SOFTWARE PATTERNS: PINWÄNDE

DIPLOMARBEIT
ABTEILUNG FÜR ANGEWANDTE INFORMATIK
WIRTSCHAFTSUNIVERSITÄT WIEN

Michael Hahsler

13. Dezember 1997

Inhaltsverzeichnis

Vorwort

Wiederverwendbare Software, Objekt-Orientierung, Design Patterns und Frameworks gehören zu den interessantesten Software Engineering Ansätzen der letzten Zeit. Alle versuchen auf ihre Art die Qualität der entstehenden Software zu steigern und gleichzeitig den damit verbundenen Aufwand zu verringern.

Das Konzept der Objekt-Orientierung ist mit dem Gedanken der Wiederverwendung von Software eng verbunden. Es bietet Techniken wie Kapselung, Vererbung und Polymorphismus, die die Wiederverwendung von Softwaremodulen erleichtern. Doch muß Wiederverwendbarkeit und Qualität durch den Designer in die Software eingebracht werden. Dazu wird ein Designer mit viel Erfahrung benötigt.

Design Patterns verfolgen einen anderen Weg. Sie bieten die Möglichkeit Erfahrungen von Designexperten zu sammeln und zu katalogisieren. Diese gesammelten Erfahrungen können andere Designer in ihre eigenen Softwareprojekte übernehmen oder sie können für Lehrzwecke verwendet werden. Design Patterns zielen nicht primär auf die Wiederverwendung von Programmteilen ab, sondern ermöglichen die Wiederverwendung von qualitativ hochwertigem Design.

Frameworks sind halbfertige Softwarearchitekturen. Sie erhalten ihre Wiederverwendbarkeit aus der Tatsache, daß die Bereiche ihrer Architektur, die bei der Verwendung für ein anderes Problem verändert werden müssen, explizit gekennzeichnet sind.

Diese Arbeit beschäftigt sich mit dem Pattern-Ansatz für die Architektur von Software. Nach einer kurzen Darstellung des Ansatzes werden das Pinwand-Pattern und seine Varianten beschrieben. Pinwände werden verwendet, um Informationen zu sammeln und Interessierten zur Verfügung zu stellen. Sie finden unter anderem in den folgenden Bereichen Anwendung:

- Groupware-Anwendungen
- Conferencing Systeme
- Diskussionsforen
- Virtuelle Bibliotheken

Die Arbeit besteht aus zwei Teilen. Der erste Teil beschäftigt sich mit den *Grundlagen des Pattern-Ansatzes* und seiner Anwendung auf *Pinwände*. Der zweite Teil beinhaltet die Implementation des im ersten Teil entwickelten Patterns *verteilte Bibliothek*.

Der erste Teil beginnt mit dem Kapitel 1 und gibt einen Überblick über den Stand der Entwicklung von Design-Patterns und Frameworks. Besonderer Wert wird auf die Analyse des Nutzens und der Probleme von Patterns gelegt.

In Kapitel 2 werden das Pinwand-Pattern und seine Varianten beschrieben. Das Pinwand-Pattern wird anhand der Kommunikationsanforderungen einer Groupware-Anwendung beschrieben. Seine Varianten sind die strukturierte Pinwand und die verteilte Bibliothek. Die strukturierte Pinwand arbeitet mit strukturierten Informationen und vereinfacht damit die automatische Verarbeitung und Auswertung ihrer Inhalte. Die verteilte Bibliothek erweitert das Pinwand-Pattern um die Abläufe, die notwendig sind, um in einem Netzwerk verteilte Dokumenten aufzufinden.

Der Zweite Teil dieser Arbeit beginnt mit dem Kapitel 3. Hier werden der Anwendungsbereich und die Voraussetzungen der virtuellen Bibliothek beschrieben. Die Kapitel 4 bis 8 enthalten den Aufbau, die Bedienung und die Installation der virtuellen Bibliothek. Außerdem wird der Befehlssatz zur Manipulation der Inhalte der Bibliothek beschrieben. Als letztes Kapitel wird die Möglichkeit zur Erweiterung der Funktionalität der Bibliothek dargestellt.

Der Anhang enthält ein Glossar, die Spezifikation des Befehlssatzes der virtuellen Bibliothek in Backus-Naur Form und den Programmcode.

Nachdem ich mich mit der verfügbaren Literatur über Patterns beschäftigt hatte, war für mich des Prinzip hinter dem Pattern-Ansatz noch sehr verschwommen. Erst durch die Entwicklung eines eigenen Patterns wurden mir die Vorteile und Probleme des Ansatzes klar. Patterns sind aus der Praxis entnommen und können daher am leichtesten im praktischen Zusammenhang erfaßt werden. Da Patterns von Experten mit langjähriger Erfahrung erstellt werden, ist das von mir beschriebene Pinwand-Pattern natürlich nicht perfekt, doch hat es den Zweck erfüllt, mir den Zugang zu Patterns zu ermöglichen.

Teil I

Grundlagen

Kapitel 1

Der Pattern-Ansatz

1.1 Einführung

Am Anfang jeder Entwicklung steht ein Problem, das es zu lösen gilt. Personen die viele, meist gleichartige Probleme gelöst haben, nennt man Experten. Die Frage ist nun, was unterscheidet Experten von normalen Menschen. Um eine Antwort zu finden, muß das Problemlösungsverhalten von Experten betrachtet werden. In den meisten Fällen erfindet der Experte keine neuen Lösungen, sondern er wendet ihm bekannte Lösungswege an. Mit der Zeit sammelt der Experte durch die praktische Arbeit Erfahrung, die er zu einer Wissensbasis verknüpft. Genau dieses Wissen über Problemlösungen und ihre Anwendung macht eine Person zum Experten.

Ein Pattern ist die Abstraktion von Expertenwissen über die Lösung eines gewissen Problems, das immer wieder in leichten Abwandlungen vorkommt. Patterns konzentrieren sich auf die Gemeinsamkeiten einer Gruppe von Problemen und den Kern der Lösung. Durch diese Abstraktion können gut durchdachte Patterns wieder und wieder angewendet werden.

Patterns werden in vielen Bereichen - auch unbewußt - angewandt, da das Denken in einem Problem-Lösungs Schema weit verbreitet ist. Beispiele für Bereiche, in denen Patterns bewußt verwendet werden, sind die Architektur [Alexander *et al.*, 1977], Jazzmusik [Coker *et al.*, 1970], das Management von Organisationen [Coplien, 1995] und auch die Entwicklung von wiederverwendbarer Software [Foote and Opdyke, 1995].

Dieses Kapitel beschäftigt sich mit der Verwendung von Patterns als Methode zur Softwareentwicklung. Im Abschnitt 1.2 geht es um die Erklärung des Begriffs *Design Pattern* und die Analyse ihres Nutzens. Im Abschnitt 1.3 wird die Verwendung von Patterns beim Design von Software Architekturen beleuchtet.

1.2 Design Patterns

1.2.1 Einführung

Der Begriff *Pattern* kommt aus der Architektur. Was darunter zu verstehen ist, wird kurz zusammengefaßt. Darauf aufbauend werden *Design Patterns* dargestellt. Schwerpunkte sind ihr Aufbau, die Beziehung zwischen verschiedenen Design Patterns und ihr Nutzen.

1.2.2 Der Begriff Pattern

Laut Christopher Alexander [Alexander, 1979], Professor für Architektur an der Universität Berkley, ist jedes Pattern eine dreiteilige Regel, welche die Beziehung zwischen einem gewissen Kontext, einem Problem und seiner Lösung ausdrückt. Jedes Pattern beschreibt ein Problem, das immer wieder vorkommt und dadurch wieder und wieder angewendet werden kann. Dabei bezeichnet man mit dem Ausdruck Pattern sowohl das Ergebnis, das durch die Anwendung der Regel entsteht, als auch die Regel selbst.

Beispiel für ein Architektur-Pattern ist das *Ring Roads*-Pattern [Alexander *et al.*, 1977]. Das Problem und der Kontext werden wie folgt dargestellt:

> It is not possible to avoid the need for high speed roads in modern society; but it is essential to place them and build them in such a way that they do not destroy communities or countryside.

Die Lösung wird durch die Skizze in Abb. 1.1 dargestellt und folgendermaßen beschrieben:

> Place high speed roads (freeways or other major arteries) so that:
>
> 1. At least one high speed road lies tangent to each local transport area.
> 2. Each local transport area has at least one side not bound by a high speed road, but directly open to the countryside.
> 3. The road is always sunken, or shielded along its length by berms, or earth, or industrial buildings, to protect the nearby neighborhoods from noise.

Natürlich bezog sich Alexander auf die Baukunst, doch ist der Pattern-Ansatz fast unverändert für die Softwareentwicklung anwendbar.

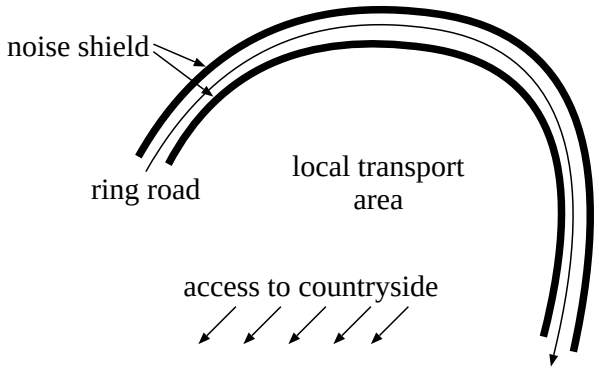

Abbildung 1.1: Pattern: Ring road

1.2.3 Das Design Pattern

Design Patterns sind Beschreibungen von Lösungen für Software-Design Probleme, die mehreren Kriterien genügen müssen. Sie müssen mindestens den Kontext, das Problem und die Lösung darstellen. Außerdem müssen die Beschreibungen in einer festgelegten Form erfolgen, damit man die Patterns miteinander vergleichen und in ein Schema einordnen kann. Für diesen Zweck haben in den letzten Jahren viele Autoren Formulare entwickelt, die auf die Anforderungen von Design Patterns für die Softwareentwicklung abgestimmt sind. Die Struktur des bekanntesten Formulars wurde von Erich Gamma und seinen Kollegen [Gamma *et al.*, 1993] entwickelt. Da dieses aber die Objekt-Orientierung überbetont, wird hier das Formular aus [Buschmann and Meunier, 1995] verwendet. Es hat folgendes Aussehen:

1. **Name (Name).** Der Name und eine kurze Beschreibung des Patterns. Der Name soll das Problem und seine Lösung bezeichnen. Er muß sehr sorgfältig gewählt werden, da er direkt in das Design-Vokabular aufgenommen wird. Das heißt, der Name wird später stellvertretend für alle wichtigen Aspekte des gesamten Pattern verwendet. Dies erleichtert die Kommunikation zwischen Designern. Schlägt einer die Verwendung eines bestimmten Patterns vor, weiß der andere sofort, was gemeint ist. Die Beschreibung soll am besten in einem Satz darstellen, welche Probleme mit diesem Pattern angesprochen werden.

2. **Auch bekannt als (Also Known As).** Ein alternativer Name, unter dem dieses Design Pattern bekannt ist.

3. **Beispiel (Example).** Es wird ein Beispiel beschrieben, das die Existenz des Design Problems zeigt. Das Beispiel hilft, die abstrakte Beschreibung des Patterns leichter zu verstehen.

4. **Kontext (Context).** Beschreibung der Situation, in der das Pattern angewandt werden kann.

5. **Problem (Problem).** Das Problem, das vom Pattern behandelt wird. Es wird zuerst der Kern des Problems allgemein beschrieben. Danach werden Erfordernisse, Einschränkungen und wünschenswerte Eigenschaften des Ergebnisses aufgezählt und beschrieben.

6. **Lösung (Solution).** Das grundsätzliche Prinzip, das der Lösung zugrunde liegt.

7. **Struktur (Structure).** Hier werden die Konfiguration der beteiligten Komponenten und ihre Beziehungen untereinander beschrieben. Es wird in der Regel eine grafische Repräsentation zur Beschreibung verwendet. Als Notation für objekt-orientiertes Design bietet sich die Object Modeling Technique [Rumbaugh *et al.*, 1991] an.

8. **Laufzeit Verhalten (Dynamics).** Beschreibt, wie die Komponenten zusammenarbeiten. Es handelt sich dabei um die dynamischen Aspekte des Systems, die das Laufzeit-Verhalten bestimmen.

9. **Implementation (Implementation).** Hinweise auf Probleme und Besonderheiten bei der Implementation des Patterns. Es handelt sich um Vorschläge und keine Regeln, da die Implementation an die speziellen Bedürfnisse jedes Einzelfalls angepaßt werden muß. Es können auch Programmteile als Beispiele zur Illustration einer möglichen Implementation angegeben werden.

10. **Gelöste Beispiele (Examples Resolved).** Eine Beschreibung der wichtigen Aspekte für die Lösung des Beispiels, die bis jetzt noch nicht erklärt wurden.

11. **Varianten (Variants).** Eine kurze Beschreibung von Varianten oder Spezialisierungen des Patterns.

12. **Bekannte Anwendungen (Known Uses).** Beispiele von existierenden Systemen, in denen das Pattern verwendet wird.

13. **Konsequenzen (Consequences).** Beschreibt die Ergebnisse und Einschränkungen, die sich aus der Anwendung des Patterns ergeben. Es ist unwahrscheinlich, daß ein Pattern alle Probleme vollständig lösen kann. Darum ist es wichtig hier auf vom Pattern ungelöste Probleme einzugehen. Die Beschreibung der Konsequenzen ist das Kriterium für die Auswahl bei alternativen Patterns.

14. **Siehe auch (See Also).** Patterns, die ähnliche Probleme lösen oder die helfen das eben beschriebene Pattern zu verfeinern.

Andere Autoren fassen Punkte aus diesem Formular zusammen oder verwenden zusätzliche, um bestimmte Aspekte zu betonen. Im großen und ganzen sind sie diesem Formular sehr ähnlich.

1.2.4 Kategorisierung von Design Patterns

Da man nicht alle Patterns kennen kann, benötigt man eine Übersicht über diese. Es muß eine Ordnung in die Patterns gebracht werden, die es erlaubt die geeigneten Patterns für ein bestimmtes Problem zu finden. Mehrere Autoren haben Patterns nach gewissen Kriterien geordnet.

Erich Gamma und seine Mitautoren [Gamma *et al.*, 1993, Gamma *et al.*, 1995] teilen Design Patterns nach zwei Klassifikationskriterien ein. Dies sind der Umfang der Patterns und ihr Zweck. Nach dem Umfang werden Patterns für *Klassen* und für *Objekte* unterschieden, da ausschließlich objekt-orientierte Patterns beschrieben werden. Nach dem Zweck werden Patterns in die Kategorien *Creational*, *Structural* und *Behavioral* eingeteilt. *Creational Patterns* beschäftigen sich mit der Erzeugung von Objekten. *Structural Patterns* beschreiben die statische Zusammensetzung von Objekten und Klassen. *Behavioral Patterns* charakterisieren das dynamische Verhalten von Objekten und Klassen. Erich Gamma hat über zwanzig Patterns beschrieben und in sein Schema eingefügt. Beispiele sind die Patterns *Abstract Factory*, *Adapter* und *Strategy*.

Frank Buschman und seine Kollegen [Buschmann and Meunier, 1995, Buschmann *et al.*, 1996] teilen Patterns nach ihrem Abstraktionsgrad ein. Er beginnt mit der höchsten Abstraktionsstufe und nennt diese *Architectural Patterns*. Diese erklären die fundamentale strukturelle Organisation von Softwaresystemen. Sie geben die nötigen Subsysteme an und spezifizieren ihre Funktionen und Beziehungen zueinander. In der mittleren Abstraktionsebene werden *Design Patterns* verwendet. Diese zeigen wie Subsysteme oder Komponenten verfeinert werden. Sie lösen immer wiederkehrende Design Probleme. Die geringste Abstraktion weisen *Idioms* auf. Sie sind speziell für Probleme, die im Zusammenhang mit einer bestimmten Programmiersprache auftreten, bestimmt. Wichtige Idioms für eine Sprache können für eine andere nutzlos sein. Als zweites Klassifikationskriterium wird der zu lösende Problembereich verwendet. Einige solcher Problembereiche sind zum Beispiel *Communication*, *Access Control* und *Distributed Systems*. Die Liste der Problembereiche ist erweiterbar und kann bei Bedarf ergänzt werden. In dieses Schema wurden auch rund zwanzig Patterns eingeordnet. Die Patterns decken sich großteils mit den schon von Gamma präsentierten.

Walter Zimmer [Zimmer, 1995] geht von den Patterns in [Gamma *et al.*, 1995] aus. Da Patterns nur selten alleine existieren, versucht Zimmer die Beziehungen zwischen den einzelnen Patterns aufzuzeigen. Zur Beschreibung der Beziehungen verwendet er drei Kategorien. Diese sind *X verwendet Y in seiner Lösung*, *eine Variante von X verwendet Y in seiner Lösung* und *X ist ähnlich wie Y*. X und Y bezeichnen zwei betrachtete Patterns. Die Beziehungen werden als Graphen zwischen den Patterns dargestellt. Danach führt Zimmer drei Schichten ein, denen er die Patterns zuordnet. Die unterste Schicht nennt er *Basic Design Patterns and Techniques*. Sie beinhaltet Patterns, die für sehr generelle Probleme angewandt werden. Sie werden regelmäßig durch Patterns der höheren Schichten verwendet. *Design Patterns for Typical Software Problems* heißt die

mittlere Schicht. Patterns dieser Schicht werden für speziellere Probleme verwendet. Die oberste Schicht nennt Zimmer *Design Patterns Specific to an Application Domain*. Sie beinhaltet Patterns, die für einen oder mehrere Bereiche des Software Designs verwendet werden können. Durch die hierarchische Einordnung der Patterns und der grafischen Darstellung der Beziehungen zwischen ihnen wird die Orientierung in einem System von Patterns sehr erleichtert. Die oft schwer erkennbaren Zusammenhänge und Abhängigkeiten werden explizit dargestellt.

Von einer *Pattern Language*, wie sie Alexander [Alexander *et al.*, 1977] für die Architektur vorstellt, ist man bei Design Patterns für Software Probleme noch weit entfernt. Eine solche Sprache müßte Lösungen für alle Design Probleme eines bestimmten Bereiches bieten, um komplett zu sein. Dies wurde bisher nur für sehr beschränkte Teilbereiche erreicht. Ein Beispiel für den Bereich Informationsintegrität ist die Pattern Language CHECKS [Cunningham, 1995]. Die Frage, die sich in diesem Zusammenhang stellt ist, ob eine vollständige Pattern Language für Softwaredesign überhaupt sinnvoll ist. Möglicherweise ist es ausreichend, die bisher vorgestellten Pattern-Systeme weiter auszubauen und zu verfeinern.

1.2.5 Nutzen von Design Patterns

Patterns werden von erfahrenen Software Designern schon sehr lange verwendet. Das Problem bis vor kurzem war, daß jeder Designer „seine“ Patterns selbst entwickeln mußte, da keine geeignete Möglichkeit bekannt war, das Wissen in geordneter Form weiterzugeben. Patterns bieten diese Möglichkeit. Ihre Verwendung zur Beschreibung von Problem-Lösungs Beziehungen bringen laut [Gamma *et al.*, 1995] folgende vier große Vorteile:

Ein gemeinsames Design Vokabular. Patterns bieten durch ihren Namen eine einfache Möglichkeit zu Kommunikation und Dokumentation. Kennen mehrere Designer bestimmte Patterns, reicht ihr Name aus, um über Designalternativen zu sprechen. Die Namen der Patterns bilden ihr gemeinsames Vokabular.

Eine Dokumentations- und Lernhilfe. Bei großen objekt-orientierten Systemen werden oft unbewußt Design Patterns verwendet. Beschreibt man so ein System mit Design Patterns, die im System gefunden werden, kann seine Funktion viel schneller und leichter verstanden werden.

Da Design Patterns für allgemeine Probleme, die immer wieder vorkommen, Lösungen bieten, liegt es nahe sie für Lehrzwecke zu verwenden. In [Goldfedder and Rising, 1996] werden Erfahrungen über den Unterricht mit Patterns beschrieben.

Eine Erweiterung zu existierenden Methoden. Existierende Design Methoden beschreiben Probleme, die während des Designs auftreten und bieten passende

Lösungen an. Sie vernachlässigen jedoch den Bereich der Erfahrung. Design Patterns können folgende Vorteile in die Design Phase einbringen:

- Design Patterns beinhalten Erfahrung von Design Experten.
- Design Patterns beschreiben, warum eine bestimmte Design Alternative (ein Pattern) verwendet wird, und die sich daraus ergebenden Konsequenzen. Daher können Design Entscheidungen später leichter nachvollzogen werden.
- Design Patterns bringen flexibles und wiederverwendbares Design aus der Design Phase in die Analyse Phase. Dadurch wird der Übergang vom Analysemodell zum Designmodell einfacher, und das Analysemodell muß später nicht nachträglich verändert werden, um flexibles Design zu ermöglichen.

Ein Ziel für Refactoring. Software, die lange verwendet werden soll, steht irgendwann vor dem Problem, daß sich mit der Zeit die Anforderungen, die sie zu erfüllen hat, ändern. Oft werden zusätzliche Anforderungen durch Erweiterungen der Software befriedigt. Dies führt aber zu einem inflexiblen System, das für spätere Anpassungen ungeeignet ist. Um die Software weiter zu entwickeln, muß sie umorganisiert werden. Dieser Prozeß heißt *Refactoring*.

Design Patterns beschreiben Strukturen, die das Resultat von Refactoring sind. Dadurch zeigen sie die Richtung, in die ein inflexibles Software System umorganisiert werden soll, um auch zukünftigen Anforderungen gewachsen zu sein. Geht man noch einen Schritt weiter, und verwendet Patterns im Design von neuer Software, wird der Bedarf für Refactoring zu einem späteren Zeitpunkt verringert oder sogar ganz vermieden.

Die Aufzählung der Vorteile wurde von einigen Autoren verfeinert, aber der Inhalt blieb weitgehend gleich. Marschall Cline [Cline, 1996] beschreibt neben den Vorteilen von Design Patterns auch folgende Problembereiche:

Überbewertung des Ansatzes. Der Anwendungsbereich, in dem Design Patterns effizient angewandt werden können, ist schwer zu erkennen. Durch die Verwendung von Design Patterns entstehen zusätzliche Kosten, die auch gerechtfertigt sein müssen. Design Patterns ersparen nur dann Kosten, wenn die Software später verändert oder angepaßt werden muß. Ist dies nicht der Fall, bringt die Verwendung von Patterns nicht alle Vorteile.

Einige Design Patterns sind unnötig schwer zu erlernen. Da viele Designer Patterns entwickeln oder Patterns von anderen verändern, geht oft die Klarheit des Designs verloren.

Unbrauchbare Kategorisierungen von Design Patterns. Design Patterns werden, wie schon beschrieben, nach verschiedenen Kriterien eingeteilt. Diese Kriterien sind aber großteils für den durchschnittlichen Designer nicht verständlich, da sie zu stark auf das gemeinsame Design-Vokabular der Patterns abgestimmt sind.

Die beschriebenen Probleme sind großteils darauf zurückzuführen, daß der Patterns Ansatz noch nicht ausgereift ist. Es ist anzunehmen, daß sie im weiteren Entwicklungsprozeß des Pattern-Ansatzes abnehmen werden.

1.3 Design Patterns und Software Architektur

1.3.1 Einführung

Design Patterns sind eine wichtige Hilfe um qualitativ hochwertige Software Architektur zu entwickeln. Mit Software Architektur wird die Beschreibung der Subsysteme und deren Beziehungen untereinander bezeichnet [Buschmann *et al.*, 1996]. Patterns ersetzen dabei nicht andere Design Methoden [Booch, 1994, Rumbaugh *et al.*, 1991], sondern erweitern sie um die Erfahrung der Experten, die sie erstellt haben.

In diesem Abschnitt wird zuerst auf die Auswirkung von Patterns auf die Eigenschaften der Software Architektur eingegangen. Danach werden Frameworks betrachtet, die halbfertige Software-Systeme darstellen. Zum Schluß wird ein Ansatz zur Unterstützung der Weiterentwicklung von Software vorgestellt.

1.3.2 Eigenschaften von Software Architektur

Die Eigenschaften von Software werden in funktionale- und nicht-funktionale Eigenschaften unterteilt. Funktionale Eigenschaften sind direkt für den Anwender sichtbar. Sie stellen implementierte Funktionen wie zum Beispiel Algorithmen dar. Da Software, die nicht die funktionalen Erfordernisse, erfüllt unbrauchbar ist, wird hier auf diese nicht näher eingegangen.

Die nicht-funktionalen Eigenschaften beeinflussen, wie der Name schon sagt, nicht die Funktionalität des Systems, haben aber trotzdem großen Einfluß auf seine Erstellung und Wartung. Solche Eigenschaften sind:

Changeability: Große Softwaresysteme haben oft eine relativ lange Lebensdauer. Da sich die Anforderungen in diesem langen Zeitraum verändern, ist es notwendig, das System anzupassen. Die Kosten für diese Anpassung hängen hauptsächlich davon ab, ob das System für Veränderungen konzipiert wurde. Die Faktoren,

die ein System anpaßbar machen sind *Extensibility*, *Flexibility*, *Performance tunability* und *Fixability* [Fayad and Cline, 1996]. Extensibility bedeutet, daß das System einfach mit neuen Funktionen erweitert werden kann. Flexibility fordert, daß das System für neue Anwendungsbereiche verwendet werden kann. Performance tunablity beschäftigt sich mit der Skalierbarkeit des Systems, wobei Wert auf die verteilte Ausführung von Teilfunktionen gelegt wird. Und Fixability bezeichnet die Fähigkeit, einen Fehler im System zu beheben, ohne dadurch neue Fehler zu erzeugen.

Interoperability: Software Systeme arbeiten nicht total getrennt von der Umwelt. Es müssen genau definierte Schnittstellen nach außen geschaffen werden, über die die Kommunikation abläuft.

Reliability: Verläßlichkeit stellt die Fähigkeit von Software dar, mit dem Auftreten von Fehlern und mit falscher Verwendung durch den Benutzer umzugehen. Der erste Aspekt wird mit Fehlertoleranz und der zweite mit Robustheit bezeichnet. Bei beiden muß im Fehlerfall mit einer geeigneten Maßnahme durch das Programm reagiert werden, um die Integrität des Systems als ganzes nicht zu gefährden.

