Konzeptionelle und technische Weiterentwicklung eines objektorientierten Frameworks nach der Werkzeug-Material-Metapher

Diplomarbeit

Universität Hamburg
Fachbereich Informatik
Arbeitsbereich Softwaretechnik

Ulfert Weiß

Betreuung

Prof. Dr. Heinz Züllighoven
Dr. Ingrid Wetzel

Diese Diplomarbeit wurde am Fachbereich Informatik der Universität Hamburg zur teilweisen Erfüllung der Anforderungen zur Erlangung des Titel Diplom-Informatiker eingereicht.

Erklärung

Ich versichere hiermit, diese Arbeit selbständig und unter ausschließlicher Zuhilfenahme der in der Arbeit aufgeführten Hilfsmittel erstellt zu haben.

Hamburg, den 10.06.1997

Ulfert Weiß
Hirschbergstraße 5
23617 Stockelsdorf

Prof. Dr. Heinz Züllighoven *(Erstbetreuer)*
Dr. Ingrid Wetzel *(Zweitbetreuerin)*

Fachbereich Informatik
Arbeitsbereich Softwaretechnik
Universität Hamburg
Vogt-Kölln-Str. 30
22527 Hamburg

Gliederung

Kapitel 1 Einleitung und Motivation

„One of the next major steps in object-oriented design and programming is framework design and programming.“

Ted G. Lewis [Lewis95]

Objektorientierte Softwareentwicklung hat den Anspruch, durch wiederverwendbare Komponenten robuste, wartbare Software zu geringen Produktionskosten erstellen zu können. Dieses Ziel kann erreicht werden durch Rahmenwerke (Framework[1]), insbesondere Application-Frameworks[2] für bestimmte Anwendungsgebiete (Domänen), in die der Anwendungsentwickler nur noch die jeweils spezifischen Komponenten seiner Applikation einfügen muß.

Frameworks bestehen aus abstrakten und implementierten Klassen. Durch Vererbung werden Klassenhierarchien gebildet. Das wesentliche Merkmal objektorientierter Frameworks gegenüber aus prozeduralen Sprachen bekannten Funktions- und Bausteinbibliotheken ist ein bereits definiertes Modell der dynamischen Beziehungen verschiedener Klassen zur Laufzeit (vgl. [Lewis95]). Konsequenz dieser Konstruktion ist die Umkehrung des Kontrollflusses, bekannt als Hollywood-Prinzip (*„don't call us, we call you“* [PLoP95]). Der Anwendungsentwickler entwirft die Komponenten seiner Anwendung, ist aber nicht für die richtige Aufrufreihenfolge der Anwendungs- und Framework-Komponenten verantwortlich.

Besonders Application-Frameworks werden für jeweils einen bestimmten Anwendungsbereich entwickelt und stellen eine Programmierschnittstelle auf sehr hoher Ebene zur Verfügung. Diese Frameworks sind bereits *„vollständige Anwendungen“* [Lewis95] bzw. beinhalten für eine Gruppe gleichartiger Anwendungen die komplette Infrastruktur. Der Entwickler kann sich somit auf die spezifischen Teile seines Projektes konzentrieren und sich dabei auf eine erprobte Architektur des System verlassen. Neben einer kürzeren Entwicklungszeit für einzelne Komponenten ermöglichen Frameworks eine hohe softwaretechnische Qualität der Produkte. Innerhalb einer Organisation stellt das verwendete Framework Gemeinsamkeiten aller auf dieser Basis entwickelten Anwendungen sicher und erleichtert somit sowohl die Wartung und Weiterentwicklung der einzelnen Anwendungen als auch deren Benutzung infolge gleichartiger Bedienung.

---

[1] In dieser Arbeit werde ich überwiegend den gebräuchlichen englischen Begriff „Framework“ verwenden.

[2] Ein gebräuchlicher deutscher Begriff ist mir nicht bekannt. Allerdings verwendet [BBE95] hier „Anwendungsumgebung“.

Für die Konstruktion qualitativ hochwertiger Software ist es hilfreich, ein Leitbild [GKZ93] und passende Metaphern zu verwenden. Application-Frameworks können nach diesen Metaphern entworfen werden und eine entsprechende Struktur der Anwendung vorgeben. Benutzern des Frameworks helfen die verwendeten Metaphern beim Verständnis der Framework-Komponenten, den Entwicklern des Frameworks liegt eine Grundidee für die Strukturierung vor. In dieser Arbeit werde ich mich an den Metaphern der am Arbeitsbereich Softwaretechnik vertretenen WAM-Methode orientieren.

Grundlegende Leitmetaphern dieser Methode sind Werkzeug und Material, die softwaretechnisch durch Werkzeug-, Aspekt- und Materialklassen modelliert werden [GKZ93]. Werkzeuge bestehen aus Funktions- (FK) und Interaktionskomponente (IAK). Somit wird eine Trennung zwischen Werkzeugfunktionalität und der Benutzungsoberfläche hergestellt. Diese Entkopplung erlaubt auch die Verwendung derselben FK mit mehreren IAK 's (vgl. MVC [Reenskaug95]). Zusätzlich wird die Unabhängigkeit vom jeweils verwendeten GUI[3] durch Interaktionstypen (IAT) erreicht. Weitere Metaphern unterstützen kooperative Arbeit sowie Persistenz. Ich werde zeigen, wie die WAM-Methode die Strukturierung eines Frameworks und der darauf aufbauenden Anwendungen erleichtert.

Konkretes Beispiel für diese Arbeit ist ein am Arbeitsbereich Softwaretechnik entwickeltes C++ Application-Framework, an dessen Weiterentwicklung ich beteiligt bin. Die Beschreibung dieser gemeinschaftlichen Entwicklung verschiedener Studenten und wissenschaftlicher Mitarbeiter und eine Bewertung der Ergebnisse ist Teil dieser Arbeit. Angestrebt sind lose gekoppelte Komponenten, die der Anwendungsentwickler jeweils bei Bedarf einsetzen und kombinieren kann. Diese Bestandteile eines nach WAM konstruierten Frameworks werde ich identifizieren und eine Zerlegung in Bibliotheken vorschlagen.

Ziel der Strukturierung ist neben unabhängiger Weiterentwicklung der einzelnen Bestandteile und korrekter Integration zukünftiger Komponenten auch eine Anleitung für die Benutzung des Frameworks. Ich werde auf Aspekte der Dokumentation eingehen, auf den Stellenwert ausführbarer Beispiele, Visualisierung der Dynamik und die Beschreibung mittels bekannter sowie neu identifizierter Design Pattern. Darüber hinaus bietet ein gut strukturiertes Framework dem Benutzer die Möglichkeit, sich jeweils nur mit einem überschaubarem Ausschnitt zu befassen und ggf. auch jeweils nur diese Teile zu verwenden.

Aufbau des Textes

Kapitel 2 dient zur Klärung der im Text verwendeten Begriffe. Es erfolgt eine Klassifizierung verschiedener **Arten von Frameworks** und die Abgrenzung gegenüber Bausteinbibliotheken. Arten der Wiederverwendung und in der Literatur übliche Einteilungen für Frameworks werden erläutert und bewertet.

---

[3] GUI = graphical user interface, grafische Benutzungsoberfläche

Design Pattern als aktuelles Wiederverwendungskonzept werden ausführlich diskutiert, neuere Ausprägungen und Schwerpunkte vorgestellt. Den Bezug zur Framework-Konstruktion hebe ich hervor, auch die Darstellung der **WAM-Methode** ist hierauf ausgerichtet. Das Ende dieses Kapitels bildet eine Zusammenfassung der begrifflichen Unterscheidungen.

Aufbauend auf diesen Ergebnissen folgt in den Kapiteln 3 und 4 eine Einordnung verschiedener Application-Frameworks. Im Überblick werde ich einige **populäre Frameworks** vorstellen (Kapitel 3), Fortschritte der Framework - Technik exemplarisch veranschaulichen und einen Vergleich zu dem **WAM-Framework** BibV30 vorbereiten.

Zentrales Anliegen in Kapitel 4 ist die Darstellung der Entwicklung dieses Application-Frameworks. Einzelne Punkte der konzeptionellen und technischen **Weiterentwicklung** werden erläutert und deren Bedeutung diskutiert. Nach einer Kurzdarstellung diverser von verschiedenen Personen durchgeführter Änderungen, werde ich die von mir aus anderem Kontext in das Framework portierte **Grafik - Abstraktion** detailliert betrachten.

Vor dem Hintergrund dieser konkreten Erfahrungen setze ich mich im 5. Kapitel mit der **Strukturierung** von Bibliotheken auseinander. Meinen Vorschlag mit Schwerpunkt auf WAM- Application-Frameworks und konkreter Umsetzung in der BibV30 vergleiche ich mit Ansätzen in der Literatur und kläre die Frage, inwieweit ein Strukturschema für WAM-Frameworks in andere Zusammenhänge übertragbar ist.

Abschließend fasse ich den Stand der Framework-Entwicklung zusammen und werfe einen Blick in die Zukunft wiederverwendbarer Softwarekomponenten.

Textkonventionen und Darstellungsmittel

> *Längere Zitate erhalten eigene Absätze in kursiver Schrift, die erkennbar eingerückt werden.*

Innerhalb des Textes werden kurze Zitate ebenfalls *kursiv* gedruckt, wichtige Begriffe und Passagen teilweise durch **Fettdruck** hervorgehoben.

> Kapitel 2 faßt Ergebnisse der Diskussion in Form eigener Begriffsdefinitionen zusammen. Diese werden als Block abgesetzt, definierte Begriffe sind **fett** gedruckt, wesentliche Stichwörter *kursiv* hervorgehoben.

Grafiken werden jeweils bei der ersten Verwendung näher erläutert.

Kapitel 2 Begriffe und Literatur

In diesem Kapitel werden die grundlegenden Begriffe eingeführt und erläutert. Verschiedene Sichtweisen und Ansätze innerhalb der Literatur ([JF88], [Gamma92], [BBE95], [Lewis95], [Pree94], etc.) werden betrachtet und gegenübergestellt. Diese Diskussion motiviert eigene Definitionen für die Verwendung der Begriffe innerhalb dieser Diplomarbeit. Die Abgrenzung gegenüber anderen Autoren soll ein Grundverständnis für die weiteren Ausführungen schaffen und meinen Standpunkt klarlegen.

Die Definitionen stellen gleichzeitig Anhaltspunkte zur Klassifikation von Frameworks zur Verfügung und ermöglichen damit eine Einordnung spezieller Application-Frameworks in den folgenden Kapiteln.

2.1 Framework - Begriffe

Klassisches Mittel für die Wiederverwendung sind sogenannte Funktionsbibliotheken[4] in prozeduralen Programmiersprachen. Häufig benötigte Algorithmen werden in Bibliotheken zur Verfügung gestellt und müssen nicht für jede Anwendung neu entwickelt werden. Standardbeispiele sind String - Manipulation sowie verschiedene Such- und Sortierverfahren. Bei der Benutzung sind keine Abhängigkeiten zu berücksichtigen, es wird jeweils die einzelne Funktion wiederverwendet.

Objektbasierte Programmiersprachen erweitern die Möglichkeiten, wiederverwendbare Elemente in Bibliotheken anzubieten. Sprachen wie Modula-2 und Ada führen ein Modulkonzept ein (Packages in Ada), das eine bessere Strukturierung der angebotenen Funktionalität und die Festlegung der Sichtbarkeit für den Anwender ermöglicht [Gamma92]. Einen wesentlichen Nachteil stellt die Abgeschlossenheit dieser Module gegen nachträgliche Erweiterungen und Anpassungen dar.

Grundlage dieser Arbeit sind getypte objektorientierte Programmiersprachen, speziell C++. [Wegner90] unterscheidet objektbasierte, klassenbasierte und objektorientierte Programmiersprachen. Modula-2 und Ada als Beispiele objektbasierter Sprachen unterstützen zwar Objekte, aber keine Klassen:

> *„But Ada is not class-based because its objects (packages) do not have a type and cannot therefore be passed as parameters, be components of arrays or records, or be directly pointed to by pointers.“*

[4] oder auch Subroutinen - Bibliotheken, vgl. [Gamma92]

Objektorientierte Sprachen erweitern wiederum die klassenbasierten Sprachen um das Konzept der Vererbung. Klassen können in objektorientierten Sprachen nach dem offen-geschlossen-Prinzip durch Vererbung angepaßt und erweitert werden [Meyer88]. Diese mächtigen Sprachmittel werden von vielen Autoren als Grundlage für die Realisierung wiederverwendbarer Softwarekomponenten betrachtet ([Gamma92], [BBE95]). Technische Möglichkeiten allein sind allerdings noch keine Garantie für wiederverwendbare Softwarekomponenten.

[Pree94] bezeichnet den Vererbungsmechanismus objektorientierter Programmiersprachen als Grundlage des „*programming by difference*“. Entsprechend dem offen-geschlossen-Prinzip nach [Meyer88] können vorhandene Klassen einfach durch Überschreiben einzelner Methoden angepaßt werden, während die übrigen Teile direkt weiter verwendet werden.

In objektorientierten Sprachen werden Funktionsbibliotheken durch Klassenbibliotheken ersetzt. Sammlungen von Bausteinklassen können dabei als erster Schritt angesehen werden. Einzelne Klassen innerhalb der Bibliothek sind voneinander (weitgehend) unabhängig und können direkt benutzt werden, eigene Klassen müssen nicht abgeleitet werden.

Als typische Beispiele für Bausteinklassen nennt [Gamma92] Datenstrukturen und Dialogelemente. So kann eine Bibliothek Listen, Arrays und einen Stack anbieten, die von anderen Klassen direkt verwendet werden. Innerhalb der Bibliothek können die Klassen voneinander abgeleitet werden und eine Hierarchie bilden, die dem Benutzer der Bibliothek durch *Klassifizierung* der Elemente einen besseren Überblick gibt. Vielfach wird jedoch Implementationsvererbung eingesetzt, um z.B. eine sortierte Liste von einer Liste abzuleiten (zu den Problemen dieses Vorgehens siehe [Traub95]).

Bausteinbibliotheken haben sich als Mittel für die Wiederverwendung „im Kleinen“ bewährt. Die Klassen lassen sich häufig und direkt wiederverwenden, das Zusammenspiel zwischen den Objekten muß die Anwendung aber jeweils neu realisieren. Implizite Annahmen betreff möglicher Kombinationen der Klassen erschweren den Anwendern die Benutzung komplexer Bausteinbibliotheken.

Frameworks haben das Ziel, Wiederverwendung „im Großen“ zu ermöglichen. Es wird nicht nur der Code einzelner Klassen wiederverwendet, sondern Entwurfsstrukturen. Ein Framework kann für eine Gruppe verwandter Problemstellungen eine allgemeine, abstrakte Lösung implementieren, dabei den Kontrollfluß einer kompletten Anwendung bestimmen und die einzelnen Komponenten miteinander verknüpfen. Der Benutzer ergänzt nur die spezifischen Teile seiner Applikation.

Nach [Pree94] sind für die Entwicklung von Frameworks neben der Vererbung besonders die abstrakten Klassen ein entscheidendes Sprachmittel. Die geeignete Kombination dieser Mittel ermöglicht die Framework-Konstruktion.

Eine besondere Form objektorientierter Frameworks sind sogenannte Application-Frameworks (Anwendungs-Rahmenwerke). Diese „großen“ Rahmenwerke stellen die komplette Infrastruktur für eine Gruppe ähnlicher Anwendungen zur Verfügung und ermöglichen damit die einfache Erstellung ganzer Produktfamilien mit einheitlichem Grundverhalten.

Durch ein Application-Framework wird eine lauffähige *generische* Anwendung implementiert, die jedoch noch keine anwendungsspezifische Funktionalität enthält. Der Benutzer eines solchen Frameworks kann sich auf eine erprobte softwaretechnische Konstruktion verlassen und daher auf die Implementation der spezifischen Aspekte der Applikation konzentrieren. Darüber hinaus erlauben Application-Frameworks durch die weitgehende Implementation allgemeiner Anwendungskomponenten auch die Sicherstellung von Richtlinien, sei es für ein bestimmtes Fenstersystem oder innerhalb einer Organisation.

[JF88] unterscheidet Frameworks nach der zur Wiederverwendung eingesetzten Technik in White-Box und Black-Box-Frameworks. Diese Bezeichnungen finden sich entsprechend auch in [Gamma92] sowie [BBE95].

Im wesentlichen basiert die Unterscheidung darauf, ob eine Klasse des Framework vom Benutzer beerbt oder mit Parameterobjekten kombiniert wird. Im ersten Fall werden Methoden der Oberklasse implementiert bzw. überschrieben, während im zweiten Fall die Spezialisierung der Framework-Klasse durch Parametrisierung mit entsprechenden Objekten erfolgt. Hier wird die Polymorphie objektorientierter Programmiersprachen genutzt, indem eine (abstrakte) Oberklasse das Protokoll aller zulässigen Parameterklassen definiert und konkrete Klassen hiervon abgeleitet werden.

Konkrete Parameterklassen können bereits im Framework vorhanden sein, so daß nur noch die geeignete Kombination verschiedener Framework-Komponenten von Benutzer konfiguriert werden muß (vgl. Taligent [Andert95]). In vielen Fällen ist es jedoch erforderlich, selbst die geeignete Parameterklasse zu schreiben. Diese eigene Klasse wird dann von der abstrakten Oberklasse abgeleitet oder spezialisiert eine im Framework vorhandene konkrete Parameterklasse.

White-Box-Frameworks bieten grundsätzlich den Vorteil, flexibler anpaßbar zu sein, da jede (virtuelle) Methode der Frameworkklassen überschrieben werden kann. Demgegenüber werden Parameterklassen für genau bestimmte Formen der Anpassung entworfen und können weitgehend auch nur hierfür eingesetzt werden. Da der Benutzer eines Black-Box-Frameworks weniger über dessen Implementation wissen muß, fällt der erforderliche Lernaufwand geringer aus. Soweit bereits fertige Parameterklassen vorliegen, können diese ohne Kenntnis der Implementation kombiniert werden.

Im folgenden werde ich die hier genannten Begriffe konkret definieren und eine Klassifikation für Application-Frameworks vorbereiten.

2.1.1 Bausteinklasse und Bausteinbibliothek

[Gamma92] definiert Bausteinklassen folgendermaßen:

> *„**Bausteinklassen** sind Klassen, die direkt ohne weitere Ableitung unabhängig von anderen Klassen wiederverwendet werden.“*

Bausteinbibliotheken können als Sammlungen definiert werden. Die einzelnen Bausteinklassen einer Bibliothek werden weitgehend unabhängig voneinander verwendet. [JF88] betont, daß diese Klassen nicht von einem bestimmten Anwendungsgebiet abhängig sind und vielseitig wiederverwendet werden können.

> *„Each component in such a library can serve as a discrete, stand-alone, context independent part of a solution to a large range of different problems. Such components are largely application independent.“*

Als Merkmal von Bausteinklassen gilt deren Unabhängigkeit von anderen Klassen innerhalb der Bausteinbibliothek. Dies kann jedoch nur begrenzt gefordert werden, da vielfach einzelne Klassen miteinander verknüpft sind. Behälterklassen sind bei vielen Autoren (u.a. [Gamma92] und [JF88]) Beispiele für Bausteinbibliotheken. Der Benutzer verwendet jeweils einzelne Behälter[5] in seiner Anwendung, um Objekte seiner eigenen Klassen zu verwalten. Dennoch sind gerade in hochentwickelten Behälterbibliotheken (z.B. ET++, siehe [Gamma92] oder ConLib [Traub95]) meist Iterator oder Cursorkonzepte integriert, die Beziehungen zwischen einer Cursor- und einer Behälterklasse herstellen.

Während in ET++ zu jeder Behälterklasse eine spezielle Iteratorklasse besteht, die auch passend zueinander verwendet werden müssen, kann der Cursor in der ConLib auf beliebigen Behältern arbeiten. In beiden Fällen muß jedoch die Forderung unabhängiger Bausteinklassen abgeschwächt werden, da jeweils Cursor (bzw. Iteratorklasse) und Behälter direkt zusammenhängen; ein Cursor - Objekt wird immer gemeinsam mit einem Behälterobjekt benutzt. Weiterhin können in Behälter vielfach keine beliebigen Objekte eingefügt werden. ET++ verfügt über eine Oberklasse Objekt (monolithische Klassenhierarchie - single rooted, siehe auch 2.1.4), von der alle anderen Klassen abgeleitet werden müssen. Dagegen können Behälter der ConLib zwar mittels Verwendung des Template-Mechanismus beliebige Objekte verwalten, einige Behälter setzen jedoch ein (minimales) Protokoll der eingefügten Objekte voraus (insbesondere Vergleichsoperationen, siehe dazu auch die Erweiterung des C++ Standards durch die STL [MS96], die hier noch weitgehendere Bedingungen stellt).

---

[5] Die Verwendung des Begriffes Behälter bezieht sich auf [Traub95]. Einige von den Autoren aufgeführte Bibliotheken bleiben allerdings auf der Ebene von Datenstrukturen.

Komplette Unabhängigkeit der Bausteinklassen kann somit nicht gefordert werden. Innerhalb von Bausteinbibliotheken bestehen Beziehungen zwischen einigen - *wenigen* - Klassen, die der Benutzer kennen und beachten muß. Diese Abhängigkeiten sind aber als Ausnahmen einzustufen, während überwiegend unabhängige Bausteinklassen in der Bibliothek enthalten sind. Demgegenüber sind Frameworks durch üblicherweise größere Klassenteams geprägt, die Klassen sind nicht einzeln wiederverwendbar (siehe 2.1.2).

Wichtigstes Unterscheidungsmerkmal zwischen Bausteinbibliotheken und Frameworks ist die direkte Benutzung der Bausteinklassen ohne weitere Spezialisierung. Johnson betont die Unabhängigkeit von Bausteinbibliotheken bzgl. konkreter Anwendungen. Bausteinklassen bieten Lösungen für sehr allgemeine Problemstellungen und können daher vielfach unverändert wiederverwendet werden, während Frameworks - besonders White-Box-Framework - die Ableitung eigener Klassen unbedingt erfordern.

Es ist im allgemeinen möglich, Bausteinbibliotheken durch eigene Klassen zu erweitern bzw. von den vorhandenen Bausteinklassen speziellere für ein konkretes Projekt abzuleiten. Gründe können fehlende Funktionalität oder auch ineffiziente Implementation im Zusammenhang mit besonderen Anforderungen sein. Ausgereiften Bibliotheken erfordern selten derartige, im Gegensatz zur reinen Benutzung aufwendigen Erweiterungen.

Bei der Implementation von Bausteinklassen können andere Klassen verwendet werden, ohne daß dies dem Benutzer dieser Klasse bewußt sein muß. Die ConLib macht intern massiv von derartigen Konstruktionen Gebrauch. [Traub95] weist in der Dokumentation darauf hin, daß nur drei tatsächliche Datenstrukturen implementiert und alle Behälter auf diese abgebildet wurden. Dem Benutzer der Bibliothek ist dies nicht bewußt, eine Erweiterung erfordert dagegen Kenntnisse der internen Struktur. Relevant für die Klassifizierung der ConLib als Bausteinbibliothek ist die Verwendung der Klassen, nicht deren Implementation.

> Kennzeichnend für ***Bausteinklassen*** ist deren direkte Benutzung. Eigene Klassen werden in der Regel nicht abgeleitet. Die Bausteinklasse erbringt ihre Leistung dadurch, daß der Benutzer die Funktionen an ihrer Schnittstelle aufruft. Sie wird nicht selbst aktiv und verwendet ihrerseits keine Methoden, die der Benutzer implementiert. Zusammenhänge mit anderen Klassen der Bibliothek können in Einzelfällen bestehen, in der Regel werden in einer ***Bausteinbibliothek*** jedoch unabhängig voneinander benutzbare Klassen zusammengefaßt.

2.1.2 Definition Framework

> *„What exactly is an object-oriented framework? It is an object-oriented class hierarchy plus a built-in model of interaction which defines how the objects derived from the class hierarchy interact with one another.“*

[Lewis95] betont bei seiner Framework-Definition die Beziehungen zwischen Frameworkklassen. Ähnlich auch Bischofberger et. al. in [BBE95] :

> *„Ein Framework besteht aus einer Menge von Objekten, die eine generische Lösung für eine Reihe verwandter Probleme implementieren. Es legt die Rollen der einzelnen Objekte und ihr Zusammenspiel fest. Damit definiert das Framework auch jene Stellen, an denen die Funktionalität erweitert und angepaßt werden kann.“*

Zusätzlich stellt diese Definition klar, daß Frameworks jeweils für verwandte Problemstellungen entworfen werden. Gegenüber Bausteinbibliotheken (z.B. Behälterklassen) ist dies ein charakteristischer Unterschied, da diese überwiegend von spezifischen Anwendungen unabhängig entworfen und wiederverwendet werden. Dazu schreibt [JF88]: „*A framework, on the other hand, is an abstract design for a particular kind of application*“. [Pree94] sieht Bausteinklassen und Frameworks als verschiedene Ebenen der Wiederverwendung („*two principal levels of software reusability: (1) reuse of single software components, (2) reuse of software architectures*“), wobei Frameworks die technischen Möglichkeiten objektorientierter Sprachen zur Wiederverwendung optimal ausnutzen:

> *„The concepts inheritance and dynamic binding are sufficient to construct frameworks, that is, reusable semifinished architectures for various application domains.“ - „Frameworks represent the highest level of reusability known today, made possible by object-oriented concepts.“* [Pree94]

Frameworks dienen der Wiederverwendung „im Großen“, der Schwerpunkt liegt nicht auf der Wiederverwendung von Code sondern von Entwurf („design reuse“). [Pree94] betont die Kombination objektorientierter Techniken zur Framework-Konstruktion, und zwar dynamische Bindung zusammen mit Vererbung und abstrakten Oberklassen („... *an appropriate combination of the basic object-oriented concepts is necessary* ...“). Auch Wirfs-Brock und Johnson definieren Frameworks als Wiederverwendung von Design mittels abstrakter Klassen:

> *„An object-oriented abstract design, also called a framework, consists of an abstract class for each major component.“* [JF88]

> *„A framework is a collection of abstract and concrete classes and the interface between them, and is the design for a subsystem.“* [WJ90]

Wiederverwendet werden nicht einzelne Klassen, sondern immer das ganze Team ([BBE95]: „*Ein Framework definiert ein Klassenteam als Wiederverwendungseinheit.*“). [JF88] motiviert die Verwendung von Frameworks gegenüber Bausteinbibliotheken, die meist einfacher zu benutzen und auch häufiger wiederverwendbar sind:

> *„Application independent components can be reused rather easily, but reusing the edifice that ties the components together is usually possible only by copying and editing it. Unlike skeleton programs, which is the conventional approach to reusing this kind of code, frameworks make it easy to ensure the consistency of all components under changing requirements.“*

Zur Abgrenzung gegenüber Bausteinbibliotheken ist die Umkehrung des Kontrollflusses die wichtigste Eigenschaft von Frameworks (vgl. Abbildung 2-1). Nur so ist es möglich, die Wiederverwendung von Design zu erreichen und eine Struktur für die Anwendung (Application-Frameworks) bzw. Teile der Anwendung vorzugeben. Dazu sagt [JF88]:

> *„One important characteristic of a framework is that the methods defined by the user to tailor the framework will often be called from within the framework itself [...]. This inversion of control gives the frameworks the power to serve as extensible skeletons.“*

Auch [Gamma92] spricht von der Wiederverwendung „*vollständiger Designstrukturen*“. Im Gegensatz zu Programmgerüsten, die vorher die einzige Möglichkeit darstellten, die Struktur einer Anwendung wiederzuverwenden, können Frameworks ohne Änderung des Quellcodes wiederverwendet werden. Damit ist es auch zu einem späteren Zeitpunkt möglich, Korrekturen oder Erweiterungen des Frameworks vorzunehmen, von denen alle darauf basierenden Anwendungen direkt profitieren.

Zusammenfassend können Frameworks folgendermaßen charakterisiert und gegenüber Bausteinbibliotheken abgegrenzt werden:

> ***Frameworks*** sind Klassenhierarchien, die abstrakte Lösungen für *verwandte Problemstellungen* realisieren. Wiederverwendet werden nicht einzelne Klassen sondern das *Klassenteam* als Ganzes („design reuse“). Dabei definieren abstrakte Klassen Schnittstellen, die von anderen, konkreten Klassen des Teams verwendet werden. Frameworks werden *durch Ableitung* eigener Klassen von abstrakten oder auch konkreten Framework-Komponenten spezialisiert. Das Framework gibt vor, welche Methoden zu überschreiben sind. Der *Kontrollfluß* wird im Framework festgelegt, somit werden auch die Methoden der vom Benutzer entwickelten Klassen aus dem Framework aufgerufen (Hollywood-Prinzip).

2.1.3 Application-Framework

Bischofberger et. al. übersetzen den Begriff Application-Framework mit *Anwendungsumgebung* (siehe [BBE95]). Aus einzelnen Frameworks „*für graphische Benutzungsschnittstellen und für ereignisorientierte, dokumentbasierte Editoren*“ (vgl. Kapitel 3) haben sich Anwendungsumgebungen entwickelt, die eine Systemumgebung für die Anwendungsentwicklung definieren (z.B. MFC, MacApp und ET++). Die Anwendungen selbst werden dadurch vom Fenstersystem und teilweise auch vom Betriebssystem unabhängig.

Frameworks und besonders Application-Frameworks werden für eine Gruppe gleichartiger Anwendungen entwickelt. [Lewis95] schreibt:

> *„A Framework is more than a class hierarchy. It is a miniature application complete with dynamic as well as static structure. It is a generic application we can reuse as the basis of many other applications. And, before I forget it, frameworks are specialized for a narrow range of applications, because each model of interaction is domain-specific, e.g., designed to solve a narrow set of problems. A framework is the product of many iterations in design. It is not something that you invent in a big bang, and then go about reusing for years. Frameworks evolve over long periods of time …“*

Ausgangspunkt für die Entwicklung eines Application-Frameworks ist meist eine relative kleine Anzahl konkreter Anwendungen, deren Gemeinsamkeiten abstrahiert werden. Für die evolutionäre Entwicklung ist die tatsächliche Verwendung des Frameworks entscheidend, nur so kann die Tragfähigkeit des Entwurfes überprüft werden. ([Rosenstein95]: „*You cannot really claim to have a reusable framework until it has been used in a variety of cases.*“)

Das Ergebnis ist eine generische Applikation: lauffähig, aber ohne konkrete Funktionalität. Anwendungen werden entwickelt, indem spezifische Komponenten ergänzt werden. Der Kontrollfluß und die Infrastruktur ist bereits vorhanden, der Entwickler muß den Aufruf seiner spezifischen Operationen nicht selbst realisieren. Dazu Weinand und Gamma in [WG95]:

> *„The application classes define the application’s natural components and how they interact. This interaction is the main difference between an application framework and a collection of abstract but not strongly related classes. The former allows the factoring of much of the control flow into the class library, while the latter requires that the developer know exactly when to call which method.“*

Abbildung 2-1 demonstriert den Unterschied zwischen der Verwendung von Bausteinbibliotheken bei der Anwendungsentwicklung, und demgegenüber einem Application-Framework als Basis der eigenen Applikation. Die Umkehrung des Kontrollflusses ist deutlich zu erkennen, der Anteil spezifischer Komponenten verringert sich. (Genauere Erläuterungen zur Verwendung von Frameworks folgen in 2.1.4).

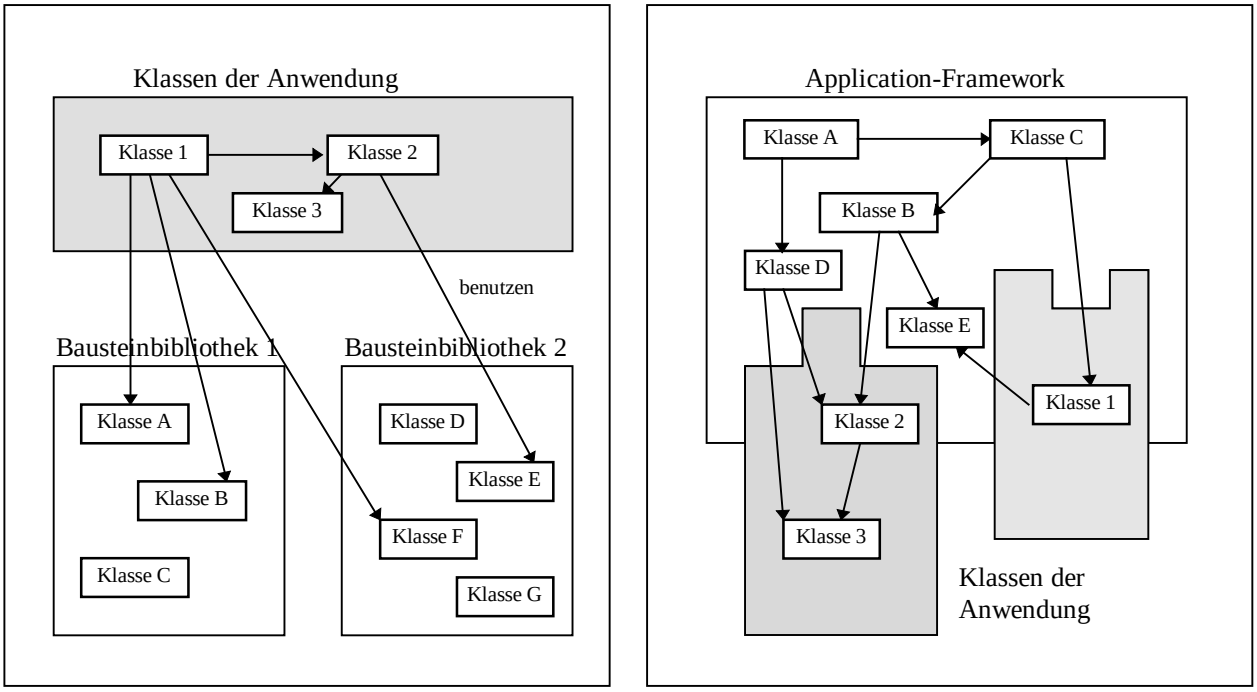


Abbildung 2 - 1: a) Applikation mit Bausteinbibliothek; b) Verwendung eines Application-Frameworks

Die Unterscheidung „kleiner“ Frameworks (Mikroarchitekturen) gegenüber Application-Frameworks läßt sich nicht scharf definieren, da sich vielfach Application-Frameworks aus kleineren Mikroarchitekturen entwickeln. [Pree94]:

> *„...it is often hard to decide whether a framework migrates to the category of application frameworks or not“*

Als Anhaltspunkt gibt Pree folgende Definition:

> *„In order to build a terminological basis, we distinguish between two categories of framework. Framework that constitute a generic application for a domain area are called application frameworks. Framework are those that represent a micro-architecture consisting of only a few components.“*

Spezielle Application-Frameworks können große Teile der Anwendungen bereits realisieren. Demgegenüber beschränken einfachere Application-Frameworks sich eventuell darauf, die Anwendung in einem Fenster mit erweiterbarem Standardmenü anzuzeigen. Implementiert sind dabei bereits Aktionen wie Datei öffnen und schließen oder Mechanismen für Drag&Drop. Komponenten, die bereits im Framework implementiert bzw. mittels abstrakter Oberklassen definiert werden, sind damit für alle Anwendungen einheitlich.

> *„Wieviel ein Framework von einer Anwendung eines bestimmten Gebietes abdeckt, hängt davon ab, wie weit man die benötigte Funktionalität standardisieren und implementieren kann.“* [BBE95]

Je enger das Anwendungsgebiet abgegrenzt ist, desto mehr Standardisierung ist möglich. So betont z.B. [Pree94] die Nutzung der Frameworktechnik in Bereichen, die über die Kapselung von Fenstersystemen hinausgehen: „*Application frameworks are not limited to the construction of direct-manipulation, graphic user interfaces*“. Beispiele sind Betrieb-

systeme, VLSI Routing und anderes. Der Schwerpunkt liegt allerdings heute noch überwiegend auf dem GUI - Sektor, da diese Frameworks weitgehend domänenunabhängig und damit für fast alle Programmierer nützlich sind.

Bischofberger betrachtet die hohen Anforderungen an technische und gleichzeitig fachliche Qualifikation im jeweiligen Anwendungsfeld als hauptsächliche Schwierigkeit, die bisher die Entstehung branchenspezifischer oder unternehmensbezogener Application-Frameworks weitgehend verhindert hat. Dazu [BBE95]:

> *„Seit längerer Zeit diskutiert man die strategischen Vorteile, spezialisierter Anwendungsframeworks.*
> *Leider erfordert deren Entwicklung neben großem softwaretechnischen Wissen auch noch ein sehr tiefes Verständnis für das Anwendungsgebiet.“*

Vorteile spezieller Frameworks sind unbestritten. Soweit ein Unternehmen ein eigenes, auf das konkrete Anwendungsgebiet zugeschnittenes Application-Framework einsetzt, erleichtert dies die Erstellung einzelner, neuer Anwendungen auf Basis des Frameworks, da die benötigte Grundfunktionalität bereits überwiegend vorhanden ist. Es werden Mechanismen vorgesehen, neue Komponenten einfach zu implementieren und zu integrieren. Die weitere Anwendungsentwicklung kann zu Framework-Erweiterungen führen, und den Ausgangspunkt für Folgeprojekte weiter verbessern.

Aus Sicht des Management läßt sich die Entwicklung von Frameworks dagegen nur schwer bewerten. Erst die Wiederverwendung in Folgeprojekten bringt wirtschaftlichen Nutzen, die Entwicklungskosten müssen auf alle Projekte verteilt werden. Dabei wird die Framework-Konstruktion selbst nicht abgeschlossen, da sich aus neuen Projekten häufig Änderungs- und Erweiterungsbedarf ergibt. [BBE95] gibt zu bedenken, daß der Einsatz der Frameworktechnik die Wartung entwickelter Anwendungen erheblich erleichtert, die Zuverlässigkeit erhöht und eine einheitliche Bedienung sicherstellt. Diesen Effekt muß das Management bei der Kostenrechnung geeignet mit berücksichtigen.

Während Styleguides gewisse Konventionen für die Gestaltung der Oberfläche und deren Handhabung vorschreiben, können Application-Frameworks wesentliche Teile bereits implementieren. Sogar eine spätere Änderung kann unter Umständen möglich sein, ohne daß Anwendungen betroffen sind. Sobald diese mit der neuen Version des Frameworks übersetzt werden, berücksichtigen sie neue Konventionen ohne weiteren Aufwand.

Zur nachträglichen Erweiterung bestehender Anwendungen durch Änderung des zugrunde liegenden Frameworks sagen Weinand und Gamma [WG94]:

> *„An object-oriented application framework transfers a lot of control flow from client code into the class library. This characteristic allows the addition of new functionality without any modification in existing applications and with little programming effort.“*

Die erhöhte Zuverlässigkeit resultiert vor allem aus der häufigen Wiederverwendung der Framework-Komponenten und deren dadurch sehr vollständigen Tests. Außerdem können später erkannte Fehler einmal im Framework behoben werden. Alle darauf aufbauenden Applikationen profitieren direkt. Der Testaufwand bleibt auf das Framework begrenzt und muß nicht in jeder Anwendung wiederholt werden. Unter der Annahme einer Verwendung des Framework in sehr vielen Anwendungen ist dies ein großer Vorteil, auch wenn an den Test innerhalb des Frameworks hohe Anforderungen zu stellen sind.

> *„The N th ensemble for a framework benefits from real-world testing by the N-1 previous ensembles. Because the framework can live in a shared library separate from the ensemble, when framework bugs are fixed, all ensembles can benefit.“* [Andert95]

Bei einem Wechsel der Version des verwendeten Frameworks wird immer auch ein Test der darauf aufbauenden Anwendungen erforderlich sein. Soweit die Änderungen hinreichend dokumentiert sind und das Framework keine neuen Fehler enthält, sollte dieser Aufwand gering sein. Probleme ergeben sich z.B. durch „workarounds“, mit denen Unzulänglichkeiten der alten Version ausgeglichen wurden. Diese müssen nach Behebung des Fehlers aus der Anwendung entfernt werden. Wartung der Applikationen selbst, sei es Fehlerkorrektur oder Erweiterung der Funktionalität, wird einerseits durch relativ geringeren Umfang infolge der Verlagerung großer Teile der Funktionalität in das Framework, andererseits durch einheitliche softwaretechnische Struktur erleichtert.

Ein ***Application-Framework*** realisiert die Infrastruktur einer vollständigen Anwendung und legt deren Kontrollfluß fest. Konkrete Anwendungen werden erstellt, indem deren spezifische Komponenten in das Framework integriert werden. Kleinere Frameworks, zur Unterscheidung Mikro-Framework genannt, realisieren jeweils nur einzelne Ausschnitte.

Soweit der Umfang der Standardisierung von Bedeutung ist, werden im folgenden *Anwendungsumgebungen* gegenüber *Domänen-Frameworks* abgegrenzt. Dabei legen *Anwendungsumgebungen* nur eine sehr allgemeine Infrastruktur fest und lassen sich in einem relativ breiten Bereich wiederverwenden. Sie ermöglichen z.B. die Entkopplung darauf aufbauender Anwendungen vom verwendeten Fenster- oder Betriebssystem und standardisieren einige Menüeinträge (Open/Close).
Je spezifischer Application-Frameworks eingesetzt werden, desto mehr standardisierte Funktionalität kann integriert werden. Dies setzt entsprechendes Wissen über den Anwendungskontext voraus. Als Basis für spezialisierte *Domänen-Frameworks* können allgemeine Anwendungsumgebungen verwendet werden, um deren Infrastruktur zu nutzen.

2.1.4 Arten der Wiederverwendung

White-Box und Black-Box

Seit [JF88] findet sich in der Literatur die Einteilung in White-Box und Black-Box-Frameworks.

> *„A framework's application specific behavior is usually defined by adding methods to subclasses of one or more of its classes. [...] We call these white-box frameworks because their implementation must be understood to use them."*

> *„Another way to customize a framework is to supply it with a set of components that provide the application specific behavior. Each of these components will be required to understand a particular protocol. All or most of the components might be provided by a component library. The interface between components can be defined by protocol, so the user needs to understand only the external interface of the components. Thus, this kind of a framework is called a black-box framework."*

Entsprechend definiert auch [BBE95] die White-Box - Wiederverwendung. Nachteilig ist demnach das notwendige Verständnis der Framework-Implementation. Der Benutzer benötigt Kenntnisse der einzelnen Komponenten und deren Verbindungen.

> *„Erweitert man ein Framework durch das Überschreiben von Methoden, spricht man von White-Box - Wiederverwendung. Das Framework legt dabei wichtige Teile und Beziehungen fest [...]. Diese müssen durch anwendungsspezifische Teile ergänzt werden [...].*
> *Will man ein Framework so erweitern, muß man in dieses hineinschauen und das Zusammenspiel der verschiedenen Teile verstanden haben."*

Dagegen können Klassen in Black-Box- Frameworks direkt benutzt werden. Spezialisierung dieser Frameworks erfolgt nicht durch Anpassung einzelner Methoden in abgeleiteten Klassen, sondern durch Parametrierung der erzeugten Objekte. Bestandteil des Frameworks sind direkt instanziierbare Klassen, die ohne weitere Ableitung benutzt werden. Vorteil dieser Methode ist (laut [BBE95]) die leichtere Wiederverwendung:

> *„Ein Black-Box Framework ist dadurch gekennzeichnet, daß es verschiedene direkt benutzbare Klassen zur Instantiierung und Parametrierung zur Verfügung stellt. Es kann verwendet werden, ohne daß man versteht, wie die Objekte intern zusammenspielen."*

Bischofberger et. al. [BBE95] sehen als wesentlichen Vorteil der White-Box - Wiederverwendung die hohe Flexibilität, mit dem Problem des dazu erforderlichen Verständnisses der Zusammenhänge.

> *„White-Box - Wiederverwendung ist die mächtigste Art der Wiederverwendung, da man in Unterklassen sehr flexibel Anpassungen und Erweiterungen vornehmen kann. White-Box - Wiederverwendung heißt aber programmieren, was relativ aufwendig ist und ein gutes Verständnis für die im Framework implementierten Mechanismen erfordert.“*

Black-Box-Wiederverwendung kann nur eingesetzt werden, wenn entsprechende Anpassungen vom Entwickler des Frameworks bereits vorgesehen wurden.

> *„Die Black-Box - Wiederverwendung erlaubt nur beschränkte Anpassungen, dafür kann man sie mit geringem Aufwand durchführen.“*

Ähnlich stellt auch [JF88] dem geringen Lernaufwand einen Mangel an Flexibilität auf Seiten des Black-Box - Frameworks gegenüber:

> *„Black-Box frameworks [...] are easier to learn to use than white-box frameworks, but are less flexible.“*

Da jedoch Black-Box-Frameworks oft auf Grundlage von White-Box-Frameworks implementiert werden, kann der Benutzer in vielen Fällen immer noch auf diese Basis zurückgreifen. Soweit die gewünschte Spezialisierung mit Hilfe des Black-Box-Frameworks möglich ist, also von dessen Entwicklern vorbereitet wurde, sind Kenntnisse des verwendeten White-Box-Frameworks nicht erforderlich. Falls aber Komponenten des Frameworks geändert werden müssen, die im Black-Box-Ansatz standardisiert wurden, so kann der Entwickler hierzu direkt in das White-Box - Framework eingreifen - benötigt dazu dann aber weitergehende Kenntnisse interner Zusammenhänge [BBE95].:

> *„Black-Box-Frameworks erfordern keine Kenntnisse über das zur Implementierung benutzte White-Box-Framework. Normalerweise kann man aber darauf zurückgreifen, wenn man Funktionalität benötigt, die sich nicht durch Black-Box - Wiederverwendung realisieren läßt.“*

Wie bereits in den vorangehenden Abschnitten erwähnt, werden Frameworks nicht einmalig entwickelt, sondern müssen eingesetzt und praktisch erprobt werden. Die Erfahrungen der Anwender müssen gesammelt und ausgewertet werden. Framework-Entwicklung ist im großen Umfang auf ein derart zyklisches Vorgehen angewiesen („*You never get it right the first time*“ [Reenskaug95]). Erst durch die Rückkopplung mit den Benutzern des Frameworks können dessen Schwächen erkannt werden.

> *„In fact, as the design of a system becomes better understood, black-box relationships should replace white-box ones.“*

[JF88] fordert, diese Ergebnisse zu nutzen, und dementsprechend das Framework zu vereinfachen. [BBE95] geht davon aus, daß die meisten Frameworks eine Mischform sein werden

> *„Wenn wir das offene White-Box Framework einfacher benutzbar machen und die internen Mechanismen verbergen, bewegen wir uns auf ein Black-Box Framework zu.“*

sich aber in Richtung eines Black-Box-Frameworks entwickeln sollten, um die Benutzung vereinfachen zu können:

> *„Der Übergang von White-Box zu Black-Box - Wiederverwendung ist fließend. Einige Frameworks ermöglichen wesentliche Erweiterungen durch Parameterobjekte. Andere erlauben nur aus einer vorgegebenen Menge von Klassen auszuwählen.“*

Grundlage dieser Betrachtung sind auf der einen Seite sogenannte White-Box - Frameworks, die vom Benutzer vollständig verstanden werden müssen. Die Benennung soll dabei hervorheben, daß ein Blick in das Innere des Frameworks, in dessen Implementation, der Schlüssel zur Verwendung ist.

Auf der anderen Seite stehen Frameworks, die gewisse Leistungen erbringen. Es handelt sich um Klassenteams, deren interne Beziehungen der Anwender nicht weiter beachten muß; vielmehr benutzt er das Black-Box-Framework ohne Kenntnis dessen Implementation.

Ziel der Entwicklung ist es, ein leicht benutzbares Framework herzustellen. Dabei ist zu berücksichtigen, daß sich der einmalig investierte Aufwand vielfach auszahlt, da erfolgreiche Frameworks vielfach verwendet werden. Der Benutzer muß sich grundsätzlich in die Struktur eines Frameworks einarbeiten. Er muß erkennen, welche Komponenten für ihn von Bedeutung sind und wie er diese einsetzen kann. Black-Box-Frameworks beschränken diesen Aufwand auf die Frage der Benutzung vorhandener Komponenten, während White-Box - Wiederverwendung zusätzlich verlangt, die Implementation der benutzten Komponenten zu verstehen.

Wieweit dem Benutzer infolge von Black-Box - Wiederverwendung eine genauere Betrachtung der Implementation erspart werden kann, hängt vor allem davon ab, ob er mit den angebotenen Komponenten die gewünschte Funktionalität erreichen kann. Sind wesentliche Grundlagen nicht vorgesehen, wird er in die Implementation eingreifen müssen, um das Framework entsprechend zu erweitern. In diesem Zusammenhang ist es von besonderem Wert, wenn als Grundlage der standardisierten Komponenten des Frameworks zur Black-Box - Wiederverwendung ein flexibles Framework dient, daß eine Anpassung im Sinne des White-Box - Ansatzes erlaubt.

Black-Box - Wiederverwendung muß vom Entwickler vorbereitet werden und dient somit ausschließlich zur Anpassung des Frameworks. Ein Verständnis der Implementation wird nicht erwartet. Dagegen dient der ***White-Box*** - Ansatz zur flexiblen, nicht vorhergesehen Erweiterung der Framework-Funktionalität innerhalb des vorgegebenen Anwendungsbereiches und setzt eine weitgehende Kenntnis der Framework - Architektur voraus.

White-Box-Frameworks sind insbesondere frühe Entwürfe, die noch wenig Annahmen über die konkrete Verwendung machen, und damit weitgehend mittels aufwendiger White-Box - Wiederverwendung benutzt werden. Demgegenüber sehen ***Black-Box-Frameworks*** bereits umfangreiche Mechanismen zur einfachen Benutzung vor. Eine scharfe Trennung zwischen White-Box und Black-Box-Frameworks ist nicht möglich. Auch Black-Box-Frameworks können als White-Box-Frameworks betrachtet und entsprechend unabhängig von den vorgesehenen Mechanismen erweitert werden, wenn sie für die konkrete Anwendung nicht ausreichen. In der Regel ist dies jedoch nicht erforderlich**.**

Implementationstechnik Parameterklassen

Obige Darstellung und Definition stellt ausschließlich auf die Form der Wiederverwendung ab, ohne die hierzu verwendete Technik zu diskutieren. Die von den Autoren vorgenommene Verknüpfung der Konzepte Black-Box und White-Box mit den dazu verwendeten objektorientierten Implementationstechniken[6] ist problematisch. Die Gleichsetzung zwischen White-Box-Wiederverwendung und Vererbung widerspricht dem grundsätzlichen Framework-Konzept: Jedes Framework soll dabei helfen, Implementationsdetails durch Kapselung vor dem Benutzer der Klassen zu verbergen. Und jedes Framework muß dem Anwender vorgeben, an welchen Stellen er Ergänzungen vornehmen kann, welche Spezialisierung zwingend erforderlich sind und welche Teile des Frameworks nicht verändert werden dürfen (bzw. nur vom Framework-Entwickler).

Vielfach werden diese Hinweise nur in der Dokumentation zu finden sein. Eine deutliche Verbesserung ist z.B. durch Namenskonventionen für die Methoden im Framework möglich, objektorientierte Programmiersprachen beinhalten meist nur sehr begrenzte Hilfsmittel. Die Programmiersprache C++ stellt als Sprachkonzepte einerseits den privaten Teil der Schnittstelle und andererseits die Mechanismen aufgeschobener (pure virtual) sowie nicht virtueller Methoden zur Verfügung. Für Frameworkklassen gilt (wie für jede Klasse!), daß die „protected“ - Schnittstelle für erbende Klassen ebenso sorgfältig zu entwerfen ist, wie das „public“ - Protokoll für Benutzung.

---

[6] Abbildung 2-2 stellt diese Techniken einander gegenüber, Erläuterung weiter unten in diesem Abschnitt

Methoden, die der Benutzer grundsätzlich überschreiben muß, können in den abstrakten Klassen des Frameworks *„pure virtual“* deklariert werden. In diesem Fall prüft bereits der Linker, ob abgeleitete Klassen diese Bedingung erfüllen. Andererseits können nicht virtuell deklarierte Operationen der Framework-Klasse nur dann überladen werden, wenn die Klasse nicht polymorph verwendet wird. Das ist innerhalb von Frameworks selten der Fall, derartig deklarierte Methoden dürfen also bei Ableitung eigener Klassen nicht verändert werden.

Die allgemeine Gleichstellung zwischen White-Box - Wiederverwendung und beerben von Frameworkklassen kann nicht aufrecht erhalten werden. Ohne näheres Wissen über die Implementation eines White-Box - Frameworks kann eine der Framework-Klassen durch Überschreiben lediglich der dafür vorgesehenen Methoden spezialisiert werden. Zudem funktionieren auch Black-Box-Framework grundsätzlich nach demselben Prinzip. Hier werden eigene Parameterklassen geschrieben, die (in getypten Umgebungen) auch von abstrakten oder konkreten Frameworkklassen abgeleitet werden müssen. Es handelt sich lediglich um meist kleinere Klassen, die weniger (eventuell verwirrende) Methoden enthalten. Dennoch muß der Benutzer das Protokoll der Oberklasse implementieren, also verstehen, wie seine Klasse mit dem Framework interagiert.

Wesentliche Vorteile können durch Parameterklassen erreicht werden, die jeweils ein klar erkennbares ***Konzept*** definieren. Wird eine „große“ Framework-Klasse beerbt, so muß der Benutzer nicht nur diverse Methoden kennenlernen - sondern auch noch verstehen, welche jeweils zusammenhängen. Parameterklassen können besser als Kommentare und die externe Dokumentation diese Verbindungen verdeutlichen, wenn separate Parameterklassen für jede vorgesehene Variante der Framework - Anpassung zur Verfügung stehen.

Defaultimplementationen erleichtern den Einstieg in die Programmierung eines Frameworks. Parameterklassen ermöglichen eine Erweiterung dieses Ansatzes, indem für einen Mechanismus mehrere konkrete Parameterklassen fertig zur Verfügung stehen, der Benutzer also verschiedene Defaultimplementationen zur Auswahl hat. Spezielle Frameworks können dann tatsächlich benutzt werden, ohne zu programmieren: einfach durch Kombination geeigneter Komponenten. Dabei ist einerseits denkbar, daß separate Bibliotheken von Parameterklassen durch Drittanbieter bereitgestellt werden, andererseits ergibt sich die Grundlage für visuelle Programmierung [Lewis95]; die Zusammenstellung der Komponenten kann durch ein Werkzeug geschehen. Noch eine Variante sind Frameworks, die sich zur Laufzeit selbst konfigurieren. [Andert95] nennt diese Eigenschaft des CommonPoint - I/O - Frameworks von Taligent, soweit es sich bei der verwendeten Hardware um Plug&Play-Komponenten handelt.

