KfK 4662
Dezember 1989

Softwareentwicklung in einer verteilten Rechnerumgebung in Verbindung mit einer Objektverwaltung unter UNIX

H. Eggert, C. Düpmeier
Institut für Datenverarbeitung in der Technik

Kernforschungszentrum Karlsruhe

KERNFORSCHUNGSZENTRUM KARLSRUHE

Institut für Datenverarbeitung in der Technik

KfK 4662

Softwareentwicklung in einer verteilten Rechnerumgebung in Verbindung mit einer Objektverwaltung unter UNIX

H. Eggert, C. Düpmeier

KERNFORSCHUNGSZENTRUM KARLSRUHE GMBH, KARLSRUHE

Kernforschungszentrum Karlsruhe GmbH
Postfach 3640, 7500 Karlsruhe 1

ISSN 0303-4003

Zusammenfassung

Im Rahmen von *Software-Entwicklungs-Vorhaben*, insbesondere in *verteilten Entwicklungsumgebungen*, entstehen häufig Probleme, die mit dem Zugriff auf die produzierten *Objekte (Dokumente und Programme)* sowie mit deren Konsistenz zusammenhängen.

Software-Entwickler verlieren den Überblick, Schnittstellen zwischen Teilsystemen werden nicht vollständig berücksichtigt oder erst gar nicht definiert u.s.w.

Eine notwendige Vorraussetzung zur Behebung der o.g. Mängel ist die Bereitstellung einer Möglichkeit zur Einordnung aller *Objekte* in eine wohldefinierte *Objektverwaltungsstruktur*, welche neben der *Objektablage* auch Mechanismen zur *Objektabfrage (Retrieval)* sowie zur *Objektversionsführung* enthalten muß.

Diese Objektverwaltungsstruktur wird systematisch in einer geschlossenen Wirkungskette definiert.

Dabei werden zunächst bestimmten *Prinzipien und Methoden des Software-Engineering* disjunkte *Datenmengen* zugeordnet. Auf diesen Datenmengen wird eine *Relation* definiert, welche den logischen Zusammenhang zwischen den Datenmengen beschreibt.

Diese Relation wird schließlich mit *UNIX-Standard-Werkzeugen* umgesetzt und im Rahmen einer *Server/Client Lösung*, welche einer verteilten Entwicklungsumgebung genügt, realisiert.

Development of Software in a Distributed Environment in Connection with the Administration of Objects under UNIX

Abstract

In connection with the development of software especially in distributed environments big problems arise with respect to the administration of all the objects (documents and programs) developed.

Developers loose control over the quick growing set of objects, interfaces between the objects get inconsistent, parts of the system will not get completely specified or implemented etc.

As a necessary precondition to solve the above problems you have to provide a possibility to integrate your objects into a well-defined object management scheme which should - beside mechanism to save and retrieve objects - also include some concepts, like version management, to preserve the integrity of objects and the relations between objects.

In this paper we will define such an object management scheme in a concise way on the basis of well-known principles of software engineering.

First we will assign certain sets of data to several principles of software engineering. These sets will define classes of objects, which will be created in a software engineering process, and the relations between the object classes.

The management of object classes and their relations are implemented as a client/server system with central object management under UNIX and local object input and output under VMS, MS-DOS and UNIX. Only standard tools are used for the implementation of the system.

INHALTSVERZEICHNIS

1 Einleitung

Im IDT werden Softwareentwicklungen durchgeführt, die den Bereich der Prozeßautomatisierung betreffen (z.B. Prozeßüberwachung, rechnergestützte Fernhantierung). Im Rahmen dieser Vorhaben werden auch Verfahren der Mustererkennung, Bildanalyse, künstlichen Intelligenz sowie formale Softwarespezifikationen angewendet.

Aufgrund dieser Anwendungsvielfalt ist unsere Rechnerumgebung von den folgenden Merkmalen geprägt:

- Wir betreiben verschiedene Rechnertypen (z.B. Workstations, Personalcomputer, LISP-Maschinen, Minirechner im Timesharingbetrieb),
- Diese Rechner arbeiten unter verschiedenen Betriebssystemen (z.B. verschiedene UNIX-Derivate [1], VMS [2], MS-DOS [3], Genera [4]).

Im Rahmen unserer Softwareentwicklung werden verschiedene Vorhaben parallel abgewickelt (in der Regel mit unterschiedlichen Rechnertypen), wobei auch durchaus in demselben Vorhaben verschiedene Rechnertypen in vernetzter Form eingesetzt werden.

Diese Inhomogenität über unterschiedliche Vorhaben hinweg aber auch innerhalb eines Vorhabens erzwingt im Rahmen der *Softwarequalitätssicherung* ein Mindestmaß an gemeinsamer Struktur, um eine Identifizierungsmöglichkeit der im Softwareentwicklungsprozeß erzeugten *Objekte (Dokumente und Programme)* sowie eine notwendige Konsistenz (z.B. zwischen Quellprogramm und dazugehöriger Dokumentation) sicherzustellen.

Diese gemeinsame Struktur, ohne Einschränkung der jeweils benötigten Flexibilität und Offenheit, läßt sich über die Festlegung und Umsetzung einiger weniger *Prinzipien und Methoden des Softwareengineering* erreichen, wobei wir u.a. nach den folgenden Prinzipien vorgehen:

(1) *Prinzip der Softwareentwicklung in Schritten mit jeweils überlagerten Richtlinen zur Qualitätssicherung*

(2) *Prinzip der Modularisierung*

(3) *Prinzip der Versionen*

Dem ersten Prinzip haben wir die folgende Methode zugeordnet:

- *Vorgehensweise nach einem Phasenmodell, welches um ein Prototypingmodell erweitert wurde und überlagerte Richtlinien zur Qualitätssicherung enthält.*

Dem zweiten Prinzip haben wir die folgenden zwei Methoden zugeordnet:

- *Zerlegung eines Systems so, daß Teilsysteme entstehen,*
- *Zerlegung einer Funktion so, daß Teilfunktionen entstehen.*

Dem dritten Prinzip haben wir die folgende Methode zugeordnet:

- *Die im Source Code Control System (SCCS) von UNIX* (siehe Rochkind1975a]) *verwendete Methode (Verfahren) zur Versionshaltung.*

Mit Hilfe der vorher genannten Methoden lassen sich vorhabensabhängige Datenmengen definieren. Unter Hinzunahme weiterer vorhabensabhängiger Datenmengen läßt sich eine *Relation* definieren, die es

---

[1] UNIX ist ein Warenzeichen von AT&T

[2] VMS ist ein Warenzeichen von Digital Equipment Corporation

[3] MS-DOS ist ein Warenzeichen von Microsoft

[4] Genera ist ein Warenzeichen von Symbolics

erlaubt, jedes bei der Softwareentwicklung produzierte *Objekt* eindeutig zu kennzeichnen und damit zu identifizieren.

Der problemlose Zugriff auf ein Objekt (z.B. zur Modifikation) setzt eine Objektverwaltung mit integrierter Versionsführung voraus.

Im Hinblick auf unsere verteilte und inhomogene Rechnerwelt war es sinnvoll, ein dazu erforderliches *Objektverwaltungssystem (OVS)* mit einem zentralen *Objektverwaltungs Server* und mit dezentralen *Objektverwaltungs Ein/Ausgabeeinheiten als Clients* zu realisieren.

Im Sinne einer *offenen Softwareentwicklungsumgebung* (siehe [Brooks1987a]) verbleibt auf diese Weise die Objekterstellung einschließlich der OVS Ein/Ausgabe in der jeweils vertrauten Entwicklungsumgebung. Der in Relation zur Ein/Ausgabe wesentlich aufwendigere eigentliche Verwaltungsdienst mußte jedoch nur einmal entwickelt werden.

Die technische Vorraussetzung für eine solche verteilte Lösung ist ein *LAN (Local Area Network)*, welches bei uns mit dem *IEEE 802.3-Standard (Ethernet)* realisiert wurde. Weiterhin setzen wir zur Kommunikation die *TCP/IP-Protokollfamilie (Transmission Control Protocol/Internet Protocol)* ein (siehe [Comer1988a]).

Der Server des OVS wurde von uns unter *UNIX* realisiert. Dabei werden die Objekte unter dem *UNIX-Dateisystem* so abgelegt, daß die Ablagestruktur sich an der oben angegebenen Relation *Schlüssel* orientiert. Die *Versionsbehandlung* wird in integrierter Weise durch das *UNIX Source Code Control System (SCCS)* durchgeführt. Zusätzlich wird innerhalb des Servers ein *Objektverzeichnis (Dictionary)* bereitgestellt, welches neben dem Objektschlüssel noch die Zusätze Versionsnummer, Freigabedatum, Autorennamen und Objekttitel enthält. Weiterhin werden durch den Server flexible Suchmechanismen bereitgestellt, welche mit Hilfe der *Regulären Ausdrücke (regular expressions)* des UNIX Systems realisiert wurden (siehe [Aho1974a, Aho1988a]).

Die Ein/Ausgabe von Objekten wird über ein spezielles Applikationsprotokoll unterstützt. Der OVS Server regelt die *Zugriffsrechte (Datenschutz)* und sorgt beim Parallelzugriff mehrerer Clients für eine *Synchronisation (Datenkonsistenz)*.
Alle Funktionen des OVS Servers wurden ausschließlich mit UNIX Standard Hilfsmitteln realisiert, so daß eine Portierung auf andere UNIX-Rechner problemlos möglich ist.

2 Verwendte Terminologie des Softwareengineering

Zur Abgrenzung und zum besseren Verständnis unserer Sichtweise im Rahmen des *Softwareengineering* soll zunächst eine *Terminologie* von den Teilen des Softwareengineering eingeführt werden, die wir hier benötigen.

2.1 Prinzipien, Methoden, Verfahren, Werkzeuge, Techniken

Der Inhalt dieses Unterkapitels orientiert sich weitgehend an [Balzert1982a].

Zur Beschreibung, auf welche Art und Weise und mit welchen Mitteln Software Systeme entwickelt werden, hat sich eine Menge von Begriffen eingebürgert. Um von eindeutigen Begriffen ausgehen zu können, werden die wichtigsten Begriffe definiert. Es werden jedoch möglichst wenig Begriffe verwendet, um verständlich zu bleiben. Auf eine starke Differenzierung wird daher verzichtet.

(1) *Prinzipien* sind Grundsätze, die man seinem Handeln zugrundelegt.

Prinzipien sind allgemeingültig, abstrakt, allgemeinster Art. Sie bilden eine Grundlage. Prinzipien werden aus Erfahrung und Erkenntnis hergeleitet und durch sie bestätigt.

Beispiele:

- Prinzip der Hierarchisierung,
- Prinzip der Modularisierung,
- Prinzip der Standardisierung,
- Prinzip der Abstraktion,
- Prinzip der Strukturierung,
- Geheimnisprinzip.

Prinzipien sagen nichts darüber aus, wie man -bezogen auf ein konkretes Anwendungsgebiet- zu ihnen gelangt. Außerdem sind sie weitgehend unabhängig vom Anwendungsgebiet. Das Prinzip der Abstraktion wird z.B. in der Philosophie ebenso angewandt wie in der Mathematik, in der Informatik usw.