Testability: Das Testen von großen und komplexen Software Systemen bringt beträchtliche Kosten mit sich. Bei gut testbaren Systemen können kleine Teile entkoppelt vom Rest auf ihre Funktion geprüft werden.

Reusability: Wiederverwendbarkeit kann in zwei Bereiche geteilt werden. Einerseits werden bei der Softwareentwicklung vorhandene Komponenten wiederverwendet. Andererseits findet eine Entwicklung von Software statt, die auf Wiederverwendung ausgerichtet ist. Im letzteren Bereich ist die Wiederverwendbarkeit höher und der Aufwand für Modifikationen geringer.

Klassische Design Methoden konzentrieren sich hauptsächlich auf den funktionalen Bereich von Software. Für nicht-funktionale Eigenschaften kann nur der Designer selbst durch seine Erfahrung sorgen. Genau hier setzen Patterns an. Einerseits bieten sie funktionale Lösungen für wiederkehrende Design Probleme und andererseits bringen sie nicht-funktionale Eigenschaften in das Design ein. Die nicht-funktionalen Eigenschaften ergeben sich daraus, daß Patterns nicht künstlich erzeugt werden, sondern aus einem Prozeß der Wiederverwendung von guten Lösungen entstehen. Dieser Wiederverwendungsprozeß führt automatisch zu den vorher beschriebenen Eigenschaften wie Wiederverwendbarkeit, Testbarkeit, Änderbarkeit und Verläßlichkeit.

1.3.3 Application Frameworks

Ein Framework ist ein wiederverwendbares, teilweise fertiggestelltes Softwaresystem. Es besteht aus fertigen und halbfertigen Teilsystemen, wobei die Architektur des Soft-

waresystems durch diese Teilsysteme vordefiniert ist. Wiederverwendbar ist das System durch die halbfertigen Teilsysteme. Wolfgang Pree [Pree, 1995] nennt diese flexiblen Bereiche eines Frameworks *hot spots*. Ihre Festlegung ist der schwierigste Teil beim Design eines neuen Frameworks. Einerseits muß man den Anwendungsbereich des Frameworks genau kennen, um festzulegen welche Bereiche flexibel gestaltet werden sollen, und andererseits muß man zwischen der Flexibilität und dem späteren Anpassungsaufwand einen Kompromiß finden.

Frameworks werden von Experten hergestellt. Angewendet werden sie hingegen von Nicht-Experten. Aus diesem Grund ist die Beschreibung eines Frameworks mindestens genauso wichtig, wie die technischen Eigenschaften selbst. Eine Möglichkeit, die Anwendung von Frameworks zu beschreiben, sind *Cookbooks*. Sie beinhalten verschiedene Rezepte und beschreiben die Verwendung des Frameworks auf informale Weise. Die Beschreibung bezieht sich auf die Anwendung und nicht auf das Design und die Implementation des Frameworks, da man davon ausgeht, daß sich ein Nicht-Experte nur für die Lösung seines Problems interessiert und nicht für das Design, das dahinter steht.

Eine Schwäche von Cookbooks ist, daß sie nur Beschreiben wie ein Framework auf typische Weise angewendet wird. Dieser Aspekt ist zwar wichtig, jedoch nicht ausreichend um ein Framework vollständig zu beschreiben. Ein gutes Framework kann auf Arten angewendet werden, an die der Designer bei der Erstellung nicht gedacht hat. Um solche Anwendungen zu ermöglichen, muß das Design des Frameworks beschrieben werden. Außerdem fehlt die Beschreibung des Zwecks. Sie ist wichtig, damit sich der Anwender schnell ein Bild machen kann, ob dieses Framework für sein Problem überhaupt geeignet ist. Beschreibungen von Frameworks sollten daher möglichst vollständig und dabei möglichst kurz sein, um den Leseaufwand von potentiellen Anwendern gering zu halten. Dieses Ziel kann durch die interne Struktur eines *Satzes von Patterns* erreicht werden [Johnson, 1992]. Für einfache Probleme, die regelmäßig auftauchen, müssen nur die ersten paar Patterns gelesen werden. Speziellere Probleme werden dann durch weiterführenden Patterns beschrieben, die erst bei Bedarf gelesen werden müssen.

Pree beschäftigt sich mit der Erstellung von Frameworks. Er beschreibt *Metapatterns*, die zeigen wie man Frameworks konstruiert, die unabhängig von einem speziellen Anwendungsbereich angewendet werden können. Er sieht diese Metapatterns als eine Ansammlung von Erfahrungen aus bereits bestehenden Frameworks, die für die Entwicklung neuer Frameworks genutzt werden sollen. Pree schlägt vor, seine Metapatterns zu einem *hot-spot-driven Approach* weiterzuentwickeln.

1.3.4 Software Architektur und Refactoring

Die Entwicklung von objekt-orientierter Software kann in die drei Phasen *Prototyping*, *Expansion* und *Consolidation* unterteilt werden. Mit Prototyping wird die Erstellung

der Software bezeichnet, die mit der Inbetriebnahme beendet ist. Es folgt die Expansion. Mit der Zeit entstehen neue Anforderungen, die durch Erweiterungen der Software gelöst werden. Durch diese Erweiterungen wird die Software aber inflexibel und immer fehleranfälliger. Soll sie trotzdem noch weiterentwickelt werden, muß man sie grundlegend reorganisieren. Dieser Prozeß heißt *Refactoring* und stellt die Konsolidation dar. Foote und Opdyke [Foote and Opdyke, 1995] haben für die drei Phasen Patterns entwickelt, die die grundlegenden Probleme und ihre Lösung beschreiben. Diese Patterns zielen nicht auf die Behebung einzelner Probleme ab, sondern auf die Verbesserung der Software Architektur, um eine kontinuierliche Weiterentwicklung der Software zu ermöglichen.

1.4 Zusammenfassung

Im Zentrum des Pattern-Ansatzes steht die Erfahrung von Experten. Kein anderer Ansatz bietet die Möglichkeit, die Erfahrung anderer in diesem Umfang für sich selbst zu verwenden. Die Patterns aus [Gamma *et al.*, 1995] beschäftigen sich weitgehend mit Design Problemen. Sie stellen kein eigenständiges Verfahren dar, weil sie noch viele Lücken aufweisen. Daher kann man sie als Ergänzung zu anderen Verfahren, wie zum Beispiel die *Object Modeling Technique* [Rumbaugh *et al.*, 1991], verwenden. Ein vollständiges Verfahren muß daher neben Design Patterns noch Patterns für die Analyse, das User Interface und vieles mehr umfassen. Einen Schritt in diese Richtung stellt Norman Kerth [Kerth, 1995] vor. Er präsentiert Patterns für den Übergang von der Analyse zum Design.

Kapitel 2

Pinwand Patterns

2.1 Einführung

Patterns beschreiben Probleme, die immer wieder vorkommen und den Kern ihrer Lösung. Im Rahmen dieser Arbeit werden Patterns für die Sammlung und die Vereinfachung des Zugangs zu Informationen beschrieben. Diese Patterns sind die *Pinwand* und ihre Varianten.

2.2 Pattern: Pinwand

Name: Das *Pinwand*-Pattern wird verwendet, um Informationen von Benutzern zu sammeln, zu speichern und allen anderen Benutzern zugänglich zu machen.

Auch bekannt als: Bulletin Board.

Beispiel: Betrachten wir eine *Groupware*-Anwendung im Intranet. Einer der wichtigsten Teile jeder Groupware-Anwendung ist die Kommunikation zwischen den Mitgliedern eines Teams. Wichtig ist dabei, daß diese Kommunikation Raum und Zeit überbrückt und daß der Informationsfluß von einem Mitglied zu mehreren oder allen anderen Mitgliedern des Teams verläuft. In unserem Fall erfolgt diese Art der Kommunikation über eine Pinwand. Jedes Teammitglied kann Nachrichten an diese Pinwand heften, die später von allen anderen gelesen werden können.

Der Aufbau der Pinwand wird in Abb. 2.1 dargestellt. Auf dieser Pinwand befinden sich einige Informationen. Das Teammitglied C heftet gerade eine neue Information an die Pinwand. Die Mitglieder A, B und D lesen verschiedene Informationen. E benötigt keine der angebotenen Informationen.

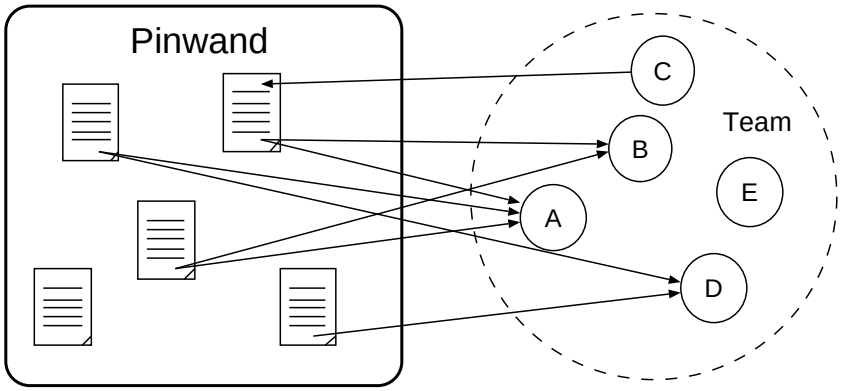

Abbildung 2.1: Pinwand für eine Groupware Anwendung

Kontext: Es besteht Informationsbedarf, wobei es wichtig ist, daß jedem die Informationen, die er benötigt, zur Verfügung stehen.

Problem: Die Verteilung von Informationen über, zum Beispiel, ein Intranet erfolgt meistens mittels Emails. Doch löst die Verwendung von Emails die Kommunikationsanforderungen bei Teamarbeit nur unvollständig. Folgende Probleme bleiben ungelöst:

- Unbekannte Interessenten. Oft kann nicht festgestellt werden, wer bestimmte Informationen benötigt. Daraus ergibt sich das Problem, daß einerseits Personen Informationen bekommen, die sie nicht benötigen, und sie andererseits benötigte Informationen nicht erhalten.
- Unbekannter Zeitpunkt. Wird eine Email verschickt, liest sie der Empfänger in der Regel kurz nachdem er sie erhalten hat und löscht sie danach. Werden die Informationen später benötigt, ist die Information beim Empfänger nicht mehr vorhanden und muß vom Absender nochmals angefordert werden.

Als Abhilfe für den unbekannten Interessentenkreis werden oft *Internet Mailing Lists* [Kantor and Lapsley, 1986] verwendet. Wobei jeder Abonnent der Liste alle Informationen, die an diese Liste geschickt werden erhält. Dies stellt zwar sicher, daß man alle Informationen erhält, führt aber zu folgendem Problem:

- Informationsüberlastung. Es darf nicht jede verfügbare Information an alle verschickt werden, da die Empfänger dann nur mehr damit beschäftigt sind, die für sie wichtigen Informationen aus dem Überangebot herauszufiltern.

Lösung: Zwischen Informationsanbieter und Informationssuchende wird eine Pinwand gestellt. Ein Informationsanbieter braucht sich nicht mehr darum zu kümmern, wer welche Informationen benötigt, da er sie durch die Pinwand für

alle zugänglich macht. Die Information wird von einer Bring- zur Holschuld. Die Interessierten holen sich benötigte Informationen genau zu dem Zeitpunkt, wann sie sie benötigen von der Pinwand. Dadurch sind die Probleme des unbekannten Zeitpunktes und der Informationsüberlastung behoben.

Struktur: Die Pinwand-Architektur besteht aus drei Komponenten. Diese sind die *Datenbasis*, das *Interface* und die *Steuereinheit*.

Die Datenbasis hat die Aufgabe, neue Informationen aufzunehmen, sie zu speichern und Abfragen zu beantworten. Es kann sich um eine Datenbank oder jedes andere Speicherungsverfahren handeln.

Das Interface dient als Verbindung der Pinwand mit den Anwendern und Informationsanbietern. Es setzt Informationen aus der Datenbasis in eine leicht lesbare Form um und überträgt sie zum Benutzer.

Die Steuereinheit stellt die Funktionen der Pinwand zur Verfügung. Diese sind die Abfrage der Datenbasis und das Hinzufügen einer neuen Information. Außerdem muß dafür gesorgt werden, daß alte Informationen gelöscht werden und daß ein Administrator die Pinwand warten kann.

Das Datenflußdiagramm in Abb. 2.2 zeigt die grundlegende Struktur der Pinwand.

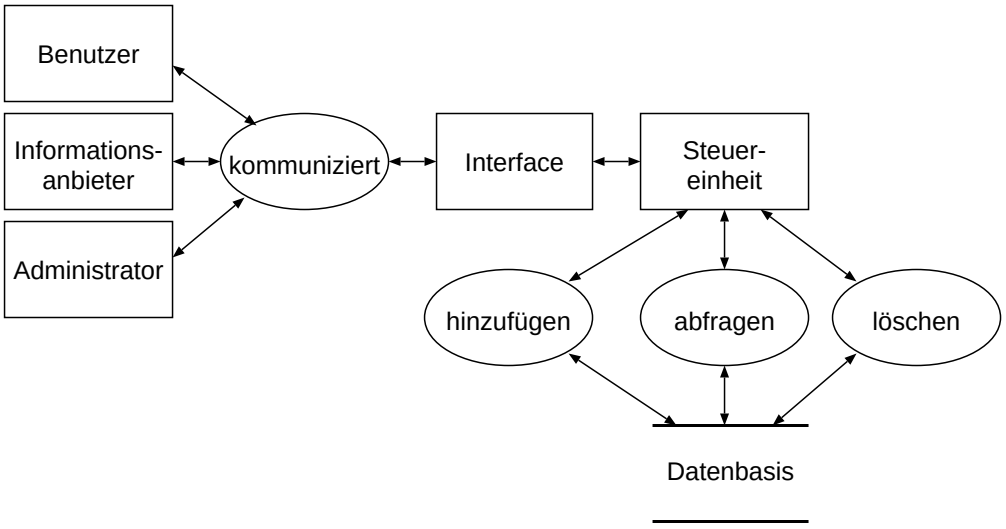

Abbildung 2.2: Datenflußdiagramm der Pinwand

Laufzeit Verhalten: Wie in Abb. 2.2 dargestellt, arbeiten drei Gruppen von Anwendern mit der Pinwand. Diese sind die *Benutzer*, die *Informationsanbieter* und der *Administrator*. Die Benutzer verwenden die Informationen, die auf der Pinwand angeboten werden. Die Informationsanbieter sind die Autoren dieser Informationen und stellen sie durch die Pinwand den Benutzern zur Verfügung.

Der Administrator betreut die Pinwand und löscht alte Informationen aus der Datenbasis.

Daraus ergeben sich folgende drei Abläufe:

1. Die Pinwand wird von einem Benutzer betrachtet.
2. Ein neuer Beitrag wird von einem Informationsanbieter auf die Pinwand geheftet.
3. Ein Beitrag wird vom Administrator gelöscht.

Es muß besonders darauf geachtet werden, daß diese Abläufe parallel stattfinden können.

Implementation: Bei der Implementation der Pinwand sollen folgende Punkte beachtet werden:

1. Wahl der Datenbasis. Es kann von einer relationalen Datenbank bis zum Speichern der Einträge, als Attribut-Wert Listen in Dateien, jedes Verfahren verwendet werden. Ausschlaggebend sind nur die Performance, die Kosten und der Implementationsaufwand.
2. Erstellung des Interfaces. Das Interface hängt von dem verwendeten Transport-Technologie ab. Eine Designalternative ist die Verwendung der Client-Server Architektur des Internets mit Hypertext. Das Interface besteht dann aus einer Mischung von statischen HTML-Seiten, Formularen und dynamisch generierten Seiten.

   Für die Benutzer wird die Pinwand durch ihre Hauptseite dargestellt. Der Aufbau dieser Hauptseite hängt von ihren Inhalten ab. Werden nur eine kleine Anzahl von kurzen Nachrichten auf der Pinwand gespeichert, können auf der Hauptseite alle Nachrichten im Volltext dargestellt werden. Gibt es sehr viele oder umfangreiche Informationen, können sie nur in einer komprimierten Form auf der Hauptseite dargestellt werden. Der Volltext ist dann erst in einem zweiten Schritt von der Hauptseite erreichbar.

   Um durch die Informationsanbieter neue Informationen zur Pinwand hinzuzufügen, wird ein Formular verwendet.

   Auch das Löschen von Informationen durch den Administrator kann durch ein Formular unterstützt werden. Dabei muß jedoch darauf geachtet werden, daß nur der Administrator diese Möglichkeit hat. Dies kann durch ein Paßwort oder andere Autentifizierungsmethoden erreicht werden.
3. Aufbau der Steuereinheit. Die Steuereinheit soll grundlegende Funktionen wie die Aufnahme neuer Beiträge, die Erzeugung der Hauptseite der Pinwand und das Löschen von Einträgen bieten. Wichtig ist, daß der Funktionsumfang der Steuereinheit variabel gestaltet werden soll, um zum Beispiel später der Pinwand eine Suchfunktion zu geben. Dies erhöht die Wiederverwendbarkeit der Applikation für andere Zwecke.

Ein Designalternative ist der Aufbau der Steuereinheit als Befehlsinterpreter, der über das *Common Gateway Interface (CGI)* vom WWW-Server seine Befehle erhält. Diese Befehle sind in Formularen auf den HTML-Seiten der Pinwand enthalten. Wird so ein Formular von einem Browser abgeschickt, startet der WWW-Server den Interpreter und übergibt ihm den Befehl. Der Interpreter arbeitet den Befehl ab und gibt das Ergebnis an den WWW-Server zurück. Dieser leitet es zum Browser weiter. Dieser Ablauf wird in Abb. 2.3 dargestellt.

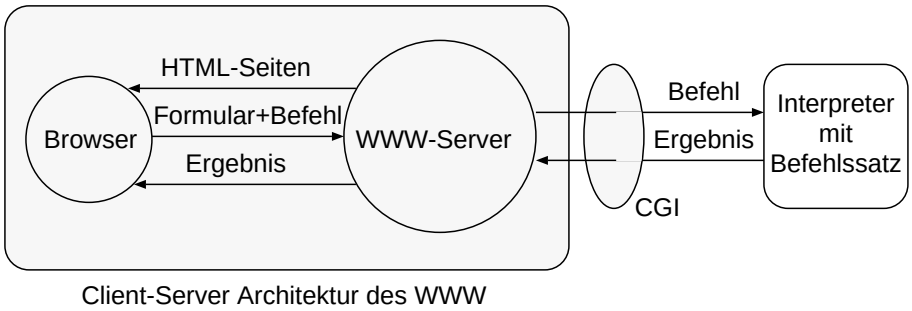

Abbildung 2.3: Befehlsinterpreter für Befehle über das CGI

Durch den Aufbau der Steuereinheit als Interpreter kann der Befehlsumfang der Pinwand einfach und übersichtlich erweitert werden.

4. Trennung zwischen den Komponenten. Durch die Trennung können später ohne Einfluß auf die anderen Komponenten die Datenbasis oder das Interface geändert werden. Zum Beispiel kann später aus Gründen der Performance von einer Speicherung der Einträge als Dateien zu einer relationalen Datenbank gewechselt werden.

Gelöstes Beispiel: Im Beispiel der Groupware-Anwendung wird die Pinwand für Informationen verwendet, die potentiell für jedes Teammitglied interessant sind. Dadurch gewinnt der Informationsanbieter Zeit, da er sich nicht überlegen muß, an wen er seine Informationen schicken soll. Daneben gewinnen auch alle anderen Mitglieder Zeit, da sie sich nur mehr Informationen holen, die sie wirklich benötigen.

Varianten: Die Varianten werden als eigene Patterns in den folgenden Abschnitten beschrieben. Dabei handelt es sich um:

- Die strukturierte Pinwand und
- die verteilte Bibliothek.

Bekannte Anwendungen: Weitere Anwendungen des Pinwand-Patterns sind unter anderem:

- News Network Transfer Protokoll [Kantor and Lapsley, 1986]
- Diskussionsforen
- Schwarzes Brett (Bulletin Board)
- Guestbooks im WWW
- Conferencing Systeme
- Annotation Tools (für Feedback und Kommentare)
- Subscriber Systeme

Konsequenzen: Die Verwendung einer Pinwand bringt folgende Vorteile:

- Die Information steht jedem, genau wenn er sie benötigt zur Verfügung. Man muß nicht Informationen sammeln, die man später eventuell benötigen könnte.
- Man wird nicht ständig durch neue Informationen, die man nicht oder nicht jetzt benötigt, überlastet. Informationen sind eine Holschuld.
- Die Informationen werden an einer Stelle gesammelt. Dadurch wird ein einfacher Zugang gewährleistet.

Nachteile und Einschränkungen einer Pinwand können sein:

- Einmal eingetragene Informationen können vom Informationsanbieter schwer oder gar nicht verändert und erweitert werden.
- Eine Pinwand ist nicht für vertrauliche Informationen geeignet, da jeder die Informationen auf der Pinwand lesen kann.
- Informationen, die schnell eine bestimmte Person erreichen müssen, sollen weiterhin durch direkte Kommunikation (zum Beispiel Email) weitergeben werden, um die Aufmerksamkeit des Empfängers sofort zu erregen.

2.3 Pattern: Strukturierte Pinwand

Name: Mit der *strukturierte Pinwand* sammelt man strukturierte Informationen und macht sie allen anderen Benutzern zugänglich.

Auch bekannt als: —

Beispiel: Es wird eine *Tauschbörse* im Internet eröffnet. Es sollen die Angebote gesammelt und in den Kategorien *Suche* und *Biete* für die Interessierten dargestellt werden.

Um die Angebote einheitlich zu gestalten, wird das Formular in Abb. 2.4 verwendet. Biete oder Suche ist anzukreuzen. Die Bezeichnung, eine kurze Beschreibung und die Email-Adresse für die Kontaktaufnahme sind auszufüllen.

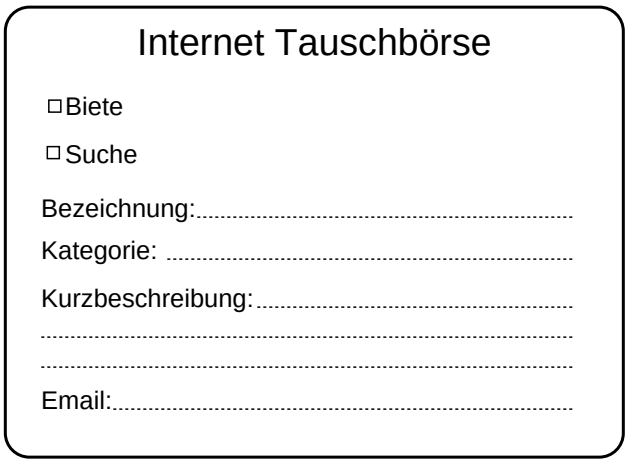

Abbildung 2.4: Formular der Tauschbörse

Zusätzlich wird noch eine Kategorie angegeben, zu der das Tauschobjekt gehört. Kategorien können für eine EDV-Tauschbörse zum Beispiel Software, Hardware und Bücher sein.

Kontext: Personen wollen bestimmte Informationen anbieten oder schnell finden.

Problem: Durch die Verwendung einer einfachen Pinwand kann das folgende Problem nicht gelöst werden:

- Unstrukturierte Informationen. Einfache Texte verlangen keine logische Strukturierung der Informationen. Dadurch wird die Suche nach bestimmten Informationen erschwert. Außerdem ist der Vergleich von ähnlichen Informationen aufwendig.

Lösung: Für die Pinwand werden strukturierte Formularen verwendet. Dadurch wird die Auswahl der passenden Information auf der Pinwand erleichtert und kann weitgehend automatisiert werden. Dies führt zum Aufbau eines *Intelligent Information-Sharing System* [Malone *et al.*, 1987].

Struktur: Siehe Pinwand-Pattern.

Laufzeit Verhalten: Siehe Pinwand-Pattern.

Implementation: Die Implementation baut auf dem Pinwand-Pattern auf. Zusätzlich müssen folgende Punkte beachtet werden:

1. Entwicklung des Formulars. Die Entwicklung eines geeigneten Formulars ist der wichtigste Teil der Implementation. Es muß genau analysiert werden, welche Informationen gespeichert werden sollen, da fehlende Felder im Formular die Funktion stark einschränken können. Felder, die mindestens vorhanden sein sollten, sind:

- Die Bezeichnung der Informationen,
- die Email-Adresse des Informationsanbieters,
- eine Kurzbeschreibung,
- ein Kategorisierungsbegriff und
- die Information selbst.

Die Kurzbeschreibung soll maximal 5 Sätze lang sein und die wichtigen Aspekte der Information umfassen. Durch den Kategorisierungsbegriff wird die Information einer Gruppe zugeteilt. Für das Tauschbörsen-Beispiel sind diese Gruppen zum Beispiel Hardware, Software und Bücher. Sie schränken den Suchbereich grob ein. Zusätzlich können Schlagwörter verwendet werden, um die Suche zu erleichtern.

Das Formular soll so gestaltet werden, daß es beim Ausfüllen keine Mißverständnisse gibt. Dies wird dadurch erreicht, daß man es ähnlich wie existierende Formulare gestaltet und falls nötig Hinweise für das richtige Ausfüllen ins Formular aufnimmt.

2. Erweiterung des Interfaces. Die Strukturierung der Einträge ermöglicht es auf der Hauptseite der Pinwand nur eine Liste der Bezeichnungen der Informationen und eventuell auch ihre Kurzbeschreibungen anzugeben. Die Informationen selbst können dann von dieser Seite aus erreicht werden. Eine weitere Möglichkeit ist, für jede Kategorie von Einträgen eine eigene Hauptseite bereitzustellen. Diese Maßnahmen erleichtern die Suche nach den benötigten Informationen.
3. Erweiterung der Steuereinheit. In die Steuereinheit kann eine Suchfunktion eingebaut werden, die nach Schlagworten oder nach dem Vorkommen bestimmter Begriffe in der Kurzbeschreibung sucht. Dies nützt die Vorteile von strukturierten Texten für die automatische Verarbeitung.

Gelöstes Beispiel: Die Tauschbörse verwendet zwei Kategorisierungen. Die Einträge werden einerseits in die zwei Gruppen Suche und Biete eingeteilt. Andererseits wird zwischen Kategorien wie Hardware, Software und Bücher unterschieden. Mit diesen Informationen kann der Suchbereich eingeschränkt werden. Gesucht wird dann nach bestimmten Begriffen in der Bezeichnung oder der Kurzbeschreibung.