Vorläufig halte ich es nicht für vorstellbar, ***komplette Applikationen*** durch reine Konfiguration vorhandener Framework-Komponenten zu erstellen. Eine Ausnahme bilden sehr begrenzte Anwendungsbereiche, zum Beispiel Datenbankschnittstellen. Voraussetzung sind in jedem Fall sehr spezialisierte Domänen-Frameworks die auch nur in einem entsprechend engem Rahmen verwendbar sind. Weitergehender Einsatz der Framework - Technik verlangt immer die Implementation eigener Komponenten.

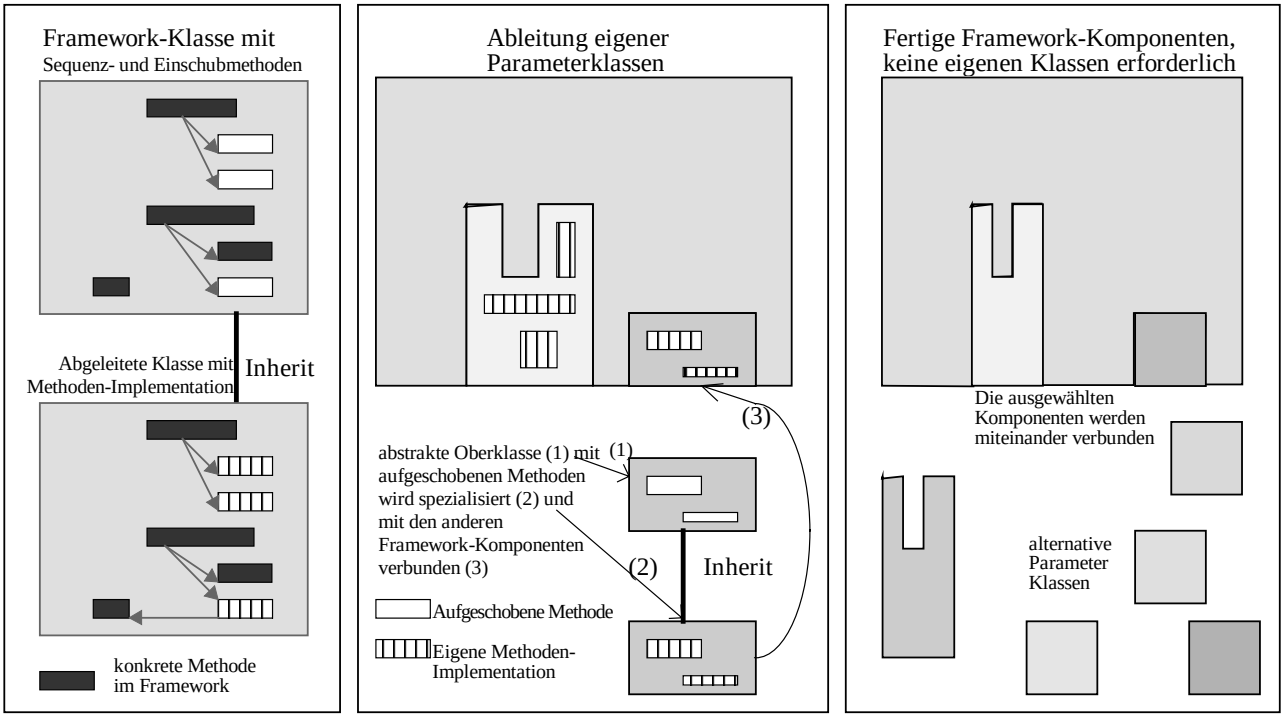


Abbildung 2-2: Anpassung von Frameworks; a) Ableitung von Framework-Klasse: Methoden überschreiben; b) Konkrete Framework-Klasse und eigene Parameterklassen; c) Framework mit konkreten Parameterklassen.

Wiederverwendung durch *Vererbung* führt nicht zwingend zu *White-Box-Frameworks*. Vielmehr sollen Frameworks unabhängig von der zur Spezialisierung verwendeten Technik grundsätzlich die internen Zusammenhänge vor dem Benutzer *verbergen*. Der Benutzer kann zwar weitergehende Änderungen vornehmen, wenn er direkt von einer Framework-Klasse erbt, die zur Implementation oder Redefinition vorgesehenen Methoden sind aber *gekennzeichnet* und setzen kein weitergehendes Verständnis interner Zusammenhänge voraus.

Parameterklassen sind ein wirksames Mittel, die *Black-Box - Technik* zu fördern. Zwar muß auch von diesen Klassen im allgemeinen abgeleitet werden, die mögliche *Trennung* verschiedener Aspekte der erforderlichen Anpassung erleichtert jedoch das Verständnis. Parameterobjekte werden vom Benutzer erzeugt und konkreten Frameworkklassen übergeben. Der Benutzer wird Methoden dieser Klassen nicht direkt aufrufen, dies geschieht nur innerhalb des Frameworks.

Weitere Vorteile der *Objektivierung*[7] sind **konkrete Parameterklassen**, aus denen der Benutzer eine passende auswählen kann. Einerseits ermöglicht dies flexible *Defaultimplementationen* - die eigenen Klassen können auch von einer konkreten Parameterklasse abgeleitet werden. Andererseits können bereits Parameterklassen für alle wesentlichen Anwendungsfälle vorliegen. Dann wird die Benutzung des Frameworks zur einer Konfiguration statt Implementation, und kann durch Werkzeuge teilweise automatisiert werden.

[7] [Gamma92]: „*Unter Objektivierung verstehen wir, die Modellierung von Abstraktionen mit Objekten, die nicht unbedingt realen Objekten des Problembereichs entsprechen.*“

Hot Spot und Frozen Spot

Objektorientierte Technik zur Framework-Konstruktion untersucht Pree im Zusammenhang mit Design Pattern (siehe Abschnitt 2.2). Seine Definition von Frameworks stellt heraus, daß jeweils einige Komponenten fertig implementiert vorliegen, während andere noch vom Benutzer angepaßt werden müssen. [Pree94a]:

> *„Application frameworks consist of ready-to-use and semi-finished building blocks. The overall architecture (= the composition and interaction of building blocks) is predefined as well.“*

Konkrete Klassen stellen die Teile der zu entwickelnden Anwendungen bereit, die durch das Framework bereits implementiert und damit standardisiert sind. Diese Klassen verwenden in ihrer Implementation Methoden anderer Klassen. Soweit es sich um abstrakte Klassen handelt, müssen diese Komponenten vom Entwickler der Anwendung ergänzt werden, indem er spezialisierte Klassen von den abstrakten Oberklassen ableitet. [Pree94]: „*... concrete classes ... can be implemented based on the contracts offered by the abstract classes.*“ Die Schnittstelle, oder genauer das zu erfüllende Protokoll ist dadurch vorgegeben, ergänzt wird die konkrete Funktionalität der Anwendung.

Das Nebeneinander festgelegter und variabler Aspekte hat zentrale Bedeutung beim Entwurf von Frameworks. Dazu [Pree94]:

> *„In general, an application framework standardizes applications for a specific domain. Usually, various aspects of an application framework cannot be anticipated.“*
> *„The core characteristic of any framework is that various aspects have to be kept flexible while others are standardized.“*

Auf Ebene der Methoden können Sequenz- und Einschubmethoden unterschieden werden [Gamma92]. Die Anpassung einer komplexen Sequenzmethoden erfolgt über die von dieser benutzten Einschubmethoden. [Pree94] verwendet die Begriffe „*hook method*“ (vgl. Einschubmethode) und „template method“ (vgl. Sequenzmethode), wobei „hook methods“ abstrakt oder regulär sein können - oder selbst wieder „template methods“. Grundsätzlich kann jede Methode als Template betrachtet werden, wenn sie andere Methoden benutzt, jede benutzte Methode somit als Hook. Diese Bezeichnungen hängt von der Betrachtungsweise ab. *Reguläre* Methoden verwenden keine andere Methoden, abstrakte Methoden müssen in abgeleiteten Klassen implementiert werden.

Flexibilität wird in Frameworks erreicht durch „hook methods“. Während standardisierte Funktionalität im Template festgeschrieben wird, können variable Teile durch Anpassung des Hook geändert werden. [Pree94] führt die Begriffe „frozen spot“ und „hot spot“ ein:

> *„The standardized frozen spots of a framework are represented by template methods. Hook methods are the flexible, gray spots [hot spots] within these frozen spots.“*

Die Unterscheidung in standardisierte und flexible Methoden enthält noch keine Angaben über die Aufteilung in Klassen. Wichtig ist erst einmal, notwendigerweise anpaßbare Teile des Frameworks zu identifizieren. Ein Framework soll möglichst viele Komponenten bereits implementieren, Gemeinsamkeiten aller Applikationen einer Domäne sollen durch das Framework als Standard bereitgestellt werden, also den Anwendungsentwickler nicht weiter belasten. Andererseits hängt der Erfolg eines Frameworks davon ab, ob es sich hinreichend an konkreten Bedürfnissen orientiert und an einzelnen Punkten entsprechend flexibel ist.

> *„The difficulty of 'good' object-oriented design is to identify the hot spots of an application framework, i.e. those aspects of an application domain that have to be kept flexible."* [Pree94a]

Als nächstes muß dann der benötigte Grad an Flexibilität ermittelt werden. Hiervon hängt es ab, ob Template und Hook zu derselben Klasse gehören können, oder aufgeteilt werden. Befindet sich die hook method in einer eigenen Klasse, so kann auch noch zur Laufzeit eine Konfiguration erfolgen bzw. diese Methode ausgetauscht werden.

Während [JF88] und [BBE95] die Verwendung von Parameterklassen als Zeichen der Black-Box - Wiederverwendung ansehen, macht [Pree94] deutlich, daß diese Klassen lediglich die Trennung von Sequenz- und Einschubmethode ermöglichen und so die Flexibilität erhöhen. Wie bereits ausgeführt, beziehen sich die Begriffe White-Box und Black-Box nicht auf die verwendete objektorientierte Technik, sondern auf das vom Framework benötigte Verständnis des Benutzers. Um das Framework als Black-Box benutzen zu können, müssen „hot spots" und „frozen spots" deutlich zu unterscheiden sein. Der Benutzer muß erkennen können, wie die Anpassung der hot spots zu erfolgen hat. Parameterklassen werden immer im Zusammenhang mit Flexibilität zur Laufzeit benötigt, können davon unabhängig aber auch in anderen Fällen verwendet werden.

Bäume und Wälder

Frameworks sind objektorientierte Klassenhierarchien. Dabei können die Mitglieder eines Klassenteams unterschiedlichen Vererbungsbäumen zugehören. Abstrakte Klassen sind häufig selbst Wurzelklassen neuer Bäume. Somit gehören zu einem Framework meist mehrere Klassenbäume („Wald"), zwischen denen vielfältige Benutzungsbeziehungen bestehen.

Klassenbibliotheken und Frameworks können andererseits über eine gemeinsame Wurzel aller Klassen verfügen (single rooted, „Baum"). Mittels Polymorphie ermöglicht diese Wurzelklasse in getypten objektorientierten Sprachen die Konstruktion generischer Behälter, die Objekte beliebigen Typs aufnehmen können, da alle Klassen (hier gleichbedeutend mit Typ verwendet) von dieser gemeinsamen Oberklasse abgeleitet sind (ET++).

Eine bessere Lösung bietet Generizität (inzwischen auch in C++ verfügbar), da diese Behälter nicht beliebige Objekte aufnehmen können, sondern jeweils nur einen spezifizierten Typ. Mittels Polymorphie können auch hier Unterklassen mit eingeschlossen werden. Dazu wird in

C++ ein Template implementiert, das selbst keine fertige Klassendefinition ist, sondern als Vorlage verwendet und mit einer konkreten Klasse instanziiert wird. Der wesentliche Vorteil dieser Lösung unter Verwendung von Generizität ist der Erhalt der Typinformation, während polymorphe Behälter grundsätzlich die Verwendung von Type-Casts erfordern.

Nachteilig wirkt sich die gemeinsame Oberklasse dann aus, wenn mehrere Frameworks miteinander kombiniert werden sollen, deren Klassen nicht alle von derselben Oberklasse abgeleitet wurden. Soweit die Quelltexte eines erworbenen Frameworks nicht verfügbar sind, ist eine Anpassung nicht möglich, anderenfalls jedenfalls sehr aufwendig. Dies ist eines der Probleme, das Allan et. al. bei der Wiederverwendung objektorientierten Klassenbibliotheken und Frameworks identifizieren [AGO95].

Gemeinsame Oberklassen bieten die Möglichkeit, diverse Mechanismen für ein komplettes Framework bereitzustellen, die in der Programmiersprache fehlen. Trotz des bereits genannten Nachteiles mangelnder Kombinierbarkeit verschiedener Frameworks, ist dieses Argument oft entscheidend für die Verwendung einer gemeinsamen Oberklasse, oft Object genannt. In dieser Klasse wird oft ein Meta-Object-Protocol spezifiziert, daß in der Programmiersprache fehlt oder ungenügend ist.

[WG95] beschreiben die Verwendung einer Klasse Object für eine Vielzahl häufig benötigter Mechanismen. In ET++ wird sowohl das Problem persistenter Daten als auch ein Beobachtungsmechanismus in dieser Klasse realisiert oder vorbereitet. Abgeleitete Klassen implementieren die aufgeschobenen Methoden (vgl. Kapitel 3).

Spezialisierte Klassen müssen somit ein mächtiges Protokoll erfüllen, unabhängig davon, ob diese Funktionalität benötigt wird. Es ergeben sich schwere Objekte mit umfangreicher Schnittstelle (vgl. [Calder95]). Speziell in ET++ wurde versucht, diese Probleme weitgehend zu vermeiden. [WG94] betont das Prinzip „*do not bother the programmer*“ bezüglich der Notwendigkeit, geerbte Methoden implementieren zu müssen, andererseits hat sich aber die gewählte Lösung für dieses Framework offenbar bewährt:

> *„ET++ is organized as a ''mostly'' single rooted class library ... [this organization] enabled the implementation of some valuable infrastructure inherited by all descendants of Object.“* [WG94]

Ebenso unterschiedlich kann auch die Bedeutung einheitlicher Mechanismen für unterschiedliche Aufgaben betrachtet werden. Gamma et. al. versuchen für ET++ wenige Mechanismen möglichst universell einzusetzen. So wurde z.B. die Beobachtung bereits in Object realisiert und steht somit allen Objekten zur Verfügung. Im Gegensatz dazu erwies sich die Verwendung eines einheitlichen Mechanismus für verschiedene Aufgaben bei dem in dieser Arbeit betrachteten Framework (siehe Kapitel 4) gerade als schwer lehrbar, diese Implementation wurde zwischenzeitlich abgeändert. Im folgenden Abschnitt werden Design Pattern als Beschreibungsmittel solcher Mechanismen vorgestellt, die WAM - Methode gibt einen Kontext und eine Anleitung für deren Verwendung (Abschnitt 2.3).

2.2 Design Pattern

> *„There is an undeniable demand to capture already proven and matured object-oriented design“* [Pree94a]

Objektorientierte Programmiersprachen wurden eingeführt mit dem Ziel, die Wiederverwendbarkeit von Software Komponenten zu verbessern. Die einfache Ebene ist dabei „*code reuse*“, also die mehrfache Nutzung einer Implementation, weitaus größere Bedeutung hat aber die Wiederverwendung des Entwurfs, also „*design reuse*“. Frameworks erlauben nicht nur die Verwendung einzelner Klassen in verschiedenen Projekten, sondern größerer Strukturen bis hin zu kompletten - generischen - Anwendungen. „*Frameworks sind ein mögliches Vorgehen, wie Design-Strukturen im Hinblick auf eine spätere Wiederverwendung beschrieben werden können.*“ [Gamma92] Als Alternative stellt Gamma Design Pattern als „*Konzepte eines neuen Ansatzes für die Unterstützung des objektorientierten Entwurfs*“ vor.

Nach Motivation und Einführung dieser Entwurfs-Muster folgt eine kurze Darstellung der derzeitigen Entwicklung und aktueller Diskussionen. Der wichtige Zusammenhang zwischen Design Pattern und Frameworks wird geklärt und damit die Bedeutung dieses Ansatzes für die Framework-Entwicklung.

2.2.1 Motivation für „Muster“

Während die meisten objektorientierten Methoden die Anwendung von Vererbung betonen, wird die Gestaltung der Interaktion zwischen Objekten sowie deren Komposition nicht ausreichend berücksichtigt. Design Pattern sollen diese Lücke schließen, wobei es sich nicht um eine neue Methode mit bestimmtem Vorgehensmodell handelt, sondern eine Ergänzung zu bestehende Methoden. Design Pattern verbessern die Qualität der Softwarearchitektur, vor allem die flexible Wiederverwendbarkeit und Erweiterbarkeit der Komponenten ([GoF95]: „*increased flexibility and reusability*“). [Gamma92] stellt fest:

> *„Es gibt in objektorientierten Systemarchitekturen bestimmte Arten der Organisation von Klassen, die immer wieder auftreten und die zu einer eleganten, gut strukturierten Lösung beitragen.“*

Das Ziel laut [Gamma92]:

> *„Die Wiederverwendung von Design-Strukturen und Design-Erfahrungen. Dem Designer werden für diesen Zweck eine Sammlung abstrakter Design-Strukturen zur Verfügung gestellt, die er beim Entwurf anwenden kann.“*

Gamma et. al. sehen Design Pattern als Mittel, die schwierige Aufgabe des objektorientierten Entwurfs hinsichtlich Wiederverwendbarkeit zu bewältigen. Ausgehend von der Erkenntnis erfahrener Entwickler, flexible Lösungen nicht im ersten Versuch zu finden, sollen Design Pattern die gewonnenen Erfahrungen der Experten („*such experience is part of what makes them experts*“ [GoF95]) festhalten, und damit anderen Entwicklern verfügbar machen:

> *„Design Patterns capture solutions that have developed and evolved over time."*
>
> *„A designer who is familiar with such pattern can apply them immediately to design problems [...]. The purpose of this book is to record experience in designing object-oriented software as design patterns. [...] Our goal is to capture experience in a form that people can use effectively."*

Booch meint im Vorwort zu [GoF95], daß „Muster" in den meisten objektorientierten System enthalten sind („.. *object-oriented architectures are full of patterns* ...") und deren ausdrückliche Beachtung zu einfacheren und besser verständlichen Architekturen führen kann.

Buschmann et. al. [BMRSS96] motiviert (Design) Pattern ähnlich:

> *„Patterns help to build on the collective experience of skilled software engineers. They capture existing, well-proven experience in software development and help to promote good design practice. Every pattern deals with a specific, recurring problem in the design or implementation of a software system."*

[Pree94] betrachtet Design Pattern als die natürliche Fortsetzung üblicher Techniken der Softwareentwicklung.

> *„Programmers tend to create parts of a program by imitating, though not directly copying, parts of programs written by other, more advanced programmers. […] Such imitation is as old as programming.*
>
> *The design pattern concept can be viewed as an abstraction of this imitation activity."*

Die besondere Leistung besteht darin, die Erfahrungen explizit zu machen und anderen Entwicklern zur Verfügung zu stellen. Damit muß dieser nicht erst aus bestehenden Programmen das verwendete Design abstrahieren, sondern bekommt eben dieses dargestellt.

Zur Beschreibung von Design Pattern gibt es unterschiedliche Formen. Weit verbreitet ist die Form Kontext - Problem - Lösung (Alexandrian Form[8]). Im allgemeinen ergänzen weitere Abschnitte die Beschreibung, u.a. Beispiele zur Erläuterung des Kontextes, Lösungsvarianten, bekannte Verwendung. [RZ95] untersucht die verschiedenen Beschreibungsformen und deren spezielle Zielsetzung. Während die Alexandrian Form dazu dient, anhand der Design Pattern konkrete Lösungen zu finden (generativ), ist die Beschreibung in [GoF95] mehr deskriptiv und damit geeignet, nicht nur Lösungen zu entwickeln, sondern diese beschriebenen Pattern auch in bestehenden Systemen zu identifizieren. [Riehle95] verwendete eine sehr allgemeine Form, die besonders Struktur und Dynamik beschreibt und damit die Diskussion und den Vergleich [RZ96] von Mustern ermöglicht.

---

[8] [AIS77] „A Pattern Language" und [Alexander79] „The Timeless Way of Building" sind als Ursprung der Pattern Diskussion anerkannt. Gamma, Johnson et. al. haben dieses Konzeptes aus der Architektur in die Softwaretechnik übertragen.

2.2.2 Aktuelle Entwicklungen

Identifikation oder Entwicklung neuer Design Pattern und vor allem deren Beschreibung ist ein wesentlicher Teil aktueller Veröffentlichungen ([Riehle96a], [BR96], [PLoP96]). Einige vorgestellte Pattern betreffen konkrete Programmiersprachen, z.B. C++ [Coplien92], andere sollen hiervon unabhängig bleiben (im Sinne von [GoF95]). Weniger erfaßt wird bisher das Feld anwendungsspezifischer Muster. Eine besondere Form neuer Pattern bringt [Riehle96b] auf, indem er durch Komposition bekannter Design Pattern neue, größere Pattern beschreibt.

Diskussionsgegenstand ist auch die geeignete Beschreibung von Design Pattern. Neben der Differenzierung betreff Absichten der Autoren (generativ, deskriptiv), werden einzelne Abschnitte der Darstellung (z.B. in [GoF95]) näher untersucht. Einen wichtigen Teil bildet hier die exemplarische Codierung, ein Beispiel zum besseren Verständnis und Ansatzpunkt für weitere Diskussion der Vor- und Nachteile. Trotz der Behauptung in [Gamma92], die Muster wären weitgehend sprachunabhängig, zeigt gerade dieser Bereich häufig das Gegenteil. Auch die Darstellung der Struktur durch Klassen und Objekte wird kritisiert. Hier schlägt [Riehle96c] getrennte Beschreibung verschiedener Rollen vor, die in einer Implementation dann möglicherweise innerhalb einer Klasse zusammengefaßt werden. [Pree94] entwickelt Metapattern als höhere Abstraktion gegenüber Design Pattern.

Pattern beschreiben Lösungen auf sehr unterschiedlichem Abstraktionsniveau, angefangen bei einzelnen Methoden in einer bestimmten Programmiersprache bis hin zur Konstruktion kompletter portabler Anwendungen. [GoF95] stellt einen Katalog dar, der zwar nach Art und Verwendung der vorgestellten Design Pattern gegliedert ist, aber auf „gleichartige“ Pattern beschränkt wurde: „*patterns at a certain level of abstraction*“. Demgegenüber unterscheiden Buschmann et. al. zwischen *Architectural Pattern*, *Design Pattern* und *Idioms* [BMRSS96].

Ähnliche Klassifikationen finden sich bei anderen Autoren, z.B. verwenden Riehle und Züllighoven Design Metaphern, Design Pattern und Programming Pattern [RZ95]. In einem späteren Artikel [RZ96] stellen die Autoren einen Bezug zwischen den Pattern-Arten und den wichtigsten Modellen der Softwaretechnik her.

Interpretations- und Gestaltungsmuster (*Conceptual Patterns*, hier werden Design Metaphern zugeordnet) helfen, das Anwendungsgebiet und dessen Aufgabenstellungen zu verstehen und das gewonnene Verständnis (application domain model) mit den Anwendungsexperten und späteren Benutzern zu diskutieren. Die Verwendung von Metaphern (vgl. WAM, Abschnitt 2.3) als Grundlage dieser Pattern ist sehr hilfreich, um Anwendungswelt und Softwaretechnik näher zueinander zu bringen.

> *„Conceptual patterns should be based on metaphors rooted in the application domain.“*

Im Software Design werden dann Design Pattern verwendet, die konzeptionell passend gewählt werden sollen („*fit or complement the conceptual space*“). Design Patterns definieren Struktur, Zusammenarbeit und Verantwortlichkeiten der Komponenten. Die Form wird durch softwaretechnische Konstrukte beschrieben, also Objekte, Klassen, Benutzungsbeziehungen und Vererbung [RZ96].

Programmiersprachliche Konstrukte dienen schließlich zur Beschreibung der *Programming Pattern*, die je nach verwendeter Sprache und Sprachkultur variieren. Mechanismen, die in der einen objektorientierten Sprache bereits integriert sind (z.B. Smalltalk Metaklassen), werden in anderen (z.B. C++) durch Programming Pattern realisiert.

[Pree94] unterscheidet den Bezug von Design Pattern auf einzelne Komponenten oder größere Zusammenhänge. Letztere Pattern sind seiner Ansicht höher zu bewerten und sollen die Konstruktion von Frameworks unterstützen.

> *„Some design pattern approaches focus on the design of single components or of a small group of components, ignoring the framework concept. […] we consider design pattern approaches as more advanced and more important if the framework aspect is stressed.“* [Pree94]

Viele der Design Pattern im Katalog [GoF95] beziehen sich auf die Framework-Konstruktion („*Most of the patterns focus on frameworks.*“ [Pree94]). Verschiedene dieser vorgestellten Design Pattern sind ähnlich, da dieselben objektorientierten Techniken verwendet werden. Pree stellt fest, daß Design Pattern meist Beispiele gelungener Konstruktion in konkreten Anwendungen bzw. Frameworks sind, sorgfältig gewählt und hinreichend allgemeingültig. Als eine höhere Abstraktionsebene, unabhängig von konkreten Implementationen, führt er Metapattern ein:

> *„… we present a more advanced abstraction that allows us to categorize the catalog patterns and to describe them on a metalevel.“*

Es handelt sich nicht um neue Design Pattern, sondern eine abstrakte Beschreibung bestehender Strukturen in Frameworks: „*The seven composition metapatterns ... repeatedly occur in frameworks.*“ Im Vorwort zu [Pree94] stellt Lewis fest, daß mit Metapattern die komplette Framework-Architektur beschrieben werden kann („*metapattern can express all of the notions of frameworks found in popular architectures*“). Ziel ist die Beschreibung bestehender Anwendungen und Frameworks sowie vor allem die Konstruktion neuer Frameworks. Dieses Ziel haben gemäß Pree auch bereits Design Pattern in [GoF95], nur muß hier jeweils noch vom Benutzer die Übertragung in neue Anwendungsgebiete vorgenommen werden.

> *„We introduce the term metapatterns for a set of design patterns that describe how to construct frameworks independent of a specific domain.“*
>
> *„Metapattern permit different levels of flexibility by combining basic object-oriented concepts.“* [Pree94]

Metapattern sind bereits abstrakt formuliert. Die Wahl des geeigneten Musters richtet sich nach dem Grad gewünschter Flexibilität, also besonders der Frage, ob die Konfiguration erst zur Laufzeit erfolgt bzw. geändert wird. Ausgangspunkt ist die Identifikation der „Hot Spots“ (vgl. 2.1.4), also der Elemente eines Frameworks, die flexibel bleiben müssen. Während dieser Analyse des Anwendungsfeldes müssen konventionelle Techniken eingesetzt werden, Metapattern ergänzen die vorhandenen Methoden dann bei der Umsetzung. [Pree94]

> *„Once the desired hot spots are identified, the characteristics of meta-patterns … assist in supporting the appropriate level of flexibility.“*

Methoden zur objektorientierten Softwareentwicklung geben bisher wenig Unterstützung bei der Framework-Konstruktion. Pree fordert die Konzentration auf Frameworks („*framework centered software development*“) und sieht seinen Vorschlag als ersten Schritt in diese Richtung: „*The hot-spot-driven approach ... is a first step in that direction*„. Nach der Identifikation flexibler Komponenten können diese durch Metapattern umgesetzt werden. Später dokumentieren Metapattern die fertiggestellten Frameworks und helfen bei der Anwendungsentwicklung.

Gegenüber Design Pattern vernachlässigen Metapattern den Bezug auf ein konkretes Problem in festgelegtem Kontext, für das eine Lösung mit Vor- und Nachteilen angeboten werden soll. Statt fachlicher Motivation für eine Anwendung stehen die technischen Möglichkeiten durch objektorientierte Mechanismen im Vordergrund. Metapattern zeigen, wie die Unabhängigkeit einzelner Komponenten erreicht werden kann. Auch damit verbundene Probleme werden auf abstrakter Ebene diskutiert.

Design Pattern sind weniger erfahrenen Entwicklern als „Kochbücher“ oft zu kompliziert für von den Autoren propagierte sofortige Umsetzung. Design Pattern geben aber jedenfalls eine Anleitung, in welchen Zusammenhängen eine gewisse Entkopplung möglicherweise angebracht sein könnte und wie dies geschehen kann. Metapattern erreichen dagegen ein Abstraktionsniveau, daß einer direkten, konstruktiven Anwendung entgegensteht.

Dokumentation eines Frameworks durch Metapattern kann die fachliche Intention der Konstruktion nicht offenlegen, hier sind übliche Design Pattern besser geeignet. Soll allerdings konkret erläutert werden, welcher Grad an Flexibilität erreicht und welches technische Mittel dazu verwendet wurde, so ergänzen Metapattern bestehende Design Pattern. Auch könnte die Beschreibung von Design Pattern vereinfacht werden, indem sich deren Autoren auf jeweils zugrunde liegende Metapattern berufen.

Pree ist eine gute Zusammenfassung der Bedeutung und Möglichkeiten objektorientierter Mechanismen für die Framework-Konstruktion gelungen. Dies kann sowohl den Entwurf neuer Frameworks unterstützen, als auch eine Ergänzung zu üblichen Design Pattern darstellen. Das hohe Abstraktionsniveau macht Metapattern von konkreten Anwendungsgebieten unabhängig, stellt aber zugleich hohe Anforderungen an die Benutzer dieser Technik.

2.2.3 Pattern Language, Pattern System, Handbuch

Während [GoF95] einen Katalog lose zusammenhängender Design Pattern vorstellt, bemühen sich derzeit diverse Autoren um enger gekoppelte Sammlungen. [BMRSS96] verwendet den Begriff „System of Patterns“, andere sprechen - in Anlehnung an Alexander - von „Pattern Language“. [RZ96] argumentiert für die Bezeichnung „Handbuch“.

Johnson [Johnson92] und Beck [BJ94]) verwenden Design Pattern, um die komplette Architektur eines Softwaresystems (das Framework „HotDraw“, vgl. 2.2.4) zu beschreiben. Vorteile dieser Beschreibung ist insbesondere die Darstellung von Designentscheidungen („Warum“), anhand derer andere Entwickler die gewählte Struktur besser verstehen können.

[Perry96] beschreibt die softwaretechnische Entkopplung von GUI und Anwendung durch die Displayer/Data Pattern Language. Dabei definiert er:

> *„A pattern-language is a set of related patterns (not necessarily design patterns). The context of each pattern is influenced by the application of other patterns in the pattern-language.“*

Als Vorzüge der Beschreibung durch aufeinander aufbauende Pattern nennt Perry die Anpaßbarkeit an spezifische Systeme. Der Anwender kann aufgrund der Beschreibung die Motivation und die Konsequenzen der Designentscheidungen nachvollziehen und Alternativen bewerten.

> *„Each pattern identifies a problem, the context in which it occurs, the forces that influence it, and suggest a solution. Consequences to accepting the solution are also considered part of the pattern.“*
>
> *„[...] It must therefore be explained in a comprehensive form, and must be adaptable to different situations. Patterns [...] force the writer to express the motivation behind decisions. [...] Patterns also encourage the writer to state the implications and drawbacks of the design decisions.“*

Ziel des von Buschmann et. al. vorgestellten „System of Patterns“ [BMRSS96] ist vor allem, dem Benutzer eine im Vergleich zu einfachen Katalogen bessere Anleitung zu geben, wann welche Pattern benutzt werden. Innerhalb des Systems soll eine ausreichende Menge an Pattern als Basis beschrieben werden - sowie die Beziehungen zwischen Pattern verschiedener Kategorien. Richtlinien zur Implementation, Kombination und praktischen Benutzung in der Softwareentwicklung sind laut Definition Teil eines Pattern Systems. Die von Alexander übernommene Bezeichnung Pattern Language lehnt [BMRSS96] ab, da die vorgestellten Sammlungen von Mustern jeweils nur Ausschnitte darstellen, nicht jedoch eine vollständige Sprache.

[RZ96] verwirft ebenfalls die Bezeichnung „Pattern Language“ („*we think that our pattern sets are neither used in a linguistic nor in a computer science sense of the term language*“). Gegenüber der Bezeichnung „System“ wird die unvereinbare Besetzung dieses Begriffes in der Systemtheorie angeführt. Während jedoch die Bezeichnung „pattern catalog“ auf lose

gekoppelte Sammlungen beschränkt bleiben sollte, wird entsprechend der geplanten Verwendung die Bezeichnung Handbuch empfohlen:

> *„We therefore propose to (re-) use the notion of handbook (Anderson 1993[9]), being defined as a handy work of print that concisely summarizes the relevant concepts from a domain."*

Hier wird klargestellt, daß ein solches Handbuch nur innerhalb einer festgelegten Domäne eingesetzt werden kann. Riehle und Züllighoven betonen im Text auch die Notwendigkeit eines gemeinsamen Hintergrundes von Schreibern und Lesern solcher Handbücher, dessen Darstellung enthalten sein sollte, ebenso die Benennung des geplanten Anwendungsgebiets und Vorgehensmodells (z.B. evolutionäre Softwareentwicklung).

Pattern innerhalb eines Handbuches sind voneinander abhängig, und können als gerichteter Graph angeordnet werden. Dabei sei an die Klassifikation erinnert (vgl. 2.2.2), wobei aber auch innerhalb einer Kategorie die einzelnen Design Pattern voneinander abhängen können. In der linearen Anordnung werden jeweils erst diejenigen Pattern beschrieben, die den Kontext für weitere Pattern bilden.

> *„Conceptual patterns logically precede design patterns which logically precede programming patterns."*

[RZ95] ist eine Beschreibung der WAM-Methode in Patternform (hier als Pattern Language bezeichnet) und als Basis für ein Handbuch in der oben dargestellten Form geeignet. Auf die Werkzeug und Material Metapher gehe ich im Abschnitt 2.3 näher ein.

2.2.4 Frameworks und Design Pattern

> *„Bei einem Framework wird der Entwurf ... in der Form von Quellcode festgehalten. Die Wiederverwendung umfasst bei einem Framework deshalb Code und Design." [Gamma92]*
> *„Each design pattern focuses on a particular object-oriented design problem or issue." [GoF95]*

Frameworks wie auch Design Pattern sollen dabei helfen, Design wiederzuverwenden. Es wird jeweils eine Lösung für ein bestimmtes Problem angeboten, die Verwendung ist nur in einem festgelegten Kontext sinnvoll. Beide Techniken adressieren ein ähnliches Ziel, wählen aber unterschiedliche Wege. Design Pattern werden meist in einer Form beschrieben, zu der auch eine Beispiel-Implementation gehört („*also provides sample code to illustrate an implementation*" [GoF95]). Dieser Code soll aber nicht direkt verwendet werden, sondern lediglich einen Hinweis auf eine mögliche Implementation geben. In der jeweiligen spezifischen Situation können sich aufgrund konkreter Anforderungen völlig andere Codierungen ergeben.

---

[9] B.Anderson: „Workshop Report: Towards an Architecture Handbook", OOPSLA'92 Addendum, OOPS Messenger, April 1993

Wiederverwendung von *Code und Design* ist dagegen erklärtes Ziel des Frameworkansatzes. Ein Framework wird direkt benutzt, lediglich spezifische Teile der Anwendung sind zu ergänzen. Für den Benutzer ist die konkrete Implementation der im Framework gekapselten Funktionalität nicht wichtig. Er muß nur verstehen, wie er diese benutzen kann, an welchen Stellen er seine eigene Applikation mit dem Framework verbinden muß. [BMRSS96]:

> *„A framework is a partially complete software (sub-)system that is intended to be instantiated. It defines the architecture for a family of (sub-)systems and provides the basic building blocks to create them. It also defines the places where adaptations for specific functionality should be made.“*

Unterschiede zwischen Frameworks und Design Pattern sah [Gamma92] neben der Implementation im spezifischeren Anwendungskontext von Frameworks, einem geringeren Abstraktionsgrad und der relativen Größe. Letzteres dürfte inzwischen als Merkmal ausfallen, da es neben großen Application-Frameworks auch viele kleinere Frameworks gibt und andererseits Pattern jenseits der Kategorie von Design Pattern in [GoF95] (z.B. Conceptual Pattern in [RZ95]) oder z.B. die Composite Pattern [Riehle96b] dem entgegen stehen.

Design Pattern sind grundsätzlich abstrakter als Frameworks und damit eher in einen anderen Bereich übertragbar. Allerdings stellen sie weitaus größere Anforderungen an den Benutzer. Trotz aller Äußerungen der Autoren (Gamma, Johnson, Buschmann, etc.) betreff Nutzung von Design-Erfahrungen durch weniger erfahrene Software - Entwickler, muß doch vor zu großen Hoffnungen gewarnt werden. Die in der Beschreibung integrierte Beispiel-Implementation kann dazu verführen, diese direkt zu benutzen - statt eine der konkreten Verwendung angemessene Adaption vorzunehmen. In vielen Fällen wird es schwerfallen, alle Voraussetzungen und Konsequenzen eines Musters zu überblicken, um die geeignete Auswahl zwischen scheinbar ähnlichen Pattern zu treffen. (Metapattern [Pree94] ignorieren anwendungsbezogene Unterschiede völlig und geben unerfahrenen Entwicklern diesbezüglich keinerlei Anleitung.)

[RZ96] warnt vor Problemen, wenn ein Pattern von dem Bereich, aus dem heraus es abstrahiert wurde, in einen neuen Verwendungszusammenhang transportiert werden soll. Hierzu muß der Anwender sowohl die Struktur, als auch den Hintergrund des Musters hinreichend verstanden haben. In jedem Fall gehört es zur Verwendung von Design Pattern, daß der Entwickler selbst die Implementation vornehmen muß. Diese liegt mit Frameworks bereits vor.

> *„Kann bei der Lösung einer Problemstellung auf einem Framework aufgebaut werden, ist bereits ein grosser Teil der Entwurfsarbeit von den Entwicklern des Frameworks geleistet worden.“* [Gamma92]

Im Gegensatz zu Design Pattern enthalten Frameworks also bereits eine fertige Lösung für das jeweilige Problem, während die Design Pattern dem Entwickler nur eine Hilfe für die eigene Implementation geben.

> *„A framework is usually defined as a collection of classes that together solve a general problem, and that are intended for specialization through subclassing. The main difference between a framework and a pattern is that while the pattern tells the reader how to solve a problem, the framework provides a canned solution.“* [Reenskaug95]

Zur Beziehung zwischen Frameworks und Design Pattern sagt [BMRSS96]:

> *„To achieve adaptability and changability with an application framework, you are not restricted to object-oriented techniques such as inheritance and polymorphism - you can also use patterns. […] From the perspective of application frameworks, patterns can be seen as their building blocks.“*

Vergleichbar verlangt auch [RZ96] die Konstruktion von Frameworks unter Verwendung von Design Pattern. Viele Pattern zielen darauf ab, die Flexibilität und Wiederverwendbarkeit einer Software zu erhöhen, und dies sind für Frameworks entscheidende Qualitätsmerkmale. Auch nach Pree sollen Design Pattern bei der Konstruktion neuer Frameworks helfen, ebenso bei deren Verwendung.

> *„Here, we see a close connection between design pattern and frameworks. A framework should incorporate and instantiate design patterns, in order to ‘‘enforce‘‘ the reuse of designs in constructive way.“* [RZ96]
>
> *„[Pattern] … can help to … construct new frameworks which incorporate matured and proven designs.“* [Pree94a]
>
> *„Design patterns support the (re)use and development of frameworks.“* [Pree94]

Durch die Verwendung von Design Pattern zur Konstruktion eines Frameworks kann gleichzeitig eine weitere Stärke dieser Muster ausgenutzt werden. Pattern erhalten einen Namen und erweitern das Vokabular für die Kommunikation innerhalb des Entwicklerteams [Gamma92]. Die Diskussion erfolgt nicht mehr auf Ebene einzelner Klassen, sondern benannter Klassenteams. Design Entscheidungen können anhand der angestrebten Technik getroffen werden, ohne daß bereits eine konkrete Implementation vorliegt.

> *„Patterns provide a common vocabulary and understanding for design principles.“* [GHJV93]
>
> *„Naming a pattern immediately increases our design vocabulary. It let us design at a higher level of abstraction.“* [GoF95]

Gleichzeitig kann ein Framework gut durch Patterns dokumentiert werden ([BJ94]: HotDraw). Gamma et. al. empfehlen, bestehende Software nachträglich durch Design Pattern zu dokumentieren. Wird ein Framework von vornherein auf Basis bekannter Design Pattern entwickelt bzw. zusätzlich benötigte Muster identifiziert und beschrieben, so können die Autoren des Frameworks dessen Benutzern gleich hieran den Entwurf erläutern (soweit zur Implementation verwendete Mechanismen für die Benutzung relevant sind). Die Wartung und

die Weiterentwicklung eines Frameworks profitiert von dieser Form der Dokumentation in besonderem Maße, da auch eventuelle Probleme und Seiteneffekte sowie die getroffenen Entscheidungen selbst anhand von Design Pattern gut dargelegt werden können. Bereits [Gamma92] nennt den Dokumentationsaspekt als Ziel des Ansatzes:

> *„Die Reduzierung des Einarbeitungsaufwandes in eine Klassenbibliothek. Der Klassenproduzent soll für den Klienten festhalten können, welche Organisationsformen und Rollen er verwendet hat.“* [Gamma92]
>
> *„the documentation of a class library is a difficult problem. Design patterns represent one powerful possibility for accomplishing that task.“* [Pree94]
>
> *„Patterns are a means of documenting software architectures.“* [BMRSS96]

Pree sieht in der Dokumentation von Frameworks den hauptsächlichen Verwendungszweck von Design Pattern. Dabei ist die Unabhängigkeit von der konkreten Implementation wichtig, also die Beschreibung auf einem höheren Abstraktionsniveau.

> *„The main purpose of the framework-centered design pattern approaches is to describe the design of a framework and its individual classes without revealing the implementation details.“* [Pree94a]
> *„Design patterns try to describe frameworks on an abstraction level higher than the corresponding code that implements these frameworks.“* [Pree94]

Design Pattern als Bestandteil der Framework - Dokumentation stellen auch [LS96] heraus. Ein weiterer Bestandteil sollten „Kochbücher“ und Beispiele sein. Diese bieten einen Einstieg in das Framework, decken jedoch nur einige Anwendungsmöglichkeiten ab („*A framework used extensively tends to be used in ways never conceived by its designers.*“). Anpassung und Erweiterung von Frameworks ist nur möglich, wenn die dynamischen Strukturen verstanden werden. Hierzu werden zusätzliche Diagrammtypen und Werkzeuge vorgeschlagen, die eine Betrachtung der Dynamik zur Laufzeit ermöglichen (vgl. Kapitel 5).

Eine weitere Verbindung zwischen Pattern und Frameworks folgt aus der Verwendung von Design Pattern zur Konstruktion von Frameworks. Die so konstruierten Teile eines Frameworks sind spezifische Instantiierungen des Pattern. Formalisierung von Pattern zwecks automatischer Generierung von Lösungen, sehen viele Autoren skeptisch (vgl. [BMRSS96] und [RZ96]), innerhalb eines Anwendungssystems gibt es aber oft für einige Pattern konkrete Implementationen. Eine direkte Wiederverwendung in anderem Kontext bietet sich an, sollte aber fallweise genau geprüft werden (vgl. „Beispiel - Code“).

Buschmann et. al. schreiben: „*... an application framework can be seen as a pattern for complete software systems in a given application domain*“. Frameworks sind konkreter Code und keine Design Pattern. Es kann aber überaus hilfreich sein, die im Framework verwendeten, bekannten Pattern und insbesondere neue zu identifizieren. Insofern könnte im Sinne von [BMRSS96] ein Architektur- Muster für einen bestimmten Anwendungsbereich auf Basis eines Application-Frameworks beschrieben werden.

2.3 Werkzeug und Material Metapher

Gegenstand dieser Arbeit ist ein Application-Framework (BibV30, siehe Kapitel 4), daß die Entwicklung von Anwendungen gemäß der Werkzeug-Material Metapher (WAM) ermöglicht. Nachfolgend wird WAM erläutert und die Beziehung zwischen den Entwurfsmetaphern und Frameworks untersucht. Der Ausgangspunkt für WAM ist das Leitbild vom „*Arbeitsplatz für qualifizierte menschliche Tätigkeit*“. Dazu [RZ96]:

> *„A leitmotif is a general principle that guides software development. [...] It makes the underlying beliefs and values explicit, that drive software design as a whole.“*
> *„Our leitmotif [...] is the well equipped workplace for qualified human work at which experts carry out their work and take the responsibility for it.“*

Die Definition dieses Leitmotivs legt eine Sichtweise auf die zu entwickelnde Software und deren Benutzer fest und dient als Anleitung für die Gestaltung der Benutzungsschnittstelle. Außerdem wird ein Rahmen der mit WAM zu entwickelnden Anwendungen gesteckt. So wie Frameworks jeweils für spezifische Problemstellungen entwickelt werden und Design Pattern Lösungen in genau beschriebenem Kontext anbieten, wird WAM in dem durch das Leitbild definierten Umfeld eingesetzt. Der Ansatz bezieht sich auf „*die Entwicklung interaktiver Anwendungssysteme auf graphischen Arbeitsplatzrechnern*“ ([GKZ93]), eignet sich in der hier betrachteten Form aber z.B. nicht für technische Real-Time-Applikationen.

WAM beinhaltet sowohl ein methodisches Vorgehen für die Softwareentwicklung, als auch eine Anleitung zur technischen Umsetzung. Das evolutionäre Modell mit starker Integration der späteren Anwender in den Entwicklungsprozeß ist nicht Gegenstand dieser Arbeit (siehe z.B. [GKZ93]). Nachfolgend wird der fachliche Entwurf sowie dessen softwaretechnische Umsetzung in Klassen betrachtet.

Laut [GKZ93] läßt sich der WAM - Ansatz „*durch eine möglichst bruchlose Verbindung der Analyse des Anwendungsbereich und der Entwicklung der DV-Technik charakterisieren*“. Dazu werden die Leitmetaphern Werkzeug und Material verwendet.

> *„From the long tradition of craftsmanship we have learned that human work has often found a physical embodiment through tools that craftsmen use to work on materials. We have taken this basic notion as a starting point for understanding what tools and materials are and how we can extend this concept to software tools and materials.“* [RZ95]

> *„Die Werkzeug - Material Metapher beruht auf der Annahme, daß in Arbeitssituationen ein intuitiver Unterschied festgestellt werden kann zwischen Dingen, an denen man arbeitet (Materialien), und Dingen, die Arbeitsmittel sind (Werkzeuge).“* [Lilienthal95]

Die fachliche Modellierung teilt die Komponenten der Arbeitswelt in Software-Werkzeuge und mit ihnen bearbeitete Materialien. Technisch setzen Material-, Aspekt- und Werkzeugklassen diese Komponenten um. Werkzeuge bestehen - ähnlich dem MVC- Muster - aus getrennter Interaktionskomponente (IAK) und Funktionskomponente (FK).

Riehle und Züllighoven haben WAM in Form einer „Pattern Language“ beschrieben:

> *„The term Tools & Materials Metaphor characterizes our overall approach. We explain the approach and its underlying idea by using a pattern language“*
>
> *„... we have three levels within our pattern language for analyzing, designing and constructing software systems:*
>
> 1. *The design metaphors within the realm of a leitmotif guiding our perception and our thinking;*
> 2. *The design metaphors helping us to transform our design ideas into a concrete software design;*
> 3. *The programming patterns as basic means and forms for expressing software building blocks.“* [RZ95]

Ausgangspunkt ist das bereits genannte Leitbild, die Design Metaphern bilden die erste Ebene der „Sprache“ (vgl. 2.2.3: [RZ96] motivieren den Begriff Handbuch statt pattern language und wählen für Design Metaphern die Bezeichnung conceptual patterns). Diese Metaphern ermöglichen eine Beschreibung der Anwendungswelt mit Begriffen, die sich an dieser orientieren. Design Pattern erlauben die Umsetzung dieses Entwurfes in softwaretechnische Konstrukte, Objekte, Klassen und deren Beziehungen. Die dritte Ebene programmiersprachlicher Muster wird hier nicht weiter betrachtet.

2.3.1 Entwurfsmetaphern

WAM basiert auf den Metaphern Werkzeug, Aspekt und Material sowie Umgebung und Automat. Materialien sind passive Elemente, über die der Benutzer Informationen erhalten möchte oder die er verändert. Die möglichen Umgangsformen sind spezifisch für ein bestimmtes Material. Das Material ist der Arbeitsgegenstand des Benutzers. Dagegen sind Werkzeuge nur Hilfsmittel, diese Arbeit an dem Material zu erledigen. Material kann nur durch ein Werkzeug dargestellt und mit diesem verändert werden.

Zwischen Werkzeug und Material besteht somit ein Zusammenhang. Nicht jedes Werkzeug kann jedes Material bearbeiten, andererseits kann ein Material je nach Verwendungszusammenhang mit verschiedenen Werkzeugen bearbeitet werden. Dieses Problem wird durch die Metapher des Aspektes gelöst.

*„**Problem** We never work with a tool on a material in a generic way, but use our knowledge to select the right tool and handle it in a specific way suitable for the combination of tool, material and work at hand.“*

*„**Solution** We make the relationship between a tool and a material explicit by introducing aspects.“* [RZ95]

Dadurch wird die Beziehung definiert, notwendige Umgangsformen festgelegt. Werkzeuge können mit jedem Material arbeiten, das den Aspekt erfüllt. Umgekehrt muß das Material nicht wissen, mit welchen Werkzeug es bearbeitet wird. Es erfüllt lediglich den Vertrag, der durch den Aspekt spezifiziert wurde (siehe Abbildung 2-3).

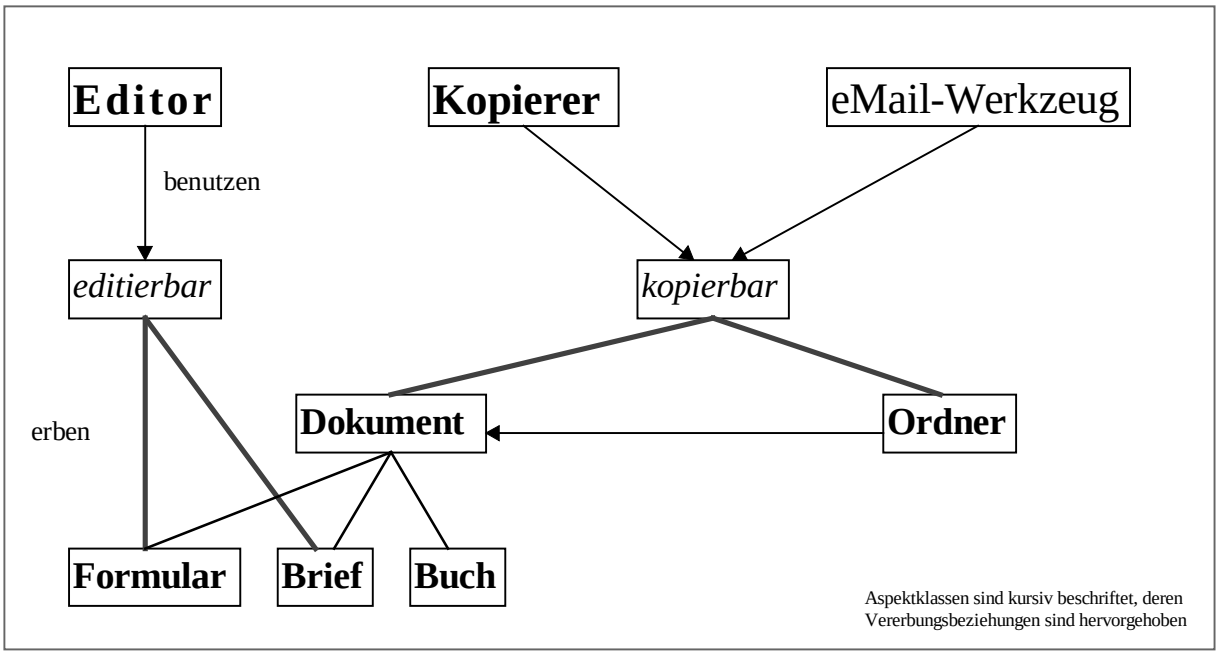


Abbildung 2-3: Werkzeug, Aspekt und Material

Ergänzt werden diese Metaphern um Automaten und die Umgebung. Automaten erledigen regelmäßige Arbeitsprozesse im Hintergrund und haben im Vergleich mit Werkzeugen nur eine rudimentäre Benutzungsschnittstelle zur Änderung variabler Einstellungen. Automaten im Sinne von WAM werden vom Benutzer gezielt benutzt und gesteuert („unterer Kontext“), im Gegensatz zu „großen Automaten“, deren Zweck die Steuerung der Arbeitsabläufe ist („oberer Kontext“). [Gryczan95] gibt eine umfassende Darstellung der Metapher und motiviert die Beschränkung auf „kleine Automaten“.

Ausgehend vom Bild der gut sortierten Werkstatt wird eine Umgebung für Software-Werkzeuge und Materialien eingeführt, entsprechend einem Ort, an dem diese angeordnet werden können (vergleichbar dem „Elektronischen Schreibtisch“). Nähere Erläuterung dieser Metapher finden sich in [Gryczan95], eine Motivation für den Bedarf an einer Umgebung gibt [RZ95]:

> *„A tool or a material is never found in isolation. We always work in well-organized places equipped with the things we need. In a computer system there is no ’’natural‘‘ environment of this kind.“*

2.3.2 Softwaretechnische Umsetzung

Die Implementation des Entwurfes anhand o.g. Design Metaphern wird durch passende Design Pattern ermöglicht. Im wesentlichen sind dies Kopplung von Werkzeugen und Material, die Konstruktion komplexer Werkzeuge sowie die unabhängige Implementation von Werkzeugfunktionalität und deren Darstellung gegenüber dem Benutzer.

