Diplomarbeit

Entwurf und Integration verschiedener Werkzeuge zur Analyse und Verwaltung persistenter ProSet-Daten

April 1995

Diplomand:

Jörg Heinrich

Grillostr. 9

45881 Gelsenkirchen

Gutachter:

Prof. Dr. E.-E. Doberkat

Prof. Dr. U. Kelter

Lehrstuhl X Softwaretechnologie

TEIL I

schriftliche Ausarbeitung

An dieser Stelle danke ich allen, die mich während der Erstellung dieser Diplomarbeit unterstützt haben. Prof. Dr. E.-E.Doberkat sowie Prof. Dr. U. Kelter danke ich für die Betreuung. Bedanken möchte ich mich auch insbesondere bei meinem direkten Ansprechpartner und Betreuer H. G. Sobottka, der mir zu jedem Zeitpunkt hilfreich zur Seite stand und mich mit konstruktiver Kritik vorangetrieben hat. Schließlich und endlich möchte ich mich auch bei meiner Frau Judith bedanken, die es verstand mich auch in schweren Tagen moralisch aufzubauen und mir einen Halt zu geben.

Gelsenkirchen, im April 1995 Jörg Heinrich

Inhaltsverzeichnis

1 Einleitung

In diesem Kapitel werden einige einleitende Informationen zu der vorliegenden Diplomarbeit gegeben. Dabei wird das Umfeld der Diplomarbeit erläutert und die grundlegenden Begriffe, die im Rahmen der Diplomarbeit wichtig sind, erklärt.

1.1 Motivation

Software-Entwicklungs-Umgebungen stellen ein mächtiges Werkzeug zur Softwareentwicklung dar. Sie sollen die Softwareentwicklung durch integrierte Werkzeuge unterstützen und beschleunigen. Dabei werden verschiedene Konzepte verfolgt, die sich an verschiedenen Entwicklungsmodellen orientieren [Kel93].

In dieser Diplomarbeit werden verschiedene Werkzeuge entworfen und konstruiert. Diese werden später im Rahmen einer Software-Entwicklungs-Umgebung eingesetzt, die an der Universität Dortmund entwickelt wird. Diese Software-Entwicklungs-Umgebung basiert im wesentlichen auf zwei Komponenten. Zum einen auf der Programmiersprache PROSET und zum anderen auf der objektorientierten Datenbank H-PCTE. Der Bereich Prototyping und damit in Zusammenhang stehende Begriffe, sowie die beiden Komponenten PROSET und H-PCTE werden im folgenden näher betrachtet.

1.2 Prototyping

Prototyping ist eine Methodik zur Softwareentwicklung. Prototyping bezeichnet dabei den Prozeß der Konstruktion eines ablauffähigen Modells, das alle wesentlichen Eigenschaften des Endproduktes besitzt. Dieses Modell erlaubt es dann, auf dem Wege der Rückkopplung mit dem Kunden, die Eigenschaften des Produkts zu prüfen und Aufschlüsse über die weitere Entwicklung zu bekommen.

Die Intention des Prototyping kann als *experimentell*, *evolutionär* oder *throw-away* klassifiziert werden. Diese Klassifizierung gibt Aufschluß darüber, auf welche Weise der Prototyp entstanden ist. Beim *throw-away* Zugang liegt der Schwerpunkt darauf, *explorativ* vorzugehen, um mögliche Lösungsvorschläge zu diskutieren. Beim *experimentellen* Zugang soll eine bereits gefundene Lösung im Hinblick auf ihre Angemessenheit untersucht werden. Der evolutionäre Ansatz verfolgt die Entwicklung mehrerer Prototypen in verschiedenen Versionen, bis sie stabil geworden sind.

Alle oben genannten Prototyping-Zugänge arbeiten nur unter wesentlicher Beteiligung des Benutzers. Der Benutzer trägt zur Gestaltung des Endproduktes maßgeblich bei, indem er dem Entwickler bei der Analyse der Prototypen eine Rückkopplung gibt [DF89].

1.2.1 Begriffe

In diesem Abschnitt sollen einige wesentliche Begriffe aus den Bereichen Software-Entwicklungs-Umgebungen und Prototyping, die auch in den folgenden Abschnitten von Bedeutung sind, erläutert werden.

Dokumente

Dokumente sind alle Arten von Daten, unabhängig von ihrer Struktur und ihrer Bedeutung, die im Rahmen einer Software-Entwicklungs-Umgebung anfallen. Die Dokumente können in verschiedene Dokumentgruppen eingeteilt werden, die den verschiedenen Aufgabenbereichen innerhalb einer Software-Entwicklungs-Umgebung entsprechen. Diese Dokumentgruppen müssen nicht zwangsläufig disjunkt sein, sondern bieten vielmehr die Möglichkeit verschiedene Sichten auf die Gesamtdatenmenge zu realisieren. Beispiele für Dokumentgruppen sind:

1. *Entwicklungsdokumente*
   sind alle Dokumente, die im Rahmen des Softwareentwicklungsprozesses anfallen. (Sourcecodes, Dokumentationen, PROSET-Daten, Programm-Daten usw.)
2. *Testdokumente*
   sind Dokumente, die Testeingaben oder Testergebnisse beinhalten.
3. *Analysedokumente*
   sind Dokumente, die Auswertungen des Datenbestandes oder der Daten der Anwendungsprogramme beinhalten.
4. *Anwenderdokumente*
   sind private Dokumente, auf die nur ein spezieller Anwender, ihr Besitzer, Zugriff hat.

Im Rahmen dieser Diplomarbeit findet der allgemeine Begriff der Dokumente immer dann Anwendung, wenn von Daten der Software-Entwicklungs-Umgebung gesprochen wird, ohne das die genaue Struktur der Daten näher betrachtet werden soll.

Prototyping-Umgebung

Eine Prototyping-Umgebung ist eine Software-Entwicklungs-Umgebung, die die notwendige Funktionalität zur Konstruktion von Softwareprototypen anbietet. Dabei müssen die verschiedenen Phasen eines Entwicklungsmodells durch Werkzeuge unterstützt werden. Die angebotenen Werkzeuge sollten kooperieren können, um eine einfache Entwicklung von Prototypen zu gewährleisten.

Offene Umgebung

Eine Prototyping-Umgebung sollte eine offene Umgebung sein. Das bedeutet, daß es mit vertretbarem Aufwand möglich sein muß, die Umgebung an neue Anforderungen anzupassen. Diese Anpassung kann zum Beispiel in der Integration neuer Werkzeuge oder der Verknüpfung neuer Dokumenttypen mit bereits vorhandenen bestehen. Verknüpfungen zwischen Dokumenttypen drücken dabei Beziehungen zwischen den Dokumenttypen aus (z.B. Sourcecode und zugehörige Dokumentation).

Integration

Integration bezeichnet im wesentlichen die Eingliederung verschiedenster Werkzeuge in eine einheitliche Umgebung, in der sie zusammenarbeiten und Daten austauschen sollen [Iso87]. Die Prototyping-Umgebung bildet dabei den Integrationsrahmen [Kel93].

<u>Datenintegration</u>

Der Begriff der Datenintegration umfaßt die Fähigkeit der beteiligten Umgebungskomponenten, Daten miteinander auszutauschen und gemeinsam auf die gleichen Daten zuzugreifen. Weiterführende Konzepte erlauben die Definition von Datenmodellen, die durch formale Schemata repräsentiert werden. Durch diese Konzepte können Daten entsprechend dem Entwicklungsprozeß strukturiert werden [Iso87].

<u>Kontrollintegration</u>

Die Kontrollintegration befaßt sich mit dem Informationsaustausch zwischen den Werkzeugen. Im Software-Entwicklungsprozeß werden viele verschiedene Werkzeuge benutzt, die mit den gleichen Daten arbeiten sollen. Deshalb ist es notwendig, wenn ein Werkzeug die Daten verändert, die anderen Werkzeuge über die Veränderungen der Daten zu informieren, um einen konsistenten Entwicklungszustand zu gewährleisten. Unter Entwicklungszustand verstehen wir an dieser Stelle eine Momentaufnahme des gesamten Datenbestandes und den Zuständen der einzelnen Dokumente. In diesem Zusammenhang ist es notwendig, daß die Werkzeuge miteinander kommunizieren können, um andere Werkzeuge über ihre Aktionen zu informieren [Iso87].

Präsentationsintegration

Der Begriff der Präsentationsintegration beinhaltet die Anforderung eines einheitlichen Erscheinungsbildes aller in einer Umgebung integrierten Werkzeuge. Anhand eines Style-Guides können gemeinsame Attribute definiert werden, worunter unter anderem auch Konventionen für eine einheitliche Handhabung der Werkzeuge fallen. In diesem Zusammenhang können Konventionen für den Aufruf von Werkzeugen, die Menügestaltung usw. vereinbart werden. Dadurch wird eine einheitliche Navigation sowohl innerhalb der Werkzeuge als auch zwischen ihnen unterstützt. Style-Guides, die Konventionen für ein einheitliches Aussehen und Verhalten von Komponenten der Umgebung (z.B. Werkzeuge) angeben (einheitliches „Look and Feel“), erleichtern den späteren Umgang mit der Software-Entwicklungs-Umgebung. Für den Benutzer ist es bei einer einheitlichen Bedienung der integrierten Werkzeuge einfacher, die Benutzung der Umgebungs-Komponenten zu erlernen. Für den Werkzeugentwickler erleichtern feste Konventionen die Designentscheidungen in Bezug auf die neu zu konstruierenden Werkzeuge, die in die Umgebung integriert werden sollen [Iso87].

1.2.2 Aufgaben einer Prototyping-Umgebung

Aufgrund immer komplexer werdender Anforderungen an die Softwareentwicklung, besteht ein Bedarf an Werkzeugen, die die Lösung von komplexen Aufgabenstellungen vereinfachen und entsprechende Vorgehensmodelle unterstützen.

Das ursprüngliche sequentielle Vorgehensmodell der Softwareentwicklung, das Wasserfall-Modell [Boe76] (siehe Abbildung 1), ist für die Bearbeitung von komplexen Problemlösungen nicht geeignet. Dies liegt in der Tatsache begründet, daß durch das Wasserfall-Modell die durchzuführenden Schritte, ihre Resultate und die genaue Abfolge der Schritte vorgeschrieben werden. Zuerst werden die Anforderungen formuliert, danach die Software entworfen und abschließend wird die Software implementiert und getestet. Das bedeutet, daß zu Beginn alle Anforderungen spezifiziert sein müssen und zu einem späteren Zeitpunkt nur noch schwer geändert werden können. Eine Änderung würde dazu führen, daß nochmals bei Schritt 1 begonnen werden müßte. Durch die zunehmende Größe und Komplexität der zu konstruierenden Software, wird es für den späteren Benutzer jedoch immer schwieriger, die gesamten Anforderungen direkt zu Beginn des Entwicklungsprozesses zu formulieren.

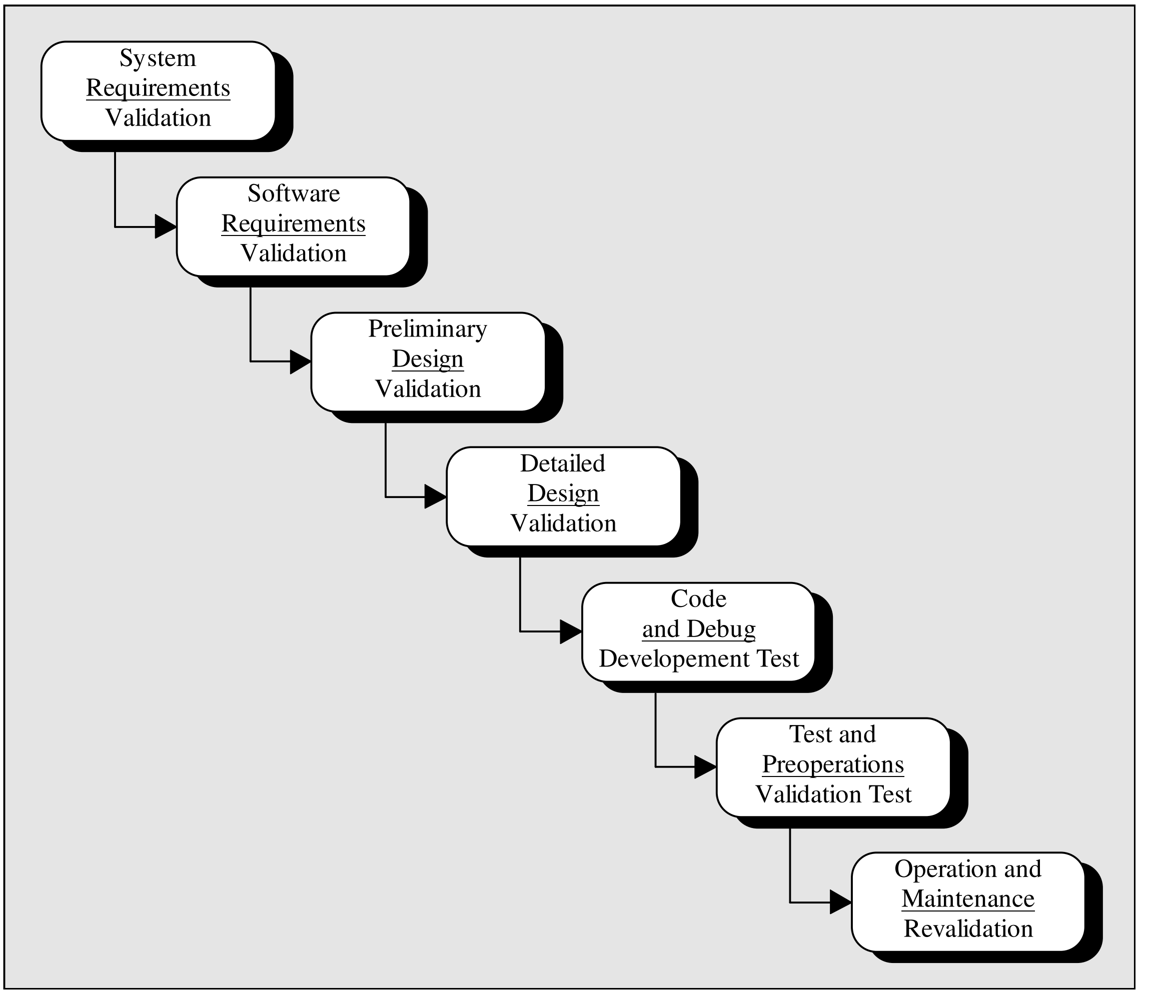

Abbildung 1: Wasserfallmodell

Der Benutzer kann die Anforderungen nicht immer genau spezifizieren, sondern sie meistens nur informell beschreiben. Das bedeutet unter anderem, daß die vom Benutzer formulierten Anforderungen in aller Regel unvollständig sind. Durch den fehlenden formalen Hintergrund des Benutzers, kann dieser den Gesamtumfang der Problemstellung nicht abschätzen und die im Laufe der Entwicklungsarbeit auftretenden Probleme nicht berücksichtigen. Hier wird deutlich, daß ein sequentielles Modell wie das Wasserfall-Modell zu unflexibel ist, um den Bedürfnissen des Benutzers und des Entwicklers gerecht zu werden. Dies liegt im wesentlichen daran, daß keine Rückschritte vorgesehen sind. Ein weiterer Nachteil des Wasserfall-Modells besteht darin, daß erst sehr spät im Verlauf der Entwicklung, ablauffähiger Code zur Verfügung steht, der eine Rückkopplung mit dem Benutzer erlaubt.