Varianten: Siehe Pinwand-Pattern.

Bekannte Anwendungen: Das Pinwand-Pattern ist ein Spezialfall der strukturierten Pinwand, der Einträge mit nur einem Feld hat. Dadurch sind alle Pinwand Anwendungen auch mit der strukturierten Pinwand realisierbar.

Konsequenzen: Die Verwendung der strukturierten Pinwand bringt folgende zusätzliche Vorteile:

- Strukturierung der Information. Strukturierte Informationen können schneller erfaßt werden.
- Automatische Verarbeitung. Durch die Verwendung von Feldern mit genau definiertem Inhalt kann die Information automatisch verarbeitet werden. Die maschinelle Suche wird erleichtert, da nur selten Volltext-Recherchen benötigt werden.
- Definiertes Formular. Das Formular bringt auch Vorteile für den Informationsanbieter. Er füllt einfach das gesamte Formular aus und kann damit sicher sein, daß nichts für die Präsentation seiner Informationen auf der Pinwand fehlt.

Zusätzliche Nachteile und Einschränkungen der strukturierten Pinwand sind:

- Spezialisierung durch das Formular. Informationen, die sich stark unterscheiden und dadurch verschiedene Formulare benötigen, können nicht gemeinsam auf einer Pinwand gesammelt werden.
- Lebenszeit von Einträgen. Wann ein Eintrag von der Pinwand gelöscht werden soll, ist bei Informationen die relativ schnell uninteressant werden, schwer zu entscheiden. Nimmt man die Lebenszeit eines Eintrags der Tauschbörse mit 4 Wochen an, kann es passieren, daß der Eintrag noch vorhanden ist, obwohl das Tauschobjekt längst den Besitzer gewechselt hat. Andererseits kann es sein, daß der Eintrag nach Ablauf der Frist gelöscht wird, ohne das ein Tausch durchgeführt wurde.

2.4 Pattern: Verteilte Bibliothek

Name: Die *verteilte Bibliothek* wird verwendet, um in einem Netzwerk vorhandene Dokumente für alle Benutzer leicht zugänglich zu machen.

Auch bekannt als: Virtual Library

Beispiel: An einer Universität existieren viele Unterlagen für diverse Kurse auch in Form von Hypertext-Dokumenten. Diese sind zwar über das universitätseigene Intranet für alle Interessierten verfügbar, doch befinden sie sich an verschiedenen Stellen, verstreut über das gesamte Netzwerk. Dadurch wird das Auffinden der gewünschten Unterlagen sehr zeitaufwendig.

Um den Zugang zu erleichtern wird eine Art erweitertes Inhaltsverzeichnis verwendet. Dieses Inhaltsverzeichnis beinhaltet, eine kurze Beschreibung und Schlagworte der Dokumente, sowie den Ort wo sie zu finden sind. In Abb. 2.5 wird der Aufbau der verteilten Bibliothek dargestellt. Die Dokumente sind im

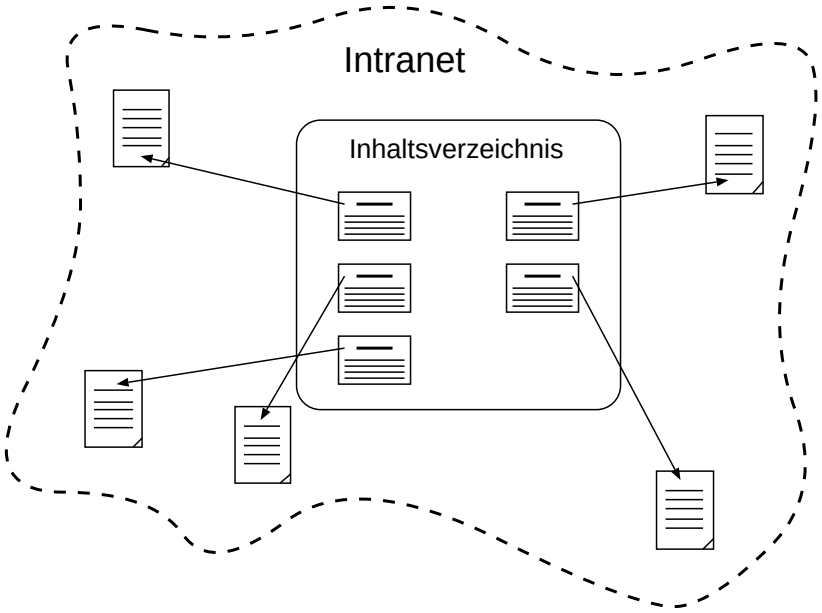

Abbildung 2.5: Aufbau der verteilten Bibliothek

Intranet verteilt. Als zentraler Anlaufpunkt für alle Benutzer dient das Inhaltsverzeichnis mit seinen Einträgen. Vom Inhaltsverzeichnis gehen Verweise, in Abb. 2.5 durch Pfeile symbolisiert, zu den betreffenden Dokumenten aus.

Da die Anzahl der Dokumente ständig ansteigt, soll die Verwaltung des Inhaltsverzeichnisses dezentral ablaufen, daß heißt, jeder Autor eines Dokuments erstellt selbst den nötigen Eintrag im Inhaltsverzeichnis mit dem Verweis auf sein Dokument. Um die Konsistenz innerhalb der Bibliothek zu wahren, ist noch eine Kontrollinstanz vorgesehen, die von den Autoren neu erstellte Einträge prüft, bevor sie endgültig zum Inhaltsverzeichnis hinzugefügt werden.

Kontext: Dokumente sind über das gesamte Netzwerk verstreut.

Problem: Bei Dokumenten, die dezentral erstellt und im Intranet angeboten werden, entstehen folgende Probleme:

- Das Auffinden von Dokumenten, von denen man den genauen Ort nicht genau kennt, ist sehr zeitaufwendig.
- Die Dokumente sind schwer zugänglich und oft nur über sehr lange Pfade erreichbar. Ein typisches Beispiel für eine Universität ist:
  1. Von der Homepage der Universität über
  2. die Homepage des Instituts zur
  3. Homepage eines Lehrbeauftragten und weiter über
  4. einige andere Seiten zum

5. Dokument.

   Jeder hier angeführte Schritt bedeutet das Suchen nach den richtigen Verweisen auf mehreren Seiten. Und das ganze funktioniert nur, falls kein Verweis ins Nichts führt, und bei jeder Verzweigung der richtige Pfad gewählt wird.

- Die Verweise zu ähnlichen Dokumenten anderer Anbieter sind nicht ausreichend. Dadurch wird Vergleichen und Auswählen schwieriger.
- Von der Existenz vieler Dokumente wissen potentiell Interessierte nichts. Diese Dokumente werden zu wenig genutzt, was eine Vergeudung von Ressourcen bedeutet.

Lösung: Das Konzept hinter der verteilten Bibliothek ist ein Karteikartensystem. Diese Karteikarten enthalten Informationen über die Dokumente, die sie repräsentieren. Sie werden im Inhaltsverzeichnis gesammelt und stehen dort für Recherchen durch Informationssuchende zur Verfügung. Von jeder Karteikarte kommt man durch einen Verweis direkt zum dazugehörigen Dokument.

Die Karteikarten werden von den Autoren selbst ausgefüllt und nur mehr durch den Administrator der Bibliothek überprüft, bevor sie in das Inhaltsverzeichnis aufgenommen werden. Dies ist nötig um die Konsistenz der Bibliothek zu wahren und Fehler in den Karteikarten zu vermeiden.

Struktur: Siehe Pinwand-Pattern.

Laufzeit Verhalten: Die verteilte Bibliothek benötigt folgende Abläufe:

1. Benutzer führen Recherchen durch. Diese werden von der Bibliothek durch die Suchfunktion unterstützt.
2. Neue Beiträge von Informationsanbietern werden für die Kontrolle durch den Administrator gesammelt.
3. Der Administrator kontrolliert neue Einträge und fügt sie dem Inhaltsverzeichnis hinzu.
4. Nicht mehr gültige Beiträge werden gelöscht oder verändert.

Wie beim Pinwand-Pattern können mehrere Abläufe parallel ausgeführt werden.

Implementation: Die Implementation baut auf der strukturierten Pinwand auf. Zusätzlich müssen folgende Punkte beachtet werden:

1. Erweiterung der Datenbasis. Die Datenbasis muß die neuen Einträge für die Benutzer nicht erreichbar, bis zu ihrer Überprüfung speichern. Es stehen dafür zwei Möglichkeiten zur Verfügung.

Da das Inhaltsverzeichnis eine strukturierte Pinwand darstellt, ist es möglich einfach eine zweite Pinwand als Zwischenspeicher vorzuschalten. Neue Einträge werden dann in eine nicht öffentliche Pinwand eingetragen und nur der Administrator kann diese nach der Überprüfung in das Inhaltsverzeichnis übertragen.

Die andere Möglichkeit ist, daß alle Einträge sofort in das Inhaltsverzeichnis eingetragen werden. Die neuen Einträge sind jedoch gekennzeichnet und werden vom Interface herausgefiltert und nicht an die Benutzer weitergegeben. Erst nach der Freigabe durch den Administrator wird diese Kennzeichnung gelöscht.

Außerdem ist wichtig, daß sich die Datenbasis für *Information Retrieval* eignet. Eine Suche, die Fehler in den Begriffen erlaubt, sowie die Unterstützung einer logischen Verknüpfung von Begriffen sind für die Brauchbarkeit der Bibliothek ausschlaggebend.

2. Erweiterung des Interface. Das Interface für die Benutzer soll folgende zwei Zugänge integrieren:

   (a) Darstellung des Inhalts der Bibliothek nach *Kategorien*. Diese Kategorien müssen einerseits das von der Bibliothek erfaßte Fachgebiet abdecken und andererseits für die Informationsanbieter und die Benutzer klar sein. Um diese geforderten Eigenschaften einhalten zu können sollten die Kategorisierungsbegriffe einem *Thesaurus* entnommen werden.

   (b) Darstellung des Inhaltes der Bibliothek als *Suchergebnis*. Das Ergebnis einer Suche nach Begriffen in den Einträgen der Bibliothek wird als Liste dargestellt.

3. Erweiterung der Steuereinheit. Die Steuereinheit muß die zusätzlich benötigten Funktionen bereitstellen. Dies sind hauptsächlich Funktionen für den Administrator, wie der Ablauf für die Überprüfung der neuen Einträge und das Verändern und Löschen von Einträgen im Inhaltsverzeichnis. Für die Administrationsfunktionen muß sichergestellt werden, daß sie nur vom Administrator ausgeführt werden können. Dies kann durch eine Paßwortabfrage oder durch die Verwendung eines getrennten Zugangs für die Administration erreicht werden.

   Zu welchem Zeitpunkt Einträge gelöscht werden, ist wieder eine Frage mit der man sich genauer beschäftigen muß. Für Dokumente, die schnell an Aktualität verlieren, kann eine festgelegte Lebensdauer verwendet werden. Für andere Dokumente, wie zum Beispiel Bücher und Artikel, ist dies nicht so einfach. Ihre Einträge sollten so lange in der Bibliothek bleiben, bis die Dokumente selbst nicht mehr angeboten werden. Man kann sich entweder darauf verlassen, daß der Autor, wenn er sein Dokument nicht mehr anbietet, den Administrator der Bibliothek verständigt, oder ständig überprüfen,

ob das Dokument noch erreichbar ist. Kann das Dokument für längere Zeit nicht erreicht werden, muß man davon ausgehen, daß es nicht mehr angeboten wird und der Eintrag in der Bibliothek wird gelöscht. Die laufende Überprüfung, ob die Dokumente erreichbar sind, kann automatisiert werden.

Gelöstes Beispiel: Die verteilte Bibliothek der Universität sammelt Informationen über Kursunterlagen wie:

- Bezeichnung des Kurses.
- URL der Unterlagen.
- Name und Email-Adressen des Kursleiters.
- Was in den Unterlagen angeboten wird. Zum Beispiel Informationen über den Kurs, Klausurergebnisse, alte Klausuren mit Lösungen, Links zu ähnlichen Informationen ...
- Kurzbeschreibung der Inhalte und Ziele des Kurses.
- Zuordnung zu einem Fachgebiet.
- Schlagworte.

Die Interessierten könne die Informationen in der Bibliothek nach Fachgebieten geordnet betrachten, oder diese nach Begriffen durchsuchen. Wurde ein passender Eintrag gefunden, kann man direkt zum Dokument wechseln. Durch die Bibliothek werden die langen Wege zu den Dokumenten einheitlich auf zwei Schritte verkürzt. Der erste Schritt ist die Eingabe des Suchbegriffs und der zweite die Auswahl des Eintrags, der die gesuchten Unterlagen beschreibt.

Die Bibliothek ist nicht nur für die Studierenden der Universität gedacht, sondern dient auch dazu externen Personen eine einfache Zugangsmöglichkeit zu Unterlagen zu geben. Ein Ziel ist es, interaktive Unterlagen zum Selbststudium an der Universität zu entwickeln und sie über die Bibliothek anzubieten. Dies soll helfen die Universität als Anbieter für durch moderne Medien unterstützte Ausbildung zu etablieren. Dieser Ruf brächte mehr Forschungsgeld für die Universität und verbesserte Zukunftschancen für ihre Absolventen.

Varianten: Es kann die Funktionalität einzelner Komponenten der verteilten Bibliothek verändert oder erweitert werden. Daraus entstehen zum Beispiel folgende Varianten:

- Verteilte Bibliothek mit *veränderbaren Einträgen*. Die Einträge in der Bibliothek können von den Autoren, die sie erzeugt haben später verändert werden. Dies setzt voraus, das jeder Eintrag bei der Erzeugung an seinen Autor gebunden wird und nur von diesem verändert werden kann. Dies kann durch die Verwendung von Paßwörtern, Zertifikaten oder anderer Verfahren erreicht werden.

- Verteilte Bibliothek mit *automatisierter Dokumentensammlung*. Durch die automatisierte Dokumentensammlung müssen die Autoren keine Informationen direkt in die Bibliothek eintragen. Ein *Intelligenter Agent* sammelt automatisch die von der Bibliothek benötigten Informationen aus den Dokumenten, die er im Netzwerk findet. Dieser Ansatz führt zu einer *Suchmaschine*, die selbständig Informationen sammelt und von den Autoren der Dokumente unabhängig ist.

  Ein Problem ist die komplizierte Analyse der Dokumente, um die benötigten Informationen zu extrahieren. Dieses Problem kann umgangen werden, indem jedes Dokument in einem für den Leser unsichtbaren Kopfteil die benötigten Informationen in automatisch verarbeitbarer Form enthält. Im Internet werden für diesen Zweck *Meta*-Einträge verwendet. Ihre Verwendung wird im *Request for Comment 1866* [Berners-Lee and Connolly, 1995] beschrieben.

Bekannte Anwendungen: Verteilte Bibliotheken können in vielen Bereichen angewendet werden. Anwendungsmöglichkeiten im Rahmen einer *virtuellen Universität* sind:

- Living Lectures – Virtual University. Projekt an der Wirtschaftsuniversität Wien mit dem Ziel Lehrveranstaltungen mit multimedialen „elektronischen“ Büchern zu unterstützen.
- Component Library – Virtual University. Sammlung von Tools zur Unterstützung der Lehre durch moderne Medien.
- Projektverfolgung durch die Sammlung von Projektdokumentationen im Rahmen von Lehrveranstaltungen.
- Teilnehmerverwaltung für Lehrveranstaltungen durch die Unterstützung von on-line Anmeldungsformularen.
- Linksammlung. Sammlung von Verweisen zu interessanten Informationsanbietern. Die Bibliothek erkennt unbrauchbare Verweise und macht dem Bibliothekar auf diese aufmerksam. Außerdem kann jeder dem Bibliothekar neue Links zur Aufnahme in die Sammlung vorschlagen.

Die gerade beschriebenen Anwendungen können auch für andere Organisationen umgesetzt werden. Ihr Schwerpunkt liegt dann nicht im Lehrbetrieb, sondern in der Unterstützung der unternehmensweiten Arbeitsabläufe *(Workflow)* oder in der Information von Kunden.

Konsequenzen: Die Verwendung der verteilten Bibliothek als Inhaltsverzeichnis bringt folgende Vorteile:

- Dokumente können einfach gefunden werden ohne, daß man genau weiß, wo sie gespeichert werden. Der Verweis aus dem dazugehörigen Eintrag in der Bibliothek genügt.

- Der Zugang zu allen Dokumenten wird vereinheitlicht. Alle Dokumente können in zwei Schritten von der Bibliothek aus erreicht werden. Dies ist eine wesentliche Verkürzung der Zugriffspfade.
- Der Vergleich mit ähnlichen Dokumenten wird erleichtert, da sie bei der Suche nach bestimmten Begriffen automatisch gemeinsam angezeigt werden.
- Es können auch Dokumente erreicht werden, von deren Existenz man nichts weiß. Dadurch werden Recherchen über ein unbekanntes Gebiet möglich.
- Durch die Sammlung von Dokumenten, die in einer Organisation erstellt werden, entsteht eine *Collective Dynabase* [Press, 1992]. Informationen, die in einer Organisation erzeugt oder verarbeitet werden, stehen damit allen Mitarbeitern zur Verfügung und können in Zukunft als Grundlage genützt werden.

Nachteile und Einschränkungen der verteilten Bibliothek sind:

- Die Bibliothek ist verteilt. Dies bedingt, daß einzelne Dokumente zeitweilig nicht erreichbar sind, falls ein Teil des Netzwerkes ausfällt.
- Die Erkennung nicht mehr verfügbarer Dokumente. Die Autoren benachrichtigen den Administrator der Bibliothek nicht darüber, daß sie ein Dokument nicht länger anbieten. Dadurch entsteht bis zum Löschen des Eintrags ein Verweis ohne gültiges Ziel.
- Die Autoren können ihre Einträge in der Bibliothek nicht selbst verändern. Daher muß der Administrator diese Veränderungen durchführen. Um den Bedarf an Veränderungen möglichst gering zu halten, sollten die Einträge in der Bibliothek keine aktuellen Informationen beinhalten.

  Werden trotzdem Einträge benötigt, die von den Autoren regelmäßig verändert werden, muß die *verteilte Bibliothek mit veränderbaren Einträgen* verwendet werden (siehe: Varianten dieses Patterns). Der Nachteil dieser Variante ist, daß die zentrale Kontrolle der Informationen in der Bibliothek fehlt. Einträge werden nur überprüft, wenn sie neu in die Bibliothek eingetragen werden und nicht nach jeder Änderung durch den Autor.

Teil II

Implementation der virtuellen Bibliothek

Kapitel 3

Einführung

In diesem Teil wird eine mögliche Implementation des Patterns *verteilte Bibliothek* beschrieben. Der Programmcode befindet sich im Anhang C.

Die virtuelle Bibliothek ist ein WWW-basierendes System, das den Zugang zu Informationen im Intranet vereinheitlicht und lange Pfade zu relevanten Informationen auf 2 Ebenen verkürzt. Durch die einheitliche Repräsentation der Informationen in der Bibliothek wird die Suche effizienter. Vorteile dieser virtuellen Bibliothek sind die Skalierbarkeit, die einfache Anpaßbarkeit und die intuitive Bedienung.

Ist auf einem WWW-Server der VirLib-Server installiert, können beliebig viele Bibliotheken unabhängig voneinander betrieben werden. Die Anpassung der einzelnen Bibliotheken erfordert nur die Veränderung von HTML-Seiten. Da der gesamte Ablauf der virtuellen Bibliothek über WWW-Browser abgewickelt wird, kann der Endbenutzer eine ihm vertraute grafische Bedienungsoberfläche verwenden. Außerdem sind dadurch die Administration der Bibliotheken und des VirLib-Servers von jedem Internetzugang aus möglich. Als Sicherheitskonzept für die Administration über das Internet wird *Secure Socket Layer (SSL)* verwendet, da es von fast allen Browsern unterstützt wird.

Die folgende Beschreibung der Implementation der virtuellen Bibliothek besteht aus fünf Teilen.

Im Kapitel 4 wird der Aufbau der virtuellen Bibliothek beschrieben. Besonderer Wert wird auf die Trennung der einzelnen Komponenten gelegt.

Das Kapitel 5 beschreibt die Bedienung der virtuellen Bibliothek nach Anwendergruppen getrennt.

Im Kapitel 6 wird auf die Installation der virtuellen Bibliothek eingegangen. Es werden alle nötigen Anpassungen beschrieben.

Die Kapitel 7 und 8 werden nur benötigt falls man den Ablauf der virtuellen Bibliothek verändern möchte. Das Kapitel 7 enthält eine Aufzählung der Befehle, die den Ablauf der Bibliothek steuern. Hier wird gezeigt wie diese Befehle in eigene HTML-Seiten

integriert werden können. Das Kapitel 8 beschreibt den Aufbau von Skripts. Es handelt sich um kleine Programme, die der Bibliothek neue Funktionalität hinzufügen. Sie können wie Befehle verwendet werden.

Folgende Beschreibungsmethoden werden in dieser Dokumentation verwendet:

- Zustandsdiagramme
- Datenflußdiagramme
- Pseudocode
- HTML-Code

Die Zustands- und Datenflußdiagramme werden nach [Rumbaugh *et al.*, 1991] verwendet.

3.1 Anwendungsbereich

Die virtuelle Bibliothek hat aufgrund ihrer Konzeption einen sehr großen Anwendungsbereich. Dieser erstreckt sich von einfachen *Pinwänden* bis zu *verteilten Bibliotheken*, welche für viele Zwecke eingesetzt werden können. Einige Beispiele aus den Bereichen Informationserschließung und Lehre sind:

- Kategorisierung und Erschließung von Unterlagen, die bereits in Hypertext vorhanden sind. Ein Beispiel dafür ist das Projekt *Living Lectures – Virtual University* an der Wirtschaftsuniversität Wien.
- Bibliotheken für Tools, Manuals ...
- Schwarzes Brett zum Informationsaustausch und Koordination innerhalb von Gruppen (Groupware).
- Organisation von Projekt-Lehrveranstaltungen durch die Sammlung der Projektunterlagen und Feedback.

3.2 Voraussetzungen

Die virtuelle Bibliothek benötigt folgende Komponenten:

- UNIX-basierendes System
- WWW-Server
- Perl 5
- GLIMPSE 4.0
- at
- Make

3.3 Distribution

Die virtuelle Bibliothek besteht aus folgenden zwei Paketen:

- VirLib-Server
- Bibliothek

Außerdem wird die Information Retrieval und Indexierungssoftware GLIMPSE benötigt. Zusätzliche Skripts, die spezielle Funktionen zu Bibliotheken hinzufügen, werden auf der *Homepage der Virtual Library* angeboten.

Kapitel 4

Struktur der virtuellen Bibliothek

Das Pattern verteilte Bibliothek beschreibt die drei Komponenten *Interface, Steuereinheit* und *Datenbasis.* Die Realisierung dieser Komponenten wird in Abb. 4.1 dargestellt. Das Interface besteht aus dem WWW-Server mit seinen HTML-Seiten und den WWW-Browsern. Die Steuereinheit ist über das *Common Gateway Interface* mit dem WWW-Server verbunden und heißt VirLib-Server. Die Datenbasis wird hier als Bibliothek bezeichnet, wobei mehrere solcher Bibliotheken von einem VirLib-Server verwaltet werden können. Um jeder dieser Bibliotheken ein individuelles Aussehen geben zu können, ist ein Teil des Interfaces mit der jeweiligen Datenbasis verbunden. Außerdem besitzt die virtuelle Bibliothek eine zusätzliche Komponente, die nicht in Pattern beschrieben wurde, nämlich die *automatischen Dienste.* Diese Dienste laufen getrennt vom Rest der Bibliothek ab und führen automatisch bestimmte Funktionen aus, um den Administrator und die Bibliothekare zu entlasten.

In den folgenden Abschnitten wird auf die einzelnen Komponenten näher eingegangen.

4.1 Steuereinheit – Virlib-Server

Die Aufgabe der Steuereinheit ist die Steuerung des Ablaufes in der virtuellen Bibliothek. Sie ist als Interpreter aufgebaut, der durch den WWW-Server aufgerufen wird und seine Befehle über das *Common Gateway Interface* bekommt. Die Befehle und ihre Parameter werden im Kapitel 7 beschrieben.

Pseudocode für den Aufbau der Steuereinheit:

```
BEGIN
  Daten vom Common Gateway Interface einlesen
  Daten konvertieren
  Befehl und Parameter erkennen
```

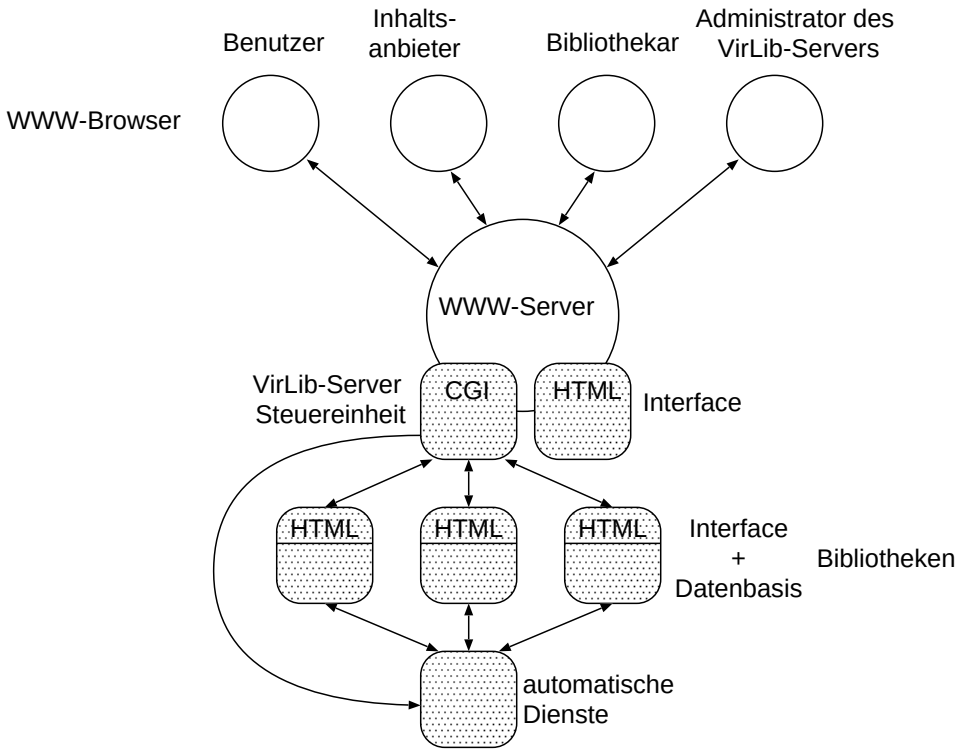

Abbildung 4.1: Aufbau des virtuellen Bibliothekssystems

```
  IF (Befehl korrekt und Benutzer zur Ausfueh-
rung berechtigt)
  BEGIN
    Befehl interpretieren
    Ergebnis an das Common Gateway Interface zurueckgeben
  END
  ELSE
  BEGIN
    Fehlermeldung an das Common Gateway Inter-
face zurueckgeben
  END
END
```

4.2 Interface

Das Interface besteht aus statischen- und dynamischen HTML-Seiten. Diese Seiten beinhalten Verweise und Formulare, die der Steuereinheit Befehle geben. Wird solch ein Verweis oder Formular von einem Anwender ausgewählt, startet der WWW-Server die Steuereinheit und übergibt ihr den Befehl mit den zugehörigen Daten. Der Befehl

wird durch die Steuereinheit abgearbeitet. Der Anwender bekommt das Ergebnis als dynamisch erzeugte Seite zurück.