Zur Entkopplung spezifischer Werkzeuge von den bearbeiteten Materialklassen, wird die Aspekt - Metapher als eigenständige, meist abstrakte Klasse umgesetzt. Das durch Aspektklassen spezifizierte Protokoll wird vom Material implementiert. Werkzeuge verwenden nicht direkt die Materialklassen, sondern nur Aspekte. Sie können damit Material immer dann bearbeiten, wenn dieses das vom Aspekt definierte Protokoll erfüllt. (Die Implementation mit C++ setzt voraus, daß Materialklassen jeweils von den Aspektklassen erben - Mehrfachvererbung. Werkzeuge nutzen dann Polymorphie zur Bearbeitung der Materialobjekte.)

Vorteil der unabhängigen Konstruktion von Werkzeugen und Material ist die einfache Wiederverwendung bestehender Werkzeuge für neue Materialklassen, die lediglich den Aspekt implementieren müssen. Umgekehrt können auf den bestehenden Aspekten neue Werkzeuge aufgebaut werden. Einen weiteren Schritt in Richtung Wiederverwendung liefert das WAM - Pattern „Tool Composition“ [RZ95]. Aufgrund der Feststellung, daß komplexe Werkzeuge häufig ähnliche Komponenten enthalten, sollen diese leicht wiederverwendet werden können. Werkzeuge werden durch Kombination bestehender, einfacher Werkzeuge konstruiert (vgl. Abbildung 2-4). Das Kontextwerkzeug delegiert seine Aufgaben an die Subwerkzeuge und koordiniert Aufgaben, an denen mehrere Subwerkzeuge beteiligt sind. Zusätzliche Funktionalität kann hinzugefügt werden. Als Beispiel dieser Konstruktion nennen Riehle und Züllighoven [RZ95] einen Kalender, der Termine auflistet (Subwerkzeug Lister). Der Benutzer kann jeweils einen Eintrag auswählen und verändern (Subwerkzeug Editor). Aufgabe des Kontextwerkzeuges (Kalender) ist es, Änderungen im Editor dann auch im Lister zu zeigen und umgekehrt den im Lister ausgewählten Termin in den Editor zu übertragen.

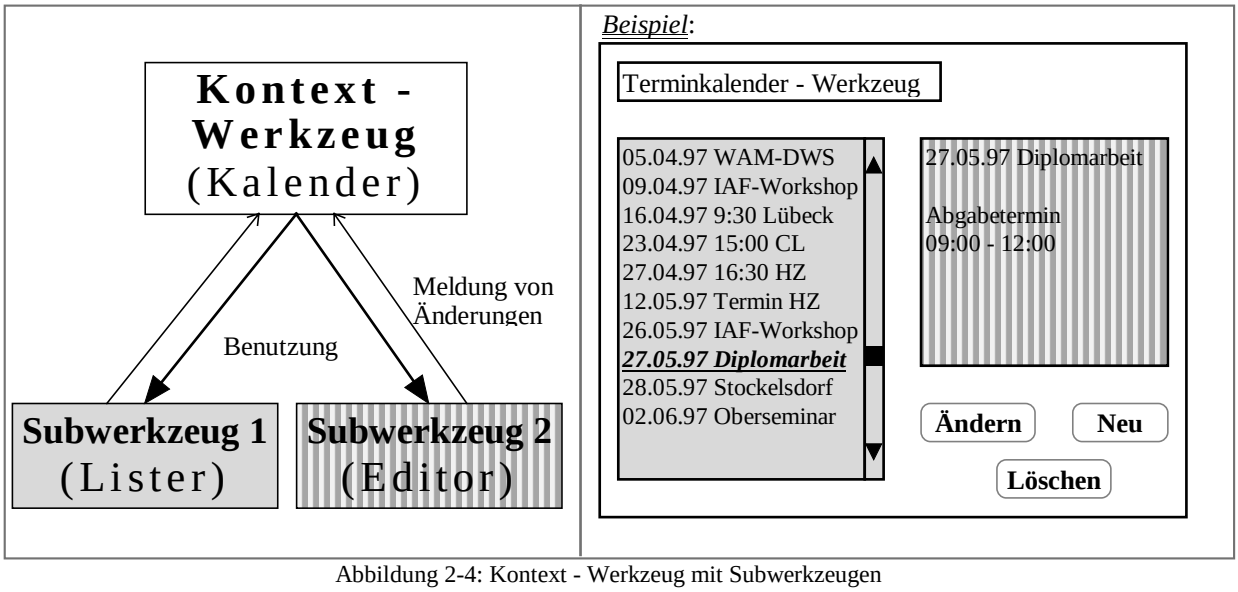


Abbildung 2-4: Kontext - Werkzeug mit Subwerkzeugen

MVC schlägt eine Trennung zwischen Funktionalität (Model), Darstellung (View) und Handhabung (Controller) vor, die in ähnlicher Form in viele Frameworks übernommen wurde. In den meisten Fällen wird die Trennung zwischen View und Controller aufgrund zu enger Kopplung dieser Komponenten aufgegeben. Auch WAM faßt beide Teile zu der Interaktionskomponente (IAK) zusammen und trennt diese von der Funktionskomponente (FK). Ziel ist sowohl die unabhängige Entwicklung beider Bereiche, als auch die Möglichkeit, verschiedene Interaktionskomponenten für dieselbe FK zu entwickeln - und ggf. gleichzeitig zu verwenden. Die Funktionskomponente verfügt über keinerlei Kenntnis der jeweiligen IAK, meldet aber Änderungen ihres abstrakten (an der Schnittstelle sichtbaren) Zustandes als Signale mittels eines Benachrichtigungsmechanismus (Erläuterungen zu diesem Design Pattern z.B. in [RW96] und [Riehle96]). Demgegenüber werden IAK' s jeweils für eine spezielle FK entwickelt und verwenden deren Methoden direkt.

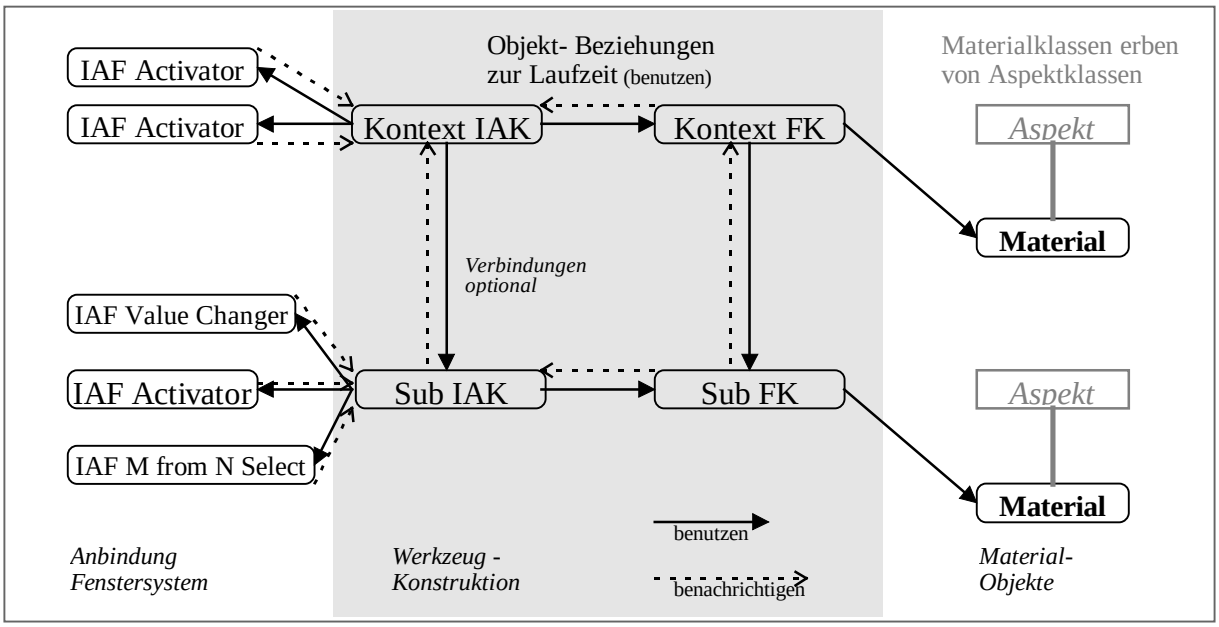


Abbildung 2-5: FK, IAK, IAF (Kontext - Werkzeug mit einem Subwerkzeug)

Bezüglich der Werkzeug/Material - Kopplung über Aspektklassen sei hier ergänzt, daß nur die Funktionskomponente das Material bzw. den Aspekt direkt benutzt (siehe Abbildung 2-5). Komplexe Werkzeuge müssen so zusammengesetzt werden, daß jeweils die Kontext-FK ihre untergeordneten Sub-FK' s benutzen kann und ggf. eine Verbindung zwischen Kontext-IAK und Sub-IAK hergestellt wird. Hierzu siehe das „IP / FP Plug In" Design Pattern in [RZ95]. Weitere Design Pattern ermöglichen die Unabhängigkeit der Interaktionskomponente vom verwendeten Fenstersystem (Interaktionstypen, IAT), die Berücksichtigung persistenter Daten (Materialversorger) und auch Unterstützung kooperativer Arbeitsprozesse (z.B. Postkörbe). Einzelheiten werden, soweit erforderlich, in Kapitel 4 näher erläutert.

An dieser Stelle soll keine tiefere Diskussion der WAM - Metaphern erfolgen, sondern diese in Bezug zur Framework und Design Pattern Diskussion gesetzt werden. Letzteres wurde bereits in [RZ95] und [RZ96] geleistet, indem eine komplette Beschreibung von WAM in Form von Design Pattern erstellt wurde. Metaphern werden dabei selbst als Pattern verstanden.

Auch die Bedeutung eines Leitbildes für die Softwareentwicklung wird erläutert. Dieses wird als Kontext der vorgestellten Pattern verstanden. MVC basiert nach [RZ95] auf dem sehr allgemeinen Leitmotiv der direkten Manipulation. WAM basiert auf dem weitaus spezifischeren Leitbild des gut ausgestatteten Arbeitsplatzes für Experten bzgl. der jeweiligen Tätigkeiten. Dieses Leitmotiv bildet den Hintergrund für die Software - Entwickler und bestimmt die Anforderungen an die konstruierten Anwendungssysteme. [RZ96]:

> *„A leitmotif ... is a ''broad picture'' which can evolve a vision of the future system. It makes the underlying beliefs and values explicit that drive software design as a whole"*

Für das Verständnis und die erfolgreiche Anwendung der WAM - Pattern ist die Kenntnis dieses Hintergrundes entscheidend. Auch andere Design Pattern setzen ein gemeinsames Grundverständnis von Autor und Anwender voraus. Hierzu dient der Abschnitt „Kontext" in den meisten Beschreibungsschemata für Pattern, wobei eine vollständige Beschreibung jedoch grundsätzlich nicht möglich ist.

> *„Patterns can only be communicated and understood sufficiently well if authors and readers share a common background in the relevant domains."*
> *„The background ... cannot be formalized or fully described, but it has to be shared to some degree in order to establish a mutual understanding and a shared practice."* [RZ96]

Anwendungsentwicklung im Rahmen der Werkzeug und Material Metapher wird durch die vorgestellten Metaphern und Entwurfsmuster unterstützt. Frameworks können hier noch einen Schritt weiter gehen, indem ein Application-Framework bereits eine generische WAM-Applikation zur Verfügung stellt. Aufgrund der geringen Einschränkung des Anwendungsgebietes kann dabei eher von einer Anwendungsumgebung gesprochen werden, als von einem Domänen-Framework.

Einige Design Pattern werden hier bereits passend implementiert, um so die Implementation eigener Funktions- und Interaktionskomponenten vorzubereiten, das Framework realisiert die übliche Infrastruktur aller auf diesem Konzept aufbauender Anwendungen. Hierzu gehört ein Benachrichtigungsmechanismus ebenso wie ggf. die Implementation von Postkörben. Auf Basis einer solchen Anwendungsumgebung werden innerhalb einzelner Firmen oder Branchen eigene Domänen-Frameworks entwickelt.

Im Zusammenhang mit der Methode WAM ergibt sich ein weiterer Aspekt der Umsetzung in konkrete Frameworks mit zugehörigen Beispielanwendungen: diese können die Lehre unterstützen. An Universitäten lernen die Studenten die Konsequenzen der Methode praktisch kennen, Erweiterungen der Design Pattern werden hier getestet. Näheres zu diesem Aspekt folgt in Abschnitt 4.5.

2.3.3 Frameworks und Subwerkzeuge

Objektorientierte Programmierung im allgemeinen und Frameworks im speziellen verfolgen das Ziel der Wiederverwendung von Software- Komponenten. Design Pattern unterstützen dieses Bestreben, indem sie durch Kombination grundlegender objektorientierter Techniken - wie abstrakte Oberklassen, Polymorphie und dynamisches Binden - flexible Verbindungen oder die Konfiguration von Frameworks zulassen.

Die Werkzeug-Material Metapher als Methode zur objektorientierten Softwareentwicklung macht unter einem speziellen Leitbild Gebrauch von Entwurfsmetaphern und Design Pattern als Mittel zur softwaretechnischen Umsetzung der Entwürfe.

Zudem führt WAM die Trennung von Funktions- und Interaktionskomponenten, Aspekten als Bindeglied zwischen Werkzeug und Material und den Begriff des Subwerkzeuges ein. Derart lose gekoppelte Komponenten sind leichter wiederzuverwenden, z.B. gibt es Verschiedene IAK' s für eine FK. Werkzeuge arbeiten nicht auf konkreten Materialklassen, sondern nur auf abstrakten Aspektklassen, und können ohne Änderungen auch neues Material bearbeiten.

Subwerkzeuge dienen der Zerlegung großer Werkzeuge in kleinere, eigenständige Klassen mit festgelegter Verantwortlichkeit und definierter Kooperation mit anderen Komponenten einer komplexen Anwendung. Ein Sonderfall sind Subwerkzeuge als Mittel zur Wiederverwendung praktisch fertiger Teil- Werkzeuge. Neue Werkzeuge aus vorhandenen Subwerkzeugen aufzubauen, ist nur begrenzt möglich. In einer konkreten Anwendung sind meist auch spezifische Komponenten erforderlich, während die Subwerkzeuge zwecks Wiederverwendung generisch sein müssen. Welchen Bezug gibt es zwischen Frameworks und Subwerkzeugen?

Zum Einen bilden Frameworks die Infrastruktur für die Entwicklung von Werkzeugen aus lose gekoppelten Komponenten, in diesem Fall Subwerkzeugen. Zum Anderen sehe ich aber gerade bzgl. generischer Subwerkzeuge den Einsatzschwerpunkt von Frameworks. Da ein Anwender mit Werkzeugen arbeiten soll, die speziell seinem Bedarf entsprechen, gilt es, die generischen Komponenten anzupassen, zu konfigurieren.

Frameworks sind nun gerade Software- Bausteine, die z.B. durch Parameterobjekte oder durch Subklassenbildung für einen bestimmten Anwendungskontext spezialisiert werden. So sollten generische Subwerkzeuge Bestandteil einer Framework - Bibliothek sein, indem diese als optional verwendbare Mikro-Frameworks, evtl. unter Verwendung anderer Mikro-Frameworks, entworfen werden. Black-Box - Wiederverwendung sollte angestrebt werden, soweit möglich auch schon konkrete Parameterklassen für Standardfälle vorhanden sein.

Da es sich bei dieser Art Subwerkzeuge dann nicht um fertige Komponenten handelt, sondern vielmehr im Gegensatz zu konkreten, fachlichen Subwerkzeugen nach WAM der konkrete Bezug zur Anwendung noch zu ergänzen ist, nenne ich diese Komponenten Halbzeuge. In der Verwendung entstehen durch Ableitung oder Parametrierung dann die Subwerkzeuge einer WAM-Applikation.

2.4 Begriffsbildung im Überblick

Aus den Darlegungen zu Frameworks, Design Pattern und Metaphern sowie deren Beziehungen untereinander läßt sich eine Einordnung bestehender Application-Frameworks ableiten. Entwicklungen an Frameworks können bezüglich deren Hintergründe bzw. der mit diesen Maßnahmen erreichten qualitativen Änderungen untersucht werden. Ich werde die relevanten Begriffe kurz zusammenfassen, um dann in den nächsten beiden Kapiteln diesen Katalog erst auf verschiedene populäre Application-Frameworks, und schließlich auf die Entwicklung der am Arbeitsbereich Softwaretechnik erstellten WAM- Bibliotheken anzuwenden.

Frameworks sind gegenüber Bausteinbibliotheken abzugrenzen, wesentliches Unterscheidungsmerkmal ist die Umkehrung des Kontrollflusses (2.1.1 und 2.1.2). Der Vorteil liegt für den Benutzer in einer vorgegebenen Struktur, in die er seine eigenen Komponenten einfach einpaßt. Für die Betrachtung einer Klasse als Bausteinklasse ist deren Schnittstelle, also die Verwendung von außen entscheidend, nicht die innere Konstruktion.

Kleine Mikro-Frameworks für einzelne Aspekte einer Anwendung werden oft mit der Zeit ausgebaut zu kompletten Application-Frameworks, die dann als generische Anwendungen bezeichnet werden (2.1.3). Voraussetzung für die Zuordnung zu dieser Gruppe ist die Vorgabe einer kompletten Infrastruktur für die Anwendung, während Mikro-Frameworks nur einzelne Aspekte unterstützen - und ggf. Bestandteile größerer Frameworks sind. Eine scharfe Abgrenzung entfällt im Falle der erwähnten Migration.

Es gibt allgemeine Anwendungsumgebungen und weitaus stärker spezialisierte Domänen-Frameworks (2.1.3). Dabei bezieht sich der Begriff Domäne entweder nur auf den Typ der mit dem Framework erstellten Applikationen, oder aber bereits sehr speziell auf Anwendungen innerhalb einer bestimmten Branche oder Firma. Domänen-Frameworks basieren vielfach auf Anwendungsumgebungen, sehr spezielle Versionen können von allgemeineren abgeleitet werden.

Bezüglich der Wiederverwendung sind zu unterscheiden (2.1.4)

- Black-Box versus White-Box Wiederverwendung; maßgeblich für die Einordnung ist die Notwendigkeit von Kenntnissen der Implementation, nicht aber die in der Literatur oft zu findende Verknüpfung mit objektorientierten Techniken der Wiederverwendung.
- Einsatz abstrakter oder konkreter Parameterklassen; abstrakte Parameterklassen sind nicht Voraussetzung für White-Box-Frameworks, erleichtern diese Form der Wiederverwendung aber wesentlich. Konkrete Parameterklassen ersetzen Implementation durch Konfiguration.
- Orientierung an Hot Spots und deren Flexibilitätsgrad; es wird identifiziert, welche Teile der Applikationen Standardisiert sind, und welche flexibel bleiben (Hot Spots). Besonders flexibel sind zur Laufzeit variable Komponenten (erreichbar durch Parameterklassen).
- Verwendung einer gemeinsamen Oberklasse; Klassenhierarchien in Baumform werden gegen sogenannte Wälder mit mehreren Wurzeln abgegrenzt.

Der Einfluß von Design Pattern zeigt sich vor allem in der Dokumentation des Frameworks. Dabei kann das komplette Framework mittels eines Handbuches aus Pattern dokumentiert sein, oder auch Design Pattern nur für einzelne Bereiche identifiziert werden (Abschnitt 2.2).

Eine besondere Bedeutung bekommen Design Pattern, wenn diese bereits die Konstruktion des Frameworks erheblich beeinflußt haben und evtl. auch Benennungen von Framework - Klassen auf die Pattern zurückzuführen sind. Gegebenenfalls implementieren sogar komplette Mikro-Frameworks jeweils genau ein Design Pattern für die Verwendung in anderen Komponenten des Application-Frameworks.

Die Anwendung von Design Pattern hilft bei der Umsetzung erkannter Hot Spots. Metapattern betrachten hier besonders den Aspekt der Flexibilität, die übliche Beschreibung konzentriert sich auf deren Hintergrund und vermittelt dem Benutzer des Frameworks ein Verständnis der gewählten Konstruktion sowie deren Motivation. Konsequenzen für die eigene Anwendung sind so besser zu erkennen, Änderungen am Framework können die explizit dokumentierten Gründe für gewisse Designentscheidungen berücksichtigen.

Als letztes wird der Einfluß von Leitmotiven und Metaphern auf die jeweiligen Frameworks untersucht (siehe Abschnitt 2.3). Diese werden jedoch nicht unbedingt explizit sein und vielfach auch extrem allgemein gehalten. Bewußte Anwendung konkreter Metaphern und deren Dokumentation hilft bei der Zuordnung eines Application-Frameworks zu einer bestimmten Problemstellung bzw. kann den Kontext der vorgesehenen Verwendung beschreiben.

Eine Besonderheit ergibt die Kombination einer Methode zur Software - Entwicklung (z.B. in dieser Arbeit WAM) mit einem Framework, daß deren Umsetzung unterstützt. Orientiert sich der Anwendungsentwickler an der Methode, so kann er davon ausgehen, daß das spezifisch hierfür entwickelte Application-Framework zu seinem Entwurf „paßt“. Bei Verwendung anderer Frameworks stellt sich dagegen die Frage, ob dort implizit verwendete Leitbilder und Metaphern überhaupt für die Implementation geeignet sind. Umgekehrt legt sich das WAM-Framework mit der Methode explizit auf einen bestimmten Anwendungstyp fest (bzw. einer Sichtweise auf die Software) und beugt einer ungeeigneten Verwendung in falschem Kontext vor.

Kapitel 3 Einsatz der Frameworktechnik

Zur Einordnung des WAM-Frameworks im nächsten Kapitel stelle ich hier einige bekannte Frameworks aus kommerziellem und universitären Umfeld vor, beschreibe deren Leistungsumfang und ausgewählte Elemente.

Diese Beispiele erfolgreicher Frameworks geben einen Überblick hinsichtlich Entwicklung und Erfolg der Framework - Technik, der Vergleich der Komponenten soll erreichte Fortschritte dokumentieren. Somit beginne ich dieses Kapitel mit einer kurzen Benennung erster Produkte wie das MVC-Framework in Smalltalk und schließe mit einer Darstellung des aktuellen Ansatzes von Taligent.

Im wesentlichen liegt das Ziel bei der Vorbereitung eines Vergleiches mit Komponenten der BibV30 (WAM-Framework, siehe Kapitel 4), einer Gegenüberstellung des Leistungsumfangs der Frameworks. Zur Beschreibung und Beurteilung dienen die in Kapitel 2 definierten und in ihren Auswirkungen diskutierten Begriffe, vor allem die Arten der Wiederverwendung, Design Pattern und Metaphern sowie Leitbilder identifiziere ich in den einzelnen Frameworks.

3.1 Historie

Ein sehr frühes, objektorientiertes Framework wurde von Reenskaug 1978/79 entworfen und durch Goldberg weiterentwickelt [Reenskaug95]. Das MVC-Framework (Model-View-Controller) ist seit Smalltalk-80 ein wichtiger Bestandteil von Objectworks/Smalltalk [Goldberg83].

Als erste kommerziell verfügbare Anwendungsumgebung nennt [Lewis95] MacApp (Version 1.0 ausgeliefert 1987): *„... MacApp, the first widely used and commercially available application framework.“* Dieses Framework erleichtert die Entwicklung von Macintosh Anwendungen mit konsistenter Benutzungsoberfäche. (*„MacApp simplifies Macintosh application programming by implementing the common application structure“* [Rosenstein95]). Der Anwendungsentwickler konzentriert sich auf seine spezifischen Teile, insbesondere die „main event loop“ und die Behandlung der standardisierten Menüs übernimmt das Framework.

Frühe Macintosh Applikationen und Gespräche mit deren Entwicklern ermöglichten es den Framework - Designern, die Anwendungsdomäne kennenzulernen. Testversionen wurden frühzeitig ausgeliefert und mit den Anwendern diskutiert. Die weitere Entwicklung entspricht laut [Rosenstein95] dem üblichen Framework-Lifecycle: Bei wachsender Anzahl an Benutzern werden verstärkt Schnittstellen konstant gehalten, im Gegensatz zu relativ häufigen Strukturänderungen in der Anfangsphase. Optimales Design auf Basis jeweils aktueller Erkenntnisse wird Kompromissen zugunsten bereits bestehender Anwendungen geopfert.

3.2 Fenstersysteme

Ausgangspunkt vieler Framework-Entwicklungen war und ist eine leichtere Programmierung graphischer Benutzungsschnittstellen. Verschiedene Autoren, z.B. [Calder95], [Pree94] oder [WG94] nennen als Motivation den hohen Aufwand, diese für den Benutzer der Anwendungen vorteilhaften Oberflächen zu programmieren ([WG94] „*... easy to learn and fun to use. Constructing such interfaces ... often requires considerable effort ...*“, [Pree94] „*The development of easy-to-use, consistent, direct-manipulation graphic user interfaces (GUIs) is a difficult and tedious task*“).

3.2.1 Interviews

So war die mangelnde Unterstützung durch GUI-Tools besonders für UNIX-Workstations Anlaß für das 1985 begonnene Projekt Interviews, ein GUI - Application-Framework für das X - Windows - System.

Schwerpunkte wurden auf die Entwicklung von Mechanismen zur Komposition einzelner Oberflächenelemente gelegt. [Calder95] sieht bereits sehr frühzeitig im Projekt einen „*significant effort to extend the support for interface objects to a finer grain than the widget level*“. Im Ergebnis sind die Objekte von InterViews so „lightweight“, daß auch eine sehr große Zahl dieser Objekte innerhalb einer Applikation möglich ist. Zudem können die graphischen Objekte, genannt Glyph, mehrfach benutzt werden, so daß z.B. ein Textsystem jeden einzelnen Buchstaben durch ein eigenes Grafikelement darstellen kann. Gleiche Buchstaben in gleicher Schriftart werden dabei durch ein einziges Objekt repräsentiert.

> *„Application programmers use InterViews objects in two ways: composition and extension“* [Calder95]

Einerseits können vordefinierte Klassen entsprechend zusammengesetzt werden, andererseits werden neue entwickelt. Zum Zeichnen wird ein „Canvas“ verwendet, dessen Protokoll Postscript ähnelt. Konkrete Canvas-Klassen setzen dieses Protokoll auf X11-Display bzw. auch echte Postscript-Dateien um, wobei die geräteunabhängigen Koordinaten transformiert werden. Eine Besonderheit ist die Anordnung der Grafikelemente nach TEX - Art: zwischen zwei Glyphs wird ein Zwischenraum eingefügt und mit einigen Parametern dessen Verhalten z.B. bei Größenanpassungen definiert. Benutzereingaben werden von sogenannten *Handlern* bearbeitet, die Darstellung und das Verhalten einzelner Glyphs kann durch Wrapper beeinflußt werden. (Einzelheiten siehe [Calder95] oder [LCITV92]).

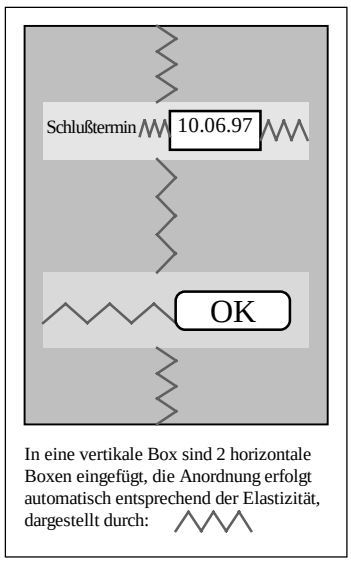


Abbildung 3 - 1: TEX - Layout

[Lewis95] betrachtet Interviews als „Toolkit“. Interviews ist kein Application-Framework, da es lediglich die Entwicklung graphischer Benutzungsschnittstellen unterstützt. Vorgefertigte Glyphs ähneln Bausteinklassen, jedoch wird die Kontrolle zwecks Meldung von Benutzereingaben grundsätzlich dem Mikro-Framework übertragen. Die Komposition von Objekten fertiger Klassen entspricht daher eher der Verwendung konkreter Parameterklassen.

Während die Verwendung fertiger Klassen der Black-Box - Wiederverwendung entspricht, setzt die Entwicklung eigener Glyphs weitgehende Kenntnisse der internen Framework - Strukturen voraus, es handelt sich um White-Box - Wiederverwendung.

Diverse Design Pattern des Kataloges [GoF95] wurden in Interviews identifiziert (obwohl zum Zeitpunkt der Entwicklung diese Diskussion noch nicht begonnen hatte). Leitbild für die Gestaltung von Interviews war direkte Manipulation, betreff Anordnung der Elemente wurden Prinzipien aus dem Textlayout übernommen. Ansonsten ist anzumerken, daß Interviews eben „nur“ die Gestaltung der Oberfläche erlaubt, eine Struktur für die Anwendung als Ganzes wird nicht vorgegeben.

3.2.2 Unidraw - Graphische Editoren

Unidraw ist ein Framework, mit dessen Hilfe graphische Editoren für unterschiedliche Anwendungsgebiete konstruiert werden können. [Lewis95] verweist in seinem Vorwort darauf, daß dieses spezialisierte Application-Framework ein anderes Framework wiederverwendet:

> *„... John Vlissides describes Unidraw, which is an application of the best principles of object-oriented application frameworks. First, Unidraw reuses Interviews instead of reinventing a new GUI-based framework. Then, Unidraw extends InterViews into new domains, e.g., editors for graphical drawing, musical scores, and circuit design.“*

Grafische Editoren ermöglichen dem Benutzer die direkte Manipulation der dargestellten Elemente. Die Grafik - Symbole stehen für spezifische Anwendungsobjekte, z.B. Transistoren im Schaltplanentwurf oder vielleicht Musiknoten. Die Bildschirmdarstellung kann in andere Formate übertragen werden, einerseits wird auf einen Postscript - Drucker ausgedruckt, andererseits soll eine (nicht grafische) Netzliste des Schaltplanes erstellt werden.

Während traditionelle Benutzungsschnittstellen Knöpfe, Menüs und andere Oberflächenelemente anbieten, stellt Unidraw eine Abstraktion zur Verfügung und vereinfacht die Entwicklung grafischer Editoren für sehr unterschiedliche Anwendungsfelder deutlich.

> [Vlissides95]: *„Unidraw defines a reusable design for graphical object editor. Hence, it can reduce design implementation time dramatically“*

Nachfolgend werden wichtige Aspekte des Unidraw - Frameworks näher erläutert, die dessen Charakter als Domänen-Framework veranschaulichen. Black-Box - Wiederverwendung ist nur sehr begrenzt möglich, Parameterklassen sind kaum vorhanden. [Vlissides90] identifiziert die wesentlichen Hot Spots ausdrücklich und realisiert deren Flexibilität durch Anwendung von Design Pattern. Diese werden in seiner Dokumentation (1990) allerdings nicht ausdrücklich erwähnt und als solche beschrieben, genannt und ausgeführt wird nur die direkte Manipulation als Leitbild und die MVC - Architektur (s.u.).

Grundlage des Unidraw Frameworks sind die vier Basisabstraktionen *Component*, *Tool*, *Command* und *External Representation*. Gemäß [Vlissides95][10] beschreibt Component die Darstellung und das Verhalten, Tool ermöglicht die direkte Manipulation, Command führt eine Operation auf Component aus und die External Representation ermöglicht eine Umsetzung zwischen der Darstellung eines Component - Objektes und des zugehörigen Dateiformates.

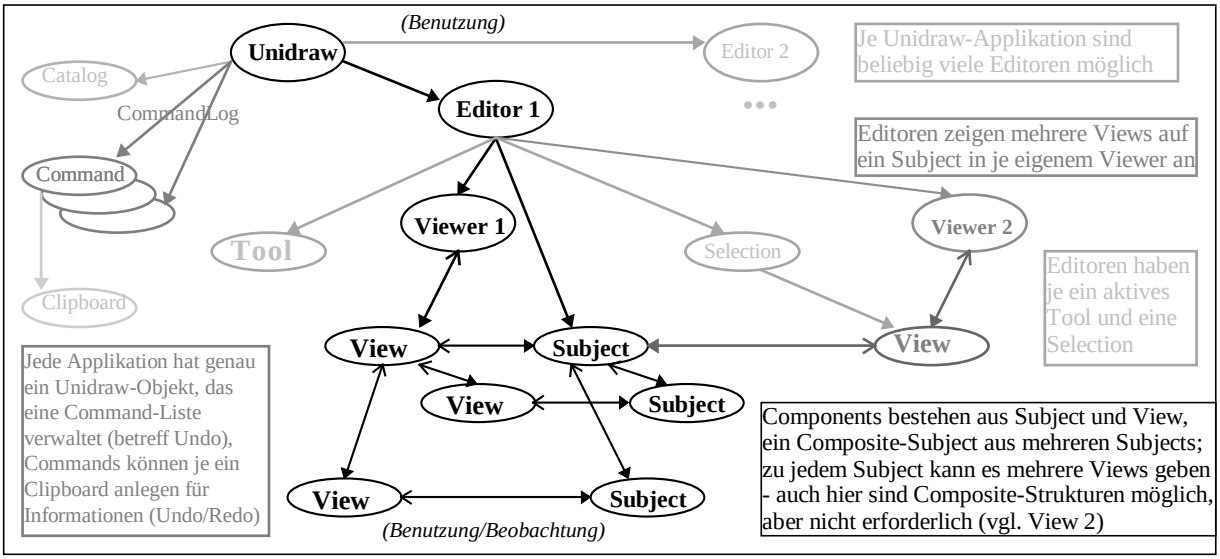


Abbildung 3 - 2: Objekte in einer Unidraw - Applikation zur Laufzeit (vgl. 4.4.2)

Eine Unidraw - Applikation besteht aus Editoren, zu denen jeweils Werkzeuge (Tool) und vom Benutzer aktivierbare Kommandos (Command) gehören. Für Kommandos werden Undo und Redo unterstützt. Jeder Editor kann seine Komponenten (Component) gleichzeitig in mehreren Sichten anzeigen (mehrere Viewer).

„Components have an MVC-like structure: the component subject corresponds to the model for the component, and the component view corresponds to the component's view. (There is no object corresponding to an MVC controller.)" [Vlissides95] Somit kann es zu einem Component-Subject mehrere Component-Views geben, die dann in jeweils eigenen Viewer- Objekten angezeigt und manipuliert werden.

[10] Eine umfassende Darstellung der Unidraw - Architektur findet sich in [Vlissides90], Kapitel 3. In dieser Arbeit gehe ich im Abschnitt 4.4, besonders unter 4.4.2 näher auf Unidraw ein.

Direkte Manipulation erfolgt mittels Werkzeugen. Der Editor bestimmt das gerade aktive Werkzeug, das dann vom jeweiligen Viewer verwendet wird. Während der Manipulation vermittelt ein *Manipulator* - Objekt dem Benutzer einen Eindruck direkter Beeinflussung der Grafikelemente. Diese Manipulatoren werden von dem aktiven Werkzeug erzeugt, ggf. von gerade selektierten Component-Views abhängig. Nach Abschluß der Aktion generiert das Werkzeug mithilfe des Manipulators ein Command - Objekt zur effektiven Ausführung der Aktion.

Unidraw kann die Beweglichkeit von Objekten durch Attribute einschränken, ebenso wird die Reaktion auf Verformung spezifiziert. Connectoren legen Beziehungen zwischen den einzelnen Grafikelementen fest (vgl. Abbildung 3-3, nähere Erläuterungen siehe 4.4.2).

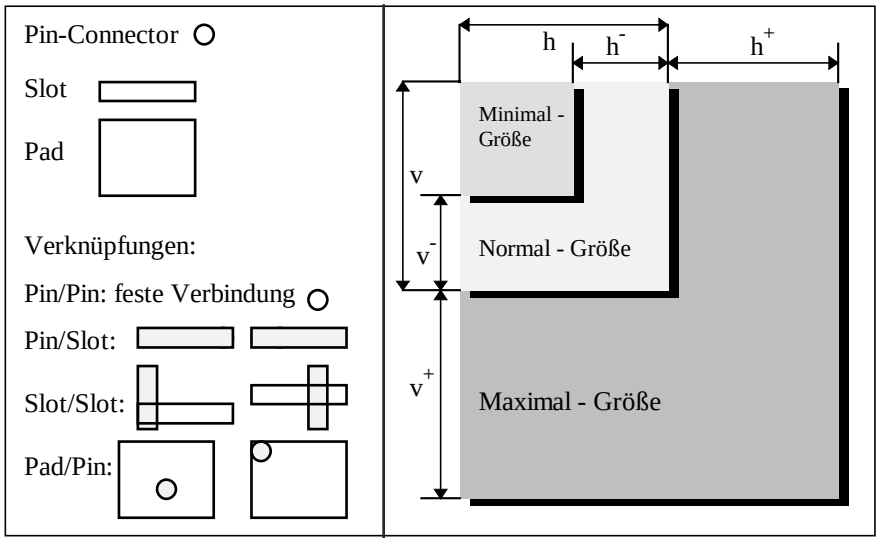


Abbildung 3 - 3: a) Connector; b) Stretchability: stretch limits v^- / v^+, h^- / h^+

Neben der Darstellung am Bildschirm durch „graphical views“ können zu jedem component subject beliebig viele „*external views*“ bestehen, die z.B. Postscript generieren oder aber auch eine Liste der im Schaltplan verwendeten Elemente in einer für andere Werkzeuge geeigneten, nicht graphischen Form (External Representation).

Unidraw ist im Gegensatz zu technischen Basis Interviews nicht als allgemeines Application-Framework (Anwendungsumgebung) ausgelegt, sondern spezifisch für den Anwendungstyp grafischer Editoren konstruiert. Die Gemeinsamkeiten derartiger Anwendungen wurden abstrahiert und im Framework bereitgestellt. Die Konstruktion grafischer Editoren wird in einem möglichst weiten Anwendungsgebiet unterstützt. Größere Applikationen können für andere Teile ihrer Funktionalität direkt auf die Möglichkeiten von Interviews zugreifen. [Vlissides95]:

> *„Unidraw is designed to provide only the graphical portions of a larger application. Thus, in addition to providing the usual graphical editing functions, it must smoothly integrate with other parts of the application.“*

3.2.3 Plattformunabhängigkeit

Interviews erleichtert den Entwurf grafischer Benutzungsoberflächen auf UNIX-Workstations mit X - Windows. Somit kann auch Unidraw (nur) in dieser Umgebung eingesetzt werden. Andererseits wurde MacApp ausschließlich für Macintosh Computer entwickelt, MFC für Microsoft Windows (vgl. 3.3.2).

Demgegenüber hat ET++ den Anspruch der Plattformunabhängigkeit in seiner Architektur umgesetzt (siehe 3.3.1). [Lewis95]: „*... this elegant framework incorporates a very strong model of interaction, and has ambitions to become a portable application development framework. You might think of ET++ as a second generation MacApp, but one that is evolving toward platform independence.*“

Inzwischen sind diverse kommerzielle Produkte als Basis für portable Anwendungen verfügbar, zum Beispiel wxWindows und StarView. Diese Frameworks bieten eine einheitliche Schnittstelle unter diversen Betriebs- und Fenstersystemen. Die Anwendung wird ohne Änderungen mit der entsprechenden Version bzw. auf dem geplanten Zielsystem übersetzt, die Umsetzung erfolgt vollständig innerhalb des Frameworks.

Voraussetzung sind ähnliche Darstellungs- und Interaktionsformen auf allen unterstützten Plattformen. Um bestehende Unterschiede zu überbrücken, gibt es verschiedene Möglichkeiten. Einfach ist die Beschränkung auf die Schnittmenge der auf allen Systemen verfügbaren „Widgets“, jedoch kann auch die Vereinigungsmenge durch zusätzliche Implementation der bisher nicht vorhandenen Elemente erreicht werden.

StarView wurde ausgehend vom Fenstersystem MS-Windows entwickelt und hat dort vorhandenen Widgets in andere Umgebungen übertragen. ET++ verwendet nicht die vorhandenen Toolkits als Basis sondern implementiert eigene Widgets. Damit kann auf demselben Rechner zur Laufzeit wahlweise Macintosh look&feel oder Motif simuliert werden. Dies zeigt aber zugleich auch ein wesentliches Problem plattformunabhängiger Frameworks auf: sollen die Anwendungen sich auf jeder Plattform gleich präsentieren oder muß der jeweilige Styleguide eingehalten werden?

Basierend auf der WAM-Methode ergibt sich ein anderer Lösungsansatz dieser Problematik. Nach WAM entwickelte Anwendungen machen grundsätzlich eine Trennung zwischen Funktionalität und Interaktion, vergleichbar MVC. Somit ist im Falle eines Wechsels des Fenstersystems ausschließlich die Interaktionskomponente zu ändern, wesentliche Teile der Applikation bleiben unberührt (Funktionskomponente, Aspekte und Material).

Verwendet nun die Interaktionskomponente zur Präsentation und Interaktion nur elementare Interaktionsformen, z.B. 1-from-N-Select, Activator, ValueChanger, ohne Annahmen über deren Darstellung durch Listen, ComboBoxen, Knöpfe oder beliebige zukünftig entwickelte Widgets zu machen, so kann auch diese Interaktionskomponente unverändert übernommen werden. Näheres zu diesen Ideen folgt in Abschnitt 4.2 (vgl. [Görtz97]).

3.3 Dokumentenbasierte Editoren

Mit Unidraw wurde in 3.2.2 ein Domänen-Framework für grafische Editoren vorgestellt. Nachfolgend gehe ich kurz auf die Gruppe von Application-Frameworks ein, die zur Erstellung dokumentenbasierter Editoren eingesetzt werden können. Da eine sehr große Anzahl Anwendungen grundsätzlich dieser Gruppe zuzuordnen ist, wurden sehr früh Abstraktionen und Konzepte in Frameworks realisiert. Erste Ansätze sind bereits bei MacApp zu finden, ET++ stellt eine wesentliche Weiterentwicklung dar. Relativ neu ist demgegenüber MFC, wobei [Lewis95] allerdings MFC noch weitgehend als eine Klassenbibliothek mit einigen wenigen Framework - Fähigkeiten sieht. Aber dies könnte sich mit der weiteren Entwicklung ändern: *„...and over time, [MFC] will evolve into a powerful framework.“*

Auch für ET++ und MFC gilt das allgemeine Leitbild direkter Manipulation. Darüber hinaus liegt eine dokumentzentrierte Sichtweise vor. Anwendungen sind dafür gedacht, jeweils ein oder mehrere Dokumente anzusehen oder zu ändern. Im Gegensatz zur Werkzeug-Material Metapher sieht der Anwender direkt auf das Material (Dokument) und kann dieses ohne ein für ihn erkennbares Werkzeug verändern.

3.3.1 ET++

> *„ET++ is a portable and homogenous object-oriented class library integrating user interface building blocks, basic data structures, and high level application framework components. ET++ eases the building of highly interactive applications with consistent user interfaces following the direct manipulation principle.“* [WG94]

Ausgangspunkt für ET++ war eine Architektur ähnlich MacApp, entwickelt hat sich eine plattformunabhängige (3.2.3), völlig neue Architektur [WG94]. Wie bereits für MacApp und Interviews motivieren auch die Autoren von ET++ die Entwicklung ihres Frameworks mit dem Aufwand, GUI - Benutzungsschnittstellen zu entwickeln. Besondere Vereinfachungsmöglichkeiten ergeben sich durch Redundanz infolge gleichförmiger Schnittstellen in verschiedenen Applikationen. Der Toolbox - Ansatz stellt einzelne Widgets zur Verfügung, nicht aber eine komplette Struktur für die Anwendung.

ET++ ist mehr als ein Mikro-Framework für graphische Benutzungsschnittstellen. Wie bereits MacApp verfügt ET++ über die Klassen „*Application*“ und „*Document*“. Unterstützt wird die Bearbeitung von beliebigen Dokumenten in mehreren Fenstern. Die Konsistenz mehrerer Sichten auf dasselbe Dokument wird über einen Mechanismus ähnlich MVC gewährleistet. Gemäß 2.1.3 ist ET++ eine Anwendungsumgebung. Die Unterstützung dokumentbasierter Editoren wurde von den Autoren gewählt, gerade weil ein sehr große Zahl unterschiedlicher Anwendungen so anzusehen sind. Die spezifische Ausrichtung eines Domänen-Framework liegt nicht vor.

Im Unterschied zu Interviews enthält ET++ als Application-Framework neben einfachen Elementen zur Programmierung der graphischen Benutzungsoberfläche vor allem auch die Klassen Application und Document. [Pree94] betont, daß erst die Kombination dieser Elemente Erscheinung und Umgang mit den so erstellten Anwendungen vereinheitlicht.

> *„The application framework ET++ provides elementary user interface building blocks (for example, buttons and menus), basic data structures (for example, the classes ObjList, ObjArray) and high-level application components such as the classes Application, Document, View, Command, and Window. Together with the elementary building blocks the high-level application components predefine as far as possible the look and feel of ET++ applications."* [Pree94]

Abgesehen von einzelnen Klassen, die wie einfache Typen der Sprache behandelt werden (z.B. String, Point), leiten sich alle ET++Klassen von Object ab. [WG95] sprechen daher von „*mostly single rooted*". So ist es möglich, in dieser Klasse eine umfangreiche Infrastruktur bereitzustellen. Vor allem ist dies ein Mechanismus zum Aktivieren/Passivieren von Objekten, das heißt die Transformation des Objektes in einen Stream. Hiermit werden neben permanenter Speicherung auch tiefe Kopien (DeepCopy), sowie die Übertragung in ein Clipboard und von da aus in andere Anwendungen realisiert. Mittels „*Dynamic loading and linking*" können sogar Objekte übertragen werden, deren Klassen noch gar nicht bekannt waren (vgl. [WG95]).

Obwohl ET++ zur Laufzeit ausreichend Laufzeitinformationen in Form eines über den C++ Standard weit hinausgehenden, per Makro realisierten RTTI (Run Time Type Information) zur Verfügung stehen, um die Passivierung automatisch durchzuführen, müssen die Methoden „*ReadFrom*" und „*PrintOn*" doch „vor Hand" implementiert werden. Der Entwickler einer Klasse soll entscheiden, welche Attribute gespeichert werden müssen und welche jederzeit durch einfache Berechnungen wieder hergestellt werden können.

Bereits in *Object* wurde der Benachrichtigungsmechanismus (*change notification*) implementiert. Somit kann dieser nicht nur benutzt werden, um verschiedene Fenster (also Sichten) auf ein Dokument zu aktualisieren, sondern zwischen beliebigen Objekten. Ein von den Autoren genanntes Beispiel sind verbundene Grafikelemente in der Anwendung ET++Draw [WG94].

ET++ unterstützt durch *Commands* Undo und Redo über beliebig viele Ebenen. Dieses Mikro-Framework ist Bestandteil der Architektur und prägt die Struktur der Applikationen. Zu der Infrastruktur der Anwendungsentwicklung gehört die Behälterklassen-Bibliothek. Es handelt sich um Bausteinklassen (vgl. 2.1.1), wobei je Behälterklasse eine passende Iteratorklasse existiert. Die Besonderheit dieser Behälterbibliothek sind robuste Iteratoren, die auch bei Änderungen am Behälter gültig bleiben (dazu werden z.B. Elemente verzögert gelöscht, also erst dann, wenn kein Iterator mehr auf dieses Objekt verweist).

Die Entwicklung der Benutzungsschnittstelle basiert auf der „leichtgewichtigen“ Oberklasse VObject und deren Unterklassen. *Clipper,* eine der von VObject abgeleiteten Klassen, verfügt über ein eigenes Koordinatensystem, die Komposition von Elementen (z.B. Ergänzung eines Rahmens oder zweier Scrollbars) folgt dem „Composite“ Design Pattern (vgl. [GoF95]). ET++ verwendet eine deskriptive Layoutbeschreibung statt fixierter Positionen. „*ET++ applications take care of windows (moving, resizing, activation on clicking, and so on) and their contents (for example scrolling)*“ [Pree94]. Müssen Teile der Benutzungsschnittstelle neu gezeichnet werden, so ist dies Aufgabe des Frameworks, die Anwendung wird nicht selbst neu zeichnen (*„no direct call to draw within application“* [WG94]). *Double buffering* gewährleistet eine flimmerfreie Anzeige.

Zum ET++ Application-Framework gehören auch verschiedene Mikro-Frameworks, z.B. für Textformatierung (RichText) und Formatkonvertierungen (z.B. Umwandlung zwischen TIFF, PICT und anderen). Die mit und für ET++ erstellte Entwicklungsumgebung umfaßt neben Klassen-Browsern und Editoren auch Werkzeuge, um zur Laufzeit Objektstrukturen und die Objekte selbst zu inspizieren. Zu den bekanntesten mit ET++ entwickelten Anwendungen dürfte die Entwicklungsumgebung Sniff gehören.

Als Problem der umfangreichen Funktionalität eines Frameworks wie ET++ nennen Weinand und Gamma [WG95] dessen Komplexität. Es wurde daher versucht, möglichst gleichförmige Mechanismen zu verwenden, wo immer dies möglich war (vgl. dazu Abschnitt 4.3). Ein „Nebenprodukt“ ist die Entwicklung von Design Pattern*: „Our focus on unifying mechanism enabled us to discover some recurring object-oriented design structures. This was a starting point for working on design patterns“*

Die Anwendungsumgebung ET++ kann zu großen Teilen im Sinne des Black-Box - Ansatzes verwendet werden. Abstrakte Parameterklassen erleichtern dies an verschiedenen Stellen und realisieren gleichzeitig Flexibilität zur Laufzeit. Der starke Bezug zu Design Pattern erleichtert die Identifikation der Hot Spots im Framework und ermöglicht ein Verständnis der internen Strukturen. Damit ist auch die Grundlage gegeben für White-Box - Anpassungen, falls die gegebene Funktionalität und Flexibilität im konkreten Fall nicht ausreichen sollte.

Innerhalb des komplexen Application-Frameworks lassen sich verschiedenartige Einzelkomponenten identifizieren. Teilweise handelt es sich um Mikro-Frameworks (z.B. Commands), die Grundlage der Infrastruktur sind, teils sind es Bausteinbibliotheken (z.B. Behälterklassen). Außerdem gibt es optional einsetzbare Mikro-Frameworks mit engerem Bezug zu konkreten Anwendungsgebieten. Das Framework für Textformatierung läßt sich insofern als Domänen-Framework als Erweiterung der allgemeineren Anwendungsumgebung ET++ verstehen. Zu bemerken ist das Angebot an konkreten Parameterklassen, die für bestimmte Anwendungen Implementationsarbeit durch Konfiguration vorhandener Klassen ersetzen. Ähnliches gilt für die o.g. Formatkonvertierung, verschiedene Parameterklassen für verbreitete Formate können direkt übernommen werden, speziellere sind noch zu implementieren.

3.3.2 Werkzeuge zur einfacheren Verwendung von Frameworks

Auch MFC (Microsoft Foundation Classes) ist ein Application-Framework für „Dokumenten-basierte Anwendungen“. Als Vorbilder dienten MacApp und ET++ [Pree95]. Zu jedem Dokument können mehrere gleichartige Fenster geöffnet werden, bewegen und vergrößern von Fenstern übernimmt das Framework. Wie auch in ET++ wird nicht direkt neu gezeichnet, sondern eine Region für ungültig erklärt und automatisch durch das Framework erneuert. Standarddialoge für Öffnen, Schließen, Speichern und Drucken sind in integriert.

Während der Ansatz einer gemeinsamen Oberklasse *Object* von ET++ übernommen wurde, fehlen *Commands* (Undo) in MFC ebenso wie direkte Manipulation. Die Unterstützung für vergrößern (zooming), rollen (scrolling) und Teilen (splitting) ist nach [Pree95] nicht *„state of the art“*, die rudimentären Metainformationen entsprechen etwa dem C++ Standard RTTI. Benutzereingaben (event handling) bearbeitet das zugrunde liegende Software Development Kit (SDK), eine objektorientierte Kapselung fehlt.

Zwei Arten dokumentbasierter Anwendungen sind möglich, MFC beinhaltet zwei insofern unterschiedliche Anwendungsumgebungen. Entweder wird jeweils genau ein Dokument betrachtet, oder es können beliebig viele Dokumente desselben Typs gleichzeitig geöffnet sein. Zu einem Dokument zeigt MFC ggf. mehrere Sichten konsistent an.

MFC wird hier nicht wegen seiner Framework - Architektur erwähnt, die gegenüber ET++ und anderen Ansätzen deutliche Defizite und wenig Neuerungen bringt. Die Besonderheit liegt vielmehr in der mitgelieferten Entwicklungsumgebung. Neben Editoren für den Quelltext, Compiler und Klassenbrowser gibt es speziell auf das Framework zugeschnittene Werkzeuge: der AppWizard erstellt und verknüpft alle für die Anwendung notwendigen Klassen, die von Basisklassen des Frameworks abgeleitet werden. Weitere Erleichterungen bringt ClassWizard und ResourceEditor (InterfaceBuilder), wobei letzterer auch in anderen Systemen üblich ist.

Diese Werkzeuge ermöglichen Anfängern, schnell eine Anwendung zu erstellen, ohne dafür große Teile der Framework-Architektur verstehen zu müssen. Damit wird ein großes Problem der Framework - Technik entschärft und die Black-Box - Wiederverwendung unterstützt. Allerdings beschränken sich die Werkzeuge auf diesen schnellen Einstieg. Die fehlende Dokumentation über interne Strukturen des MFC- Frameworks erschwert die Verwendung für größere Applikationen. Weder das Leitbild ist konkret beschrieben, noch werden Hot Spots z.B. durch Design Pattern dokumentiert.

Aufgrund des relativ geringen Leistungsumfanges des MFC- Frameworks sind Anpassungen und Erweiterungen häufig erforderlich. Es handelt sich um White-Box - Wiederverwendung, detaillierte Kenntnisse der Struktur sind notwendig. Grundsätzlich kann dieses Problem nicht durch die Erweiterung des Frameworks oder dessen Werkzeuge gelöst werden, sondern nur durch geeignete Dokumentation. Diese ist auch erforderlich, um überhaupt die Eignung von MFC für eine bestimmte Anwendung zu prüfen.

3.4 Kombination kleiner Frameworks - Taligent

Taligents Entwicklung CommonPoint bezeichnet [Andert95] als eine große Sammlung von Frameworks, auf deren Basis portable Systeme entwickelt werden können. *„CommonPoint uses frameworks at both the lowest levels in the system and the highest levels.“* Dabei stellt jedes Framework einen Dienst für Frameworks auf den höheren Ebenen zur Verfügung. Taligent beschränkt sich nicht auf die Unterstützung der Anwendungsentwicklung, sondern versucht gleichzeitig die Ebene des Betriebsystems mit abzudecken.