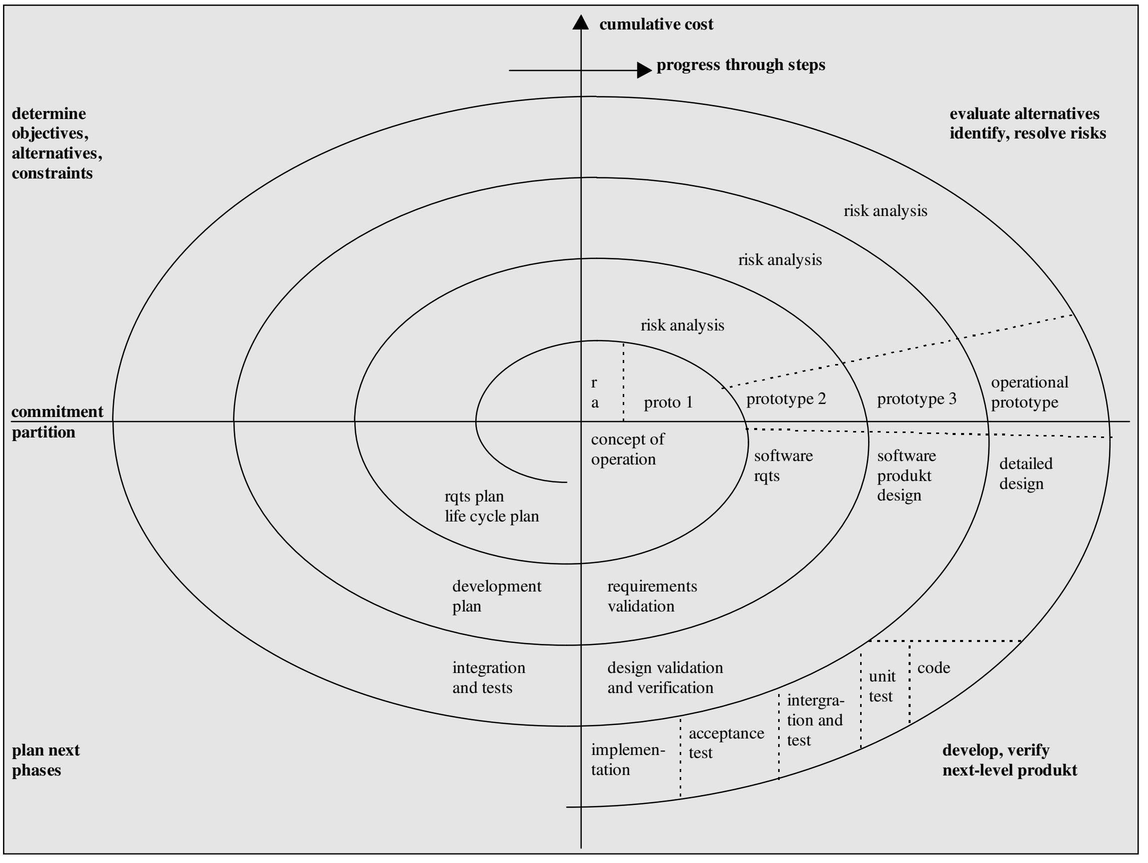

Abbildung 2: Spiralmodell

Aus den oben genannten Gründen ist es notwendig, andere leistungsfähigere Modelle zu entwickeln. Ein iteratives Modell, das es erlaubt flexibel auf Anforderungsänderungen zu reagieren, ist beispielsweise das Spiral-Modell [Boe88] des Software-Lebenszyklus (siehe Abbildung 2). In diesem Modell können mehrere Prototypen konstruiert werden. Mit jedem Prototyp können die Anforderungen, auch mit dem Kunden, neu überprüft und gegebenenfalls ergänzt werden. Man spricht in diesem Zusammenhang auch von Risikoanalyse. Risikoanalyse bedeutet in diesem Fall, die Analsyse der Ungenauigkeit und Risiken an dieser Stelle der Entwicklung. Damit kann das Risiko für das weitere Vorgehen abgeschätzt werden. Eventuelle Mißverständnisse zwischen Benutzer und Entwickler können unter Umständen rechtzeitig erkannt werden.

Ein aus unserer Sicht wesentlicher Punkt bei der Entwicklung von Prototypen ist, daß die Prototypen ausführbar sein müssen. Dies ist notwendig, um eine Validierung der Prototypen zu ermöglichen. Auf diesem Weg kann der Benutzer überprüfen, ob die Anforderungen geändert oder ergänzt werden müssen. Ein weiteres, speziell auf Prototyping-Bedürfnisse zugeschnittenes Modell, ist das in Abbildung 3 gezeigte Modell [DF89].

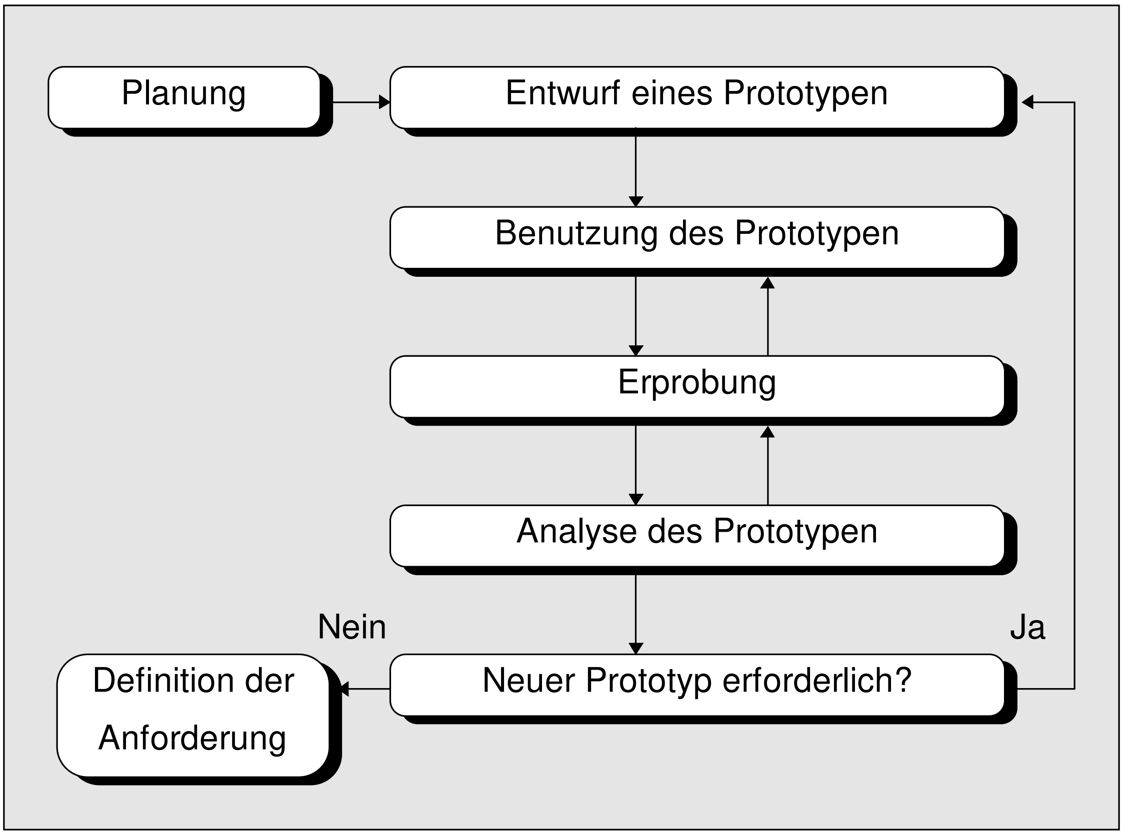

Abbildung 3: Spezielles Prototyping-Modell

Das Modell bringt Prototyping in die frühen Phasen des Entwurfs ein. Die Prototyping-Phase tritt dabei zwischen die Planung und die Anforderungsdefinition und wird als Zyklus beschrieben. Passend zum verwendeten Modell benötigt der Entwickler eine Umgebung, die ein solches flexibles Vorgehen unterstützt. Prototyping-Umgebungen sollen genau dies leisten. Während im klassischen, sequentiellen Vorgehensmodell die Implementation sehr spät im Software-Lebenszyklus angesiedelt ist, wird im Rahmen von Prototyping-Umgebungen der Ansatz verfolgt, möglichst früh einen *Prototypen* zu implementieren. Dieser Prototyp ist ein Modell des zu entwerfenden Produkts, das im Laufe des Entwicklungsprozesses verfeinert werden kann. Dieses Vorgehen erlaubt es dem Entwickler die Anforderungen des Benutzers, zu einem sehr frühen Zeitpunkt, in Form eines Prototypen, sichtbar zu machen. Weiterhin gibt das Vorgehen dem Benutzer die Möglichkeit, Anforderungsänderungen frühzeitig vorzunehmen. Eine Prototyping-Umgebung die ein solches Vorgehen unterstützt, muß zwangsläufig einen komplexen Aufbau haben und hohen Anforderungen genügen. Der komplexe Aufbau ergibt sich aus der Notwendigkeit vieler verschiedener Werkzeuge, die die Phasen des Prototyping unterstützen müssen. Diese Werkzeuge müssen im Rahmen der Integration in verschiedenen Bereichen zusammenarbeiten. Die dadurch notwendigen Beziehungen zwischen den Werkzeugen und zwischen den Dokumenten verlangen einen erheblichen Verwaltungsaufwand.

Eine wesentliche Anforderung an eine solche Prototyping-Umgebung ist, daß die zur Verfügung gestellten Funktionen bezüglich ihrer Daten, ihrer Benutzungsschnittstellen und ihrer Steuerung *integriert* sind (siehe Abschnitt 1.2.1). Um die Flexibilität einer solchen Umgebung zu gewährleisten, sollte sie *offen* sein in dem Sinne, daß neue Werkzeuge mit wenig Aufwand in die vorhandene Umgebung *integriert* werden können. Weitere Informationen zu diesem Themengebiet finden sich in [DF89].

1.3 Persistenz

Der Begriff Persistenz wird im Zusammenhang mit der Lebensdauer von Daten gebraucht. Persistenz bezeichnet die Eigenschaft von Daten, ihre Gültigkeit auch nach Beendigung eines Programms zu behalten, also nicht transient zu sein. Traditionelle Beispiele für persistente Daten sind die Daten in jeglicher Art von Datenbanken oder externe Dateien auf einem Massenspeicher, wie einer Festplatte oder einem Bandlaufwerk. Hier können Daten, unabhängig von der Lebensdauer der sie benutzenden Anwendungen, abgelegt werden.

Im Rahmen der Softwareentwicklung ist es vorteilhaft, das Konzept der Persistenz auf die Programmvariablen mit ihren Werten der zu entwickelnden Programme auszudehnen. Dadurch wird es möglich, Testdaten zu generieren und bereits erzeugte Daten weiterzuverwenden. Dabei ist die Granularität mit der der Datenzugriff erfolgen kann, von großer Bedeutung.

Der Begriff der Granularität wird an dieser Stelle im Zusammenhang mit Daten gebraucht. Die Granularität liefert eine Aussage über die Größe der Dateneinheiten, auf die der Benutzer zugreifen kann. Je kleiner die Dateneinheit ist, auf die in einem Schritt zugegriffen werden kann, desto feingranularer ist der Zugriff. Dies ist besonders im Zusammenhang mit Datenbanken wichtig. Je feingranularer der Zugriff auf die Daten einer Datenbank ist, desto effizienter in Bezug auf die Performanz ist der Zugriff, da nur eine kleine Information eingelesen werden muß. Dadurch kann gewährleistet werden, daß keine unnötigen Informationen eingelesen werden. Dementgegen steht der Verwaltungsaufwand, der mit einem feingranularen Datenzugriff verbunden ist. Je kleiner die Dateneinheiten sind, desto mehr Verwaltungsaufwand entsteht, da die Verwaltungstabellen größer und damit schlechter zu handhaben sind.

Die Lebensdauer von Programmvariablen entspricht normalerweise der Ausführungszeit des sie erzeugenden oder benutzenden Programms. Besteht jedoch die Möglichkeit, auch Programmvariablen persistent zu machen, so kann mit dem gleichen oder auch mit einem anderen Programm zu einem späteren Zeitpunkt, mit den gleichen Daten weitergearbeitet werden. Zur Realisierung eines feingranularen Persistenzkonzeptes ist es notwendig, daß die verwendete Programmiersprache die Möglichkeit zur Anbindung an eine Datenbank bietet, die eine feingranulare Datenverwaltung unterstützt. Sicherheitsaspekte zum Schutz der persistenten Daten müssen vom Laufzeitsystem der Programmiersprache unterstützt werden. Desweiteren muß die verwendete Datenbank die Verwaltung verschiedener Dokumenttypen erlauben und eine hohe Performanz bieten.

1.4 Die Prototyping-Sprache ProSet

ProSet ist eine prozedurale, mengen-orientierte Programmiersprache mit einem sehr hohen semantischen Niveau (VHLL) [DFG92]. Die Sprache erlaubt es dem Entwickler, Prototypen zu erstellen und stellt dem Programmierer Hilfsmittel in Form von mächtigen Operationen zur Modellierung von Algorithmen und von komplexen Datenstrukturen zur Verfügung. ProSet unterstützt Datenabstraktion, durch flexible Datenstrukturen (`tuple`,`set`...) sowie Kontrollabstraktion durch ein Ausnahmebehandlungskonzept. Weiterhin wird die Datenmodellierung, durch einen Persistenzmechanismus und parallele Programmierung unterstützt [DFG92].

1.4.1 Die Datentypen von ProSet

ProSet stellt drei Arten von Datentypen zur Verfügung, *primitive*, *zusammengesetzte* und *Datentypen höherer Ordnung*. Zu den primitiven Datentypen gehören `integer`, `real`, `boolean`, `string` und `atom`. Zu den zusammengesetzten Datentypen gehören Mengen (`set`) und Tupel (`tuple`). Diese Datentypen sind angelehnt an Vorlagen aus der endlichen Mengenlehre. Die zusammengesetzten Datentypen werden also, wie in der Mathematik, durch Aufzählen der Elemente oder Beschreibung der Elemente über Eigenschaften beschrieben.

Datentypen höherer Ordnung sind beispielsweise Funktionen (`function`) und Moduln (`module`). Ein weiterer wichtiger Aspekt der Sprache PROSET ist die schwache Typisierung. Sie ermöglicht es, daß sich die Typen der Variablen zur Laufzeit verändern können.

1.4.2 Persistenz in PROSET

Die Persistenz in PROSET soll es ermöglichen, Daten über die Programmausführung hinweg zu erhalten, um sie später wiederzuverwenden. In PROSET kann jeder Wert, der einen Typ mit Bürgerrechten erster Klasse besitzt, persistent gemacht werden. Typen mit Bürgerrechten erster Klasse können zugewiesen werden. Persistente Werte werden von PROSET in sogenannten P-Files abgelegt. Diese P-Files werden von einer Datenbank verwaltet. Um innerhalb eines PROSET-Programmes auf einen persistenten Wert zugreifen zu können, muß der Programmierer bei der Deklaration einer solchen Variable das Attribut `persistent` vor den Variablennamen schreiben und den Namen des P-Files angeben, in dem der Wert abgelegt wurde. Zusätzlich kann einer Variablen das Attribut `constant` zugeordnet werden, das zum Ausdruck bringt, daß dieser Wert nicht geändert werden kann.