(2) *Methoden* sind planmäßig angewandte, begründete Vorgehensweisen zur Erreichung von festgelegten Zielen.

Methoden enthalten also den Weg zu etwas hin, d.h. sie machen Prinzipien planmäßig anwendbar. Planmäßig bedeutet, daß beim Einsatz einer Methode nicht herumprobiert wird. Methoden sollen konkret sein, im Sinne einer Wegleitung, einer Aufteilung in Arbeitsschritte. Anwendung und Erfolg einer Methode sollen verifizierbar und meßbar sein.

Die folgende Tabelle zeigt einige Beispiele:

PRINZIP	METHODE
Hierarchisierung	Zerlegen eines Problems in Teilprobleme so, daß eine Baumhierarchie entsteht.
Modularisierung	Enwicklung von Produkten oder Teilprodukten, die nur über eine definierte Schnittstelle mit der Umwelt kommunizieren können und selbst kontextunabhängig sind.
Strukturierung	Entwurf von Programmen so, daß nur Auswahl und Wiederholung vorkommen.

Tab. 2.1–1 : Beispiele für die Abhängigkeit von Prinzipien und Methoden

Methoden sind anwendungsneutral. Die Kennzeichen eines Moduls treffen sowohl auf ein elektronisches Bauteil (Modul in einem Fernsehgerät) als auch auf einen Programmbaustein zu.

(3) *Verfahren* beinhalten meist formale Vorschriften und bilden den Inhalt von Standards.

Bei der Entwicklung von Verfahren entstehen manchmal als Verallgemeinerung der gewonnenen Erkenntnisse Methoden. Verfahren beschreiben konkrete Wege zur Lösung bestimmter Probleme oder Problemklassen. Ein Verfahren kann methodisch sein, ist aber i.a. wegen seiner Beschränkung stärker einsatzbezogen als eine Methode.

Beispiel:

Methode

Anwendung der Kontrollstrukturen *Sequenz*, *Auswahl* und *Wiederholung*.

Verfahren

Bei der Entwicklung von PASCAL-Programmen dürfen nur die Kontrollstrukturen *begin-end*, *if-then*, *if-then-else*, *case-of*, *repeat-until*, *while-do*, *for-to* verwendet werden.

Da die Abgrenzung zwischen Methoden und Verfahren oft schwierig ist, werden in der Regel die Begriffe Methode und Verfahren synonym verwendet.

(4) *Werkzeuge bzw. Hilfsmittel* dienen der Unterstützung von Methoden und Verfahren.

Werkzeuge werden eingeführt, um den Einsatz von Methoden und Verfahren zu erleichtern, zu beschleunigen und abzusichern.

Beispiel:

Verfahren

Festgelegte Kontrollstrukturen in PASCAL (s. vorhergehendes Beispiel).

Werkzeuge bzw. Hilfsmittel

Überprüfung durch einen Präprozessor, ob keine *label-Vereinbarung* und kein *goto* verwendet wurde.

Durch automatisierte Werkzeuge kann die Einhaltung von Verfahren und Standards erzwungen und die Produktivität erhöht werden.
Insbesondere sollten solche Tätigkeiten automatisiert werden, die hohe Präzision erfordern, die oft wiederholt werden müssen und die eine Überprüfung erfordern.

Eine Sammlung von Werkzeugen bezeichnet man auch als *Werzeugkasten* oder *Werkzeugbank.*

(5) *Technik* ist die konsistente Anwendung von Prinzipien, Methoden und Werkzeugen.

Häufig wird der Begriff Technik auch als reiner Sammelbegriff verwendet.

2.2 Qualitätssicherung

Die Sicherstellung einer geforderten *Software Qualität* eines *Software Produktes* kann nur durch eine *entwicklungsbegleitende Qualitätssicherung (QS)* erreicht werden. Dazu wird eine Terminologie zur Software Qualität eingeführt.

Software Qualität wird über zwei *Merkmalsklassen* definiert:

(1) *Merkmalsklasse mit direkter Beziehung zum Software Produkt*

Beispiele:

- Hierarchisierungs Prinzip
- Modularisierungs Prinzip

(2) *Merkmalsklasse mit indirekter Beziehung zum Software Produkt*

Beispiele:

- Dokumentation (entwicklungsbegleitend)
- Überprüfungsmöglichkeit der Funktionalität durch eine Testumgebung

Die o.g. Beispiele aus den beiden Merkmalsklassen sind *qualitative Eigenschaften (qualitative Merkmale).*

Um im Prozeß der Qualitätssicherung die Qualitätsmerkmale auch anwenden (umsetzen) zu können, sind zusätzlich *quantitative Eigenschaften (quantitative Merkmale)* erforderlich, da nur so eine *Meßbarkeit (Software Metrik)* erreichbar ist.

Beispiele:

- Definition einer Dokumentationsstruktur
- Definition von Testfällen

Die Menge aller definierten Qualitätsmerkmale sind ein Maß für die *Güte* des gewünschten Software Produktes.

Die Umsetzung der definierten Qualitätsmerkmale heißen *Maßnahmen zur Qualitätssicherung.*

Man unterscheidet dabei drei Typen:

(1) *Konstruktive Maßnahmen* (diese dienen zur *Fehlerverhütung*)

Beispiele:

- Methoden der Strukturierten Programmierung
- Werkzeug zur Überprüfungsmöglichkeit von Quellcode (Präprozessor)

(2) *Analytische Maßnahmen* (diese dienen zur *Fehleraufdeckung*)

Beispiele:

- übersichtlich strukturierter Quellcode
- Testumgebung

(3) *Organisatorische Maßnahmen* (diese dienen zur *organisatorischen Durchführung der Qualitätssicherung*)

Dabei gilt nach unserer Auffassung der Grundsatz, daß organisatorische Maßnahmen in möglichst engen Grenzen gehalten werden und diese Maßnahmen werkzeugunterstützt ablaufen sollten.

Z.B. können bei der rechnergestützten Erstellung eines Dokuments bestimmte Organisationsstrukturen des Dokuments von einem Automaten gelesen werden, welcher anschließend daraus einen Eintrag in ein Objektverzeichnis (Dictionary) vornimmt sowie das Objekt selbst unter einer wohldefinierten Ablagestruktur (Dateiverzeichnis) eines Dateisystems speichert.

3 Prinzip der Softwareentwicklung in Schritten

Wir ordnen diesem Prinzip methodisch eine Vorgehensweise zu, bei der ein Phasenmodell um ein Prototypingmodell erweitert wird und bei der weiterhin Richtlinien zur Qualitätssicherung definiert werden, die den Phasen jeweils überlagert sind.

Die Richtlinien zur Qualitätssicherung wollen wir hier jedoch nicht weiter konkretisieren, da diese für unser eigentliches Thema "Objektverwaltung unter UNIX" nur von formaler Bedeutung sind.

3.1 Software-Entwicklung nach Phasenmodellen

Allen heute beschriebenen Phasenmodellen (auch Life Cycle Modelle) liegt die gleiche idealisierte Vorstellung zugrunde:

Die beschriebenen Elemente, also die verschiedenen Aktivitäten und Ergebnisse (vom Entwurf bis zum Einsatz eines Software Systems), als in sich abgeschlossene Einheiten, können zeitlich geordnet werden und der sich so ergebende "Zyklus" soll im Idealfall je Entwicklungsschritt nur einmal durchlaufen werden.

Die folgende Abbildung zeigt ein derartiges *Life-Cycle Modell:*

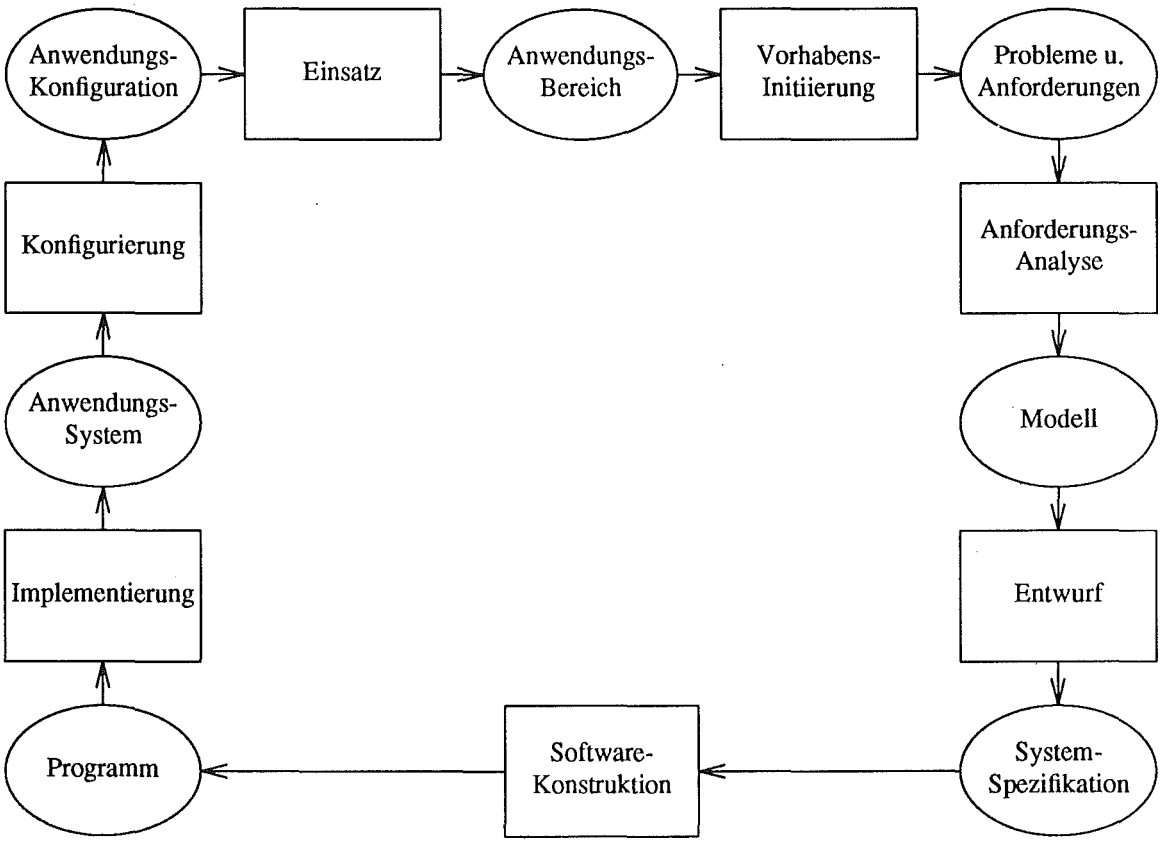

Abb. 3.1–1 : Die Grundelemente der Systementwicklung in einem Netz

Die Praxis hat jedoch gezeigt, daß sich Softwaresysteme nicht rein sequentiell konstruieren lassen. Das gilt übrigens für jede Systementwicklung und ist aus den klassischen Ingenieurdisziplinen wohlbekannt.

Daher werden Iterationen über die einzelnen Aktivitäten bei der Software Entwicklung durchgeführt, die eine Prüfung der Eingangs- und der Ausgangsdokumente (z.B. Machbarkeitsstudie, Inspektionen, Modultest u.a.) enthalten.