Nachfolgend wird CommonPoint nicht hinsichtlich seiner Eigenschaften als Application-Framework betrachtet, sondern die Komposition einzelner Frameworks in der Vordergrund gestellt. [Lewis95] betont die Aufteilung von Anwendungen in viele kleine Frameworks, die jeweils leicht zu erstellen sind und dabei die Portabilität des kompletten Systems nutzen:

> *„Taligent … is trying to rejuvenate the software industry by changing the rules. Instead of writing large monolithic applications which are tied to one GUI platform, the Taligent strategy is to make it not only possible, but economically attractive to write small, highly leveraged applets (small applications) that are easily ported to many platforms.“*

Die Kooperation zwischen Frameworks kann auf zwei Arten geschehen. Einerseits kann ein Framework ein anderes über dessen öffentliche Schnittstelle benutzen, also die abstrakt angebotenen Dienste des Klassenteams verwenden. Gewissermaßen wird in diesem Fall das gesamte Framework als ein Baustein betrachtet, der von außen erzeugt wird und der danach Aufträge ausführt. Die Umkehrung des Kontrollflusses findet dabei nur innerhalb der Framework-Komponenten statt, nicht aber bzgl. des Benutzers (vgl. Abbildung 3-4).

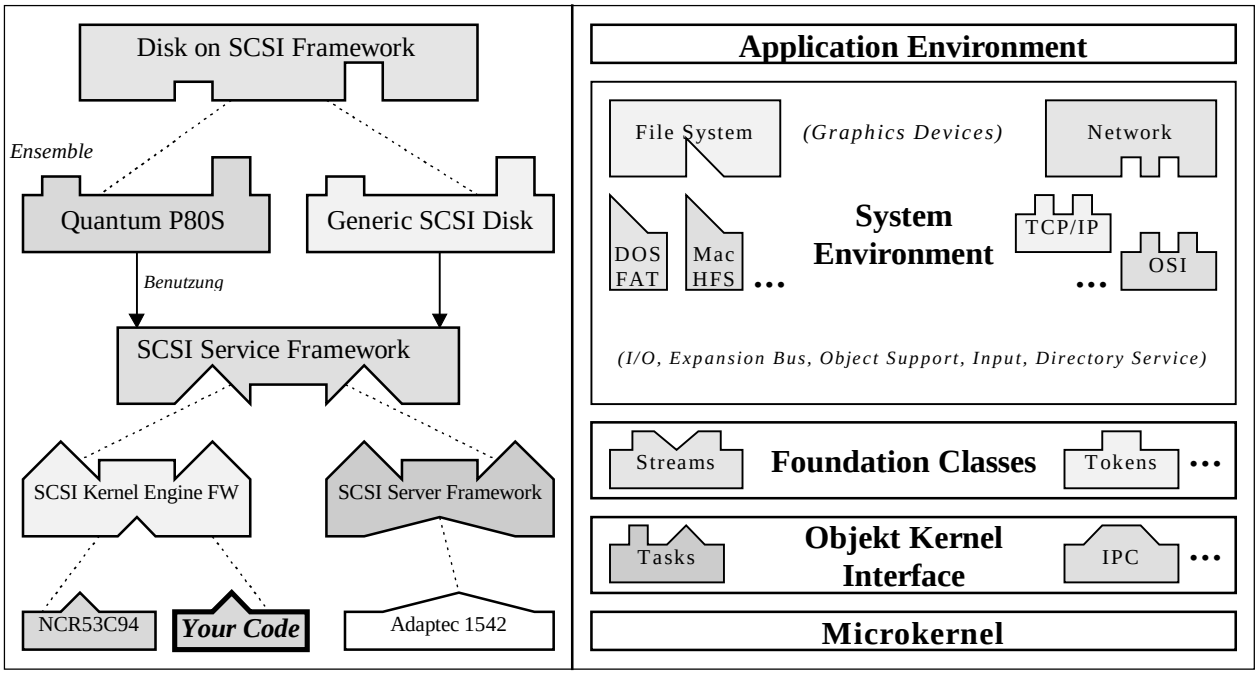


Abbildung 3 - 4: a) Kooperation von Taligent Frameworks; b) aus Frameworks konstruiertes Betriebssystem

Andererseits sind Frameworks keine kompletten Lösungen, sondern müssen durch spezifische Komponenten ergänzt werden. Dies bedeutet die Ableitung eigener, konkreter Klassen von meist abstrakten Frameworkklassen. Hier wird das in der Oberklasse spezifizierte Protokoll implementiert, damit das Framework die Dienste seiner öffentlichen Schnittstelle erfüllen kann. [Andert95] bezeichnet die Menge dieser Klassen, die konkrete Funktionalität für ein Framework zur Verfügung stellen, als „*ensemble*“ und setzt dabei implizit die Verwendung von Parameterklassen voraus.

> *„An ensemble is a set of one or more classes that are derived from the framework and provide concrete implementations for pure virtual methods declared by the framework, Together with the framework, ensemble is a complete concrete implementation of the service offered by the framework's public interface.“*

Letztere Art der Framework - Benutzung entspricht der üblichen Sichtweise auf Application-Frameworks. Anwendungen benutzen das Framework nicht „von außen“ als einen Baustein, sondern konkretisieren die vorhandene generische Anwendung durch Implementation nicht standardisierter Hot Spots.

Im Zusammenhang mit der Kombination verschiedener Frameworks unterschiedlicher Ebenen erhält der erste Punkt jedoch größere Bedeutung, also die Benutzung eines Mikro-Frameworks tieferer Ebene durch ein anderes Framework, wobei einfach auf die Dienste der öffentlichen Schnittstelle zugegriffen wird. Hinsichtlich der Unterscheidung zwischen Bausteinklassen und Frameworks wird hier die Sichtweise entscheidend: die Definition von Bausteinklassen (siehe 2.1.1) stellt auf deren Verwendung von außen ab, intern durchaus ein komplettes Framework hinter der als Baustein verstandenen Interface - Klasse stehen. Ebenso lassen sich Frameworks zu neuen, größeren Frameworks kombinieren, wie dies schon Mikro-Frameworks innerhalb der ET++ Anwendungsumgebung gezeigt haben (z.B. die Commands).

CommonPoint basiert auf dieser Möglichkeit. Mikro-Frameworks bilden eine Vielzahl aufeinander aufbauender Schichten. Zudem kann auch ein fertig implementiertes Framework als Grundlage für ein weiteres, spezielleres Framework dienen, indem dessen Klassen von dem allgemeineren abgeleitet werden:

> *„CommonPoint frameworks are composable. They can be layered, one on top of another. And a framework itself can be a specialization of a more abstract framework.“ [Andert95]*

Das Taligent System besteht aus vielen zusammenarbeitenden Frameworks. Diese Architektur und die einzelnen Frameworks sind in großem Maße wiederverwendbar, z.B. das SCSI-Framework innerhalb der CommonPoint I/O- Architektur [Andert95]. Probleme ergeben sich jedoch aus der mangelnden Übersichtlichkeit eines derartigen „Geflechtes“.

Frameworks müssen korrekt instantiiert werden, der Klient gibt dazu die Ensembles vor, oder das Frameworks erzeugt diese automatisch - z.B. besteht jeweils ein „Default“. Erzeugung spezieller Ensembles innerhalb des Frameworks ohne Einwirkung des Klienten setzt voraus, daß weitere Informationen vorliegen. Technisch helfen z.B. Fabriken oder späte Erzeugung (vgl. [BR96]). Das CommonPoint I/O- Framework installiert für erkannte Plug&Play-Hardware automatisch die richtigen Treiber. Soweit manuelle Konfiguration wegen fehlender Hardware - Erkennung erforderlich wird, steht ein Configuration Recorder zur Verfügung, der die Konfiguration mittels direkter Manipulation ermöglicht. Diese Informationen speichert das Framework (bzw. das integrierte Werkzeug) persistent, so daß nach einmaliger manueller Konfiguration später dann dieser Vorgang automatisch innerhalb des Frameworks abläuft.

Die automatische Instanziierung der Frameworks basiert auf den benötigten Protokollen unterliegender Schichten. [Andert95] spricht von einem „*stack of services*“. Speziell mit der I/O- Architektur wird es möglich, Gerätetreiber unabhängig vom jeweiligen Betriebsystem zu entwickeln. Es werden jeweils große Teile des Frameworks wiederverwendet, somit sind neue Fehler meist auf die überschaubaren neuen Komponenten beschränkt.

Taligent setzt auf die extensive Benutzung vieler, jeweils kleiner Frameworks. Mit dieser Architektur werden Komponenten realisierbar, deren vollständige Plattformunabhängigkeit durch das Framework gewährleistet ist. Anpassungen einzelner Komponenten sind leicht zu implementieren, oft stellt sich jedoch die Frage, welche Komponenten überhaupt zuständig ist. Zur Beantwortung kommt es wesentlich auf die Struktur der kompletten Sammlung an.

3.5 Zusammenfassung

Die vorgestellten Frameworks reichen von GUI - Frameworks (Interviews) für X - Windows bis zu Ansätzen plattformunabhängiger Benutzungsoberflächen. Mit MFC und ET++ finden sich Anwendungsumgebungen mit dem Leitbild dokumentenbasierter Applikationen, Unidraw ist ein Domänen-Framework für graphische Editoren. ET++ zeigt die Bereitstellung mehrerer Teil- Frameworks innerhalb eines großen Application-Frameworks, Taligent versucht sich an der kompletten Aufteilung der Software-Systeme in kleine Frameworks - angefangen auf der Ebene von Betriebsystemen bis hinauf zu Anwendungsumgebungen.

In dieses breite Spektrum werde ich im nächsten Kapitel das WAM-Framework einordnen und dem Leser so einen Anhaltspunkt für dessen Bewertung geben. ET++ und besonders Taligent zeigen zudem die Tendenz auf, mehrere Frameworks miteinander zu kombinieren, woraus sich Fragen der Strukturierung ergeben, die ich im 5. Kapitel anspreche.

Merkmale weiter entwickelter Frameworks hinsichtlich Funktionalität sind Application-Frameworks, angefangen mit allgemeinen Anwendungsumgebungen bis hin zu sehr speziellen Domänen-Frameworks. Je weiter ein Gebiet standardisiert werden kann, desto größer ist der Nutzen für Anwendungen, die auf dem Framework basieren. Anzustreben ist z.B. der Bezug auf spezielle Branchen oder Firmen.

Flexibilität und Portabilität sind weitere Qualitätskriterien für Frameworks. Abhängigkeiten der Anwendungen vom Fenstersystem sind zu vermeiden, indem diese in den Komponenten des Frameworks gekapselt werden. So muß die Portierung auf ein neues System nur noch einmal erfolgen, nicht für jede einzelne Anwendung.

Design Pattern sind das Mittel der Wahl, um die Hot Spots eines Frameworks flexibel zu gestalten. Technisch lassen sich diese Mechanismen gewissermaßen als Erweiterungen der Sprache in einer allgemeinen Oberklasse bereitstellen (vgl. ET++). Die Probleme des daraus resultierenden Klassen- Baumes sind zu vermeiden, indem jeweils einzelne, möglichst voneinander unabhängige Mikro-Frameworks explizit je einen Mechanismus bereitstellen. Vorteil ist dabei auch die ausdrückliche, besser nachvollziehbare Benutzung.

Funktionalität und Flexibilität gleichermaßen zeigen sich in optionalen Komponenten, jeweils als eigenständige Mikro-Frameworks für verschiedene Zusammenhänge implementiert. Ein Beispiel ist die Textformatierung in ET++, die WAM-Bibliotheken im folgenden Kapitel zeigen weitere Möglichkeiten.

Für die Benutzung eines Frameworks vorteilhaft sind Leitmetaphern, die es dem Anwendungs- Entwickler erleichtern, Intention und Einsatzkontext des Frameworks zu verstehen. Üblich ist die direkte Manipulation als Handhabungsform für Oberflächenelemente, mit Ausnahme von Unidraw verfolgen alle hier vorgestellten Application-Frameworks einen Dokumentenzentrierten Ansatz. Die Werkzeug und Material Metapher, deren Leitbild und softwaretechnischen Konzepte ich in Abschnitt 2.3 vorgestellt habe, bietet dem Entwickler eine Methode für den Entwurf und die Konstruktion, spezifische Frameworks helfen bei der technischen Umsetzung.

Die Frameworks in diesem Kapitel folgen keiner bestimmten Methode, können daraus resultierende Vorteile nicht nutzen. Softwaretechnisch wichtig ist die Identifikation von Hot Spots, deren flexible Implementation in Form dokumentierter Design Pattern und die Zerlegung einer großen Bibliothek in überschaubare, lose gekoppelte Komponenten.

Black-Box - Wiederverwendung ist in allen standardisierbaren Bereichen anzustreben. Ein Mittel sind Parameterklassen, die gut nachvollziehbare Abstraktionen objektifizieren, konkrete Parameterklassen für übliche Anwendungsfälle sind ein weiterer Fortschritt. Beispiele gibt es in ET++ z.B. bei der Formatkonvertierung (fertige Klassen für GIF u.a.). Soll die Flexibilität zur Laufzeit bestehen, so sind Parameterklassen zwingend erforderlich.

Baumstrukturen mit einer gemeinsamen Oberklasse sollten hinsichtlich möglicher Kombination mit anderen Frameworks oder Bibliotheken vermieden werden. Objektorientierte Mechanismen lassen sich als kleine Mikro-Frameworks bereitstellen (Implementation von Design Pattern), und machen dann die erreichte Flexibilität nach außen deutlich. Polymorphe Behälter können durch generische ersetzt werden, in C++ sind dies Templates.

Inwieweit die BibV30 diesen Ansprüchen genügt, ist Thema des folgenden Kapitels.

Kapitel 4 Frameworks für WAM: C++ Bibliotheken am Arbeitsbereich Softwaretechnik

Am Arbeitsbereich Softwaretechnik werden Frameworks entwickelt und verwendet, um die Anwendungsentwicklung nach WAM zu unterstützen. Motivation dieser Arbeit sind hieraus gewonnene praktische Erfahrungen mit objektorientierten Klassenbibliotheken hinsichtlich deren Probleme und Nutzen. Vor diesem Hintergrund beschreiben die folgenden Abschnitte Rahmenbedingungen und Entstehungsprozeß des heutigen WAM-Frameworks „**BibV30**“ (4.1), stellen entsprechend Kapitel 3 den Bezug zu anderen Produkten her (4.2) und bewerten die geleistete konzeptionelle und technische Weiterentwicklungen unter Berücksichtigung der in Kapitel 2 dieser Arbeit aufgestellten Kriterien (4.3).

Ein eigener Abschnitt (4.4) ist der neu entwickelten Unterstützung graphischer Applikationen gewidmet. Es handelt sich um eine Kombination von Konzepten aus früheren Versionen des Frameworks mit jenen, die [Vlissides90] im Unidraw - Framework realisiert hat. Der Schwerpunkt liegt auf der für graphische Editoren wichtigen direkten Manipulation, die im Kontext von WAM-Frameworks bisher nicht unterstützt wurde. Bei der Übernahme des in Unidraw tragfähigen Manipulator - Konzeptes mußte ich dessen Verträglichkeit mit den Metaphern der WAM-Methode prüfen, bzw. eine geeignete Anpassung des Konzeptes vornehmen.

4.1 Entstehung und Rahmenbedingungen

Ausgehend von konkreten Anwendungen (im Rahmen einer Studien- bzw. Diplomarbeit) entwickelten zwei Studenten 1992/93 die erste Version der heutigen Bibliothek. Anlaß war die fehlende technische Basis oder auch Infrastruktur für die Implementation von Software - Werkzeugen nach WAM. Als Hardwareplattform wurden Unix - Workstations gewählt, als Programmiersprache C++ und als graphische Oberfläche OSF/Motif. Bezüglich der Entscheidung für C++ sei verwiesen auf [Pree94]: „*despite all its deficiencies ... C++ has proved to be ‘good enough’ for programming in the large*“. Wichtig ist die Typisierung von C++, einige Probleme der hybriden Sprache C++ lassen sich durch Programmierkonventionen ([Taligent94], [RWW95]) umgehen.

Kern dieser ersten Entwicklung war die objektorientierte Kapselung der OSF/Motif - Widgets in Form von Interaktionstypen (IAT) gemäß WAM. Für Kopplung zwischen IAT und IAK (Interaktionskomponente, Abschnitt 2.3) diente ein einfacher Beobachtungsmechanismus, der zugleich auch für die Verbindung zwischen IAK und FK benutzt wurde, sowie zwischen Funktionskomponenten (Kontext / Subwerkzeug). Das FIAK-Framework als zweiter Teil der Bibliotheken übernahm bereits in erster Version die Konstruktion von Kontext - Werkzeugen mit FK und IAK unter Verwendung von Subwerkzeugen (vgl. [Riehle93] und [Riehle93a]). Beobachtung, Interaktionstypen und FIAK-Framework gemeinsam bildeten die erste Anwendungsumgebung nach WAM am Arbeitsbereich Softwaretechnik.

Zu einem sehr frühen Zeitpunkt der Entwicklung begann der Einsatz in einer Lehrveranstaltung. Aus diesem Projekt ergaben sich verschiedene Studien- und Diplomarbeiten, infolgedessen erhielten die Framework-Entwickler umfassendes Feedback zu ihrer Arbeit.

Allerdings führten sowohl Mängel der halbfertigen Version als auch dadurch erforderliche Korrekturen, Erweiterungen und Umstrukturierung der Klassenschnittstellen zu erheblichen Schwierigkeiten in der Anfangsphase. Gerade die direkte Rückkopplung bewirkte jedoch frühzeitig eine stabile Bibliotheksversion, die dann festgeschrieben wurde.

In der Folgezeit fand eine von der benutzten Version entkoppelte Erweiterung besonders im Bereich des Application-Frameworks (FIAK) statt, neue Komponenten wurden hinzugefügt. Außerdem wurde mit der ConLib eine unabhängige Behälterklassen-Bibliothek [Traub95] entwickelt. Die neuen Entwicklungsstände wurden jeweils nach Fertigstellung neben weiter bestehenden Vorversionen freigegeben, für den Umstieg geeignete Anleitungen verfaßt.

Nach einer Phase geringer Nutzung der Frameworks wurde Mitte 1995 eine Restrukturierung eingeleitet. Es hatte sich gezeigt, daß in einigen Bereichen eine starke Ausrichtung auf den Bedarf der ersten entwickelten Anwendungen vorlag. Bessere Abstraktionen, verschiedene Erweiterungen und Integration neuer, bereits außerhalb der Bibliothek implementierter Konzepte sind angestrebte Ziele, vor allem mußte aber die bereits vorhandene Komplexität untersucht werden. Aufgrund fehlender Übersichtlichkeit entstand ein weiteres, stark vereinfachtes Framework für den Lehrbetrieb, dessen Integration erwünscht ist.

Wichtiges Merkmal dieser neuen Entwicklungsphase ist die Bildung einer Gruppe zur Koordination der Arbeiten am Framework. Während der Startphase befaßten sich lediglich zwei Personen mit dem Projekt, die sich direkt koordinieren konnten. Später haben dann vorwiegend einzelne Studenten jeweils ihre Schwerpunkte bearbeitet, die Diskussion von Design - Entscheidungen in größerem Rahmen fand kaum statt, Richtung und Ziel fehlten.

Frameworks für Lehre und Forschung

Framework-Entwicklung reduziert durch Wiederverwendung und standardisierte, robuste Komponenten die Kosten der Software - Produktion. Das hier betrachtete WAM-Framework entstand an der Universität und wird (bisher) nicht außerhalb des Lehrbetriebes eingesetzt. Offensichtlich kann dieses Framework nicht mit üblichen Maßstäben kommerzieller Produkte gemessen werden.

Der Nutzen dieser aufwendigen Entwicklung eines eigenen Frameworks besteht aus zwei Bereichen: in der Lehre sammeln Studenten praktische Erfahrungen mit der Framework - Technik und der Methode WAM, innerhalb des bestehenden Frameworks können neue Konzepte ausprobiert werden. Anderen Studenten dient das Framework einfach als Entwicklungsplattform, das Framework wird also im üblichen Sinne benutzt als Infrastruktur für die Anwendungsentwicklung.

Gegenüber Produkten fremder Anbieter besteht der Vorteil, daß Sourcecode des Frameworks zugänglich ist und direkter Kontakt zu den Entwicklern besteht. Nachteilig für die Benutzung im eigentlichen Sinne ist teilweise nicht professionelle Dokumentation, vor allem aber die mangelnde Konzentration auf das Framework als Produkt.

Hauptargument für die Entwicklung des Frameworks ist sicher der Entwicklungsprozeß selbst - direkt gefolgt von dem Wunsch, die Methode WAM praktisch umzusetzen. Überwiegend wurde das Framework von Studenten entwickelt, die neue Konzepte entwerfen und testen wollten. So ist auch zu erklären, daß in vielen Fällen der Nachweis der Tragfähigkeit dieser Ideen im Vordergrund steht, nicht das tatsächlich implementierte Produkt. Einige Konzepte wurden neben dem Framework, teils unter Verwendung dessen Komponenten realisiert, dann aber nicht in die Bibliothek integriert.

Fehlende Produktorientierung stellt damit einen Schwachpunkt der Framework-Entwicklung dar. Hinzu kommt fehlende Koordination der Aktivitäten diverser Studenten. Dieses Problem adressiert die jetzige Phase der Entwicklung durch Workshops (z.B. über Interaktionsformen) und Diskussion der Konzepte innerhalb von Lehrveranstaltungen. Ein umfangreicher Einsatz der WAM-Frameworks in größeren Projekten, bestehend aus Studien- und Diplomarbeiten, erfordert stärkere Produktorientierung bei der Weiterentwicklung des Frameworks insgesamt sowie einzelner neuer Bestandteile. Diese Arbeiten sind effektiv zu koordinieren.

Zu den derzeitigen Bemühungen gehört auch die konsistente, vollständige Dokumentation. Bereits vorhandene Beispiele als erprobter Einstieg in ein Framework werden erweitert und angepaßt, zusätzliche Werkzeuge für die Darstellung der Dynamik sind derzeit ein aktuelles Forschungsvorhaben (vgl. [LS96]). Das eigene Framework kann für diese Werkzeuge Testfeld aber vor allem auch technische Grundlage sein.

Auf der anderen Seite konnten die Entwickler des WAM-Frameworks ihre neuen Konzepte testen, und mußten nicht voreilige Implementationen als „praktikable“ Lösungen erstellen. Das wohlstrukturierte Konzept hatte immer Vorrang vor Kompromissen zugunsten schnell einsatzfähiger Produkte.

Am Arbeitsbereich Softwaretechnik ergeben sich im Zusammenhang mit der vertretenen Methode WAM weitere Argumente für eine eigene Entwicklung. Studenten hören nicht nur in Vorlesungen, was die Werkzeug-Material Metapher ist, sondern erleben deren praktische Anwendung. Dies wäre mit Fremdprodukten kaum zu leisten, die anderen Leitbildern folgen und nur schwer adaptierbar sind. Eben dieses Problem hätte sich auch für Studenten ergeben, die konzeptionelle Weiterentwicklung der Methode anstreben, dies aber innerhalb eines nicht hierfür geeigneten Umfeldes eines anderen Frameworks hätten demonstrieren müssen.

Über die Grenze des Arbeitsbereiches und der Universität hinaus bieten die WAM-Frameworks eine ausgezeichnete Möglichkeit, einen Eindruck in die Methode zu geben, die sonst nur in Artikeln und Büchern [GKZ93] beschrieben ist. Bestes Mittel hierzu ist eine gute Dokumentation und Veröffentlichung des Frameworks im World-Wide-Web.

4.2 Einordnung in die Framework - Landschaft

Vergleichend mit Kapitel 3 vermittelt dieser Abschnitt einen Eindruck der Leistungen des WAM-Frameworks. Hier jeweils nur kurz erwähnte Komponenten werden zum Teil in den Abschnitten 4.3 und 4.4 näher erläutert. Der Leser erhält durch diesen Abschnitt nur einen groben Überblick, zwecks Bewertung im Verhältnis zu anderen Produkten.

Die C++ Bibliotheken am Arbeitsbereich Softwaretechnik unterstützen die Konstruktion interaktiver Anwendungen nach der Werkzeug und Material Metapher. Einsatzbereich sind entsprechend dem WAM - Leitbild Einzelplatzsysteme für Büroanwendungen. Es wurden im Kontext der Methode verschiedene Kooperationsformen analysiert und Erweiterungen der ursprünglichen Metaphern für Einzelplätze entwickelt. (z.B. Prozeßmuster und Postkörbe gemäß [Wulf95] und [Gryczan95], aktuelle Arbeiten vgl. [KRW96]).

Teil der C++ Bibliotheken ist die hiervon unabhängig implementierte und auch mit anderen Frameworks verträgliche Behälterklassen-Bibliothek ConLib [Traub95]. Basierend auf dem Template Mechanismus lassen sich in diese Behälter Objekte beliebigen Typs einfügen. Von besonderem Wert ist neben der Klassifikation von Behälterkonzepten und deren effizienten Implementation vor allem das hochentwickelte Konzept stabiler Cursor.

Die Interaktionstypen - Bibliotheken IATMotif (für OSF/Motif) sowie IATTcl (für Tcl/TK) bilden die Verbindung zur graphischen Benutzungsoberfläche, ein in das Framework integrierter Interface - Builder erleichtert deren Gestaltung (derzeit nur für Motif). Durch die Interaktionstypen wird Plattformunabhängigkeit erreicht: nicht die Anwendung muß bzgl. eines neuen GUI portiert werden, sondern lediglich die IAT - Bibliothek. Interaktionstypen sind mit der Anwendung über das Command Pattern verbunden.

Werkzeugkonstruktion nach WAM ist Aufgabe des FIAK - Application-Frameworks. Die Oberklassen für Funktionskomponenten (FKBase) und Interaktionskomponenten (IAKBase) sichern passende Komponenten, die Integration von Subwerkzeugen kapselt das Framework gegenüber der Anwendung und verwendet dabei das Entwurfsmuster „*Late Creation*“ [BR96]. Voraussetzung ist ein per Präprozessor-Makros realisiertes Meta Objekt Protokoll (MOP), für das je Klasse eine einfache Zeile zu ergänzen ist.

Neben dieser sehr allgemeine Anwendungsumgebung bestehen zwei spezialisierte Varianten, deren Klassen jeweils von FKBase und IAKBase abgeleitet wurden. Einerseits helfen FKMain und IAKMain bei der Implementation von Anwendungen mit einer Standard - Menüzeile (Datei öffnen, schließen, drucken, ..., beenden; cut, copy paste), andererseits bietet der ToolCoordinator eine Sniff - artige Infrastruktur für Applikationen, die mehrere gleichartige Materialien in jeweils eigenen Fenstern anzeigen. Das erste Framework (FKMain / IAKMain) stellt z.B. vor dem „Beenden“ einer so erstellten Applikation die Sicherung evtl. geänderten Materials sicher. Der ToolCoordinator läßt dem Benutzer der Anwendung die Entscheidung, ein Fenster für neues Material „wiederzuverwenden“, oder immer neue Fenster zu erzeugen.

Vergleich mit anderen Frameworks

Große Ähnlichkeiten im Leistungsumfang bestehen zwischen dem WAM-Framework BibV30 und ET++. Beide Frameworks orientieren sich bei den bereitgestellten Basis - Mechanismen an Design Pattern (z.B. Observer, Command), verfügen über hoch entwickelte Behälter mit Iteratoren (bzw. Cursors) und eine Kapselung des Fenstersystems. Beide bauen hierauf eine Anwendungsumgebung mit klar definiertem Leitbild sowie weitere, optionale Komponenten.

Konzeptionell unterschiedlich sind die Arten der Wiederverwendung in beiden Frameworks. ET++ folgt dem Ansatz einer gemeinsamen Oberklasse (Baum) und benutzt diese Klasse auch für diverse Mechanismen, die so allen anderen Klassen **implizit** bereitgestellt werden. Der größte Nachteil dieser allgemeinen Oberklasse ist die weitgehende Unverträglichkeit mit fremden Bibliotheken (vgl. 2.1.4), die Integration diverser Mechanismen macht diese Klasse und alle anderen - also abgeleiteten - Klassen unübersichtlich, der Anwender ist gezwungen, einige Methoden immer zu implementieren (z.B. Activation).

Die BibV30 realisiert alle Mechanismen zur Gewährleistung flexibler Komponenten **explizit** als eigene Klassen bzw. Mikro-Frameworks. WAM als Methode gibt dazu eine Anleitung, in welchen Situationen einige dieser Mechanismen einzusetzen sind (vgl. Diskussion über die Beobachtungsmuster in Abschnitt 4.3). Dieses Prinzip wirkt sich auf Applikationen aus, die mit den Frameworks erstellt werden, auch hier werden alle Mechanismen explizit angegeben.

ET++ verwendet in weitaus größerem Maße Parameterklassen, als die BibV30. Dennoch streben beide Frameworks die Black-Box - Wiederverwendung an und ermöglichen durch geeignete Dokumentation gleichzeitig umfassende Anpassung nach dem White-Box - Ansatz. Gründe für verstärkten Einsatz von Parameterklassen in ET++ sind sowohl eine größere Zahl standardisierter Komponenten, als auch ein höherer Bedarf an Flexibilität zur Laufzeit.

Interviews ist kein Application-Framework, das Fenstersystem wird nicht portabel gekapselt. Eine Gemeinsamkeit mit ET++ liegt in der deskriptiven Anordnung der Oberflächenelemente anstelle absoluter Positionierung. WAM-Frameworks verfolgen einen grundsätzlich anderen Ansatz durch die Verwendung von Interaktionstypen (oder Interaktionsformen), die sämtliche Aspekte der Präsentation vor der Anwendung verbergen. Statt dessen werden diese außerhalb des C++ Programmes in Ressourcen beschrieben. Das Layout soll mit einem Prototyping - Werkzeug erstellt und unabhängig von der Applikation geändert werden können, im einzelnen wird dieser Punkt in 4.3.3 ausgeführt, ebenso die unterschiedliche Herangehensweise an den Anspruch einer portablen Lösung.

Ein Vergleich mit dem Domänen-Framework leistet 4.3.4, grundsätzlich ist die BibV30 breiter angelegt, also nicht auf die Domäne graphischer Editoren beschränkt - aber auch nicht auf diesen Anwendungstyp zugeschnitten. Gegenüber CommonPoint besteht eine sehr klare Abgrenzung, da systemnahe Ebenen nicht im WAM-Framework enthalten sind. Konzeptionell finden sich auch andere Beziehungen zwischen den einzelnen Frameworks in der BibV30, dies ist Thema im 5. Kapitel.

4.3 Entwicklung: Maßnahmen, Ziele und Ergebnisse

Restrukturierung und Erweiterung des WAM-Frameworks begann vor etwa eineinhalb Jahren im Jahren einer Lehrveranstaltung. Eine systematische Analyse des bestehende Produkt führte zur Diskussion der Schwachpunkte und Lösungsansätze.

Unter Berücksichtigung des Einsatzes in Lehrveranstaltungen streben die Entwickler ein leicht erlernbares Framework an. Die Verteilung der umfangreichen Funktionalität auf unabhängige und lose gekoppelte Komponenten trägt hierzu bei. Außerdem sind verschiedene prototypisch bereits implementierte Mikro-Frameworks in die BibV30 zu integrieren.

Zum jetzigen Zeitpunkt ist die Entwicklung noch nicht abgeschlossen. In diesem Abschnitt beschreibe ich vollzogene und auch geplante Änderungen und bewerte deren Auswirkungen auf das Framework als Ganzes hinsichtlich der angestrebten Ziele und in Bezug zu Kapitel 3. Mein Anteil an dieser Arbeit liegt einerseits in der gemeinschaftlichen Diskussion in diversen Gruppen, andererseits habe ich verschiedene Komponenten konzipiert bzw. implementiert. Die folgende Darstellung soll gerade auch eine Zusammenfassung dieser Arbeiten einer großen Gruppe leisten, meinen Anteil an konkreten Elementen gebe ich dabei mit an. Einzelheiten der Implementation sind in anderen Arbeiten oder der Dokumentation zur BibV30 nachzulesen, entsprechende Verweise gebe ich direkt bei der Darstellung der jeweiligen Komponenten.

Zur Orientierung in der Vielzahl einzelner Neuerungen, sind diese den drei Bereichen GUI - Anbindung (4.3.3), Application-Frameworks (4.3.1) und allgemeine Grundlagen zugeordnet, innerhalb dieser Bereiche sollen Unterpunkte den Überblick erleichtern. Die Auswertungen beziehen sich auf die Ergebnisse aus Kapitel 2 (vgl. 2.4), Lösungsdetails werden mit anderen Produkten (Kapitel 3) verglichen.

4.3.1 Erweiterung der Anwendungsumgebung

Als WAM-Framework berücksichtigt die BibV30 diverse Metaphern und Design Pattern, die Grundlage dieser Methode sind oder in deren Umfeld eingeführt wurden. Ausgehend von der allgemeinen Infrastruktur (z.B. MOP) und der Umsetzung grundlegender Metaphern (vgl. Umgebung) beschreiben die Unterpunkte verschiedene Application-Frameworks in der BibV30 (z.B. MainFIAK) und generische Subwerkzeuge.

Late Creation, MOP - und deren Kopplung

[BR96] beschreiben das Muster „späte Erzeugung“ und motivieren dessen Verwendung. Kern sind zwei Annahmen: Die erzeugende Komponente benötigt nur ein Objekt vom statischen Typ einer Oberklasse und der dynamische Typ richtet sich nach einer Spezifikation. Zwecks höherer Flexibilität soll kein Objekt einer konkreten Klasse mittels „New“ erzeugt werden, ggf. ist die Klasse selbst in der erzeugenden Komponente nicht einmal bekannt - sondern nur deren Oberklasse. Die Auswahl der konkreten Klasse erfolgt „spät“, also zur Laufzeit.

Im Kontext von WAM sind besonders zwei Einsatzfälle hervorzuheben, die zu einer Integration dieses Design Pattern in die BibV30 geführt haben. Werkzeuge bestehen grundsätzlich aus den getrennten Komponenten für Funktionalität und Interaktion, deren Objekte zur Laufzeit miteinander verbunden sind. Ein Kontext - Werkzeug oder die Umgebung (s.u.) erzeugt eine konkrete Funktionskomponente. Jede FK benötigt eine passende IAK, für deren Erzeuger ist aber nur wichtig, daß dies eine IAK ist (Oberklasse), Kenntnis der konkreten IAK wäre eine vermeidbare Komplikation des System und eine Einschränkung der Flexibilität.

Späte Erzeugung in der Anwendungsumgebung BaseFIAK löst dieses Problem für die BibV30. Die Kontext-FK erzeugt eine spezielle Sub-FK, die Kontext-IAK eine Sub-IAK. Dabei kennt die Kontext-IAK nur die allgemeine Oberklasse tIAKBase aller Interaktionskomponenten, als Spezifikation für die Erzeugung einer passenden IAK dient die FK selbst.

Die zweite Anwendungssituation im WAM-Framework ist die Zuordnung von Materialtyp und Werkzeug durch die Umgebung. Innerhalb der Umgebung (vgl. Unterpunkt „Aufgaben der Umgebung“) sind verfügbare Werkzeuge bekannt, Aspekte spezifizieren den Zusammenhang zwischen Materialklasse und Werkzeugklasse. Die Umgebung öffnet Werkzeuge passend zum Material, erzeugt die Werkzeug - Objekte „spät“.

Technisch basiert die späte Erzeugung im WAM-Framework auf dem hauptsächlich hierfür eingeführten Meta Objekt Protokoll. Zum Zeitpunkt der Entwicklung enthielt C++ noch keine Typinformationen zur Laufzeit, auch heute ist das RTTI gemäß Standard weitgehend in den Compilern nicht realisiert. Das MOP in der BibV30 entspricht den Lösungen in MFC und ET++ hinsichtlich der Implementation mittels C++ Makros.

Der Leistungsumfang des MOP geht über das zukünftige C++ RTTI nicht hinaus, damit ist später ein Verzicht auf das eigene Meta Objekt Protokoll denkbar. Dies ist auch bezüglich der MFC ähnlich zu bewerten, nur ET++ nimmt große Erweiterungen vor, die der C++ Standard nicht berücksichtigt. Für den Benutzer ergibt sich die Notwendigkeit, zusätzlich eine Makro - Zeile in seine Klasse aufzunehmen und die Klasse von tUsesMOP abzuleiten. Dies entspricht der ET++ Oberklasse Object, und führt für Teile der BibV30 zu annähernd baumförmigen Klassenhierarchien. Die Verwendung des C++ RTTI vermeidet dieses Problem (vgl. 2.1.4).

Derzeit sind MOP und späte Erzeugung eng gekoppelt. Ziel ist ein optionaler Mechanismus für späte Erzeugung. Zur Laufzeit abrufbare dynamische Typen zählen zur allgemein notwendigen Infrastruktur, späte Erzeugung ist auf besondere Fälle zu beschränken. Einsteigern bereiten derart komplexe Mechanismen vermeidbare Schwierigkeiten, da sie diese vorläufig selbst nicht einsetzen oder benötigen.

Technische Probleme ergeben sich zudem aus der vorliegenden Implementation. Einerseits sind zwar abstrakte Klassen, nicht aber aufgeschobene Methoden zulässig. Andererseits weisen heutige Debugger erhebliche Mängel bezüglich Umgang mit C++ Makros auf, der Benutzer kann diese nur sehr schwer nachvollziehen. Das Design Pattern „Late Creation“ sollte daher zukünftig als unabhängiges Mikro-Framework reimplementiert werden.

Persistenz, Filesystem und Datenbanken in der BibV30

In der ersten Version der Bibliothek fehlten Konzepte zur Persistenz von Materialien. Diese Frage wurde frühzeitig in einer Studienarbeit behandelt ([WW94]), deren Ergebnisse jedoch nicht in die BibV30 integriert. Derzeit enthält die Anwendungsumgebung einen einfachen Mechanismus zur Speicherung im Unix - Filesystem, die Speicherung in (objektorientierten) Datenbanken ist aktuell wieder ein Thema der Bibliothekserweiterung. ([BG96])

Ziel der kommenden Monate ist die Implementation eines Persistenz - Konzeptes mit Bezug auf die bereits geleisteten Arbeiten in diesem Bereich. Gedacht ist an das Atomizer Design Pattern (oder auch Serializer, vgl. [BMRSZ96]). Der Mechanismus könnte ähnlich dem Change Propagation in ET++ funktionieren, als Unterschied ist wieder zu nennen, daß dies nicht in eine allgemeine Oberklasse Object integriert, sondern optional verfügbar wird.

Kooperation: Prozeßmuster, Gruppen und Räume

Verschiedene Projekte am Arbeitsbereich Softwaretechnik beschäftigen sich mit kooperativer Arbeit. WAM begann mit starkem Bezug auf Einzelarbeitsplätze, neue Metaphern erweitern das Konzept hinsichtlich Koordination von Büroarbeiten mit mehreren Beteiligten. [Gryczan95] stellt eine mögliche Kooperationsform vor, die durch Prozeßmuster beschrieben wird. Postkörbe [Wulf95] realisieren den Informationsaustausch über die weiterhin erhaltene Grenze des einzelnen Arbeitsplatzes - entsprechend der WAM-Metapher Umgebung.

Die Bedeutung eines privaten, für Außenstehende nicht direkt zugänglichen Bereiches (also die Arbeitsumgebung), sehen auch [RW97] in ihren Überlegungen, für die Koordination ein Bild von Räumen einzuführen. Während der Ursprung für Prozeßmuster und Postkörbe im Bereich von Büroarbeitsplätzen liegt (Bankenbereich, vgl. [GWZ96]), ist ein Projekt für Krankenhaus - Software Auslöser für neue Kooperationsformen [KRW96]. Offen ist noch die Berücksichtigung von Gruppenarbeitsplätzen innerhalb des WAM - Umfeldes, zu erwarten sind in jedem Fall neue Impulse für die Methode sowie die BibV30.

Frameworks für WAM folgen dem Leitbild selbstbestimmter Arbeit, ursprünglich beschränkt auf Einzelarbeitsplätze. Eine Erweiterung des Einsatzkontextes der BibV30 setzt mit WAM konforme Mechanismen voraus, um kooperative Arbeitszusammenhänge über die einzelne Umgebung hinaus zu unterstützen. Zur Zeit liegt der Entwicklung keine bestimmte Domäne zugrunde, angestrebt wird vielmehr eine Bibliothek, die als Grundlage für viele verschiedene Anwendungsbereiche benutzbar ist. Erreichbar ist dies durch Integration diverser Konzepte, jeweils als unabhängige, wahlweise verwendbare Subwerkzeuge, Automaten oder Mikro-Frameworks realisiert.

Zum gegenwärtigen Zeitpunkt enthält die BibV30 keine Komponenten für die Koordination kooperativer Arbeiten. Eine prototypische Implementation für Prozeßmuster und Postkörbe liegt vor, die Integration in das WAM-Framework wäre naheliegend. Konzeptionell anzumerken ist bezüglich dieser Kooperationsform deren Unterscheidung von Workflow-Systemen

und die Verträglichkeit mit dem Leitbild der WAM-Methode. [Gryczan95] diskutiert beides ausführlich, hier soll nur betont werden, daß die Software keine Steuerung vornimmt, sondern selbstbestimmtes Handeln des Anwenders für die Koordination maßgeblich ist. Kooperative Arbeitssituationen, deren Ablauf alle Beteiligten kennen, vergegenständlicht das Material „Prozeßmuster“ in Form einer Beschreibung des Vorganges. Das Prozeßmuster dient als Laufzettel an einer Vorgangsmappe, gibt Auskunft über den nächsten zu erledigenden Schritt und bereits durchgeführte Arbeiten; es sagt aber nicht aus, wie die Arbeiten zu geschehen haben. Der Benutzer selbst entscheidet, ob er seinen Teil der Arbeit erledigt hat, und gibt die Vorgangsmappe dann explizit weiter, indem er diese in seinen Postausgangskorb legt. Nur an dieser einen Stelle wird ein Automat aktiv, der den Transport aus diesem bestimmten Ort der Umgebung in einen Posteingangskorb einer anderen Umgebung veranlaßt.

Grundsätzlich ist zur konzeptionellen Berücksichtigung kooperativer Arbeit anzumerken, daß nicht „die Kooperation“, sondern jeweils genau bestimmte Situationen unterstützt werden. Hier wird eine Schwäche in der CCW-Diskussion gesehen. ([GWZ96]). Konsequenz für die BibV30 ist die parallele Bereitstellung mehrerer, jeweils zugeschnittener Konzepte.

Verteilung, Prozeßräume und asynchrone Beobachtung

Verteilte Prozesse werden bezüglich WAM derzeit untersucht, jedoch ohne Zusammenhang mit der BibV30 (vgl. [Bleek97], auch [Fricke96]: CORBA). Ein besonderes Problem ergibt sich aus asynchronen Ereignissen, wenn die Funktions- und Interaktionskomponente eines Werkzeuges in eigenen Prozessen laufen. Die BibV30 kennt zur Zeit keine Konzepte für die Aufteilung einer Anwendung in parallele Prozesse.

Ansatzpunkt wären Asynchrone Ereignisse als Alternative zur synchronen Beobachtung, die in 4.3.3 vorgestellt wird. Vor der effektiven Implementation bedarf es einer prinzipiellen Klärung weiterer Auswirkungen.

Aufgaben der „Umgebung“

Gemäß WAM (vgl. 2.3) werden die Werkzeuge und Materialien eines Einzel - Arbeitsplatz in eine Umgebung eingebettet. Diese stellt einen Ort dar, ähnlich dem eigenen Schreibtisch (vgl. elektronischer Schreibtisch), und bildet eine Abgrenzung gegenüber Arbeitsbereichen anderer Personen. Von außen besteht kein Zugriff auf die Umgebung, dort abgelegtes Material ist für andere nicht sichtbar. Auch neues Material kann der Benutzer nur selbst in seine Umgebung holen, es wird nicht (wie in Workflow-Systemen) automatisch zugeteilt und nach Erledigung der Arbeit wieder weggenommen (vgl. hierzu den Unterpunkt *Kooperation: Prozeßmuster, Gruppen und Räume* betreff der Lösung mit Postkörben unter der Kontrolle des Benutzers).

Umgesetzt in eine technische Komponente gibt es je Benutzer eine spezifische Umgebung, in der genau jene Werkzeuge und Materialien vorliegen, die er für seine Arbeit benötigt. Die Umgebung startet diese Werkzeuge auf Anforderung des Benutzers und kennt den Zusammenhang zwischen Werkzeugtypen und Materialklassen (vgl. *Late Creation* weiter oben).

Innerhalb seiner Umgebung kann der Benutzer Material und Werkzeuge so anordnen und ggf. einrichten, wie es für seine Arbeitssituation und persönliche Arbeitsweise geeignet erscheint. Diese (persistente) Konfiguration des „Schreibtisches“ kann der Benutzer jederzeit ändern. Andererseits richtet sich die Einrichtung seiner Umgebung mit bestimmten Werkzeugen meist nach den funktionellen Rollen (vgl. [Floyd94]), die der Benutzer im Unternehmen erfüllt. Es bietet sich an, ein Konzept von Benutzerprofilen vorzusehen.

Weiterhin ist die Umgebung zuständig für die Synchronisation von gleichzeitig benutzten Werkzeugen, die auf demselben Material arbeiten (innerhalb dieser Umgebung!). Gemäß WAM erfolgt die Steuerung nicht „von unten“ über das Material, sondern explizit zwischen den Werkzeugen und der Umgebung. Auch die Verbindung zu Datenbankautomaten gehört nach [WW94] zu den Aufgaben der Umgebung.

Die derzeitige Version der BibV30 setzt nur ein sehr eingeschränktes Umgebungskonzept um. Persistenz, Benutzerprofile und auch der visualisierte elektronische Schreibtisch sind nicht umgesetzt, die Umgebung übernimmt nur die Aufgaben der Anbindung an ein Fenstersystem (siehe 4.3.3) und den Start der Werkzeuge (nächster Unterpunkt).

Werkzeug - Framework in der „SEBib“ *(Software Engineering Project - Bibliothek)*

Aufgrund der softwaretechnischen Trennung zwischen FK und IAK (vgl. Abschnitt 2.3) muß die Umgebung in der BibV30 jeweils beide Teile erzeugen, da das ein Werkzeug als ganzes nicht als Klasse vorhanden ist. Ich habe eine konkrete Werkzeugklasse für die Erzeugung und Konfiguration der Hauptwerkzeuge eingeführt, um dieses Problem zu lösen. Die Umgebung erzeugt ***Werkzeuge***, Zusammenstellung passender Funktionskomponente und evtl. mehrerer Interaktionskomponenten ist Aufgabe des Werkzeug - Frameworks. Die Zuständigkeiten bei Abbau einzelner Werkzeuge oder Schließen der ganzen Umgebung sind ebenso verteilt.

Noch arbeitet die BibV30 nach dem alten Prinzip. Das Fehlen der Werkzeug-Klasse ist in den bisher entwickelten Anwendungen unproblematisch, da es in einer Anwendungsumgebung selten mehrere Werkzeuge gibt. Entwickelt wurden weniger komplette Werkzeugsätze für tatsächliche Arbeitsplätze, als vielmehr einzelne Applikationen. Die Werkzeugklasse fehlt als Konzept, technisch erzeugt die Umgebung jedoch meist nur genau ein Haupt - Werkzeug und kann diesbezüglich auch mit der Erzeugung aller zugehörigen Komponenten, also einer FK und einer IAK belastet werden, auch mehrere IAK’ s wären unkritisch.

Konzeptionell stellt das Werkzeug - Framework einen Fortschritt gegenüber der Umgebung im WAM-Framework dar. Die Metapher „Umgebung“ wird getrennt von der Konstruktion der Werkzeuge mittels FK und IAK umgesetzt. Infolge der geplanten Erweiterung der Umgebung hinsichtlich verschiedener hier genannter Bestandteile, wird sich der Bedarf dieser Trennung ergeben. Der geplante Einsatz der BibV30 in einem größeren zusammenhängenden Projekt ist eine weitere Motivation. Realisieren läßt sich das Werkzeug - Framework auf Grundlage der bestehenden Anwendungsumgebung BaseFIAK, wie ich dies in der SEBib, eine vereinfachte Bibliothek für eine spezielle Lehrveranstaltung, erfolgreich getestet habe (Abb. 4-1).

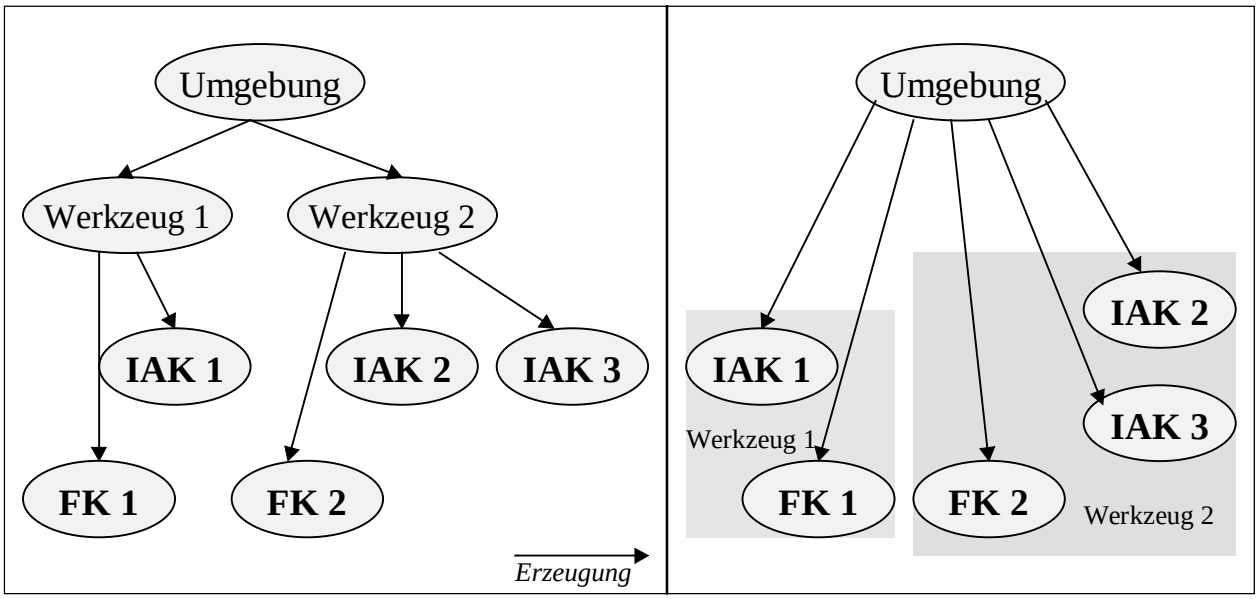


Abbildung 4 -1: Erzeugung von 2 Werkzeugen, mit und ohne Werkzeug - Framework

Application-Frameworks: BaseFIAK, MainFIAK und ToolCoordinator

Application-Frameworks in der BibV30 sind nicht domänenbezogen. Die vorhandenen drei Anwendungsumgebungen realisieren Varianten der allgemeinen Infrastruktur, ToolCoordinator und MainFIAK basieren beide auf BaseFIAK. Dessen Aufgabe liegt in der Konstruktion von Werkzeugen gemäß WAM, daß heißt die Verbindung zwischen Kontext - Werkzeug und den Subwerkzeugen. BaseFIAK macht dem Anwendungsentwickler keine Vorgaben bezüglich der Präsentation oder Handhabung seines Werkzeuges.

Erste Spezialisierung dieser Anwendungsumgebung war MainFIAK für typische MS-Windows Applikationen mit Menüzeilen und jeweils einem dargestellten Dokument. [Lilienthal97], die WWW-Dokumentation zur BibV30, beschreibt dies so:

> *„Dieser Layer kann verwendet werden, wenn Werkzeuge gebaut werden sollen, die sich wie ein Single Document - Werkzeug aus der MFC verhalten. Es gibt ein Menü für das Öffnen, Schließen und Speichern der Materialien und es wird sichergestellt, daß das Werkzeug ohne eine Rückfrage beim Benutzer nicht geschlossen werden kann, wenn das Material verändert wurde.“*

Vornehmlich der Entwicklung eines Hilfesystems [Lilienthal95] verdankt die BibV30 eine weitere Anwendungsumgebung. Das Hilfematerial besteht aus einem Hypertext, der Benutzer sieht und ändert jeweils eine beliebige Anzahl von Hilfeknoten in jeweils eigenen Fenstern.

ToolCoordinator hat als populäres Vorbild die populäre Programmierumgebung Sniff +, deren hier betrachtete Besonderheit der Schalter „frozen“ ist. Der Benutzer entscheidet für jeden Editor, ob der dort angezeigte Knoten erhalten bleibt, wenn er einen neuen Knoten lädt. Die Anwendungsumgebung ToolCoordinator realisiert Subwerkzeuge, die diesen Schalter in der

IAK anzeigen und die Funktionalität in der FK implementieren. Subwerkzeuge der Anwendung leiten sich von diesen Frameworkklassen ab.

Das Kontextwerkzeug verwaltet diese Subwerkzeuge. Wird z.B. in einem Hilfeknotenbrowser per Doppelklick auf einen Hyperlink ein weiterer Knoten geöffnet, so prüft die Kontext-FK, ob dieser Knoten gerade in einem anderen Browser angezeigt wird und bringt diesen ggf. in den Vordergrund. Anderenfalls wird der Knoten als neues Material in einen bestehenden Browser geladen, soweit dieser als wiederverwendbar (bei Sniff nicht frozen) gekennzeichnet ist. Sonst erzeugt das Kontext - Werkzeug einen neuen Hilfeknotenbrowser für das Material. Zusätzlich realisiert das Framework einen komfortablen History - Mechanismus.

Sowohl MainFIAK als auch ToolCoordinator fallen in die Kategorie Anwendungsumgebung. Es wird keine fachliche Domäne und kein bestimmter Anwendungstyp unterstützt, sondern nur eine Infrastruktur für die Applikation vorgegeben. Betreff MainFIAK wird dies schon aus dem Vergleich mit MFC deutlich, da auch dies (vgl. Kapitel 3) eindeutig zu klassifizieren ist. Die Standardisierung des ToolCoordinator - Frameworks beschränkt sich auf das Kontext - Werkzeug, Angaben über die Funktionalität der Subwerkzeuge oder deren Domäne werden nicht gemacht. Diese müssen nur als Werkzeuge gemäß WAM konstruiert sein (BaseFIAK).