Wie Befehle in Verweise oder Formularen integriert werden, wird im Kapitel 7.1 beschrieben.

In Abb. 4.1 ist ersichtlich, daß das Interface mit vier Anwendergruppen zusammenarbeitet. Im Pattern verteilte Bibliothek wurden aber nur drei beschrieben. Dies folgt aus der Trennung der Administrationstätigkeiten bei der virtuellen Bibliothek in zwei Bereiche. Es gibt einen Administrator für den VirLib-Server (Steuereinheit) und dann noch die Bibliothekare (Administratoren der einzelnen Bibliotheken). Für jede dieser Anwendergruppen sind auf ihre Bedürfnisse zugeschnittene HTML-Seiten (Administrationsseiten) vorhanden.

4.3 Datenbasis – die Bibliotheken

Der VirLib-Server kann mehrere Bibliotheken verwalten. Jede Bibliothek hat einen eigenen Bibliothekar und eine eigene Datenbasis. Die Datenbasis ist folgenderweise aufgebaut.

Jeder Eintrag wird als eigene Datei gespeichert. Diese Datei enthält in einer Attribut-Wert Liste die Informationen des Eintrags.

Beispiel für die Form eines Eintrags:

```
Titel := {Virtual Library}
URL := {miss.wu-wien.ac.at/~virlib/}
```

Die virtuelle Bibliothek unterstützt mehrere Stapel. Das heißt, die Datenbasis ist in mehrere Bereiche unterteilt, die unterschiedliche Informationen speichern. Nämlich die Stapel *Neue Einträge (new)*, *Bibliothek (library)* und *Mistkübel (old)*. Der Aufbau der Datenbasis wird durch das Datenflußdiagramm in Abb. 4.2 dargestellt.

Einträge nehmen folgenden Weg durch die Stapel der Datenbasis. Ein neuer Eintrag wird durch den Inhaltsanbieter in den Stapel *Neue Einträge* eingetragen. Dabei wird das aktuelle Datum hinzugefügt. In regelmäßigen Abständen überprüft der Bibliothekar die neuen Einträge und überträgt sie in den Stapel *Bibliothek*. Falls ein Eintrag unbrauchbar ist, kann er sofort gelöscht werden. Solange die Informationen, auf die ein Eintrag in der *Bibliothek* weist, vorhanden sind, bleibt der Eintrag in der Bibliothek und steht für Abfragen durch die Benutzer bereit. Wird ein Eintrag nicht mehr benötigt, kann er vom Bibliothekar gelöscht werden. Dieser Eintrag wird nicht sofort gelöscht, sondern nur markiert, und erst bei der nächsten Indexierung gelöscht. Der Grund dafür liegt im verwendeten Indexierungsverfahren. Zusätzlich werden die Einträge in der *Bibliothek* regelmäßig durch die automatischen Dienste überprüft.

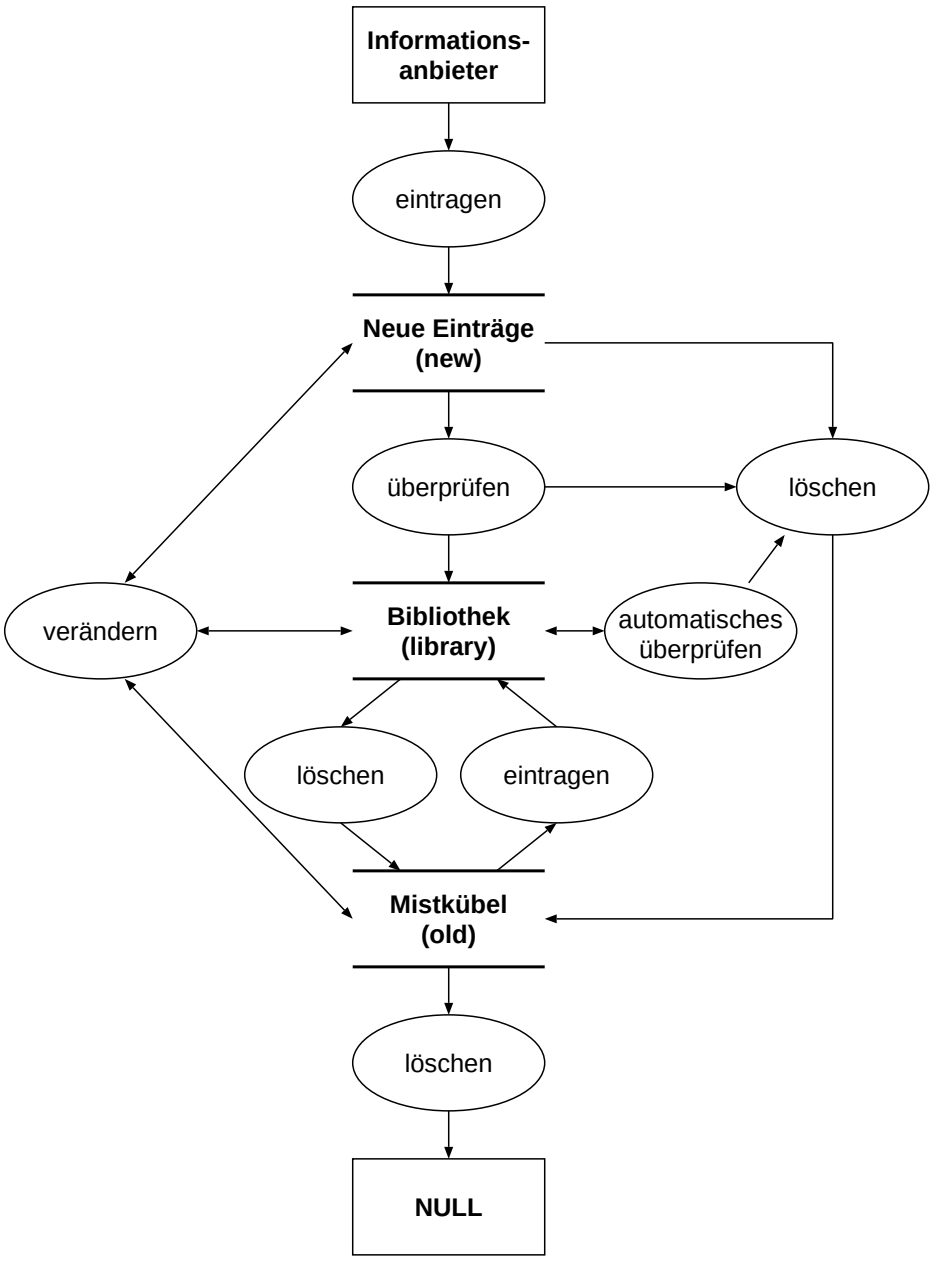

Abbildung 4.2: Verwaltung von Einträgen

Gelöschte Einträge werden im Stapel *Mistkübel* gesammelt. Von dort kann man sie wieder in die Bibliothek eintragen oder endgültig löschen. Außerdem kann der Bibliothekar die Einträge in jedem Stapel verändern.

Zur Kommunikation zwischen den Informationsanbietern oder anderen Benutzern und dem Bibliothekar, ist der Stapel Rückmeldung *(feedback)* vorhanden.

4.4 Automatische Dienste

Der VirLib-Server führt einmal pro Tag bestimmte Funktionen für alle angemeldeten Bibliotheken aus. Welche Funktionen ausgeführt werden, bestimmen die Bibliothekare der einzelnen Bibliotheken auf ihren Administrationsseiten. Folgende Funktionen können automatisch ausgeführt werden:

- Überprüfen, ob die Verweise der Einträge erreichbar sind.
- Unerreichbare Einträge in den *Mistkübel (old)* verschieben.
- Index der Bibliothek aktualisieren.
- Ein *Administrations-Skript (siehe Kapitel 8)* der jeweiligen Bibliothek ausführen.

Das Ergebnis der letzten automatischen Dienste kann der Administrator des Virlib-Servers in der *Log*-Datei überprüfen. Sollte ein Problem aufgetreten sein, erhält der Bibliothekar der betroffenen Bibliothek eine Nachricht als Feedback.

Pseudocode der automatischen Dienste:

```
BEGIN
  Liste der angemeldeten Bibliotheken lesen
  FOR (jede angemeldete Bibliothek)
  BEGIN
    Konfiguration der Bibliothek der automatischen Dien-
ste lesen
    Funktionen ausfuehren
    Ergebnis an die Log-Datei haengen
    IF (Problem aufgetreten) THEN Feedback fu-
er den Bibliothekar
  END
END
```

Kapitel 5

Bedienung der virtuellen Bibliothek

Die Bedienung der virtuellen Bibliothek ist nach den Rollen Benutzer, Informationsanbieter, Bibliothekar und Administrator des VirLib-Servers gegliedert.

5.1 Bedienung durch den Benutzer

Der Benutzer verwendet die Bibliothek um Informationen abzufragen. Diese Abfrage ist in der virtuellen Bibliothek zweistufig aufgebaut. In der ersten Stufe schränkt der Benutzer den Suchbereich durch die Eingabe von Begriffen ein. Als Ergebnis erhält er eine Liste der passenden Einträge, aus welcher der Benutzer den gewünschten Eintrag auswählen kann. Dieser Eintrag enthält die Beschreibung der verfügbaren Information und den Verweis dorthin. Außerdem ist es möglich, direkt aus der Liste den Verweis zur Information zu verwenden, und den Umweg über den Eintrag abzukürzen.

Für den gerade beschriebenen Ablauf stehen dem Benutzer folgende Funktionen, die auch im Zustandsdiagramm der Abb. 5.1 dargestellt werden, zur Verfügung:

Bibliothek durchsuchen: *Suche* nach Begriffen. Das *Ergebnis* der Suche ist eine Liste der gefundenen Einträge, aus welcher der Benutzer sich die gesamten *Einträge* anzeigen lassen kann, oder direkt zur *Information* gelangt.

Bibliothek: Eine *Liste* aller Einträge in der Bibliothek erzeugen. Dies ist ein Sonderfall der normalen Suche und folgt daher dem vorher beschriebenen Ablauf.

Feedback: Dem Bibliothekar *Feedback* geben. Nachdem das ausgefüllte Feedback-Formular *abgeschickt* wurde, erhält der Benutzer eine *Bestätigung*.

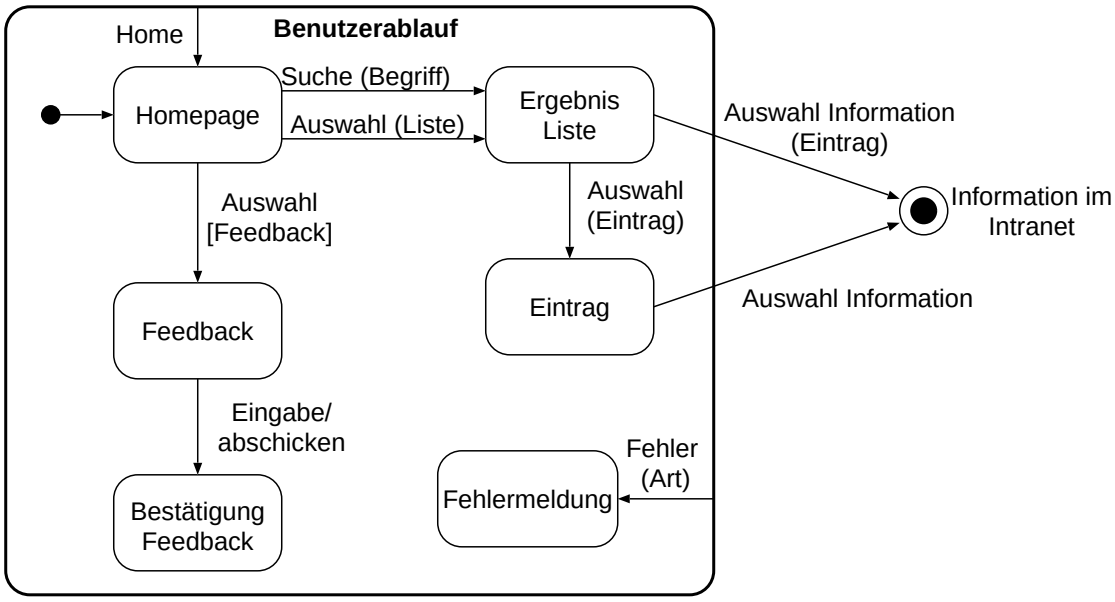

Abbildung 5.1: Benutzerablauf

5.2 Bedienung durch den Informationsanbieter

Der Informationsanbieter stellt seine Informationen durch die virtuelle Bibliothek den Benutzern zur Verfügung. Er füllt das Formular für neue Beiträge aus und schickt es ab. Dieser Vorgang wird im Zustandsdiagramm der Abb. 5.1 dargestellt. Es stehen dem Informationsanbieter folgende Funktionen zur Verfügung:

Neuer Beitrag: Einen *Beitrag* zur Bibliothek hinzufügen. Nachdem das ausgefüllte Beitrags-Formular *abgeschickt* wurde, erhält der Benutzer eine *Bestätigung*.

Feedback: Dem Bibliothekar *Feedback* geben. Es handelt sich um denselben Ablauf wie für die Benutzer.

5.3 Bedienung durch den Bibliothekar

Auf der Administrationsseite befinden sich alle Funktionen, die für die Betreuung der Bibliothek benötigt werden. Die Administrationsseite kann nur nach einer Abfrage des Paßworts erreicht werden. Sie enthält folgende Funktionen:

Neu: Liste der *Neuen Beiträge (new)*. Diese Beiträge können entweder zur Bibliothek hinzugefügt oder gelöscht werden.

Feedback: Liste des Stapels *Rückmeldungen (feedback)*.

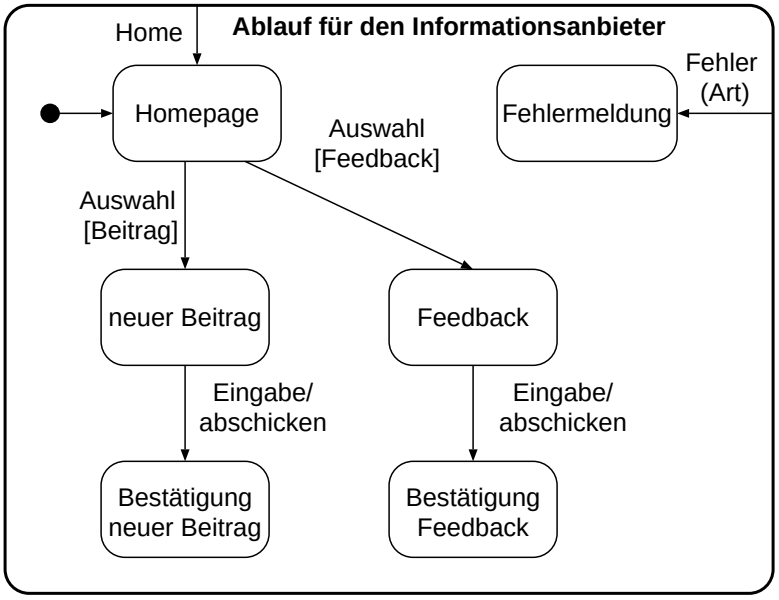

Abbildung 5.2: Ablauf für den Informationsanbieter

Bibliothek: Liste aller Einträge in der *Bibliothek (library)*. Die Einträge können verändert oder gelöscht werden.

Bibliothek durchsuchen: Ermöglicht dem Administrator die Suche nach Einträgen in der Bibliothek.

Bibliothek indexieren: Erstellt den Index und die Sortierungsdatei für die Bibliothek. Neue Einträge können erst nach ihrer Indexierung abgefragt werden. Da der Indexierungsvorgang bei größeren Bibliotheken recht lange dauern kann, sollte er durch die automatischen Dienste des VirLib-Servers ausgeführt werden.

Die Sortierungsdatei wird für die alphabetisch sortierte Ausgabe der Einträge in den Listen benötigt. Ist die Sortierungsdatei nicht aktuell, führt die sortierte Ausgabe zu einem unvollständigen oder falschen Ergebnis.

Automatische Dienste: Einstellungen für die Verwendung der automatischen Dienste. Hier kann man sich beim VirLib-Server für die Dienste an- und abmelden. Außerdem kann man die Konfiguration verändern, die festlegt, welche Dienste automatisch ausgeführt werden sollen. Die verfügbaren Funktionen werden im Anschluß beschrieben.

Admin-Skript ausführen: Führt das angegebene Skript aus. Es muß sich im Verzeichnis `./progs/admin` befinden.

Mistkübel: Beinhaltet alle gelöschten Einträge aus der *Bibliothek (library)* und den *Neuen Beiträgen (new)*. Wenn ein Eintrag im *Mistkübel (old)* gelöscht wird, ist dies endgültig.

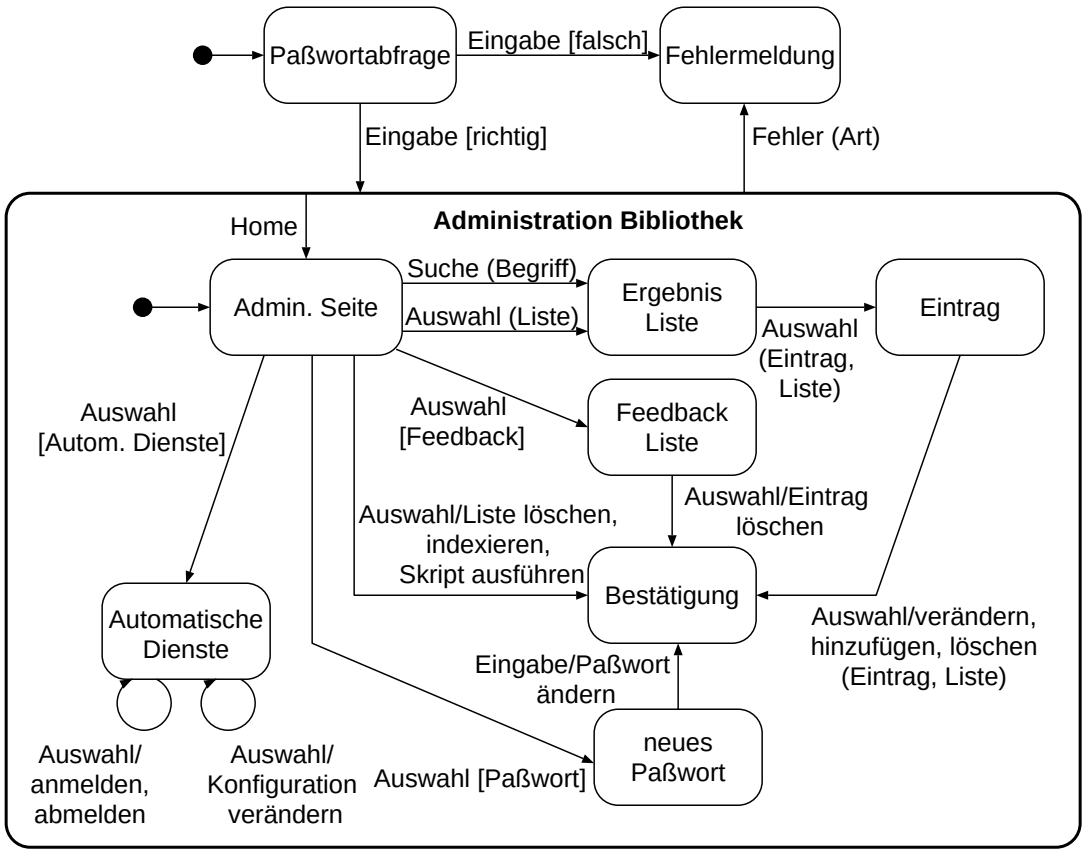

Abbildung 5.3: Administrationsablauf Bibliothek

Paßwort verändern: Ändert das Paßwort des Bibliothekars.

Feedback für den VirLib-Server: Erlaubt die Kommunikation mit dem Administrator des VirLib-Servers.

Neben den Stapelnamen auf der Administrationsseite, ist in eckigen Klammern die aktuelle Anzahl der Einträge angegeben. Der gesamte Ablauf dieser Funktionen wird im Zustandsdiagramm der Abb. 5.3 dargestellt.

Der VirLib-Server führt einmal pro Tag bestimmte Funktionen für die angemeldeten Bibliotheken aus. Welche Funktionen für jede einzelne Bibliothek ausgeführt werden sollen, kann der Bibliothekar auf seiner Administrationsseite unter dem Punkt *automatische Dienste* festlegen. Zur Verfügung stehen folgende Funktionen:

URLs überprüfen: Die Verweise in allen Einträgen der *Bibliothek (library)* werden auf Erreichbarkeit überprüft. Ist ein Verweis nicht erreichbar, wird dies im Eintrag vermerkt (*condition* erhält den Wert *unreachable*, *count* enthält seit wievielen Tagen der Eintrag nicht mehr erreicht werden konnte).

Automatisches Löschen: Bestimmt, ob ein Eintrag, der für eine festgelegte Anzahl von Tagen unerreichbar war, in den *Mistkübel (old)* verschoben werden soll.

Indexieren: Der Index der Bibliothek wird aktualisiert, und die Sortierungsdatei wird angelegt bzw. erneuert. Im Eingabefeld *Sortieren nach* wird der Variablenname angegeben, nach dessen Inhalt die Einträge alphabetisch sortiert ausgegeben werden sollen.

Admin-Skript: Führt ein *Administrations-Skript (siehe Kapitel 8)* der jeweiligen Bibliothek aus.

Nach der Installation ist die Bibliothek nicht angemeldet. Daher müssen die automatischen Dienste einer neuen Bibliothek erst angemeldet und konfiguriert werden.

5.4 Bedienung des VirLib-Servers durch den Administrator

Für den Administrator des VirLib-Servers wird eine spezielle Administrationsseite verwendet. Um auf diese Seite zu gelangen, benötigt man das Administrationspaßwort.

Von der Administrationsseite aus können alle Funktionen, die man für die Arbeit mit dem VirLib-Server benötigt, aufgerufen werden. Diese Funktionen und ihr Ablauf wird im Zustandsdiagramm der Abb. 5.4 dargestellt.

Es handelt sich um folgende Funktionen:

Feedback: Diese Liste enthält die Rückmeldungen für den VirLib-Server. Aus der Liste können Einträge gelöscht werden.

Automatische Dienste verwalten: Führt zum Interface für die Verwaltung der automatischen Dienste. Hier können die automatischen Dienste gestartet und gestoppt werden. Beim Starten kann der Zeitpunkt an dem die Dienste ausgeführt werden sollen, festgelegt werden. Nach der Installation sind die Dienste gestoppt.

Liste für automatische Dienste: Zeigt eine Liste der für die automatischen Dienste eingetragenen Bibliotheken an. Diese Liste kann verändert werden.

Log der automatischen Dienste: Zeigt das Ergebnis der letzten Durchführung der automatischen Dienste an.

Paßwort verändern: Verändert das Paßwort für den VirLib-Server.

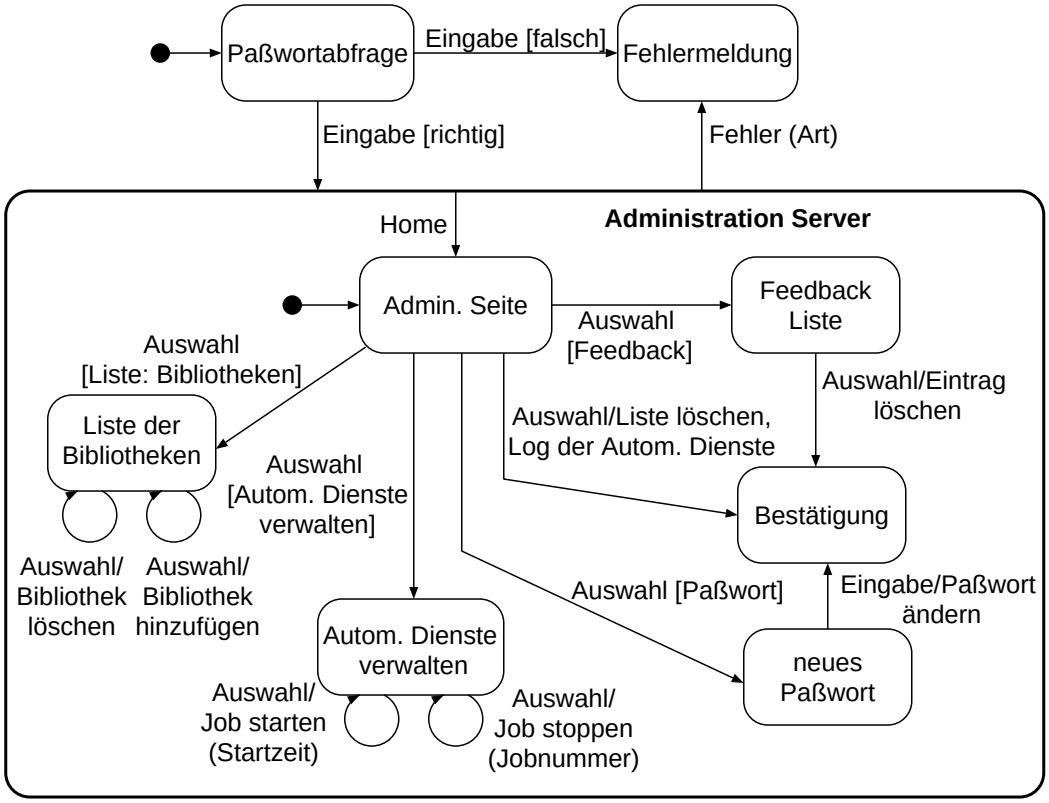

Abbildung 5.4: Administrationsablauf VirLib-Server

Kapitel 6

Installation

6.1 Der Virlib-Server

Die Installation läuft weitgehend automatisch ab. Da besonderen Wert auf die problemlose Installation gelegt wurde, muß hier auf die sich dadurch ergebenden Einschränkungen hingewiesen werden.