Mit diesen Konzepten werden durchaus gute Ergebnisse erzielt. Sie sind aber häufig mit großem zusätzlichen Organisationsaufwand verbunden. Vom Softwareentwickler wird dieser zusätzliche Organisationsaufwand oft als lästiger Formalismus empfunden, kann zu Frustrationen führen und wird daher häufig gezielt unterlaufen (siehe [Brooks1987a]).

Was kann also besser gemacht werden, mit dem Ziel einer als "vernünftig" empfundenen Kosten/Nutzen-Relation für Software Systeme?

3.2 Software Entwicklung mit Konzepten des Prototyping

Die folgenden Inhalte zu den *Konzepten des Prototyping* orientieren sich an [Budde1987a].

Evolutionäre Systementwicklung stellt die Grundelemente der Systementwicklung in eine veränderte Beziehung zueinander.

- Spezifikation und Implementierung werden als verschiedene, sich ergänzende *Tätigkeiten* betrachtet, die nicht in unterschiedliche Arbeitsabschnitte separiert sein dürfen.
- Zu den einzelnen Modellen bei der Software Entwicklung werden soweit wie möglich *Prototypen* konstruiert, um die Bewertung ihrer Angemessenheit zu erleichtern.
- Die Kommunikation zwischen Entwicklern und Benutzern findet während des *gesamten Vorhabens* statt und ist nicht nur auf die Phasen der Bedarfsanalyse und Systemeinführung beschränkt.
- Systementwicklung wird als *Lernprozeß* aller Beteiligten aufgefaßt und nicht als mehr oder minder formale Transformation einer gegebenen Systemspezifikation in ein Zielsystem.
- Ein Gesamtsystem wird in *kleinen Schritten* und in überschaubaren Einzelteilen konstruiert, um nach minimalem Aufwand jeweils die weitere Entwicklungsrichtung bestimmen zu können.

3.3 Prototyping innerhalb einer evolutionären Strategie

Im Rahmen einer evolutionären Entwicklungsstrategie verstehen wir unter einem Prototyp:

- Ein Prototyp ist ein ablauffähiges Modell des Zielsystems; er realisiert bestimmte *Aspekte* des zukünftigen Systems.
- In der Kommunikation zwischen Entwicklern, Benutzern und Management sind Prototypen ein *anschaulicher Diskussionsgegenstand.* Sie werden ggf. durch schriftliche Systembeschreibungen ergänzt.
- Jeder Prototyp dient als *Spezifikation* des nächsten Prototyps oder des Zielsystems.
- Das Zielsystem sollte an der Benutzerschnittstelle nach Funktionalität und Verhalten weitgehend dem letzten Prototyp entsprechen.

Ein Prototyp wird also iterativ *konstruiert* und *bewertet.* Die folgende Abbildung zeigt eine entsprechende Abstraktion:

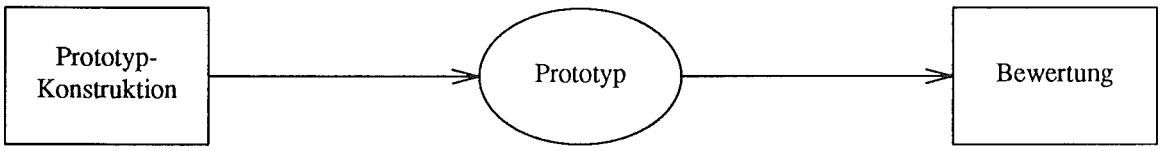

Abb. 3.3–1 : Prototyping

Im Prinzip können drei Systemmodelle zu Prototypen führen. Diese drei Modelle sind mit bestimmten Fragestellungen verbunden, d.h. bei der Modellierung stehen bestimmte Aspekte des Systems im Vordergrund.

Im folgenden unterscheiden wir entsprechend drei Arten von Prototypen und ordnen ihnen drei Aspekte des Prototyping zu.

(1) *Exploratives Prototyping*

Von einem *Prototyp im engeren Sinne* sprechen wir dann, wenn die Problem- u. Anforderungsdefinition des Informationssystems durch ein ablauffähiges vorläufiges Softwaresystem ergänzt wird. Diese Art von Prototypen gibt den Anwendern und Benutzern eine "greifbare" Vorstellung über die angestrebte Problemlösung und unterstützt dadurch die Bewertung der Problem- u. Anforderungsdefinition. Für den Entwickler sind solche Prototypen eine *ablauffähige Spezifikation der Benutzeranforderungen.* Sie

erleichtern die Ausarbeitung eines Modells des Softwaresystems und reduzieren den Interpretationsspielraum bei der Software-Konstruktion.

Prototyping dient hier der Klärung der *Problemstellung*. Veränderte Arbeitsinhalte stehen genauso zur Diskussion, wie die Frage nach dem Umfang und der Art der DV-Unterstützung. Dabei sollte Wert auf unterschiedliche Lösungskonzepte gelegt werden, um den Denkhorizont nicht verfrüht auf einen Ansatz einzuengen. Die Entwickler erhalten bei dieser Vorgehensweise einen Einblick in den Anwendungsbereich.

Erweitert man nun das vorher gezeigte Life-Cycle Modell um dieses Prototyping-Modell, so zeigt die folgende Abbildung den Zusammenhang:

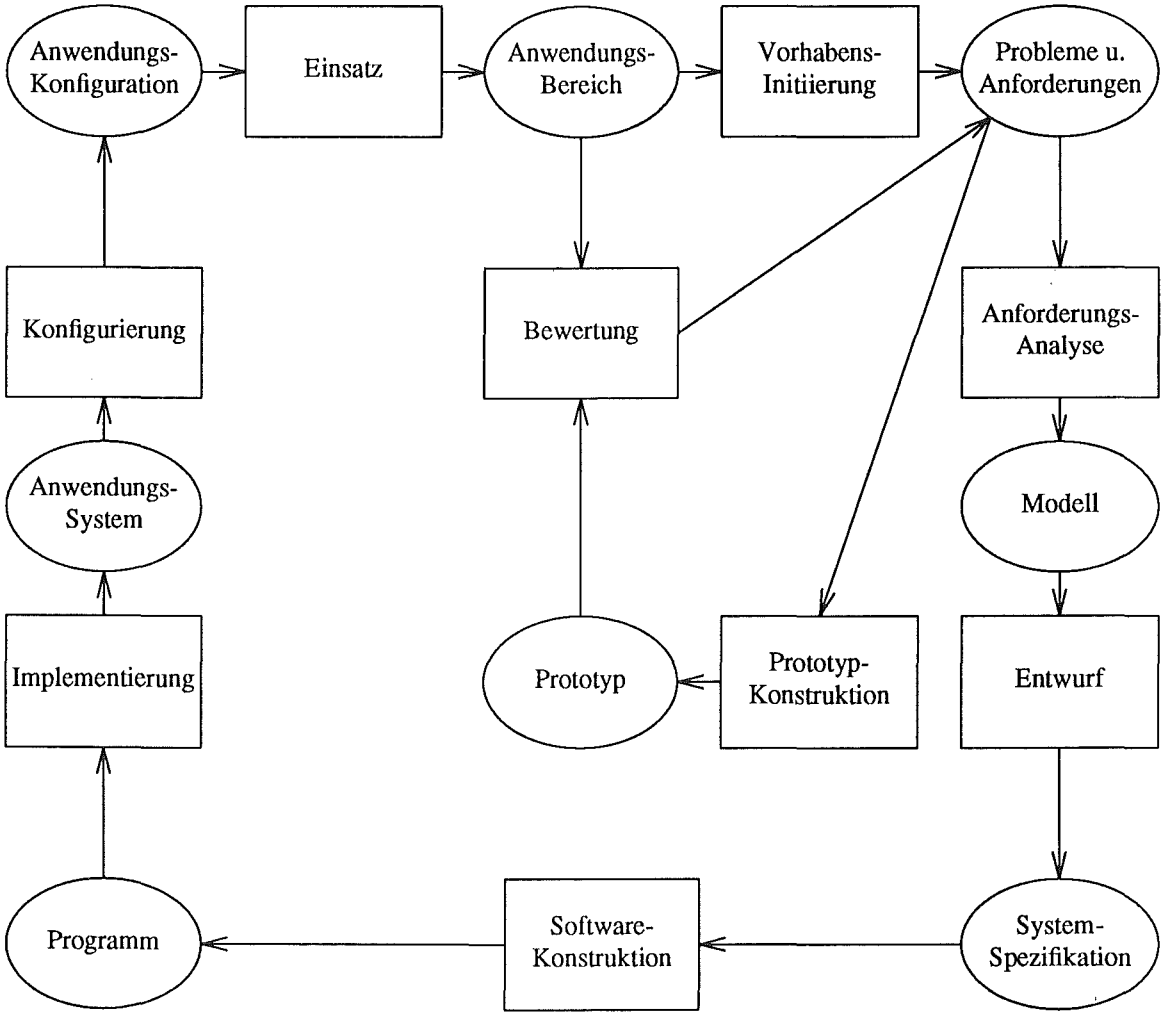

Abb. 3.3–2 : Exploratives Prototyping

(2) *Experimentelles Prototyping*

Dient ein Prototyp vorwiegend der Diskussion in der Gruppe der Entwickler und wird zu Fragen der technischen Realisierung herangezogen, so sprechen wir von einem *Labormuster*.

Seine experimentelle Erprobung zeigt die Dynamik und die Restriktionen des Modells und läßt Rückschlüsse auf das Verhalten des Zielsystems zu, ohne daß dieses bereits durchkonstruiert und installiert ist.

Der betrachtete Aspekt ist die technisch-konstruktive Umsetzung eines Entwicklungsziels. Einerseits sollen die Benutzer im Experiment ihre Vorstellungen von der DV-Lösung weiter detaillieren, andererseits erhalten die Entwickler eine Grundlage für die Einschätzung der Machbarkeit und Angemessenheit eines Anwendungssystems. Die Kommunikation zwischen Benutzern und Entwicklern über technische und software-ergonomische Fragen steht im Vordergrund.

Es geht also um die Erprobung und Bewertung einer technischen Lösung.