Mit BaseFIAK, MainFIAK und ToolCoordinator stehen dem Benutzer drei Alternativen für den Einstieg in die Entwicklung seiner Anwendung offen. Dies entspricht dem grundsätzlichen Gedanken einer Framework - Sammlung für WAM, die verschiedenartige Applikationen innerhalb dieses methodischen Rahmens unterstützen soll. Wird die BibV30 als Basis für eine Anwendungsfamilie in einer bestimmten Domäne eingesetzt, so können spezifische Domänen-Frameworks auf diesen - und evtl. noch weiteren - Anwendungsumgebungen aufbauen.

Subwerkzeuge und Automaten

Zur BibV30 gehören fertige Subwerkzeuge, innerhalb des WAM-Konzeptes wurden diverse Automaten vorgeschlagen (z.B. Postversand - vgl. Kooperation). Letztere sind bisher nicht bzw. nur prototypisch realisiert und nicht Teil des WAM-Frameworks. Hier besteht einiger Nachholbedarf, da sowohl Kooperation als auch Persistenz (Datenbankautomat) von solchen Automaten abhängen.

Abschnitt 2.3 erläutert die softwaretechnische Umsetzung der Methode WAM und diskutiert die Bedeutung von Subwerkzeugen für die Konstruktion wiederverwendbarer Komponenten. Zu unterscheiden sind allgemeine, generische Subwerkzeuge (beispielsweise der in der BibV30 vorhandener Lister), ggf. nicht wiederverwendbare, anwendungsspezifische Subwerkzeuge, und als dritte Kategorie generische Vorlagen für Subwerkzeuge, die noch durch Ableitung eigener Klassen angepaßt werden müssen.

Für die Zukunft der Bibliothek sollte die Frage näher betrachtet werden, inwieweit generische Subwerkzeuge verstärkt in die BibV30 eingefügt oder statt dessen Vorlagen gemäß der dritten Kategorie geschrieben werden. Es handelt sich dann um spezifische Frameworks, eigentlich

Domänen-Frameworks. Allerdings kann mit diesen nur ein Subwerkzeug, nicht eine komplette Anwendung konstruiert werden, vielmehr erfolgt eine Einbettung in ein weiteres Application-Framework.

4.3.2 Grundlagen und Struktur

Design Pattern beeinflussen die Struktur der BibV30 in großen Teilen. Diese Mechanismen für flexibel gestaltete Hot Spots sind explizit in Form einzelner Bibliotheken vorhanden, damit erhält der Benutzer eine bessere Übersicht über die Vielzahl an Klassen in der BibV30, deren Verwendung und die Zusammenhänge.

Application-Frameworks in der BibV30 benutzen diese implementierten Versionen der Muster als besondere objektorientierte Mechanismen neben den eigentlichen Spracheigenschaften. Das Vertragsmodell zählt dagegen zu elementaren Grundlagen, die andere Programmiersprachen, namentlich Eiffel, bereits beinhalten. Auch die Implementation der Klassen String und Bool muß genannt werden, da diese zwar zwischenzeitlich zum C++ Sprachstandard gehören, aber noch nicht zum verwendeten Compiler.

Als heterogene Sprache stellt C++ ein Array-Konzept bereit, zusammen mit der Zeiger-Arithmetik als Relikt aus der Welt der C Programmierung anzusehen. Der folgende Unterpunkt zeigt auf, daß objektorientierte Sprachen höhere Behälterkonzepte ermöglichen und fordern. In Verbindung mit Unzulänglichkeiten einer heterogenen Sprache stehen auch Richtlinien.

Behälterklassen - die ConLib !

Praktisch jedes Projekt benötigt Behälter in irgendeiner Form, z.B. Liste, Array, Stack oder auch Dictionary. Die Sprache C++ trägt dieser Notwendigkeit erst jetzt im Zuge des neuen Standards durch die STL (Standard Template Library) Rechnung, C - Arrays bilden höchstens eine technische Grundlage.

Behälter zeichnen sich durch ihre Generizität[11] bezüglich enthaltener Elemente aus: der Typ enthaltener Elemente hat keine Auswirkung auf die Implementation. In einem Behälter befinden sich jedoch nicht beliebige Elemente, sondern jeweils Objekte einer Klasse (oder deren Unterklassen). Leider kannte die Sprache C++ vor der Einführung von Templates keine Generizität, ersatzweise wurde Polymorphie benutzt (vgl. ET++) oder das technische Konstrukt der „void-Pointer“. Ersteres erlaubt alle von der allgemeinen Wurzelklasse (Baumstruktur, 2.1.4) abgeleiteten Objekte in einem Behälter zu speichern - Spezialisierung auf eine Unterklasse ist nur durch Ableitung eines neuen Behälters möglich.

Das WAM-Framework wählte daher die andere Möglichkeit und stellte Typsicherheit dadurch her, daß der Benutzer die Listenklasse des Frameworks nicht direkt benutzen konnte, sondern

[11] Diese Problematik betrifft nur statisch getypte Sprachen. Typ und Klasse werden hier synonym benutzt.

jeweils eigene Klassen ableiten mußte. Zu implementieren waren jene Methoden, die Objekte in die Liste einfügten bzw. wieder herausgaben: beide Methoden benutzten dieselbe Klasse, der wegen Implementation mit void-Pointern notwendige Type-Cast war also sicher. Der Benutzer dieser Listen mußte immer selbst eigene Klassen implementieren, andere Behälterkonzepte fehlten in diesem Stand des WAM-Frameworks (vgl. [Riehle93]).

Mit den Templates in C++ entstand der Wunsch nach „besseren“ Behältern, die nun technisch möglich wurden. [Traub95] stellte fest, daß Implementationsvererbung für Behälterklassen - Bibliotheken dominiert, daß heißt z.B. die Ableitung einer sortierten Liste von einer Liste (dabei geht die Listeneigenschaft verloren, an beliebiger Position einfügen zu können). Die ConLib folgt mit ihrer Hierarchie einer echten Klassifikation nach Behälterkonzepten (siehe Abbildung 4-2). Der Benutzer entscheidet sich für das gewünschte Konzept, einzig an der Stelle der Erzeugung eines Behälters wählt er die Implementationsform - z.B. ein dynamisches Array mit schnellem Zugriff aber aufwendigerem Löschen und Einfügen, oder eine Double-Linked-List mit erhöhtem Platzbedarf. Bezeichner und Schnittstellenparameter sollten statisch vom Typ einer möglichst abstrakten Oberklasse deklariert sein.

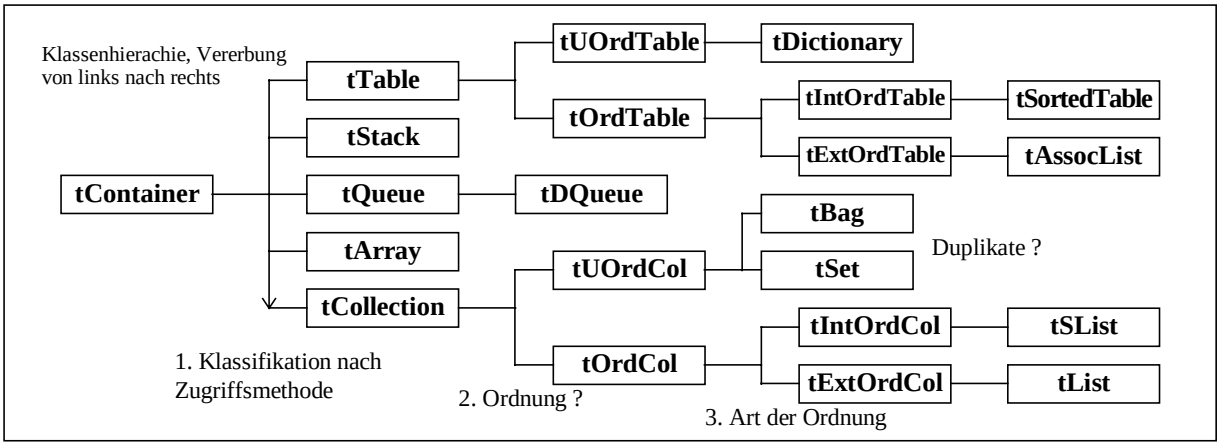


Abbildung 4 - 2: Behälter - Klassifikation der ConLib

Vergleichbar den robusten ET++ Iteratoren enthält die ConLib ein Konzept stabiler Cursors (zu den Begriffen vgl. [Traub95]), die in beliebiger Anzahl auf einem Behälter arbeiten und auch bei Änderung dessen Inhaltes in einem definierten Zustand bleiben. Die ET++ erreicht dies z.B. durch den Trick „verzögertes Löschen“ (Elemente bleiben erhalten, bis kein Iterator mehr auf sie zeigt) nur scheinbar, da die Semantik teilweise vorhandenen Elemente fraglich ist. Statt dessen erfaßt die ConLib den Sonderfall explizit im Zustandsmodell der Cursor-Klasse, im Anwendungsprogramm kann dies sicher behandelt werden.

Zur Implementation der Behälterkonzepte verwendet [Traub95] nur drei tatsächliche, höchst effiziente Behälter, deren aufwendige interne Struktur dem Benutzer vollständig verborgen bleibt. Mit der STL wird die ConLib nicht überflüssig, da deren ***Konzepte*** der STL überlegen sind. Denkbar wäre ein Austausch der genannten drei tatsächlichen Behälter als technische Basis der Bibliothek durch eine Abbildung auf die STL.

Die ConLib hat sich ohne Integration in die WAM-Frameworks bewährt: sie war mit diesen ***verträglich***, wurde neben deren Listen verwendet. Die BibV30 integriert ConLib - Behälter, die früheren eigenen Listen werden auch intern nicht mehr benutzt. Für den Benutzer entfällt die Problematik, verschiedene Listen verstehen zu müssen, um sich das Framework anzusehen. Außerdem profitiert die BibV30 von höherer Effizienz und Funktionalität den ConLib.

Vertragsmodell: Eiffel's Zusicherungen in C++ ?

Eiffel verfügt über Sprachkonstrukte zur Formulierung von Klasseninvarianten sowie Vor- und Nachbedingungen für Methoden. Dies sind mächtige Konstrukte für die Anwendung des Vertragsmodelles [Meyer88], die andere objektorientierten Sprachen nicht kennen. Um in C++ über die Kommentierung der Klassen und Methoden hinaus Vorbedingungen zu prüfen, kann der Makro - Mechanismus verwendet werden.

[Traub95] verweist in der Dokumentation zur ConLib auf den Gebrauch von Zusicherungen. Für jede Methode ist vorgegeben, welche Bedingungen bei dem Aufruf gültig sein müssen, Verletzungen sind Fehler des Benutzers der ConLib - Klassen und führen - bei Prüfung der Vorbedingung - zum Abbruch des Programmes und Ausgabe einer Fehlermeldung. So wird die Konsistenz der Behälter gewährleistet.

In der BibV30 stehen Makros für Vor- und Nachbedingungen zur Verfügung, deren Prüfung per Compiler - Option gesteuert wird. Innerhalb des Frameworks findet das Vertragsmodell Anwendung, die Vorbedingungen sind grundsätzlich in der Schnittstelle (Header-File) der Klasse dokumentiert. Allerdings können dort auch Bedingungen angegeben werden, die nicht wirklich überprüft werden - z.B. aufgrund zu hohen Aufwandes. Da die Dokumentation der Vorbedingungen leider nur in Form eines Kommentars geschieht, bleibt deren Korrektheit in der Verantwortung des Entwicklers. Eine vollständige Nachbildung des Eiffel - Konzeptes ist in dieser einfachen Form nicht möglich, da die Prüfung im Rahmen der Implementation, die Dokumentation jedoch in der Schnittstelle erfolgt (Source-File und Header-File). Die korrekte Verwendung des Vertragsmodells richtet sich nach den Programmierrichtlinien.

Programmierrichtlinien

Frameworks werden von sehr vielen Benutzern gelesen und wiederverwendet. Andererseits erfolgt auch die Entwicklung durch mehrere Personen und oft über einen langen Zeitraum. Schon aus diesem Grund ist die Anwendung einheitlicher Programmierrichtlinien für Frameworks noch wichtiger, als für sonstige Softwareprojekte (ein Beispiel für Richtlinien ist [Taligent94]). Am Arbeitsbereich Softwaretechnik wurden früh einige Regeln aufgestellt und von den Konstrukteuren der Bibliotheken in der Dokumentation genannt [Riehle93a], „verbindliche" Programmierrichtlinien [RWW95] aber erst am Beginn der Redesign-Phase erstellt. Diese Richtlinien enthalten Hinweise zur Gestaltung der Klassenschnittstelle (z.B. Reihenfolge der Abschnitte -public/protected/private- und Methoden; Namenskonventionen für Methoden, Präfix für Klassen etc.), wichtige Hinweise zur Verwendung objektorientierter

Prinzipien innerhalb der heterogenen Sprache C++ und auch ganz spezielle Vorschriften für das WAM-Framework und damit entwickelter Anwendungen.

Ein wesentlicher Punkt ist die Vorschrift, Objekte grundsätzlich dynamisch anzulegen (als Grundlage für Polymorphie, vgl. [Pree94]) und Methoden grundsätzlich virtual zu deklarieren (außer, deren Änderung in abgeleiteten Klassen ist ausdrücklich verboten). Ein sehr spezieller Bestandteil ist der Aufbau eines Init-Baumes - parallel zum Konstruktor (vgl. [Riehle93]).

Design Pattern und die BibV30

Gegenüber den ersten, relativ kleinen Versionen des WAM-Frameworks (BibV1.1 und V1.2), hatte sich mit Erweiterungen der Anwendungsumgebungen und verschiedener Mechanismen, z.B. späte Erzeugung, eine große, unübersichtliche Bibliothek ergeben. Erste Schritte zur Strukturierung waren die Trennung der Application-Frameworks (BaseFIAK, MainFIAK, ToolCoordinator) in einzelne Bibliotheken, daneben bildet IATMotif eine eigene Bibliothek, die Umgebung eine weitere. Die ConLib war ohnehin eine gesonderte Bibliothek.

Implementierte Subwerkzeuge, z.B. der in jede Anwendung integrierbare GUI-Builder für Motif, wurden ebenfalls getrennt, als letzte Komponente bleibt dann noch eine Bibliothek mit Standardklassen zu nennen (Bool, String, ...), in der auch das MOP angesiedelt ist.

Neue Komponenten der heutigen Framework - Sammlung sind vor allem Implementationen auf Grundlage verschiedener Design Pattern. Als Muster für Problemlösungen müssen Design Pattern je nach Anwendungsfall angepaßt werden, lassen sich nicht allgemeingültig realisieren (siehe Abschnitt 2.2, vgl. [BMRSS96]). Die Implementationen verschiedener Design Pattern beziehen sich jeweils auf konkrete Verwendungszusammenhänge innerhalb der BibV30, vor einer Benutzung für andere Komponenten einer Anwendung bedarf es einer näheren Prüfung.

Die Design Pattern halten gemäß [Pree94] Hot Spots des Frameworks flexibel. Eigenständige Mikro-Frameworks für diese Mechanismen tragen zu besserem Verständnis der Framework - Architektur bei, da der Benutzer deren Verwendung und somit die Hot Spots leicht erkennt.

Im Unterschied hierzu steht das Konzept der Entwickler von ET++. [WG94] beschreiben die Konzentration der Infrastruktur in einer gemeinsamen Oberklasse Object. Gewissermaßen ist dies eine erhebliche Erweiterung der Sprachfunktionalität, da alle Mechanismen implizit dem Anwender verfügbar sind. Daraus folgt aber auch, daß nur bei einer näheren Untersuchung der Implementation die verwendeten Design Pattern gefunden werden.

Die BibV30 strebt explizite Bezüge auf höhere objektorientierte Mechanismen an, sieht diese nicht als einfache Spracherweiterungen. Infolge dessen gibt es eine Reihe Mikro-Frameworks, die jeweils ein Design Pattern zur Vorlage haben. Eines dieser Design Pattern ist das in [GoF95] beschriebene Singleton Pattern. [Willamowius97] beschreibt eine verbesserte Version des Musters, eine abstrakte Implementation (vgl. [Vlissides96]). Im Gegensatz zum Original sind konkrete Singleton- Klassen hiervon ableitbar.

Einige derzeit implementierte Mikro-Frameworks für Design Pattern beschreibt 4.3.3 unter dem Stichwort ***Beobachtung*** vor. Technisch ließe sich statt dessen ein einziger Mechanismus verwenden, z.B. „Change Propagation“ wie in ET++, oder der im früheren WAM-Framework vorhandene „Notifier“. Die Vorteile der jetzt gewählten Lösung werden unten beschrieben. Weitere Implementationen sind bereits geplant (z.B. Serializer für Persistenz), ein Mangel stellt die momentan nicht vom MOP trennbare späte Erzeugung (s.o.) dar. Dieses Design Pattern sollte durch ein eigenes Mikro-Framework repräsentiert sein.

4.3.3 Anschluß des Fenstersystems

Interaktionstypen als Design Pattern nach WAM entkoppeln das Werkzeugs, genauer dessen IAK, von dem verwendeten Fenstersystem. Statt spezifischer, je nach GUI unterschiedlicher Widgets, orientieren sich IAT’s an ***Formen*** der Benutzer- Interaktion, die dann auf konkreten Systemen umzusetzen sind. Es stellt sich dabei die Frage, an welcher Stelle der Software eine ***Layout*** - Beschreibung stattfindet.

Graphische Benutzungsoberflächen gelten als Voraussetzung für die Entwicklung ***reaktiver*** Systeme (im Prinzip ließen sich derartige Systeme aber auch an Textterminals einsetzen). Die Kontrolle geht an das System über, das entsprechend eingehender „***Events***“, z.B. Eingaben des Benutzers, mittels Callback (Reaktionsmechanismus) Methoden der Anwendung aktiviert. IAT’s leisten den Anschluß an die Events, die Kontrolle wird an das Fenstersystem abgegeben.

Reaktionsmechanismen haben im Kontext von WAM eine große Bedeutung. Nicht nur das Fenstersystem wird in dieser Form mit der Anwendung verbunden, auch die Beobachtung (2.3) zwischen Werkzeugkomponenten folgt diesem Konzept. Aus historischen Gründen beschreibe ich die Design Pattern ***Observer*** und ***Chain-Of-Responsibility*** hier in Verbindung mit dem Fenstersystem, statt im Bereich der Application-Framework (vgl. 4.3.1).

Im folgenden werde ich zeigen, wie die Erweiterung der BibV30 vom jeweiligen Fenstersystem unabhängige Anwendungen vorbereitet, wo die Trennung zwischen der Funktionalität und Interaktion eines Werkzeuges und dessen Erscheinungsbild liegt und wie die Application-Frameworks im WAM-Framework von den jeweils spezifischen IAT- Bibliotheken entkoppelt sind. Die Unabhängigkeit ist wesentliche Voraussetzung für eine einfache Portierung sowohl der Anwendungen als auch des Frameworks selbst auf neue Umgebungen. Alle Änderungen tragen zum besseren Verständnis der Bibliothek und damit entwickelter Anwendungen bei.

Hierarchien von GUI-Widget-Objekten zur Laufzeit

Viele Fenstersysteme ordnen Oberflächenelemente hierarchisch an, Rahmen fassen elementare Komponenten, z.B. Knöpfe, zusammen, mehrere Rahmen sind in Fenster eingebettet. Zur Laufzeit besteht ein Baum der Oberflächenobjekte, der aus dieser Einbettung resultiert. Als Wurzel wird z.B. in Motif ein nicht sichtbares Objekt eingeführt.

Das Application-Framework erzeugt die Wurzel des Baumes und stellt so die Anbindung des Fenstersystems her. Jedes Widget benötigt einen Verweis auf den Knoten, unter dem es in den Baum eingefügt wird. Als Konsequenz ergibt sich ein Verknüpfung zwischen IAT- Bibliothek als Kapsel um das Fenstersystem und der nicht GUI - abhängigen Anwendungsumgebung.

Eine abstrakte Hook- Klasse als Oberklasse für spezielle IAT- Bibliotheken löst die ungewollte Verbindung des Application-Framework mit einem bestimmten Fenstersystem. Derzeit lassen sich die Anwendungsumgebungen in der BibV30 sowohl mit OSF/Motif als auch wxWindows benutzen, die Interaktionstypen für Tcl/TK lassen sich leicht anpassen. Dennoch ist diese Hook- Klasse nur eine partielle Lösung des Problems.

Layout - Informationen in der IAK ?

[Calder95] beschreibt die deklarative Positionierung von Oberflächenelementen in Interviews als Vorteil gegenüber festen Koordinatenangaben. Fenster, deren Größe der Benutzer ändert, positionieren ihre Inhalte unter Berücksichtigung der neuen Rahmenwerte selbst neu, wobei sich Interviews am Beispiel der Textformatierung orientiert (vgl. Kapitel 3).

Auch ET++ verfügt über ähnliche Mechanismen, der Softwareentwickler definiert die relative Positionierung durch Einbettung seiner Widgets in spezielle Layout - Objekte. Die Hierarchie von Oberflächenelementen in Systemen wie Interviews, ET++ aber auch OSF/Motif und Tcl/TK dient also dieser vorteilhaften Positionierung ohne feste Koordinaten. Im Unterschied zu ET++ erfolgen Einstellungen in einer separaten Ressource - Datei, nicht wie in ET++ an den einbettenden Objekten selbst.

Interaktionstypen sollen sich gemäß WAM auf Formen der Interaktion beschränken, nicht auf deren Umsetzung in bestimmten Systemen. Eine IAT-Bibliothek dieser Art läßt sich auf jedes Fenstersystem portieren, bestimmte Layout - Informationen dagegen nicht oder nur mit sehr großem Aufwand. Ressource - Dateien als externe Beschreibungen entschärfen dieses Problem wesentlich, da die Werkzeug selbst, also die IAK vom Layout getrennt wird. OSF/Motif trennt Einstellungen an den Layout - Oberflächenelementen (Board, Frame, ...) von den Objekten, der Objekt- Baum ist aber in der IAK aufzubauen.

Mit dem Ziel einer kompletten Trennung der Angaben zur Präsentation von den eigentlichen Interaktionstypen, wurde die IATTcl - Bibliothek entwickelt und in der SEBib[12] getestet (noch in die BibV30 integriert). Interaktionstypen entsprechen elementaren Oberflächenelementen, für Layout - Widgets gibt es keinen IAT. Sämtliche Angaben zur Präsentation finden sich in dem Tcl/TK- Skript, der Ressource - Datei für dieses System. Die Verbindung zwi-

[12] Die SEBib wurde für eine konkrete Lehrveranstaltung, ein Projekt über zwei Semester entwickelt. In diesem Zusammenhang hat ein Student (Jan Raap) die IATTcl - Bibliothek implementiert. Das Konzept der Beschränkung aller Layout - Informationen auf das Tcl/TK-Skript bei kompletter Entlastung der IAK diesbezüglich, habe ich gemeinsam Martina Wulf entworfen (derzeit wissenschaftliche Mitarbeiterin am Arbeitsbereich und Veranstalterin des Projektes).

schen IAT und Beschreibung im Skript stellt eine fachliche Bezeichnung her, z.B. Beenden für einen Knopf oder Menüeintrag mit der Beschriftung Beenden oder auch Quit.

Portable Anwendungen gehören auch zu den Zielen von ET++, obwohl hier sämtliche Angaben zum Layout in der Applikation direkt stehen. Dafür investieren die Framework-Entwickler in die Entwicklung der Klassen für Oberflächenelemente einen weitaus höheren Aufwand und greifen nicht auf bestehende Widget - Bibliotheken der einzelnen Systeme zurück. Ein Problem bleibt aber das look&feel der Anwendungen: ein einheitliches Layout für alle Plattformen muß zwangsläufig die unterschiedlichen Styleguides verletzen.

Einhaltung von Styleguides und reduzierter Aufwand sind Argumente für die im WAM-Kontext gewählte Lösung. Ein weiteres ist die Vorbereitung für Oberflächen - Prototyping.

Eine Voraussetzung für das Prototyping hinsichtlich graphischer Benutzungsoberflächen ist die Möglichkeit, diese mit komfortablen Werkzeugen schnell entwerfen und danach ohne große Wartezeiten testen zu können (vgl. BBLSSZ94]). Die Integration der Beschreibung in den Teil des Programmes, der jeweils neu zu compilieren ist, erfüllt diese Ansprüche nicht, da lange Wartezeiten bei kleinen Änderungen entstehen. Ressourcen als externe Beschreibungen des Layout wertet das Laufzeitsystem direkt aus.

IAT- Bibliotheken als Verbindung zwischen Werkzeug (IAK) und Fenstersystem trennen die Implementation von der Gestaltung der Benutzeroberfläche. Prototyping setzt darüber hinaus komfortable Werkzeuge voraus, um diese Oberfläche schnell und ohne spezielle Kenntnisse im Bereich der Informatik zu entwerfen, z.B. unter Mitwirkung von Arbeitspsychologen oder auch Grafikern. Direkte Manipulation hat sich als geeignete Umgangsform erwiesen, zur Zeit ist dies in der BibV30 leider noch nicht umgesetzt. Der Interface- Builder für IATMotif ist textbasiert, allerdings kann die Anpassung zur Laufzeit erfolgen, also mit sofortiger Kontrolle der Auswirkung.

Neuer Ansatz: Trennung Interaktionsformen und Präsentation

WAM-Frameworks mit Interaktionstypen haben mit der Entkopplung zwischen Application-Framework und konkreter IAT-Bibliothek sowie der *Auslagerung* der Oberflächengestaltung in spezielle Werkzeuge und damit erzeugter Ressourcen das Ziel portabler Anwendungen zu einem Großteil erreicht. Verschiedene Arbeiten beschäftigen sich mit der Problematik (z.B. [Fröse96] für StarView). Der größte Nachteil heutiger Lösungen und der BibV30 liegt in der direkten Orientierung implementierter Interaktionstypen an vorhandenen Widgets eines GUI - Systems, statt der Beschränkung auf „Formen der Interaktion“.

Grundlage der folgenden Darstellung sind neben eigenen Überlegungen vor allem Ergebnisse einer Serie von inzwischen elf halbtägigen Workshops zum Thema Interaktionsformen in etwa monatlichem Abstand. Zentrales Diskussionsthema im (festen) Teilnehmerkreis von ca. 6 bis 9 Personen sind allgemeine Formen der Benutzerinteraktion und deren Präsentation.

Wichtig ist die Trennung zwischen Präsentation und Interaktion, da nur letztere Einfluß auf die IAK haben sollte. Angestrebt ist die vollständige Behandlung von Darstellungsaspekten durch Werkzeuge zur Oberflächengestaltung. Styleguides bestimmter Plattformen und die Widgets eines konkreten Fenstersystems haben hinsichtlich der Präsentation Bedeutung, die Interaktion wird im Workshop hiervon unabhängig diskutiert.

Interaktionsformen sind auf der Ebene „1 from N Selection“, „M from N Selection“ und „Activation“ angesiedelt. Ob der Benutzer ein Kommando über einen Knopf, das Menü oder durch einen gesprochenen Befehl aktiviert, ist für die IAK ohne Bedeutung. Auch Auswahlen stellen heutige Fenstersystemen in unterschiedlichster Form dar, z.B. Liste, Combobox oder Drop-Down-Feld. Die Entwicklung neuer Widgets erweitert ständig die Auswahl auf Seiten der Präsentation, völlig neue Formen der Interaktion sind dies im allgemeinen nicht.

An dieser Stelle sollen vorliegende Teilergebnisse und Ansätze für zukünftige Lösungen nicht im einzelnen aufgeführt werden. Ich gehe davon aus, daß ein vollständiger Katalog und vor allem dessen Umsetzung in [Görtz97] zum Jahresende vorliegen wird. Titel der Diplomarbeit ist die *„Abstraktion der GUI - Komponente in einem Rahmenwerk“*, vorläufige Ergebnisse fließen in den Workshop ein und prägen die Diskussion. Nachgewiesen wird die Tragfähigkeit des Ansatzes dann durch konstruktive Umsetzung auf mehreren Plattformen, die Integration in die BibV30 als Ersatz für heutige IAT-Bibliotheken.

GUI und Event-Loop

Thema der letzten Unterpunkte war die Erzeugung von Oberflächenelementen unabhängig vom Fenstersystem. Reaktive Systeme zeichnen sich zudem durch einen Informationsfluß in der umgekehrten Richtung aus. Die aktive Komponente der Anwendung ist nicht das Werkzeug, sondern das Fenstersystem. Dorthin gelangen Systemereignisse, z.B. Benutzereingaben. Fast allen Fenstersystemen gemeinsam ist eine integrierte Event-Loop, also eine Schleife, die als zentraler Teil des laufenden Programmes Benutzereingaben in Form von Events der Reihe nach bearbeitet. Das Application-Framework stellt die Verbindung zum GUI her, gibt die Kontrolle an dieses ab. Später wird die Anwendung an verschiedenen, zeitlich wie auch hinsichtlich der Abfolge nicht vorhersehbaren „Stellen“ (Methoden beliebiger Klassen) vom Fenstersystem aufgerufen, und muß darauf reagieren. Diese Abkehr von der Ablauf - Steuerung in früheren Systemen wird reaktiv genannt, da nicht die Software aktiv ist, sondern jeweils vom Benutzer durch Eingaben aktiviert wird.

Anwendungsumgebungen in der BibV30, namentlich BaseFIAK, da die alle anderen hiervon abgeleitet sind, arbeiten mit einem abstrakten Window-System-Hook, Unterklassen für Motif und wxWindows implementieren dessen abstraktes Protokoll. Bei den Hook - Klassen handelt es sich um Singletons, die Objekte erzeugt die für C++ vorgeschriebenen Start- Prozedur „Main“, hier erfolgt die Festlegung des konkreten Fenstersystems.

Aufgabe des Hook ist die Verbindung zwischen GUI und IAT als dessen Kapsel. Als nächstes geht es um die Weiterleitung der Ereignisse von den Interaktionstypen an das Werkzeug bzw. dessen Interaktionskomponente.

Der (alte) Beobachtungsmechanismus …

Interaktionskomponenten erzeugen IAT's und nehmen Einstellungen an diesen vor, z.B. indem sie einen IAT vorübergehend deaktivieren. Umgekehrt erreichen Ereignisse infolge Benutzeraktion den Interaktionstyp und müssen an die IAK weitergeleitet werden. Die Klassen kennen sich gegenseitig, wechselseitige Benutzung als enge Kopplung ist aber unerwünscht.

Mechanismen bzw. Design Pattern für lose gekoppelte Komponenten arbeiten mit abstrakten Oberklasse, das heißt eine Klasse kennt nicht die konkret verbundene andere Klasse, sondern nur deren Oberklasse. Das Protokoll der abgeleiteten Klassen bleibt unabhängig von dieser Benutzt-Beziehung, Änderungen wirken sich nicht auf die (lose) gekoppelte Klasse aus.

Das erste WAM-Framework erreichte die lose Kopplung durch *Beobachtung* [vgl. Riehle93], die Oberklasse „tNotifier" enthielt diesen Mechanismus. Im wesentlichen heißt Beobachtung einer Komponente, deren Zustand zu sondieren. Zur Vermeidung fortgesetzter Abfrage der Werte meldet die beobachtete Klasse jede eintretende Zustandsänderung an den Beobachter. Konkret benutzt hier also die IAK einen IAT, kennt dessen Schnittstelle, kann Werte setzen und abfragen. Der IAT hat nur einen Verweis auf eine abstrakte Oberklasse der IAK, kann somit nur diese gemeinsame Funktionalität aller Interaktionskomponenten benutzen, um ein Signal im Falle einer Benutzeraktion (also Änderung des Zustandes) zu melden.

Eine IAK registriert sich bei dem von ihr erzeugten IAT als Beobachter. Diese Methode ist in der Oberklasse tNotifier aller beobachteten Klassen implementiert. Ändert jetzt der IAT seinen Zustand, so ruft er die Methode „Tell" der Oberklasse auf, diese wiederum meldet das Ereignis an registrierte Beobachter. Dazu wird die Methode „Notify" der IAK benutzt, die in deren Oberklasse tNotifier[13] deklariert ist. Jede IAK muß diese Methode überschreiben, um auf die eingehenden Meldungen zu reagieren.

Ein Parameter identifiziert den Ursprung der Meldung, ein zweiter, numerischer Parameter den Typ des Ereignisses, den IAT bei Aufruf der Methode Tell festlegt. Die Methode Notify der IAK besteht aus einem großen „switch"- Konstrukt, um entsprechend Ereignis und IAT zu reagieren. Dazu müssen meist Werte des meldenden IAT sondiert werden.

Vergleichbar ist der Mechanismus mit ET++ „Change Propagation" (vgl. [WG95]) und auch der entsprechenden Basisfunktionalität in Smalltalk (wird hier als nicht getypte Sprache nicht weiter untersucht). Wie im Beispiel ET++ wurde Beobachtung universell benutzt, zwischen IAT und IAK, zwischen IAK und FK sowie zwischen Kontext und Subwerkzeugen.

---

[13] Der Klassenname tNotifier bringt eine begriffliche Verwirrung mit sich, da von der Oberklasse sowohl der Beobachter als auch der Beobachtete abgeleitet wird. Dieser Entwurf hatte technische Gründe und soll hier nicht ausgeführt werden, es wurden beide Konzepte in einer Klasse vereint.

… und die neuen Design Pattern

Heute ersetzen verschiedene Design Pattern je nach Verwendungszusammenhang den eher technisch orientierten Universalmechanismus. Command, Observer und Chain of Responsibility sind als eigenständige Mikro-Frameworks implementiert, Kommandos betreffen IAT und IAK, die Kette und auch das Observer - Pattern verbinden Werkzeug - Komponenten.

Benutzeraktionen lösen bestimmte Methodenaufrufe in der IAK aus, dazu übergibt diese dem IAT nach der Erzeugung spezielle Command - Objekte. Statt in einer Methode Notify über alle möglichen Ereignisse informiert zu werden und den Zustand des IAT zu sondieren, erzeugt die IAK nur für relevante Ereignisse Kommandos. Der Aufruf der zugeordneten Methode enthält alle notwendigen Parameter (sondieren entfällt) und erfolgt direkt durch den IAT, nicht mehr in einem „switch“ - Konstrukt der IAK. Abbildung 4-3 skizziert die Struktur des Musters und den Ablauf (vgl. Darstellung des Design Pattern in [GoF95]).

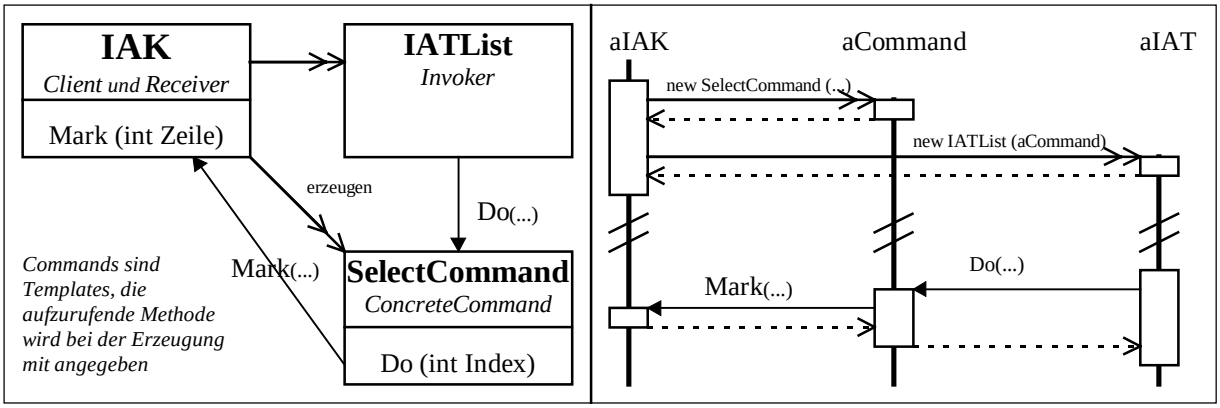


Abbildung 4 - 3 a) Klassendiagramm, b) Interaktionsdiagramm (Erläuterung vgl. Abb. 4-8)

Konzeptionelle Bedeutung der Änderung

Motivation des Beobachtermechanismus war die Annahme, daß eine Komponente die andere beobachtet, also deren Zustand sondiert. Um ständiges Abfragen zu vermeiden, meldet diese beobachtete Komponente Zustandsänderungen mittels eines Signals an seine Beobachter. Es können beliebig viele Beobachter registriert sein und die beobachtete Komponente kann keine Annahmen darüber treffen, ob das Signal empfangen bzw. beachtet wird.

Zwischen IAK und FK wie auch zwischen Kontext-FK und Sub-FK bestehen Verbindungen dieser Art. Die BibV30 verwendet dafür jetzt das Design Pattern Observer. Im wesentlichen entspricht dies dem Konzept des Beobachters (vgl. [RW96]), verbessert wurde insbesondere die Umsetzung. Es gibt die Oberklassen „Observable“ und „Observer“, „Events“ stehen als eigene Objekte für die Ereignisse. Implementiert wurden die Mikro-Frameworks für Observer und Command per Template und Methoden - Pointer, Einzelheiten finden sich in [RW96].

Konzeptionell stellen die explizit an der Schnittstelle einer FK kenntlichen Ereignisse einen großen Fortschritt dar. Umgekehrt dokumentieren zu den Events gehörende Event-Stubs in der IAK, für welche Ereignisse sich die IAK registriert. Beides ließ sich früher nur mühsam aus der Implementation der Klassen ermitteln, Zusammenhänge sind jetzt direkt zu erkennen.

Die Neuerung löste am Arbeitsbereich eine ausführliche Diskussion über die Verwendung von Statecharts bei der Konstruktion eines Werkzeuges aus, da Ereignisse als Zustandsübergänge innerhalb der FK interpretiert werden. Eine Analyse bestehender Werkzeuge ergab, daß hier teilweise Fehler in der Konstruktion vermeidbar gewesen wären. Zudem zeigte sich eine nicht adäquate Verwendung des Beobachtungsprinzips in einigen Anwendungssituationen.

[Riehle93] betont hinsichtlich der Beobachtung, daß mit dem Aussenden eines Ereignisse keine Erwartung einer Reaktion verbunden ist. Eine FK weiß nicht, ob sie überhaupt beobachtet wird. Die Annahme einer Reaktion widerspricht dem Prinzip der losen Kopplung. Im neuen WAM-Framework realisiert eine Chain-Of-Responsibility (vgl. [GoF95] und [Riehle96a]) die Weiterleitung von Anforderungen an verantwortliche Komponenten, wenn die jeweilige Klasse nicht zuständig ist. Bisher wurde hierzu die Beobachtung als technisches Konstrukt benutzt (bzw. mißbraucht).

Zwei Beispiele für den Einsatz der Verantwortlichkeitskette: erstens im Application-Framework BaseFIAK Beendigung eines Werkzeuges durch dessen Kontext, zweitens in der spezialisierten Anwendungsumgebung ToolCoordinator die Anforderung eines Subwerkzeuges für neues Material. Da sich ein Werkzeug nicht selbst beenden bzw. ein Objekt nicht selbst löschen kann, wird dieser Auftrag an eine umgebende Komponente weitergeleitet, also z.B. ein Kontext - Werkzeug oder die Umgebung.

Der zweite Fall liegt vor, wenn z.B. im Hypertextbrowser per Doppelklick ein Verweis auf einen anderen Knoten verfolgt wird. Das Subwerkzeug interpretiert die Benutzeraktion als Aufforderung, diesen Knoten zu öffnen, ist dafür aber nicht selbst zuständig. Ob ein anderes Subwerkzeug mit diesem Material bereits besteht, ein neues erzeugt werden muß oder eines der bestehenden, z.B. auch eben jenes, in dem der Benutzer den Doppelklick ausgeführt hat, mit neuem Material versorgt wird, entscheidet das Kontext - Werkzeug als verantwortliche Komponente. Das Subwerkzeug leitet die Anforderung in der Kette weiter und geht von der Erledigung aus.

Ähnlich dem Command - Muster für IAT’s kann die IAK (oder eine Kontext-FK) als Observer entscheiden, welche Events von Bedeutung sind. Diese werden mit Event-Stubs verbunden, die für den Aufruf spezieller Methoden in der IAK sorgen. Bei den Events lassen sich beliebig viele Stubs registrieren, also z.B. einer für die Kontext-FK und einer für die IAK.

Somit beinhaltet die BibV30 jetzt zwei konzeptionell passende Design Pattern zur Kopplung von Werkzeugkomponenten. Ein weiterer Mechanismus, das Command Pattern, wird zwischen IAT und IAK benutzt. Technisch gemeinsam ist beiden Verfahren, daß die Funktionalität an der Klassenschnittstelle erkenntlich ist und direkt durch die Anmeldung für spezielle Ereignisse bzw. die Übergabe konkreter Commands die Verbindung mit der zuständigen Methode erfolgt. Der Universalmechanismus führte zuvor oft zur Verwirrung, da in der Methode Notify einer IAK gleichermaßen „Ereignisse“ von IAT’s als auch der FK eintrafen. Ausschlaggebend für die jetzt unterschiedliche Anbindungen ist aber vor allem der methodische Hintergrund.

Funktionskomponenten haben einen Zustand, den beliebig viele Interaktionskomponenten und Kontext-FK' s sondieren. Demgegenüber sind Interaktionstypen als zustandslose Funktionen zu verstehen, die irgendwann ein Ergebnis zurück liefern (Benutzerinteraktion) - und zwar an genau die IAK, die den IAT erzeugt hat. Die IAK kann vorher festlegen, welche Aktion dann auszuführen ist, der IAT liefert den „berechneten" Wert direkt mit (Parameter). Demgegenüber kann die FK keine Aussage treffen, welcher Teil ihres abstrakten Zustandes für den Beobachter relevant ist, diesen also nicht zusammen mit ihrem Signal übermitteln.

Erfolg der Diversifizierung sind übersichtliche Anwendungen, die Mechanismen sind jeweils dem Verwendungszusammenhang angepaßt. Eigenständige Mikro-Frameworks sind leichter nachzuvollziehen und die Verwendung wird explizit, besonders hinsichtlich der Ereignisse einer FK. Hinzu kommen technische Verbesserungen durch direkten Aufruf zuständiger Methoden, Beschränkung auf relevante Ereignisse und im Falle der Commands Übergabe von Parametern.

4.4 Integration fremder Konzepte: Graphik

Während Oberflächenelemente wie Knöpfe, Listen und Menüzeilen in verschiedenen Formen zu jeder graphischen Benutzerschnittstelle gehören, gibt es bisher in GUI - Systemen kaum Abstraktionen für die Unterstützung von Anwendungen mit spezifischen Grafikaktionen, z.B. Malprogramme mit direkter Manipulation.

Zu den ersten Anwendungen der WAM Bibliotheken zählte Sane, ein graphischer Editor für Aufgabennetze. Im Gegensatz zum AT - Konzept für „normale" Oberflächenelemente bzw. als Abstraktion der Benutzerinteraktion mit diesen Elementen, fehlte ein geeigneter Ansatz für direkte Grafik - Manipulation. Implementiert wurde graphische Grundfunktionalität in einer für Sane tragfähigen Form (siehe 4.4.1).

Im Zuge der Redesignphase sollte dieser Bereich der C++ Bibliotheken erneuert werden, es schien angebracht, vorhandene Konzepte aus fremden Systemen zur Vorlage zu nehmen. Ein diesbezüglich passendes Framework ist Unidraw (vgl. 3.2.2), dessen Ansatz ich in die BibV30 integriert habe. Dazu stelle ich den Bezug zwischen WAM und den Unidraw - Konzepten her (4.4.2) und nehme entsprechende Anpassungen vor (4.4.3). Vorrangig ist die Metapher der direkten Manipulation, in 4.4.4 beschreibe ich die konkrete technischen Umsetzung.

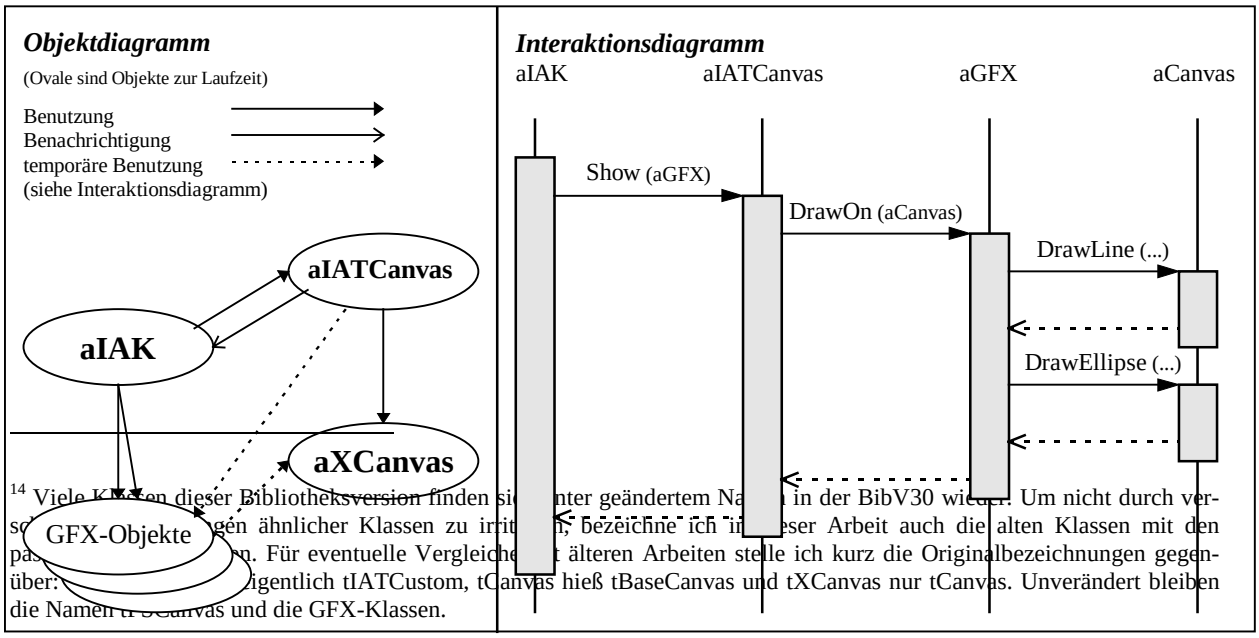
4.4.1 Graphikkonzept der Anfangsphase

Grundlegende Komponente des Ansatzes ist ein Interaktionstyp (tIATCanvas[14]) innerhalb der IATMotif - Bibliothek, der einem Bildschirmbereich für Grafikausgaben entspricht. Anwendungen zeichnen in diesem Bereich, umgekehrt meldet der IAT bei gedrücktem Knopf die Position des Mauszeigers. Verschiedene Klassen adressieren Zeichenfunktionalität, direkte Manipulation realisieren die Applikationen selbst mittels der gemeldeten Ereignisse (MouseUp, MouseDown, MoveToXY) - ohne Abstraktion innerhalb des Frameworks.

Zeichnen unterstützt das WAM-Framework durch die Klasse tCanvas, die Zeichenprimitiven als abstrakte Operationen definiert. Graphischer GFX - Klassen repräsentieren die konkreten Elemente (z.B. Dreieck, Kreis). Die Zeichenfläche tCanvas leistet eine Abstraktion bezüglich der tatsächlichen Ausgabe, es gibt konkrete Unterklassen für die Bildschirmausgabe im X-Windows - System und für den Ausdruck als Postscript - Datei. [Calder95] hebt für Interviews dessen nicht an Bildpunkten der technischen Basis orientierte Koordinaten hervor. tCanvas in den C++ Bibliotheken ist durch Koordinaten im Millimeterformat plattformunabhängig, zudem erfolgt der Ausdruck im richtigen Maßstab.

Abstrakte Beschreibung komplexer Formen stellt ein Kernproblem graphischer Anwendungen dar. GFX-Klassen in der C++ Bibliothek lösen dieses durch Bezug auf abstrakte Zeichenprimitiven: graphische Figuren sind Exemplare jeweils spezifischer auf die einzelnen Formen bezogener GFX-Klassen (z.B. tGFXRectangle, tGFXCircle, ...) und zeichnen sich selbst auf dem Canvas. Die Objekte „kennen“ ihre Position, Größe und Form, an der Schnittstelle gibt es Methoden zum Verschieben der Objekte und für Größenänderung.

Der Anwendungsentwickler implementiert für verschiedene Materialtypen jeweils eine eigene GFX-Klasse, abgeleitet von der abstrakten Oberklasse tGFXDrawable. Das Werkzeug benötigt einen tIATCanvas, jedes Materialobjekt wird durch ein Exemplar der passenden



[14] Viele Klassen dieser Bibliotheksversion finden sich unter geändertem Namen in der BibV30 wieder. Um nicht durch verschiedene Bezeichnungen gegen ähnlicher Klassen zu irritieren, bezeichne ich in dieser Arbeit auch die alten Klassen mit den passenden Namen. Für eventuelle Vergleiche mit älteren Arbeiten stelle ich kurz die Originalbezeichnungen gegenüber: tIATCustom hieß eigentlich tIATCustom, tCanvas hieß tBaseCanvas und tXCanvas nur tCanvas. Unverändert bleiben die Namen tIATCanvas und die GFX-Klassen.

Abbildung 4 - 4: a) Objekt-Beziehungen zwischen IAK, GFX-Objekt, tIATCanvas und tXCanvas;
b) Interaktionsdiagramm (vgl. Abb. 4-8): Darstellung eines GFX-Objekt im tIATCanvas

GFX-Klasse repräsentiert. Zur Darstellung wird die Methode Show des tIATCanvas aufgerufen und dabei das GFX-Objekt als Parameter übergeben. tIATCanvas verfügt über den tXCanvas, auf dem sich (double dispatch) das GFX-Objekt selbst zeichnet (Abbildung 4-4).

Mängel der Grafik- Komponenten liegen im Entwurf der abstrakten Klasse tCanvas, in deren Implementation durch tXCanvas und tPSCanvas sowie der Beziehung zwischen tIATCanvas und den GFX-Objekten. Letzteres betrifft vor allem die fehlende Funktionalität zur Verwaltung dargestellter GFX-Objekte in der Zeichenfläche. Resultat dieser Entscheidung der Framework-Entwickler ist eine hohe Belastung der Anwendung mit fachfremden Aufgaben, die sich in den Frameworkklassen zudem effizienter erledigen lassen. Zusammen mit der fehlenden direkten Manipulation ergibt sich für eine neue Lösung in der BibV30 die Gelegenheit, große Teile der technisch bedingten Aufgaben einer Applikation zu übernehmen.

Technisch bedingt ist die Notwendigkeit, Bildschirminhalte zu erneuern, wenn diese z.B. von Fenstern anderer Anwendungen zwischenzeitlich überdeckt wurden. Interaktionstypen, oft auch Widgets des Fenstersystems, erledigen dies ohne Eingriff des Anwendungsentwicklers, nur tIATCanvas bildet eine Ausnahme. Da der Zeichenfläche Informationen über ihren Inhalt fehlen, ist dessen Wiederherstellung Aufgabe der Anwendung. Während tIATCanvas aufgrund seiner Kenntnis des dargestellten Ausschnitts und des überdeckten Bereiches dies optimieren könnte, zeichnet die Anwendung grundsätzlich alle Elemente neu, z.B. auch beim Scrolling.

[Weiß94] beschreibt den spezifischen tIATSymbolNet, der für einen eingeschränkten Anwendungsbereich (vgl. [Ritz94]) die Funktionalität auf höherer Ebene anbietet. Die Komplexität der Implementation unter Zuhilfenahme von tIATCanvas demonstriert die Wichtigkeit einer verbesserten technischen und konzeptuellen Basis.

Neben dem Mangel an Funktionalität fällt die prozedurale Schnittstelle der abstrakten Klasse tCanvas auf. tCanvas sammelt diverse einfache Methoden mit jeweils langen Parameterlisten und einigen Defaultwerten zur einfacheren Handhabung. Offensichtlich verfügt das Objekt über keinen Zustand, von dem die Methoden abhängig wären. Außerdem fehlen Zeichenprimitiven, die getroffene Auswahl läßt deutlich den Bezug zur ersten mit der Bibliothek entwickelten Anwendung Sane erkennen.

Dies setzt sich in der Implementation der Klassen tXCanvas und tPSCanvas fort, tPSCanvas implementiert einfach die Methoden nicht (bzw. leer), die Sane zum Ausdruck nicht braucht. Konzeptionell wichtiger ist aber die Entscheidung, im tXCanvas den Zeichenmodus XOR des X-Windows - Systems einzusetzen. Der Vorteil ist schnelles Löschen und Neuzeichnen der GFX-Objekte bei Manipulationen. Verschiebt der Benutzer ein Grafikelement, so wird dieses einfach an der alten Position neu gezeichnet - und damit die vorher dort befindlichen

Elemente wieder hergestellt - und dann an der neuen Position noch einmal gezeichnet[15]. Werden zwei identische Objekte gezeichnet, ist allerdings infolge XOR keines zu sehen.

GFX-Klassen und tIATCanvas unterscheiden an ihren Schnittstellen zeichnen und verstecken der Elemente. Bei der Ausgabe mittels XOR ergibt sich dies aber automatisch abwechselnd, falls nicht zwischenzeitlich die komplette Zeichenfläche gelöscht wurde. Da eine Verwaltung der Elemente in der Zeichenfläche fehlt und auch die GFX-Objekte ihre Sichtbarkeit nicht als Zustand halten können (zumindest nicht, wenn die Zeichenfläche gelöscht wurde nach einer Überdeckung), sind die Methoden Show und Hide irreführend. Implementiert werden beide identisch, die mit dem Namen angedeutete Wirkung ist nicht garantiert.

Aus diesen Nachteilen der alten Implementation läßt sich ein Forderungskatalog für die BibV30 erstellen, um den Ansprüchen zu genügen, folgt das neue Konzept einer bewährten Infrastruktur für graphische Editoren, nämlich dem Domänen-Framework Unidraw.

