eingangs angeführte Einschätzung zu.

OLE (Object Linking and Embedding) Ein von Microsoft entwickelter und in Windows realisierter Standard, der es ermöglicht die Funktionalität einer Applikation anderen Applikationen zur Verfügung zu stellen (Interoperabilität). Die Applikationen spielen bei dieser Verbindung verschiedene Rollen: In der Server–Anwendung wird ein Objekt vom Benutzer oder automatisch (via DDE) generiert, bzw. verändert und in der Client–Anwendung wird das Objekt verwendet, also z.B. angezeigt. Die Speicherung des Objekts kann in einem Dokument des Clients erfolgen (eingebettetes Objekt), in einer separaten Datei (verknüpftes Objekt) oder ebenso in der Client–Anwendung diesmal als statisches Objekt. Letzteres hat allerdings zur Folge, daß die Referenz auf den Server verloren geht und somit die Möglichkeit, das Objekt zu bearbeiten.
Im neueren OLE 2.0–Standard kommt hier neben anderem als Erweiterung die *Inplace–Activation* hinzu, die eine grundlegende Veränderung der Sichtweise ermöglicht: Statt dem Export von Teilen eines Dokumentes in andere Applikationen zur dortigen Bearbeitung wird je nach selektiertem Bereich eines Dokumentes die entsprechende Funktionalität als Menüzeile eingeblendet. An Stelle der applikationszentrierten Denkweise kann hier also die **dokumentenzentrierte** angeboten werden, (kann deshalb, weil bisher kaum eine Applikation diese Funktionalität bietet – außer den hauseigenen, Word und Excel – und auch der OLE–Standard selbst noch einiges zu wünschen übrig läßt).
Die DDE–Komponente wurde in OLE2 unter dem Namen *OLE Automation* integriert, wohl um die Übersichtlichkeit zu erhöhen.
Für nähere Informationen über OLE2 siehe [Micro93] .

OpenDoc Von IBM, Apple und Novell entwickelter Standard, der in Konkurenz zu Microsofts OLE steht und von der herstellerunabhängigen Einrichtung CI Labs (*Component Integration Laboratories*, http://www.cilabs.org) weitergetrieben wird. Über Rechnergrenzen

hinweg portable Verbunddokumente stehen im Zentrum dieses Ansatzes. Sie bestehen aus beliebig geschachtelten Objekten (*Parts*), wie z.B. Text, Tabellen, Graphik, Ton, Animation etc., die von verschiedenen, auf jeweils einen bestimmten Objekttyp spezialisierten Applikationen (*Part–Handler*) bearbeitet werden. Der Vorteil besteht in der Vermeidung von Redundanz zwischen den Applikationen (die Bildverarbeitung benötigt jetzt keine Rechtschreibkorrektur mehr ...) aber vor allem in ganz neuen Möglichkeiten der Verwaltung von komplexen Dokumenten, wie sie z.B. in der Software–Entwicklung auftreten. Die Steuerung von Konsistenzerhaltung, Mehrbenutzerzugriff und Versionsverwaltung ist zentralisiert in einer Komponente wie OpenDoc, die auf Betriebssystemebene installiert ist. Konkret ist OpenDoc eine durch die Verwendung von SOM/DSOM sprachenunabhängig definierte Klassenbibliothek, die auf einen CORBA–konformen ORB (*Object Request Broker*) aufsetzt.
Eine Übersicht über OpenDoc ist in [Troy+95], S.382ff zu finden. Dort werden auch die auftretenden Probleme (plattformübergreifendes Dateiformat, nicht–rechteckige Objekt–Bereiche, gemeinsame Untermengen von Kommandos, einheitliches Layout der Part–Handler, Unterscheidung in Editoren und reine Viewer, etc.) angesprochen.