Die folgende Abbildung zeigt den Zusammenhang:

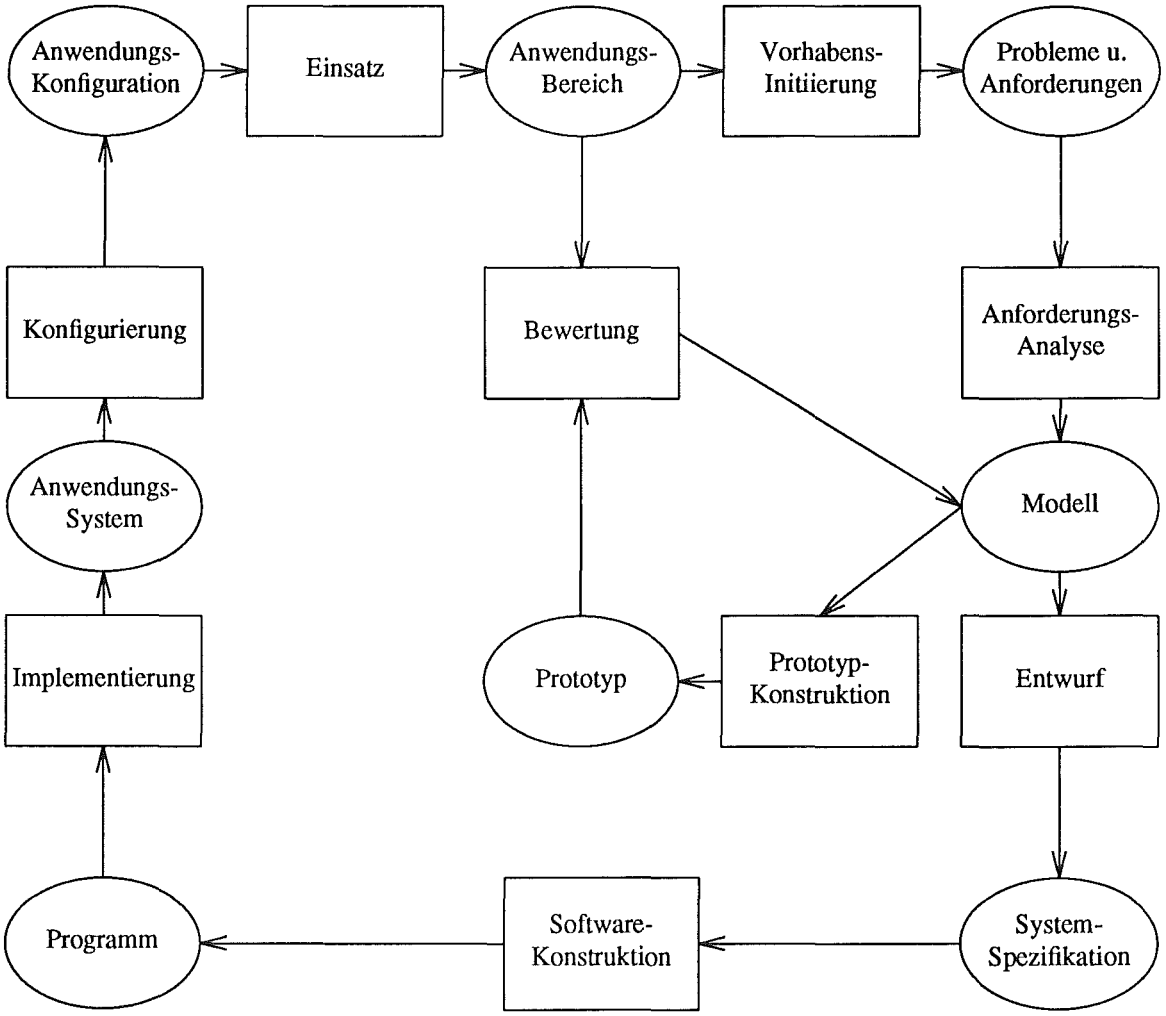

Abb. 3.3–3 : Experimentelles Prototyping

(3) *Evolutionäres Prototyping*

Wird ein Prototyp nicht nur zu Experimenten oder als "Anschauungsmaterial" verwendet, sondern bereits als Kern des Zielsystems im Anwendungsbereich eingesetzt, so bezeichnen wir ihn als *Pilotsystem.*

Der Einsatz von Pilotsystemen hilft die organisatorische Installation des Zielsystems weit besser vorzubereiten als das bei einem "Wegwerf"-Prototypen möglich ist. Die Benutzer erhalten nicht nur einen "greifbaren" Eindruck eines Prototyps in einer Laborumgebung (Rechnerfeld), sie können mit echten Daten in ihrer Arbeitsumgebung arbeiten. Dies erleichtert die Auswertung des Systems und minimiert

den Aufwand an rein "experimenteller" Arbeit.

Konsequent angewendet bedeutet dies, daß Prototyping nicht nur als Hilfsmittel innerhalb eines einzelnen Entwicklungszyklus eingesetzt wird, sondern zur kontinuierlichen Anpassung eines DV-Systems an sich rasch verändernde Randbedingungen dient. Eine derartige Systementwicklung ist ein Prozeß, der die Anwendung begleitet.

Die folgende Abbildung zeigt den Zusammenhang:

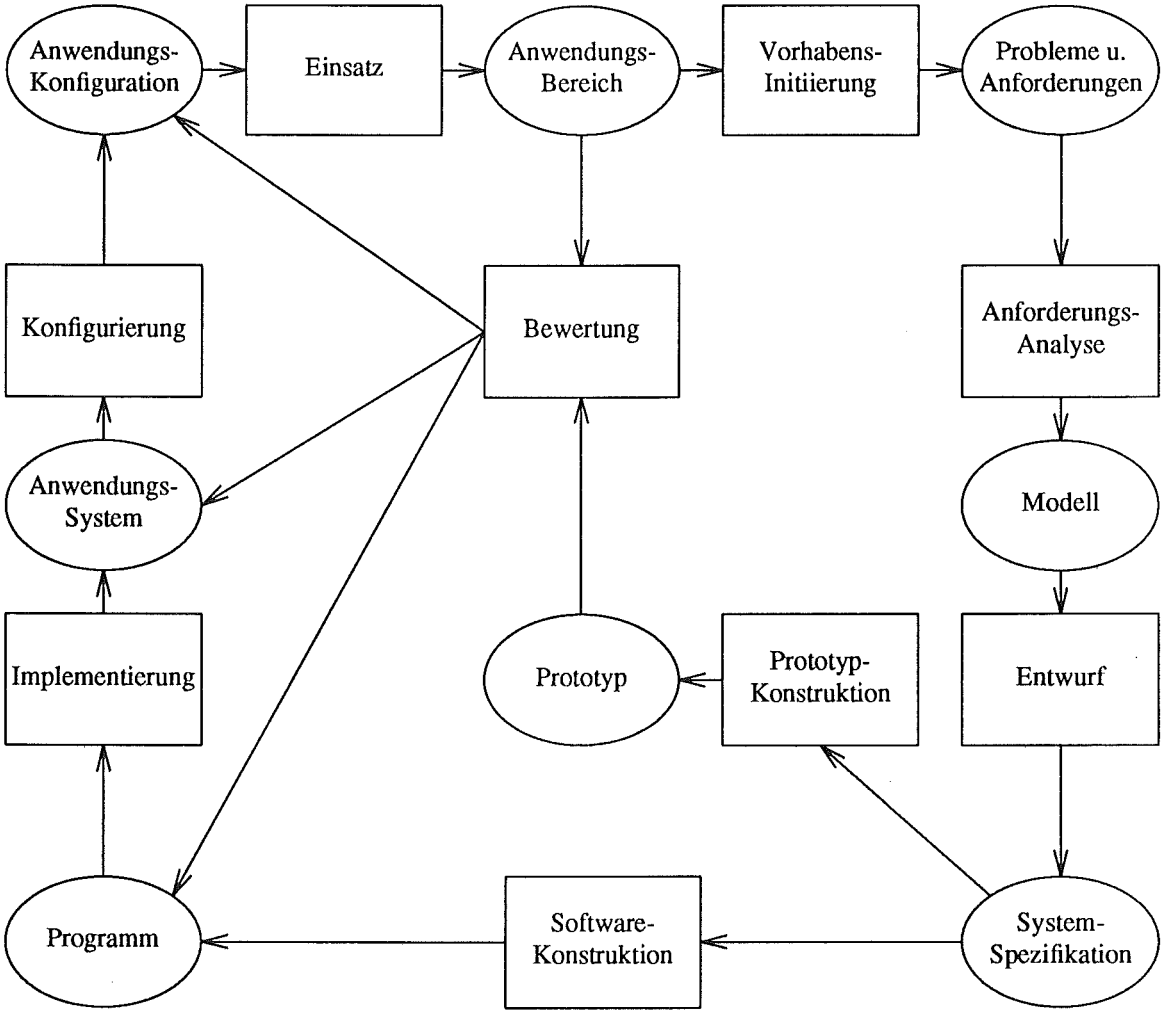

Abb. 3.3–4 : Evolutionäres Prototyping

4 Prinzip der Modularisierung

Unter einem Modul verstehen wir eine funktionelle Einheit mit bekannten Eigenschaften. Systeme sollen modular aufgebaut sein, damit einzelne, miteinander verbundene Systemkomponenten entfernt, hinzugefügt oder durch andere ersetzt werden können.

Rechnersysteme erlauben z.B. das Ersetzen und Hinzufügen von Speichermodulen. Im Softwarebreich verstehen wir unter einem Modul ein getrennt übersetzbares Programmstück zusammen mit den von ihm benutzten Daten (siehe [Sethi1989a]).

Wir ordnen dem Prinzip der Modularisierung die folgenden beiden Methoden zu:

- *Zerlegen eines Gesamtsystems so, daß Teilsysteme entstehen*
- *Zerlegen einer Gesamtfunktion so, daß Teilfunktionen entstehen*

Im Sinne einer offenen Softwareentwicklung ist es nicht mehr sinnvoll hier weitere Angaben zu machen. Es bleibt der jeweiligen Sicht des Softwaresystem Entwicklers überlassen, in welcher Weise er seine Modularisierung vornimmt. Für uns ist hier lediglich formal die Möglichkeit der Modularisierung von Bedeutung.

5 Prinzip der Versionen

Im Rahmen der Software Entwicklung entstehende Objekte (Dokumente und Programme) sind in der Regel von Änderungen und Erweiterungen betroffen (z.B. Fehlerbehebung, zusätzliche Funktion). Es wird also auf der inhaltlichen Basis eines Objektes eine neue Fassung dieses Objektes erstellt.

Wir gehen davon aus, daß dabei alle ursprünglichen Fassungen der Objekte erhalten bleiben und sprechen in diesem Kontext von *Versionen* des betroffenen Objektes.

Methodisch setzen wir dieses Prinzip der Versionen mit dem *Source Code Control System (SCCS)* von UNIX um (siehe Rochkind1975a]).

6 Eine relationale Struktur zur Kennzeichnung von Objekten

Mit Hilfe der vorher beschriebenen Prinzipien sowie den jeweils zugeordneten Methoden lassen sich nun formal *Datenmengen als Aufzählungstypen* beschreiben.

Der Methode, bei der ein *Phasenmodell um ein Prototypingmodell erweitert wird sowie eine phasenbezogene Zuordnung von Richtlinien zur Qualitätssicherung erfolgt*, wird die folgende Datenmenge zugeordnet:

(1) $Objekt\_Typ = \{QS, AB, PA, MO, SP, PR, AS, AK, PT\}$

Dabei bedeuten

QS: Qualitätssicherung
AB: Anwendungs Bereich
PA: Probleme und Anforderungen
MO: Modell
SP: System Spezifikation
PR: Programm
AS: Anwendungs System
AK: Anwendungs Konfiguration
PT: Prototyp

Bemerkung:

> Die Elemente QS und PT übernehmen hier Rollen, die als "schrittübergreifende" Merkmale zu verstehen sind.

Der Methode der *Zerlegung eines Systems in Teilsysteme* wird die folgende Datenmenge zugeordnet:

(2) *System = {***, TS1, TS2, ... , TSn}*

Dabei bedeuten

***: Leeres Element der Menge System
TS1: Teilsystem 1
TS2: Teilsystem 2
TSn: Teilsystem n

Bemerkung:

> Das Element *** übernimmt hier eine Rolle, die als "systemübergreifendes" Merkmal zu verstehen ist.