4.4.2 Vorstellung einiger Unidraw - Konzepte

Als Framework für die Domäne graphischer Editoren basiert Unidraw auf fortschrittlichen Konzepten im Bereich der Grafik und direkten Manipulation. Den Leistungsumfang habe ich in 3.2.2 im Überblick vorgestellt, ausgewählte Elemente erläutere ich nachfolgend detailliert, um deren Übernahme in die BibV30 vorzubereiten (siehe 4.4.3). Zum Application-Framework sei gesagt, daß in einer Anwendung beliebig viele Editoren mit jeweils mehreren Sichten auf das Material gleichzeitig geöffnet sein können. Nachfolgend stelle ich die Konzepte Grafik-Objekt (Component), direkte Manipulation und abschließend Verbindungen (Connector) vor.

Kern der Unidraw - Architektur sind Grafikobjekte, bezeichnet als „***Component***“ und (siehe Abbildung 4-5) entsprechend MVC aufgeteilt in „***Subject***“ und „***View***“. Der Component-View repräsentiert das Component-Subject als „Model“ im Sinne von MVC. Angenommen, das Domänen-Framework Unidraw wird zur Konstruktion eines graphischen Editors für Partituren eingesetzt, so sind die einzelnen Noten die Components. Während das Component-Subject der fachlichen Bedeutung einer Note entspricht (z.B. Tonhöhe, Tonlänge), sorgt der Component-View für deren graphische Präsentation.

[15] Anmerkung: Implementiert wurde ausschließlich Schwarz - Weiß - Grafik

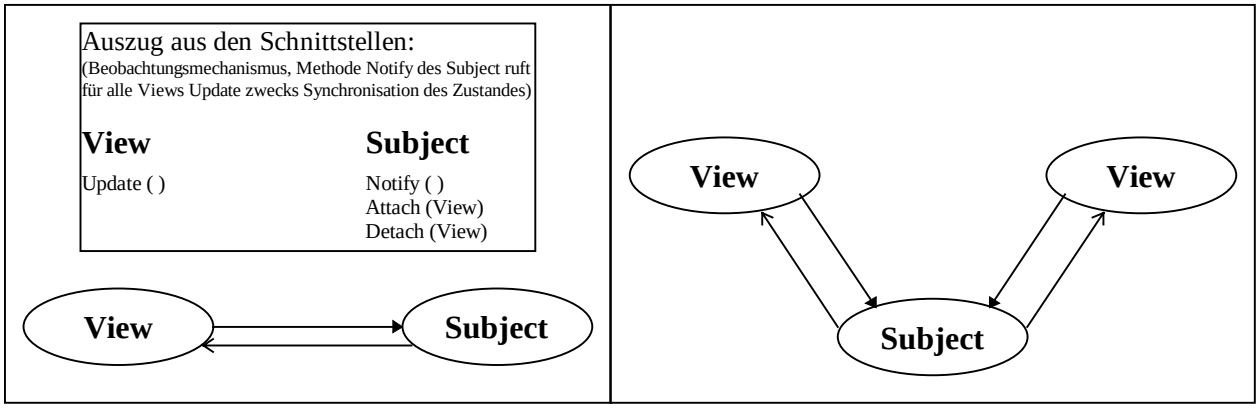


Abbildung 4 - 5: a) Struktur der Components: Subject und View; b) mehrere Views

Gemäß MVC- Vorbild gibt es mehrere Views zu einem Subject, beispielsweise stellt eine Sicht die komplette Partitur dar, eine andere nur wenige Takte eines einzelnen Instrumentes. Verändert nun der Benutzer Noten in einer der Sichten, so erfolgt diese Änderung im Subject und wird mittels eines Benachrichtigungsmechanismus allen Views bekanntgegeben. Alle Sichten zeigen somit immer ein konsistentes Bild. Neben graphischen Views für die Bildschirmanzeige gibt es externe Repräsentationen, die u.a. eine Postscript - Darstellung der Components erzeugen oder ein fachliches Format zwecks Speicherung erstellen. Im Beispiel könnte in dieser Form eine Ausgabe der Komposition über die Soundkarte stattfinden.

Komplexe Grafik- Elemente ergeben sich als ***Composite*** (Design Pattern, vgl. [GoF95]) aus anderen Objekten, im einfachsten Fall könnte z.B. ein Dreieck aus einzelnen Linien bestehen. Diese Komposition erfolgt hinsichtlich der Subjects wie auch der Views, ein Composite-View delegiert die Darstellung einfach an die enthaltenen elementaren Views (siehe Abbildung 4-6). Ein Benachrichtigungsmechanismus zwischen Composite und Komponenten erhält genau wie zwischen View und Subject die Konsistenz (vgl. Methoden Notify und Update, Abb. 4-5).

Gleichheit der Strukturen ist jedoch nicht vorgeschrieben, zwecks optimierter Darstellung

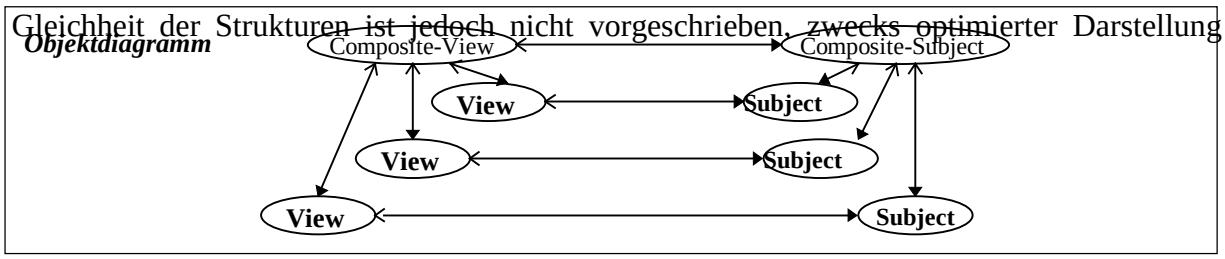


Abbildung 4 - 6: Composite - Components, gleiche Struktur von Subject und View

kann ein einziges, elementares Component-View für die Präsentation eines komplexen, aus vielen Komponenten bestehenden Composite-Component-Subject verantwortlich sein. Abbil-

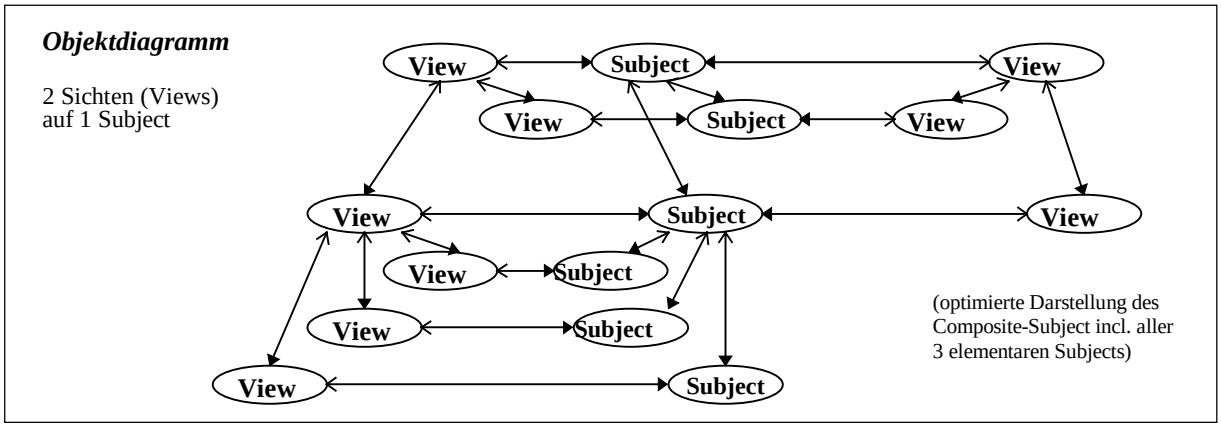


Abbildung 4 - 7: Composite-Component-Subject und Component-Views

dung 4-7 zeigt zwei Sichten auf ein Composite-Subject, links mit strukturgleichen Views, rechts mit einer optimierten Version für einen Teilbereich des Composites.

Components ermöglichen durch ihre Zerlegung gemäß MVC in Subject und View Sichten auf das graphische Material und durch Kombination elementarer oder bereits zusammengesetzter Objekte zu Composites den einfachen Aufbau komplexer Strukturen. Grafikeditoren arbeiten mit diesen Components, ermöglichen dem Benutzer die ***direkte Manipulation*** der Objekte. Das Unidraw - Framework wird um drei Komponenten ergänzt: Tool, Manipulator und Command. Jedem Editor im Application-Framework ist ein aktives ***Tool*** zugeordnet, ***Commands*** haben die gleiche Bedeutung wie in ET++, die zentrale Klasse ist der ***Manipulator***.

Abhängig vom aktiven Werkzeug (Tool) des Editors wird ein Manipulator - Objekt erzeugt. Das Werkzeug wendet sich eventuell an alle selektierten Views, die jeweils einen spezifischen Manipulator erzeugen. In diesem Fall faßt das Werkzeug selbst alle Manipulator-Objekte zu einem komplexen Manipulator zusammen, es kann aber auch selbst einen Manipulator erzeugen, ohne die selektierten Views mit einzubeziehen (siehe Abbildung 4-8).

Das Manipulator - Objekt empfängt nach der Aktivierung alle Benutzeraktionen, bis es selbst den Handlungsabschluß feststellt. Während der Interaktion ist der Manipulator für visuelles Feedback zuständig und vermittelt dem Benutzer den Eindruck direkter Manipulation. Nach Abschluß der Interaktion wird ein Command erzeugt, und zwar wieder vom aktiven Tool und ggf. unter Berücksichtigung der selektierten Views.

Ausführung der Commands und somit die Änderung der Component-Subjects veranlaßt das Application-Framework, genauer dessen Komponente „Viewer“, die Benachrichtigung aller Views entspricht der oben beschriebenen Synchronisation zwischen Subject und View. Ein kurzes Beispiel soll die Zusammenhänge veranschaulichen, Abbildung 4-8 zeigt den generellen Ablauf und wird durch das Beispiel erläutert.

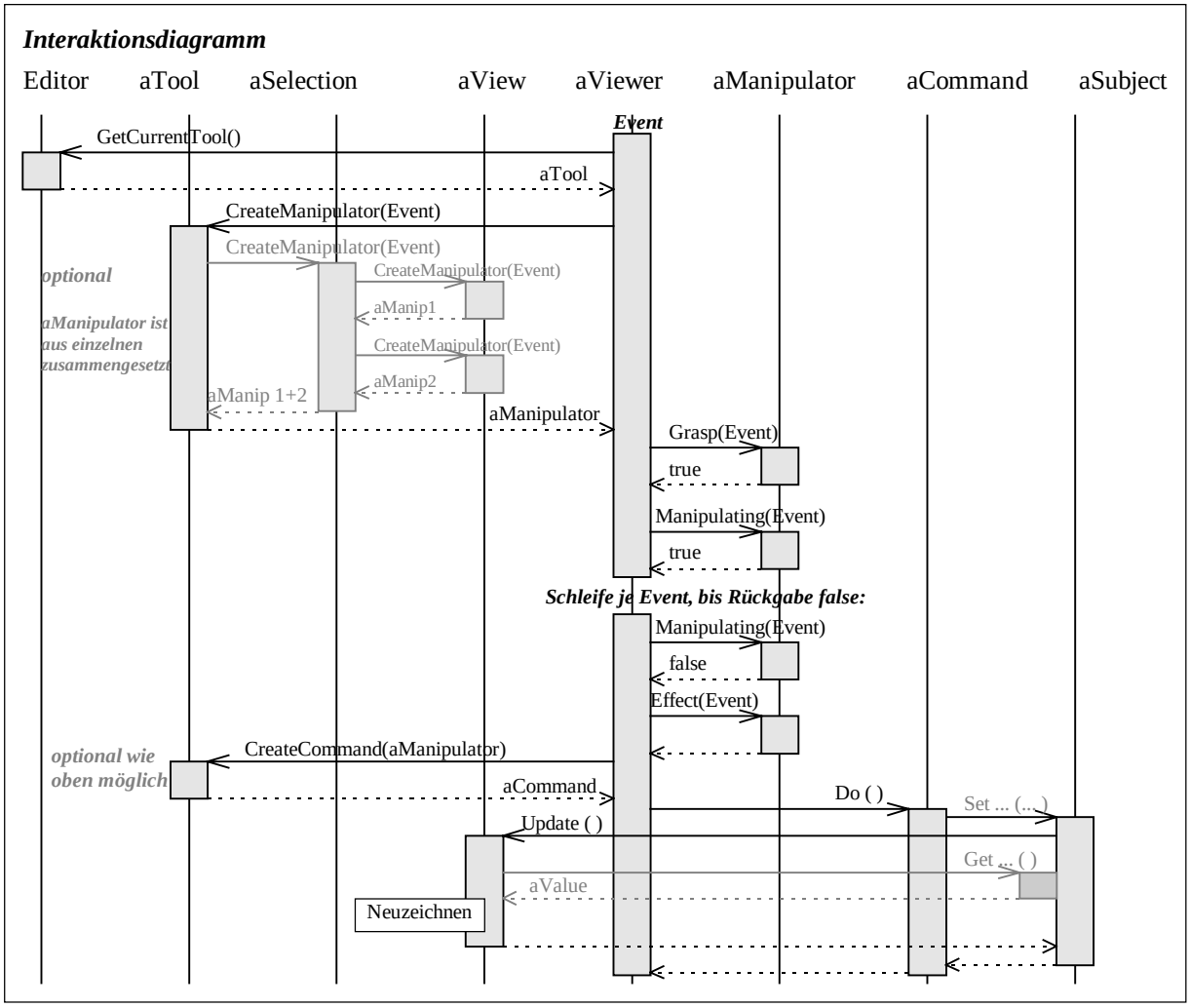


Abbildung 4 - 8: Ablauf der direkten Manipulation [Vlissides90]

Erläuterung zum Interaktionsdiagramm:

- senkrechte Linien stellen Objekte zur Laufzeit dar,
  die Zeitachse verläuft von oben nach unten
- Balken kennzeichnen die Phasen, in denen ein Objekt aktiv ist
- Horizontale Pfeile sind Methodenaufrufe, mit Name und Parametern beschriftet,
  gestrichelte Pfeile kennzeichnen den Kontrollrückfluß, ggf. Beschriftet mit Rückgabewerten
- ein 2. Balken am Objekt kennzeichnet einen Methodenaufruf, bevor die aktive Methode beendet wurde
- in einem Kasten am Objekt werden Methodenaufrufe innerhalb des Objektes vermerkt,
  diese können mit Namen der (Ober-) Klassen qualifiziert sein
- die grauen Bereiche in Abbildung 4-8 sind innerhalb des Ablaufs optional

- Abbildung 4-3 in Abschnitt 4.3: Doppelpfeile kennzeichnen Objekterzeugung,
  die Unterbrechung der senkrechten Linie unterscheidet zeitlich unabhängige Vorgänge

In einem universellen Grafikeditor verschiebt der Anwender selektierte Elemente mit der Maus. Die Anwendung besteht aus einem ***Editor*** und einem ***Viewer***, also einer Sicht auf das Material. Ausgangspunkt der Interaktion ist dieser Viewer, der die Benutzeraktion empfängt. Im Editor hatte der Anwender zuvor das Tool „Verschieben“ ausgewählt, verschiedene Elemente sind selektiert. Erste Aufgabe des Viewer - Objektes ist die Erzeugung eines Manipulators.

Dazu läßt sich der Viewer vom Editor das aktive Werkzeug geben und fordert von diesem den Manipulator an. Daß Werkzeug erhält als Parameter das Ereignis, also „Button Down“. Dieser einfache Fall setzt keine Mitwirkung der selektierten Views voraus, da die Aktion verschieben unabhängig vom Typ der Grafikelemente ist (der graue Bereich in Abbildung 4-8 entfällt).

In einer Abwandlung des Beispiels will der Anwender die Objekte nicht verschieben, sondern verformen. Diese Aktion wirkt sich je nach Typ des Elementes unterschiedlich aus, z.B. darf ein Kreis nur proportional vergrößert oder verkleinert werden. Das Tool trägt dem Rechnung, indem es sich von jedem selektierten View einen eigenen Manipulator geben läßt, und selbst einen speziellen Manipulator als Kombination kreiert (grauer Bereich in Abb. 4-8).

Der nächste Schritt ist in beiden Varianten des Beispiels die direkte Manipulation selbst. Die Methode ***Grasp*** bereitet den Manipulator vor, danach ruft der Viewer solange die Methode ***Manipulating*** auf, bis diese den Wert ***false*** zurück liefert. Innerhalb dieser Schleife geschieht die Benutzerinteraktion. Aufgabe des Manipulators bei diesem Vorgang ist eine Rückmeldung an den Benutzer und die Erkennung des Abschlusses der Handlung.

Entsprechend der Vorbereitung mit ***Grasp***, gibt es am Ende einen Methodenaufruf ***Effect***. Ein Abbruch der Interaktion ist während des Vorgangs jederzeit möglich, da noch keine Änderung am Component-Subject erfolgt, der Manipulator täuscht lediglich eine Veränderung der Views vor. Tatsächlich übergibt der Viewer das Manipulator - Objekt dem Werkzeug, damit dieses dazu passend ein Command erzeugt. Auch hier besteht wieder die Option, daß das Tool selbst dieses Command herstellt oder die Aufgabe teilweise an die selektierten Views delegiert (in der Abbildung wurde diese Variante nicht gesondert eingezeichnet).

Erst mit Ausführung des Commands, veranlaßt innerhalb des Application-Frameworks durch den Viewer, erfolgt die Beeinflussung des Subjects. Dieses teilt meldet die Zustandsänderung an seinen View (Methode ***Update***) und der View sondiert ggf. einige Werte des Subjects, bevor er sich neu zeichnet.

Als eine weitere Variante sei die gleichzeitige Ansicht der Grafik in mehreren Sichten (Viewer) genannt. Da die Manipulation in einer dieser Sichten erfolgt, ändert sich an der Beschreibung des Ablaufes nicht - während der Manipulation nimmt Unidraw eine Inkonsistenz der Sichten inkauf. Erst im allerletzten Schritt ergibt sich der Unterschied zum Interaktionsdiagramm, da das Subject nicht nur einen View, sondern alle benachrichtigt, sich

alle Views synchronisieren und neu zeichnen. Infolge dieser Trennung entsprechend MVC entsteht also für die direkte Manipulation bei mehreren Sichten kein zusätzlicher Aufwand.

Unidraw führt mit dem Manipulator - Objekt eine konkret verantwortliche Instanz ein, die dem Anwender direktes Feedback gibt. Oft sieht der Benutzer nicht das spätere Ergebnis, sondern eine spezielle Darstellung - z.B. nur Umrisse der Objekte an der neuen Position - in jedem Fall führt aber der Manipulator die Aktion nicht durch, sondern simuliert deren Auswirkungen nur. Gewissermaßen produziert der Benutzer durch seine Aktionsfolge mit interaktivem Feedback letztlich eine komplexe Anweisung, ein Command, das grundsätzlich auch als Kommando mit entsprechenden Parametern hätte eingegeben werden können.
Direkte Manipulation ist eine bequeme Form der Bedienung, aber keine neue Befehlsart !

Connectoren verknüpfen Grafikelemente in Unidraw zwecks Einhaltung von Vorschriften zur relativen Anordnung. Beispielsweise soll die Spitze eines Pfeiles mit einem Symbol auch nach dessen Bewegung verbunden bleiben, sich also mit bewegen. Ein anderes Beispiel wäre ein per direkter Manipulation einstellbarer Schieberegler, der nur im vorgegebenen Bereich beweglich ist (siehe Abbildung 4-9).

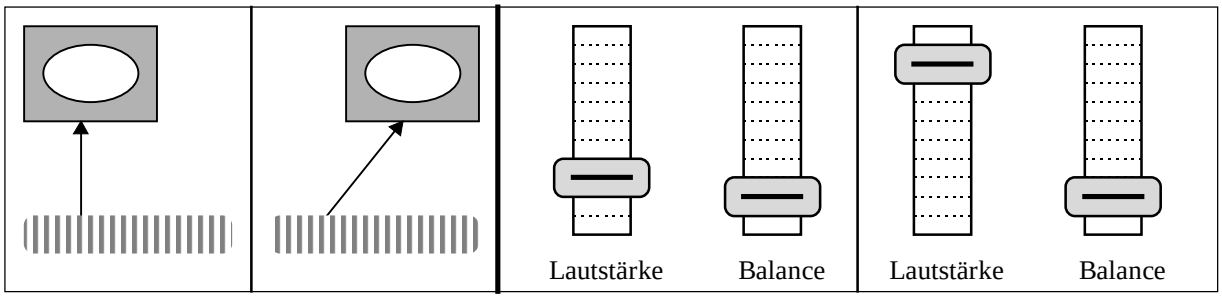


Abbildung 4 - 9: a) Pfeil verknüpft mit bewegtem Objekt; b) beschränkte Beweglichkeit eines Schiebereglers

Generell kennt Unidraw verschiedene Connectoren, die miteinander kombiniert Freiheitsgrade entsprechend Abbildung 4-10 ergeben. Soweit es sich nicht um eine feste Verbindung handelt (Punkt/Punkt), kann ein Objekt innerhalb des vorgegebenen Bereiches frei bewegt werden. Verläßt es diesen, muß das verbundene Objekt zwecks Einhaltung der Anordnungsvorschrift mit bewegt - oder die Bewegung über den Bereich hinaus verhindert werden. Der Parameter Mobility regelt das Verhalten, weitere Attribute der Objekte betreffen Größenänderungen - vgl. Abbildung 3-3 b): Minimal-, Normal- und Maximalgröße, Elastizität (siehe [Vlissides90]).

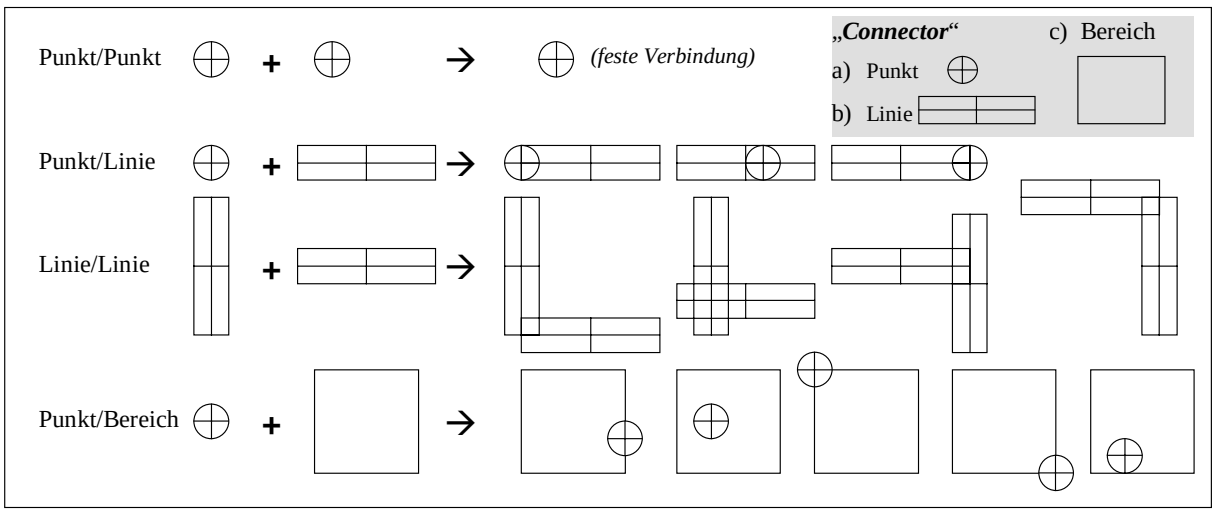


Abbildung 4 - 10: Connectoren beschränken die relative Beweglichkeit zweier Grafikelemente

4.4.3 Konzepte von Unidraw für WAM

Leitbilder und vorgesehene Anwendungsbereiche der BibV30 und des Domänen-Frameworks Unidraw unterscheiden sich deutlich, erklärtes Ziel ist jedoch die Übernahme und Integration bewährter Konzepte in das WAM-Framework, um graphische Komponenten in den mit der BibV30 entwickelten Anwendungen zu vereinfachen.

Klassen im Unidraw-Framework (4.4.2) lassen sich Metaphern nach WAM gegenüberstellen, bzw. auf die vorhandenen Grafikkomponenten (4.4.1) abbilden, teils sind Anpassungen nötig. Einige Konzepte finden in der BibV30 keine Berücksichtigung, da sie prinzipiell nicht in den Rahmen passen oder die angestrebten Ziele bereits durch andere Maßnahmen erreicht sind.

Abbildung 4-8 zeigt mit Editor, Viewer und Selection, Subject und View sowie Manipulator, Tool und Command die wesentlichen Klassen in Unidraw, bzw. deren Beteiligung an direkter Manipulation als zentrale Aufgabe des Frameworks. Das WAM-Framework ersetzt den Editor durch Werkzeuge mit der softwaretechnischen Konstruktion aus IAK und FK, die Selection als Menge ausgewählter Grafikelemente ist dem Werkzeug zugeordnet (in Unidraw dem Editor).

Viewer sind als Teil des Editors zu betrachten (stellen Sichten auf dessen Material dar). Bei Einschränkung auf die Funktionalität „Anzeige“ entspricht ein Viewer dem tIATCanvas, also der Zeichenfläche nach 4.4.1 - allerdings entsprechend dort genannter Kritik erweitert um die Verwaltung angezeigter Elemente (siehe unten).

Component-Subjects entsprechen dem (fachlichen) Material gemäß WAM, Component-Views sind dagegen Oberflächenelemente. Für deren Darstellung sind Interaktionstypen erforderlich, Unidraw überträgt jedoch die Aufgabe der Interaktion (also direkte Manipulation) an spezielle Manipulator-Objekte. Gemäß 4.4.1 lassen sich GFX-Objekte und Views vergleichen, beide ermöglichen eine Darstellung der graphischen Elemente.

Problematisch sind trotz aller Ähnlichkeiten die überwiegend wechselseitigen Beziehungen zwischen Unidraw - Klassen, eine Richtung entspricht meist dem Konzept der Beobachtung. WAM beschränkt die Verwendung des Mechanismus auf spezielle Komponenten (4.3.3), das Material als passive Komponente kann keine Änderungen melden. Composite-Subjects melden nicht nur nach außen, an ihre Views, die erfolgten Änderungen, sondern benutzen auch intern diesen Mechanismus, um das Composite mit seinen Bestandteilen zu synchronisieren.

Nicht akzeptabel in einem WAM-Framework wäre auch die direkte Verbindung zwischen Subject (also dem Material in der FK) und Views (in der IAK). Eine direkte Änderung des Materials durch die IAK - oder sogar durch IAT’s - ist ausgeschlossen, dies ist immer Aufgabe der Funktionskomponente.

Damit ergibt sich aber auch gleich eine passende Lösung für die 1 zu N Beziehung zwischen Subject und View, da WAM eine solche zwischen der FK und den Interaktionskomponenten ohnehin vorsieht. Statt mehrere IAT’s (entsprechend Views) direkt mit fachlichen Material (also Subject) zu verbinden, realisiert das Werkzeug verschiedene Sichten zwischen FK und IAK. Hier findet sich auch der Beobachtungsmechanismus wieder.

Eine Abbildung der Komponenten für die Grafik- Darstellung im Unidraw - Framework auf scheinbar ähnliche Konzepte gemäß WAM kann nicht über strukturelle Differenzen hinweg täuschen. Mit der direkten Kopplung zwischen Subject und View reduziert sich der große Einfluß von Composites auf die Anwendungen. Im WAM-Framework kann ein Werkzeug beliebiges Material bearbeiten, dieses muß nicht hierarchisch aufgebaut sein, also letztlich ein Composite bilden (der Unidraw - Editor arbeitet immer mit genau einem Composite-Subject). Umgekehrt werden auch GFX-Objekte als Ersatz für Views nicht zwingend zu Composites zusammengefaßt. Die Mächtigkeit dieses Konzeptes steht außer Frage, das Design Pattern kann auch im Bereich von WAM angewendet werden. Es ist hier nur keine Voraussetzung für graphische Anwendungen, sondern eine Option.

Völlig neu in der BibV30 und zentrale Motivation für die Beschäftigung mit einem graphischen Domänen-Framework ist das Konzept der direkten Manipulation, dort verwirklicht in Gestalt der Klassen Tool, Manipulator und Command. Genauer sind dies jeweils abstrakte Oberklassen für ganze Bäume konkreter Manipulatoren etc., die anwendungsspezifisch und passend zu den Components implementiert werden. Zusätzlich erweitern Connectoren das Konzept durch Beschränkungen zulässiger Benutzeraktivitäten.

Konzeptionell bedeutend ist die Trennung zwischen Manipulator und Command im Unidraw - Framework. Manipulatoren stellen gemäß ihrer Beschränkung auf visuelles Feedback und die Erzeugung der Parameter für ein komplexes Kommando (vgl. 4.4.2) eine Interaktionsform dar. Commands verändern das Material (Component-Subject), dies ist in einem WAM-Framework Aufgabe der Funktionskomponente. Problematisch ist insofern die Erzeugung des Commands durch den View, der als graphische Repräsentation Teil der Interaktionskomponente ist. Der Manipulator wie auch das Command - Objekt wird immer von der Klasse Tool erzeugt. Nach WAM gehört jedoch die Manipulation in die IAK, das Command in die FK.

Infolgedessen berücksichtigt die BibV30 das Konzept Tool nicht. [Vlissides90] argumentiert diesen Bestandteil seines Frameworks unter anderem mit der Abhängigkeit des Manipulators vom graphischen Material. Tool - Objekte erzeugen Manipulator - Objekte als Kombination mehrerer von den selektierten Views erstellter Manipulatoren (grauer Teil in Abbildung 4-8). Die GFX-Klassen der BibV30 übernehmen nicht die Aufgabe, für sich selbst Manipulatoren zu kreieren, die entsprechende Funktionalität der Tool - Klasse entfällt folglich.

WAM bietet mit der Werkzeug- Konstruktion aus IAK und FK ein bewährtes Schema für den Entwurf, die Interaktionskomponente übernimmt die Erzeugung von Manipulatoren. In der technischem Umsetzung (4.4.4) zeigt sich zudem eine größere Flexibilität der für die BibV30 gewählten Lösung, da die Annahme jeweils eines aktiven Tools entfällt.

Prinzipiell unvereinbar mit der Werkzeug und Material Metapher bzw. der softwaretechnischen Konzepte dieser Methode sind Commands, wie sie in Unidraw (oder auch ET++) vorliegen. Aufgabe dieser Objekte ist eine Änderung des fachlichen Materials, nach WAM muß dies in der FK geschehen. Erzeugt werden die Commands aber in der IAK, bei Beteiligung der Views sogar genau genommen in Interaktionstypen.

Commands als Design Pattern nach [GoF95] erleichtern die Konstruktion von Undo/Redo-Mechanismen, und eben dies ist die Motivation für ihren Einsatz in graphischen Editoren. Für einen Mechanismus im WAM-Framework gibt es zwei Alternativen, die unabhängig von den Grafikkomponenten als optionale Erweiterung der Anwendungsumgebungen realisierbar sind. Eine Lösung ist die Implementation der Funktionalität innerhalb der FK. Das Command Design Pattern ist dabei nur eine technische Variante, entscheidend ist Erweiterung der Schnittstelle der FK. Das Application-Framework kann zudem in der IAK, z.B. im Standard - Menü, die Funktionalität dem Anwender bereitstellen.

Alternativ ist die komplette Implementation innerhalb der IAK. Statt direkt Methoden der FK aufzurufen, erzeugt die Interaktionskomponente Command - Objekte, führt diese zu gegebener Zeit aus und realisiert mittels einer Liste Undo und Redo. Ein weiterer Vorteil ergibt sich aus der Möglichkeit, durch Commands „Makro“- Befehle zu schaffen: das Command - Objekt ruft eine Folge von Methoden der Funktionskomponente auf, die FK selbst bleibt unverändert.

Die BibV30 verzichtet vorläufig auf die Integration von Commands, die skizzierte Erweiterung ist von der Grafik unabhängig. Werkzeuge nach WAM übernehmen sowohl die Aufgaben der Commands, wie auch der Tool - Klassen. Diese Überlegungen isolieren das Unidraw - Konzept der Manipulator - Objekte als Kern der direkten Manipulation, als die Komponente, die in das WAM-Framework zu integrieren ist.

Manipulatoren simulieren die dem Benutzer das direkte Feedback seiner Interaktion, also die direkte Manipulation der Grafikelemente. Die Benutzerinteraktion wird festgehalten, aufgrund dieser Information aktiviert die IAK entsprechende Methoden der Funktionskomponente, der Manipulator selbst führt die Aktion nicht aus. Er stellt aber den Handlungsabschluß fest. Im Repertoire von WAM beschreibt dies exakt einen IAT, der daß Ergebnis seiner Berechnung an die IAK weiter reicht (vgl. 4.3.3).

Während der Manipulation stellen die Manipulator-Objekte das Zwischenergebnis oder dessen Abstraktion in Form von Umrissen in der Zeichenfläche dar, nehmen aber keine Änderungen der Grafik selbst vor. Ein Abbruch durch den Benutzer oder auch das Werkzeug ist möglich. Manipulatoren beachten Bedingungen, die sich aus Verbindungen zwischen Grafikelementen ergeben (Connectoren, umgesetzt als Eigenschaften der GFX-Objekte).

Zusammengefaßt erweitern Konzepte aus Unidraw die BibV30 um folgende Komponenten:

- Interaktionstyp tIATGraphic zur Anzeige eines GFX-Objektes auf der Zeichenfläche,
- Interaktionstyp tIATManipulator für direkte Manipulation,
- Connectoren als Eigenschaften der GFX-Objekte.

Erhalten bleiben aus dem alten Framework die Zeichenfläche tIATCanvas, deren technische Basis tCanvas mit tXCanvas und tPSCanvas und die GFX-Klassen. Editor, Viewer und Tool aus Unidraw gehen in die Werkzeuge ein, Commands wären eine optionale Ergänzung, nicht nur für die Grafik. Subjects finden sich nicht im Framework, wohl aber als fachliches Material konkreter Anwendungen.

4.4.4 Technische Umsetzung innerhalb des Frameworks

Konzepte aus Unidraw ergänzen die Grafik- Komponenten in der BibV30. Einzelheiten zu den konkreten Klassen dokumentiert das Benutzerhandbuch ([Weiß97], WWW - Dokument), im Text der Diplomarbeit drucke ich Klassenschnittstellen nur auszugsweise ab. Gegenwärtig hat das Mikro-Framework für graphische Komponenten innerhalb der BibV30 noch den Charakter eines Prototyps, insbesondere fehlt der Einsatz und Test im konkreten Projekt. Änderungen gegenüber dem hier dargestellten Stand sind somit zu erwarten, die grundlegenden Konzepte sollte dies aber nicht betreffen.

Aus der Diskussion in 4.4.3 ergeben sich verschiedene Bereiche eines Frameworks in der BibV30. Zum einen ist dies ein Interaktionstyp zur Präsentation graphischer Elemente (tIAT-Graphic), zum anderen die direkte Manipulation als neue Form der Interaktion (tIATManipulator). Die Portabilität der Interaktionskomponenten soll erhalten bleiben, ein Wechsel des Fenstersystems darf nur den Austausch der Interaktionstypen bedingen.

Dazu beschreiben anwendungsspezifische GFX-Klassen die Grafiken, tIATGraphic wird mit einem GFX-Objekt parametrisiert. Die abstrakte Klasse tCanvas garantiert die Unabhängigkeit vom Fenstersystem, indem diese ein Protokoll für elementare Zeichenoperationen definiert; GFX-Klassen zeichnen sich selbst mittels dieser Operationen. Die Darstellung der tIATGraphic - Objekte erfolgt innerhalb einer Zeichenfläche (tIATCanvas), die diese Objekte auch verwaltet und mit einem Fenstersystem-spezifischen Canvas verbindet (z.B. tXCanvas[16]).

Beginnen werde ich die Vorstellung neuer Klassen mit dem abstrakten Protokoll tCanvas und dessen Implementation tXCanvas. Die GFX-Klassen beschreibe ich zuerst hinsichtlich ihrer Bedeutung zur Darstellung graphischer Figuren (ohne Aspekte der Manipulation). Auf dieser Grundlage erläutere ich die Interaktionstypen tIATGraphic und tIATCanvas und diskutiere deren Abgrenzung gegenüber GFX-Klassen und tXCanvas. Ausgangspunkt der neuen Klassen sind deren Vorgänger im WAM-Framework, die ich in 4.4.1 vorgestellt habe, die erzielten Fortschritte hinsichtlich vorgebrachter Kritik hebe ich hervor.

Direkte Manipulation betrachte ich als zusätzlichen Mechanismus in der BibV30, dem ich mich nach den Konzepten zur Präsentation von Grafiken zuwende. Für die Integration der Neuerung erfolgen Anpassungen an den GFX-Klassen und am Interaktionstyp tIATCanvas.

Die abstrakte Oberklasse tIATManipulator stellt das abstraktes Konzept der Manipulatoren dar, konkrete Manipulatoren müssen abgeleitet werden, wie konkrete GFX-Klassen auch.

Abschließend untersuche ich die Eignung der vorliegenden Implementation für verschiedene Anwendungsfälle und den Bezug zu der in 4.3.3 genannten Trennung von Präsentation und Interaktion. Ich möchte damit weitere Diskussionen zu diesem Thema anregen, sicher aber nicht die Lösungen vorgeben.

---

[16] Implementiert sind Interaktionstypen und Canvas für X-Windows, in Arbeit befinden sich Klassen für wxWindows..

Darstellung graphischer Elemente

Als abstrakte Oberklasse definiert tCanvas Zeichenprimitiven für Linie, Pfeil, Polygon und Kreissegment sowie die jeweils eingeschlossenen Flächen. Anzugeben sind Koordinaten (in der Einheit Millimeter, also unabhängig von der Hardware - vgl. [Calder95]) und gegebenenfalls Winkel. Verschiedene Parameter sind als Zeichenmodus am Canvas - Objekt einzustellen: Farbe der Linien, Breite, Füllung und Muster sowie Form, Farbe und Größe von Pfeilspitzen. Im Vergleich zur Vorgängerversion weist die Schnittstelle (Ausschnitt siehe Abbildung 4-11) praktisch alle Methoden aus, um beliebige Figuren zu zeichnen.

Konzeptionell unterscheidet sich vor allem die Festlegung der diversen Parameter. Getrennte Methoden zu deren Einstellung ersetzen die extrem langen und inflexiblen Parameterlisten der Vorversion. Im alten WAM-Framework erhielt jede Zeichenoperation alle Parameter, davon waren zwecks vereinfachter Handhabung einige mit Default-Werten belegt. Der neu tCanvas hält den Zeichenmodus als seinen Zustand, verliert ansatzweise den Charakter einer schlichten Methoden - Sammlung ohne objektorientierte Elemente. Dieser Zustand wird gesetzt (Set...) und kann abgefragt werden (Get...). spezielle Methoden setzen mehrere Parameter gleichzeitig. Kommen später weitere Parameter hinzu, z.B. für dreidimensionale Figuren, so bedarf es nicht der Änderung sämtlicher Methoden in tCanvas und den abgeleiteten Klassen, es werden einfach Attribute und einstellende sowie abfragende Methoden ergänzt. Bieten abgeleitete konkrete Klassen die erweiterte Funktionalität nicht an, so ignorieren sie diese Parameter.

```
class tCanvas {
public:
    //Grafik Parameter für Linien
    virtual void SetLineThickness          (int _iThickness=1)    = 0;
    virtual int  GetLineThickness          ( ) = 0;
    virtual void SetLinePattern            (int _iPattern=100)    = 0;
    virtual int  GetLinePattern            ( ) = 0;
    virtual void SetLineColor              (tString _sColor=tString("black") ) = 0;
    virtual tString GetLineColor           ( ) = 0;
    ...
    //vereinfachtes Setzen mehrerer Parameter
    virtual void SetLineAttributes (int _iThickness, int _iGraytone, eArrow _eArrow = NO_ARROW) = 0;
    virtual void SetLineAttributes (int _iThickness, char _cDash, tString _sColor, eArrow _eArrow)    = 0;
    //Grafik Parameter für Flächen
    virtual void SetInteriorPattern(int _iPattern=0) = 0;
    ...
    //Zeichnen (X, Y: linke obere Ecke)
    virtual void ClearWindow    ( ) = 0;
    virtual void DrawRectangle (long _X, long _Y, long _Width, long _Height)          = 0;
    virtual void FillRectangle  (long _X, long _Y, long _Width, long _Height)          = 0;
    virtual void DrawEllipse    (long _X, long _Y, long _Width, long _Height)          = 0;
    virtual void FillEllipse    (long _X, long _Y, long _Width, long _Height)          = 0;
    virtual void DrawSector     (long _X, long _Y, long _Width, long _Height, int _Alpha, int _Beta) = 0;
    virtual void FillSector     (long _X, long _Y, long _Width, long _Height, int _Alpha, int _Beta) = 0;
          //_Alpha: Winkel zwischen X-Achse und Sektor-Beginn; _Beta: Sektor-Groesse
    virtual void DrawLine       (long _X1, long _Y1, long _X2, long _Y2)   = 0;
    virtual void DrawPolyLine   (tOrdCol<tCanvasPoint*> *_pNodeList)       = 0;
    virtual void FillPolygon    (tOrdCol<tCanvasPoint*> *_pNodeList)       = 0;
    virtual void DrawArrowhead(long _X1, long _Y1, long _X2, long _Y2)   = 0;
    ...
};
```

Abbildung 4 - 11: Schnittstelle tCanvas, Ausschnitt

Technisch zeichnet sich die neue Klasse tXCanvas neben einigen signifikanten Optimierungen vor allem durch die Unterstützung von Farben aus. Im X-Windows GUI- System ergeben sich durch begrenzte Farbpaletten verschiedene Probleme, abhängig von der Hardware, die eine gegenüber der Oberklasse erweiterte Schnittstelle explizit behandelt (vgl. Ausschnitt der Schnittstelle in Abbildung 4-12). Eine Änderung der abstrakten Klasse tCanvas entfällt, da die Probleme nur den Interaktionstyp betreffen, und dieser ohnehin vom konkreten Fenstersystem, und damit von der konkreten Klasse tXCanvas abhängt.

```
class tXCanvas : public tCanvas {
    public:
    tXCanvas                (Widget _xWidget);
    virtual ~tXCanvas       ( );

    //Hintergrund der Zeichenfläche setzen, verschiedene Varianten
    virtual void SetBackground          (char _cBitmap[], int _iWidth, int _iHeight);
    virtual void SetBackground          (tString _sColor, tString _sBitmapColor = tString("black"));

    //X-spezielle Schnittstelle fuer Entwicklungszwecke
    virtual int  GetDrawingMode ( );
    virtual bool IsDrawingMode  (int _iMode = GXcopy);
    virtual void SetDrawingMode(int _iDrawingMode); //X-Funktion, z.B. GXcopy zeichnet, GXxor fuer "Move"

    virtual const char** GetColorStrings( );
    virtual bool IsColorString  (tString _sColor);
    virtual bool IsColorInMap   (tString _sColor);
    virtual tCollection<tString*>* GetColorsInMap ();
    //Die Farboption setzt freie Colormap-Entries voraus. An Rechnern des Arbeitsbereichs bewirkt die Reservierung
    //solcher Einträge oft die Meldung "Cannot allocate Colormap ...", z.B. bei Nutzung vieler Netscape - Fenster.
    //GetColorStrings liefert eine Liste bekannter Farben; IsColorString prueft, ob eine Farbe bekannt ist.
    //Dagegen versucht IsColorInMap, diese Farbe zu reservieren - soweit Eintrag in Colormap noch fehlt.
    //Achtung: durch diesen Seiteneffekt wird ein Eintrag belegt, der auch nicht wieder freigegeben werden kann!
    //GetColorsInMap versucht, alle Farben gemäß GetColorStrings zu belegen, und gibt die effektiv benutzbaren
    //zurueck. Achtung: danach sind alle Farben belegt, auch für andere Programme !

    //Implementation der abstrakten Oberklasse
    virtual void SetLineThickness       (int _iThickness=1);
    ...
    virtual void ClearWindow    ( );
    virtual void FillRectangle  (long _lX, long _lY, long _Width, long _Height);
    ...
};
```

Abbildung 4 - 12: Schnittstelle tXCanvas, Ausschnitt

Übernommen aus der Vorversion findet sich die Möglichkeit, Flächen und auch Linien mit Bit - Mustern zu hinterlegen. Dies kann eine spezielle 2-Farb-Bitmap sein, früher wurde die Option vor allem auch für Grautöne benutzt (daher Variante mit Angabe in Prozent - es gab nur eine Schwarz/Weiß- Ausgabe). Die Beschränkung der Bitmap auf zwei Farben bleibt bestehen, diese müssen aber nicht gerade Schwarz und Weiß sein.

GFX-Klassen stehen für graphische Formen. Eine Auswahl üblicher Formen für allgemeine Grafikeditoren wäre z.B. tGFXLine, tGFXRectangle und tGFXEllipse, aber auch Polygone und Kantenzüge will der Anwender zeichnen. Quadrate und Kreise lassen sich als Sonderfälle von Rechtecken und Ellipsen verstehen - dagegen werden Rechtecke in bekannten Systemen nicht einfach als spezielle Polygone gesehen. Erst im Zusammenhang mit der Manipulation dieser Formen ergeben sich semantische Unterschiede (s.u.).

Soll die Software Statistiken visualisieren, so werden u.a. GFX-Klassen für Kreisausschnitte bzw. Segmente erstellt, um „Tortendiagramme“ zu präsentieren. CAD - Systeme benötigen je Anwendungsgebiet spezifische Formen, die GFX-Klassen sind also immer spezifisch für die konkrete Applikation oder Domäne. (Abbildung 4-13 zeigt Beispiele)

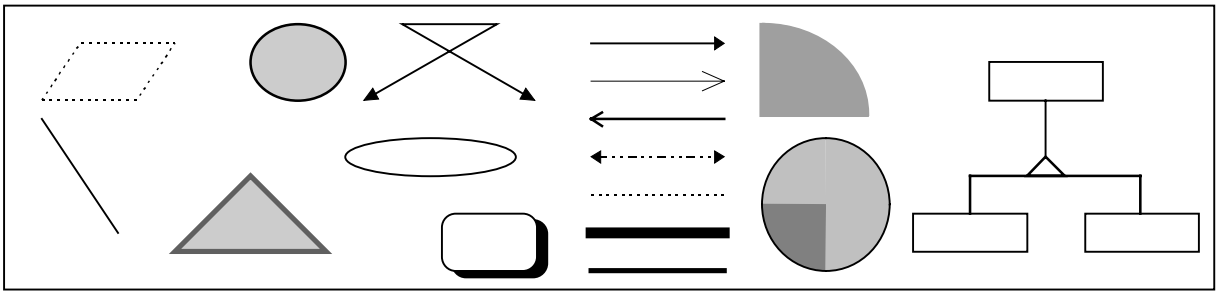


Abbildung 4 - 13: Beispiele für GFX - Figuren, teilweise durch konkrete Klassen realisiert

Aufgabe der GFX-Objekte ist es, die entsprechende Grafik auf einem tCanvas zu zeichnen. Dies schließt die Bildschirmanzeige mittels tXCanvas (oder tOS2Canvas, tMacCanvas, etc.) ein, aber auch den Ausdruck über z.B. tPSCanvas für Postscript. Durch die Beschränkung auf Methoden der abstrakten Oberklasse tCanvas bleiben die in der IAK benutzten GFX-Klassen und damit die Interaktionskomponente vom Fenstersystem unabhängig.

Für den Anwender erscheint die Grafik am Bildschirm als Interaktionstyp, meist kann er mit der Maus durch anklicken eine Selektion vornehmen, evtl. auch ein objektbezogenes Menü aufklappen. Objekte vom Typ tIATGraphic stellen die Verbindung zwischen einem tCanvas und dem GFX-Objekt her, mit dem sie parametrisiert sind. Als Interaktionstyp sind sie über das Command - Design Pattern mit der IAK verbunden (vgl. 4.3.3 bzgl. IATMotif - Bibliothek).

Schließlich repräsentiert tIATCanvas die Zeichenfläche, also einen Bereich auf dem Bildschirm, innerhalb dessen tIATGraphic - Objekte angezeigt werden. Aufgabe des Interaktionstyp ist die Verwaltung der in ihn eingefügten Objekte (Behälterfunktionalität), die Schnittstelle bietet Methoden zum Einfügen und Entfernen. Grafiken liegen logisch übereinander auf einzelnen Ebenen, später eingefügte überdecken früher vorhandene. Diese Reihenfolge läßt sich durch die IAK beeinflussen, in den meisten Grafikeditoren ist es üblich, diese Funktionalität dem Anwender z.B. durch spezielle Schaltflächen oder Menüoptionen anzubieten.

Gegenüber der früheren Version des Grafik-Frameworks ergibt die Lösung zweier Probleme (vgl. 4.4.1). Besonders wichtig ist die Verwaltung der Grafik- IAT’s durch den tIATCanvas. Während früher die Interaktionskomponente jederzeit in der Lage sein mußte, den Inhalt der Zeichenfläche wieder herzustellen - z.B. nach Überdeckung des Fensters - übernimmt diese lästige Aufgabe jetzt der IAT. Dies ermöglicht auch eine Optimierung des Vorgangs, da der tIATCanvas Objekte nicht anzeigen muß, die außerhalb des sichtbaren Ausschnitts liegen. Bei dieser Besserung ist anzumerken, daß der Vorteil nicht nur technisch motiviert ist. Es handelte sich auch um einen konzeptionellen Bruch gegenüber allen anderen Interaktionstypen, die auch im alten Framework ohne Mitwirkung der IAK wiederhergestellt wurden.

Das zweite Problem ergibt sich aus dem Zeichenmodus XOR der alten Zeichenfläche. Dort gab es keine Ebenen, die Objekte wurden nicht in logischer Reihenfolge übereinander gezeichnet, sondern alle Objekte waren in derselben Ebene gleichzeitig zu sehen. Identische Objekte zeigte die alte Zeichenfläche überhaupt nicht an, da sich diese gegenseitig auslöschten.

Notwendig ist die Verwendung dieses Modus unter X-Windows, um schnelle Bewegungen optisch akzeptabel anzuzeigen. Mechanismen wie „Double-Buffering“ schaffen zwar Abhilfe gegen den „Flimmer-Effekt“, dafür erscheinen die Bewegung aber ruckartig, wenn eine sehr große Anzahl an Elementen ständig neu gezeichnet wird. Mit der gegebenen Hardware kam es zu Situationen, da die Ereignisse der Maus - Bewegung nicht mehr zeitgerecht verarbeitet wurden. Eine Kombination aus beiden Mechanismen, Zeichenmodus XOR und gleichzeitig „Double-Buffering“, erwies sich als optimale Lösung für die Anzeige bewegter Objekte.

Im Framework gab es keinen Begriff der direkten Manipulation und somit auch keine explizite Unterstützung hierfür. Infolgedessen übernahm die IAK diese Aufgabe und benutzte für das visuelle Feedback während der Benutzerinteraktion dieselben Methoden zur Darstellung der graphischen Elemente, wie für die statische Anzeige des Ergebnisses nach der Ausführung. Somit konnte grundsätzlich nur im Modus XOR gezeichnet werden, um direkte Manipulation nicht völlig auszuschließen.

Die BibV30 führt Manipulatoren als eigene Interaktionstypen ein (siehe unten), und trennt so die Darstellung graphischer Elemente von deren Manipulation. Nur für letzteres bestehen die genannten Probleme, nur während der Bewegung von Elementen trifft die Argumentation für den Zeichenmodus XOR zu. Interaktionstypen für die Darstellung der Grafikobjekte, also tIATGraphic, können dem in Editoren verbreiteten Prinzip übereinander liegender Elemente folgen, diese können sich ganz oder teilweise überdecken - aber nicht gegenseitig auslöschen.

Für den Interaktionstyp tIATManipulator besteht die Alternative, den Modus XOR weiterhin zu benutzen, und zwar gezielt dann, wenn dies notwendig erscheint. Dazu ist anzumerken, daß die Oberklasse tCanvas von diesem Modus freigehalten wird, ohnehin GUI- spezifische IAT’s benutzen direkt die Schnittstelle der konkreten Klasse, in diesem Fall tXCanvas.

Direkte Manipulation

Anwender erwarten von Grafikeditoren eine intuitive Bedienung. Zwar handelt es sich meist um einfache Aktionen, die auch als Kommando über die Tastatur aktiviert werden könnten (z.B. „verschiebe das Dreieck um 2 cm nach links“), anschaulicher ist aber die Bedienung des Systems mit der Maus. Der Anwender zeigt mit der Maus auf das Element und verändert es, während dieser Aktion zeigt ihm das System jeweils die Wirkung seines soweit eingegebenen Befehls an (im Beispiel muß die Strecke von 2 cm nicht vorher ermittelt werden, vielmehr wird das Dreieck solange immer weiter nach links verschoben, bis es sich an der richtigen Position befindet). Dieses Konzept der direkten Manipulation verwirklicht die BibV30 in Anlehnung an Unidraw unter Verwendung des Interaktionstypen tIATManipulator.

An dieser Stelle sei darauf hingewiesen, daß die vorhergehenden Ausführungen betreff der Darstellung graphischen Materials von dessen Manipulation unabhängig sind. Die nachfolgend genannten Klassen und Erweiterungen bereits genannter Klassen müssen dazu nicht benutzt werden. Auch ein Grafikeditor, der tatsächlich nur eine Kommando- Schnittstelle kennt, wird die Manipulatoren nicht verwenden. Entsprechend den in 4.3 aufgestellten Anforderungen an die BibV30 handelt es sich um einen zusätzlichen, optionalen Mechanismus.

Nach diesem kleinen Ausflug komme ich zurück zur technischen Beschreibung. Im einzelnen gibt es eine kleine Anpassung der Klasse tIATGraphics, wesentliche Erweiterungen betreffen tIATCanvas und die GFX-Klassen. Kern des Konzeptes ist die genannte abstrakte Oberklasse tIATManipulator und deren konkreten Implementationen für spezifische Anwendungen.

Manipulatoren haben die Aufgabe, dem Benutzer während der Interaktion direktes Feedback zu geben, den Abschluß der Handlung festzustellen bzw. einen Abbruch zuzulassen und stellen für die tatsächliche Änderung der graphischen Komponenten erforderliche Informationen über die Interaktion bereit. Die Auswertung dieser Informationen und somit die Durchführung der Interaktion ist Aufgabe der Interaktionskomponente eines Werkzeuges.