1.4.3 Ein Beispiel

In der Abbildung 4 ist ein Programm zur Lösung des n-Damen Problems für n=4 beschrieben (siehe [DFK93]). Das n-Damen Problem besteht darin, auf einem n x n Schachbrett n Damen so zu positionieren, daß sie sich nicht gegenseitig schlagen können.

Im Rahmen dieses Programms werden zwei persistente Variablen deklariert und den Variablennamen `npow` und `abs` zugeordnet, die im P-File `StdLib` abgelegt sind. Die Funktion `npow (k,s)` liefert alle Teilmengen von `s`, die genau `k`-Elemente enthalten. Die Funktion `abs` liefert den Absolutwert des ihr übergebenen Ausdrucks zurück. Ob sich die Damen in der Position `Position` schlagen können, überprüft die Prozedur `NonConflict`. Das i-te Element eines Tupel `T` wird durch den Ausdruck `T(i)` geliefert.

Das Programm liefert die Menge aller Positionen aus, in denen sich die n Damen nicht gegenseitig schlagen können. Die Positionen werden dabei durch Tupel, die jeweils aus den Koordinaten für eine in Frage kommende Position bestehen, dargestellt.

```
program Queens;
   constant N := 4;
   persistent constant npow, abs : "StdLib";
begin
   fields := {[x,y]: x in [1..N], y in [1..N]};
   put ({NextPos:
         NextPos in npow(N,fields)|
         NonConflict(NextPos)});

   procedure NonConflict (Position);
   begin
      return forall  F1 in Position,
                     F2 in Position |
                     ((F1/=F2) !implies
                      (F1(1)/=F2(1) and
                      (F1(2)/=F2(2) and
                      abs (F2(1)-F1(1)/=
                       abs (F2(2)-F1(2))));

      procedure implies (a,b);
      begin
         return not a or b;
      end implies;

   end NonConflict;

end Queens;
```

Abbildung 4: PROSET Beispiel-Programm

1.4.4 Parallelität in PROSET

Viele Problemstellungen aus dem Gebiet der Softwareentwicklung beinhalten Teilbereiche in denen Parallelität ein Rolle spielen kann. Ein Prototypingsprache sollte deshalb auch die parallele Programmierung unterstützen. PROSET bietet Operationen zur Prozeßkreation und zur Koordination paralleler Prozesse an. Die Unterstützung erfolgt dabei durch Multilisp's Futures für die Prozeßkreation und dem Linda-Konzept der Tupelräume. Weiter Informationen dazu finden sich in [Gel85] und [Hal85].

1.5 H-PCTE

In diesem Abschnitt werden kurz die grundlegenden Begriffe zum Verständnis von H-PCTE erläutert. Dabei werden nur die wichtigsten Attribute von H-PCTE beschrieben, auf detaillierte Darstellungen wurde an dieser Stelle verzichtet. Weitere Informationen finden sich in der angegebenen Literatur [Kel91], [Kel93], [Heu92].

1.5.1 Begriffe

Die Objektdatenbank H-PCTE basiert auf dem ER-Modell. Das bedeutet, daß Objekte verschiedenen Typs in der Datenbank abgelegt und durch Links (Beziehungen) verknüpft werden können.

ECMA-PCTE

PCTE (Portable Common Tool Environment), ist eine strukturell objektorientierte Datenbank, die einen Schnittstellen-Standard für Software-Entwicklungs-Umgebungen definiert. PCTE wurde zur Unterstützung von maschinen-unabhängiger Software-Entwicklungsarbeit entworfen. ECMA-PCTE richtet sich im wesentlichen nach dem ECMA-Referenzmodell, das eine Rahmenarchitektur für Software-Entwicklungsumgebungen darstellt (siehe Abbildung).

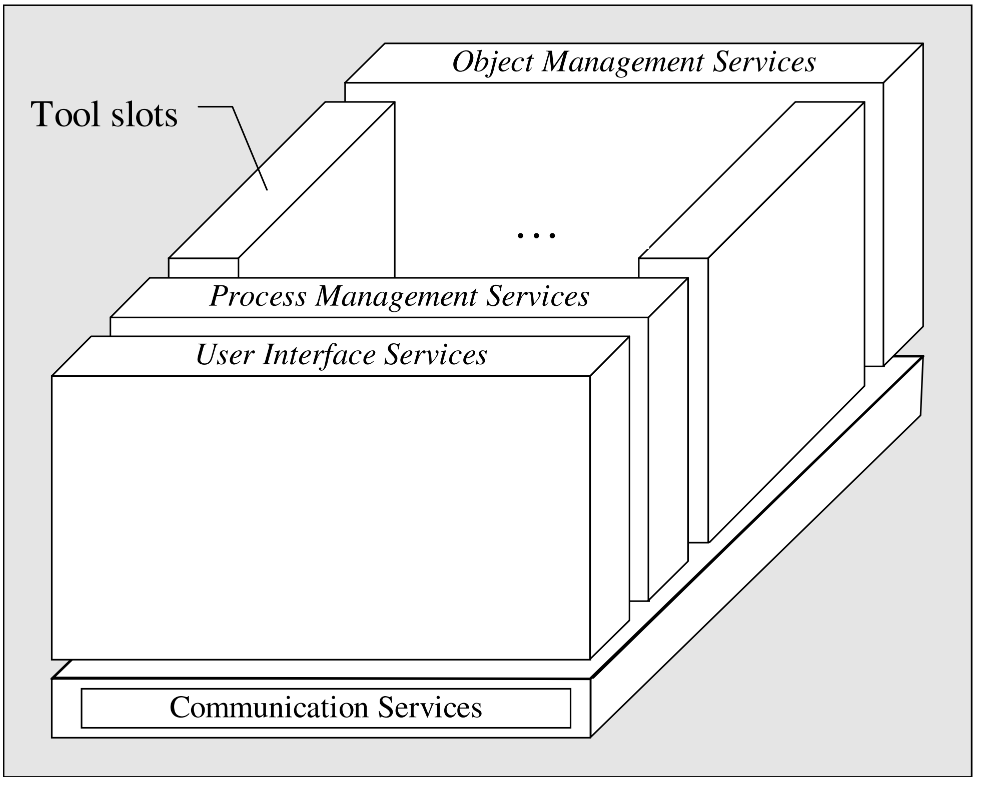

Abbildung 5: ECMA-Referenzmodell für Software-Entwicklungs-Umgebungen [HS93]

Die von PCTE angebotene Funktionalität unterstützt dabei im wesentlichen die Fähigkeit, eine konsistente Basis für Werkzeuge innerhalb einer Software-Entwicklungs-Umgebung zu bilden. Aus dem PCTE-Projekt ging ein Standard für eine Werkzeugschnittstelle hervor, der von der European Computer Manufacturers Association (ECMA) fixiert wurde, ECMA-PCTE [ECM93]. Als eine Weiterentwicklung des ECMA-PCTE-Standards entstand H-PCTE (siehe auch Abschnitt 1.5.2).

Objekt/Objekttyp

Ein Objekttyp in H-PCTE entspricht der abstrakten Datenstruktur eines Verbundes (Records), wie man sie auch in gängigen Programmiersprachen wie Pascal oder C vorfindet.

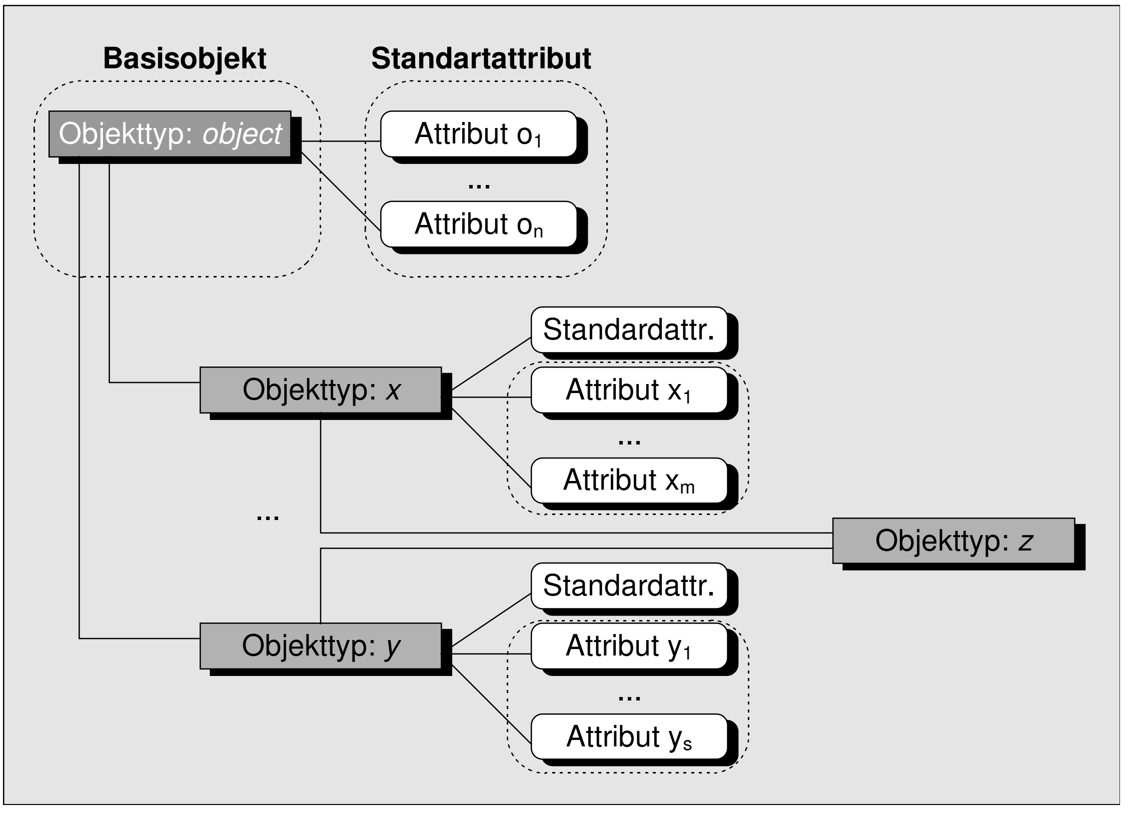

Abbildung 6: Objekthierarchie

Ein Objekttyp kann eine Menge von Attributen besitzen, die die einzelnen Eigenschaften des Objekts beschreiben. Zusätzlich können einem Objekttyp eine Menge von ausgehenden Linktypen zugeordnet werden. Ein Objekttyp kann einen oder mehrere Elterntypen haben. Von seinen Elterntypen erbt der Objekttyp alle Attribute und zulässigen Linktypen. Die Objekttyp-Hierarchie hat genau eine Wurzel, den Objekttypen *object*. Die Attribute vom Objekttypen *object* werden an alle abgeleiteten Objekttypen vererbt und als *Standardattribute* bezeichnet. Zusätzlich hat ein neu definierter Objekttyp Spezialisierungen, die durch weitere Attribute zum Ausdruck kommen ($x_1..x_n,y_1..y_s$). Innerhalb der Datenbank können Instanzen von Objekttypen verwaltet werden, diese werden als Objekte bezeichnet. Objekte in H-PCTE werden durch eindeutige Namen identifiziert (siehe Abbildung 6).

Link/Linktyp
Die Links innerhalb von H-PCTE entsprechen Beziehungen zwischen den Objekten. Ebenso wie Objekte, besitzen Links Attribute und eine zulässige Menge von Zielobjekten auf die sie zeigen können. Es gibt verschiedene Kategorien von Links, durch die bestimmte semantische Eigenschaften, d.h. Beziehungen zwischen Objekten, ausgedrückt werden können. Auf die einzelnen Linktypen und ihre Funktionen wird hier nicht näher eingegangen. Siehe dazu auch [Kel91] und [Kel93]. Links werden in H-PCTE ebenfalls durch eindeutige Namen identifiziert (siehe Abbildung 7).

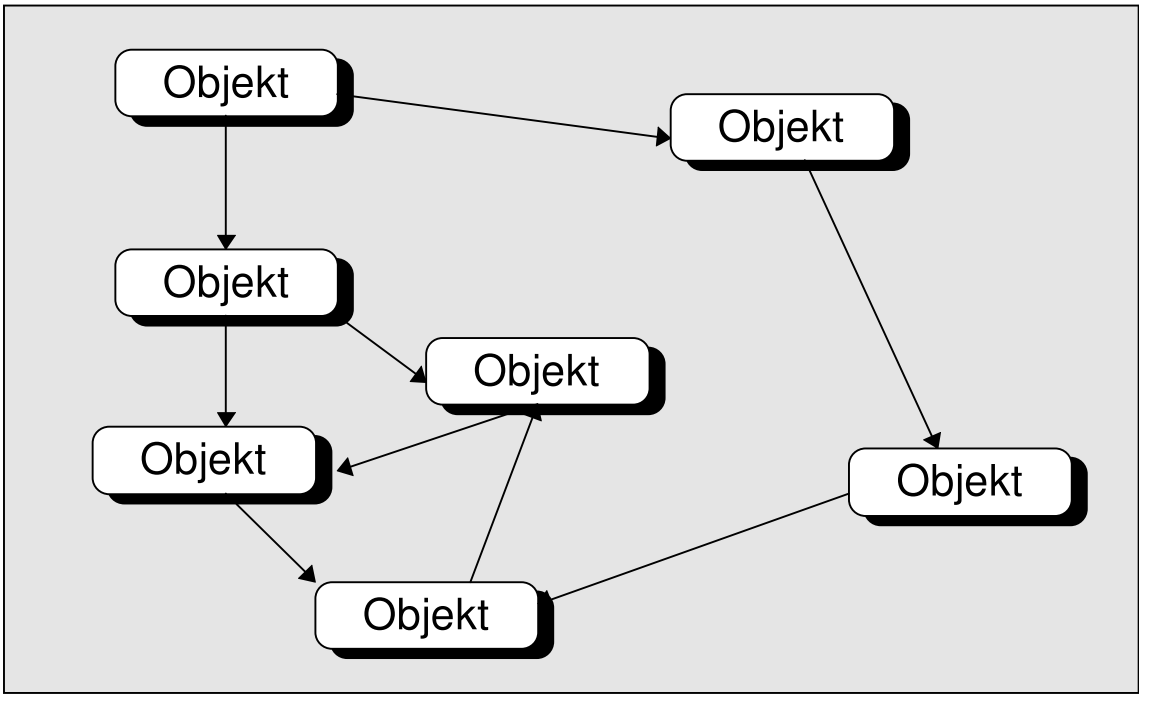

Abbildung 7: Mit Links verknüpfte Objekte

SDS

Um einer H-PCTE-Installation Typen von Objekten und Links bekannt zu machen, müssen diese vor ihrer Benutzung in einem Schema Definition Set (SDS) definiert werden. Ein SDS gruppiert eine Menge zusammengehörender Typdefinitionen und muß systemweit einen eindeutigen Namen haben. Die Typnamen sind innerhalb eines SDS lokal eindeutig. H-PCTE bietet eine Reihe von vordefinierten SDSs an, die die Basistypen zur Arbeit mit H-PCTE zur Verfügung stellen. Darüber hinaus können vom Benutzer weitere SDS formuliert werden, um die Funktionalität von H-PCTE und die standardmäßig verfügbaren Typen zu erweitern. Typen, die in neuen SDSs definiert werden, sollten auf einem Basistyp, wie etwa `object`, aufbauen. Ein SDS wird in Form einer Text-Datei beschrieben, wobei die Defintion der Typen einer festen Syntax folgt. Diese Text-Dateien können danach durch einen SDS-Compiler (*DDL*) übersetzt und in H-PCTE eingebunden werden (siehe Abbildung 8).