Falls Sie keine Schreibrechte für das CGI-Verzeichnis des WWW-Servers haben, der WWW-Server SSL nicht unterstützt oder Sie keine symbolischen Links verwenden können, lesen Sie sich den Abschnitt *Probleminstallationen (6.1.2)* vor der Installation durch.

Außerdem bietet die Standardinstallation keinerlei Sicherheit gegen Manipulation des VirLib-Servers durch Dritte, daher sollten Sie den Abschnitt *Sicherheitsaspekte (6.1.1)* beachten.

Für die Installation müssen folgende Schritte ausgeführt werden:

1. Sicherheitsvorkehrung im Abschnitt *Sicherheitsaspekte (6.1.1)* wählen und die nötigen Maßnahmen durchführen.

2. Erstellen des Verzeichnisses für den VirLib-Server und Entpacken des Archivs.

   ```
   mkdir VirLib
   cp VirLib.tar ./VirLib
   cd VirLib
   tar -xvf VirLib.tar
   ```

   Dadurch wird die Verzeichnisstruktur des VirLib-Servers, wie in Abb. 6.1 dargestellt, erzeugt. Im Verzeichnis `www` befinden sich die statischen, und in `html` die dynamischen HTML-Seiten. Das Verzeichnis `progs` beinhaltet die PERL-Programme. Das CGI-Skript des VirLib-Servers heißt `virlib.pl`. In `DOKU`

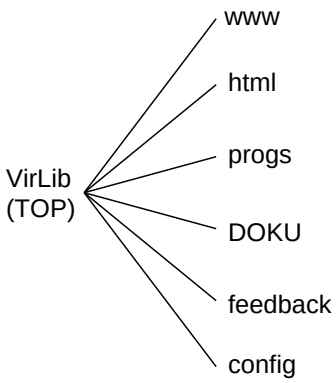

Abbildung 6.1: Verzeichnisstruktur des VirLib-Servers

befindet sich die Dokumentation der virtuellen Bibliothek. Die Verzeichnisse für den Rückmeldungsstapel (`feedback`) und die Konfigurationsdateien (`config`) werden erst später vom Makefile erzeugt.

3. Makefile: Pfade und Variablen anpassen.

   TOP: Pfad zum Verzeichnis des VirLib-Servers. Er muß mit einem Schrägstrich (`/`) enden.

   CGIPATH: Pfad zum CGI-Verzeichnis mit dem Namen des Programms (`virlib.pl`).

   PLHOME: URL, mit dem das CGI-Programm erreicht werden kann.

   PLHOMESECURE: URL, mit dem das CGI-Programm über einen verschlüsselten Kanal erreicht werden kann (zum Beispiel `https://...` bei *SSL*). Soll keine Verschlüsselung verwendet werden, kann man die Zeile `PLHOMESECURE = $(PLHOME)` verwenden.

   WWWPATH: Pfad zum Verzeichnis, in dem die HTML-Seiten abgelegt werden sollen. Dieses Verzeichnis muß vom WWW-Server für HTML-Seiten vorgesehen sein.

   WWWHOME: URL mit dem die gerade genannten HTML-Seiten erreichbar sind. Dieser Basis-URL ist nur das Verzeichnis in dem sich die Seiten befinden. Um eine Seite anzusprechen, muß noch ihr Name angefügt werden.

   GLIMPSEINDEX und GLIMPSE: Programmnamen mit den vollen Pfaden für die Indexierungsprogramme.

   FREE: Bestimmt das Paßwort, das der VirLib-Server nach der Installation verwendet. Das Standardpaßwort ist *„free“*. Das Paßwort sollte als erstes nach der Installation über die Administrationsseite des VirLib-Servers verändert werden.

ACCESS: Zugriffsrecht für das Verzeichnis des VirLib-Servers. Der Wert hängt von den gewählten Sicherheitsvorkehrungen ab.

4. `make install`

Jetzt ist der VirLib-Server bereit. Die Einstellungen im Makefile können jederzeit verändert werden. Durch erneutes Aufrufen von `make install` übernimmt der VirLib-Server die neuen Einstellungen.

Falls erwünscht, kann das Design der HTML-Seiten in `./www` und `./html` angepaßt werden. Diese Anpassung kann automatisch erfolgen. Dazu müssen die gleichen Schritte wie bei der automatischen Anpassung des Designs einer Bibliothek durchgeführt werden (siehe Abschnitt 6.2.2).

6.1.1 Sicherheitsaspekte

In der Grundkonfiguration können alle Benutzer des Rechners auf die Verzeichnisse des VirLib-Servers zugreifen. Dies ist notwendig, damit sichergestellt ist, daß der WWW-Server Zugriff hat. Es gibt mehrere Alternativen für dieses Sicherheitsproblem.

1. Die Standardeinstellung ist keine Sicherheitsvorkehrungen. Diese Alternative ist nur für Testzwecke ratsam. Der Makefile enthält den Befehl `ACCESS = 777`.

2. Den WWW-Server zum Eigentümer der Verzeichnisse des VirLib-Servers machen und die Zugriffsrechte nur auf ihn beschränken. Es hat die Nachteile, daß dies in der Regel nur der Administrator des Rechners kann, und danach der Administrator des VirLib-Servers keinen direkten Zugriff auf die Verzeichnisse des VirLib-Servers hat. Dazu muß der Befehl `ACCESS = 700` im Makefile verwendet werden. Diese Möglichkeit sollte in Betracht gezogen werden, falls der Administrator des VirLib-Servers gleichzeitig Administrator des Rechners ist.

3. Es wird ein neuer Benutzer (zum Beispiel *virlib*) erstellt, der derselben Gruppe wie der WWW-Server angehört. Im Verzeichnis dieses neuen Benutzers wird dann der VirLib-Server installiert. Im Makefile wird der Befehl `ACCESS = 770` verwendet. Diese Methode ist zu bevorzugen.

Die beschriebenen Maßnahmen verhindern den Zugriff über das Filesystem. Ein weiteres Sicherheitsproblem ist der WWW-Server selbst. Hier ist wichtig, daß Unbefugte keine CGI-Skripts installieren können, da diese mit den Rechten des WWW-Servers ausgeführt werden, und somit vollen Zugriff auf die Verzeichnisse des VirLib-Servers haben.

6.1.2 Probleminstallationen

Keine Schreibrechte auf das CGI-Verzeichnis. In diesem Fall muß der zuständige Administrator den Link vom CGI-Verzeichnis auf `./progs/virlib.pl` erzeugen. Danach sollte `make install` ohne Fehlermeldung ablaufen.

Der WWW-Server unterstützt kein SSL. Der VirLib-Server kann auch ohne Verschlüsselung arbeiten. Dazu muß im Makefile die Zeile
`PLHOMESECURE = https://...`
durch
`PLHOMESECURE = $(PLHOME)`
ersetzt werden. Durch diese Einstellung werden auch die Administrationsseiten unverschlüsselt übertragen, woraus natürlich ein Sicherheitsproblem resultiert.

Es können keine symbolischen Links verwendet werden. In diesem Fall müssen alle Dateien, die sonst durch Links erreichbar sind an den richtigen Platz kopiert werden. Folgende Schritte müssen ausgeführt werden:

1. Makefile: die Zeilen in denen der Befehl `ln -s` vorkommt, löschen.
2. `./progs/virlib.pl`: die Zeile
   `$plpath=readlink($0);`
   muß durch
   `$plpath =$0;`
   ersetzt werden.
3. `make install`
4. `./progs/virlib.pl` und `./progs/global.pl`: müssen in das CGI-Verzeichnis kopiert werden.
5. Verzeichnis ./www: muß mit allen Files in ein Verzeichnis für WWW-Seiten kopiert werden.

Jetzt sollte eine normale Installation möglich sein.

6.2 Die Bibliothek

Um eine Bibliothek zu installieren, muß der VirLib-Server auf dem WWW-Server installiert sein. Außerdem muß der WWW-Server auf die Verzeichnisse der Bibliothek Zugriff haben. Dies führt zu Sicherheitsproblemen, die im Abschnitt *Sicherheitsaspekte (6.2.1)* beschrieben werden.

Zur Installation der Bibliothek müssen folgende Schritte ausgeführt werden:

1. Informationen über den VirLib-Server einholen. Benötigt werden die Variablen `PLHOME` und `PLHOMESECURE` für den Makefile.

2. Sicherheitsvorkehrungen im Abschnitt *Sicherheitsaspekte (6.2.1)* wählen und die nötigen Maßnahmen durchführen.

3. Erstellen des Verzeichnisses für die Bibliothek und Entpacken das Archivs.

   ```
   mkdir aLibrary
   cp aLibrary.tar ./aLibrary
   cd aLibrary
   tar -xvf aLibrary.tar
   ```

   Dadurch wird die Verzeichnisstruktur der Bibliothek, wie in Abb. 6.2 dargestellt, erzeugt. Im Verzeichnis `www` befinden sich die statischen HTML-Seiten. Das Unterverzeichnis `images` enthält alle auf den Seiten der Bibliothek verwendeten Grafiken. Das Verzeichnis `html` enthält die dynamischen HTML-Seiten. Im Verzeichnis `progs` befinden sich die PERL-Programme. Die Unterverzeichnisse `admin` und `user` sind für Skripts der Bibliothek vorgesehen. Diese werden in Kapitel 8 beschrieben. In `DOKU` befindet sich die Dokumentation der virtuellen Bibliothek. Die Verzeichnisse für die Stapel, für die Konfigurationsdateien (`config`) und für die Indexierungsdateien (`index`) werden erst später vom Makefile erzeugt.

4. Makefile: Pfade und Variablen anpassen. Die Variablen sind:

   Name: Name der Bibliothek, der ihre Funktion beschreibt.

   TOP: Pfad zum Verzeichnis des VirLib-Servers. Er muß mit einem Schrägstrich (`/`) enden.

   PLHOME und PLHOMESECURE: Variablen vom Administrator des VirLib-Servers.

   WWWPATH: Pfad zum Verzeichnis, in dem die HTML-Seiten abgelegt werden sollen. Dieses Verzeichnis muß vom WWW-Server für HTML-Seiten vorgesehen sein.

   WWWHOME: URL mit dem die gerade genannten HTML-Seiten erreichbar sind. Dieser Basis-URL ist nur das Verzeichnis in dem sich die Seiten befinden. Um eine Seite anzusprechen muß noch ihr Name angefügt werden.

   FREE: Bestimmt das Paßwort, das der VirLib-Server nach der Installation hat. Das Standardpaßwort ist *„free“*. Das Paßwort sollte als erstes über die Administrationsseite des VirLib-Servers verändert werden.

   ACCESS: Zugriffsrecht für das Verzeichnis des VirLib-Servers. Der Wert hängt von den gewählten Sicherheitsvorkehrungen ab.

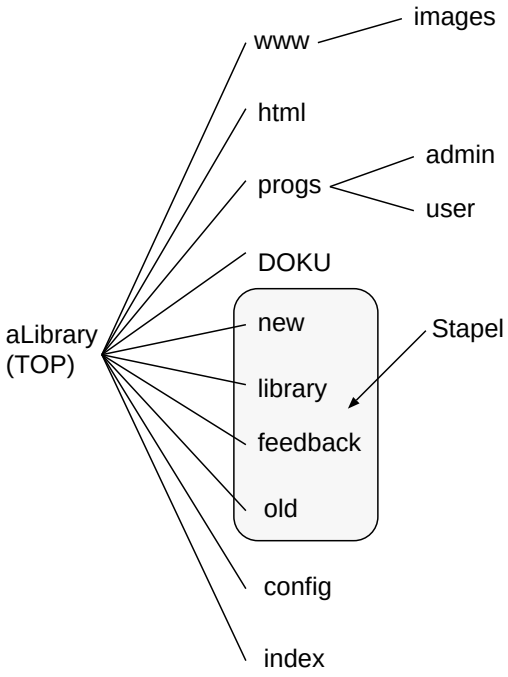

Abbildung 6.2: Verzeichnisstruktur der Bibliothek

5. `make install`.

6. Die HTML-Seiten in `./www` und `./html`, wie im Abschnitt *Anpassen der Bibliothek (6.2.2)* beschrieben, verändern.

Jetzt ist die Bibliothek bereit. Die Einstellungen im Makefile können jederzeit verändert werden. Durch `make install` übernimmt die Bibliothek die neuen Einstellungen ohne einen Einfluß auf den Inhalt der Bibliothek zu nehmen.

Der URL zur Homepage der Bibliothek setzt sich aus dem Basis-URL (festgelegt durch `WWWHOME` im Makefile) und dem Seitennamen `virlib.html` zusammen. Somit ergibt sich zum Beispiel der URL:
`http://miss.wu-wien.ac.at/~virlib/aLibrary/virlib.html`
Außerdem sollte man von dieser Datei einen symbolischen Link auf die vom WWW-Server automatisch aufgerufene Datei legen. Im Normalfall heißen diese Dateien `welcome.html`, `local.html` oder `index.html`. Dadurch gelangt man durch die Angabe des Basis-URL direkt auf die Homepage der Bibliothek.

6.2.1 Sicherheitsaspekte

In der Grundkonfiguration können alle Benutzer des Rechners auf die Verzeichnisse der Bibliothek zugreifen. Dies ist nötig, damit sichergestellt ist, daß der WWW-Server Zugriff hat. Es gibt mehrere Alternativen für dieses Sicherheitsproblem.

1. Keine besonderen Sicherheitsvorkehrungen durch den Befehl `ACCESS = 777` im Makefile. Diese Standardeinstellung ist nur für unsensible Informationen ratsam.

2. Den WWW-Server zum Eigentümer der Verzeichnisse der Bibliothek machen und die Zugriffsrechte nur auf ihn beschränken. Dies hat die Nachteile, daß dies in der Regel nur der Administrator des Rechners kann, und danach der Administrator der Bibliothek keinen direkten Zugriff auf die Verzeichnisse der Bibliothek hat. Es muß der Befehl `ACCESS = 700` im Makefile verwendet werden.

3. Es wird ein neuer Benutzer erstellt (oder ein schon vorhandener verwendet wie zum Beispiel `virlib`), der derselben Gruppe wie der WWW-Server angehört. Im Verzeichnis dieses Benutzers wird dann die Bibliothek installiert. Danach müssen die Zugriffsrechte auf den Eigentümer und dessen Gruppe beschränkt werden. Im Makefile wird der Befehl `ACCESS = 770` verwendet. Diese Methode ist zu empfehlen.

6.2.2 Anpassen der Bibliothek

Nachdem die Bibliothek installiert wurde, muß ihr Inhalt festgelegt werden. Dies ist der wichtigste Punkt beim Erstellen einer neuen Bibliothek. Er besteht aus einer genauen Analyse der benötigten Informationen und der Anpassung einiger HTML-Seiten. Die Beschreibung beschränkt sich nur auf die Anpassung der Seiten.

Ist dies geschehen, muß noch das Design der Bibliothek an das *Corporate Design* oder ein, speziell auf die Inhalte der Bibliothek abgestimmtes Design angepaßt werden. Wichtig ist, daß alle Seiten ein durchgängiges Design besitzen, um den Benutzer nicht zu verwirren. Dies wird am einfachsten durch die automatische Anpassung des Designs erreicht.

Anpassen des Inhalts

Um festzulegen, welche Informationen die Bibliothek beinhalten soll, muß man folgende Schritte ausführen:

1. Analyse der benötigten Informationen.

2. Erstellen einer Eingabemaske, die alle benötigten Informationen beinhaltet.

3. Einfügen der Feldnamen aus der Eingabemaske in einige HTML-Seiten.

Der Aufbau der Eingabemaske. Die Eingabemaske besteht aus Feldern eines HTML-Formulars. Es können folgende Eingabefeld-Typen verwendet werden:

```
  <INPUT TYPE="text" NAME="Titel">
  <INPUT TYPE="checkbox" NAME="Material" VA-
LUE="Unterlagen">
  <INPUT TYPE="radio" NAME="Abschnitt" VALUE="1.">
  <TEXTAREA NAME="Beschreibung"> </TEXTAREA>
  <SELECT NAME="Kategorie"><OPTION> ... </SELECT>
```

Wichtig ist, daß die Feldnamen *cue, data, date, file, key, plpath, pwd, type* und *wwwurl* nicht verwendet werden dürfen, da sie für Funktionen des VirLib-Servers reserviert sind. Außerdem darf das Schlüsselwort `checked` in der Maske nicht vorkommen, da es auf allen Seiten außer `input.html` zu Fehlfunktionen führt.

Beispiel: Kurze Eingabemaske

```
<!-- cut_here begin -->
<i>URL der Information:</i><br>
<INPUT NAME="URL" TYPE="text"><br>

<i>Bezeichnung:</i><br>
<INPUT NAME="Bezeichnung" TYPE="text"><br>

<i>Kategorie:</i><br>
<SELECT NAME="Kategorie">
<OPTION>Beruf
<OPTION>Ausbildung
<OPTION>Freizeit
</SELECT>
<!-- cut_here end -->
```

In der Maske fehlen die für ein vollständiges HTML-Formular nötigen Tags `<FORM ...>` und `</FORM>`, sowie der Submit-Button. Diese werden nicht benötigt, da sie auf der Seite, in die die Eingabemaske eingefügt wird, schon vorhanden sind.

Jetzt zur Bedeutung der einzelnen Eingabefelder. Das Feld mit dem Namen Bezeichnung bewirkt, daß die Eingabe später als Variable `{$Bezeichnung}` in speziellen HTML-Seiten eingefügt werden kann. Mit dem Select-Feld wird eine Kategorisierung vorgenommen, wobei die Kategorien Beruf, Ausbildung und Freizeit verfügbar sind. Das Eingabefeld mit dem Namen URL ist reserviert und muß den URL zu den Informationen enthalten, da es später von den automatischen Diensten des VirLib-Servers verwendet wird.

Anpassung der Seiten Nachdem die Eingabemaske fertig ist, müssen folgende Seiten angepaßt werden.

1. `./www/input.html`
   Diese Seite wird vom Endbenutzer dazu verwendet neue Einträge zu erstellen. Darum muß die gesamte Eingabemaske zwischen `<!-- cut_here begin -->` und `<!-- cut_here end -->` eingefügt werden.

2. `./html/review.html`
   Diese Seite stellt den Inhalt eines Eintrags editierbar für den Administrator dar. Auch hier muß die gesamte Eingabemaske zwischen
   `<!-- cut_here begin -->` und `<!-- cut_here end -->` eingefügt werden. Der VirLib-Server füllt diese Seite bei Aufruf mit dem Inhalt des ausgewählten Eintrags auf. Falls erwünscht können auch der URL oder andere Links in die Maske eingebunden werden. Dies bietet den Vorteil, daß mit einem Klick der URL überprüft werden kann.

   Beispiel: Kurze Eingabemaske mit Link für die Seite `review.html`

```
<!-- cut_here begin -->
<a href="http://{$URL}">
<i>URL der Information:</i></a><br>
<INPUT NAME="URL" TYPE="text"><br>

<i>Bezeichnung:</i><br>
<INPUT NAME="Bezeichnung" TYPE="text"><br>

<i>Kategorie:</i><br>
<SELECT NAME="Kategorie">
<OPTION>Beruf
<OPTION>Ausbildung
<OPTION>Freizeit
</SELECT>
<!-- cut_here end -->
```

3. `./html/entry.html`
   Diese Seite stellt den Inhalt eines Eintrags dar, wie er dem Endbenutzer präsentiert wird. Es werden im HTML-Text Feldnamen aus der Eingabemaske in der Form `{$Feldnamen}` verwendet.

   Beispiel: Ausschnitt aus `entry.html`

```
<!-- change_here begin -->
<b>Informationen:</b><p>
```

```
<i>Bezeichnung:</i> <a href="http://{$URL}">
{$Bezeichnung}</a><br>
<i>URL der Information:</i> {$URL}<p>
<i>Kategorie:</i> {$Kategorie}<br>
<!-- change_here end -->
```

In diesem Beispiel wird die Bezeichnung der Information mit dem Link darauf unterlegt. Danach werden der URL und die Kategorie angezeigt.

4. `./html/result.html` und `./html/result.line`
   Dies ist eine Liste. Listen bestehen aus einer Grund-Seite mit der Endung `.html` und einer Datei, die den Aufbau einer Zeile der Liste festlegt. Sie hat die Endung `.line`.

   In diesem Fall wird das Ergebnis einer Abfrage durch den Endbenutzer dargestellt. Die gewünschten Feldnamen aus der Eingabemaske werden in `result.line` verwendet, damit sie der VirLib-Server bei einer Abfrage durch die Inhalte der Einträge ersetzt. In `result.html` muß die Bedeutung der Feldnamen zwischen `<!-- change_here begin -->` und `<!- - change_here end -->` eingetragen werden.

   Beispiel: Ausschnitt aus `result.html`

```
<!-- change_here begin -->
<tr>
<td><b>Bezeichnung</b></td>
<td><b>Kategorie</b></td>
<td></td>
</tr>
<!-- change_here end -->
```

   Zur Ausgabe der Liste wird eine Tabelle verwendet. In der ersten Spalte steht die Bezeichnung der Information in der zweiten, die Kategorie der sie zugeordnet ist. Die dritte Spalte wird für einen Link auf die Kurzbeschreibung des Eintrags leer gelassen. Die Zeilen der Tabelle werden bei Bedarf vom VirLib-Server aus der Datei `result.line` erzeugt und eingefügt.

   Beispiel: `result.line`

```
<tr>
<td><A HREF="http://{$URL}">
{$Bezeichnung}</a></td>
<td>{$Kategorie}</td>
<td><A HREF="{$stuffh}showentry">Kurzbeschreibung</a></td>
</tr>
```

Die Bezeichnung aus dem Eintrag ist mit einem Link auf die Informationen unterlegt. Die Zeile
`<A HREF="{$stuffh}showentry">Kurzbeschreibung</a>`
in der letzten Spalte beinhaltet den Befehl, den Inhalt dieses Eintrags durch die Seite `entry.html` darzustellen.

5. `./html/resultadmin.html` und `./html/resultadmin.line`
   Die Liste für das Suchergebnis des Administrators. Enthält in etwa dieselben Variablen wie die vorher beschriebene `result`-Liste. Der Unterschied besteht darin, daß in `resultadmin.line` nicht der Befehl *showentry* zur Darstellung des Eintrags verwendet wird, sondern *review* um den Eintrag zu editieren. Außerdem sollte man unbedingt die Dateinummer des Eintrags (`{$file}`) angeben und auch mit dem Befehl *review* unterlegen, da bei der Eingabe eines neuen Eintrags nicht überprüft wird, ob dieser eine Bezeichnung hat. Eine leere Bezeichnung führt dazu, daß der vorgesehene Link fehlt.

   Beispiel: `resultadmin.line`

```
<tr><td><A HREF="{$stuffh}review-{$cue}">
{$file}</a></td>
<td><A HREF="{$stuffh}review-{$cue}">
{$Bezeichnung}</a></td>
<td>{$date}</td>
<td>{$condition}</td>
</tr>
```

   In diesem Beispiel werden in der ersten Spalte die vorher erwähnten Dateinummern angezeigt und mit Link unterlegt. Auch die zweite Spalte ist unterlegt und enthält die Bezeichnungen aus den Einträgen. Die letzten zwei Spalten enthalten Variablen, die automatisch jedem Eintrag hinzugefügt werden. Es handelt sich um das Datum *(date)*, an dem der Eintrag erstellt wurde und dessen Zustand *(condition)*. Der Zustand ist im Zusammenhang mit der Verwendung der automatischen Dienste wichtig.

6. `./html/admin.html`
   Dies ist die Administrationsseite. Hier muß festgelegt werden, nach dem Inhalt welchen Feldnamens die Ausgabe sortiert werden soll. Der Befehl *doindex* erstellt einen Index für die Suche nach Einträgen in der Bibliothek, wobei der Parameter *sortby* den Feldnamen für die Sortierung angibt. Es können auch mehrere Feldnamen, durch Beistrich getrennt, angegeben werden. Dadurch kann die Ausgabe nach dem Inhalt eines dieser Felder sortiert erfolgen.

   Beispiel: Befehl *dosort* aus `admin.html`

```
<!-- change_here begin -->
```

```
<A HREF="{$stuffh}doindex&sortby=Bezeichnung,Kategorie">
Bibliothek sortieren</A><br>
<!-- change_here end -->
```

In diesem Beispiel wird eine Sortierungsdatei für die Felder *Bezeichnung* und *Kategorie* angelegt. Dadurch können die Einträge alphabetisch nach dem Inhalt dieser Felder sortiert ausgegeben werden. Ist die Sortierungsdatei nicht aktuell, führt die sortierte Ausgabe zu einem unvollständigen Ergebnis.

7. `./www/virlib.html`
   Diese Seite ist die Homepage der Bibliothek. Sie beinhaltet das Suchinterface und die Links zu der Seite für die Eingabe eines neuen Beitrags und dem Feedback. Im Suchinterface muß für den Parameter *sortby* ein Feldname angegeben werden, nach dem in `admin.html` sortiert wird, da die Suche sonst immer zu einem leeren Ergebnis führen würde.

   Auf der Homepage sollte man sehr kurz den Inhalt und den Zweck der Bibliothek beschreiben.

Anpassung des Designs

Das Design muß auf allen Seiten die der Endbenutzer verwendet, einheitlich angepaßt werden. Diese Seiten sind:

Im Verzeichnis: `./www`
`virlib.html, input.html, feedback.html`

Im Verzeichnis: `./html`
`entry.html, result.html, noresult.html, suboknew.html, suboklibrary.html, error.html`

Beim Verändern der HTML-Seiten muß folgendes berücksichtigt werden:

- Die Grafiken müssen für den WWW-Server erreichbar sein (zum Beispiel im Verzeichnis `./www/images`).
- Alle Links in den dynamisch erzeugten Seiten (Seiten in `./html`) müssen absolute URLs enthalten, das heißt mit `http://` beginnen. Dies gilt auch für Grafiken.
- Beim Editieren der Seiten dürfen die Zeilen `<!-- ... begin -->` und `<!-- ... end -->` nicht verändert werden, da diese für die automatische Anpassung des Designs benötigt werden.