Diese IAK erzeugt auch die Manipulator-Objekte, gemäß 4.4.3 entfällt im WAM-Kontext die Unidraw - Klasse Tool. Abbildung 4-14 zeigt die Schnittstelle der Klasse tIATManipulator, in Verbindung mit den Aufgaben anderer Komponenten ergibt sich deren Bedeutung.

```
class tIATManipulator : public tIATBase {
    public:
    virtual void Show( );

    void CancelManipulation  ( );        //nicht virtual ! in abgeleiteten Klassen  NICHT  überschreiben !
    void CommitManipulation  ( );
    bool IsActivated         ( );

    //Vorbereitung der direkten Manipulation, falls zustaendig: setze IsActivated( )
    virtual void PrepareEvent  (int _iKey, int _iState) { };                          //Keyboard-Event
    virtual void PrepareEvent  (int _iButton, int _iState, long _lX, long _lY) { };   //Handle ?
    virtual void PrepareEvent  (int _iButton, int _iState, bool _bHit) { };           //Mouse-Event

    virtual void SetGraphics   (tExtOrdCol<tIATGraphic*>* _pGraphics) { };
    virtual void SetCanvas     (tXCanvas* _pCanvas) { };

    //solange IsActivated() gueltig ist, gelangen alle Events zum aktiven Manipulator
    virtual void AcceptEvent   (int _iKey, int _iState) { };                                    //Keyboard-Event
    virtual void AcceptEvent   (int _iButton, int _iState, long _lX, long _lY, bool _bHit) { }; //Mouse (Button Down)
    virtual void AcceptEvent   (int _iButton, int _iState, long _lX, long _lY) { };             //Mouse (ButtonUp)
    virtual void AcceptEvent   (int _iState, long _lX, long _lY) { };                           //Mouse-Event (Move)
};
```

Abbildung 4 - 14: Schnittstelle tIATManipulator, Ausschnitt

Zur Aufgabe des tIATCanvas gehört neben Verwaltung und Anzeige der Grafik- IAT's die Verwaltung und Aktivierung von Manipulatoren. Die IAK kann ein oder mehrere Manipulator-Objekte in den Canvas einfügen, alle oder einzelne wieder entfernen. Verwaltet ein Canvas mehrere Manipulatoren, so entscheiden diese selbst, ob sie für jeweilige Ereignisse zuständig sind. Im wesentlichen hängt dies von dem benutzten Maus - Knopf ab, evtl. von gleichzeitig gedrückten Modus-Tasten (Alt, Shift, ...). Flexibilität bringt tCanvasMousePolicy, eine Strategieklasse für ggf. auch zur Laufzeit wechselnde Belegung (z.B. für Linkshänder).

Konkret ruft tIATCanvas nacheinander die Methode PrepareEvent aller Manipulator-Objekte, bis sich einer aktiviert (IsActivated). Das Ereignis und alle folgenden leitet der Canvas dann per AcceptEvent an den Manipulator, bis dieser den Handlungsabschluß erkennt und sich in den Zustand „nicht - aktiviert“ versetzt.

Grafikeditoren beinhalten oft drei Arten von Manipulatoren. Die erste Art sind Manipulatoren, die nicht für bestimmte bestehende Objekte aufgerufen werden. Gemeint sind z.B. Lassos, aber auch die Erzeugung neuer Elemente. Der Anwender betätigt die Maus an beliebiger Position der Zeichenfläche oder konkret an einer Stelle, an der sich bisher kein Element befindet. Den Kontrast bildet die zweite Art der Manipulatoren mit direktem Bezug zu bestehenden Grafiken, z.B. um diese zu verschieben, wird der Anwender einfach auf das Objekt klicken.

Als dritte Art ergeben sich Manipulatoren, die an bestimmten Punkten gekennzeichnet sind und dort aktiviert werden. Üblich sind vier oder acht Punkte um die selektierten Objekte, mit denen Verformung bzw. an den Ecken oft proportionale Größenänderungen durchzuführen sind. Es gibt auch „Punkte“ für Drehung von Objekten um ihren Mittelpunkt oder um frei definierbare Punkte, die ebenfalls per direkter Manipulation dieser „Handle“ bestimmt werden. Fast nur in dieser speziellen Form lassen sich frei gezeichneter Linienketten verformen, hier erhält jeder Berührungspunkt zweier Linien einen solchen Punkt zum „Anfassen“.

Unterschiede dieser dritten Art von Manipulatoren gegenüber den anderen beiden betreffen sowohl Darstellung als auch Handhabung. Die genannten Punkte oder Handle gehören nicht zu den Grafikobjekte, sondern zum Manipulator. Während ein Lasso als Interaktionstyp solange nicht sichtbar in Erscheinung tritt, bis der Benutzer diesen Manipulator aktiviert, ist der Handle immer anzuzeigen. Aus diesem Grund verfügt die Klasse tIATManipulator über eine Methode Show, die per Default leer implementiert ist.

Nach dem Zeichnen aller tIATGraphics ruft tIATCanvas für alle eingefügten tIATManipulator - Objekte die Methode Show auf. Somit liegen die Handle in einer zusätzlichen Ebene oberhalb alle Grafiken. Abbildung 4-15 a) zeigt sichtbare Punkte für die Verformung eines teilweise überdeckten Objektes. Eben diese Eigenart wirkt sich auf die Handhabung aus: obwohl das Grafikelement an seiner unteren rechten Ecke nicht sichtbar ist, kann es mittels des Handle doch vergrößert werden - Abbildung 4-15 b).

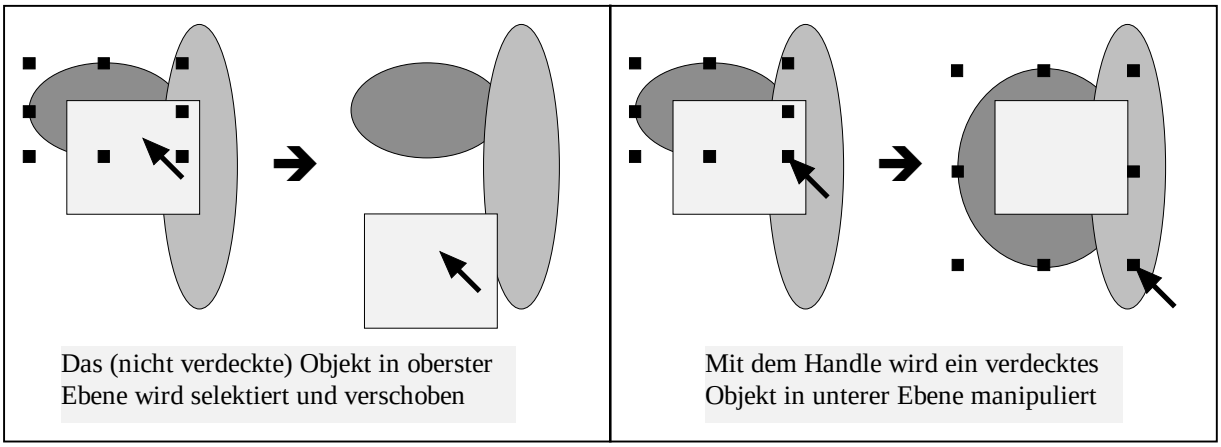


Abbildung 4 - 15: a) Objekt mit normalem Manipulator bewegen; b) Vergrößern mit Handle - Manipulator
(zu a: der Move-Manipulator zeigt keine Handle an, die Handle im linken Teil verschwinden beim Drücken der Maus - Taste)

An feste Positionen gebundenen Manipulatoren haben Vorrang vor anderen, im Beispiel von Abbildung 4-15 läßt sich eine selektierte Figur mittels des Handle vergrößern, die im diesem Bereich von einem anderen Objekt überdeckt wird. Um dies zu erreichen, ist der beschriebe Ablauf der Aktivierung eines Manipulators zu erweitern. Dazu gibt es zwei Methoden namens PrepareEvent in der Schnittstelle von tIATManipulator (Abbildung 4-14), eine gilt für Handle und übermittelt dem Manipulator die Position des Mauszeigers - der Manipulator zeichnet die selbst die Punkte und kennt folglich deren Lage.

tIATCanvas durchläuft erst eine Schleife, um diese Methode für alle Manipulatoren aufzurufen, und dann eine zweite zur Aktivierung „normaler“ Manipulatoren. Wie oben beschrieben bricht die Schleife (bzw. eine von beiden) ab, sobald sich ein tIATManipulator - Object zuständig fühlt und aktiviert. Der weitere Ablauf, also die Weiterleitung aller folgenden Ereignisse bis zur Deaktivierung, ist unabhängig vom Typ des Manipulators.

Zusammengefaßt haben Manipulatoren folgende Aufgaben:

- ggf. Darstellung von Punkten oder Symbolen für die Aktivierung (nur „Handle“)
- Entscheidung, ob sie für ein Ereignis zuständig sind in zwei Stufen (Handle und Normal)
- Bearbeitung einer Kette von Ereignissen durch Simulation deren Wirkung
- Feststellung des Handlungsabschlusses
- Meldung der erfolgten Ausführung einer Interaktion an die IAK

Die zusätzlichen Aufgaben des tIATCanvas bezüglich direkter Manipulation sind Verwaltung und Aktivierung dieser Objekte, Weiterleitung aller folgenden Ereignisse ab der Aktivierung bis zum Abschluß der Interaktion, und der Abbruch dieser Handlung aufgrund spezieller Aktion des Benutzers. Eine gute Praxis ist die Belegung einer speziellen Taste, z.B. Escape, um jede Manipulation abbrechen zu können. Ähnlich der Klasse tCanvasMousePolicy wird auch diese Tastenbelegung variabel gestaltet, tCanvasKeyboardPolicy ermöglicht die Festlegung spezieller Tasten für Abbruch sowie vorzeitigen Abschluß der direkten Manipulation (vgl. [Weiß97]).

Abbildung 4-16 zeigt die Klassenschnittstelle tIATCanvas. Es können tIATGraphic - Objekte eingefügt und entfernt werden, ebenso Manipulatoren, abgeleitet von tIATManipulator. Der Canvas übernimmt für tIATGraphic Fabrik- Funktionalität, um diese IAT’s mit der konkreten Zeichenfläche (ein Objekt vom Typ tXCanvas) zu verbinden. Die sonstigen bereits erläuterten Aufgaben in Kurzform:

- Anzeige der eingefügten IAT’s (tIATGraphic und tIATManipulator)
- Wiederherstellung der Anzeige, wenn vom Fenstersystem gefordert
- Änderung der Anzeigereihenfolge von tIATGraphic - Objekten (Ebenen)
- Suche des verantwortlichen Manipulators in zwei Durchläufen für Handle und andere
- Abarbeiten einer Kette von Events durch Weitergabe an den aktiven Manipulator
- ggf. Abbruch der Interaktion oder vorzeitige Bestätigung gemäß Strategieobjekt

```
class tIATCanvas : public tIATBase {
    public:
    ...
    virtual void InitMouse      (tCanvasMousePolicy* _pMouse);      //Strategieklasse für Mouse-Belegung
    virtual void InitKeyboard   (tCanvasKeyboardPolicy* _pKey);     //Strategieklasse für Tastatur-Funktionen
    //Graphics & Manipulator
    virtual void InsertGraphic  (tIATGraphic* _pIAT);
    virtual void RemoveGraphic(tIATGraphic* _pIAT);
    virtual void InsertGraphics (tOrdCol<tIATGraphic*>* _pGraphics);
    ...
    virtual void InsertManipulator      (tIATManipulator* _pIAT);
    virtual bool IsManipulator          (tIATManipulator* _pIAT);
    virtual void RemoveManipulator      (tIATManipulator* _pIAT);
    ...
    virtual void ShowSelection ( );
    virtual void HideSelection ( );
    //Fabrikmethoden, Reihenfolge der tIATGraphics
    virtual tIATGraphic* NewIATGraphics (tGFXDrawable* _pGFX);
    virtual void MoveTop   (tIATGraphic* _pIAT);    //Darstellung in oberster Ebene, über allen anderen
    ...
};
```

Abbildung 4 - 16: Schnittstelle tIATCanvas, Ausschnitt

Grafik- IAT's verfügen über eine minimale Schnittstelle (Abbildung 4-17). Selektion und deren Aufhebung sind mit Commands verbunden, für die sich die IAK registrieren kann. Direkt im Konstruktor wird die Verbindung mit einem GFX-Objekt hergestellt, die IAK verwendet dazu die Fabrikmethode des tIATCanvas (Abb. 4-16). Ansonsten merkt sich der IAT seine Position und Größe, Änderungen der Form werden direkt am GFX-Objekt vorgenommen.

```
class tIATGraphic : public tIATBase {
    public:
    tIATGraphic                 (tGFXDrawable* _pGFX);          //Konstruktor
    virtual ~tIATGraphic ( );                                   //Destruktor
    virtual tIATGraphic* Clone  ( );                            //"tiefe" Kopie
    virtual void RegisterCommand(tSelectCommandBase* _pCmd);    //"selektiert", kein Parameter
    virtual void RegisterCommand(tUnselectCommandBase* _pCmd);  //"deselektiert", kein Parameter
    //fuer Manipulatoren und IATCanvas
    virtual tGFXDrawable* GetDrawable ( );
    virtual void Update ( );    //meldet Änderungen am GFXDrawable
    virtual void Select ( );
    virtual void Deselect ( );
    virtual bool IsSelected ( );
    virtual int  GetSizeX ( );
    virtual int  GetLeft ( );
    ...
};
```

Abbildung 4 - 17: Schnittstelle tIATGraphic, Ausschnitt

Update muß nach jeder Änderung am GFX-Objekt aufgerufen werden, damit die Angaben zur Position und Größe synchronisiert werden. Im Gegensatz zum Vorbild Unidraw verzichtet die BibV30 hier aber auf wechselseitige Benutzung, das GFX-Objekt hält keinen Verweis auf den IAT, zu dem es gehört. Die ändernde Komponente ist für den Aufruf der Methode Update selbst zuständig, kann somit auch den Zeitpunkt bestimmen, da die Änderung ausgewertet wird. Beispiele in 4.4.5 und vor allem [Weiß97] machen hiervon Gebrauch.

4.4.5 Anwendung der Grafik-Frameworks

Welche Arten von Anwendungen unterstützt das Mikro-Framework für Grafik in der BibV30 konkret? Wie läßt sich der vorgesehene Einsatzkontext beschreiben, für welche Applikationen sollten die Konzepte besser nicht benutzt werden? Passen die neuen Interaktionstypen in die IATMotif - Bibliothek, besteht ein Bezug zur Diskussion zur Trennung von Interaktionsform und Präsentation?

Anhand diverser Beispiele versuche ich, Antworten auf diese und weitere Fragen zu geben, unklare Bereiche aufzuzeigen und die Diskussionen über diesen Beitrag zur C++ Bibliothek einzuleiten. Mein Anliegen ist es, die Benutzung des Frameworks zu motivieren, um dessen Tauglichkeit im praktischen Gebrauch und die Grenzen der zugrunde gelegten Konzepte zu testen und bewerten.

Vom Malprogramm bis zum CAD-System ?

Weder das Eine noch das Andere liegt im üblichen Einsatzkontext des WAM-Frameworks. Im Prinzip sind technische Grundlagen für CAD-System zwar integriert, komplexe Anwendungen dieser Art dürften aber wohl bessere Anwendungsumgebungen finden, Domänen-Frameworks wie Unidraw oder noch spezieller auf den Anwendungskontext zugeschnitten.

Umgekehrt basieren Malprogramme nicht auf graphischen Objekten aus Linien und Flächen, sondern orientieren sich an einzelnen Bildpunkten (Pixel). Abgesehen von der Darstellung von Bitmaps unterstützt tCanvas keine Pixel - Grafik. Damit scheidet beispielsweise Bearbeitung von Foto- oder Videomaterial aus, für bewegte Bilder ist zudem weder die BibV30 ausgelegt, noch Unidraw.

Als WAM-Framework wird die BibV30 benutzt, um Anwendungen für selbstbestimmte oder qualifizierte Tätigkeiten, meist im Bürobereich zu entwickeln, hinsichtlich der Grafik ist nicht an eine Erweiterung des Einsatzkontextes in völlig neue Bereiche gedacht.

Grafik in Büroanwendungen?

Wo gibt es aber nun im klassischen Kontext von WAM- Applikationen Bedarf an graphischen Komponenten, z.B. für den Banksachbearbeiter?

Börsenkurse und Statistiken, oder allgemein die graphische Darstellung numerischen Materials ist sicher eine gute Antwort auf diese Frage. Auch graphisches Material gibt es, z.B. gehören zu den Unterlagen für Immobilien - Kredite immer auch Fotos des betreffenden Hauses. Und am Selbstbedienungsterminal sieht der Kunde ein Bild seines Kundenberaters … . Besser als Bitmap - Beispiele eignen sich vielleicht Übersichten über Hypertexte zwecks Orientierung, Organigramme oder technische Zeichnungen, Konstruktionspläne und Bauanleitungen in einer Datei herzustellender Produkte und lieferbarer Waren eines Versandhauses, vielleicht abrufbar für die Kundenberatung bei Verkauf oder Reklamationen.

Direkte Manipulation?

Viele dieser Beispiele stehen für graphische Darstellungen ohne Benutzerinteraktion, z.B. die Anzeige von Konstruktionsplänen, deren Erstellung Aufgabe externer CAD-Systeme bleibt. Möglich sind vielleicht Ausschnittsvergrößerungen, der Anwender könnte Details ein- und ausblende, nicht aber das Material selbst manipulieren. Für die Darstellung statistischer Daten ließen sich z.B. durch das Anklicken mit der Maus nähere Informationen selektieren, Balken eines Diagramms per direkter Manipulation anordnen und ggf. die Farben beeinflussen.

Besser geeignet ist die Übersichtsanzeige von Hypertexten. Direkte Manipulation eignet sich für das Anlegen, Korrigieren und Löschen von Verbindungen oder Knoten und zum Navigieren im Netz - Knoten öffnen per Doppelklick. Zusätzlich nimmt der Anwender eine topographische Anordnung vor, um sich so eine eigene Sicht auf das Netz zu schaffen.

Echte graphische Editoren ergeben sich, wenn z.B. Organigramme nicht nur gezeigt sondern auch geändert werden, wenn wirklich graphisches Material vom Anwender manipuliert wird.

Anwendungen innerhalb des WAM-Frameworks

Gute Beispiele finden sich bei den WAM- Konzepten für kooperatives Arbeiten. Prozeßmuster sind ein spezielles, graphisches Material, daß der Bankmitarbeiter zwecks Koordination der Tätigkeit einsieht, um Erledigungsvermerke ergänzt oder ggf. auch entsprechend konkreter Umstände anpaßt. Ohne direkte Manipulation als einfaches Mittel der Bedienung läßt sich ein akzeptables Werkzeug kaum konstruieren.

Der elektronische Schreibtisch als sichtbares Element der Umgebung (fehlt leider noch in der BibV30) ist auch ein Beispiel, der Benutzer kann und soll Materialien und Werkzeuge selbst entsprechend seiner Arbeitssituation anordnen (die Ordnung ist persistent), er wird Werkzeuge per Mausaktion starten und Material mit passenden Werkzeugen öffnen.

Aufgabenteilung zwischen IAT, IAK und FK

Variante eins ist ein WAM- Werkzeug für graphisches Material (z.B. Aufgabennetze). Dessen Änderung geschieht immer in der FK, seitens der IAK erfolgt nur der Aufruf entsprechender Methoden entsprechend dem Ergebnis des Manipulator - IAT's. Umgekehrt meldet die FK ihre Zustandsänderung und die IAK aktualisiert den tIATGraphic bzw. dessen GFX-Objekt.

Eine zweite Variante bilden Werkzeuge, die fachlich mit numerischen Material arbeiten. Die FK hat keine Kenntnis, ob die Ergebnisse tabellarisch angezeigt oder graphisch aufbereitet werden, direkte Manipulation ändert nicht das Material, sondern nur dessen Darstellung. Somit wertet die IAK selbst den tIATManipulator aus und ändert die GFX-Objekten.

Als dritte und letzte Version sei die direkte Manipulation als Bedienform eines IAT verstanden. Typische Listen verfügen über Scrollbars, mit denen der Anwender den sichtbaren Ausschnitt verschiebt - ohne Beteiligung der IAK. Dieses Beispiel läßt sich direkt auf die Zeichenfläche übertragen, zusätzlich denkbar ist z.B. eine Vergrößerung durch Auswahl des gewünschten Bereiches durch direkte Manipulation. Hier könnte der Manipulator direkt den Maßstab und Ausschnitt des tIATCanvas verändern, die Aktion ist für IAK und FK ohne Bedeutung.

Konkrete Beispiele für Grafiken und Manipulatoren in der BibV30

Mit dem Grafik- Framework entstand ein allgemeiner grafischer Editor als Beispielanwendung für die Dokumentation sowie als Testrahmen während der Entwicklungsphase. Enthalten ist ein kleines Sortiment geometrischer Figuren und diverse verschiedenartige Manipulatoren. In der WWW-Dokumentation [Weiß97] sind die Klassen und Implementationsdetails beschrieben, dieser kurze Überblick vermittelt einen Eindruck betreff der praktischen Verwendung des Mikro-Frameworks und veranschaulicht vorangehende Ausführungen.

Für jede zu unterscheidende geometrische Form wird eine eigene GFX-Klasse implementiert, im Beispiel tGFXCircle, tGFXEllipse, tGFXRectangle, tGFXSector und andere. Objekte dieser Klassen erzeugt die Interaktionskomponente, zur Darstellung parametrisiert sie Grafik-IAT's mit diesen Objekten und fügt die IAT's in einen tIATCanvas ein. Es müssen keine speziellen Klassen von tIATGraphic abgeleitet werden, dieser Interaktionstyp kann beliebige Grafiken anzeigen. Dazu dient die Parametrierung mit einem GFX-Objekt (s.o., Abb. 4-17).

In der Darstellung unterscheiden sich z.B. Kreise und Ellipsen nicht, vielmehr sind Kreise nur eine Spezialform von Ellipsen. Objekte vom Typ tGFXEllipse könnten auch für die Anzeige von Kreisen benutzt werden. Da es sich bei der Anwendung aber um einen graphischen Editor handelt, können diese Figuren vom Benutzer manipuliert werden, hier ergibt sich zwischen beiden Formen ein semantische Unterschied. Operationen der Klasse tGFXCircle stellen sicher, daß das Objekt bei allen Änderungen kreisförmig bleibt. Technisch sieht die Schnittstelle zu jeder setzenden Methode auch sondierende Funktionen vor zwecks Prüfung der Zulässigkeit. Definiert ist diese Schnittstelle bereits in der abstrakten Oberklasse tGFXDrawable, Default ist der Ausschluß aller Operationen.

Handle - Manipulatoren erlauben Verformung der Grafiken, z.B. ändert ein Manipulator den Winkel eines Kreisausschnittes, auch die üblichen Manipulatoren für Größenänderungen sind als Handle realisiert, die Punkte von frei gezeichneter Figuren (Kette von Linien) können mit Manipulatoren verschoben werden. Dagegen erlauben normale Manipulatoren bewegen von Grafikelementen, Selektion einzelner Objekte oder Gruppenselektion. Selektion geschieht durch anklicken der Elemente oder durch Lassos.

Der Anwender des Grafikeditors kann gleichzeitig verschiedene Manipulatoren auswählen, die einzelnen Tasten der Maus zugeordnet werden. So hängt es von der gedrückten Maustaste ab, ob das angeklickte Objekt eine bestehende (mehrfach) - Selektion erweitern, oder nachher nur das gewählte Grafikelement selektiert ist. Eine Taste wird in vielen gängigen Systemen mit der Funktionalität „Menü" belegt, um z.B. Attribute wie Strichstärke oder Farbe einzustellen.

Darstellung der Objekte ist Aufgabe der Zeichenfläche, solange kein Manipulator aktiv ist. In dieser Zeit sorgt das Manipulator - Objekt für eine der Benutzeraktion entsprechende Anzeige, dazu erhält es vollen Zugriff auf die tIATGraphics im Canvas. Im Konzept ist vorgesehen, daß Manipulatoren keine Änderungen am graphischen Material vornehmen, technisch greifen sie jedoch auf die GFX-Objekte zu. Vorgesehen ist eine Kopie der Objekte und die Manipulation dieser Kopie statt der Originale, innerhalb des Frameworks wird dies aber nicht sichergestellt.

Auch im Beispiel arbeiten die Manipulator - Klassen direkt mit den Original- Objekten. Dies ist solange unproblematisch, wie vorgenommene Änderungen nach Abschluß bzw. bei Abbruch der Benutzeraktion vom Manipulator automatisch zurückgenommen werden.

Als letzte Anmerkung soll noch erwähnt werden, daß im Gegensatz zur Klasse tIATGraphic der Interaktionstyp tIATManipulator durch spezialisierte Unterklassen implementiert wird. Je nach Anwendungsfall gibt es sehr unterschiedliche Formen der direkten Manipulation, und so auch jeweils eigene Manipulatoren. Diese müssen - wie auch die GFX-Klassen - implementiert werden. Vorteile für die Anwendungsentwicklung ergeben sich aus der klaren Trennung der Verantwortlichkeiten von Interaktionskomponente, Zeichenfläche (tIATCanvas), GFX-Klassen und den Manipulatoren. GFX- und Manipulator - Klassen lassen sich aber z.B. in ähnlichen Projekten direkt wieder verwenden.

Das Beispiel enthält dabei einen Manipulator zum Erzeugen neuer Grafikelemente. Realisiert wurde dieser unter Verwendung der GFX-Klassen: mit derselben Manipulator - Klasse kann der Benutzer unterschiedliche graphische Figuren erzeugen, der Manipulator wird einfach mit der Kopie eines GFX-Objektes parametrisiert. Dies funktioniert z.B. auch mit speziellen, vom Anwender frei entworfenen Figuren, die sich so beliebig kopieren lassen (vgl. [Weiß97]).

Sind Manipulatoren IAT's ?

Interaktionstypen dienen der Kommunikation mit dem Anwender, verfügen über darstellende Komponenten und nehmen Eingaben des Benutzers entgegen, tIATManipulator ist eindeutig ein IAT. Auch tIATGraphic stellt logische Einheiten dar, Benutzeraktionen erfolgen aber nicht über diesen IAT erfolgen, sondern immer über einen Manipulator.

In der vorliegenden Implementation ist die Selektion dem tIATGraphic - Objekt zugeordnet. Bei der Zeichenfläche tIATCanvas meldet die IAK dagegen keine Commands an. Entsprechend 4.3.3 handelt es sich eher um ein Layoutelement, einen Rahmen um die tIATGraphic.

Zur Diskussion über Interaktionsform und Präsentation

Führe ich diese Überlegungen hinsichtlich der angestrebten Trennung von Präsentation und Interaktion fort, so stellen Canvas und Grafik- IAT offensichtlich keine Interaktionsform dar, tIATCanvas hat eigentlich auch keine Präsentation, sondern dient nur der Verwaltung. Bleibt ein einziger IAT, der das Repertoire an Interaktionsformen erweitern könnte: als neue Form neben 1-from-N-select, Activator und anderen reiht sich der „*graphische - Manipulator*“ ein.

Listen präsentieren vielfach die übliche Interaktionsform Selektion (1 aus N, M aus N). Viele Systeme sehen als Alternative eine graphische Präsentation vor, z.B. für Dateiverwaltung oder elektronische Schreibtische. Aus dieser Sicht bilden die hier gezeigten Grafik- Komponenten der BibV30 möglicherweise zukünftig eine technische Basis für spezielle Präsentationsformen zu bekannten und neuen Formen der Interaktion.

4.5 Erzielte Fortschritte und weitere Planung

Die Vielzahl der in diesem Kapitel dargestellten Erweiterungen und Änderungen des WAM-Frameworks dokumentieren einen Teil der Entwicklung zur BibV30. Konzeptionell läßt sich der Fortschritt auf zentrale Anliegen zusammenfassen:

- Die Methode WAM prägt die Komponenten des Frameworks, in diesem Kontext entwickelte Konzepte und Design Pattern sollen umgesetzt, erprobt und vor allem miteinander kombiniert werden.
- Angestrebt wird der wahlweise Einsatz derjenigen Komponenten, die für bestimmte Anwendungen erforderlich sind. Die Grafik gemäß Abschnitt 4.4 ist z.B. nur für eine relativ kleine Gruppe mit dem Framework entwickelter Applikationen notwendig.
- Zur Umsetzung dieses Anspruchs optional kombinierbarer Komponenten werden kleine Mikro-Frameworks mit minimaler Kopplung entwickelt, ein Beispiel ist die Anbindung des Fenstersystems (siehe 4.3.3).
- Es wird eine Trennung zwischen objektorientierten Mechanismen in Form genereller Design Pattern nach Maßgabe [GoF95] und den damit realisierten WAM-Metaphern durch eigene Mikro-Frameworks für die Design Pattern durchgeführt.
- Alternative Application-Frameworks als oberste Ebene der WAM-Frameworks sowie ein Satz an Hilfswerkzeugen und Automaten runden den Ansatz ab.

Zusammenfassend lassen sich die C++ Bibliotheken am Arbeitsbereich Softwaretechnik als Sammlung von Frameworks auffassen, die im Rahmen der Methode WAM entwickelt und aufeinander abgestimmt wurden, um die Grundlage für verschiedene Anwendungsdomänen zu legen. Hinsichtlich dieser Domäne gibt es als einzige Einschränkung das Leitbild der Werkzeug und Material Metapher vom Arbeitsplatz für qualifizierte menschliche Tätigkeit, vorzugsweise in einer Büroumgebung.

Benutzern soll ein leichter Einstieg in neue Frameworks gegeben werden. Lose gekoppelte Komponenten der BibV30 sind ein Weg, dies zu erreichen. Gleichzeitig ergeben sich aus der Vielzahl an Komponenten jedoch neue Probleme, die das nächste Kapitel thematisiert.

Weiter verfolgen will ich den Bezug zwischen der BibV30 und spezifischen Frameworks für bestimmte Domänen. Nach meiner Vorstellung soll dieses WAM-Framework als Basis *verschiedener* Domänen-Frameworks dienen, auch unter diesem Gesichtspunkt betrachte ich nachfolgend die Struktur der Bibliotheken.

Kapitel 5 Strukturierung von Framework-Bibliotheken

Dokumentation in gedruckter Form und auch Online als Hypertext ist eine Voraussetzung für die Verständlichkeit größerer Klassenbibliotheken. Besonders gilt dies für Frameworks, deren vorgegebener Kontrollfluß vielfach nur mittels zusätzlicher Werkzeuge verstanden werden kann (vgl. [LS96])

Explizite Leitbilder erleichtern den Einstieg in ein Framework, erste Programme können auf Basis dokumentierter Beispiele geschrieben werden. Später helfen vor allem Design Pattern bei dem Verständnis der Architektur, wobei das angestrebte Ziel einer Black-Box - Wiederverwendung kein näheres Verständnis der internen Strukturen verlangt. Während ein Benutzer nur verstehen muß, welche Klassen er spezialisieren oder benutzen kann bzw. welche Methoden anzupassen sind, erfordert die Erweiterung oder Änderung von Framework-Komponenten weitergehende Kenntnisse.

Kapitel 4 beschreibt die Komponenten der BibV30. Grundprinzip des Entwurfs sind kleine, eigenständige Frameworks, die jeweils ein spezielles Konzept, einen Mechanismus für flexible Framework-Komponenten (z.B. Observer Design Pattern) implementieren. Für den Benutzer wird es dadurch leichter, den Zweck und Verwendungszusammenhang der Mikro-Frameworks zu erkennen, die Zugehörigkeit der Klassen zu kleinen Teams ist explizit. Nachteilig erweist sich dagegen im praktischen Einsatz der Bibliotheken die große Anzahl an Frameworks, durch die auf den ersten Blick die Framework-Sammlung BibV30 unübersichtlich wirkt. Die Lösung sehe ich in einer nachvollziehbaren Struktur der Framework - Sammlung.

Abschnitt 5.1 definiert Anforderungen an eine Strukturierung, indem die zu lösenden Probleme beschrieben und bewertet werden. Daraus ergibt sich in Abschnitt 5.2 ein Ansatz für die Struktur der BibV30 und deren geplante Erweiterung, zudem sind Anforderungen ableitbar, die Einzelkomponenten des Frameworks zu erfüllen haben.

Meinen Strukturvorschlag für die BibV30 setze ich in Bezug zu einem auf GEBOS basierenden Beitrag [BGKLRZ97] (siehe Abschnitt 5.3). GEBOS ist ein auf WAM bezogenes Framework für die Banken - Domäne (vgl. [BGKZ96]). Damit greife ich den Gedanken des 4. Kapitels wieder auf, die C++ Bibliotheken des Arbeitsbereiches Softwaretechnik außerhalb der Lehre einzusetzen, und zwar als „Meta - Framework“ für die Entwicklung konkreter Domänen-Frameworks.

Die Einordnung der inhaltlich in Kapitel 4 vorgestellten Komponenten in die Struktur der BibV30 gemäß Abschnitt 5.2 hilft mir, den Bezug der Mikro-Frameworks zu dem Domänen-Framework GEBOS aufzuzeigen. Ich verallgemeinere die Beziehung zwischen BibV30 und GEBOS auf die Konstruktion verschiedener Domänen-Frameworks unter Verwendung der nicht auf einen bestimmten Anwendungskontext bezogenen BibV30.

5.1 Ziele der Strukturierung

Probleme bereiten große Klassenbibliotheken durch eine Vielzahl an Klassen bzw. auch kleiner Frameworks, aus denen im gerade betrachteten Kontext die jeweils relevanten herauszufinden sind. Unterschiedliche Anforderungen entstehen hinsichtlich Benutzung und Entwicklung des Frameworks.

Für Benutzer eines Frameworks sollte der Zugang leicht sein, damit das Framework die durch Wiederverwendung erwartete Minderung des Aufwandes tatsächlich erreicht. Dagegen darf Wartung, Änderung und Erweiterung dieser Bibliotheken durchaus einen höheren Aufwand erfordern. Der Schwerpunkt der Strukturierung betreff Erweiterung eines Frameworks verschiebt sich mehr auf die internen Beziehungen der Komponenten zueinander.

Im letzten Teil dieses Abschnittes werden Möglichkeiten untersucht, verschiedene Versionen eines Frameworks bereitzustellen, bzw. in Applikationen jeweils nur einige Komponenten zu verwenden. Als Motivation dienen Anwendungsfälle im Bereich der Industrie sowie des universitären Lehrbetriebes. Abschnitt 5.3 wird die Frage nach Framework - Konfiguration auf die Konstruktion neuer Framework - Sammlungen erweitern.

5.1.1 Lernaufwand bei der Framework-Benutzung

Wiederverwendung ist ein anerkanntes Ziel objektorientierter Programmierung, um Entwicklungskosten und -zeit einzusparen und die Qualität der erstellten Software zu verbessern. Die Framework - Technik kombiniert Design und Code und wird als heutzutage optimales Mittel zur Wiederverwendung angesehen. [Pree94]

Voraussetzung für eine erfolgreiche Benutzung der vorhandenen Architektur und der verschiedenen Komponenten sind entsprechende Kenntnisse des Anwendungsentwicklers. Dieser erwartet geringen Einarbeitungsaufwand und guten Überblick.

Die im Kapitel 4 vorgestellten C++ Klassenbibliotheken wurden in den vergangenen Jahren mehrfach in Lehrveranstaltungen eingesetzt (Projekte). Die beteiligten Studenten betrachteten den Lernaufwand im Verhältnis zu der späteren Benutzung als sehr hoch. Diese Einstellung änderte sich bei späterer Erstellung von Studien- oder Diplomarbeiten mithilfe der Application-Frameworks: Während die Vorteile des Frameworks im Verlauf eines zeitlich beschränkten Projektes kaum zur Geltung kommen, werden diese bei der Entwicklung und Implementation größerer Anwendungen ausgiebig genutzt.

Application-Frameworks im Lehrbetrieb unterscheiden sich somit von deren Anwendung in der industriellen Software-Entwicklung. Auch in diesem Rahmen sind häufig neue Mitarbeiter einzuarbeiten oder auch Frameworks von Fremdanbietern zu testen, dem hohen Lernaufwand steht aber regelmäßig eine langfristige Nutzung in konkreten Projekten gegenüber, wenn die Entscheidung für ein Application-Framework getroffen oder dieses selbst entwickelt wurde.

Schwierigkeiten ergeben sich bei dem Einstieg in ein neues Framework aus der Fülle verwendeter Mechanismen. Verschiedene Mittel implementieren einen festgelegten Kontrollfluß und erlauben dennoch eine flexible Anpassung der variablen Anteile, also der Hot Spots. Auch wenn der Einstieg häufig über ein mitgeliefertes Beispiel erfolgt - oder auch durch Werkzeuge erleichtert wird (vgl. MFC), - sind auf dem Weg zu einer eigenen Applikation verschiedene Komponenten kennenzulernen.

Es stellen sich für den Benutzer dabei die Fragen:

- welche Komponenten beinhaltet das Framework, welche Funktionalität ist implementiert
- wo befinden sich die anzupassenden Klassen, welche Methoden sind zu überschreiben

In diesem Zusammenhang stellt sich die Frage nach Kenntnissen über interne Strukturen für die Benutzung des Frameworks. Soweit die Dokumentation keine ausreichenden Informationen über die Dynamik gibt, also die Einbindung eigener Klassen in das Frameworks, bleibt häufig nur die Möglichkeit, diese Dynamik mit einem Debugger nachzuvollziehen.

Konventionelle Debugger entsprechen weitgehend dem Verständnis prozeduraler Sprachen, objektorientierte Konzepte finden wenig Beachtung (vgl. [FH96]: „*Betrachtet man, welche Debugger und Profiler angeboten und eingesetzt werden, so fällt auf, daß sich objektorientierte Technologien nur in einem geringen Maß durchgesetzt haben [...] beim Debugging ist der Entwickler gezwungen, nicht objekt-, sondern kontrollflußorientiert vorzugehen.*“). Klassen und Klassenhierarchien gehen zur Laufzeit verloren, der Debugger arbeitet nur mit Methoden, einzelnen Anweisungen.

Look! [West93] ist ein Werkzeug mit völlig anderen „Debugging“ Ansätzen. Der Benutzer verfolgt Objekt - Erzeugung, das Werkzeug visualisiert dynamische Benutzt-Beziehung. Die Klassen und Objekte sind in diesem System Ausgangspunkt der Betrachtung und nicht nur dem „eigentlichen“ Debugging beigefügt. Eine aktuelle Betrachtung dieses Thema und eine Reihe neuer Vorgehensweisen, Darstellungen und Werkzeugkomponenten diskutiert [LS96]. Das Werkzeug VOOP, teilweise noch in der Entwicklung, verwirklicht diese Ideen. Neben der wichtigen Ergänzung zur konventionellen Dokumentation durch Klassenhierarchien, Design Pattern und Kochbücher zur Verwendung in der vorhergesehenen Standardfällen[17], hilft VOOP bei Integration in die Entwicklungsumgebung auch bei der Fehlersuche und Optimierung, ist also eine echte Alternative zum Debugger oder eine nützliche Ergänzung.

Auch neue, objektorientierte Debugging - Techniken verlangen Vorkenntnisse des Benutzers, vermittelt durch konventionelle, statische Dokumente. Die Größe heutiger Application-Frameworks verlangen nach einer Eingrenzung des relevanten Bereiches, der Benutzer muß entscheiden, wie detailliert er einzelne Vorgänge verfolgen will. Die Struktur eines Frameworks als Leitfaden der Dokumentation gibt eine Orientierung und ggf. auch einen Anhaltspunkt vorgegebener Szenarien für das Debugging.

---

[17] Kochbücher sind immer nur unvollständig [vgl. Pree94], da nicht alle Anwendungsfälle vorhersehbar sind. „A framework used extensively tends to be used in ways never conceived by its designers.“ [LS96]

Für die Ordnung der großen Anzahl an Klassen benutzt [Meyer94] den Begriff Cluster, bei Verwendung der Programmiersprache C++ bietet sich die Verwendung eigener, dynamisch gelinkter Bibliotheken an. Der (neue) Benutzer beschränkt sich auf einige wenige Bibliotheken, interne Details werden in eigene Bibliotheken gekapselt.

Jede Bibliothek bildet ein Framework bzw. kapselt einen Mechanismus, die Namen dieser Bibliotheken geben einen Überblick über die Leistungen des kompletten Application-Frameworks. Dokumentierte Abhängigkeiten zwischen Bibliotheken lassen sich nutzen, die Komponenten in einer daran orientierten Reihenfolge zu betrachten. Gekennzeichnete, nur intern verwendete oder durch weitere Frameworks auf höherer Abstraktionsebene gekapselte Bibliotheken betrachtet der Benutzer nicht weiter.

5.1.2 Wartung und Erweiterung um neue Komponenten

Bei der Entwicklung eines Application-Frameworks handelt es sich nicht um einen kurzen und abgeschlossenen Prozeß, dessen Ergebnis die Erstellung eines fertigen Produktes ist. Vielmehr führt die Verwendung des Framework zu neuen Erkenntnissen, aus denen sich notwendige Änderungen sowie Erweiterungswünsche ergeben. Im Vergleich zur reinen Benutzung des Produktes stellt dessen Bearbeitung höhere Anforderungen an den Software - Entwickler. An dieser Stelle soll nur der Zusammenhang mit Framework-Strukturierung betrachtet werden, Ausführungen zur Framework - Evolution finden sich z.B. bei [Willamowius97].

Im Falle einer Fehlerkorrektur ist die verantwortliche Komponente zu identifizieren, von der Änderung indirekt betroffene Klassen zu ermitteln. Basieren bereits Anwendungen auf dem Framework, so ist eine genaue Analyse der Auswirkungen erforderlich. Zur Bewältigung dieser Aufgabe ist ein wesentlich höherer Lernaufwand als bei einfacher Benutzung unvermeidlich. Ein gut strukturiertes Application-Framework erleichtert die Identifikation jeweils betroffener Komponenten. Stärker noch als bei der Benutzung wirken sich dokumentierte Abhängigkeiten positiv auf die Wartungstätigkeit aus.

Erfolgreiche Frameworks führen durch ihre Verwendung zu neuen Wünschen der Anwender. Es ergibt sich aus den Erfahrungen der Anwendungsentwickler, daß gewisse Bereiche besser zu unterstützen sind, Zusammenhänge flexibler gestaltet werden müssen und dafür andere Komponenten zu standardisieren sind. Wird ein bewährtes Application-Framework außerhalb des ursprünglichen Anwendungskontextes eingesetzt, benötigt es dafür neue Mechanismen.

Auch bei diesen Erweiterungen der Funktionalität stellt sich die Frage, welche bestehenden Teile als Basis zu verwenden sind, und welche Konsequenzen die Ergänzung an anderer Stelle bewirkt. Auswirkungen wichtiger Fehlerkorrekturen auf bestehende Applikationen sind oft nicht zu vermeiden, neue Komponenten mit zusätzlicher Funktionalität sollten aber optional verwendbar sein (hierzu siehe 5.1.3).

Auch wenn die Arbeit an dem Application-Framework weitgehende Kenntnisse der Architektur voraussetzt, diverse Mechanismen bekannt sein und neue Komponenten in das Leitbild „passen“ müssen, so ist doch eine Begrenzung auf möglichst enge Bereiche anzustreben. Ein umfassendes Verständnis kompletter Frameworks sollte nicht Voraussetzung sein.

Taligent’ s Architektur (vgl. Abschnitt 3.4: CommonPoint) verwendet Frameworks ausgehend vom Betriebssystem bis hin zur Unterstützung des Anwendungsentwicklers. Das Konzept ist dabei gerade die Kapselung technischer Details auf unterer Ebene vor den Entwicklern der anwendungsorientierten Frameworks. Ein neue Komponente des Application-Frameworks wird erstellt, ohne die Steuerung eines SCSI- Laufwerkes zu verstehen. Und genauso läßt sich eine Portierung auf neue Hardware vornehmen, nur durch Anpassung der technischen Basiskomponenten. Die Konstruktion auf höherer Ebene ist nicht betroffen.

Die Struktur des Application-Frameworks erleichtert die Identifikation jeweils betroffener Komponenten und der Zusammenhänge mit abhängigen Teilen der Klassenbibliothek. Wechselwirkungen von Änderungen und Erweiterungen mit bestehender Funktionalität werden überschaubarer, unabhängige Bereiche können unbeachtet bleiben.

Ein Schema zur Strukturierung muß zudem die Einordnung neu erstellter Komponenten vorsehen, um auch zukünftig den Überblick zu erhalten. Dem Entwickler zusätzlicher Klassen ist somit eine Anleitung zu geben, wie er seine Erweiterung in die Architektur integrieren kann. Dabei ist das Ziel loser Kopplung weiterhin zu beachten.

5.1.3 Aspekte der Konfiguration, DLL’s

Sammlungen von Frameworks lassen sich in verschiedenen Anwendungsfeldern einsetzen, eine Bibliothek erfüllt unterschiedliche Anforderungen. Beispielsweise enthält die Framework - Sammlung verschiedenen Application-Frameworks. Die Verwendung der BibV30 sowohl für Lehrveranstaltungen als auch für Anwendungs- Entwicklung im Rahmen von Diplomarbeiten ist ein weiteres Beispiel.

Lehrveranstaltungen zielen darauf ab, theoretische Kenntnisse der WAM- Methode praktisch zu vermitteln. Gerade der Umgang mit einem objektorientierten Framework wird vermittelt. Demgegenüber wollen Diplomarbeiter das Framework in der üblichen Form als Infrastruktur für ihre eigene Anwendung benutzen. Die dritte Gruppe sind Studien- und Diplomarbeiter, die selbst konzeptionelle Erweiterungen des Frameworks vornehmen.

Bäumer et. al. [BGKLRZ97] stellen einen Ansatz zur Strukturierung großer Frameworks vor. Ausgangspunkt ist ein Domänen-Framework im Bankenbereich (GEBOS), das im Zusammenhang mit Anwendungen für praktisch alle Bereiche des Bankgeschäftes („*family of applications ... cover almost the complete area of banking*“) entwickelt wurde. Die vorgeschlagene Struktur wird in Abschnitt 5.3 näher betrachtet, folgende zwei Aspekte beziehen sich auf unterschiedliche Konfiguration erstellter Anwendungen.

Für jedes Framework wird eine eigene DLL (dynamic link library) erzeugt, Anwendungssysteme mit den jeweils benötigten DLL' s ausgeliefert („*Each application system consists of only those libraries that are required for functioning*."). Innerhalb der Entwickler - Organisation benutzen die Anwendungsentwickler Frameworks nur als DLL und nehmen selbst keine Änderungen am Framework vor („*The different frameworks are provided in DLLs to ensure that framework classes cannot be changed at will by application developers*."). Änderungen am Framework sind Aufgabe einer speziellen Gruppe innerhalb der Organisation.

Zudem gibt es neben technischer (*Technical Kernel Layer*) und methodischer Schicht zur Werkzeug-Konstruktion (*Desktop Layer*) weitere gemeinsame Frameworks für alle Anwendungen (*Business Domain Layer*), auf deren Basis dann spezialisierte Frameworks für einzelne Geschäftsbereiche entwickelt werden (*Business Section Layer*). Anwendungen verwenden jeweils die Application-Frameworks des entsprechenden Geschäftsbereiches.

Die Schichtenbildung macht deutlich, welche Komponenten des GEBOS- Systems von allen Anwendungen verwendet werden. Abhängigkeiten sind geregelt, Frameworks einer explizit eingeführten Schicht werden wahlweise verwendet, je nach *Geschäftsfeld* der Anwendung.

5.2 Strukturkonzept für die BibV30

Benutzern gibt die Struktur einen Überblick über das Framework, erlaubt die Orientierung in einer Vielzahl kleiner Mikro-Frameworks, aus denen z.B. die BibV30 besteht. Als Hilfe für den Einstieg wird die Verbindung zwischen Komponenten geklärt, so daß sich der neue Benutzer jeweils auf die kleinere Bereiche beschränken kann. Später lassen sich die gerade benötigten Mikro-Frameworks für einen bestimmten Anwendungszusammenhang in der gut strukturierten Bibliothek leicht identifizieren.

Entwickler, die Änderungen und Erweiterungen am Framework selbst vornehmen, erwarten Informationen über Verbindungen zwischen Komponenten, Abhängigkeiten zwischen Klassen und Frameworks, um die Auswirkungen einer Änderung auf andere Teile der Sammlung zu beurteilen. Das Strukturkonzept muß als Schema nicht nur bestehende Komponenten sortieren, sondern auch eine Anleitung für die Einordnung neuer Mikro-Frameworks darstellen.

Schließlich ist die BibV30 ein konfigurierbares Framework für die Methode WAM, wird mit unterschiedlichen Zielen in diversen Kontexten verwendet. Die Struktur muß Varianten aufzeigen, wahlweise zu benutzende Komponenten von Grundlegenden trennen und abhängige Mikro-Frameworks in Beziehung setzen.

Dies soll durch die Bildung von Schichten erreicht werden. Zuerst stellt 5.2.1 Kategorien für die Einteilung der Frameworks vor, der konkrete Vorschlag für die BibV30 folgt dann in 5.2.2 (zur Motivation für 5.2.1 ist ein Blick auf Abbildung 5-1 zu empfehlen). Vorgestellte, aber nicht vorhandene Kategorien greift 5.2.3 wieder auf.

5.2.1 Kategorien der Struktur

Ausgangspunkt ist die Identifikation dreier Dimensionen in einer Framework - Bibliothek: ***technische*** Mechanismen als Fundament bilden mit ***methodisch*** geprägte Komponenten den Rahmen für ***fachliche*** Mikro-Frameworks und Anwendungen.

Technische Klassen und Frameworks ergeben die Basis für alle weiteren Komponenten einer Bibliothek. Innerhalb dieser Dimension finden sich Frameworks verschiedener Kategorien, mit deren Benennung und Abgrenzung wird die Zuordnung konkreter Frameworks möglich.

Spracherweiterungen bezwecken den Ausgleich von Defiziten einer konkreten Programmiersprache. Es werden elementare Mechanismen ergänzt, die in andere Sprachen automatisch integriert sind. Vor der aktuellen Standardisierung fehlten in C++ beispielsweise die Klassen Bool und String, sowie ein Meta Objekt Protokoll (MOP). Eine weitere wichtige Ergänzung wäre ein Garbage Collector, der auch heute noch zu C++ gehört.

Ein Merkmal oder Anhaltspunkt für die Zuordnung von Frameworks zu dieser Kategorie ist der Vergleich mit anderen Sprachen, die den Mechanismus bereits enthalten. So kann auch eine sicher nicht zwingende Integration des Vertragsmodelles in eine Bibliothek als Erweiterung der Sprachmächtigkeit aufgefaßt werden, da sich dieses Konzept in Eiffel bewährt hat.

Behälterklassen in irgendeiner Form benutzt jede Anwendung, nahezu jede objektorientierte Sprache liefert diese mit, wieder mit der Ausnahme C++. Allerdings verfügt genaugenommen auch diese heterogene Programmiersprache über ein diesbezügliches Konzept, nur stammt es noch aus purem C: das Array. In einer objektorientierten Umgebung ist die damit verbundene Pointer - Arithmetik natürlich indiskutabel, die Sprache wird um Behälterklassen erweitert. Seit kurzem bestätigt die Normierung der STL (Standard Template Library) die Auffassung, daß dieser Typ von Klassen zur Sprache selbst gehören muß.

Diskussionsbedarf ergibt sich bei der zweiten Kategorie der ***objektorientierten Techniken***, die ich auch der technischen Dimension zurechne. [Pree94] argumentiert für die Kombination der elementaren objektorientierten Konzepte (Vererbung, Polymorphie und abstrakte Klassen) als notwendige Grundlage für die Framework - Konstruktion. Er identifiziert sieben Varianten der Kombinationen und nennt diese Metapattern. Umgekehrt suchen aus Richtung des praktischen Konstruktion Autoren, z.B. [GoF95] nach Design Pattern, die Pree als konkrete Ausprägungen der abstrakten Metapattern einordnet (vgl. Abschnitt 2.2).

Aus Sicht der Framework-Entwicklung sehe ich in diesen Mechanismen eine Erweiterung des objektorientierten Fundamentes für wiederverwendbare Software - Bausteine, insbesondere Frameworks. Gemäß [Pree94] sind flexible Klassen und Teams ohne diese überhaupt nicht realisierbar. Viele Bibliotheken implementieren Mikro-Frameworks nach Vorlage von Design Pattern, um mit diesen fachliche Application-Frameworks flexibel zu konstruieren. Ich rechne diese Art von Mikro-Frameworks bzw. Komponenten einer Bibliothek als objektorientierte Techniken der technischen Dimension zu, wenn auch Implementationen von Design Pattern an der Schnittstelle zu methodisch motivierten Bereichen liegen.

Eine weitere Begründung für diese Sichtweise entnehme ich ET++. Dort faßt die allgemeine Oberklasse Object verschiedene Spracherweiterungen und Design Pattern zusammen, um so eine bessere Infrastruktur für das Framework und damit entwickelte Anwendungen zu bieten. Es handelt sich also um eine Sammlung an Mechanismen, die beliebig verwendbar sind. In der methodischen Dimension erwarte ich dagegen einen konkreten Bezug dort implementierter Komponenten auf bestimmte Verwendungszusammenhänge.

Neben diesen selbst entwickelten technischen Basisklassen und Frameworks werden vielfach auch Produkte von Drittanbietern verwendet, die z.B. als ***Schnittstellen*** zum Betriebssystem, zu einer Datenbank und vor allem zur graphischen Benutzungsschnittstelle dienen. Auf höheren Ebenen kann eine Kapselung vorgenommen werden, oder die angebotene Funktionalität wird von der Anwendung direkt benutzt. In jedem Fall lassen sich diese externen Bibliotheken (nicht notwendigerweise Frameworks) als Teil der technischen Infrastruktur auffassen, die Grundlage höherer Schichten ist. Eine Gleichsetzung mit Spracherweiterungen oder Design Pattern scheint dagegen nicht angebracht, beide Kategorien bestehen nebeneinander.