Der Methode der *Zerlegung einer Funktion in Teilfunktionen* wird die folgende Datenmenge zugeordnet:

(3) *Funktion = {***, TF1, TF2, ... , TFn}*

Dabei bedeuten

***: Leeres Element der Menge Funktion
TF1: Teilfunktion 1
TF2: Teilfunktion 2
TFn: Teilfunktion n

Bemerkung:

> Das Element *** übernimmt hier eine Rolle, die als "funktionsübergreifendes" Merkmal zu verstehen ist.

Der Methode des *Source Code Control Systems (SCCS)* wird die folgende Datenmenge zugeordnet:

(4) *Version = {1.1, 1.2, ..., n.1}*

Weiter lassen sich die folgenden Datenmengen definieren:

(5) *Vorhaben = {VO1, VO2, ..., VOn}*

Dabei bedeuten

VO1: Vorhaben 1
VO2: Vorhaben 2
VOn: Vorhaben n

(6) *Zähler = {1, 2, ... ,n}*

(7) *Freigabe_Datum = {D1, D2, ..., Dn}*

Dabei bedeuten

D1: Freigabe Datum 1
D2: Freigabe Datum 2
Dn: Freigabe Datum n

(8) *Autor = {N1, N2, ... , Nn}*

Dabei dedeuten

N1: Name 1
N2: Name 2
Nn: Name n

Wir sind der Auffassung, daß die oben beschriebenen acht Datenmengen eine *notwendige* aber auch *hinreichende* Basis für eine Objektkennzeichnung in einem Softwarevorhaben bilden.

Da diese Datenmengen voneinander isoliert keine Aussagekraft besitzen, wird die folgende Relation definiert:

Objekt $\subseteq$ *Vorhaben* $\times$ *System* $\times$ *Funktion* $\times$ *Objekt_Typ* $\times$ *Zähler* $\times$ *Version* $\times$ *Freigabe_Datum* $\times$ *Autor*

Eine zusätzliche Beschreibung des logischen Zusammenhangs zwischen den Datenmengen (*Assoziationen zwischen den Datenmengen*) führt zu sinnvollen Zusammenhängen.

Zwischen Datenmengen können drei Typen von Assoziationen existieren:

- *1:1 - Assoziation,*
- *1:n - Assoziation,*
- *m:n - Assoziation.*

In unserem Fall ist nur die m:n - Assoziation von Bedeutung. Dieser Typ hat die größte Komplexität, da hier im Gegensatz zu den anderen beiden Typen keine Abbildung (Funktion) zwischen zwei Datenmengen existiert.

Sie wird wie folgt definiert:

Zwischen zwei Datenmengen a und b besteht eine m:n - Assoziation, wenn folgendes gilt:

Jedem Element aus a sind entweder kein, ein oder mehrere Elemente aus b zugeordnet und umgekehrt.

Die folgenden Abbildungen zeigen diese Zusammenhänge im Rahmen unserer oben definierten Relation:

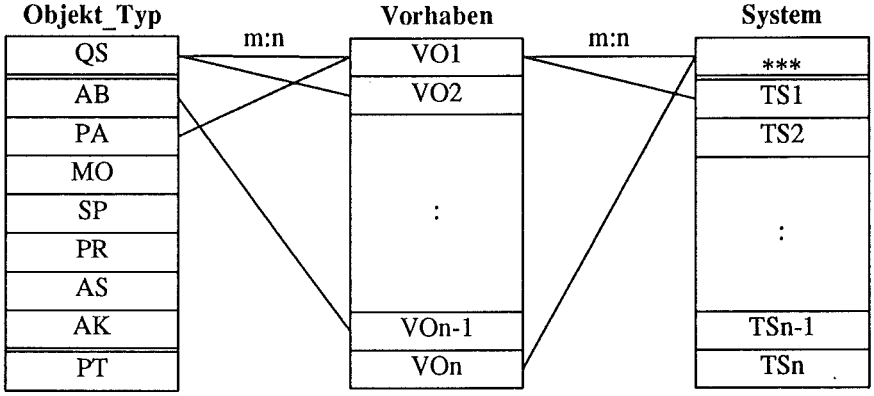

Abb. 6–1 : Objekt_Typ, Vorhaben, System

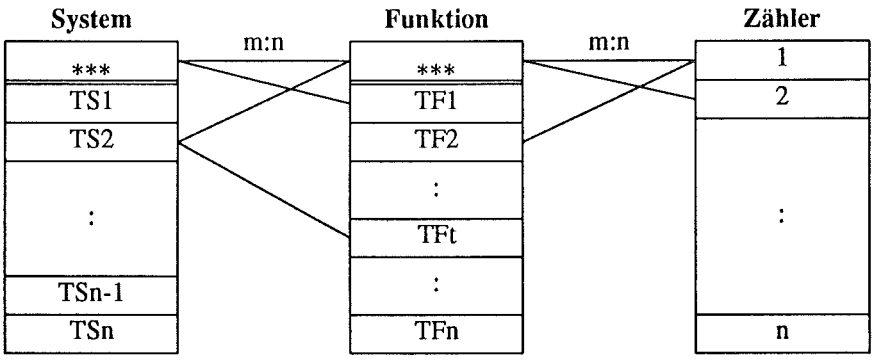

Abb. 6–2 : System, Funktion, Zähler

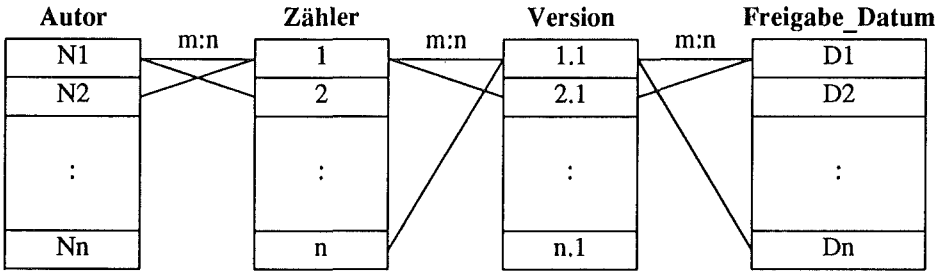

Abb. 6–3 : Autor, Zähler, Version, Freigabe_Datum

Unter Verwendung dieser Relation läßt sich jedes bei der Softwareentwicklung produzierte Objekt eindeutig kennzeichnen, was die Vorraussetzung für ein *rechnergestütztes Objektverwaltungs System (OVS)* ist.

Zu dem OVS gehören die folgenden Komponenten:

- *Ablage* (realisiert mit dem *UNIX Dateisystem*)
- *Versionsführung* (realisiert mit dem *Source Code Control System (SCCS)* von UNIX)
- *Suchmechanismen* (realisiert mit den *Reguären Ausdrücken (regular expressions) von UNIX*)
- *Verteilte Objekt Ein/Ausgabe* (realisiert mit *IEEE 802.3-Standard (Ethernet)* und *TCP/IP-Protokollfamilie (Transmission Control Protocol/Internet Protocol)*)

Es wurden also für die Realisierung, die in den folgenden Kapiteln beschrieben wird, ausschließlich *UNIX Standard Hilfsmittel* verwendet, so daß eine problemlose *Portierung* des OVS auf andere UNIX Systeme sichergestellt ist.

7 Eine Objektverwaltungsstruktur unter UNIX

Da bei der Implementierung des *OVS* auf dem jeweiligem Betriebssystem nur standardmäßig vorhandene Werkzeuge genutzt werden sollten, wurde auf eine Implementierung der *Objektverwaltungsstruktur* unter der Verwendung eines Datenbanksystems verzichtet und das hierarchische Dateisystem von UNIX zur Ablage der Objekte ausgenutzt.

Dies hat aber auch noch andere Gründe. Die zur Zeit erhältlichen Datenbanksysteme sind in der Regel satzorientiert und eignen sich daher nicht zur Verwaltung großer Dateneinheiten. Weiter fehlen Datenbanksystemen in der Regel Konzepte der *Versionsführung* der gespeicherten Daten, die zur Verwaltung der in einem Softwareentwicklungsvorhaben erstellten Objekte unbedingt erforderlich sind (siehe [Ullman1986a]).

Die Mächtigkeit eines Datenbanksystems und seiner Abfragesprache ist auch gar nicht erforderlich, um eine effektive und komfortable Verwaltung von allen im Softwareentwicklungsprozeß anfallenden Objekten zu gewährleisten, da unter dem Betriebssystem UNIX schon einfach zu benutzende Werkzeuge vorhanden sind, die eine komfortable Abfrage (Retrieval) erlauben.

Objekte werden vom *Server des OVS* in Teilbäume des UNIX Dateisystems, sogenannte *Ablagen*, eingetragen, wobei eine *Versionsführung* mit dem unter UNIX vorhandenen Werkzeug *SCCS* (*Source Code Control System*) erfolgt (siehe [Rochkind1975a]). Um die in den vorherigen Kapiteln beschriebenen *Methoden* zur Softwareentwicklung zu unterstützen, ordnet das *OVS* jedem Objekt *Verwaltungsdaten* aus den unter Kapitel 6 eingeführten Datenmengen zu.

Insbesondere ist jedem Objekt ein *Objektschlüssel* aus Bestandteilen zugeordnet, die die Vorhabens-, Teilsystem-, Teilfunktionszugehörigkeit und den Objekttyp beschreiben. Der *Schlüssel* faßt damit die Informationen (1), (2), (3), (5) und (6) von Kapitel 6 zusammen.

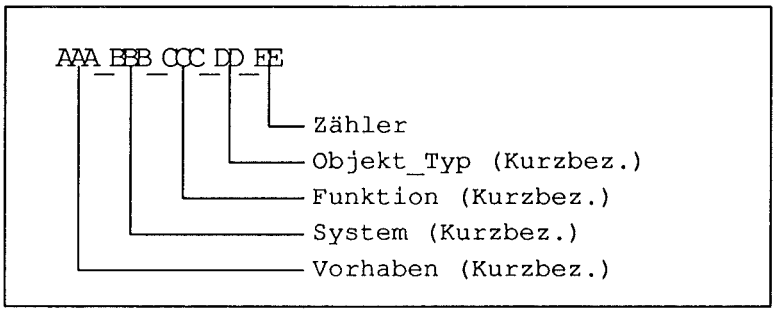

Abb. 7–1 : Aufbau des Objektschlüssels

Ein jeder *Ablage zugeordnetes Objektverzeichnis* (*Dictionary*) speichert alle *Verwaltungsdaten* eines Objektes und bietet im Zusammenhang mit verschiedenen UNIX Werkzeugen, die *reguläre Ausdrücke* zur Textsuche verwenden, umfangreiche Möglichkeiten zur Suche nach bereits eingetragenen Objekten (siehe [Aho1974a, Aho1988a, Kernighan1976a, Lesk1975a]). Das *Objektverzeichnis* einer Ablage realisiert die in Kapitel 6 beschriebene Relation *Objekt*.