Protokoll Protokoll beschreibt die Erweiterung des Interface–Konzepts, das aus der Begriffswelt der abstrakten Datentypen (ADT) kommt, um erlaubte Anforderungs–Sequenzen. Es wird nicht mehr nur die Menge der öffenlichen Funktionen und ihre Parameter beschrieben, sondern z.B. mit Hilfe von Interaktions– oder Szenariodiagrammen (vgl. [Rum+93], S.106f) eine zulässige Abfolge von Methodenaufrufen, die dann als Protokoll bezeichnet wird.

Template / Rahmen Dieser Begriff ist zumindest zweifach überladen. Zum einen werden im C++–Kontext die generischen Typen so bezeichnet – Klassen oder Funktionen, die eine vordefinierte Semantik auf einen erst bei der Deklaration einer Variablen übergebenen Typ anwenden. Beispiele hierfür sind z.B. Listen–Templates, die Daten beliebiger Typen verwalten.
Zum anderen werden im Framework–Kontext die festgelegten Methoden, bzw. Klassen als Template–Methoden/Klassen im Gegensatz zu den vom Anwender eines Frameworks zu adaptierenden Hook–Methoden/Klassen bezeichnet (vgl. [Pree94], S.107ff und S. 118ff und das Gamma–Pattern *Template Method* in [Gam+95], S.325ff).

7 Literaturverzeichnis

[Adob94] Adobe Systems Inc.: *Adobe Acrobat CD Sampler*. München: Adobe 1994.

[Alex77] Alexander, C.: *A Pattern Language*. 6. Auflage New York: Oxford University Press 1977.

[Asym94] Asymetrix: *Toolbook Benutzerhandbuch*: 1994.

[Beck+94] Beck, K., Johnson, R.: *Patterns Generate Architectures*. ECOOP'94

[Bisch95] Bischofberger, W. R.: *Frameworkbasierte Softwareentwicklung*. In: Proceedings of OOP'95 München. S. 41–46.

[Busch95] Buschmann, F.: *Software–Architektur mit Entwurfsmustern*. Vortragsfolien von der OOP'95 München.

[Coad+91] Coad, P., Yourdon, E.: *Object–oriented Design*. Englewood Cliffs: Yourdon Prees, Prentice–Hall Inc. 1991

[Coad95] Coad, P.: *Object Models: Strategies, Patterns and Applications*. o.O.: Yourdon Press 1995.

[Eis+94] Eisenecker, U.W., Hirschfeld, R.: *Eine Einführung in Hewlett–Packards Distributed Smalltalk*. In: OBJEKTspektrum 5/1994. S. 20–28.

[Gam92] Gamma, E.: *Objektorientierte Software–Entwicklung am Beispiel von ET++*. Berlin: Springer 1992.

[Gam+93] Gamma, E. et. al.: *Design Patterns: Abstraction and Reuse of Object–Oriented Design*. In: Proceedings of the ECOOP 1993. S. 406–431.

[Gam+95] Gamma, E. et. al.: *Design Patterns – Elements of Reusable Object–Oriented Software*. Reading, Massachusetts: Addison–Wesley 1995.

[Gloor90] Gloor, P. A.: *Hypermedia–Anwendungsentwicklung*. Stuttgart: Teubner 1990.

[John92] Johnson, R.: *Documenting Frameworks using Patterns*. In: ACM SIGPLAN Notices 27(19), 1992. S.63ff.

[Kras+88] Krasner, G., Pope, S.: *A Cookbook for Using the Model–View–Controller User Interface Paradigm in Smalltalk–80*. In: JOOP 8/9, 1988. S.26ff.

[Lehn94] Lehner, F.: *Software–Dokumentation und Messung der Dokumentationsqualität*. München: Hanser 1994.

[Mey90] Meyer, B.: *Lessons from the design of the Eiffel libraries*. In: Communications of the ACM 33(9) 1990.

[Micro93] Microsoft OLE2 Design Team: *OLE 2.01 Design Specification*. Microsoft Corporation, 1993 (In den Online–Dokumenten zu MS Vicual C++ V1.5 enthalten)

[Micro94] Microsoft Visual C++: *Introducing Visual C++*. Document No. DB57154–0694: Microsoft Corporation 1994.