Die automatische Anpassung des Designs. Die Veränderung des Kopfes und des Fußes auf allen HTML-Seiten ist automatisch möglich. Dazu sind folgende Schritte nötig:

1. Anpassen der Design-Dateien in `./html` (`head.design`, `foot.design` bzw. `headadmin.design` und `footadmin.design`).
   Beispiel: `head.design`

   ```
   <!-- head begin -->
   <HEAD>
           <TITLE>The Virtual Library</TITLE>
   </HEAD>
   <BODY>
   <CENTER>
           <H2>The Virtual Library - aLibrary</H2>
   </CENTER>
   <hr>
   <!-- head end -->
   ```

   Beispiel: `foot.design`

   ```
   <!-- foot begin -->
   <HR>
   <img src=
   "http://miss.wu-wien.ac.at/~virlib/aLibrary/images/book.gif"
   align="left">
   (c) 1997
   <A HREF="http://aie.wu-wien.ac.at/~hahsler">
   Michael Hahsler</A>,<br>
   <A HREF ="http://wwwai.wu-wien.ac.at/">
   Abteilung f&uuml;r Angewandte Informatik</A>,
   <A HREF="http://www.wu-wien.ac.at/home.html">WU-
   Wien</A><br>
   <!-- foot end -->
   ```

   Die beiden Admin-Dateien enthalten das Design für die Administrationsseiten. Die Trennung wurde hier vorgenommen, da diese Seiten über SSL aufgerufen werden, und es dadurch zu Problemen bei Grafiken kommen kann.

2. `make design`

Dadurch wurden automatisch in allen Seiten, die im Makefile angegeben sind, der Kopf und der Fuß ersetzt. Das Design kann jederzeit ohne Einfluß auf die Inhalte der Bibliothek verändert werden. Natürlich können die HTML-Seiten, falls notwendig, noch per Hand nachbearbeitet werden.

Kapitel 7

Befehlssatz des VirLib-Servers

Dieser Befehlssatz kann beim Verändern der HTML-Seiten und des Ablaufs verwendet werden. Funktionen, die nicht durch vorhandene Befehle ausgeführt werden können, müssen als Skripts (siehe Kapitel 8) der Bibliothek hinzugefügt werden.

7.1 Aufbau der Befehle

Der Client (WWW-Browser) kommuniziert mit dem VirLib-Server (WWW-Server) durch Befehle und deren Parameter, die in den HTML-Seiten enthalten sind. Diese Befehle bestimmen die nächste Aktion des VirLib-Servers.

Da die Befehle in HTML-Seiten eingebettet sind, müssen sie entweder als *HTML-Formular* oder *URL* angegeben werden. Mit jedem Befehl müssen die Parameter *pathvl* und *wwwurl* übergeben werden, da der Virlib-Server daran erkennt, welche Bibliothek angesprochen werden soll.

7.1.1 Befehle als HTML-Formular

Die nötigen Parameter für den Befehl müssen versteckt bzw. als Eingabefelder definiert werden. Unterschieden werden Formulare auf statischen Seiten und Formulare auf dynamisch erzeugten Seiten, da der VirLib-Server nur in den letzteren Variablen ersetzen kann.

Statische Seiten. Diese Seiten befinden sich im Verzeichnis `./www`. Das Feld *Action* im *Form-Tag* weist auf das CGI-Skript des VirLib-Servers. Dieser URL wird automatisch bei der Installation angepaßt. Dies muß bei Formularen, die nicht an den VirLib-Server gerichtet sind, beachtet werden, da diese nach jedem Aufruf von `make install` per Hand korrigiert werden müssen.

Beispiel: HTML-Formular für den Befehl `doquery` in einer statischen Seite.

```
<!-- query begin -->
<FORM METHOD="POST"
        ACTION="http://miss.wu-wien.ac.at/USERCGI/virlib.pl">
<input type="hidden" name="type" value="doquery">
<input type="hidden" name="pathvl"
        value="/net/logins/virlib/aLibrary/">
<input type="hidden" name="wwwurl"
        value="http://miss.wu-wien.ac.at/~virlib/aLibrary/">
<input type="hidden" name="sortby" value="Bezeichnung">
<b>Suchbegriffe:</b><br>
<INPUT NAME="query" TYPE="text" SIZE="40">
<input type="hidden" name="qflags" value="-1">
<INPUT TYPE="submit" VALUE="   suchen...   ">
</FORM>
<!-- query end -->
```

Das Feld *type* enthält den Befehl `doquery`. Dieser benötigt die Parameter *query, qflags* und *sortby*. Außerdem müssen *pathvl* und *wwwurl* mit jedem Befehl übergeben werden.

Dynamische Seiten Diese Seiten befinden sich im Verzeichnis `./html`. In dynamischen Seiten kann der VirLib-Server Variablen ersetzen. Die Variable `{$plhome}` enthält den URL des CGI-Skripts des VirLib-Servers. Und die Variable `{$stuff}` ersetzt der VirLib-Server mit den versteckten Feldern für die Parameter *pwd, pathvl, wwwurl* und *file*, wobei die nicht benötigten Parameter ignoriert werden.

Beispiel: HTML-Formular für den Befehl `doquery` in einer dynamischen Seite.

```
<!-- query begin -->
<FORM METHOD="POST" ACTION="{$plhome}">
{$stuff}
<input type="hidden" name="type" value="query">
<input type="hidden" name="sortby" value="Bezeichnung">

<b>Suchbegriffe:</b>
<INPUT NAME="query" TYPE="text" SIZE="40">
<input type="hidden" name="qflags" value="-1">
<INPUT TYPE="submit" VALUE="   suchen...   ">
</FORM>
<!-- query end -->
```

7.1.2 Befehle als URL

Der verwendete URL hat denselben Effekt wie ein HTML-Formular, das mit der Methode *GET* arbeitet. Auch hier wird zwischen statischen und dynamisch generierten Seiten unterschieden.

Statische Seiten. Es müssen alle nötigen Parameter angegeben werden. Wichtig ist dabei die Kodierung von Sonderzeichen mit Escapesequenzen nach dem *Http-Standard* [Berners-Lee, 1994]. Die Kodierung erfolgt durch das Escapezeichen % gefolgt durch den ASCII-Wert des Sonderzeichens in hexadezimaler Darstellung. Beispielsweise wird aus `/` die Sequenz `%2F`.

Beispiel: URL für den Befehl `doquery` in einer statischen Seite.

```
<!-- query begin -->
<A HREF="http://miss.wu-wien.ac.at/USERCGI/virlib.pl?
type=doquery&
pathvl=%2Fnet%2Flogins%2Fvirlib%2aLibrary%2F&
wwwurl=http:%2F%2Fmiss.wu-wien.ac.at%2F~virlib%2FaLibrary%2F&
query=Kategorie:='Freizeit'&
sortby=Bezeichnung&
qflags=-1">Freizeit</A></td>
<!-- query end -->
```

Dynamische Seiten. Die Variable `{$stuffh}` erfüllt die gleiche Funktion wie `{$stuff}` in den HTML-Formularen. Der Unterschied ist, daß sie auch den URL für das CGI-Skript des VirLib-Servers enthält, und der Befehl direkt an diese Variable angefügt werden muß. Dadurch lassen sich die Befehlsaufrufe, wie das folgende Beispiel zeigt, sehr kompakt formulieren.

Beispiel: URL für den Befehl `doquery` in einer dynamischen Seite.

```
<!-- query begin -->
<A HREF="{$stuffh}query&
query=Kategorie:='Freizeit'&
sortby=Bezeichnung&
qflags=-1">Freizeit</A>
<!-- query end -->
```

7.2 Die Parameter

Die Befehle des VirLib-Servers benötigen meist zusätzliche Informationen. Diese werden als Befehlsparameter übergeben.

Regelmäßig verwendete Parameter sind:

cue: Stapel auf den der Befehl angewendet wird. Um die Befehle kürzer und klarer zu formulieren, wird dieser Parameter in der Form `Befehl-cue` direkt an den Befehl angehängt.
Beispiel: `getlist-library`
Die Stapel der Bibliothek sind *library*, *feedback*, *new* und *old*.

file: Dateinummer des betroffenen Eintrages. Diese Nummer identifiziert jeden Eintrag in einem Stapel eindeutig.

pwd: Paßwort.

qflags: Einstellungen für die Suchmaschine, die im Zusammenhang mit *query* verwendet werden. Beispielsweise gibt `-0`, `-1` usw. die Anzahl der erlaubten Fehler an. Durch `-i` werden bei der Suche Klein- und Großbuchstaben nicht unterschieden. Mehrere Einstellungen können durch ein Leerzeichen getrennt, hintereinander angegeben werden. Eine Beschreibung aller Einstellungsmöglichkeiten findet man in der Dokumentation von GLIMPSE [Manber and Wu, 1994].

Wird dieser Parameter nicht angegeben, werden die Standardeinstellungen `-1 -W -i` (= 1 Fehler, ganzer Eintrag als Bereich für boolesche Verknüpfungen und keine Unterscheidung von Groß- und Kleinbuchstaben) verwendet.

query: Abfragestring. Die logische Verknüpfungen *AND* und *OR*, sowie das Setzen von Klammern *()* werden unterstützt. Außerdem sind alle Funktionen von GLIMPSE, wie zum Beispiel die Platzhalter *?* und * verwendbar.

Zusätzlich ist auch eine strukturierte Abfrage möglich. Dazu muß der Abfragestring den Aufbau `Kategorie := Wert` haben. Wobei *Kategorie* die gewünschte Variable in den Einträgen bezeichnet und *Wert* deren Inhalt. Für die strukturierte Abfrage gelten einige Einschränkungen. So funktioniert sie nicht mit Variablen, die aus einem Feld vom Typ *Textarea* stammen, und in strukturierten Abfragen können derzeit keine booleschen Operatoren verwendet werden.

script: Programmname eines Skripts.

sortby: Sortiert die Ausgabe alphabetisch nach dem Inhalt des hier angegebenen Feldes. Voraussetzung dafür ist, daß eine Sortierungsdatei für dieses Feld erstellt wurde.
Beim Erstellen der Sortierungsdatei können mit diesem Parameter mehrere Feldnamen, durch Beistrich getrennt, angegeben werden. Danach ist eine Sortierung der Ausgabe nach diesen Feldern möglich.

Spezielle Parameter, die für den Aufbau *jedes* Befehls benötigt werden, sind:

type: Dieser Parameter enthält den Befehl selbst.

pathvl: Durch diesen Parameter weiß der VirLib-Server, welche Bibliothek angesprochen werden soll. Es handelt sich um den Pfad zum Verzeichnis der Bibliothek.

wwwurl: Enthält den URL, auf den von den folgenden dynamisch erzeugten Seiten zurückgesprungen werden kann. Normalerweise enthält er den Basis-URL der Bibliothek.

Beispiel: URL für den Rücksprung in einer dynamischen Seite

```
<a href="http://{$wwwurl}>Home</a>
```

7.3 Die Befehle

In diesem Abschnitt werden alle verfügbaren Befehle des VirLib-Servers vorgestellt. Sie werden nach ihren Benutzern in vier Gruppen eingeteilt. Diese Gruppen sind Benutzerbefehle, Befehle für Informationsanbieter, Befehle für Bibliothekare und Administrationsbefehle.

In den folgenden Aufstellungen wird für jeden Befehl kurz die Funktion beschrieben. Danach werden die benötigten Parameter aufgezählt, wobei optionale Parameter mir (o) gekennzeichnet sind. Die Parameter *pathvl* und *wwwurl* müssen mit jedem Befehl übergeben werden, und sind in der Aufstellung nicht extra angegeben. Zum Schluß wird die HTML-Seite angegeben, die auf den Befehl folgt.

7.3.1 Befehle für Benutzer

Die folgenden Befehle werden ohne Kontrolle des Administrationspaßworts ausgeführt. Sie stellen die Verbindung zwischen Benutzer und Bibliothek her.

doquery: Führt eine Bibliotheksabfrage durch.
Parameter: query, qflags(o), sortby(o).
Folgeseite: result.html /result.line

getlist: Gibt alle Einträge eines Stapels aus.
Parameter: cue, sortby(o).
Folgeseite: result.html / result.line

script: Führt ein Benutzer-Skript aus, das kein Administratorpaßwort benötigt. Das Skript muß sich in `./progs/usr` befinden.
Parameter: script.
Folgeseite: ok.html

showentry: Zeigt den Inhalt eines Eintrags an.
Parameter: cue, file.
Folgeseite: entry.html

7.3.2 Befehle für Informationsanbieter

Der folgende Befehl wird hauptsächlich von Informationsanbietern benutzt, um neue Einträge an die Bibliothek zu übergeben. Außerdem wird er auch noch für die Sammlung von Rückmeldungen durch Benutzer und Informationsanbietern für den Bibliothekar verwendet.

newentry: Übergibt den Inhalt eines Formulars an die Bibliothek und fügt ihn dem Stapel *cue* hinzu. Neue Beiträge der Informationsanbieter werden im Stapel *new* gesammelt. Der Stapel *library* kann nicht direkt verändern werden, da der Bibliothekar neue Beiträge zuerst überprüfen muß. Dies kann nur durch ein Skript in der Bibliothek umgangen werden.
Parameter: cue, die Variablen des Eintrags.
Folgeseite: subokcue.html

7.3.3 Befehle für Bibliothekare

Alle Seiten benötigen das Paßwort des Administrators als Parameter *pwd*. Die Seiten für die Administration werden dynamisch generiert, daher können die Parameter *pwd*, *pathvl*, *wwwurl* und falls nötig, auch *file* durch die Variable `{$stuff}` bzw. `{$stuffh}` ersetzt werden.

addauto: Meldet die Bibliothek beim VirLib-Server für die automatischen Dienste an.
Parameter: pwd.
Folgeseite: testauto.html

adminlist: Gibt alle Einträge eines Stapels für den Administrator aus.
Parameter: pwd, cue.
Folgeseite: adminresult.html / adminresult.line

adminquery: Führt eine Abfrage der Bibliothek für den Administrator durch.
Parameter: pwd, query, qflags(o), sortby(o).
Folgeseite: adminresult.html /adminresult.line

adminscript: Führt ein Admin-Skript aus. Es muß sich im Verzeichnis `./progs/admin` befinden.
Parameter: pwd, script.
Folgeseite: okadmin.html

changeit: Ersetzt den Inhalt eines Eintrags durch die mit diesem Befehl übertragenen Variablen.
Parameter: pwd, cue, file, Variablen des Eintrags.
Folgeseite: okadmin.html

doindex: Startet die Indexierung für den Stapel *library* und erstellt die Sortierungsdatei.
Parameter: pwd, sortby.
Folgeseite: okadmin.html

enterit: Verschiebt einen Eintrag zum Stapel *Bibliothek (library)*.
Parameter: pwd, cue, file.
Folgeseite: okadmin.html

getadmin: Fordert die Administrationsseite an. Diese Seite enthält die Anzahl der in den Stapeln befindlichen Einträge. Soll ein zusätzlicher Stapel verwendet werden, wird er als Parameter `cuetop` angegeben.
Parameter: pwd, cuetop(o).
Folgeseite: admin.html

killit: Löscht einen Eintrag. Einträge im Stapel *Mistkübel (old)* und *feedback* werden endgültig gelöscht, alle anderen werden nach *Mistkübel (old)* verschoben.
Parameter: pwd, cue, file.
Folgeseite: okadmin.html

killlist: Löscht alle Einträge eines Stapels, das heißt alle Dateien im Stapelverzeichnis werden gelöscht.
Parameter: pwd, cue.
Folgeseite: okadmin.html

moveit: Verschiebt einen Eintrag zum durch `to` festgelegten Stapel.
Parameter: pwd, cue, file, to.
Folgeseite: okadmin.html

pwdnewget: Fordert die Seite zur Eingabe eines neuen Paßworts an.
Parameter: pwd.
Folgeseite: newpwd.html

pwdnewput: Setzt ein neues Administratorpaßwort, wobei der Parameter *pwdold* nochmals das alte Paßwort enthalten muß. Die Parameter *pwdnew1* und *pwdnew2* enthalten das neue Paßwort zweimal, um Tippfehler auszuschließen.
Parameter: pwd, pwdold, pwdnew1, pwdnew2.
Folgeseite: okadmin.html

removeauto: Meldet die Bibliothek vom VirLib-Server für die automatischen Dienste ab.

Parameter: pwd.
Folgeseite: testauto.html

review: Stellt den Inhalt eines Eintrags editierbar dar.
Parameter: pwd, cue, file.
Folgeseite: review.html

saveautoconfig: Speichert die Konfigurationsdatei für die automatischen Dienste des VirLib-Servers. Alle Parameter außer *autokillat*, *sortby* und *script* sind aktiv, falls ihr Wert „*yes*“ ist, wobei *autokillat* festlegt, nach wievielen Tagen Unerreichbarkeit, ein Eintrag gelöscht werden soll.
Parameter: checkURL, autokill, autokillat, index, sort, sortby, script.
Folgeseite: testauto.html

testauto: Zeigt die Seite für die Einstellungen der automatischen Dienste an.
Parameter: pwd.
Folgeseite: testauto.html

7.3.4 Befehle für den Administrator des VirLib-Servers

Der Aufruf aller dieser Funktionen benötigt als Parameter `pwd` das Paßwort des Administrators des VirLib-Servers. Diese Funktionen dienen zur Verwaltung der automatischen Dienste.

changeautolist: Änderungen der Liste der angemeldeten Bibliotheken. Die Parameter *add* und *kill* enthalten die Pfade der Bibliotheken, die der Liste hinzugefügt bzw. entfernt werden sollen.
Parameter: pwd, add(o), kill(o).
Folgeseite: listauto.html / listauto.line

doat: Zeigt die Administrationsseite für das Hilfsprogramm *at* an. Dieses Programm startet automatisch zu einer festgelegten Zeit die automatischen Dienste.
Parameter: pwd.
Folgeseite: at.html

killat: Löscht den automatischen Job, der durch den Parameter `jobnumber` bezeichnet wird.
Parameter: pwd, jobnumber.
Folgeseite: at.html

showautolist: Zeigt die Liste aller für die automatischen Dienste angemeldeten Bibliotheken an.
Parameter: pwd.
Folgeseite: listauto.html / listauto.line

startat: Fügt einen automatischen Job hinzu. Dieser führt automatisch, zum Zeitpunkt der in *starttime* angegeben wird, die automatischen Dienste aus.
Parameter: pwd, starttime.
Folgeseite: at.html

viewlog: Zeigt das Ergebnis der letzten Durchführung der automatischen Dienste an, das sich in der Datei `./config/log` befindet.
Parameter: pwd.
Folgeseite: okadmin.html

Kapitel 8

Skripts

Es besteht die Möglichkeit, daß jede Bibliothek eigene Skripts verwendet, um ihren Funktionsumfang zu erweitern. Einige Skripts mit Installationsanleitung befinden sich auf der Homepage der Virtual Library.

Eigene Skripts müssen in Perl programmiert werden, da sie zur Laufzeit in das Programm des VirLib-Servers eingebunden werden. Dies bedingt, daß bestimmte Konventionen eingehalten werden müssen.

8.1 Konventionen für das Erstellen von Skripts

Skripts werden in den Verzeichnissen `./progs/usr` bzw. `./progs/admin` abgelegt. Dadurch wird festgelegt, ob zur Ausführung des Skripts ein Paßwort benötigt wird, oder ob es von jedem benutzt werden kann.

Die Parameter werden an das Skript als Array *%in* übergeben. Es beinhaltet alle mit dem Aufruf übergebenen Feldnamen und deren Wert aus der HTML-Seite.

Da das Skript mit dem Perl-Befehl *require* in das Programm des VirLib-Servers eingebunden wird, können die Funktionen der Programmdateien `fileIO.pl` und `httpIO.pl` verwendet werden. Ihre Arbeitsweise ist aus dem Programmcode ersichtlich.

Das *Ergebnis*, das das Skript durch *return* zurückgibt, wird durch die Seite `okadmin.html` dargestellt. Es besteht auch die Möglichkeit die Ausgabe dieser Seite zu unterdrücken. Dazu muß der Wert `!no output` zurückgegeben werden. Die gewünschte Ausgabe erfolgt dann durch das Skript über den *Standard Output*. Dazu können die Funktionen von `httpIO.html` verwendet werden.

Beispiel: Pseudocode für den Rumpf eines Skripts

#!/usr/bin/perl

```
sub StartScript {
    local (%in) =@_;
    local (<verwendete Variablen>);

        <Programmcode>

    return <Ergebnis>;
}
1;
```

8.2 Beispielskript directin.pl

Dieses Skript ermöglicht dem Informationsanbieter seinen Eintrag direkt in den Stapel `library` zu schreiben, ohne den Umweg über neue Beiträge (Stapel `new`) zu nehmen. Außerdem ist es dadurch möglich, den Eintrag sofort nach der Eingabe zu verändern, zu überprüfen, oder dem Informationsanbieter eine individuelle Rückmeldung zu schicken.

Perlcode des Skripts directin.pl:

```
#!/usr/bin/perl

##########################################################################
#
# The Virtual Library
#
# Script: directin.pl
# Date: 4.8.1997
#
# Description: Overrides newentry and appends an entry to
# stack 'library'
#

sub StartScript {
    local (%in) = @_;
    local ($to);

    delete $in{script};

    $to="library";

#
# Actions that should be done!
# e.g. security measures, conversions ...
#

    &NewEntry($to,%in);
```

```
    return "!no output";    # suppress output
}
1;
```

Anhang A

Glossar

at: Dienstprogramm, das Programme zu einem festgelegten Zeitpunkt automatisch startet. Es wird für die automatischen Dienste verwendet.

Basis URL: Der URL unter dem die HTML-Seiten der Bibliothek erreichbar sind. Er wird im Makefile durch die Variable `wwwhome` festgelegt.

CGI-Skript: Ein Programm, das vom WWW-Server ausgeführt wird und mit diesem über eine festgelegte Schnittstelle (das *Common Gateway Interface*) kommuniziert.

Client-Server-Architektur: Die Lösung einer Aufgabe ist zwischen mehreren Rechnern verteilt. Es gibt zwei Arten von Rechnern. Die *Server* bieten Dienste über das Netzwerk an und die *Clients* nutzen diese.

Cookbook: Eine Anleitung, die einen Ablauf schrittweise erklärt und mit Beispielen verdeutlicht.

Eingabemaske: Sie legt fest, welche Informationen in der Bibliothek gespeichert werden sollen und besteht aus Eingabefeldern eines HTML-Formulars.

Feld, Variable: Diese zwei Begriffe werden als Synonyme verwendet, da ein Eingabefeld in der Eingabemaske durch den Eintragungsprozeß zu einer Variable mit dem gleichen Namen im erzeugten Eintrag wird.

Make: Ein Tool für die Installation von Programmen.

GLIMPSE: Das hier verwendete Indexierungswerkzeug. Mehr darüber findet man in [Manber and Wu, 1994] und [Wu and Manber, 1992]. Die Homepage von GLIMPSE ist:
`http://glimpse.cs.arizona.edu/`

HTML: *Hypertext Markup Language* ist die Formatierungssprache für verteilten Hypertext im WWW.

Intelligenter Agent: Als Intelligenter Agent wird alles bezeichnet, das seine Umwelt duch Sensoren wahrnimmt und diese Umwelt beeinflussen kann [Russell and Norvig, 1995].

Intranet: Internes Rechnernetz, in dem auf Internet-Technologie basierende Dienste verwendet werden.

Parameter: Informationen, die mit einem Befehl an den VirLib-Server übergeben werden.

Pfadangaben: Die Angaben der Pfade in dieser Beschreibung haben immer die Form `./Pfad/Datei`. Der Punkt steht hier für das Verzeichnis in dem sich die Bibliothek bzw. der VirLib-Server befindet. Dieses Verzeichnis wird im Makefile durch die Variable `top` festgelegt.

Stapel: Eine Organisationsform für Informationen. Bei der virtuellen Bibliothek werden die Einträge in verschiedenen Stapeln gespeichert.

Statische und dynamisch generierte HTML-Seiten: Statische Seiten sind vollständig und werden unverändert an den Browser weitergeleitet. Solche Seiten befinden sich im Verzeichnis `./www`. Dynamisch generierte Seiten werden erst nach der Anforderung durch den Browser fertiggestellt, da sie aktuelle Informationen enthalten. Der Aufbau dieser Seiten wird durch die Dateien im Verzeichnis `./html` festgelegt.

Symbolische Links: Werden in Unix-Systemen angeboten. Sie stellen eine Verknüpfung im Dateisystem dar. Dadurch wird ein Verzeichnis oder eine Datei über mehrere Pfade erreichbar. Symbolische Links werden von der virtuellen Bibliothek für das CGI-Skript und das Verzeichnis der statischen HTML-Seiten verwendet.

Thesaurus: Ein Thesaurus ist eine verbindliche Schlagwortliste. Schlagwörter aus einem Thesaurus eignen sich gut für die Indexierung von Dokumenten.

WWW: Das *World Wide Web* ist der bekannteste Internet-Dienst. Es wird in HTML formatierter Hypertext von WWW-Servern an Browser übertragen und dort dargestellt.

Anhang B

Aufbau der Befehle in BNF

Die Befehle für die Bibliothek sind in Form eines Verweises aufgebaut, da sie in HTML-Seiten verwendet werden. Der gesamte Verweis stellt eine Nachricht (`<library messages>`) an die Steuereinheit dar. Die Steuereinheit wird durch `<Object identifier>` bezeichnet. Er beinhaltet den URL für das CGI-Skript der Steuereinheit. Der Text von dem der Verweis ausgeht wird in `<Linkname>` angegeben.