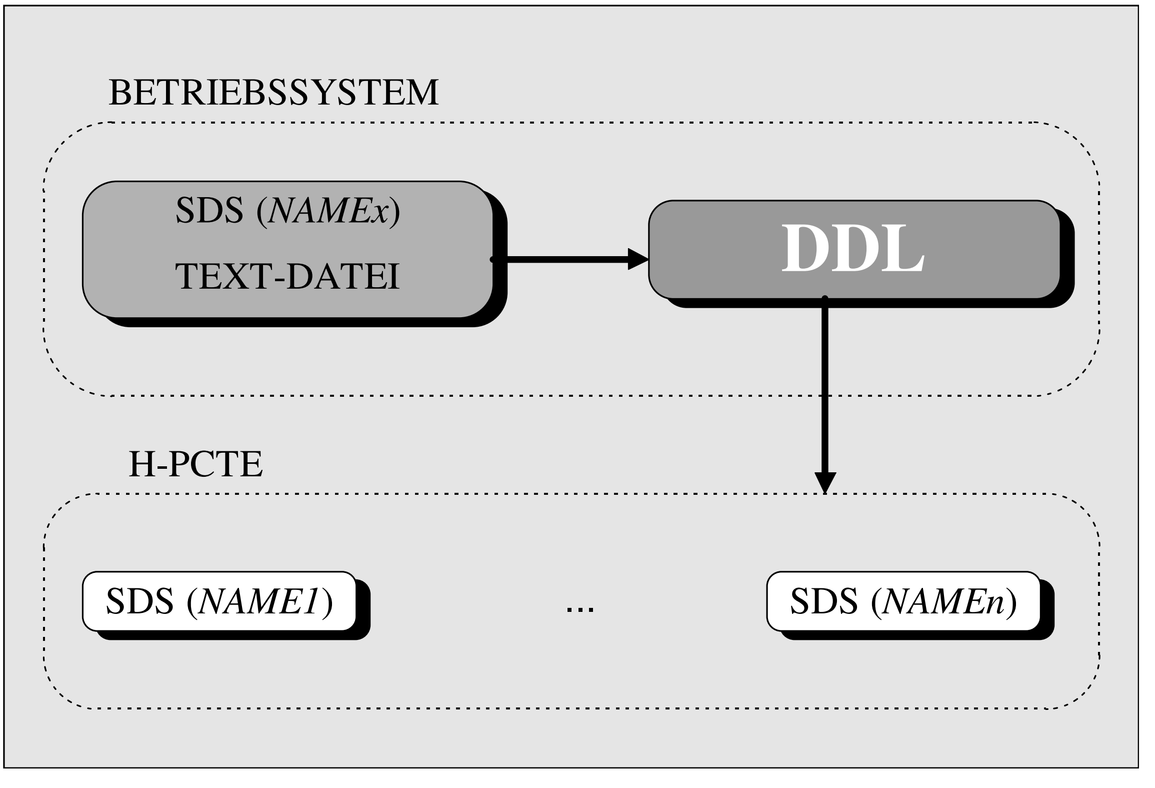

Abbildung 8: SDS Definition und Einbindung

Strukturelle Objektorientiertheit

Der Begriff der strukturellen Objektorientiertheit bezieht sich auf die Datenbank und die Art ihrer Objektorientiertheit. Der Begriff *objektorientiert* beinhaltet, daß Daten und Funktionalität in einem Objekt gebunden sind. Für die Datenbank würde dies bedeuten, daß die Methoden um mit den Datenobjekten zu arbeiten, an die Datenobjekte gebunden und damit ebenfalls in der Datenbank abzulegen sind. Dies ist bei struktureller Objektorientiertheit nicht der Fall. Hier werden nur die Daten als Objekte behandelt und in beliebiger Struktur in der Datenbank abgelegt. Die Funktionalität ist jedoch nicht an die Datenobjekte gebunden, sondern wird von zentraler Stelle zur Verfügung gestellt. Diese zentrale Stelle ist die Datenbank als solche, die eine Benutzungsschnittstelle zur Verfügung stellt, um Funktionen auf Datenobjekte anwenden zu können.

1.5.2 Die Datenbank H-PCTE

H-PCTE (Highperformant - Portable Common Tool Environment) ist ein verteiltes, strukturell objektorientiertes Datenbankmanagementsystem, das als eine dedizierte Betriebssystem-schnittstelle für eine Software-Entwicklungs-Umgebung genutzt werden kann und die Konstruktion von Software-Entwicklungs-Umgebungen erleichtern soll.

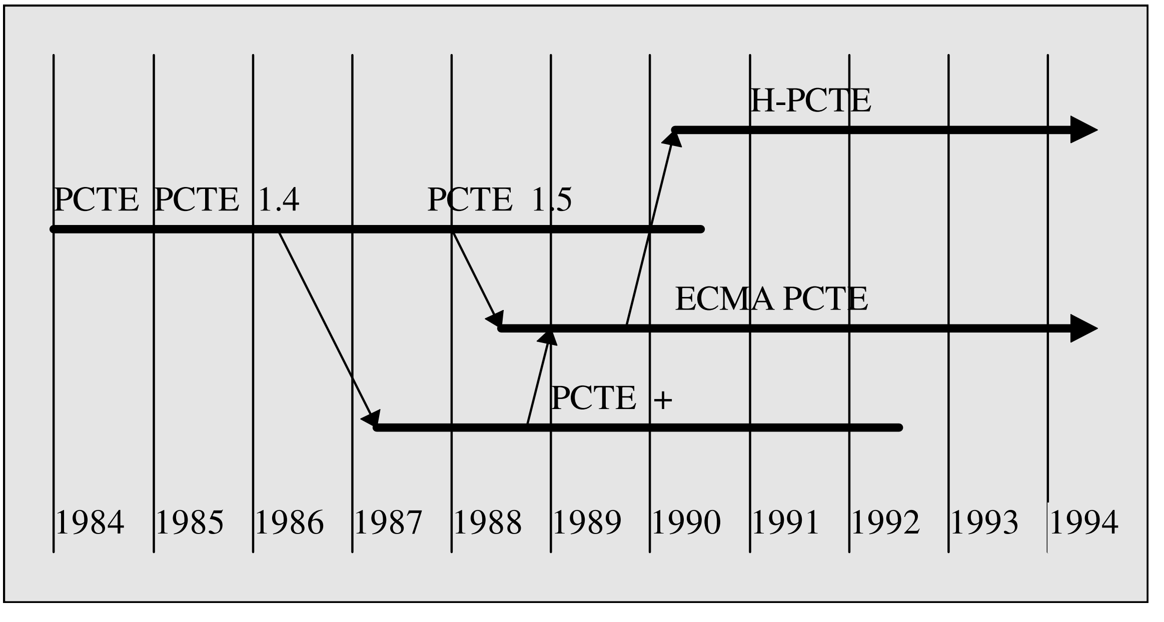

Abbildung 9: Entwicklung der verschiedenen PCTE-Versionen

H-PCTE baut auf ECMA-PCTE auf (siehe Abbildung 9) und ist wie der Name schon sagt eine hochperformante beziehungsweise hauptspeicherbasierte Version von ECMA-PCTE. Durch H-PCTE wird insbesondere die Integration von Werkzeugen unterstützt [Kel91]. Der

Datenzugriff ist navigierend und erfolgt über verschiedene Typen von Links. H-PCTE unterstützt weitergehende Konzepte aus den folgenden Funktionsbereichen:

- Objektverwaltung
- Prozesse
- Verteilung
- Zugriffskontrolle
- Abrechnungsfunktionen

1.6 Gliederung der Arbeit

In Kapitel 2 wird die Aufgabenstellung der Diplomarbeit näher erläutert. Die Teilaufgaben, Schnittstellenentwicklung und Werkzeugentwicklung, werden in einzelnen Abschnitten beschrieben.

Kapitel 3 erläutert die Anforderungen, die an eine Prototyping-Umgebung gestellt werden. Die Anforderungen werden rollenspezifisch betrachtet, um daraus die von den verschiedenen Nutzern der Prototyping-Umgebung benötigten Werkzeuge abzuleiten. Desweiteren ergibt sich aus den Betrachtungen die Notwendigkeit des in einem späteren Kapitel behandelten Meta-Tools.

Danach wird im 4. Kapitel die Konstruktion der Schnittstelle und der damit in Zusammenhang stehenden Elemente beschrieben.

Das aus Kapitel 3 abgeleitete Meta-Tool wird im 5. Kapitel erläutert.

Im 6. Kapitel wird die Erzeugung der konkreten Werkzeuge mit Hilfe des Meta-Tools beschrieben. Abschließend wird die weitere Implementation der Funktionalität der Werkzeuge erläutert. Dieses Kapitel beschreibt gleichzeitig die Verifikation des in Kapitel 5 konstruierten Meta-Tools.

Kapitel 7 gibt noch einmal einen zusammenfassenden Überblick über den Inhalt der Diplomarbeit und Kapitel 8 beinhaltet einen Ausblick auf mögliche Erweiterungen, die im Rahmen der in der Diplomarbeit betrachteten Problemstellungen interessant sein könnten.

Der Anhang beschreibt das allgemeine Vorgehen, bei der Generierung eines neuen Werkzeugs.

Die vorliegende Diplomarbeit umfaßt zwei Teile. Teil I beinhaltet die schriftliche Ausarbeitung und Teil II beinhaltet den Sourcecode der generierten und programmierten Werkzeuge. Der Teil II wurde hinterlegt am Lehrstuhl X Fachbereich Informatik der Universität Dortmund und kann dort eingesehen werden.

2 Aufgabenstellung

Die Basis der Diplomarbeit bilden die beiden Komponenten, die in den Abschnitten 1.2 und 1.3 beschrieben wurden, die Prototypingsprache PROSET und die Datenbank H-PCTE. Aufbauend auf diese beiden Komponenten werden im Rahmen dieser Diplomarbeit zwei Aufgaben bearbeitet. Die erste Aufgabe ist die Konstruktion einer Schnittstelle zwischen PROSET und H-PCTE. Die Schnittstelle soll es ermöglichen, PROSET-Daten unter H-PCTE abzulegen und zu verwalten. Im zweiten Schritt sollen aufbauend auf dieser Schnittstelle zwei Aufgabenbereiche bearbeitet werden.

Der erste Aufgabenbereich beschäftigt sich mit der Neu- und Weiterentwicklung sowie der Implementierung der internen Funktionen. Die internen Funktionen sind die Funktionen, die von der Programmiersprache PROSET genutzt werden, um das Persistenzkonzept zu realisieren. Im Bereich dieser internen Funktionen sollen unter anderem Konzepte zu den Bereichen Transaktionen, Schutzmechanismen und Parallelität erarbeitet werden. Dieser Aufgabenbereich wird in einer zweiten Diplomarbeit bearbeitet und soll deshalb hier nicht weiter berücksichtigt werden. Siehe dazu [Kap95].

Der zweite Aufgabenbereich, der im Rahmen dieser Diplomarbeit bearbeitet werden soll, besteht in der Konstruktion verschiedener Werkzeuge zur Verwaltung der Daten der PROSET-Entwicklungs-Umgebung und insbesondere der PROSET-Daten. Dabei müssen sowohl allgemeine Anforderungen der Prototyping-Umgebung als auch PROSET spezifische Anforderungen berücksichtigt werden. Unter allgemeinen Anforderungen der Prototyping-Umgebung werden hier zum Beispiel Integrationsaspekte berücksichtigt. PROSET spezifische Anforderungen beziehen sich im wesentlichen auf die spezielle Handhabung der PROSET eigenen Datentypen.

2.1 Die Schnittstelle zwischen PROSET & H-PCTE

Aufbauend auf der Implementation des Persistenzkonzeptes in PROSET soll eine Schnittstelle zwischen PROSET und H-PCTE entworfen und implementiert werden. Dazu muß zuerst auf H-

PCTE-Ebene ein SDS entworfen werden, das es ermöglicht PROSET-Daten unter H-PCTE abzulegen und zu verwalten. Danach muß eine auf das SDS aufbauende Schnittstelle konstruiert werden, die zum einen die Anbindung der Sprache PROSET an H-PCTE gewährleistet und zum anderen die von externen Werkzeugen und Applikationen benötigten Funktionen und Dienste berücksichtigt.

2.2 Die Werkzeuge

Der zweite Aufgabenbereich beschäftigt sich mit dem Entwurf und der Implementation von Werkzeugen zur Benutzung und Verwaltung von PROSET-Daten. Beim Entwurf der Werkzeuge sind zwei wesentliche Einflußfaktoren zu berücksichtigen.

1. Die allgemeinen Anforderungen, die durch eine Prototyping-Umgebung gestellt werden, müssen bereits beim Entwurf des Konzepts für die später konkret zu realisierenden Werkzeuge berücksichtigt werden. Darunter fallen zum einen die Anforderungen, die durch die verschiedenen, innerhalb der Prototyping-Umgebung vorhandenen Benutzer-Rollen entstehen. Zum anderen sind beim Entwurf der Werkzeuge allgemeine Integrationsaspekte zu berücksichtigen, die sich auch auf die spätere Entwicklung weiterer Werkzeuge auswirken. Hier muß auf ein einheitliches Konzept geachtet werden. Das bedeutet, daß das Konzept nach dem die Werkzeuge entworfen und implementiert werden sollen, auf Werkzeuge die zu einem späteren Zeitpunkt konstruiert werden sollen, übertragbar sein muß.

2. PROSET-spezifische Anforderungen entstehen in Bezug auf den Entwurf der konkret zu realisierenden Werkzeuge. Hier muß auf eine PROSET-orientierte Realisierung geachtet werden, die eine möglichst optimierte Verwaltung der PROSET-Daten unterstützt. Von PROSET benötigte Datentypen müssen im SDS als Objekttypen mit entsprechenden Attributen formuliert werden. Die P-File-Struktur muß mit Hilfe von Objekten mit entsprechenden Links nachgebildet werden.

Konzeptionell soll sich die Diplomarbeit damit befassen, inwieweit Dienste zur Verwaltung der persistenten Daten in die Sprache PROSET integriert werden können oder ob sie in externe Werkzeuge ausgelagert werden sollten (interner / externer Funktionsbereich). Dabei sollen die für den Zugriff auf PROSET-Daten benötigten Grundoperationen analysiert und die notwendigen Funktionen in die zu entwerfende Schnittstelle zwischen PROSET und H-PCTE integriert werden. Die Konzepte für die zu entwerfenden Werkzeuge sollen übertragbar und offen sein, damit auch zu einem späteren Zeitpunkt neu zu entwickelnde Werkzeuge mit vertretbarem Aufwand integriert werden können.

Im praktischen Teil dieser Arbeit soll zuerst das H-PCTE-SDS und darauf aufbauend, die für die Kommunikation zwischen ProSet und H-PCTE notwendige Schnittstelle implementiert werden. Anschließend soll ein allgemeines Konzept für die Werkzeugkonstruktion erarbeitet werden, das als Basis für die Konstruktion der konkret zu realisierenden Werkzeuge dient. Desweiteren kann für die entworfenen Werkzeuge eine integrierte graphische Oberfläche konstruiert werden. Bei der Implementierung sollen besonders die P-File-Verwaltungswerkzeuge sowie Werkzeuge zur Datenanalyse berücksichtigt werden.