Die Struktur einer Ablage und die Versionsführung über *SCCS* wollen wir im folgenden Unterkapitel näher erläutern. Anschließend werden wir auf die Struktur der Objektverzeichnisse eingehen.

7.1 Objektablage und Versionsführung

Im *OVS* werden die Benutzer in einzelne *Gruppen* eingeteilt, wobei jeder Benutzer durchaus verschiedenen *Gruppen* angehören kann. Jeder *Gruppe* ist eine eigene *Ablage* zugeordnet, die durch ein *Paßwort* nur den Mitgliedern einer *Gruppe* Zugriff auf die in ihr enthaltenen Objekte erlaubt. Die *Gruppen*, die das System kennt, lassen sich dabei während der Laufzeit des Systems frei konfigurieren.

Die *Ablage einer Gruppe* stellt einen Teilbaum des UNIX Dateisystems dar, der an einer beliebigen Stelle im UNIX Dateisystem eingehängt werden kann. Der Zugriff auf diesen Teilbaum wird durch Wahl der Zugriffsrechte für den *Wurzelkatalog* des Baumes auf den *Server des OVS* und eine oder mehrere *Administrationspersonen* beschränkt.

Zur Bestimmung des eindeutigen Ablageortes eines Objektes wird der *Objektschlüssel*, der jedem Objekt zugeordnet ist, verwendet. Die Bestandteile dieses *Schlüssels* bestimmen, ausgehend von der Wurzel der Ablage, den *Pfadnamen* des Ablagekataloges und den *Dateinnamen* der Datei, unter dem das Werkzeug *SCCS* das Objekt im Ablagekatalog anspricht. Die *SCCS* Kommandos verwalten die *Versionen* eines Objektes in einem speziellen *SCCS* Katalog innerhalb einer spezifisch aufgebauten Datei, die man *s-Datei* nennt. Auf Grund der *Eindeutigkeit des Objektschlüssels* wird durch dieses Verfahren ein *eindeutiger* Ablageort garantiert. Die Abb. 7.1-1 verdeutlicht die Struktur einer Ablage.

Die *SCCS* Kommandos speichern Objekte im *OVS* in der Weise, daß zu einer *ersten Version* des Objektes stets nur noch die *Änderungen einer Version* zu der vorherigen gespeichert werden. Auf diese Weise kann der Speicherbedarf für die *Ablagen des OVS* gesenkt werden. Der Zugriff auf eine *Version eines Objektes* erfolgt so, daß nie mehr als ein Exemplar zur Editierung freigegeben ist, sodaß man die *Konsistenz von Objekten* auf einfache Weise im *OVS* sichern kann.

Als Objekte, die gespeichert werden können, akzeptieren die *SCCS* Kommandos Dateien mit einem *beliebigen Format*. Die Speicherung eines Objektes erfolgt unabhängig vom Datenformat des Objektes als *einfache Folge von Bytes*, sodaß neben ASCII Dateien oder speziellen Textdateien von Textverarbeitungssystemen auch binäre Programme oder Rasterbilder gespeichert werden können. Die

Interpretation eines Objektes erfolgt nur über eine Typinformation, die jedem Objekt im *Objektverzeichnis der Ablage* zugeordnet ist. Den Aufbau eines *Objektverzeichnisses* wollen wir nun erörtern.

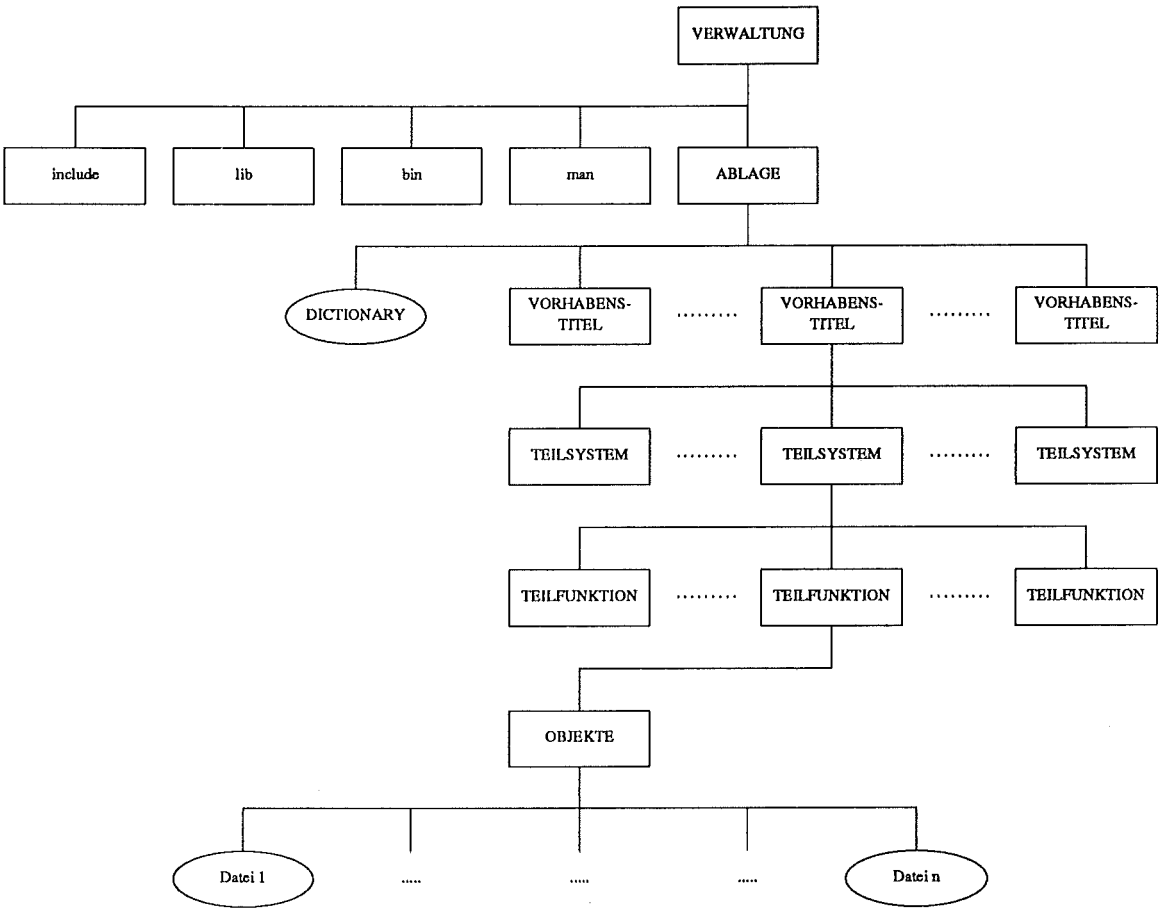

Abb. 7.1–1 : Dateistruktur des OVS

7.2 Objektverzeichnis

Das *Objektverzeichnis* (*Dictionary*) einer Ablage faßt die *Verwaltungsdaten* aller Objekte einer Ablage zusammen. Das *Objektverzeichnis* hat logisch die Form einer Tabelle (Relation) der folgenden Gestalt:

Schlüssel	Titel	Name	Version	Datum
SPI_***_AVE_SP_01	Spezifikation des OVS	CD	1.1	12.11.88
SPI_***_***_QS_01	Dokumentstruktur	Eg	1.1	24.02.88
SPI_ULX_AVE_PR_01	Prototyp eines OVS Servers	CD	1.1	21.02.89
DES_INF_FRD_MO_01	Modell einer Inferenzstrategie	KPS	1.1	22.06.88
.....	....	....	....	....
.....	....	....	....	....
.....	....	....	....	....
.....	....	....	....	....
.....	....	....	....	....
.....	....	....	....	....
.....	....	....	....	....
.....	....	....	....	....

Tab. 7.2–1 : Struktur des Objektverzeichnisses einer Ablage

- Jeder Eintrag im Objektverzeichnis enthält den *Objektschlüssel.* Die Mengen der vorhandenen Schlüsselteile werden zu Beginn eines Softwareprojektes festgelegt, können aber im weiteren Verlauf noch erweitert werden. Der Systementwickler wird so gezwungen, schon am Anfang eines Projektes über eine *modulare Struktur* seines Systemes nachzudenken, die sich unmittelbar im *Aufbau der Schlüssel* für alle beteiligten Objekte widerspiegelt. Das System überprüft die *Zulässigkeit eines Schlüssels* und dessen *Eindeutigkeit.*

  Bei der Abfrage erlaubt der *Schlüssel* die Suche nach Objekten zu gegebenen Vorhaben, Teilsystemen, Teilfunktionen oder zu konkreten Objekttypen.
- Der *Titel* eines Objektes kann frei gewählt werden und erlaubt bei einer Ablage eine inhaltsorientierte Suche nach Objekten.
- Eine *Namensinformation* identifiziert den *Autor* eines Objektes und ermöglicht daher bei einer Abfrage eine Auswahl nach Verfassern von Objekten. Sie entspricht einem Element der Menge (8) aus Kapitel 6.
- Die *Versionsnummer* identifiziert alle *Versionen* eines Objektes, die im *OVS* eingetragen sind. Sie entspricht einem Element der Menge (4) aus Kapitel 6. Der Benutzer kann jede vorhandene *Version* durch Angabe ihrer *Versionsnummer* extrahieren.
- Das *Freigabe-Datum* ist in der Regel mit dem Datum der Eintragung in das OVS identisch (Default). Die *Freigabe eines Objektes* ist eine organisatorische Maßnahme der *Softwarequalitätssicherung*, die in jedem Vorhaben im Institut individuell gehandhabt wird. Sie soll sicherstellen, daß ein in das *OVS* eingetragenes Dokument vorher definierten *Qualitätsanforderungen* genügt. Ein Objekt wird erst nach seiner *Freigabe* in das *OVS* eingetragen. Das *Freigabe-Datum* entspricht einem Element der Menge (7) aus Kapitel 6.

Neben diesen aus der *Methodik* des Softwareengineering begründeten *Verwaltungsdaten eines Objektes* enthält das *Objektverzeichnis* zwei weitere Informationen, die auf Grund der *heterogenen Rechnerumgebung* (einschließlich der Betriebssysteme und Applikationsprogramme) benötigt werden.

- Der *Datentyp* eines Objektes gibt darüber Auskunft, welche interne Struktur ein Objekt besitzt. Die Datentypspezifikation besteht dabei aus zwei Teilen. Ein Teil identifiziert in eindeutiger Weise den Zeichencode, in dem das Objekt auf einem bestimmten Rechner im Netzwerk erstellt wurde. Auf diese Weise kann man z.B. MS-DOS ASCII Dateien von UNIX ASCII Dateien unterscheiden.

  Das zweite Feld beschreibt, ob das Objekt ein Format besitzt, daß ein spezielles Applikationsprogramm verwendet. Mit *worstar* sind hier z.B. Objekte gekennzeichnet, die über das

Textverarbeitungssystem mit gleichem Namen auf einem Personal Computer (PC) erstellt wurden. Mit *postscript* wird man z.B. ein Objekt mit Postscript Programmen kennzeichnen.