[Niel93] Nielsen, J.: *Hypertext and Hypermedia*. Cambridge: Academic Press Inc. 1993.

[Pree94] Pree, W.: *Design Patterns for Object–Oriented Software Development*. Reading, Massachusetts: Addison–Wesley 1994.

[Rum+93] Rumbaugh, J. et al.: *Objektorientiertes Modellieren und Entwerfen*. München: Hanser 1993.

[Sam+95] Sametinger, J., Stritzinger, A.: *Exploratory Software Development with Class Libraries*. WWW–Server des Instituts für Wirtschaftsinformatik, Johannes Kepler Universität, Linz.

[Stal95] Stal, M.: *Der Zug rollt weiter*. In: iX 5/1995. Heise–Verlag, Hannover.

[Troy+95] Troyer, K., Loviscach, J.: *Patchwork*. In: c't 4/1995. Heise–Verlag, Hannover.

[Vos94] Vossen, G.: *Datenmodelle, Datenbanksprachen und Datenbank–Management–Systeme*. 2. Auflage Bonn: Addison–Wesley 1994.

[Wei92] Weinand, A.: *Objektorientierte Architektur für graphische Benutzeroberflächen*. Berlin: Springer 1992.

Weitere Literatur auf die ich nicht explizit verwiesen habe, die im Kontext aber auch interessant ist:

Behme, H.: *Vorfrühling, Objektorientierte Solaris–Erweiterung: Distributed Objects Everywhere*. In: iX 6/1994. S. 38–44. Heise–Verlag, Hannover.

Birrer, A., Eggenschwiler, T.: *Frameworks in the Financial Engineering Domain – An Experience Report*. In: Proceedings of the ECOOP 1993. S. 21–35

Bogdan, F., et al.: *Entwicklung industrieller Anwendungssysteme in einer CORBA–basierten Softwarearchitektur*. In: Proceedings of OOP'95 München. S. 115–117.

Burger, E.: *Communicate it, PDO: Nexts portable Lösung für objektorientierte Kommunikation im heterogenen Netz*. In: iX 6/1994. S. 46–52. Heise–Verlag, Hannover.

Buschmann, F., Meunier, R.: *Software–Konstruktion mit Mustern*. In: OBJEKTspektrum 5/1994. S. 48–57.

CORBA 2.0/Interoperability: *Universal Networked Objects*. OMG TC Document 95.3.xx [REVISED 1.8 jm], 20.3.1995.

Debatin, F.: *Über einen praktischen Einsatz von Entwurfsmustern*. In: OBJEKTspektrum 4/1995. S. 62–64.

Drespling, W.: *Arbeiten mit verteilten Objekten in CORBA–Implementierungen*. In: OBJEKTspektrum 5/1994. S. 30–32.

Freitag, B.: *A Hypertext–Based Tool for Large Scale Software Reuse*. o.A.

Puder, A.: *Objekt im Objekt im Objekt, Microsofts Object Linking and Embedding für zusammengesetzte Objekte*. In: iX 6/1994. S. 54–60. Heise–Verlag, Hannover.

Pürckhauer, C.: *OpenDoc – Basis einer Komponentenarchitektur*. In: OBJEKTspektrum 4/1995. S. 49–52.

Stal. M.: *Object Request Broker und OLE2 im Vergleich*. In: Proceedings of OOP'95 München. S. 253–256.

Stal, M.: *Componentware – die Vision von verteilten Objekten*. In: OBJEKTspektrum 4/1995. S. 77–81.

Vlissides, J.: *Using Design Patterns: Elements of Reusable Architectures*. In: Proceedings of OOP'95 München.

Wagner. P.: *CORBA 2.0 – Details des Interoperabilitätsstandards der OMG*. In: OBJEKTspektrum 3/1995 S. 62–70 und 4/1995 S. 53–55.