Als zweiten Teil des Rahmens für Anwendungen haben ich **methodisch** geprägte Frameworks innerhalb einer Bibliothek identifiziert. Diese Sichtweise ist sicherlich begründet durch den großen Einfluß der Methode WAM auf die Entwicklung des Frameworks BibV30. Allgemein betrachtet gehören hierhin all jene Kategorien von Klassen und Frameworks, die direkte Grundlage der Anwendungsentwicklung sind. Vor allem also ***Anwendungsumgebungen*** prägen diese Schicht, da solche Frameworks eine komplette Infrastruktur für konkrete Anwendungen erstellt.

Ergänzt wird die Kategorie Application-Frameworks um ***Konzepte***, am besten in einzelnen Mikro-Frameworks verpackt, aus denen sich letztlich das große Framework zusammensetzt. Zielt die Anwendungsumgebung auf Arbeitsplätze für mit kooperativen Tätigkeiten, so gibt es sicher auch Konzepte für deren Koordination. Zu unterscheiden ist dies von Design Pattern als Muster zur technischen Entkopplung von Komponenten innerhalb der Anwendung oder eines Frameworks. Methodische Konzepte werden immer durch äußere Situation motiviert, nicht durch softwaretechnische Anforderungen innerhalb des Entwurfes.

Bleibt als letzte Dimension dieser Aufzählung die **fachliche**. Domänen-Frameworks gehören hierzu, und Mikro-Frameworks mit Bezug zum Kontext der Anwendungen. Die Applikationen selbst sind nicht Teil einer Bibliothek, aus Sicht von WAM kann es aber fertige Subwerkzeuge geben, die in späteren Projekten wiederverwendet werden.

Da die BibV30 bisher auf keine bestimmte Domäne ausgerichtet ist, gibt es solche fachlichen Framework - Komponenten nicht. Am Ende des Abschnittes greife ich dieses Thema auf und diskutiere die Ergänzung derartiger Komponenten in der BibV30, bzw. zeige den prinzipiellen Unterschied zwischen dieser Framework - Sammlung und einer Bibliothek mit festgelegtem Anwendungskontext.

5.2.2 Umsetzung der Struktur

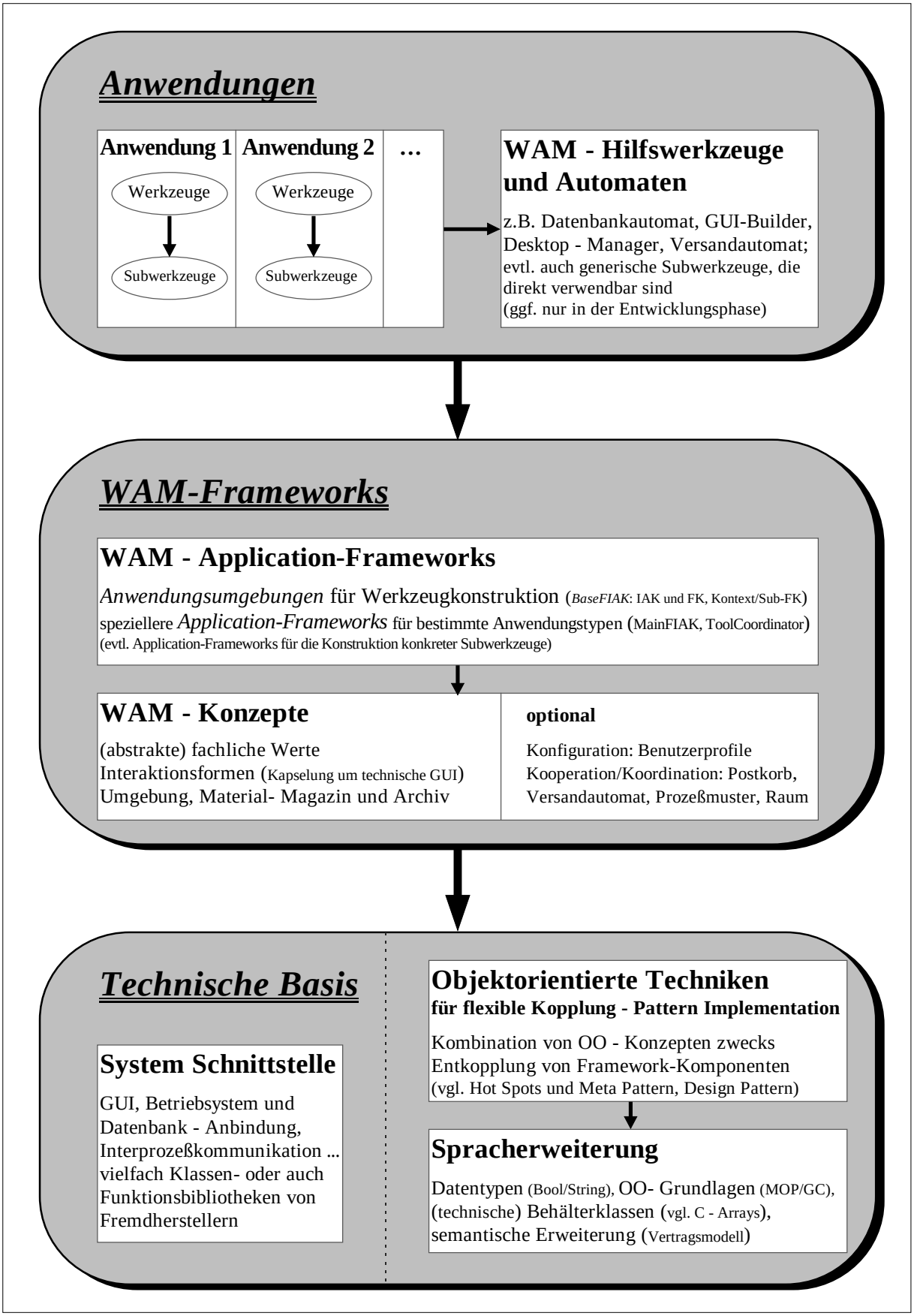


Abbildung 5-1: Strukturschema für die BibV30

Abbildung 5-1 zeigt die Struktur der BibV30. In der Grobstruktur handelt es sich um drei große Schichten, die aufeinander aufbauen. Unter Verwendung der ***Technischen Basis*** stellen die ***WAM-Frameworks*** die softwaretechnische Design - Komponente der für die ***Anwendung***sentwicklung empfohlenen Methode WAM bereit.

Schichten strukturieren das Framework, im Zusammenhang mit diesem Konzept ist aber nicht gemeint, daß jeweils die nächst höhere Schicht nur die Funktionalität der direkt unterliegenden benutzt. WAM-Frameworks sind also nicht als Kapselung der technischen Basis gedacht, die Anwendungen verwenden vielmehr diverse Klassen und Mikro-Frameworks dieser Schicht, Spracherweiterungen wären ansonsten auch gar nicht verständlich, Behälterklassen nützen vor allem der Anwendung. Außerdem umfaßt das Strukturschema keine Vorgaben betreff der Beziehungen zwischen Frameworks innerhalb einer Schicht oder verschiedener Schichten. Die Pfeile in Abbildung 5-1 bedeuten nicht etwa „Benutzung“ im objektorientierten Sinne zwischen Klassen. Gerade die andere Variante, erben von Klassen eines Frameworks oder ergänzen von Parameterklassen macht die Stärke objektorientierter Frameworks aus.

Innerhalb der Schichten finden sich die beschriebenen Kategorien, konkrete Komponenten der BibV30 sind eingeordnet. Die implementierten *Spracherweiterungen* tauchen bereits unter 5.2.1 auf, einige Klassen (z.B. Bool) entfallen voraussichtlich, sobald alle Compiler den neuen C++ Standard erfüllen. Zu der Bedeutung des MOP für die BibV30 gibt 4.3.1 Auskunft, einen Garbage Collector (GC) gibt es leider bisher nicht. Technische Behälterklassen meinen konkret die ConLib (siehe 4.3.2).

Objektorientierte Techniken verwenden in der BibV30 die Spracherweiterungen, seien es die Behälter oder auch das Vertragsmodell. In diesen Bereich eingeordnet sind Implementationen von Design Pattern, in der BibV30 also (vgl. Abschnitte 4.3.2 und 4.3.3) Command, Observer, Chain Of Responsibility, Late Creation[18], Singleton und zukünftig vielleicht ein Atomizer.

System - Schnittstellen zählen zur technischen Basis, jedoch ohne direkte Beziehung zu den Schichten der Spracherweiterungen und objektorientierten Mechanismen. Es handelt sich um GUI - Bibliotheken von Fremdanbietern, derzeit OSF/Motif, Tcl/TK und wxWindows. Der Anschluß an eine Datenbank wird an dieser Stelle ergänzt werden, ebenso evtl. die technischen Grundlagen für verteilte Systeme. Sollten zukünftig eigene System Schnittstellen für die BibV30 entwickelt werden, profitieren diese selbstverständlich von Spracherweiterungen und objektorientierten Techniken. Die fehlende Verbindung beschreibt nur den jetzigen Zustand, da Systemschnittstellen unabhängig von der BibV30 erstellt sind.

Die Integration dieser fremden, eigentlich nicht zur Bibliothek selbst gehörenden Komponenten in das Strukturschema halte ich für angemessen, da diese Struktur vor allem eine Orientierung, auch für neue Benutzer, gibt. Auch wenn es sich um extern entwickelte Klassen handelt, ist die BibV30 ohne diese unvollständig.

---

[18] Leider ist späte Erzeugung wegen starker Verknüpfung mit dem MOP derzeit eine Schicht tiefer zu finden (4.3.1).

Die zweite Ebene der BibV30 sind die ***WAM-Frameworks***. Im Gegensatz zu früheren Kapiteln wird der Begriff in engerem Sinne gebraucht, nicht für die komplette Framework - Sammlung, sondern nur für deren am WAM-Kontext orientierten Kern. Bei Übertragung des Struktur- Vorschlages auf Bibliotheken unabhängig von der WAM-Metapher wäre *methodische* Ebene die passende Bezeichnung, dies entspricht dem Begriff in 5.2.1.

Innerhalb der Ebene sind zwei Schichten zu differenzieren. Die Basis bilden *WAM- Konzepte*, also z.B. Metaphern (vgl. Abschnitt 2.3). Gegenwärtig dominiert in der BibV30 die Kapselung des Fenstersystems durch Interaktionstypen diese Schicht, daneben steht die Umgebung. Zu den Interaktionstypen gehören Command - Klassen, die als Spezialisierung auf Grundlage des Command - Pattern in der technischen Basis realisiert sind.

Außerhalb der BibV30 wurden Postkörbe, Versand- und Datenbankautomaten mit Archiven und Magazinen sowie Prozeßmuster implementiert, die Integration fehlt aber noch. Konzepte für WAM sind beispielsweise fachliche Werte, die genannten Prozeßmuster und Postkörbe, funktionelle Rollen, Benutzerprofile/rechte, und Gruppenarbeitsplätze, Lokalität und verteilte Umgebungen ([Bleek97], [RW97], [KRW96]).

Für die Ausführungen im folgenden Abschnitt sind *optionalen Konzepte* wichtig, in Kapitel 4 wurden z.B. Prozeßmuster zur Koordination kooperativer Tätigkeiten ausführlich diskutiert. Gemäß den Zielen der Strukturierung muß das Schema vorgeben, an welche Position neue Komponenten gehören. Die Beispiele in Abbildung 5-1 helfen bei der Orientierung, welche Mikro-Framework absehbar tatsächlich entstehen, ist damit nicht gesagt.

Über den WAM- Konzepten stehen Application-Frameworks, in der BibV30 eine allgemeine Anwendungsumgebung und zwei spezieller Versionen für unterschiedliche Anwendungstypen (vgl. 4.3.1: BaseFIAK, MainFIAK, ToolCoordinator). Letztere sind durch Ableitung von den Klassen der allgemeinen Version implementiert, innerhalb der Schicht gibt es also nicht nur Benutzt - Beziehungen zwischen Frameworks, sondern auch Vererbung.

An dieser Stelle greife ich kurz meinen Vorschlag aus 2.3.3 wieder auf, Frameworks für die Konstruktion angepaßter Subwerkzeuge in die Bibliothek aufzunehmen. Diese verstehe ich als Application-Frameworks für Teil-Werkzeugen als Alternative zu generischen Subwerkzeugen.

Oberhalb der Frameworks stehen als letzte Ebene die Anwendungen. Werkzeuge nach WAM bestehen aus Kontext und Subwerkzeugen, diese Konstruktion unterstützten die Application-Frameworks. Neben den konkreten Anwendungen mit jeweiligen Werkzeugkomponenten sind Automaten und Subwerkzeuge als direkte Bestandteile der Bibliothek vorhanden, zur Zeit z.B. ein in die Anwendung integrierbarer GUI-Builder für Layout - Anpassungen zur Laufzeit. Als Beispiel für generische Subwerkzeuge sei ein Lister genannt (1 aus N Selektion).

Alle Subwerkzeuge und Automaten werden selbst unter Verwendung des Application-Framework erstellt und stehen auf einer Ebene mit Werkzeugen und Subwerkzeugen konkreter Applikationen. Eine harte Abgrenzung zwischen allgemeinen und speziellen

Komponenten ist insofern nicht erforderlich, als Subwerkzeuge gemäß WAM ohnehin als wiederverwendbare Einheiten verstanden werden bzw. so zu entwerfen sind.

5.2.3 Fachliche Framework-Komponenten

Auf Grundlage der BibV30 (bzw. Vorgängerversionen) wurden verschiedene Anwendungen erstellt. Dabei handelte es sich jeweils um einzelne Projekte, nicht um eine Anwendungsfamilie in gegebener Domäne[19]. Die BibV30 ist nicht für einen bestimmten Kontext konstruiert bzw. zugeschnitten, vielmehr dient sie als Plattform für Applikationen unterschiedlicher Bereiche.

Fachliche Frameworks beinhalten gemeinsame Konzepte verschiedener Applikationen, jeweils aus mehreren Anwendungen heraus abstrahiert. Einzelanwendungen implementieren selbst ihre Material-, Aspekt- und Werkzeugklassen, eine Wiederverwendung dieser speziellen Elemente in anderen Anwendung steht nicht an. Der gemeinsame Kontext fehlt, es wurden keine abstrakten fachlichen Komponenten für die BibV30 aus diesen Anwendungen heraus gewonnen.

Zwischen WAM- Konzepten in der Framework - Sammlung und dem Anwendungskontext besteht der Bezug, daß optionale Komponenten gerade nicht für jede Domäne geeignet sind, sondern aus konkreten Anforderungen heraus entwickelt wurden. Beispielsweise kommt aus dem Bereich der Krankenhaussoftware der Bedarf an Gruppenarbeit, während in Banken die Kooperationsform über Vorgangsmappen mit Prozeßmustern entworfen wurde.

Weitere Entwicklungen im Rahmen der BibV30 sollen sich auf die Integration eben solcher Komponenten mit unterschiedlichen Anwendungsbereichen konzentrieren. Diese Konzepte sind Erweiterungen der WAM-Metaphern und keine Ad Hoc Lösungen für eine Anwendung. Der Entwurf strebt Mechanismen für den Einsatz in verschiedenen Domänen mit ähnlichen Anwendungssituationen an. Die Mikro-Frameworks sind nicht auf *eine* fachliche Domäne zugeschnitten, aber auch nicht für *beliebige* Anwendungsbereiche einsetzbar. Verbunden mit der Konstruktion derartiger Mechanismen ist der Wunsch, diese in Projekten geeigneter Domänen zu testen, eine Weiterentwicklung der BibV30 in fachlicher Richtung wäre die Folge.

Abbildung 5-1 beschreibt einerseits den heutigen Zustand, andererseits sieht die Struktur die Erweiterung vor, soweit diese im technischen oder methodischen Bereich liegen. Hier besteht noch einiger Bedarf, Komponenten sind softwaretechnisch zu verbessern, außerhalb der BibV30 entwickelte Konzepte zu integrieren und neue zu entwerfen. Insgesamt kann das die Bibliothek jedoch als hinreichend ausgereift betrachtet werden, um deren Einsatz in größeren Projekten einzuleiten. Damit stelle ich die Forderung auf, zukünftig den fachlichen Bereich der Framework - Sammlung anzugehen.

---

[19] Anmerkung: in der Anfangsphase der Entwicklung wurde daran gedacht, am Arbeitsbereich Werkzeuge für eine Software-Entwicklungsumgebung zu erstellen. Bei einer Fortsetzung dieses Projektes hätte sich eine auf diese Domäne spezialisierte Bibliothek ergeben können, dies ist aber nicht geschehen.

Im Unterschied zu Domänen-Frameworks in kommerziellen Organisationen soll die BibV30 offen bleiben für alle Anwendungen, für die sich die Methode WAM gemäß ihres Leitbildes eignet, es soll nicht *ein* Domänen-Framework entstehen, sondern die Bibliothek als eine Art Meta-Framework die Basis diverser Domänen bilden.

Zur Erläuterung der neuen Schicht in Abbildung 5-2 ist neben der Aufteilung in eigentlich mehrere getrennte Schichten nebeneinander deren innere Zusammensetzung aus Domänen-Frameworks und den Domänen - Konzepten zu begründen, außerdem der Bezug dieser Schicht zu den WAM-Frameworks zu klären.

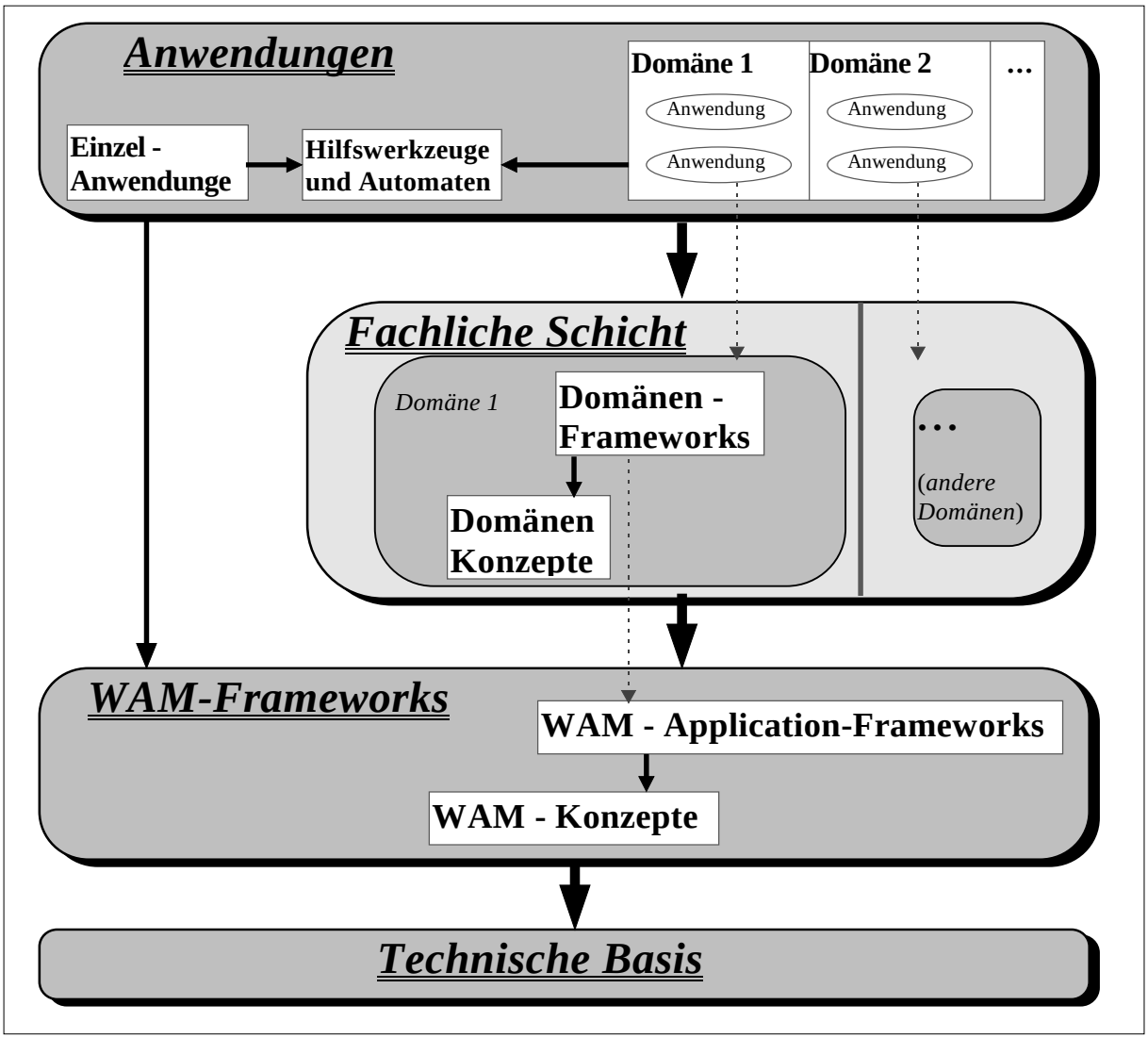


Abbildung 5-2: Schema für BibV30 mit Erweiterung für Komponenten aus Anwendungsdomänen

Mit Domänen-Konzepten ist der fachliche Kern eines Anwendungsbereiches gemeint. Hier finden sich einzelne Klassen (z.B. Materialtypen und Aspekte) und auch Mikro-Frameworks.

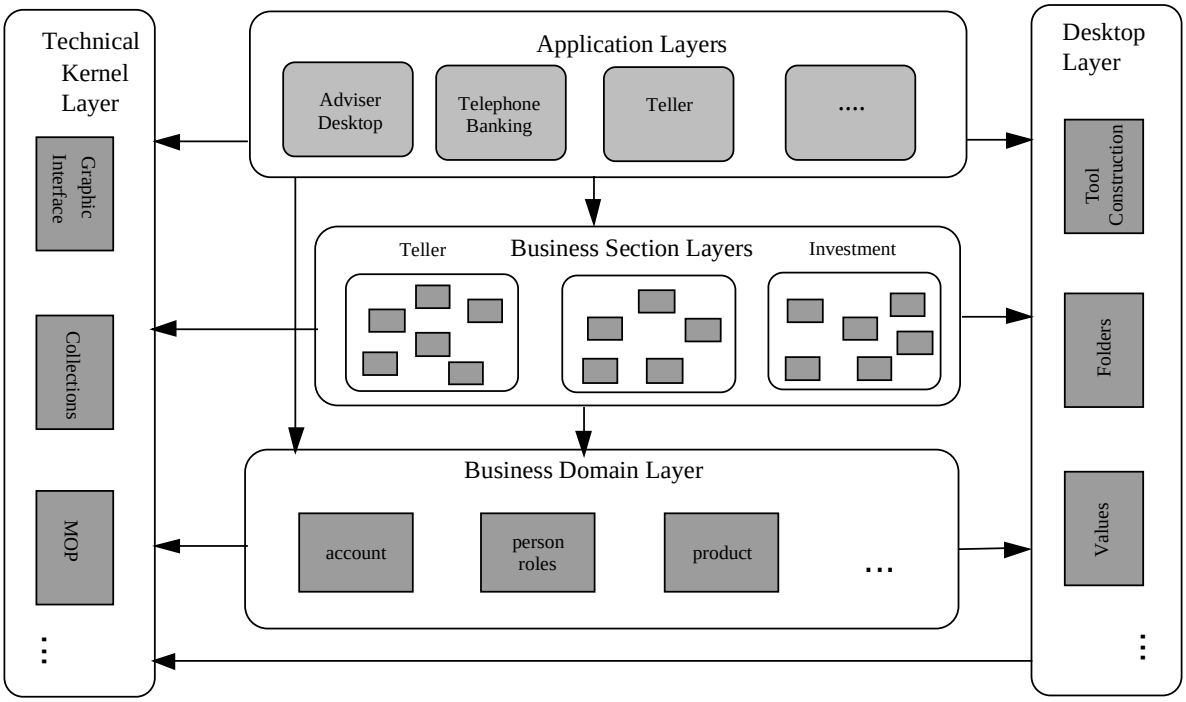
Domänen-Frameworks leiten sich von WAM - Application-Frameworks ab, erweitern deren Infrastruktur um neue Konzepte der Domäne und standardisieren Bereiche der Anwendungen (vgl. Abschnitt 2.1). Für die Infrastruktur stehen alle WAM- Konzepte zur Verfügung, jede Domäne trifft hieraus eine eigene Auswahl.

Zentrales Anliegen dieser Struktur ist die Einführung neuer Sichten auf die Bibliothek. Für die einzelnen Domänen ergibt sich jeweils einen Satz an relevanten Komponenten, in der WAM-Schicht befinden sich die Komponenten aller Domänen. Im folgenden Abschnitt diskutiere ich als Alternative zur Integration mehrerer fachlicher Bereiche in die BibV30 die Verwendung dieser Bibliothek zur Konstruktion getrennter Bibliotheken für je eine Domäne.

5.3 Eine Verbindung zwischen der BibV30 und GEBOS

Nachfolgend beschreibe ich die Beziehung zwischen einer Domänen- Bibliothek und der BibV30 an einem Beispiel. Und zwar vergleiche ich die Struktur des GEBOS- Systems mit dem Vorschlag für die BibV30 im letzten Abschnitt. GEBOS wurde von der RWG für eine Anwendungsfamilie im Bankenbereich entwickelt, die Anwendungsentwicklung bei der RWG[20] folgt der Methode WAM

Meine Betrachtung des GEBOS- Systems stützt sich auf [BGKLRZ97]. Die Autoren stützen sich auf diese Framework-Sammlung bei der Untersuchung von Framework-Entwicklungen für große Systeme. Die Beschreibung der Struktur (siehe Abbildung 5-3) ist direkt dem Artikel entnommen, ich werde einen Vergleich mit der BibV30 vornehmen, und dann erklären, wie ich mir die Konzeption eines diesem vergleichbaren Domänen-Frameworks unter Benutzung der BibV30 vorstelle. Zuerst einige Hinweise zu der Darstellung selbst, um diese meinem Strrukturvorschlag gegenüberzustellen.



[20] Die RWG erstellt Software für einen Verbund unabhängiger Banken, nähere Informationen zum GEBOS- System, dessen Entwicklung und das organisatorische Umfeld in der RWG gibt z.B. [BGKZ96]

Abbildung 5-2: Struktur des GEBOS - Frameworks gemäß [BGKLRZ97]

Im mittleren Bereich der Abbildung finden sich alle fachlichen GEBOS Elemente. Dieser Teil fehlt in der BibV30 gemäß Abb. 5-1, entspricht der fachlichen Schicht in Abb. 5-2. Es ergibt sich in der Darstellung offensichtlich eine andere Gewichtung der beteiligten Komponenten, der fachliche Bereich überwiegt im GEBOS- System und wird weiter strukturiert.

GEBOS ist eine große Anwendungsfamilie, die Domäne „Bank“ wird nicht durch eine einzige Schicht unterstützt, sondern zuvor in Geschäftsbereiche gegliedert (Business Section Layers, z.B. Anlagengeschäft oder Immobilien- Finanzierung). Deren Gemeinsamkeiten, die das Bankgeschäft als Ganzes ausmachen, ergeben eine eigene Schicht (Business Domain Layer).

Im Vergleich zur Abbildung 5-2 bestehen zwar auch mit den einzelnen Geschäftsbereichen mehrere Domänen innerhalb der Bibliothek, nur sind diese nicht voneinander unabhängig. Dies ist die Annahme für eine weiterentwickelte BibV30, im GEBOS- System dagegen sollen insgesamt Banken unterstützt werden - und die *haben* eine gemeinsame Geschäftsgrundlage.

Meine Annahme, in eine Bibliothek Elemente für verschiedene Branchen zu integrieren, ist auf derart große Systeme kaum übertragbar. Ergebnis der Konzentration auf Banken ist eine Größe, deren Handhabung Schwierigkeiten mit sich bringt. Also diskutiere ich nachfolgend nicht etwa eine Integration des GEBOS- System in die Bibliothek, sondern die Konstruktion eines neuen Systems unter Verwendung der BibV30. GEBOS bzw. die Domäne Bank dient als Beispiel, wie es auch Krankenhäuser oder Autovermietungen wären.

Implementation eines Domänen-Frameworks für Banken mit der BibV30

Die BibV30 richtet sich nicht auf eine bestimmte Domäne, sondern soll gewissermaßen als „Meta - Domänen-Framework“ für beliebige Anwendungsbereiche anpaßbar sein. Konkret ersetzt die Bibliothek in einer Struktur nach Abbildung 5-3 sowohl die linke als auch die rechte Säule und vereinfacht die Implementation fachlicher Frameworks (mittlere Säule).

In der technischen Basis, untere Schicht meiner Struktur, bietet die BibV30 ähnliches, wie das GEBOS- System erwartet (linke Säule). Als Erweiterung kommt die Kategorie der „objektorientierten Mechanismen“ hinzu, also die Implementationen von Design Pattern.

WAM-Frameworks decken den Desktop Layer ab, wobei an diesem Punkt nicht die komplette BibV30, sondern nur einige Teile zu benutzen sind. Als Standardbeispiel dieser Arbeit nenne ich wieder die Kooperation über den Austausch von Vorgangsmappen (mit Prozeßmustern) und Postkörbe in der Bank gegenüber Gruppenarbeitsplätzen im Krankenhaus.

Stellt sich heraus, daß die softwaretechnischen Metaphern und Konzepte der Methode WAM für einige Arbeitszusammenhänge nicht genügen, so kommt eine Erweiterung des Repertoires infrage. Die Implementation eines neuen Konzeptes sollte nicht nur im Domänen-Framework, sondern auch direkt in der BibV30 erfolgen, also die WAM-Frameworks erweitern.

Der fachliche Bereich muß in jeder Anwendungsdomäne neu realisiert werden. Hierzu zählen z.B. Materialklassen und Aspekte sowie spezifische Subwerkzeuge. Anwendungsumgebungen der BibV30 werden konkretisiert, durch Ableitung entstehen neue Domänen-Frameworks. Wie im Abschnitt 2.1 dargelegt, ist gerade die Möglichkeit einer Standardisierung großer Teile der Anwendungen die eigentliche Stärke eines Domänen-Frameworks.

Für die Konstruktion eines Banken - Frameworks ist somit die komplette Infrastruktur der BibV30 verfügbar, angefangen von Spracherweiterungen über objektorientierte Mechanismen und die Systemschnittstelle bis hin zur methodischen Schicht der WAM-Frameworks.

Die Darstellung in [BGKLRZ97] (Abbildung 5-3) verdeutlicht, daß diese Komponenten einen bedeutenden Teil der Entwicklung einer Anwendungsbibliothek ausmachen, auch wenn deren Schwerpunkt die fachliche Bereiche bleiben. Letztere Arbeit ist jeweils spezifische zu leisten, eine allgemeine Bibliothek wie die BibV30 als Basis entlastet von den übrigen Aufgaben, die Entwickler können sich auf die Infrastruktur verlassen und auf ihre Domäne konzentrieren.

Bis auf weiteres wird die konkrete Verwendung der BibV30 in der hier beschriebenen Form nicht infrage kommen, Vorbedingung wäre erst einmal ein praktischer Einsatz, wie in 5.2.3 beschrieben und gefordert. Dies sollte in nächster Zeit forciert werden, den Anfang können in bewährter Form (vgl. Anfangsphase der Bibliotheks-Entwicklung) studentische Projekte als Lehrveranstaltungen bilden, mit der Fortsetzung in Studien- und Diplomarbeiten. Dabei ist sicherzustellen, daß eine gemeinsames große Aufgabe anstelle vieler Einzelprojekte tritt.

Kapitel 6 Fazit und Ausblick

Objektorientierte Technologie verspricht wiederverwendbare Software - Komponenten. Um dieses Versprechen einzulösen, kombinieren moderne Frameworks Mechanismen der Programmiersprachen und den Ansatz der Design Pattern, orientieren sich eventuell zudem an Methoden zur Softwareentwicklung.

Ein Beispiel solcher Frameworks sind die am Arbeitsbereich Softwaretechnik entwickelten C++ Bibliotheken mit dem Namen BibV30. Entstanden über einen Zeitraum mehrerer Jahre in kooperativer Tätigkeit hilft diese auf die Werkzeug und Material Metapher ausgerichtete Framework - Sammlung in der Lehre am Arbeitsbereich, dient Diplomarbeitern als Infrastruktur für die Anwendungsentwicklung - und ist vor allem eine Plattform für die konzeptionelle und softwaretechnische Erweiterung der Methode WAM.

Über die Jahre hat die BibV30 einen Funktionsumfang erreicht, den ich in dieser Arbeit an anderen Produkten messe, z.B. ET++ und MFC. Es ist meine Überzeugung, auch als Ergebnis dieses Vergleichs, daß die Bibliothek künftig stärker als Produkt betrachtet werden sollte. In den letzten Monaten hat eine Gruppe von Entwicklern (Studenten und Mitarbeiter des Arbeitsbereiches) einige Schritte vollzogen, um die BibV30 mit der wichtigsten Eigenschaft aller Frameworks auszustatten: unabhängig von aller Funktionalität ist die Verständlichkeit einer Softwarekomponente Voraussetzung für deren Wiederverwendung, der Nutzen infolge der Anwendung muß den Aufwand der Einarbeitung übertreffen.

Diese Diplomarbeit zeigt zweierlei auf, die Mittel und Ergebnisse des Redesigns und eine mögliche zukünftige Verwendung der BibV30. Grundlage der konzeptionellen und technischen Weiterentwicklung ist die Annahme, durch überschaubare Komponenten mit loser, auf gut dokumentierbaren Design Pattern basierender Kopplung eine Anwendungsumgebung einfacher zu gestalten, das Verhältnis zwischen Lernaufwand und flexibler Funktionalität zu bessern.

Zur softwaretechnischen Umsetzung der Methode WAM besteht ein Repertoire an Metaphern und Konzepten, daß ständig aufgrund neuer Anwendungssituationen erweitert wird. Ein großer Abschnitt dieser Arbeit beschreibt, wie ich ein bewährtes (Grafik-) Konzept aus einer fremden Entwicklung in die Welt der Werkzeug und Material Metapher übertragen und die BibV30 damit aufwerten konnte. Die ständige Erweiterung der Methode und des Frameworks deutet die Richtung weiterer Entwicklungsarbeit an.

Zielvorgabe für die Bibliothek ist die möglichst umfassende Anwendungsunterstützung in dem durch das WAM- Leitbild gesteckten Rahmen. Konsequenz ist eine Bibliothek aus sehr vielen Komponenten, die damit wieder das Ziel der Verständlichkeit gefährdet. Als Lösung schlage ich als ersten Schritt eine Kategorisierung und Ordnung dieser Bestandteile vor, der zweite Schritt orientiert sich entsprechend der unterschiedlichen Herkunft und Zielrichtung der Komponenten auf eine logische Aufspaltung der Framework - Sammlung.

Gemeint ist ein Rückbezug auf die Anwendungsdomänen, aus denen sich neue Konzepte ergeben haben. Motiviert wurden einige Kooperationsformen durch konkrete Anwendungen in Banken, andere durch die Arbeit im Krankenhaus. Umgekehrt lassen sich diese Bereiche mit spezifischen Domänen-Frameworks weitaus besser unterstützen, als mit einer allgemeinen Anwendungsumgebung für WAM.

Verbunden mit dem Richtungswechsel zu Frameworks für bestimmte Domänen ist aber auch die Forderung nach praktischen Einsatz der BibV30 in größeren, zusammenhängenden Projekten. Nur so läßt sich das Potential nutzen, Domänen wirklich erfassen, Einzelanwendungen leisten dies nicht. Konsequenz dieser Empfehlung ist ein stärkerer Produkt - Charakter der Bibliothek, Voraussetzung dagegen noch höheres Engagement seitens der Studenten, folglich auch der Mitarbeiter am Arbeitsbereich. Ich selbst will dieses Thema weiter verfolgen, andere erkennen es hoffentlich auch als lohnen. Und vielleicht findet dann jemand einen geeigneten Namen für dieses Produkt, das derzeit noch BibV30 heißt.

Diese Diplomarbeit faßt eine Entwicklungsphase zusammen, kennzeichnet deren konzeptionellen Hintergrund und bewertet das Ergebnis. Mit der Übernahme eines für die Bibliothek neuen Konzeptes der direkten Manipulation für Grafikkomponenten, ermutige ich zum Bau von „Brücken“ zwischen der WAM - Welt und fremden Entwicklungen, die Konzepte sind vielleicht nicht direkt verträglich, aber vielfach doch anpaßbar.

Mit der Strukturierung der BibV30 stelle ich den Überblick wieder her. Benutzer können sich daran orientieren, Entwickler ihre Ergebnisse einordnen. Was noch fehlt sind eben diese Erweiterungen der Framework - Sammlung, ist die Fertigstellung lange geplanter sowie neuer Komponenten. Erst diese können meine Arbeit evaluieren, die Benutzer bewerten und prüfen meine Auffassung zur Verständlichkeit.

Völlig offen ist meine Hoffnung, die C++ Bibliotheken praxisnah zu verwenden und meine theoretische und am Beispiel vollzogene Einbettung in ein Domänen-Framework umzusetzen. Den Rahmen für weitere Entwicklung der BibV30 und deren Einsatz in zusammenhängen Arbeiten gibt derzeit eine Lehrveranstaltung am Arbeitsbereich, im kommenden Jahr kann dann wohl Bilanz gezogen werden.

Literaturverzeichnis

[AIS77] Christopher Alexander, Sara Ishikawa, Murray Silverstein, mit Shlomo Angel, Ingrid Fiksdahl-King, Max Jacobson: „A Pattern Language: Towns, Buildings, Construction“, Oxford University Press, New York, 1977

[AGO95] Robert Allan, David Garlan, John Ockerbloom: „Architectural Mismatch: Why Reuse Is So Hard“, IEEE Software 12, 6 (November 1995)

[Alexander79] Christopher Alexander: „The Timeless Way of Building“, Oxford University Press, New York, 1979

[Andert95] Glenn Andert: „Frameworks in Taligent’s CommonPoint“ in [Lewis95]

[BG96] Ulrich Brammer, Markus Göhmann: „Objektorientierung und Persistenz: Ein Anforderungskatalog an die Verwaltung von Materialien in Einzelplatz-Arbeitsumgebungen aus fachlicher und technischer Sicht“, Studienarbeit am Arbeitsbereich Softwaretechnik, Fachbereich Informatik, Universität Hamburg, Juli 1996 (http://swt-www.informatik.uni-hamburg.de/~1brammer)

[BBE95] Andy Birrer, Walter R. Bischofberger, Thomas Eggenschwiler: „Wiederverwendung durch Frameworktechnik - vom Mythos zur Realität“, OBJECTspektrum, September/Oktober 1995

[BBLSSZ94] Dirk Bäumer, Walter R. Bischofberger, Horst Lichter, Matthias Schneider-Hufschmidt, Veronica Sedlmeier-Scholz, Heinz Züllighoven: „Prototyping von Benutzungsoberflächen“, Mitteilung FBI-HH-M 242/94, Fachbereich Informatik, Universtät Hamburg, 1994

[BGKLRZ97] Dirk Bäumer, Guido Gryczan, Rolf Knoll, Carola Lilienthal, Dirk Riehle, Heinz Züllighoven: „Framework Development for Large Systems“, *angenommen für OOPSLA’ 97*

[BGKZ96] Dirk Bäumer, Guido Gryczan, Rolf Knoll, Heinz Züllighoven: „Large Scale Object-Oriented Software-Development in a Banking Environment - An Experience Report“ in [ECOOP96]

[BJ94] Kent Beck, Ralph E. Johnson: „Pattern generate architectures“ in [ECOOP94]

[Bleek97] Wolf-Gideon Bleek: „Erweiterung des Umgebungsbegriffs für Anwendungssysteme, die nach der WAM Metapher konstruiert wurden“, Diplomarbeit am Arbeitsbereich Softwaretechnik, Fachbereich Informatik, Universität Hamburg, 1997

[BMRSS96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal: „Pattern-Oriented Software Architecture - A System of Patterns“, Wiley, 1996

[BMRSZ96] Dirk Bäumer, Daniel Megert, Dirk Riehle, Wolf Siberski, Heinz Züllighoven: „Serializer“ / „The Atomizer - Efficiently Treaming Object Structures“ in [PLoP96]

[BR96] Dirk Bäumer, Dirk Riehle: „Late Creation - A Creational Pattern“ in [PLoP96]

[Calder95] Paul Calder: „InterViews: A Framework for X-Windows“ in [Lewis95]

[Coplien92] James O. Coplien: „Advanced C++ - Programming Styles and Idioms“, Addison-Wesley, 1992

[ECOOP93] O. Nierstrasz (Ed.): „ECOOP '93 - Object-Oriented Programming“, Proceedings of 7th European Conference, Kaiserslautern, Germany, July 1993, Lecture Notes in Computer Science 707, Springer-Verlag, Berlin Heidelberg New York, 1993

[ECOOP94] Mario Tokoro, Remo Pareschi (Eds.): „ECOOP '94 - Object-Oriented Programming“, Proceedings of 8th European Conference, Bologna, Italy, July 1994, Lecture Notes in Computer Science 821, Springer-Verlag, Berlin Heidelberg New York, 1994

[FH96] Stephen Friedrich, Keno Hamer: „Entwicklung eines Werkzeugs zur Visualisierung des Objektverhaltens in C++ - Programmen“, Studienarbeit am Arbeitsbereich Softwaretechnik, Fachbereich Informatik, Universität Hamburg, 1996 (http://stw-www.informatik.uni-hamburg.de/~1hamer)

[Floyd94] Christiane Floyd: „Arbeitsunterlagen zur Lehrveranstaltung Einführung in die Softwaretechnik“, Universität Hamburg, 1994

[Fricke96] Nils Fricke: „Das Projekttagebuch - Ein Fallbeispiel für eine verteilte Anwendung nach WAM“, Studienarbeit am Arbeitsbereich Softwaretechnik, Fachbereich Informatik, Universität Hamburg, 1996 (http://stw-www.informatik.uni-hamburg.de/~1fricke)

[Fröse96] Frank Fröse: „Konzepte der Fenstersystemanbindung mit Interaktionstypen“, Studienarbeit am Arbeitsbereich Softwaretechnik, Fachbereich Informatik, Universität Hamburg, 1996 (http://stw-www.informatik.uni-hamburg.de/~1froese)

Literaturverzeichnis

[Gamma92] Erich Gamma: „Objektorientierte Software-Entwicklung am Beispiel von ET++“, Springer-Verlag, Berlin Heidelberg, 1992

[GHJV93] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: „Design Patterns: Abstraction and Reuse of Object-Oriented Design“ in [ECOOP93]

[GKZ93] Guido Gryczan, Klaus Kilberth, Heinz Züllighoven: „Objektorientierte Anwendungsentwicklung“, Vieweg, Braunschweig/Wiesbaden, 1993

[GWZ96] Guido Gryczan, Martina Wulf, Heinz Züllighoven: „Prozessmuster für situierte Koordination kooperativer Arbeit“ in H. Krcmar, H. Lewe, G. Schwabe (Hrsg.): „Herausforderung Telekooperation, D-CSCW’96“, Springer 1996

[Görtz97] Thorsten Görtz: „Abstraktion der GUI-Komponente in einem Rahmenwerk“, Diplomarbeit am Arbeitsbereich Softwaretechnik, Fachbereich Informatik, Universität Hamburg, voraussichtlich 1997

[GoF95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: „Design Patterns: Elements of Reusable Object-Oriented Software“, Reading, Massachusetts, Addison-Wesley, 1995

[Goldberg83] Adele Goldberg: „Smalltalk-80: The Interactive Programming Environment“, Addison-Wesley, Menlo Park, 1984

[Gryczan95] Guido Gryczan: „Situierte Koordination computergestützter qualifizierter Tätigkeit über Prozeßmuster“, Dissertation, Universität Hamburg, 1995 (bzw. „Prozessmuster zur Unterstützung kooperativer Tätigkeit“, Deutscher Universitäts-Verlag, DUV:Informatik, Wiesbaden, 1996)

[Hess93] Hauke Hess: „Grundsteine für eine STEPS-Werkzeugumgebung“, Diplomarbeit am Arbeitsbereich Softwaretechnik, Fachbereich Informatik, Universität Hamburg, Juni 1993

[JF88] Ralph E. Johnson, Brian Foote: „Designing Reusable Classes“, The Journal of Object-Oriented Programming, Vol. 1, No. 2, 1988, Seiten 22-35

[Johnson92] Ralph E. Johnson: „Documenting frameworks using patterns“ in [OOPSLA92]

[Kornstädt94] Andreas Kornstädt: „Ein objektorientierter Interaktionstyp für die dreidimensionale Darstellung von dynamischen Statistiken unter Motif in C++“, Studienarbeit am Arbeitsbereich Softwaretechnik, Universität Hamburg, 1994 (http://swt-www.informatik.uni-hamburg.de/~1kornsta)

[KRW96] Anita Krabbel, Sabine Ratuski, Ingrid Wetzel: „Objektorientierte Analysetechniken für übergreifende Aufgaben“ in Jürgen Ebert (Hrsg.): „GI-Fachtagung Softwaretechnik ‘96“

[LCITV92] M. A. Linton, P. R. Calder, J. A. Interrante, S. Tang, John M. Vlissides: „InterViews Reference Manual, Version 3.1“, Stanford University, 1992

[Lewis95] Ted Lewis (Editor): „Object-oriented application frameworks“, Manning Publications Co., Greenwich, 1995

[Lilienthal95] Carola Lilienthal: „Konzeption und Realisierung eines an der Anwendungssprache orientierten Hilfesystems nach der Werkzeug-Material Metapher“, Diplomarbeit am Arbeitsbereich Softwaretechnik, Universität Hamburg, 1995 (http://swt-www.informatik.uni-hamburg.de)

[Lilienthal97] Carola Lilienthal: „Dokumentation der BibV30“, im WWW, Arbeitsbereich Softwaretechnik, Universität Hamburg, 1995 (http://swt-www.informatik.uni-hamburg.de)

[LS96] Carola Lilienthal, Wolfgang Strunk: „Documenting frameworks by visualizing dynamics“, Beitrag zur TOOLS’96 (http://swt-www.informatik.uni-hamburg.de) *Proceedings noch nicht erschienen*

[Meyer88] Bertrand Meyer: „Object-oriented Software Construction“, Prentice Hall, 1988

[Meyer94] Bertrand Meyer: „Reusable Software - The Base Object-Oriented Component Libraries“, Prentice Hall (UK), Hertfordshire, 1994

[MS96] David R. Musser, Atul Saini: „STL Tutorial and Reference Guide: C++ programming with the Standard Template Library“, Addison-Wesley, Reading, Massachusetts, 1996

[OOPSLA92] „Object-Oriented Programming Systems, Languages, and Applications“, Conference Proceedings, ACM Press, Vancouver, British Columbia, Canada, 1992

[Perry96] Ron Perry: „The Displayer / Data Pattern - Language: A Software Engineering Technique for GUI/Application Decoupling“, Department of Computer Science, Tel-Aviv University, 1996

Literaturverzeichnis

[PLoP95] John M. Vlissides, N. Korth, James O. Coplien (Eds.): „Pattern Languages of Program Design“, Addison-Wesley, 1996 (a book publishing the reviewed Proceedings of the Second International Conference on Pattern Languages of Programming, Monticello, Illinois, 1995)

[PLoP96] Robert C. Martin, Dirk Riehle, Frank Buschmann (Eds.): „Pattern Languages of Program Design 3“, Addison-Wesley, 1997

[Pree94] Wolfgang Pree: „Design Patterns for Object-Oriented Software Development“, ACM Press, Addison-Wesley, 1994

[Pree94a] Wolfgang Pree: „Meta Patterns - A Means for Capturing the Essentials of Reusable Object-Oriented Design“ in [ECOOP94]

[Pree95] Wolfgang Pree: „Reusing Microsoft’s Foundation Class Library - A Programmer’s Perspective“ in [Lewis95]

[Reenskaug95] T. Reenskaug: „Working with Objects“, Manning, Greenwich, 1996

[Riehle93] Dirk Riehle: „Objektorientierte Architektur nach der Werkzeug-Material Metapher am Beispiel eines grafischen Aufgabennetzeditors“, Studienarbeit am Arbeitsbereich Softwaretechnik, Fachbereich Informatik, Universität Hamburg, 1993 (http://swt-www.informatik.uni-hamburg.de/~riehle)

[Riehle93a] Dirk Riehle: „Dokumentation zur FIAK v1.0 Bibliothek“, 27.3.93 und „Interaktionstypen Dokumentation zur IATMotif v1.0 Bibliothek“, 14.3.93 (http://swt-www.informatik.uni-hamburg.de)

[Riehle95] Dirk Riehle: „Patterns for Encapsulating Class Trees“ in [PLoP95]

[Riehle96] Dirk Riehle: „The Event Notification Pattern – Integrating Implicit Invocation with Object-Orientation“, Theory and Practice of Object Systems 2, 1, 1996 (http://swt-www.informatik.uni-hamburg.de/~riehle)

[Riehle96a] Dirk Riehle: „Bureaucracy - A Composite Pattern“ in [PLoP96]

[Riehle96b] Dirk Riehle: „Describing and Composing Patterns Using Role Diagrams“, Proceedings of WOON’96, the 1st International Conference on Object-Orientation in Russia, Edited by A. Smolyani and A. Shestialtynov, St. Petersburg Electrotechnical University (reprinted in Proceedings of the Ubilab Conference’96, Zürich, Edited by Kai-Uwe Mätzel and Hans-Peter Frei, Universitätsverlag Konstanz), 1996

[Ritz94] Michael Ritz: „Eine grafische Benutzeroberfläche zur Versionsverwaltung der Shape Tools als Grundlage einer objektorientierten Software-Entwicklungsumgebung“, Diplomarbeit am Arbeitsbereich Softwaretechnik, Fachbereich Informatik, Universität Hamburg, 1994

[Rosenstein95] Larry Rosenstein: „MacApp: First Commercially Successful Framework“ in [Lewis95]

[RZ95] Dirk Riehle, Heinz Züllighoven: „A Pattern Language for Tool Construction and Integration Based on the Tools & Materials Metaphor“ in [PLoP95]

[RZ96] Dirk Riehle, Heinz Züllighoven: „Understanding and Using Patterns in Software Development“, Theory and Practice of Object Systems 2, 1 (1996)

[RW96] Stefan Roock, Henning Wolf: „Konzeption und Implementierung eines ‘‘Reaktionsmusters’’ für objektorientierte Softwaresysteme“, Studienarbeit am Arbeitsbereich Softwaretechnik, Fachbereich Informatik, Universität Hamburg, 1996 (http://swt-www.informatik.uni-hamburg.de/~1roock)

[RW97] Stefan Roock, Henning Wolf: „Der Werkzeug & Material - Ansatz und Mehrbenutzerumgebungen: Die Raummetapher zur Kooperation und Koordination über persistente Materialien“, Diplomarbeit am Arbeitsbereich Softwaretechnik, Fachbereich Informatik, Universität Hamburg, 1997

[RWW95] Stefan Roock, Ulfert Weiß, Henning Wolf: „Programmierrichtlinien am Arbeitsbereich Softwaretechnik“, HTML-Dokument, entstanden innerhalb eines Projektseminares, WS 1995 (http://swt-www.informatik.uni-hamburg.de)

[Taligent94] Taligent, Inc.: „Taligent’s Guido To Designing Programs - Well-Mannered Object-Oriented Design in C++“, Addison-Wesley, Reading, Massachusetts, 1994

[Traub95] Horst-Peter Traub: „Objektorientierte Behälterklassen-Bibliotheken - Konzepte, Entwurf und Implementation“, Mitteilung FBI-HH-M 252/95, Fachbereich Informatik, Universtät Hamburg, Februar 1996 (http://swt-www.informatik.uni-hamburg.de)

[Vlissides90] John M. Vlissides: „Generalized Graphical Object Editing“, Stanford University, Technical Report: CSL-TR-90-427, June 1990

[Vlissides95] John M. Vlissides: „Unidraw: A Framework for Building Domain-Specific Graphical Editor“ in [Lewis95]

[Vlissides96] John M. Vlissides: „Pattern Hatching - To Kill a Singleton“, C++ - Report, June 1996

[Wegner90] Peter Wegner: „Concepts and Paradigms of Object-Oriented Programming“, ACM OOPS Messenger, Volume 1, No. 1, August 90, Seiten 8 - 87

[Weiß94] Ulfert Weiß: „Spezifikation und Implementation eines Interaktionstypen zur graphischen Darstellung von Dokumenten und Dokumentbeziehungen als Erweiterung einer objektorientierten Bibliothek unter Verwendung des Fenstersystems Motif“, Studienarbeit am Arbeitsbereich Softwaretechnik, Universität Hamburg, 1994 (http://swt-www.informatik.uni-hamburg.de/~1weiss)

[Weiß97] Ulfert Weiß: „Dokumentation und Benutzerhandbuch für Grafik-IAT’s und Manipulatoren“, Dokumentation zur BibV30 im WWW, noch nicht vollständig, wird später integriert in [Lilienthal97] (http://swt-www.informatik.uni-hamburg.de)

[West93] A. West: „Animating C++ Programs. Dynamic C++ Animation“, White Paper, Objective Software Technology Ltd., 1993

[WG94] André Weinand, Erich Gamma: „ET++ - a Portable, Homogeneous Class Library and Application Framework“, Proceedings of UBILAB Conference ’94, Universitätsverlag Konstanz, 1994

[WG95] André Weinand, Erich Gamma: „ET++ - a Portable, Homogeneous Class Library and Application Framework“ in [Lewis95]

[Willamowius97] Jan Willamowius: „Framework-Evolution am Beispiel eines Frameworks für Telefonie-Anwendungen“, Diplomarbeit am Arbeitsbereich Softwaretechnik, Fachbereich Informatik, Universität Hamburg, voraussichtlich 1997

[WJ90] Rebecca J. Wirfs-Brock, Ralph E. Johnson: „Designing Object-Oriented Software“, Englewood Cliffs, NJ, Prentice Hall, 1990

[WW94] Dirk Weske, Martina Wulf: „Konzepte zur Materialversorgung verteilter Werkzeugumgebungen am Beispiel einer objektorientierten Datenbank“, Studienarbeit am Arbeitsbereich Softwaretechnik, Fachbereich Informatik, Universität Hamburg, 1994

[Wulf95] Martina Wulf: „Konzeption und Realisierung einer Umgebung zur Koordination rechnergestützter Tätigkeiten in kooperativen Arbeitsprozessen“, Diplomarbeit am Arbeitsbereich Softwaretechnik, Universität Hamburg, Juli 1995 (http://swt-www.informatik.uni-hamburg.de)