Die *Datentypinformation* eines Objektes wird von den *OVS* Programmen benutzt, um zu entscheiden, ob ein Objekt auf einem Bildschirm oder Drucker ausgegeben werden kann und welche Konvertierungen hierzu nötig sind. Aus einer MS-DOS ASCII Textdatei sind z.B. die "Carriage Returns" am Ende einer Zeile zu entfernen, wenn die Datei auf einem UNIX System auszugeben ist, und binäre Objekte können in der Regel gar nicht auf dem Bildschirm oder Drucker ausgegeben werden.

Dem Anwender, der sich eine Datei kopiert, gibt die *Datentypinformation* darüber Auskunft, auf welchem Rechnertyp er das Objekt weiterbearbeiten kann.

- Der *Formatierer* eines Objektes ist eine Kommandosequenz, die auf einem Objekt ausgeführt werden muß, um eine für das Objekt definierte Ausgabe zu erhalten.

  Bei einem Textdokument ist dies eine Kommandosequenz, die das Dokument formatiert auf einem Drucker ausgibt. Bei Rasterbildern kann dies entsprechend der Aufruf eines Filterprogrammes sein, welches das Rasterbild in einen Druckercode für die Ausgabe konvertiert. Bei anderen Objekten können hier beliebige Kommandosequenzen angegeben werden, die zum Beispiel eine Applikation mit dem Objekt als Eingabe aufrufen.

Das *Objektverzeichnis* wird intern von den Serverprozessen in Form einer *Direktzugriffsdatei* (Zugriff über Schlüssel), die einen *Index* zum Suchen nach Wörtern innerhalb der Tabelle besitzt, verwaltet. Die Dateien mit Tabelle und Index sind im *Wurzelkatalog* der jeweiligen Ablage untergebracht. Die Serverprozesse benutzen den *Zugang über den Schlüssel*, um bei neuen Eintragungen und der Ausgabe von Dokumenten einen schnellen Zugriff auf die Verwaltungsdaten eines Dokumentes zu haben.

Die Ausgabeeinheiten erlauben es dem Benutzer bei einer Abfrage *reguläre Ausdrücke* für Textmuster anzugeben, die in der Tabelle gesucht werden sollen. Diese Suche wird durch den *Index* eines Objektverzeichnisses unterstützt.

8 Verteilte Ein-/Ausgabe von Objekten im Rahmen einer Server/Client-Lösung

Auf Grund der inhomogenen Rechnerwelt im IDT ist das OVS als *Server/Client Lösung* implementiert, in der ein *zentraler Server* auf einem UNIX Rechner alle eingetragenen Objekte in Ablagen zentral verwaltet und Ein- bzw. Ausgabeeinheiten auf Rechnern verschiedener Architekturen mit den Betriebssystemen UNIX, MS-DOS und VMS die Benutzerwünsche entgegennehmen und als *Clients des Servers* in definierten Protokollen mit dem Server ausführen (siehe [Coffield1987a, Gauweiler1987a, IPC-Primer1986a, Leffler1983a]).

Auf diese Weise bewegt sich der Benutzer in seiner gewohnten Rechnerumgebung, während die Verwaltung der Objekte für den Benutzer unsichtbar auf einem zentralen Rechnerknoten erfolgt. Insbesondere werden die normaler Weise für jeden Rechner anfallenden Arbeiten der *Datensicherung* der im Softwareentwicklungsprozeß vorhandenen Objekte auf dem zentralen Verwaltungsrechner nur einmal für alle im OVS eingetragenen Objekte durchgeführt.

Die Infrastruktur des IDT mit einem allen zugänglichen LAN auf Basis des *Ethernet Standards*, sowie die Verfügbarkeit der *TCP/IP Protokolle* (siehe [Comer1988a]) auf allen beteiligten Rechnern, führten dazu, die Kommunikation zwischen dem Server und Client auf diesen Protokollen aufzusetzen. Eine noch höhere Protokollebene, wie z.B. das *SUN RPC Protokoll* wäre durchaus sinnvoll gewesen, war jedoch nicht auf allen Rechnern verfügbar. Die von uns spezifizierten und implementierten Anwenderprotokolle werden im folgenden Unterkapitel erläutert.

Die *Benutzerschnittstelle der Clients* ist mit den einfachsten auf allen Rechnern eines bestimmten Betriebssystemtyps verfügbaren Hilfsmitteln realisiert. Wir haben hier bewußt auf eine "schönere" Oberfläche auf der Grundlage von Window Systemen verzichtet, da sie uns einerseits nicht eine höhere Funktionalität bereitgestellt hätte, und andererseits Standard Window Systeme noch nicht für einen

gegebenen Betriebssystemtyp für alle beteiligten Rechner (z.B. SUN und VAX UNIX Rechner) zur Verfügung stehen. So wurde die Oberfläche der Clients auf den UNIX und VMS Rechnern mit der *curses Bildschirmbibliothek* implementiert, während unter MS-DOS der Bildschirm direkt angesteuert wurde. Die Benutzerschnittstelle wird im Anschluß an die Beschreibung der Protokolle genauer erläutert.

Da der Zugriff auf eine Ablage von mehreren Clients gleichzeitig schreibend und lesend erfolgen kann, handelt es sich bei einer Ablage um eine *gemeinsame Datenstruktur*, die unter *gegenseitigem Ausschluß* (*mutual exclusion*) stehen muß. Es wird daher eine *Synchronisation der Zugriffe* notwendig. Diese ist im letzten Unterkapitel dieses Kapitels beschrieben.

8.1 Die Protokolle zwischen Clients und Server des OVS

Die Protokolle basieren auf dem Austausch von natürlich sprachlichen Nachrichten in Form von ASCII Zeilen. Jede Nachricht beginnt mit einem *Schlüsselwort*, das die Funktion dieser Nachricht eindeutig bestimmt. Der Dialog wird dabei *aktiv vom Client* geführt, während der Server die vom Client spezifizierten Aktionen zunächst auf ihre *Konsistenz* überprüft und gegebenenfalls ausführt. Der Server sendet daher nur zwei Typen von Nachrichten. Eine Statuszeile zeigt den Erfolg einer vom Client angeregten Aktion an, während eine Fehlernachricht einen Fehler meldet. Jede Status- bzw. Fehlermeldung vom Server enthält neben einer näheren Erklärung eine eindeutige Zahl, die programmintern zur Fehlerbehandlung verwendet wird.

Die Clients verwenden eine Reihe von Schlüsselwörtern, die als Argumente Informationen zum Server übertragen, die dieser auswertet, und die vom Server bestimmte Aktionen verlangen. So übersendet eine Ausgabeeinheit, wenn sie die Ausgabe eines Objektes wünscht, dem Server eine Mitteilungsfolge der Gestalt:

```
KEY:     Objektschlüssel
VERSION:Versionsnummer des gewünschten  Objektes
PART:    spezifiziert, welcher Teil des Objektes auszugeben ist
OPTION: Ausgabeoption
```

Der Server prüft dann in dem *Objektverzeichnis* der anzusprechenden Ablage, ob die Version des auszugebenden Objektes vorhanden ist, und ob das Objekt mit der *PART* Spezifikation und *OPTION* Spezifikation ausgegeben werden kann. *OPTION* kann hier die Werte *SEND* zur Übertragung des Objektes zum Client oder *PRINT* zur Ausgabe eines Objektes auf dem Drucker des Servers annehmen. Der Server sendet dann eine entsprechende Status- oder Fehlermeldung zum Client, der in dem Fall, daß kein Fehler aufgetreten ist, die Ausgabe des Objektes einleitet.

Die Funktionalität der Eingabe und Ausgabe wurde in zwei getrennten Protokollen realisiert, wobei der Server jeweils verschiedene Prozesse zur Behandlung einer Eingabe- bzw. Ausgabeeinheit erzeugt, während er selbst nur ankommende Verbindungwünsche von Clients bearbeitet.

Client	Server
openconnection();	acceptconnection();
send("NEWENTRY:");	receive(message);

receive(message);	if (message != OK)
	{
	send("ERROR: ERRMESG1");
	error();
	}
	else send("STATUS: STATMESG1");

if (NOT-STATUS-MESG)
error();
else send("LOGIN:.....");
	receive(message);

receive(message);	if (NO-CORRECT-LOGIN)
	{
	send("ERROR: ERRMESG2");
	error();
	}
	else send("STATUS:STATMESG2");

if (NOT-STATUS-MESG)
error();
else do {
switch (mode)
{
case ENTRY:
send("KEYS:...");
 	
receive(message);
 	
if (NOT-STATUS-MESG)
continue;
else doctransfer(dok);
 	
receive(message)
 	
break;
case END:send("END:");
endsession();
 	
}
} while (TRUE);
	do {
	
	receive(message);
	switch (message)
	{
	case KEY:
	if (NOT-ACCEPTED-KEYS)
	{
	send("ERROR:ERRMESG3");
	continue;
	}
	else send("STATUS:STATMESG3");
	
	receivedok(dok);
	if (transfer != OK)
	{
	send("ERROR:ERRMESG4");
	continue;
	}
	else send("STATUS:STATMESG4");
	break;
	case END:endsession();
	
	default:send("ERROR:ERRMESG5");
	}
	while (message != END);

Tab. 8.1–1 : Protokoll zwischen Eingabeeinheit und Server

Die beiden Kommunikationsprotokolle sind in den Tabellen 8.1-1 und 8.1-2 in Form eines Pseudo Codes in C Syntax dargestellt, wobei den Aktionen der Clients jeweils die Aktionen des Servers gegenübergestellt sind. Man beachte aber, daß dieser Code stark vereinfacht ist und keinerlei weitergehende Fehlerbehandlung beschreibt.

Eine Eingabeeinheit kann nach ihrer Anmeldung beim Server und der Login Sequenz, die die Ablage bestimmt, innerhalb einer Schleife im Protokoll Eintragungen von Objekten vornehmen. Hierzu sendet sie die *Verwaltungsdaten des Objektes* zum Server, und überträgt anschließend das *Objekt*, wenn die *Verwaltungsdaten des Objektes* vom Server akzeptiert werden. Eintragungen von Objekten werden so lange vorgenommen, bis der Client durch eine Ende-Mitteilung die Verbindung mit dem Server beendet.