3 Anforderungen

In diesem Kapitel soll analysiert werden, welche Anforderungen durch eine Prototyping-Umgebung bzgl. der benötigten Werkzeuge gestellt werden. Dabei müssen die verschiedenen Rollen berücksichtigt werden, die im Rahmen des Prototyping anfallen. Die Anforderungen sollen daher rollenspezifisch analysiert werden. Im wesentlichen müssen folgende Rollen berücksichtigt werden:

1. Der Benutzer
   nutzt eine Applikation und gibt dem Entwickler die Anforderungen, die an ein zu entwickelndes Produkt zu stellen sind. Er ist in diesem Zusammenhang auch Validierer der Prototypen.
2. Der Entwickler
   konstruiert die Prototypen und das Produkt, das später vom Benutzer angewendet werden soll.
3. Der Administrator
   wartet die Entwicklungsumgebung und verwaltet Rechte und Ressourcen.
4. Der Werkzeugentwickler
   konstruiert Werkzeuge, die in die Entwicklungsumgebung integriert werden sollen.

3.1 Anforderungen aus der Sicht des Benutzers

Die Rolle des Benutzers soll hier durch die von ihm zu erbringenden Aufgaben charakterisiert werden. Zu den Aufgaben des Benutzers zählt unter anderem das Validieren der Prototypen. Um einen Prototypen benutzen zu können, braucht der Benutzer folglich Zugriff auf die ausführbaren Prototypen. Der Zugriffsrahmen des Benutzers wird in Abbildung 10 dargestellt. Die Möglichkeit auf ausführbare Programme zugreifen zu können, muß andererseits auf bestimmte Bereiche eingeschränkt sein, da es dem Benutzer nicht erlaubt sein sollte, auf Entwicklungs- oder Administrationswerkzeuge und -daten zuzugreifen. Seine Aufgabe beschränkt sich nur auf das Validieren der Prototypen und das Anpassen der Anforderungen. Aus diesem Grund müssen im Rahmen der Rolle des Benutzers Schutzmechanismen greifen, die ihn und das System vor unerlaubten Zugriffen schützen. Bei der Ausführung eines Prototypen braucht der Benutzer zusätzliche Rechte für den Zugriff auf Programmdaten und auf die Strukturen in denen die Programmdaten in der Datenbank abgelegt sind.

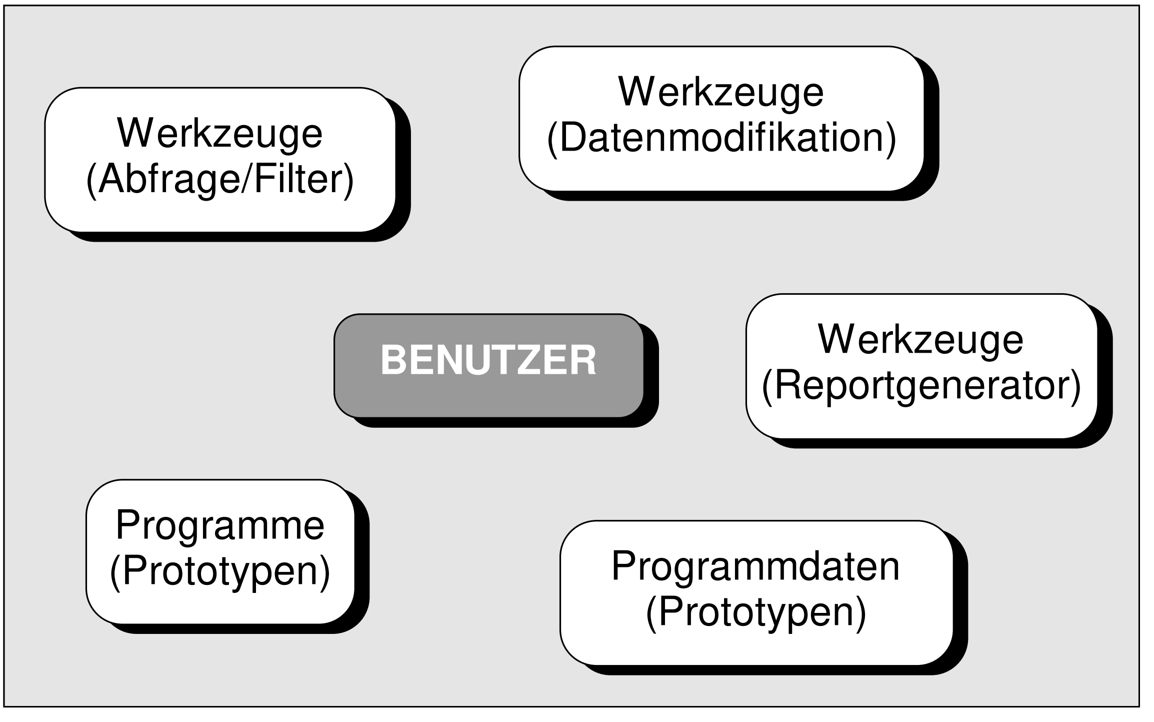

Abbildung 10: Zugriffsrahmen des Benutzers

Dieser Zugriff ist auf der einen Seite zur Ausführung des Prototypen notwendig und auf der anderen Seite auch für die Bewertung der konstruierten Prototypen. Die Bewertung des Prototypen bezieht sich im wesentlichen darauf, ob er die an ihn gestellten Anforderungen erfüllt. In diesem Zusammenhang können durch den Benutzer gegebenenfalls auch neue Anforderungen formuliert werden, die im Rahmen der vorhergehenden Anforderungsanalyse übersehen wurden. Um dem Benutzer die Ergebnisbewertung zu erleichtern, sollten ihm Möglichkeiten zur Verfügung stehen, um seine Daten auszuwerten und zu modifizieren. Im Bereich der Auswertung kann dies ein Abfragetool leisten, das es erlaubt verschiedene Sichten auf die Daten zu realisieren und sie in gewissem Rahmen auszuwerten. Für die Modifikation der Programmdaten können spezielle Werkzeuge realisiert werden, die es dem Benutzer erlauben, einzelne Datenzustände zu erzeugen, um spezielle Situationen zu simulieren und zu testen. Der Begriff Datenzustände bezieht sich dabei auf eine spezielle Beschaffenheit der Programmdaten des Prototyps. Ein weiteres Werkzeug, das im Rahmen der Benutzerrolle sinnvoll erscheint, ist ein Reportgenerator. Dieses Werkzeug soll es erlauben, die Daten auszuwerten und zusammenzufassen, so daß der Benutzer sie später zur Änderung der Anforderungen an den nächsten zu erstellenden Prototypen verwenden kann.

3.2 Anforderungen aus der Sicht des Entwicklers

Die Aufgabe des Entwicklers ist die Konstruktion der Prototypen, entsprechend den vom Benutzer gestellten Anforderungen. Zur Konstruktion eines solchen Prototypen benötigt der Entwickler Zugriff auf die Entwicklungswerkzeuge. Der Zugriffsrahmen des Entwicklers wird in Abbildung 11 dargestellt. In welchem Rahmen auf die Entwicklungswerkzeuge zugegriffen werden soll, hängt von den weiteren spezifischen Aufgaben des Entwicklers ab. Unter spezifischen Aufgaben sind Einzelaufgaben zu verstehen, die im Rahmen der Aufgabenteilung entstehen können. So ist es möglich, daß ein Entwickler nur für einen speziellen Teilbereich der Problemstellung zuständig ist und keinen Zugriff auf die Daten anderer Entwickler haben soll. Es kann also notwendig sein Schutzmechanismen einzuführen, um zu verhindern, daß ein Entwickler in Aufgabenbereichen tätig wird, in denen er nicht tätig werden soll.

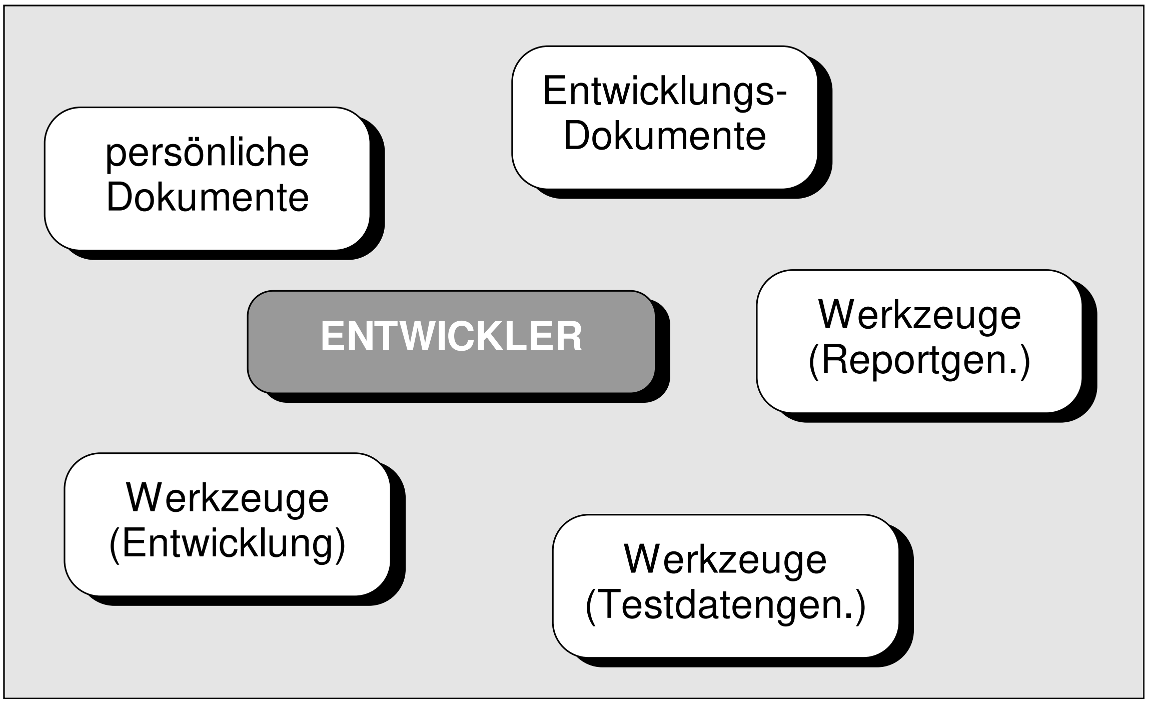

Abbildung 11: Zugriffsrahmen des Entwicklers

Weitere Bereiche, auf die der Entwickler Zugriff haben muß, sind zum einen seine Anwender-Dokumente und zum anderen die Entwicklungsdokumente. Der Zugriff auf die Entwicklungsdokumente kann dabei aufgrund spezifischer Aufgaben eines einzelnen Entwicklers, entsprechend dem Zugriff auf die Entwicklungswerkzeuge, eingeschränkt sein. Bezogen auf den Zugriff auf die Entwicklungsdokumente, entstehen von Seiten des Entwicklers weitere Anforderungen. Er muß die Möglichkeit haben, verschiedene Sichten auf die Daten zu realisieren, wofür ihm spezielle Werkzeuge zur Verfügung gestellt werden müssen. Sichten bezeichnen dabei die Einschränkung der angezeigten Daten, aufgrund eines vom Benutzer vorgegebenen Kriteriums. Weiterhin benötigt er Werkzeuge zur Übersetzung der Programme und eventuell Werkzeuge die Testmethodiken unterstützen. Ebenso muß er, wie auch der Benutzer, Strukturen zur Ablage anwenderspezifischer Daten in seinem Zugriffsrahmen modifizieren können.

Anforderungen in Bezug auf die Datenbank entstehen auch aus Entwicklersicht. Die Datenbank muß die verteilte Programmentwicklung unterstützen und die verschiedenen Entwickler vor gegenseitiger Beeinflussung schützen. Diese Bedingungen sind zur Erhaltung eines konsistenten Datenbestandes zwingend notwendig und sollten zusätzlich durch ein Versions- und Konfigurationsmanagement unterstützt werden.

Werkzeuge, die im Rahmen der Rolle des Entwicklers wichtig sind, sind zum einen ein Testdatengenerator, der den Test erstellter Prototypen erleichtert, sowie ein Reportgenerator zur Auswertung der Datenbestände und zur Dokumentation des Entwicklungsprozesses, um gesammelte Erfahrungen bei späteren Entwicklungen wiederverwenden zu können.

3.3 Anforderungen aus der Sicht des Administrators

Die Aufgabe des Systemadministrators besteht im wesentlichen in der Installation und der Wartung des Systems.

In diesem Rahmen müssen ihm verschiedene Werkzeuge zur Verfügung stehen, die diese Tätigkeiten unterstützen. Der Zugriffsrahmen des Administrators wird in Abbildung 12 dargestellt. Grundlegend hat der Administrator Zugriff auf alle im System vorhandenen Daten und Werkzeuge, egal welcher Art. Um die administrativen Aufgaben zu erfüllen, sind verschiedenste Werkzeuge notwendig, die im folgenden genannt werden.

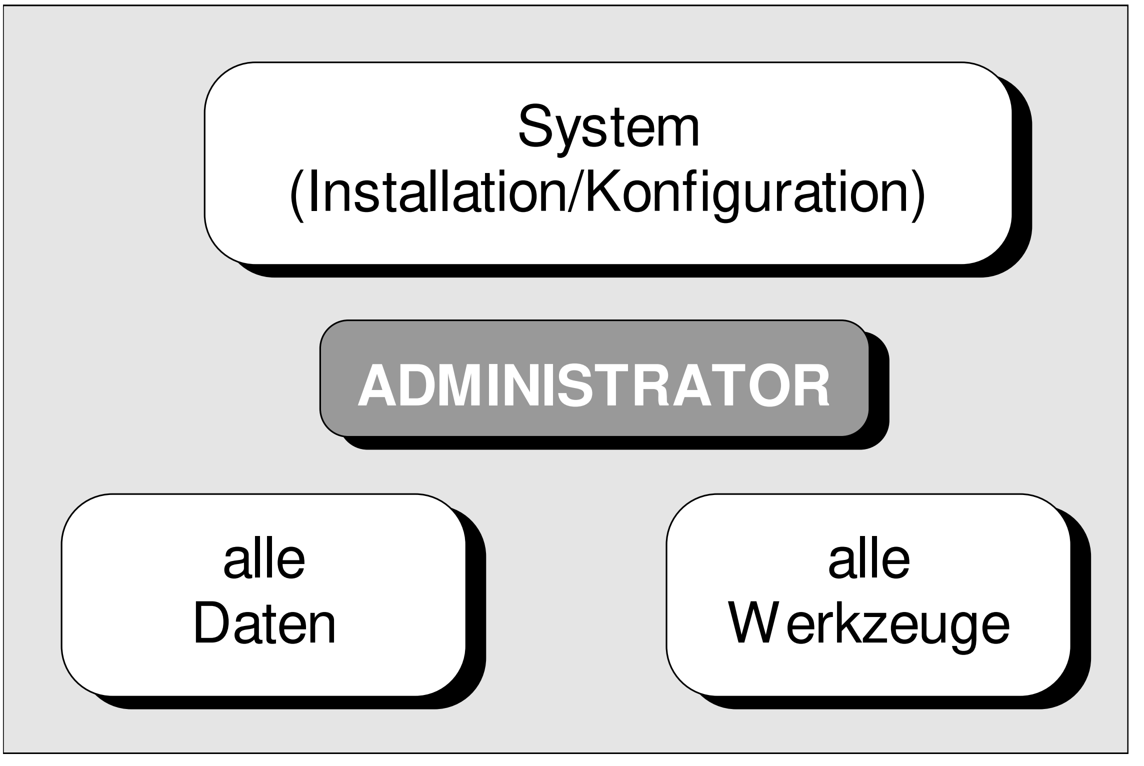

Abbildung 12: Zugriffsrahmen des Administrators

3.3.1 Installation und Konfiguration des Systems

Zur Installation des Systems benötigt der Administrator Werkzeuge zur Erzeugung von Strukturen für die Datenbestände sowie Werkzeuge zur Erzeugung initialer Datenbestände in der Datenbank. Diese Werkzeuge müssen es ihm auch zu späteren Zeitpunkten erlauben, die Installation beziehungsweise die Konfiguration des Systems den Umständen entsprechend anzupassen.