Spezifikation in Backus-Naur Form:

<library messages> :=
'<A HREF="' <Object identifier> '?type='
<commands> <mandatory args> '">' <Linkname> '</A>';

<commands> :=
<user commands> | <info−provider commands> |
<librarian commands> <password> |
<admin. commands> <password>;

<user commands> :=
'doquery' <querystring> [<queryflags>] [<sort output by>] |
'getlist' <stack> [<sort output by>] |
'script' <programname> |
'showentry' <stack> <entry number>;

<info−provider commands> :=
'newentry' <stack> <information>;

<librarian commands> :=
'addauto' |
'adminlist' <stack> |
'adminquery' <querystring> [<queryflags>] [<sort output by>] |
'adminscript' <programname> |
'changeit' <stack> <entry number> <information> |
'doindex' [<sort output by>] |

```
        'enterit' <stack> <entry number> |
        'getadmin' [<additional stack>] |
        'killit' <stack> <entry number> |
        'killlist' <stack> |
        'moveit' <from stack> <entry number> <to stack> |
        'pwdnewget' |
        'pwdnewput' <old password> <new password>
                <new password retyped> |
        'removeauto' |
        'review' <stack> <entry number> |
        'saveautoconfig' <check the URL> <automatic kill>
                [<automatic kill at>] <do index> <do sort>
                [<sort output by>] [<programname>] |
        'testauto';

<admin. commands> :=
        'changeautolist' [<add library>] [<remove library>] |
        'doat' |
        'killat' <number of at−job> |
        'showautolist' |
        'startat' <time to start at> |
        'viewlog';

<mandatory args> :=
        <library identifier> <return link>;

<library identifier> :=
        '&pathvl=' <pathvl>;

<return link> :=
        '&wwwurl=' <wwwurl>;

<add library> :=
        '&add=' <add>;

<additional stack> :=
        '&cuetop=' <cue>;

<automatic kill> :=
        '&autokill=yes' |
        '&autokill=no';

<automatic kill at> :=
        '&autokillat=' <autokillat>;

<check the URL> :=
        '&checkURL=yes' |
        '&checkURL=no';

<entry number> :=
        '&file=' <file>:
```

```
<from stack> :=
        '&cue=' <cue>;

<do index> :=
        '&index=yes' |
        '&index=no';

<do sort> :=
        '&sort=yes' |
        '&sort=no';

<information> :=
        '&' <attribute> '=' <value> |
        <information> '&' <attribute> '=' <value>;

<kill library> :=
        '&kill=' <kill>;

<number of at-job> :=
        '&jobnumber=' <jobnumber>;

<new password> :=
        '&pwdnew1=' <pwd>;

<new password retyped> :=
        '&pwdnew2=' <pwd>;

<old password> :=
        '&pwdold=' <pwd>;

<password> :=
        '&pwd=' <pwd>;

<querystring> :=
        '&query=' ( <word> |
        <word> ( 'AND' | 'OR' ) <word> );

<queryflags> :=
        '&qflags=' <qflags>;

<programname> :=
        '&script=' <script>;

<sort output by> :=
        '&sortby=' <sortby>;

<make output sortable by> :=
        '&sortby=' <sortby>;

<stack> :=
```

```
        '&cue=' <cue>;

<time to start at> :=                                                    130
        '&starttime=' <starttime>;

<to stack> :=
        '&to=' <cue>;
```

Einige nicht-terminale Symbole stellen Platzhalter dar, die hier nicht weiter formal erklärt werden. Diese sind:

- Die folgenden Symbole.

  `<add>, <autokillat>, <cue>, <file>, <jobnumber>, <kill>, <pathvl>, <pwd>, <qflags>, <script>, <sortby>, <starttime>, <wwwurl>`

  Ihre Funktionen und Inhalte werden im Kapitel 7 erklärt.

- Das Symbol `<word>`. Es stellt einen Suchbegriff dar.

- Die Symbole `<attribute>` und `<value>`. Sie bezeichnen jeweils den Namen einer Variable und ihren Inhalt.

Anhang C

Programmcode der virtuellen Bibliothek

C.1 CGI-Skript virlib.pl

Steuereinheit der virtuellen Bibliothek. Dieses PERL-Programm wird vom WWW-Server als CGI-Skript aufgerufen. Die Kommunikation mit dem WWW-Server wird über das Modul `httpIO.pl` abgewickelt. Die Verbindung zur Datenbasis schaffen die Routinen im Modul `fileIO.pl`. Die Indexierung und Suche im Index wird vom Modul `index.pl` bewerkstelligt.

```
#!/usr/bin/perl

##########################################################################
# Virtual Library v1.0
##########################################################################
#
# (C) 1997 Michael Hahsler.
#
# This software is WITHOUT ANY WARRANTY!
#                                                                        10
#

# Get plpath. This works only if symbolic links are supported
# for the cgi-directory. Otherwise copy virlib.pl and
# global.pl to the cgi-directory directory and use $plpath =$0;
# instead of $plpath =readlink($0);

$plpath =readlink($0);
# $plpath =$0;
$plpath =~ s/virlib.pl//;                                              20

# load modules
```

```
require "${plpath}/global.pl";
&global;          # this redefines plpath !!!

require "${plpath}progs/fileIO.pl";
require "${plpath}progs/httpIO.pl";
require "${plpath}progs/index.pl";

                                                                                30
##########################################################################
# Main
##########################################################################
#
#

local (%input,$cue,$key,$salt);

$salt = "vl";    # salt for crypt
                                                                                40
umask(umask && 007|7); # rw for owner and group

if (%input=&ReadParse) {

$type=$input{type};
if ($type =~ /-/) {                    # get $cue
     $type =~ /(.*)-(.*)/;
     $type = $1;
     $cue = $2;
}                                                                               50

# get global variables
$pathvl=$input{pathvl};
$wwwurl=$input{wwwurl};

$pwd=$input{pwd};
$file=$input{file};
$file=~ s/.*;\s*(\d+)/$1/;             # manual override of file

$pathvlh = $pathvl;                    # for the GET-method-variables             60
$pathvlh =~ s/\//%2F/gi;
$wwwurlh = $wwwurl;
$wwwurlh =~ s/\//%2F/gi;

&DefineSub;

# public functions

     if ($type eq "newentry") {&NewEntry ($cue, %input);}
     elsif ($type eq "doquery") {&DoQuery                                       70
               ($input{query},0,"library",$input{sortby},$input{qflags});
     }
     elsif ($type eq "getlist") {&DoQuery
```

```
            ("#",0,$cue,$input{sortby},$input{qflags});
    }
    elsif ($type eq "showentry") {&ShowEntry ($file);}
    elsif ($type eq "script") {&Script ($input{script},"usr",
                                        %input);}
# admin functions
                                                                        80
    elsif (&CheckPwd($pwd)!=1) {
        &Error("Password or Command unknown");
        exit;
    } else {

        # secure communication from here on!
        $plhome=$plhomesecure;
        &DefineSub;

        if ($type eq "getadmin") {&GetAdmin("new","old","library",      90
                                            "feedback",$input{cuetop});}

# manipulation of entries
        elsif ($type eq "getquery") {&Message("adminquery");}
        elsif ($type eq "adminquery") {&DoQuery
              ($input{query},1,"library",$input{sortby},$input{qflags});
        }
        elsif ($type eq "adminlist") {
            &DoQuery ("#",1,$cue,$input{sortby});
        }                                                               100
        elsif ($type eq "review") {&GetReview ($cue, $file);}
        elsif ($type eq "changeit") {&ChangeIt ($cue, $file, %input);}
        elsif ($type eq "enterit") {&EnterIt ($cue, $file, %input);}
        elsif ($type eq "moveit") {&MoveIt ($cue, $file,$input{to},%input);}
        elsif ($type eq "killit") {&KillIt ($cue, $file);}
        elsif ($type eq "killlist") {&KillList ($cue);}

# administration of the Password
        elsif ($type eq "pwdnewget") {&Message ("pwdnew");}
        elsif ($type eq "pwdnewput") {&PutPwd ($input{pwdold},          110
                                               $input{pwdnew1},
                                               $input{pwdnew2});
                }
# index
        elsif ($type eq "doindex") {&DoIndex ($input{sortby});}

# script
        elsif ($type eq "adminscript") {&Script ($input{script},"admin",
                                                %input);}
                                                                        120
# automatic stuff with at
        elsif ($type eq "saveautoconfig") {&SaveAutoconfig(%input);}
        elsif ($type eq "testauto") {&TestAuto;}
        elsif ($type eq "addauto") {&AddAuto;}
```

```
        elsif ($type eq "removeauto") {&RemoveAuto;}

# automatic stuff for the admin of the VirLib-server
        elsif ($pathvl eq $plpath) {
            if ($type eq "showautolist") {&ShowAutoList;}
            elsif ($type eq "changeautolist") {&ChangeAutoList                    130
                                                ($input{kill},$input{add});}
            elsif ($type eq "doat") {&DoAt;}
            elsif ($type eq "killat") {&KillAt ($input{jobnumber});}
            elsif ($type eq "startat") {&StartAt ($input{starttime});}
            elsif ($type eq "viewlog") {&ViewLog;}

        }
        else { &Message("unknown Command");}
        exit;
   }                                                                          140
} else {    #No Data submited!
    &ShowVersion;
}
exit;


##############################################################################
# Functions
##############################################################################
                                                                             150
# Here I need to redirect the errormessages to CgiError !
sub Error {
    local ($number) = @_;
    &CgiError ($number);
    exit;
}

##############################################################################
# Public-Section
                                                                             160
# Process new entries
sub NewEntry {
    local ($cue, %in) = @_;

    $in{date}=&Date;      # a new entry needs a date
    &AppendEntry($cue, %in);
    &Message ("subok$cue");
}

# Puts an entry on the stack $cue                                            170
sub AppendEntry {
    local ($cue, %in) = @_;

    &Lock($cue);
```

```
    $top = &GetTop($cue);
    $top++;
    &SaveFile($cue, $top, %in);
    &PutTop($cue, $top);
                                                                        180
    &Unlock ($cue);
}

# Prints an entry for the enduser
sub ShowEntry {
    local ($file) = @_;
    local (%in);

    %in = &ReadFile ("library", $file);
    &Message ("entry", " ", %in);                                       190
}

#########################################################################
# Admin-Section

sub GetAdmin {
    local (@cuelist) = @_;
    local ($cue,%in,$cuetop);

    foreach $cue (@cuelist) {                                           200
        $cuetop ="${cue}top";
        if ($cue ne "") {
            $in{$cuetop}=&GetTop($cue);
        }
    }
    &Message("admin"," ",%in);
}

# Produces an editable review-page
sub GetReview {                                                         210
    local ($cue, $file) = @_;
    local (%data);

    %data = &ReadFile($cue,$file);

    &FillForm("review",%data);
}

# replaces an entry with a new one
sub ChangeIt {                                                          220
    local ($cue, $file, %in) = @_;
    local ($key);
    if ($in{condition} eq "ok") {delete $in{count};}
    &SaveFile($cue, $file, %in);
    &Message("okadmin");
}
```

```
# moves an entry from $cue to library
sub EnterIt {
    local ($cue, $file, %in) = @_;                                    230
    &AppendEntry("library",%in);

    &RemoveFile($cue,$file);
    &Message ("okadmin");
}

# moves an entry from $cue to $to
sub MoveIt {
    local ($cue, $file, $to,  %in) = @_;
                                                                      240
    if (!-e $pathvl.$to) {&Error($to);}

    &AppendEntry($to,%in);

    &RemoveFile($cue,$file);
    &Message ("okadmin");
}

# kill the entries $files in $cue (moves them to old)
# only entries in old are physically destroyed                        250
sub KillIt {
    local ($cue, $files) = @_;
    local (%in,$file,@filelist);

    @filelist= split("; ", $files);
    foreach $file (reverse sort numerically @filelist) {
        if ($cue eq "library") {
            %in = &ReadFile ($cue,$file);
            $in{condition}="killmark";
            &SaveFile($cue,$file,%in);                                260
        }else{
            if ($cue ne "old" && $cue ne "feedback") {
                # killit moves an entry into list old
                %in = &ReadFile ($cue,$file);
                &AppendEntry("old",%in);
            }
            &RemoveFile ($cue,$file);
        }
    }
    $files="";                                                        270
    &DefineSub;
    &Message ("okadmin");
}

# kills all entries in the given list the hard way
sub KillList {
    local ($cue) =@_;
```

```
    local ($rm,$res);
    $rm ="rm -f ".$pathvl.$cue."/*.dat";
    $res = `$rm`;                                                    280
    &Message ("okadmin");
}

#########################################################################
# Password-Section

# Change the passwordfile
sub PutPwd {
    local ($pwdold,$pwdnew1,$pwdnew2) =@_;
    local (%out,$salt);                                              290
    $salt="vl";

    if (&CheckPwd($pwdold)!=1) {&Error("wrong Password"); exit;}
    if ($pwdnew1 eq $pwdnew2) {
        $pwd=$pwdnew1;
        $out{passwd}=crypt($pwd,$salt);
        &SaveFile("config",".pwd",%out);
        chmod 0600, "${pathvl}config/.pwd.dat"; # only for the owner.

        # we need to redefine stuff with the new password.           300
        &DefineSub;

        &Message("okadmin");
    } else {
        &Error("new Passwords don't match");
    }
}

# Read the passwordfile
sub GetPwd {                                                         310
    local (%passwd,$salt);
    $salt ="vl";

    if (!-e $pathvl."config/.pwd.dat") {return (crypt($free,$salt));}
    %passwd=  &ReadFile("config",".pwd");
    return $passwd{passwd};
}

# Check the input against the real password
sub CheckPwd {                                                       320
    local ($pwd) =@_;
    local ($salt);
    $salt ="vl";

    if (crypt($pwd,$salt) ne &GetPwd) {
        return 0;
    }else{
        return 1;
```

```
    }
}

##########################################################################
# Script-Section

# Starts a script that is not part of the VirLib-System.
# Generally it is prefered to use this than to change the code
# of the VirLib-Server.

sub Script {
    local ($prog,$directory,%in) = @_;
    local ($result);

    $prog = $pathvl."progs/".$directory."/".$prog;

    if (!-e $prog) { &Error ("Script");}

    require "$prog";
    $result = &StartScript (%in);

    if ($result=~ s/!no output//) {
        print &HttpHeader,$result;
    } else {
        &Message ("okadmin",$result);
    }
}

##########################################################################
# Index-Section

# produce an index for library
sub DoIndex {
    local ($sortby)=@_;
    local ($output);

    $output = &KillMark."\n\n";
    $output .= &Index."\n\n";
    $output .= &DoSort ("library",$sortby);

    &Message ("okadmin", $output);
}

# Query
sub DoQuery {
    local ($query,$admin,$cue,$sortby,$qflags) = @_;
    local ($output, $i,  %data, $line, $line2, $key, $errors, $matches,
           @files, %list, $top, $outputstyle);

# set the outputstyle
    if ($cue eq "feedback") {
```

```
    $outputstyle="listfeedback";
}elsif ($admin==1) {
    $outputstyle="resultadmin";
} else {
    $outputstyle="result";
}

$qflags =~ s/;/ /g;

#read a line with the selected outputstyle
$line=&ReadLine($outputstyle);

if ($query eq "#") {          # this will produce the whole list
    $matches =0;
    if ($LANGUAGE eq "deutsch") {
        $query="-gesamte Liste-";
    } else {
        $query="-whole list-";
    }
    $top = &GetTop($cue);
    $matches = $top;
    @files=();          # get a list of the files that match
    foreach $i (1 .. $top) {
        @files=(@files,$i);
    }

}else{
    @files=&Query($query,$qflags);
}

if ($sortby) {
    $matches=0;
    # sort it with the existing sort-file
    %list=&ReadFile($cue,"sort");
    foreach (split("; ", $list{$sortby})) {
        $file = $_;
        if(grep(/^$file$/,@files)) {
            $matches++;
            &DefineSub;        # to insert the right $file
            $line2=$line;
            %in = &ReadFile ($cue, $file);
            foreach $key (sort keys(%in)) {
                ${$key} = $in{$key};
                ${$key} =~ s/\n/<BR>/g;
            }
            $line2 =~  s/{\$(.*)}/${$1}/gi;
                        # add normal variables to the page
            $output .= $line2;
            foreach $key (sort keys(%in)) {
                ${$key} = "";       # kill variables, they would
                                    # influence the following entry!
```

```
                }
            }
            $file="";
            &DefineSub;
        }
    } else {

        $matches =0;
        # unsorted output
        foreach $file (@files) {                                    440
            $matches++;
            $line2=$line;
            &DefineSub;         # enter $file in the $stuff and $stuffh

            %dat = &ReadFile ($cue, $file);
            foreach $key (sort keys(%dat)) {
                    ${$key} = $dat{$key};
            }
            $line2 =~ s/{\$(.*)}/${$1}/gi;
      # add normal variables to the page                          450
            $output .= $line2;

            foreach $key (sort keys(%dat)) {
                ${$key} = "";
            }
        }
    }

    if ($matches == 0) {        # tries to use a message called no...
        if (-e $pathvl."html/no".$outputstyle.".html") {            460
            &Message ("no$outputstyle");
        }else{
            &Message ($outputstyle);
        }
        return;
    }
&Message($outputstyle,$output);
}

# sorts all enrties in library alphapetically by the value of the   470
# variable given by sortby
sub DoSort {
    local ($cue,$sortitby) =@_;
    local (%list, %in ,$file, $top, $key,%output,$sortby);

    $top = &GetTop($cue);

    @sortlist= split(/,\s*/, $sortitby);
    foreach $sortby (@sortlist) {
        %list=();                                                   480
    foreach $file (1 .. $top) {
```

```
        %in=&ReadFile($cue,$file);
        while ($list{$in{$sortby}} ne "") {        # I need all entries even
            $in{$sortby}.="I";                     # with the same key!!!
        }
        $list{$in{$sortby}}=$file;
    }
    foreach $key (sort keys(%list)) {
        $output{$sortby}.=$list{$key}."; ";
    }                                                                          490
    chop ($output{$sortby});
    chop ($output{$sortby});
}
    &SaveFile ($cue,"sort",%output);
    return("Library sorted by:  $sortitby");
}

sub KillMark {
    local ($cue, $top, $file, $kills, $result,$output, %in, $index);
                                                                               500
    $cue="library";
    $kills=0;

    $top = &GetTop ($cue);

    foreach $file (1 .. $top) {
        %in=&ReadFile($cue,$file);

        if ($in{condition} eq "killmark") {
            &AppendEntry("old",%in);                                           510
            &RemoveFile($cue,$file);
            $top --;
            $file --;
            $kills ++;
        }
    }
    $output = "Killmark\n";
    $output .= "\tEntries killed: $kills\n";
    return $output;
}                                                                              520

##########################################################################
# Automatic-Section with at

sub TestAuto {
    local (%libs,$output,$i,$line,$line2,$pathvlold,%out);

    $i=0;
    $pathvlold=$pathvl;
    $pathvl=$plpath;                                                           530

    %libs=&ReadFile("config","libs");      #read from the VirLib-Server
```

```
    if ($libs{libs} =~ /$pathvlold/) {
        if ($LANGUAGE eq "deutsch") {

            $output="Ja";
        } else {
            $output="Yes";
        }
    } else {
        if ($LANGUAGE eq "deutsch") {
            $output="Nein";
        } else {
            $output="No";
        }
    }
    $pathvl=$pathvlold;

    %out=&ReadFile("config","autoconfig");
    $out{service}=$output;
    &FillForm ("testauto",%out);
}

# saves autoconfig.dat and goes back to TestAt
sub SaveAutoconfig {
    local (%in) =@_;

    &SaveFile("config","autoconfig",%in);
    &TestAuto;
}

# enables the automated services
# (adds $pathvl to the list of the VirLib-Server)
sub AddAuto {
    local ($cue) =@_;
    local (%libs,$pathvlold);

    $pathvlold=$pathvl;
    $pathvl=$plpath;

    &Lock("config");

    %libs=&ReadFile("config","libs");
    if (!($libs{libs} =~ /$pathvlold/)) {
        $libs{libs}.="; $pathvlold";
        $libs{libs} =~ s/^;\s//;
        &SaveFile("config","libs",%libs);
    }
    &Unlock ("config");

    $pathvl=$pathvlold;
    &TestAuto;
```

```
}

# disables the automated services
sub RemoveAuto {
    local ($cue) =@_;
    local (%libs,$pathvlold);
                                                                    590
    $pathvlold=$pathvl;
    $pathvl=$plpath;

    &Lock("config");

    %libs=&ReadFile("config","libs");
    $libs{libs} =~ s/;*\s*$pathvlold//g;
    $libs{libs} =~ s/^;\s//;
    &SaveFile("config","libs",%libs);
                                                                    600
    &Unlock("config");

    $pathvl=$pathvlold;
    &TestAuto;
}

##########################################################################
# Automatic-Section with at for the VirLib Admin

# prints a list of the libraries which currently use the            610
# automated services
sub ShowAutoList {
    local (%libs,$output,$i,$line,$line2);

    $i=0;
    %libs=&ReadFile("config","libs");
    $line=&ReadLine("listauto");

    foreach (split("; ", $libs{libs})) {
        $lib = $_;                                                  620
        $line2=$line;
        $line2 =~ s/{\$(.*)}/${$1}/gi;
        $output .= $line2;
    }
    &Message ("listauto",$output);
}

# allows to edit the list
sub ChangeAutoList {
    local ($kills,$new) = @_;                                       630
    local (%libs,$wait);

    &Lock ("config");
```

```
    %libs=&ReadFile("config","libs");
    foreach (split("; ", $kills)) {
        $kill = $_;
        $libs{libs} =~ s/;*\s*$kill//g;
        $libs{libs} =~ s/^;\s//;
    }
    if ((!($libs{libs} =~ /$new/)) && ($new =~/\w/)) {
        $libs{libs}.="; $new";
        $libs{libs} =~ s/^;\s//;
    }
    &SaveFile("config","libs",%libs);

    &Unlock ("");

    &ShowAutoList;
}

# administrate at
sub DoAt {
    local ($atout,$output,$at,$number,$when,%in);

    $at = "at -l ";
    $atout = `$at`;
    if ($LANGUAGE eq "deutsch") {
        $output ="<pre>Ergebnis von at:\n";
        if ($atout) {
            $output .= $atout;
        } else {
            $output .= "Kein Job vorhanden.";
        }
        $output .="</pre>";
    } else {
        $output ="<pre>Result of at:\n";
        if ($atout) {
            $output .= $atout;
        } else {
            $output .= "No job.";
        }
        $output .="</pre>";
    }
    $in{number}=$number;
    &Message ("at",$output,%in);
}

# stops an at-job
sub KillAt {
    local ($jobnumber)= @_;
    local ($output,$atrm);
        $atrm = "atrm $jobnumber";
        $output=`$atrm`;
    &DoAt;
```

```
}

# starts an at-job
sub StartAt {
    local ($starttime)=@_;                                              690
    local ($at,$output);
    $at = "at -f ${plpath}progs/auto.sh $starttime";
    $output = `$at`;
    &DoAt;
}

# prints the log-file of the automated services
sub ViewLog {
    local ($log);
                                                                        700
    open (LOG, "${pathvl}config/log") or
         &Error("Log");

    while (<LOG>) {
        $log .= $_;
    }
    close LOG;
    &Message("okadmin",$log);
}
                                                                        710
##########################################################################
# Stuff

sub Date {
    local ($sec,$min,$hour,$mday,$mon,$year,$yday,$isdst);

    ($sec,$min,$hour,$mday,$mon,$year,$yday,$isdst)=localtime(time);
    return ("$mday/".($mon+1)."/$year $hour:$min");
}
                                                                        720
sub DefineSub {

    $stuff = "<input type=\"hidden\" name=\"pwd\" value=\"$pwd\">\n".
        "<input type=\"hidden\" name=\"pathvl\" value=\"$pathvl\">\n".
        "<input type=\"hidden\" name=\"wwwurl\" value=\"$wwwurl\">\n".
        "<input type=\"hidden\" name=\"file\" value=\"$file\">\n";

    $stuffh = "$plhome?&pathvl=${pathvlh}&pwd=$pwd".
        "&wwwurl=${wwwurlh}&file=${file}&type=";
}                                                                       730

sub numerically {$a <=> $b;}

sub ShowVersion {
    print &HttpHeader,"<HTTP><HEAD>\n".
        "<TITLE>The Virtual Library</TITLE>".
```

```
        "</HEAD><BODY><H1>The Virtual Library<HR></H1>".
        "<tt>".
        "<b>Version:</b> 1.0<BR>" .
        "<b>Date:</b> 3.7.1997<p>" .                                    740
        "For Informations about a newer version of the Virtual Library " .
        "try:<p><center>".
        "http://miss.wu-wien.ac.at/~virlib/</center></tt>" .
        "<p><hr>(c) 1997 Michael Hahsler, ".
        "<a href=\"http://miss.wu-wien.ac.at/~virlib/\">".
        "The Virtual Library</a>\n".
        "</BODY></HTML>";
    exit;
}
```