Client	Server
openconnection();	acceptconnection();
send("WETTABLE:");	receive(message);

receive(message);	if (NO-CORRECT-ENTRY)
	{
	send("ERROR: ERRMESG1");
	error();
	}
	else send("STATUS: STATMESG1");

if (NOT-STATUS-MESG)
error();
else send("LOGIN:.....");
	receive(message);

receive(message);	if (NO-CORRECT-LOGIN)
	{
	send("ERROR: ERRMESG2");
	error();
	}
	else send("STATUS:STATMESG2");

if (NOT-STATUS-MESG)
error();
else do {
send("REGEXP:....");
 	
receive(message);
 	
} while (NOT-STATUS-MESG);
	do {
	receive(message);
	if (NOT-RIGHT-EXP)
	send("ERROR: ERRMESG3");
	else send("STATUS:STATMESG3");
	} while (NOT-RIGHT-EXP);

do {
 	
switch (MODE)
{
case TABLE:
send("RANGE:min-max");
receivetable();
 	
break;
case DOC:
send("KEY:.....");
 	
receive(message);
if (NOT-STATUS-MESG)
continue;
 	
switch (OPTION)
{
case SEND:
receivedoc();
break;
case PRINT:
 	
receive(message);
 	
break;
}
break;
case SEARCH:
send("LINE:....");
receive(message);
 	
break;
case END:send("END:");
endsession();
}
} while (TRUE);
	do {
	receive(message);
	switch (message)
	{
	case RANGE:
	if (NOT-RIGHT-RANGE)
	send("ERROR:ERRMESG4");
	else sendtable();
	break;
	case DOC:
	if (NOT-RIGHT-KEYS)
	{
	send("ERROR:ERRMESG5");
	continue;
	}
	else send("STATUS:STATMESG4");
	switch (OPTION)
	{
	case SEND:
	senddoc();
	break;
	case PRINT:
	if (printdoc()!=OK)
	send("ERROR:ERRMESG6");
	else send("STATUS:STATMESG5");
	break;
	}
	break;
	case SEARCH:
	if (NOT-RIGHT-SEARCHPARAM)
	send("ERROR:ERRMESG7");
	else send("STATUS:STATMESG6");;
	break;
	case END:
	endsession();
	}
	} while (TRUE);

Tab. 8.1–2 : Protokoll zwischen Ausgabeeinheit und Server

Eine Ausgabeeinheit bestimmt nach Anmeldung und Login Sequenz zunächst über einen *regulären Ausdruck*, welcher *Teil des Objektverzeichnisses* vom Server extrahiert werden soll. Der Server lädt den angegebenen Teil der Tabelle in einen internen Pufferbereich und gibt dem Client die Anzahl der Zeilen im Puffer zurück. Innerhalb einer Schleife kann der Client dann einen Teilbereich der Tabelle anfordern, eine Suche nach Zeilen mit vorgegebenem Muster vom Server durchführen lassen oder die Ausgabe eines Objektes vom Server veranlassen. Die Ausgabe eines Objektes erfolgt dann entweder auf dem Drucker des Servers, wenn das Objekt ein formatierbares Dokument ist, oder durch Übertragung des Objektes zum Client, der dann weiter entscheidet, ob er das Objekt in einer Datei ablegt oder auf dem Bildschirm ausgibt.

Auf einige Besonderheiten des Protokolls, die nicht aus dem Pseudo Code zu entnehmen sind, soll noch gesondert eingegangen werden:

- Um einen *Deadlock von Serverprozessen* zu vermeiden, deren Clients sich einfach illegal aus dem Protokoll verabschieden, wie dies z.B. bei einem MS-DOS Rechner der Fall ist, den sein Benutzer beim laufenden Protokoll einfach abschaltet, ist die Serversoftware so spezifiziert, daß ein Klient die Kommunikation mit dem Server an jeder *beliebigen Stelle* im Protokoll abbrechen kann.

  Dies heißt, daß der Server erst dann *Aktionen auf den Objektverwaltungsdatenstrukturen* durchführen darf, wenn er *alle* vom Client benötigten Informationen besitzt. Insbesondere wird ein Objekt erst dann eingetragen, wenn sowohl das Objekt als auch seine Verwaltungsdaten vollständig und korrekt übertragen wurden. In dem Fall, daß die Verbindung zum Client vor Beendigung des Protokolls unterbrochen wird, ignoriert der Server einfach die bis dahin übermittelten Informationen, sodaß ein Client sie gegebenenfalls nach Aufbau einer neuen Verbindung erneut senden muß. Man nennt solche Server auch *zustandslos* (*stateless*).
- Um einer *Verklemmung von Clients* vorzubeugen, verwendet der Server weiter einen *Timeout Mechanismus*, der die Zeit beschränkt, in der ein Server in einem Systemaufruf auf Nachrichten vom Client wartet. Wird ein solcher Timeout Wert überschritten, geht der Server davon aus, daß ein Client nicht mehr aktionsfähig ist, und schließt die Verbindung zum Client.

8.2 Die Funktionalität der Ein- und Ausgabeeinheiten

Eine Eingabeeinheit erlaubt die *Eintragung von Objekten* in eine Ablage des OVS entweder interaktiv oder im Kommandomode. Neben der Spezifikation des einzutragenden Objektes muß der Anwender dabei die *Verwaltungsdaten des Objektes*, sowie die *Ablage*, in der das Objekt einzutragen ist, und das zugehörige *Paßwort* angeben. Im Kommandomode geschieht die Angabe der Verwaltungsdaten in Form einer Datei, die die Verwaltungsdaten des Objektes in einem festen Format enthält. Diese Datei kann über ein spezielles Kommando oder im interaktiven Mode der Eingabeeinheit erstellt werden.

Im interaktiven Mode baut die Eingabeeinheit eine *Bildschirmmaske* auf, in der der Anwender die *Verwaltungsdaten des Objektes* interaktiv eingeben, in einer Datei sichern oder von einer Datei einlesen kann. Defaults ersparen dabei dem Anwender an vielen Stellen Schreibarbeit. Wenn die Eingabe eines Objektes gestartet wird, werden zunächst die *Verwaltungsdaten des Objektes* verifiziert. Nur ein Objekt mit *syntaktisch und semantisch korrekten Verwaltungsdaten* wird eingetragen. Alle Fehler werden dem Benutzer in verständlicher Form angezeigt, ein eingebautes Help Kommando erleichert ihm die Benutzung.

Die Ausgabeeinheit besitzt nur einen interaktiven Mode. Nach Angabe der gewünschten *Ablage* und dem zugehörigen *Paßwort* kann der Benutzer angeben, welchen *Teil des Objektverzeichnisses der Ablage* er sich zu Abfragezwecken näher ansehen will. Dieser *Teil des Objektverzeichnisses* wird dann in Form einer Tabelle in einen internen Pufferbereich geladen und auf einem Window in einem Teilausschnitt ausgegeben.

Der Benutzer kann sich dann über Editorkommandos, wie *Blättern, Scrollen* und einer *Find Operation*, die Tabelle ansehen und Einträge über einen *Scrollbar* selektieren. Das Objekt, dessen Tabelleneintrag mit dem *Scrollbar* selektiert ist, kann ausgegeben werden. Als Ausgabeoperationen kann man dabei eine Ausgabe auf dem Drucker oder Bildschirm oder eine Kopieroperation vom Server auf das Dateisystem des Clients auswählen. Server und Clients überprüfen dabei, ob die gewählte Ausgabeform überhaupt möglich ist. Nötige Datentransformationen, z.B. zwischen den verschiedenen ASCII Formaten der einzelnen Betriebssysteme, werden automatisch vorgenommen, sofern dies möglich und sinnvoll ist.

Die Suche nach Objekten wird im wesentlichen durch einen Mechanismus realisiert, der damit auch die Mächtigkeit der Abfrage bestimmt. Dieser Mechanismus sind die *regulären Ausdrücke* (*regular expression*), die in sehr vielen UNIX Werkzeugen Verwendung finden (siehe z.B. [Aho1974a, Aho1988a]). *Reguläre Ausdrücke*, die im Compilerbau häufig für die Spezifikation und Implementierung der *lexikographischen Analyse* (siehe [Aho1986a, Lesk1975a]) verwendet werden, spezifizieren nach einer definierten Syntax Zeichenmuster, denen Dateien, Textzeilen oder auch nur Wörter entsprechen können.

Wenn der Benutzer einer Ausgabeeinheit einen *regulären Ausdruck* zur Spezifikation eines Teilbereiches des Objektverzeichnisses einer Ablage angibt, den er sich ansehen will, werden vom Server alle Zeilen des Objektverzeichnisses extrahiert, deren Zeichenmuster diesem *regulären Ausdruck* genügen. Bei der *Find Operation* wird die nächste Zeile selekiert, die einem vorgegebenem *regulärem Ausdruck* entspricht. Die folgende Tabelle verdeutlicht die Mächtigkeit der *regulären Ausdrücke.*

A,B,..,Z,a,b,..,z,0,1,..,9	stehen für sich als Zeichen
Aaabcde19ad	steht damit für sich als String
[a,b,c]	steht für a oder b oder c
1[a,b,c]	steht damit für 1a, 1b oder 1c
[A-Z]	steht für A,B,.... oder Z
[^A-Z]	steht für alle Zeichen außer A,B,.. oder Z
a*	steht für a beliebig oft
a+	steht für a beliebig oft, aber mindestens 1 mal
ba*	steht damit für b, ba, baa, ...
ba+	steht damit für ba, baa, baaa, ...
.	steht für ein beliebiges Zeichen
.*	steht damit für beliebige Strings
^a	steht für alle Zeilen mit a am Anfang
a$	steht für alle Zeilen mit a am Ende
(abc)*	steht für abc, abcabc, ...
.*	alle DICTIONARY Einträge
.*SPI_.*	alle Dokumente zum Vorhaben SPI
.*_PT_.*	alle Prototyping Dokumente
Dokument	alle Dokumente mit Dokument im Titel

Tab. 8.2–1 : Beispiel für reguläre Ausdrücke

8.3 Synchronisation an den Ablagen

Eine Ablage und ihr Objektverzeichnis muß man als *gemeinsame Datenstruktur* auffassen, die zwar von mehreren Serverprozessen gleichzeitig gelesen aber nur von einem Prozeß zu einem Zeitpunkt verändert werden darf (siehe [Wettstein1984a]).

Der Server setzt daher zur *Synchronisation* der Aktivitäten an einer Ablage ein *Semaphortripel* ein, dessen Werte nur in einer *unteilbaren Operation* verändert werden können. Jedes *Semaphortripel* ist im System durch eine *Semaphor-ID* gekennzeichnet, die jeder Ablage eindeutig zugeordnet ist.

Bei der Behandlung von Ausgabeeinheiten setzt der Server beim Zugriff auf die Ablage und das Objektverzeichnis das *Semaphortripel* so, daß zwar weitere Serverprozesse, die Ausgabeeinheiten bedienen, auf die Ablage zugreifen können, nicht aber Eingabeeinheiten diese verändern können. Dies ist allerdings nur dann möglich, wenn nicht bereits ein Serverprozeß, der eine Eingabeeinheit bedient, die Ablage zum Schreiben reserviert hat. In diesem Falle warten die Prozesse solange, bis der Schreiber die Ablage wieder freigibt.

Ein Serverprozeß, der eine Eingabeeinheit bedient, reserviert für sich zunächst eine Ablage zum Lesen, wenn er die Verwaltungsinformationen von einzutragenden Objekten vergleichen muß. Diese Reservierung zum Lesen ist so realisiert, daß ein weiterer Schreiberprozeß an der Ablage blockiert wird, da sonst ein Schreiber eventuell Daten liest, die durch Neueintragung des bereits in der Ablage befindlichen Schreibers nicht mehr auf dem neuesten Stand sind. Wenn der Schreiber dann die Ablage verändert, setzt er weiter eine Sperre für alle Leser, da diese sonst während des Schreibens inkonsistente Daten lesen könnten.

Alle Semaphoroperationen werden zurückgenommen, wenn die erforderlichen Aktionen unter Zugriff auf die Ablage durchgeführt wurden. Das UNIX System sorgt dabei *automatisch* für ein Zurücksetzen der Semaphorwerte, wenn ein Prozeß auf Grund eines Fehlers plötzlich stirbt, ohne daß er die Semaphorwerte noch zurücksetzen kann (siehe [Thomas1987a, Bach1986a]).