3.3.2 Sicherheitsaspekte des Systems

Im Rahmen der Sicherheitsaspekte kommt dem Administrator die Aufgabe der Datensicherung zu. Zu diesem Zweck sollte ein Werkzeug vorhanden sein, das die Einrichtung einer automatischen Datensicherung der Datenbestände sowie eine selektive Wiederherstellung eventuell verlorener Daten erlaubt.

Ein zweiter relevanter Punkt bei der Betrachtung von Sicherheitsaspekten ist die Benutzer-verwaltung. Der Administrator benötigt Werkzeugunterstützung, um Benutzer verwalten zu können. Die Benutzerverwaltung umfaßt dabei zum einen das Anlegen beziehungsweise Registrieren und das Löschen von Benutzern und zum anderen die Rechtevergabe an die einzelnen Benutzer. Unter Rechtevergabe ist hierbei zum einen die Erteilung von Zugriffsrechten auf Programme beziehungsweise Werkzeuge und zum anderen die Erteilung von Rechten bzgl. verschiedener Dokumente zu verstehen. Zusätzlich obliegt dem Administrator die Ressourcenvergabe des Systems.

Der dritte Punkt im Bereich der Sicherheitsaspekte betrifft die Reparatur des Systems. Durch nicht vorhersehbare Umstände können in den verschiedenen Bereichen des Systems Defekte entstehen. Zur Behebung dieser Defekte müssen dem Administrator neben der Möglichkeit der Datenwiederherstellung geeignete Werkzeuge zur Verfügung stehen, um die Funktionsfähigkeit des Systems wiederherstellen zu können.

3.3.3 Wartung des Systems

Die Wartung des Systems umfaßt neben der Instandsetzung auch Optimierungsaspekte. Zur Optimierung des Systems benötigt der Administrator Monitoring-Werkzeuge, die eine Analyse des Systemzustandes zulassen und es ermöglichen, potentielle Fehlerquellen und Performance-Engpässe zu erkennen. Die Optimierung kann sich dabei auf Bereiche wie Clustering, Replikation und Zugriffspfade (siehe auch [Kel91], [Kel92] und [Kel93]) beziehen und sollte zusätzlich durch einen Reportgenerator unterstützt werden, der die Dokumentation und Auswertung der Monitoring-Daten unterstützt.

3.4 Anforderungen aus der Sicht des Werkzeugentwicklers

Die Rolle des Werkzeugentwicklers ist bzgl. der Anforderungen, ähnlich der Rolle des Entwicklers von Prototypen. Die Aufgabe des Werkzeugentwicklers ist die Konstruktion von Werkzeugen, die später in die Umgebung integriert werden sollen. Zur Konstruktion der Werkzeuge benötigt der Entwickler Zugriff auf die Entwicklungswerkzeuge. Dabei ist wie beim Prototyp-Entwickler auf die Zugriffsrechte zu achten, da eventuell der Zugriff auf einzelne Werkzeuge nicht möglich sein soll. In welchem Rahmen auf die Entwicklungswerkzeuge zugegriffen werden soll, hängt von den weiteren spezifischen Aufgaben des Entwicklers ab. Auch der Werkzeugentwickler muß Zugriff auf seine persönlichen Dokumente und die Entwicklungsdokumente im Rahmen der Werkzeugentwicklung haben. Der Zugriff auf die Entwicklungsdokumente kann dabei, aufgrund spezifischer Aufgaben eines einzelnen Entwicklers entsprechend dem Zugriff auf die Entwicklungswerkzeuge, eingeschränkt sein. Spezifische Aufgaben bezeichnen hier wie auch beim Prototypentwickler Teilaufgaben, die getrennt von anderen Teilaufgaben zu bearbeiten sind und aus diesem Grund eine Einschränkung der Rechte notwendig machen können.

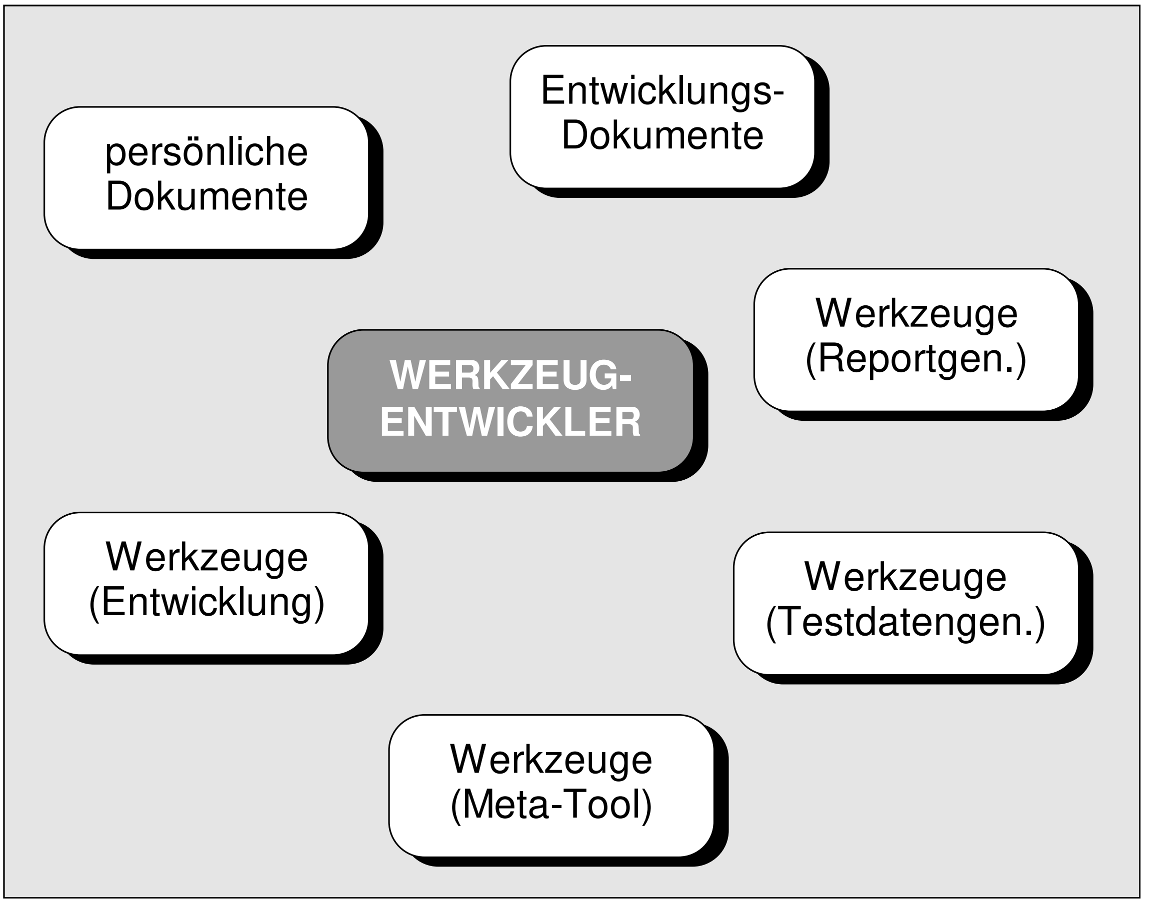

Abbildung 13: Zugriffsrahmen des Werkzeugentwicklers

Weitere Anforderungen entsprechen denen des Prototyp-Entwicklers:

- Es werden Werkzeuge zur Realisierung verschiedener Sichten auf die Daten benötigt.
- Es werden Werkzeuge zur Übersetzung (Compiler) und Überprüfung (Debugger) benötigt.
- Es werden Werkzeuge zur Modifikation von Anwender-Datenstrukturen benötigt.

Ein weiterer wichtiger Punkt der sich ausschließlich auf den Werkzeugentwickler bezieht, ist die Anforderung, daß die innerhalb einer Prototyping-Umgebung zu konstruierenden Werkzeuge integriert sein sollen (siehe auch Abschnitt 1.2.1). Das bedeutet, daß den Werkzeugen ein einheitliches Konzept zugrunde liegen muß, damit eine Einbindung in die

Gesamtumgebung mit möglichst geringem Aufwand realisiert werden kann. Um ein solches einheitliches Konzept durchzusetzen, ist die Vorgabe von Entwicklungsrichtlinien notwendig. An diese Entwicklungsrichtlinien muß sich der Entwickler bei der Konstruktion neuer Werkzeuge halten. Entwicklungsrichtlinien können in Form von schriftlich fixierten Konventionen für die Konstruktion von Werkzeugen realisiert werden. Der Nachteil bei dieser Methode ist schlechte beziehungsweise unmögliche Überwachung, ob die Richtlinien eingehalten werden. Die Überprüfung der Einhaltung der Richtlinien kann in der Regel nicht automatisiert werden. Die einzige Möglichkeit ist die Prüfung der Entwürfe durch einen zweiten Entwickler. Diese Möglichkeit würde jedoch zum einen wiederum eine Fehlerquelle beinhalten (den zweiten Entwickler/Menschen) und zum zweiten einen erheblichen Aufwand bedeuten. Das kann dazu führen, daß Inkonsistenzen auftreten, die zu einem späteren Zeitpunkt nur noch schwer oder gar nicht mehr zu beseitigen sind. Aus diesem Grund sollte ein Konzept vorhanden sein, das diese Fehlerquelle (Mensch) ausschließt.

Eine zweite Möglichkeit für die Durchsetzung eines einheitlichen Werkzeugkonzeptes ist die Konstruktion eines Meta-Tools. Das Meta-Tool übernimmt dabei die Aufgabe eines Codegenerators, der es ermöglicht, einheitliche Werkzeuge zu erzeugen. Die Aufgabe des Entwicklers besteht in diesem Fall darin, das erzeugte Basiswerkzeug mit Funktionalität zu füllen. Vorteil bei dieser Vorgehensweise ist die einheitliche Schnittstelle der erzeugten Werkzeuge. Dadurch kann den oben beschriebenen Kontrollproblemen aus dem Weg gegangen werden. Ein weiteres Werkzeug, das für diesen Ansatz notwendig ist, ist ein Werkzeug das die Modifikation vorhandener Werkzeuge zu einem späteren Zeitpunkt erlaubt. Aus diesem Grund ist ein solches Konzept zu bevorzugen, da es eine bessere Integration des Gesamtsystems und eine leichtere Wartung zuläßt.

3.5 Werkzeuge einer Prototyping-Umgebung

Aus den vorangegangenen Abschnitten läßt sich ableiten, welche Werkzeuge für eine Prototyping-Umgebung benötigt werden.

- **Meta-Tool**
  Das Meta-Tool dient zur Erzeugung konsistenter Werkzeuge.
- **Datenverwaltungs-Werkzeug**
  Das Datenverwaltungs-Werkzeug dient zur Verwaltung der Strukturen, in denen die

verschiedenen Dokumente abgelegt werden sollen und zur Erzeugung und Modifikation von Dokumenten.

- **Datenanalyse-Werkzeug**
  Das Datenanalyse-Werkzeug stellt einfache Funktionen zur Verfügung, um die Suche von Dokumenten beziehungsweise Dokumentinhalten zu erleichtern.
- Browser
  Der Browser dient zur Betrachtung der Daten/Dokumente und zur Realisierung verschiedener Sichten auf diese.
- Abfrage-Werkzeug
  Das Abfrage-Werkzeug stellt eine Query-Language zur Auswertung der Datenbestände zur Verfügung.
- Werkzeuge zur Realisierung von Schutzkonzepten
  Dieses Werkzeug ermöglicht die Schutzattributzuweisung an bestimmte Dokumente
- Systeminstallations-/Pflegewerkzeuge
  Diese Werkzeuge ermöglichen die Installation der Software-Entwicklungs-Umgebung und die spätere Anpassung der Konfiguration an die aktuellen Bedürfnisse.
- Datensicherungswerkzeug
  Dieses Werkzeug dient zur Datensicherung und selektiven Wiederherstellung beschädigter Daten und Dokumente.
- Reportgenerator
  Dieses Werkzeug dient zur Erstellung von Listen und Auswertungen, die die Analyse des Datenbestandes unterstützen.
- Reparatur-Werkzeuge
  Diese Werkzeuge dienen zur Reparatur von Defekten innerhalb des Systems, die sich durch die Wiederherstellung von Daten nicht beheben lassen.
- Optimierungs-/Monitoring-Werkzeug
  Dieses Werkzeug dient zur Kontrolle des Systems und zur Auswertung der Ressourcennutzung, sowie der Modifikation des Ressourcenzugriffs zum Zweck der Optimierung.
- Benutzerverwaltung
  Dieses Werkzeug dient zur Registration und Löschung von Benutzern sowie der Zuweisung von Zugriffsrechten auf Ressourcen.

Im Rahmen dieser Diplomarbeit sollen die ersten drei Punkte dieser Auflistung bearbeitet werden.

4 Die Schnittstelle zwischen PROSET & H-PCTE

In diesem Kapitel soll die Konstruktion der Schnittstelle zwischen PROSET und H-PCTE beschrieben werden. Die Vorgehensweise kann dabei in drei wesentliche Schritte unterteilt werden. Zunächst werden im ersten Abschnitt die Anforderungen an die Schnittstelle analysiert. Danach folgt im Abschnitt Design der Entwurf der Schnittstelle und ihre Modularisierung. Im dritten Abschnitt wird abschließend die Implementation der Schnittstelle beschrieben. Dabei werden nur die wesentlichen Merkmale beschrieben. Nähere Informationen zur Implementation der Schnittstelle finden sich bei [Kap95].

4.1 Anforderungen

Die Schnittstelle zwischen PROSET und H-PCTE soll es PROSET-Programmen und Werkzeugen der Prototyping-Entwicklungs-Umgebung ermöglichen, Werte in der Datenbank persistent abzulegen und zu einem späteren Zeitpunkt wiederzuverwenden.

Das zu betrachtende Gesamtsystem, sowie die Einordnung der Schnittstelle in dieses Gesamtsystem sind in Abbildung 14 dargestellt. Die Schnittstelle setzt auf H-PCTE auf und soll die von H-PCTE zur Verfügung gestellte Funktionalität kapseln. Da PROSET-Programme bei der Übersetzung zuerst in ANSI-C Code und danach mittels eines C-Compilers in eine ausführbare Datei übersetzt werden, soll die Schnittstelle selber ihre Funktionalität in Form von C-Funktionen zur Verfügung stellen. Diese können sowohl von PROSET-Programmen als auch von anderen Anwendungen genutzt werden. Die zur Verfügung gestellten Funktionen sollen zum einen die Verwaltung von PROSET-Daten und zum anderen weiterführende Konzepte unterstützen, die im Rahmen der Softwareentwicklung mit PROSET wichtig sind (siehe auch [Kap95]).

Beim Entwurf müssen sowohl allgemeine Anforderungen bzgl. der Schnittstellenkonstruktion als auch spezielle Anforderungen, die sich aus der Anbindung von PROSET ergeben, berücksichtigt werden. Diese Anforderungen werden in den folgenden Abschnitten besprochen.