C.2 Modul httpIO.pl

Modul zur Kommunikation mit dem WWW-Server über das *Common Gateway Interface*.

```
#!/usr/bin/perl

##############################################################################
# Virtual Library v1.0
# HTTP I/O
##############################################################################
#
# (C) 1997 Michael Hahsler.
#
# This software is WITHOUT ANY WARRANTY!                                    10
#

#
# %data = ReadParse [($data)]
# HttpHeader
# CgiError ($number)
# Message ($file,$data,%in)
# FillForm ($form,%data)
#
                                                                            20
# Reads data from the httpd and transforms dem into an ass. array
sub ReadParse {
#   local (*in) = _ if _;
    local (%in, @in, $in, $i, $key, $val);

#   Read (POST or GET?)
    if ($ENV{'REQUEST_METHOD'} eq "GET") {
        $in = $ENV{'QUERY_STRING'};
    } elsif ($ENV{'REQUEST_METHOD'} eq "POST") {
        read(STDIN,$in,$ENV{'CONTENT_LENGTH'});                             30
    }
    @in = split(/&/,$in);
    foreach $i (0 .. $#in) {
        # + -> space
        $in[$i] =~ s/\+/ /gi;
        # split
        ($key, $val) = split(/=/,$in[$i],2);
        # translate the escapesequences
        $key =~ s/%(..)/pack("c",hex($1))/gie;
        $val =~ s/%(..)/pack("c",hex($1))/gie;                              40
        # Corrections
        $val =~ s/\r\n/\n/gi;       # kill CRLF's

        $val =~ s/}/)/gi;           # translate preserved charactetrs ({};)
        $val =~ s/{/(/gi;
```

```
        $val =~ s/;/,/gi;              # ; -> ,

        if ($key eq "URL") {
            $val =~ s/^http:\/\///gi;#kill http:// at the begin of an URL
        }
        $val =~ s/''/\"/g;      # repair the "
        # construct an ass. array
        if ($val ne "") {
            $in{$key} .= "; " if (defined($in{$key}));
            # ; is the multiple separator
            $in{$key} .= $val;
        }
    }
    if ($in{condition} eq "ok") {delete $in{condition};}
                    # saves diskspace (17 bytes per entry)
    return %in;
}

# The Httpd needs this
sub HttpHeader {
    return "Content-type: text/html\n\n";
}

# Errormessage
# !!! do not use &Message here - the filesystem might be unreachable !!!
sub CgiError {
    local ($number) =@_ if @_;

    if (-e $pathvl."html/error.html") {
        &Message("error",$number);
    } else {
        if ($LANGUAGE eq "deutsch") {
            print &HttpHeader .
                "<title>Fehlermeldung</title>" .
                "<html><HR><center><h1>Der Fehler <I>".$number .
                " </I> ist aufgetreten!</h1><HR><p>\n" .
                "Bitte wenden Sie sich an den " .
                "Betreuer\n</center><\html>";

        }else{
            print &HttpHeader .
                "<title>Error Message</title>" .
                "<html><HR><center><h1>The error <I>".$number .
                " </I> has occured!</h1><HR><p>\n" .
                "Please contact the " .
                "administrator\n</center><\html>";
        }
    }
    die;
}
```

```
# Read a file from ./html, replace the variables and put it to the httpd.
sub Message {
    local ($file, $data, %in) = @_;
    local ($key);                                                    100
    foreach $key (sort keys(%in)) {
        ${$key} = $in{$key};
    }
    open (HTML,"${pathvl}html/${file}.html") or
         &CgiError("Message ".$file);
    print &HttpHeader;
    while (<HTML>) {
        s/{\$([^}]*)}/${$1}/gi;
        print;
    }                                                                110
    close (HTML);
}

# Prints the Form $form completed with the variables in %data
sub FillForm {
    local ($form, %data) = @_;
    local ($key, $value, $value2, @buf2, $date);

    open (HTML, "${pathvl}html/$form.html")
        or   &CgiError("Message $form");                             120

    @buf2=<HTML>;
    print &HttpHeader;

# inserts the information of the entry to the html-page
    foreach $key (sort keys(%data)) {
        foreach (split("; ", $data{$key})) {

            s/\"/''/g; # you can't use " as a preset for a form
            $value = $_;                                             130

            $value2=$value;
            $value2 =~ s/(\W)/\\$1/g;

            foreach (@buf2) {
                s/(INPUT\s+NAME\s*=\s*\"$key\")/$1 VALUE =\"$value\"/gi;

                s/(TEXTAREA NAME=\"$key\".*>)/$1$value/gi;

                $value && s/(<OPTION>$value2$)/                      140
                           <OPTION SELECTED>$value/gi;

                s/(NAME=\"$key\" VALUE=\"$value2\")/
                           $1 CHECKED /gi;
                }
            }
        }
```

```
    foreach $key (sort keys(%data)) { # get all variables
        ${$key} = $data{$key};     # for substitution
    }
    foreach $i (0 .. $#buf2) {
        $buf2[$i] =~ s/{\$([^}]*)}/${$1}/gi;
        # add normal variables to the page
    }
    print @buf2;
    close HTML;
}

1;
```

C.3 Modul fileIO.pl

Modul zur Manipulation der Datenbasis.

```
#!/usr/bin/perl

##########################################################################
# Virtual Library v1.0
# File I/O
##########################################################################
#
# (C) 1997 Michael Hahsler.
#
# This software is WITHOUT ANY WARRANTY!
#

#
# $top = GetTop ($cue)
# PutTop ($cue,$top)
# SaveFile ($cue,$file,%data)
# %data = ReadFile ($cue,$file)
# Lock ($cue)
# Unlock ($cue)
# WhileLock ($cue)
# Readfile ($file)
# KillFile ($cue,$file)
# RemoveFile ($cue,$file)
# $top = RepairTop ($cue)
#

# Read top.dat
sub GetTop{
    local ($cue) = @_;
    local (%top);
    # is top ok?
    if (!-e $pathvl.$cue) {return(-1);} # no cue with this name!
    if (!-e $pathvl.$cue.'/top.dat') {&RepairTop($cue);}
    # insert file
    %top=&ReadFile($cue,"top");
    return $top{top};
}

# Write top.dat
sub PutTop{
    local ($cue, $top) =@_;
    local (%thetop);
    $thetop{top}=$top;
    &SaveFile($cue,"top",%thetop);
}
```

```
# Save an entry
sub SaveFile {
    local ($cue, $file, %in) =@_;
    # kill administrative lines                                    50
    delete $in{pathvl};
    delete $in{type};
    delete $in{file};
    delete $in{pwd};
    delete $in{wwwurl};
    delete $in{to};

    open (FILE, ">$pathvl${cue}/${file}new.dat") or  &Error("File");
    foreach $key (sort keys(%in)) {
        print FILE "$key := {$in{$key}}\n";                         60
    }
    close (FILE);
#     chmod 660, '$pathvl${cue}/${file}.dat';
    `mv $pathvl${cue}/${file}new.dat $pathvl${cue}/${file}.dat`;

}

# Read a file and transform it into an ass. array
sub ReadFile {
    local ($cue, $file) =@_;                                        70
    local (%in, $in, @in);

    $/="}";       # so we can use NEWLINE and all the stuff
    $*=1;
    open (FILE, "<$pathvl${cue}/${file}.dat") or return 0;
    while(<FILE>) {
        /^(.+\b)\s*:=\s*{([^}]*)}/;
        $in{$1} = $2;
    }
    close FILE;                                                     80
    $/="\n";      # restore the standard
    $*=0;
    return %in;
}

sub Lock {
    local ($cue)=@_;
    local ($wait);
    $wait =10;
                                                                    90
    while ((-e $pathvl.$cue."/busy.dat") && ($wait>0)) {sleep 1;$wait--;}
    &SaveFile($cue,"busy","busy\n");  #let one write after the other
}

sub Unlock {
    local ($cue)=@_;
```

```
    &KillFile ($cue,"busy");
}
                                                                    100
sub Whilelock {
    local ($cue)=@_;
    local ($wait);
    $wait=10;

    while ((-e $pathvl.$cue."/busy.dat") && ($wait>0)) {sleep 1;$wait--;}

}

sub ReadLine {                                                      110
    local ($file)=@_;
    local ($line);

    open (LINE, "${pathvl}html/$file.line") or
            &Error("Message $file.line");

    while (<LINE>) {
        $line .= $_;
    }
    return $line;                                                   120
}

# Delete a file
sub KillFile {
    local ($cue,$file) = @_;
    local ($top,$rm,$res);
    $rm ="rm -f ".$pathvl.$cue."/".$file.".dat";
    $res = `$rm`;

}                                                                   130

# Remove a file from a list and fix top
sub RemoveFile {
    local ($cue, $file) = @_;
    local ($wait);

    &Lock($cue);

    $top = &GetTop($cue);
    if (($file<=$top) && ($top!=0)) {                               140
        if ($file!=$top){           # no rename for the top-entry!
            &SaveFile($cue,$file,&ReadFile($cue,$top));
        }
        &KillFile($cue,$top);
        $top--;
        &PutTop($cue, $top);
    }
    &Unlock ($cue);
```

```
}
                                                                    150
# Reconstruct top.dat
sub RepairTop {
    local ($cue) = @_;
    local ($file);
    $file = 1;
    while ((-e $pathvl.$cue."/".$file.".dat")
            || (-e $pathvl.$cue."/".($file+1).".dat")) {
        $file++;        # a file may be lost! -> $file+1
    }
    $file--;                                                        160
    &PutTop ($cue, $file);
    return $file;
}

1;
```

C.4 Modul index.pl

Modul zur Indexierung der Datenbasis und für die Suche im Index.

```
#!/usr/bin/perl

##########################################################################
# Virtual Library v1.0
# Index
##########################################################################
#
# (C) 1997 Michael Hahsler.
#
# This software is WITHOUT ANY WARRANTY!
#

#
# Index
# Query ($query,$qflags)
#

# Starts GLIMPSEindex
sub Index {
    local ($output,$indexit);

    if (!-e $glimpseindex) { &Error ("GLIMPSEINDEX");}
    if (!-e "${pathvl}indexnew") {`mkdir ${pathvl}indexnew`;}
    $indexit=$glimpseindex." -H ${pathvl}indexnew -z -n ".
        "-B ${pathvl}library/";
    $output = `$indexit`;

    $output = "<pre>$output</pre>";
    #htmlify the output   $line2 =~ s/{\$(.*)}/${$1}/gi;
            # add normal variables to the page
    `chmod 660 ${pathvl}indexnew/.glimpse*`;
    `cp ${pathvl}indexnew/.* ${pathvl}index/ 2>/dev/null`;
    return ($output);
}

# Starts GLIMPSE
sub Query {
    local ($query,$qflags) = @_;
    local (@queryout, $queryout, $i, $matches, @files,$category);

    $query =~ s/\s+AND\s+/;/gi;
    $query =~ s/\s+OR\s+/,/gi;
    $query =~ s/\s+/,/gi;

    $query =~ s/\(/\{/gi;
    $query =~ s/\)/\}/gi;
```

```
    if (!$qflags) {
        $qflags="-1 -W -i";# 1 error, boolean scope = file, case-insensitive
    }
    $qflags = " ".$qflags." -H ${pathvl}index -O -y -T $pathvl ";
          # vital flags (used directories, outputstyle, yes)

    if ($query =~ /(.*):=(.*)$/) {
        $category = $1;
        $query = $2;
    }

    if (!-e $glimpse) { &Error ("GLIMPSE");}
    $glimpse .=$qflags."'".$query."'";
    $queryout = `$glimpse`;

    if ($queryout eq "") {
        $matches = 0;  # this means no matching entries found
    }else{
        @queryout = split(/\n\n/,$queryout);          # get a list of hits

        @files=();          # get a list of the files that match
        $matches = $#queryout+1;
        foreach $i (0 .. $#queryout) {
            $queryout[$i] =~ /\/(\d+).dat/;
            $file=$1;

            # structured query (category:=value)
            if ($category) {
                $hits ="";                  # extract the categories of hits
                while ($queryout[$i] =~ s/\s+(.+):=//){
                    $hits .= $1."; ";
                }
                if ($hits =~ /$category/) {
                    @files=(@files,$file);
                } else {
                    $matches --;
                }
            } else {
                @files=(@files,$file);
            }
        }
    }
    return (@files)
}

1;
```

C.5 Programm auto.pl

Automatische Dienste der virtuellen Bibliothek. Dieses Programm wird automatisch jeden Tag gestartet und führt für alle angemeldeten Bibliotheken bestimmte Dienste aus. Die Routine zur Abfrage, ob bestimmte HTML-Seiten über das Internet erreichbar sind, ist dem Buch *Learning Perl* [Schwartz, 1993] entnommen.

```
#!/usr/bin/perl

##########################################################################
# Virtual Library v1.0
# Automated Services
##########################################################################
#
# (C) 1997 Michael Hahsler.
#
# This software is WITHOUT ANY WARRANTY!                                  10
#
#

# get plpath

$plpath =$0;
$plpath =~ s/auto.pl//;

#Global Variables
require "${plpath}/global.pl";                                            20
&global;          # this redefines plpath

require "${plpath}progs/fileIO.pl";
require "${plpath}progs/index.pl";

local ($i,%input,%in,$ok,$unreachable,%libs,%result,$message,$pathvl,$cue);

$cue="library";
$message=0;     # no feedback if there are no errors
$input{eMail}="VirLib-Server";                                            30
umask(umask && 007|7); # makes new created files rw for the group

# restart at
$at = "at -f ${plpath}progs/auto.pl  now + 1 day 2>${plpath}config/log";
$output = `$at`;

# create a log-file
open (LOG, ">>${plpath}config/log") or  &Error("LOG");
print LOG "At restarted.\n".$output."\n";
print LOG "Date: ".&Date."\n";                                            40

$i=0;
```

```
$pathvl=$plpath;
%libs=&ReadFile("config","libs");

foreach (split("; ", $libs{libs})) {
    $pathvl = $_;          # this is global !!!!
    print LOG "-----------------------------------------\n";
    print LOG "Library: $pathvl\n\n";                                    50
    $input{Feedback} .= "Automated Services for: $pathvl\n";
    $i++;

    if (!-e $pathvl) {
        &Error("Library does not exist");

    } else {
    if  (!-e $pathvl."config/autoconfig.dat") {
        &Error("No autoconfig-file");
    } else {                                                             60
        %conf=&ReadFile("config","autoconfig");

    # Kill all files in Library which have the condition "killmark"
        $result = &KillMark;
        $input{Feedback} .= $result;
        print LOG $result;


    # Checks the URLs and kills files which are unreachable for the
    # "autokillat" time if "autokill" is yes                             70
    if ($conf{checkURL} eq "yes") {
        $result = &DoCheckURL($conf{autokill},$conf{autokillat});
        $input{Feedback} .= $result;
        print LOG $result;
    }

    # Indexes the library
    if ($conf{index} eq "yes") {
        $output = &DoIndex($conf{sortby});
        $input{Feedback} .= "Entries indexed.\n";                        80
        print LOG "Entries indexed and sorted by: $conf{sortby}\n";

    }

    # Executes a script stored in progs/admin of the library
    if ($conf{script} eq "yes") {
        $result = &Script($conf{scriptname},"admin");
        $input{Feedback} .= "Script executed.\n";
        print LOG "Script\n$result\n";
    }                                                                    90
    }

    # Sends the results as feedback to the owner of the library
    # if wanted.
```

```
    if (($message==1) || ($input{Feedback}=~/error/gi)) {
        &NewEntry ("feedback",%input);
    }
    }
    $input{Feedback}="";
                                                                    100
}
close LOG;

exit;

sub DoCheckURL {
    local ($autokill,$autokillat)=@_;
    local ($cue,$top, $file, $kills,$ok, $result,$output,  %in, $index);

    $cue="library";                                                 110
    $ok=0;
    $unreachable=0;
    $kills=0;

    $top = &GetTop ($cue);
    if ($top==-1) {
        &Error ("Automated Service has problems with your library!");
        return;
    }
    $file=$top;                                                     120
    while ($file>0) {
        %in=&ReadFile($cue,$file);
        $result = &CheckURL("http://".$in{URL});
        if ($result!=200) {
            $in{condition}="unreachable";
            $in{count}++;
            $unreachable++;
        } else {
            $ok++;
            delete $in{condition};                                  130
            delete $in{count};
        }

        print $file," ",$top," ",$autokill," ",$in{count}," ",$autokillat,"\n";

        if ($autokill eq "yes"&&$in{count}>=$autokillat) {
            &AppendEntry("old",%in);
            &RemoveFile($cue,$file);
            $kills++;
        } else {                                                    140
            &SaveFile($cue,$file,%in);
        }
        $file--;
    }
    $output = "CheckURLs\n".
```

```
        "\tTotal number of entries: $top\n".
        "\tEntries ok: $ok\n".
        "\tEntries unreachable: $unreachable\n".
        "\tEntries killed: $kills\n";
    return $output;
}

##############################################################################
# Indexing

sub DoIndex {
    local ($sortby)=@_;
    local ($output);

    $output .= &Index."\n\n";
    $output .= &DoSort ("library",$sortby);

    return ($output);
}

# sorts all entries in library alphapetically by the value of the
# variable given by sortby
sub DoSort {
    local ($cue,$sortby) =@_;
    local (%list, %in ,$file, $top, $key,%output);

    $top = &GetTop($cue);

    foreach $file (1 .. $top) {
        %in=&ReadFile($cue,$file);
        while ($list{$in{$sortby}} ne "") {        # I need all entries even
            $in{$sortby}.="I";                     # with the same key!!!
        }
        $list{$in{$sortby}}=$file;
    }
    foreach $key (sort keys(%list)) {
        $output{$sortby}.=$list{$key}."; ";
    }
    chop ($output{$sortby});
    chop ($output{$sortby});

    &SaveFile ($cue,"sort",%output);
    return("Library sorted by:  $sortby");
}

sub KillMark {
    local ($cue, $top, $file, $kills, $result,$output, %in, $index);

    $cue="library";
    $kills=0;
```

```
    $top = &GetTop ($cue);

    foreach $file (1 .. $top) {
        %in=&ReadFile($cue,$file);                                          200

        if ($in{condition} eq "killmark") {
            &AppendEntry("old",%in);
            &RemoveFile($cue,$file);
            $top --;
            $file --;
            $kills ++;
        }
    }
    $output = "Killmark\n";                                                 210
    $output .= "\tEntries killed: $kills\n";
    return $output;
}

##############################################################################
# Script
# Starts a script that is not part of the VirLib-System.
# Generally it is prefered to use this than to change the code
# of the VirLib-Server
sub Script {                                                                220
    local ($prog,$directory,%in) = @_;
    local ($result);

    $prog = $pathvl."progs/".$directory."/".$prog;

    if (!-e $prog) { &Error ("Script");}
    require "$prog";

    $result = &StartScript (%in);
    return($result);                                                        230

}

##############################################################################
# Manipulation of entries

# Process new entries
sub NewEntry {
    local ($cue, %in) = @_;
                                                                            240
    $in{date}=&Date;      # a new entry needs a date
    &AppendEntry($cue, %in);
    &Message ("subok$cue");
}

# Puts an entry on the stack
sub AppendEntry {
```

```
    local ($cue, %in) = @_;
    local ($wait);
                                                                    250

    $wait =10;
    while ((-e $pathvl.$cue."/busy.dat") && ($wait>0)) {sleep 1;$wait--}
    &SaveFile($cue,"busy","busy\n");  #let one write after the other

    $top = &GetTop($cue);
    $top++;
    &SaveFile($cue, $top, %in);
    &PutTop($cue, $top);
                                                                    260
    &KillFile ($cue,"busy");
}

# replaces an entry with a new one
sub ChangeIt {
    local ($cue, $file, %in) = @_;
    local ($key);
    if ($in{condition} eq "ok") {delete $in{count};}
    &SaveFile($cue, $file, %in);
    &Message("okadmin");                                            270
}

# moves an entry from $cue to library
sub EnterIt {
    local ($cue, $file, %in) = @_;
    &AppendEntry("library",%in);

    &RemoveFile($cue,$file);
    &Message ("okadmin");
}                                                                   280

# kill the entries $files in $cue (moves them to old)
# only entries in old are physically destroyed
sub KillIt {
    local ($cue, $files) = @_;
    local (%in,$file,@filelist);
        @filelist= split("; ", $files);
        foreach $file (reverse sort numerically @filelist) {
            if ($cue ne "old" && $cue ne "feedback") {
    # killit moves an entry into list old                           290
                %in = &ReadFile ($cue,$file);
                &AppendEntry("old",%in);
            }
            &RemoveFile ($cue,$file);
        }

    $files="";
    &DefineSub;
```

```
    &Message ("okadmin");
}                                                                              300

##############################################################################
# Stuff

sub Error {
    local ($number)=@_;

    $input{Feedback} .= "*** Error: $number\n";
    print LOG "*** Error: $number\n";
}                                                                              310

sub Message {
}

sub Date {
    local ($sec,$min,$hour,$mday,$mon,$year,$yday,$isdst);

    ($sec,$min,$hour,$mday,$mon,$year,$yday,$isdst)=localtime(time);
    return ("$mday/".($mon+1)."/$year $hour:$min");
}                                                                              320

##############################################################################
# HTTP - check URL

sub CheckURL {

    local($URL) = @_;
    # http://host:port/path

    # Get the host and port                                                    330
    if ($URL =~ m#^http://([^:/]+):?(\d*)($|/(.*))#i) {
        $host = $1;
        $port = $2;
        $path = $3;
    }else {
        return -10;
    }
    if ($path eq "") { $path = '/'; }
    if ($port eq "") { $port = 80; }
                                                                               340
    # Delete name anchor. (check if the anchor is present in the doc?)
    $path =~ s/#.*//;

# The following is largely taken from Lerning Perl Appendix B

$AF_INET = 2;
$SOCK_STREAM = 1;

$sockaddr = 'S n a4 x8';
```

```
                                                                    350
chop($hostname = `hostname`);

($name,$aliases,$proto) = getprotobyname('tcp');
($name,$aliases,$port) = getservbyname($port,'tcp') unless $port =~ /^\d+$/;
($name,$aliases,$type,$len,$thisaddr) = gethostbyname($hostname);
if (!(($name,$aliases,$type,$len,$thataddr) = gethostbyname($host))) {
  return -1;
}

$this = pack($sockaddr, $AF_INET, 0, $thisaddr);                    360
$that = pack($sockaddr, $AF_INET, $port, $thataddr);

# Make the socket filehandle.
# ** Temporary fix, this is NOT The way to do it. 15-APR-96
# But we'll still use it anyway, cannot rely on Perl to be
# installed correctly everywhere.
if (!(socket(S, $AF_INET, $SOCK_STREAM, $proto))) {
  $SOCK_STREAM = 2;
  if (!(socket(S, $AF_INET, $SOCK_STREAM, $proto))) { return -2; }
}                                                                   370

# Give the socket an address
if (!(bind(S, $this))) {
  return -3;
}

if (!(connect(S,$that))) {
  return -4;
}
                                                                    380
select(S); $| = 1; select(STDOUT);

print S "HEAD $path HTTP/1.0\n\n";

$response = <S>;
($protocol, $status) = split(/ /, $response);
while (<S>) {
}
close(S);
                                                                    390
return $status;
}
```

Literaturverzeichnis

[Alexander *et al.*, 1977] Christopher Alexander, S. Ishikawa, and M. Silverstein. *A Pattern Language*, volume 2 of *Center for Environmental Structure Series*. Oxford University Press, New York, 1977.

[Alexander, 1979] Christopher Alexander. *The Timeless Way of Building*, volume 1 of *Center for Environmental Structure Series*. Oxford University Press, New York, 1979.

[Berners-Lee and Connolly, 1995] T. Berners-Lee and D. Connolly. Hypertext Markup Language – 2.0. Request for Comments 977, Network Working Group, November 1995. Category: Standards Track.

[Berners-Lee, 1994] T. Berners-Lee. Universal Resource Identifiers in WWW. Request for Comments 1630, Network Working Group, June 1994.

[Booch, 1994] G. Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin/Cummings, Redwood City, California, second edition, 1994.

[Buschmann and Meunier, 1995] Frank Buschmann and Regine Meunier. A System of Patterns. In James O. Coplien, editor, *Pattern Languages of Program Design*. Addison-Wesley, 1995.

[Buschmann *et al.*, 1996] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern–Oriented Software Architecture, A System of Patterns*. John Wiley & Sons Ltd, Chichester, 1996.

[Cline, 1996] Marshall P. Cline. The Pros and Cons of Adopting and Applying Design Patterns in the Real World. *Communications of the ACM*, 39(10):47–49, October 1996.

[Coker *et al.*, 1970] Jerry Coker, Jimmy Casalw, Gary Campbell, and Jerry Greene. *Patterns for Jazz*. Studio Publications Recordings, Miami, Florida, 1970.

[Coplien, 1995] James O. Coplien. A Generative Development-Process Pattern Language. In James O. Coplien, editor, *Pattern Languages of Program Design*. Addison-Wesley, 1995.

[Cunningham, 1995] Ward Cunningham. The CHECKS Pattern Language of Information Integrity. In James O. Coplien, editor, *Pattern Languages of Program Design*. Addison-Wesley, 1995.

[Fayad and Cline, 1996] Mohamed Fayad and Marshall P. Cline. Aspects of Software Adaptability. *Communications of the ACM*, 39(10):58–59, October 1996.

[Foote and Opdyke, 1995] Brian Foote and William F. Opdyke. Lifecycle and Refactoring – Patterns That Support Evolution and Reuse. In James O. Coplien, editor, *Pattern Languages of Program Design*. Addison-Wesley, 1995.

[Gamma *et al.*, 1993] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Abstraction and Reuse in Object-Oriented Designs. In O. Nierstrasz, editor, *Proceedings of ECOOP'93*, Berlin, 1993. Springer-Verlag.

[Gamma *et al.*, 1995] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements od Reusable Object–Oriented Software*. Addison–Wesley Professional Computing Series. Addison–Wesley Publishing Company, New York, 1995.

[Goldfedder and Rising, 1996] Brandon Goldfedder and Linda Rising. A Training Experience with Patterns. *Communications of the ACM*, 39(10):60–64, October 1996.

[Johnson, 1992] Ralph E. Johnson. Documenting Frameworks using Patterns. *ACM SIGPLAN Notices*, 27(10):63–76, October 1992. *OOPSLA '92 Proceedings*, Andreas Paepcke (editor).

[Kantor and Lapsley, 1986] Brian Kantor and Phil Lapsley. Network News Transfer Protocol. Request for Comments 977, Network Working Group, February 1986.

[Kerth, 1995] Norman L. Kerth. Caterpillar's Fate: A Pattern Language for the Transformation from Analysis to Design. In James O. Coplien, editor, *Pattern Languages of Program Design*. Addison-Wesley, 1995.

[Malone *et al.*, 1987] Thomas W. Malone, Kenneth R. Grant, Franklyn A. Turbak, Stephen A. Brobst, and Michael D. Cohen. Intelligent Information-Sharing Systems. *Communications of the ACM*, 30(5):390–402, May 1987.

[Manber and Wu, 1994] Udi Manber and Sun Wu. GLIMPSE: A Tool to Search Through Entire File Systems. In USENIX Association, editor, *Proceedings of the Winter 1994 USENIX Conference: January 17–21, 1994, San Francisco, California, USA*, pages 23–32, Berkeley, CA, USA, Winter 1994. USENIX.

[Pree, 1995] Wolfgang Pree. *Design Patterns for Object–Oriented Software Development*. ACM Press Books. Addison–Wesley Publishing Company, Wokingham, 1995.

[Press, 1992] Larry Press. Collective Dynabases. *Communications of the ACM*, 35(6):26–32, June 1992.

[Rumbaugh *et al.*, 1991] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Loensen. *Object-Oriented Modeling and Design*. Prentice-Hall International Editions, New York, 1991.

[Russell and Norvig, 1995] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Upper Saddle River, 1995.

[Schwartz, 1993] Randal L. Schwartz. *Learning Perl*. O'Reilley & Associates, Inc., Sebastopol, 1993.

[Wu and Manber, 1992] Sun Wu and Udi Manber. Fast Text Searching Allowing Errors. *Communications of the ACM*, 35:83–91, October 1992.

[Zimmer, 1995] Walter Zimmer. Relationships Between Design Patterns. In James O. Coplien, editor, *Pattern Languages of Program Design*. Addison-Wesley, 1995.