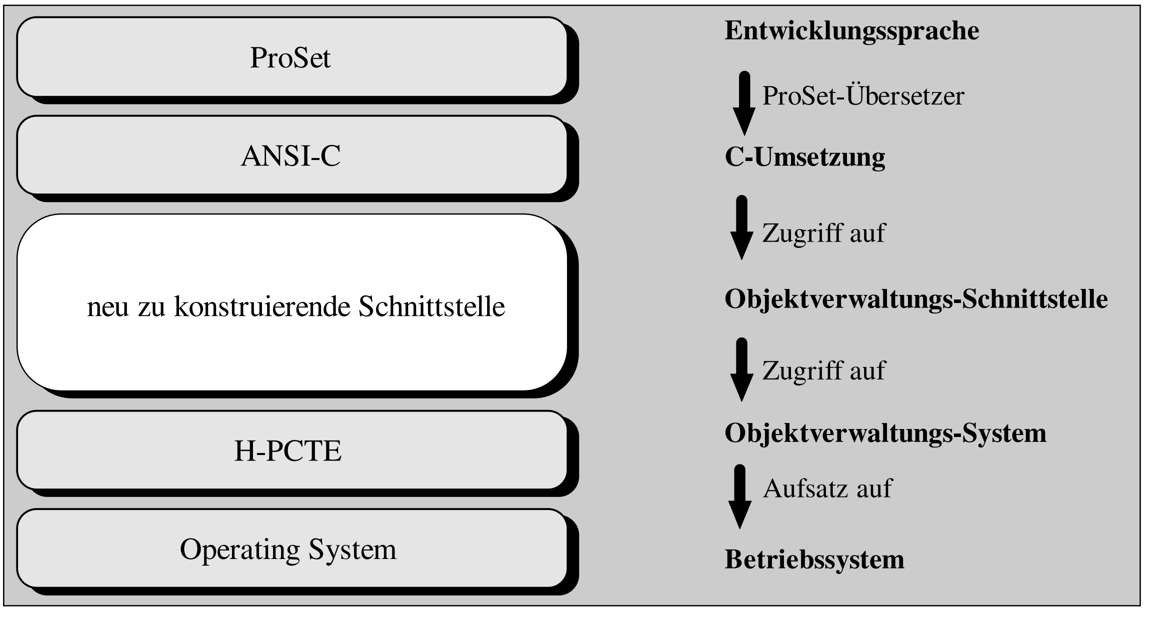

Abbildung 14: Einbettung der Schnittstelle

4.1.1 Allgemeine Anforderungen

Um einen Überblick über die Einordnung der Schnittstelle in das Gesamtsystem zu geben, wird der Zusammenhang zwischen der zu konstruierenden Schnittstelle und den übrigen Systemkomponenten in Abbildung 14 dargestellt.

Die Kommunikation zwischen PROSET und H-PCTE umfaßt im wesentlichen den Datenaustausch. In der hier zu konstruierenden Version der Schnittstelle sollen die Basisoperationen, die für einen solchen Datenaustausch notwendig sind, realisiert werden. Die Schnittstelle soll es ermöglichen, alle PROSET-Datentypen in der Datenbank abzulegen und weiterzuverarbeiten. Außerdem soll die Schnittstelle auch die Verwaltung weiterer in der Prototyping-Umgebung anfallender Dokumenttypen erlauben.

Die Implementation soll in ANSI-C auf einer SUN-Workstation mit SUN-OS erfolgen. Der ANSI-C-Standard soll aus zwei Gründen verwandt werden. Der erste Grund besteht in der größeren Portabilität die durch den ANSI-C-Standard gewährleistet wird. Hier sollen alle Elemente, die im Rahmen der Umgebung eingesetzt und konstruiert werden, auf einer gemeinsamen Basis aufbauen. Der zweite Grund besteht in der Tatsache, daß der PROSET-Compiler ebenfalls ANSI-C Code erzeugt. Um eine Basis für die Schnittstelle zu schaffen, muß zunächst ein SDS entworfen werden. Das SDS muß die Daten die in H-PCTE abgelegt werden sollen, auf H-PCTE-Objekte abbilden. Zusätzlich müssen alle notwendigen Verknüpfungen zwischen den Daten auf H-PCTE-Links abgebildet werden.

Ein besonders wichtiger Aspekt ist die Abbildung von P-Files, die PROSET als Datenstruktur zum Abspeichern von PROSET-Daten nutzt. Die Schnittstelle muß geeignete Funktionen zur Verwaltung von P-Files zur Verfügung stellen. Abbildung 15 zeigt eine typische P-File-Struktur. Der P-Store beinhaltet alle P-Files die in der Umgebung verwaltet werden. Aus der Abbildung läßt sich ableiten, daß die P-File-Struktur als Baumstruktur abzubilden ist. Hier sieht man zwei P-Files (P-File 1 und P-File 2), die im P-Store abgelegt wurden. P-File 1 enthält verschiedene Datenobjekte und ein Sub-P-File. Mit Hilfe von Sub-P-Files können auch geschachtelte Strukturen erzeugt werden.

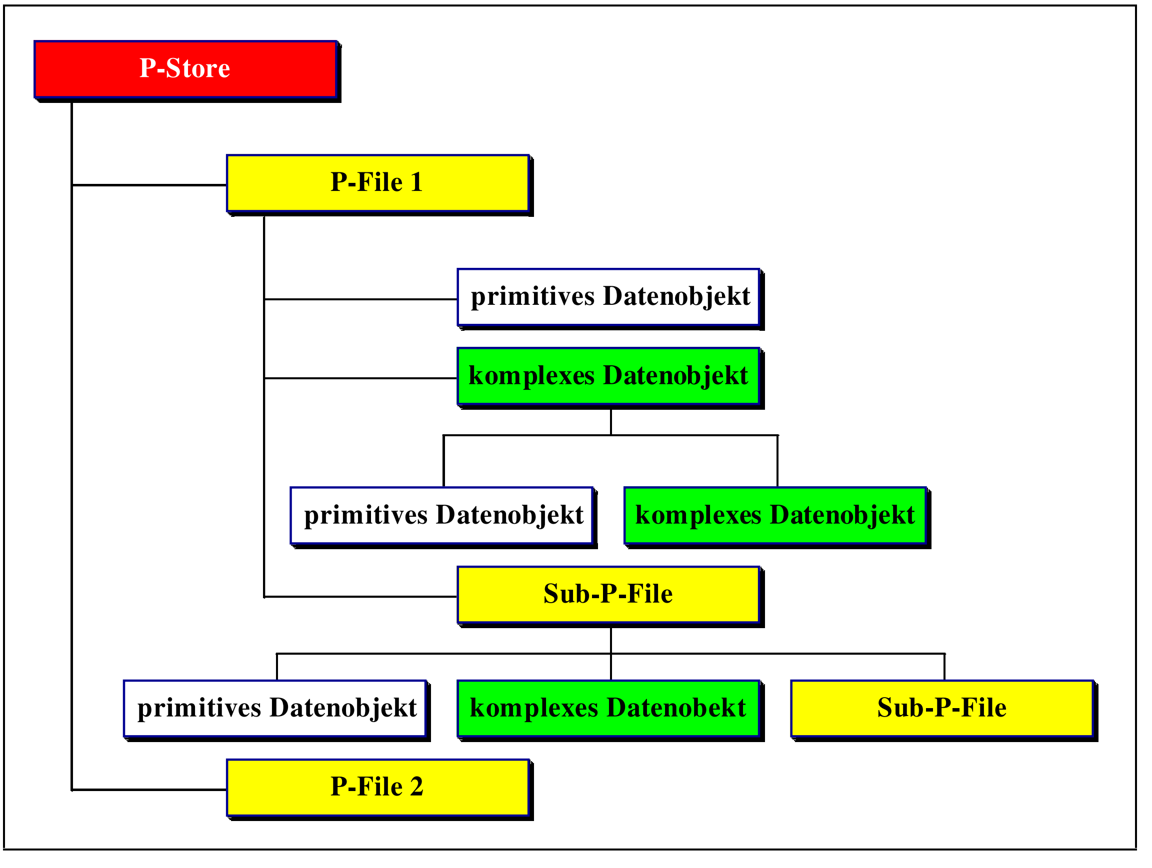

Abbildung 15: P-File Struktur mit Datenobjekten

Die Nutzung der Schnittstelle wird von zwei Seiten erfolgen. Zum einen wird die Schnittstelle von der Sprache PROSET und zum anderen von den zu konstruierenden Werkzeugen genutzt.

Dadurch ergeben sich zwei Nutzungsbereiche, die unterschiedliche Anforderungen an die Funktionalität der Schnittstelle stellen. Diese beiden Bereiche werden im folgenden als interner und externer Bereich bezeichnet. Der interne Bereich steht dabei für die Anforderungen von Seiten der Sprache PROSET und der externe Bereich für die Anforderungen von Seiten der Werkzeuge (siehe Abbildung 16).

Die Schnittstelle soll gekapselte Funktionen für den Zugriff auf H-PCTE-Objekte zur Verfügung stellen. Durch die Zugriffsfunktionen soll es den Applikationen, die im Rahmen der Software-Entwicklungs-Umgebung benutzt werden, ermöglicht werden auf alle in der Datenbank abgelegten Daten zuzugreifen. Die Sprache PROSET zählt hier auch zu den externen Applikationen.

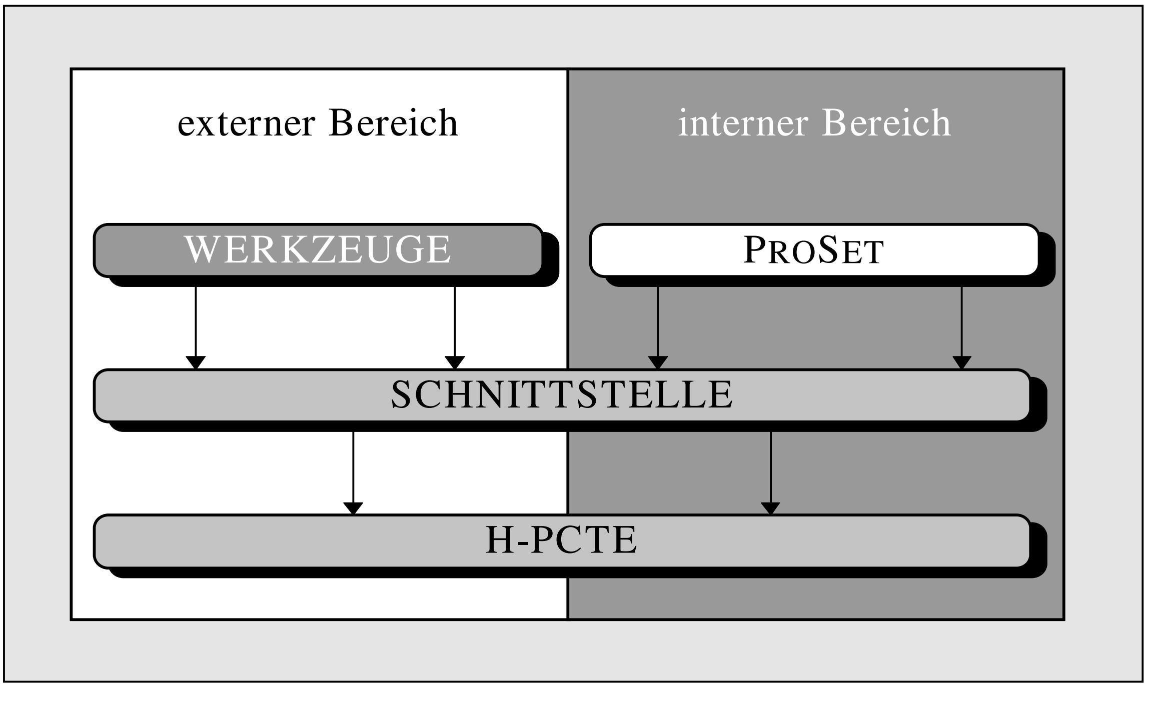

Abbildung 16: Interner und externer Funktionsbereich

Die funktionalen Anforderungen lassen sich gemäß den im vorangegangenen Abschnitt stehenden Erläuterungen nach internen und externen Anforderungen klassifizieren. Die beiden Anforderungsbereiche sollen im folgenden näher betrachtet werden.

4.1.2 Anforderungen des internen Bereichs

Im internen Bereich werden Funktionen benötigt, um PROSET Daten in beliebiger Granularität abzulegen. Da die persistenten PROSET-Daten in P-Files abgelegt werden, ist es notwendig, innerhalb der Datenbank H-PCTE einen Datenbereich einzurichten, in dem P-Files abgelegt und verwaltet werden können. Die Trennung der unterschiedlichen Datenbereiche hat verschiedene Vorteile. Zum einen wird durch das Einrichten verschiedener Datenbereiche eine logische Trennung verschiedener Datengruppen innerhalb der Gesamtdatenbank erreicht und zum anderen kann ein Geschwindigkeitsvorteil bzgl. des Laufzeitverhaltens der Werkzeuge sowie der auf PROSET-Daten zugreifenden Funktionen erwartet werden. Der Performancegewinn ergibt sich unter anderem aus der Tatsache, daß alle Verwaltungsfunktionen nur auf einen Teilbereich der Datenbank zugreifen und sich nicht mit weiteren Strukturen aus anderen Datenbereichen auseinandersetzen müssen. Weitere Anforderungen des internen Bereichs werden an dieser Stelle nicht diskutiert, da der interne Bereich nicht der Schwerpunkt dieser Diplomarbeit ist. Weitere Informationen zu diesem Bereich finden sich bei [Kap95].

4.1.3 Anforderungen des externen Bereichs

Der externe Bereich beschäftigt sich mit den Anforderungen von seiten der Werkzeuge der Prototyping-Umgebung. Hier beachten wir zum einen ein Datenverwaltungswerkzeug und zum anderen ein Werkzeug zur Unterstützung von Datenanalysefunktionen.

Anforderungen des Datenverwaltungswerkzeugs

Die Schnittstelle muß alle notwendigen Funktionen zur Verfügung stellen, die vom Datenverwaltungswerkzeug benötigt werden. Welche Funktionen das Datenverwaltungswerkzeug benötigt, läßt sich daraus ableiten, welche Funktionalität das Datenverwaltungswerkzeug zur Verfügung stellen soll.

Funktionalität des Datenverwaltungswerkzeugs

1. Funktionen zum Anlegen von Strukturen zur Verwaltung von PROSET-Daten (P-Files).
2. Funktionen zum Löschen von P-Files.
3. Funktionen zum Lesen von P-Files und zur Anzeige von P-Files.
4. Funktionen zum Erzeugen von Datenobjekten (PROSET-Daten).
5. Funktionen zum Löschen von Datenobjekten (PROSET-Daten).
6. Funktionen zum Lesen von Datenobjekten (PROSET-Daten).
7. Funktionen zum Schreiben von Datenobjekten (PROSET-Daten).

Hier wurden zunächst nur PROSET-spezifische Dokumente berücksichtigt. Es besteht jedoch auch die Möglichkeit, die Funktionalität des Werkzeugs zu einem späteren Zeitpunkt auf weitere Dokumenttypen auszudehnen.

Anforderungen des Datenanalysewerkzeugs

Die Schnittstelle muß alle notwendigen Funktionen zur Verfügung stellen, die vom Datenverwaltungswerkzeug benötigt werden. Welche Funktionen das Datenverwaltungswerkzeug benötigt läßt sich daraus ableiten, welche Funktionalität das Datenverwaltungswerkzeug zur Verfügung stellen soll.

<u>Funktionalität des Datenanalysewerkzeugs</u>

1. Funktionen zum Auffinden von P-Files und PROSET-Daten. Das Suchkriterium für Strukturen ist der Name des P-Files beziehungsweise des PROSET-Datums.
2. Funktionen zum Auffinden von Datenobjekten. Die Suchkriterien für Datenobjekte können Attribute, wie Name oder Typ sein.
3. Funktionen zur Unterstützung verschiedener Views. Unter Views soll an dieser Stelle die selektive Sicht auf die Datenobjekte verstanden werden. Dies ist notwendig, um Daten eines bestimmten Typs oder Daten für die keine Zugriffsberechtigung besteht, ausblenden zu können.

4.2 Design

Das Design der Schnittstelle basiert auf dem für die abzulegenden Daten entworfenen SDS. Aufbauend auf dem SDS können die verschiedenen Teile der Schnittstelle konstruiert werden, wobei die im vorangegangenen Abschnitt beschriebenen Anforderungen zu berücksichtigen sind.

4.2.1 Das SDS

Das SDS muß alle Dokumenttypen berücksichtigen, die persistent in der Datenbank H-PCTE abgelegt werden sollen. Im Rahmen dieser Diplomarbeit werden an dieser Stelle zunächst alle notwendigen PROSET-Typen berücksichtigt. Im Rahmen der unter PROSET verwandten Datentypen ist eine Klassifizierung in primitive und komplexe Datentypen möglich. Diese beiden Gruppen müssen auf H-PCTE-Objekte abgebildet werden. Die notwendigen Verknüpfungen werden auf H-PCTE Links abgebildet.

P-Files

P-Files werden als reine Verwaltungsstruktur genutzt. Das bedeutet, daß P-Files keine eigenen Wertattribute zugeordnet sind. Aufgrund der Verwaltungsstrukturen, die anhand von P-Files abgebildet werden sollen, müssen P-Files H-PCTE Links auf alle anderen Datentypen haben. In P-Files können von der Sprache PROSET alle verschiedenen, zur Verfügung stehenden Datentypen abgelegt werden. Da diese Datentypen als H-PCTE Datenobjekte instanziiert sind, müssen sie über entsprechende Links referenziert werden. Zusätzlich müssen zur Bildung von Unterstrukturen auch Links auf weitere P-Files abbildbar sein (siehe Abbildung 15).

Primitive Datentypen

Primitive Datentypen sind die feingranularste Struktur die PROSET zur Verfügung stellt. Sie enthalten lediglich Werte, also das zu einer persistenten Variablen gehörige Datum. Primitive Datentypen können keine Unterstrukturen enthalten. Das bedeutet, daß von ihnen keine weiteren H-PCTE Links ausgehen können. Aufgrund der zu realisierenden Baumstruktur und der PROSET-Definition von primitiven Datentypen, dürfen von primitiven Datentypen keine weiteren Links ausgehen. Der in einem primitiven Datentyp abgelegte Wert muß in der Definition des SDS-Objektes durch ein entsprechendes Attribut berücksichtigt werden.

Zusammengesetzte Datentypen

Die zusammengesetzten Datentypen von PROSET sind die Basis der mengentheoretischen Eigenschaften von PROSET. Sie können aus einer Anzahl weiterer verschiedener Datentypen bestehen. Diese weiteren Datentypen können entweder primitive oder zusammengesetzte Datentypen sein. Die zusammengesetzten Datentypen müssen also durch eine Menge von H-PCTE Links auf andere Datenobjekte bestehen. Ein Wertattribut wird für die zusammengesetzten Datentypen nicht benötigt.

Datentypen höherer Ordnung

Datentypen höherer Ordnung können verschiedene Inhalte haben, die nicht mit anderen Datentypen repräsentiert werden können. Sie enthalten keine Unterstrukturen. Da die Art des Wertes unterschiedlich sein kann und nicht immer einem primitiven oder zusammengesetzten Datentypen zugeordnet werden kann, wird der Wert eines Datentyps höherer Ordnung durch eine Zeichenkette dargestellt. Diese Zeichenkette kann mit beliebigen Inhalten gefüllt werden, wodurch eine sehr flexible Nutzung möglich wird. Es können beispielsweise auch Objectcodes in diesem Datentyp abgelegt werden, um sie während der Laufzeit nachzuladen. Ausgehende Links werden nicht benötigt, da ein Datentyp höherer Ordnung keine Unterstrukturen haben kann.

4.2.2 Die Schnittstelle

Die Schnittstelle soll die im Abschnitt 4.1 genannten Anforderungen erfüllen. Da sich die Anforderungen im wesentlichen auf Zugriffsfunktionen für die verschiedenen unter PROSET verfügbaren Datentypen beziehen, ist es sinnvoll die Schnittstelle entsprechend dieser Typen in mehrere Funktionsbereiche zu unterteilen. Diese Funktionsbereiche werden durch entsprechende C-Moduln zur Unterstützung der verschiedenen Datentypen repräsentiert.

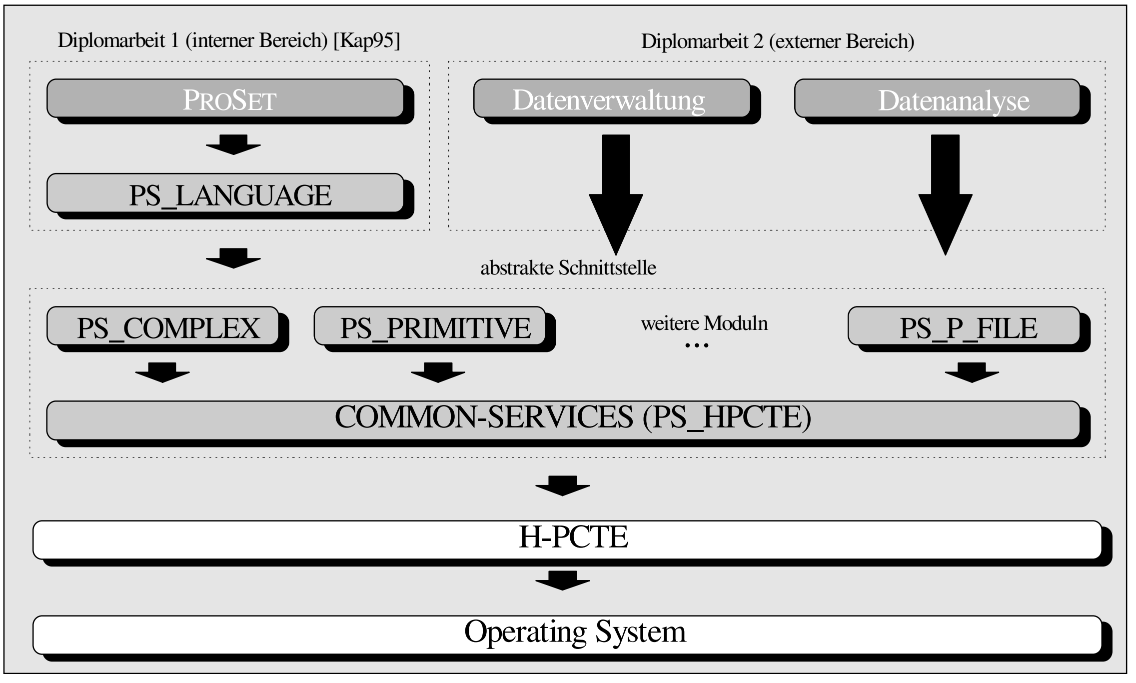

Abbildung 17: Design der Schnittstelle

Abbildung 17 zeigt den Aufbau der Schnittstelle. Der Aufbau der Schnittstelle gliedert sich in mehrere Ebenen. Die erste Ebene (Common-Services) kapselt die von H-PCTE zur Verfügung gestellten Funktionen, so daß ein datentypunabhängiger Zugriff unterstützt wird. In der darüberliegenden Ebene sollen aufgabenspezifische C-Moduln implementiert werden, die dokumentspezifische Funktionen anbieten. Eine weitere Ebene stellt spezielle Anpassungen und anwendungsspezifische Funktionen zur Verfügung. In dieser Ebene sollen unter anderem die Funktionen, die für die Erweiterung der von PROSET angebotenen Konzepte (Transaktionen, Schutzmechanismen, Parallelität) notwendig sind, angeordnet werden. Weitere Informationen zu den erweiterten PROSET-Konzepten finden sich bei [Kap95]. Parallel zu den PROSET-spezifischen Funktionen werden auf dieser Ebene auch die speziell für die zu konstruierenden Werkzeuge notwendigen Funktionen angeordnet.

Für alle weiteren C-Moduln sollen die oben beschriebenen Konventionen beibehalten werden. Für jede neue Funktionsgruppe soll ein eigenes Modul angelegt werden.

4.3 Implementation

In diesem Abschnitt wird die Implementation des SDS, das als Basis für die Schnittstelle dient und die Implementation der Schnittstelle selber beschrieben. Die hier beispielhaft genannten Sourcecode-Dateien in der Sprache C, sind einzelne Moduln aus denen sich die Schnittstelle zusammensetzt. Die Moduln bestehen aus einer Header-Datei (`.h`) und einer Implementations-Datei (`.c`). Die Namen der jeweils behandelten Moduln werden dabei wie folgt abgekürzt:

`modul.h/c` entspricht `modul.h` und `modul.c`.

Diese Konvention wird auch in allen folgenden Abschnitten beibehalten, die sich mit speziellen Moduln und der Implementation befassen.

4.3.1 Das SDS

Das SDS, in dem die für PROSET notwendigen Datenobjekte definiert sind, trägt den Namen `proset.sds`. In diesem SDS sind die Datentypen von PROSET abgebildet. Die Beziehungen der Datentypen zueinander werden durch Links repräsentiert. Die folgende Aufzählung nennt die wichtigsten Objekttypen und ihre Funktion:

- `ps_integer`
  Objekttyp zur Verwaltung von Integerwerten
- `ps_string`
  Objekttyp zur Verwaltung von Zeichenketten
- `ps_boolean`
  Objekttyp zur Verwaltung von Wahrheitswerten
- `ps_real`
  Objekttyp zur Verwaltung von Fließkommawerten. Dieser Typ wird auf eine Zeichenkette abgebildet, da H-PCTE keinen passenden Typ anbietet.
- `ps_set`
  Objekttyp zur Verwaltung einer Menge. Eine Instanz dieses Typs besteht aus einer Anzahl von Links auf Instanzen anderer Objekttypen.

- `ps_tuple`
  Objekttyp zur Verwaltung eines Tupels. Eine Instanz dieses Typs besteht aus einer Anzahl von Links auf Instanzen anderer Objekttypen.

Im Rahmen des SDS werden noch weitere Objekttypen definiert, die an dieser Stelle aber nicht näher erläutert werden. Die Abbildung 18 zeigt einen Ausschnitt aus der Datei `proset.sds`.

```
-- String
objecttype ps_string : child type of ps_object
with
  attribute
    string_value;
end ps_string;

-- Objekttyp fuer Set
objecttype ps_set : child type of ps_object;

linktype set_element : composition link (number)
      to    ps_set,
            ps_tuple,
            ps_boolean,
            ps_integer,
            ps_string,
            ps_real,
            ps_atom,
            ps_function,
            ps_modtype,
            ps_instance;

extend objecttype ps_set
with
  component
     set_element;
end ps_set;
```

Abbildung 18: Ausschnitt aus der Datei `proset.sds`

4.3.2 Die Schnittstelle

Die Schnittstelle besteht insgesamt aus 21 C-Moduln, die jeweils die Funktionalität für einen speziellen Bereich zur Verfügung stellen. Da der Implementation der Moduln im Rahmen dieser Diplomarbeit keine wesentliche Bedeutung zukommt, werden sie im folgenden nur kurz vorgestellt. Der Funktionalitätsbereich wird beschrieben und die Funktionen werden genannt. Eine detaillierte Beschreibung der Funktionen sowie der Sourcecode der Funktionen sind bei [Kap95] nachzulesen.

Das Modul `ps_auditing.h/c`

Diese Modul stellt verschiedene Funktionen zur Realisierung eines Auditing-Konzeptes zur Verfügung. Dies ermöglicht eine Protokollierung von Zugriffen der Benutzer auf ein Datum.

Liste der Funktionsnamen:

1. `adg_auditing_is_on`
2. `adg_auditing_turn_on`
3. `adg_auditing_turn_off`
4. `adg_time_attribute_set`

Das Modul `ps_command.h/c`

Das Modul ps_command.h/c stellt gekapselte Funktionen zur Verfügung, die einen einfachen Zugriff auf den P-Store ermöglichen.

Liste der Funktionsnamen:

1. `cmd_p_file_create`
2. `cmd_p_file_delete`
3. `cmd_proset_value_write`
4. `cmd_proset_value_read`
5. `cmd_proset_value_delete`
6. `cmd_proset_value_rights_set`
7. `cmd_proset_value_grp_rights_set`

Das Modul `ps_complex.h/c`

Das Modul `ps_complex.h/c` beinhaltet Funktionen für den Zugriff auf zusammengesetzte PROSET-Datentypen.

Liste der Funktionsnamen:

1. `cp_compound_object_read`
2. `cp_compound_object_write`
3. `cp_function_read`
4. `cp_function_write`
5. `cp_instance_read`
6. `cp_instance_write`
7. `cp_modtype_read`
8. `cp_modtype_write`
9. `cp_unit_read`

10. `cp_unit_write`
11. `cp_all_objects_read`
12. `cp_all_objects_write`

Das Modul `ps_debug.h/c`
Das Modul `ps_debug.h/c` beinhaltet einige Hilfsfunktionen. Diese Hilfsfunktionen erzeugen diverse Ausgaben, die bei der Fehlersuche in PROSET-Programmen hilfreich sein können.

<u>Liste der Funktionsnamen:</u>
1. `dbg_frozen_env_print`
2. `dbg_print_info`
3. `dbg_print_object`
4. `dbg_list_print`

Das Modul `ps_distribution.h/c`
Das Modul `ps_distribution.h/c` stellt Funktionen zur Realisierung der verteilten Speicherung von persistenten ProSet-Werten zur Verfügung. Dabei wird die von H-PCTE unterstützte Segmentbildung genutzt [Kel91], [Kel92], [Kel93].

<u>Liste der Funktionsnamen:</u>
1. `dtb_segment_create`
2. `dtb_segment_load`
3. `dtb_segment_handle_get`

Das Modul `ps_dynlink.h/c`
Das Modul `ps_dynlink.h/c` stellt Funktionen zur Verfügung, die es erlauben in PROSET-Programmen einen dynamischen Linker (z.B. DLD [Kap95]) zu nutzen. Dadurch können zur Laufzeit eines PROSET-Programms Datentypen höherer Ordnung, die zum Beispiel Objectcode enthalten, zum laufenden Programm dazugebunden werden.

<u>Liste der Funktionsnamen:</u>
1. `dl_reset_unit_loaded_twice`
2. `dl_is_unit_loaded_twice`
3. `dl_library_link`
4. `dl_object_code_read`

Das Modul `ps_error.h/c`

Das Modul `ps_error.h/c` definiert Fehlercodes für die Persistenz-Schnittstelle zu PROSET und Makros zur geeigneten Fehlerbehandlung.

Liste der Makronamen:

1. `RETURN_IF_ERROR`
2. `PRINT_PS_ERROR`

Eine Liste aller definierten Fehlercodes ist im Abschnitt zum Sourcecode bei [Kap95] nachzuschlagen.

Das Modul `ps_hpcte.h/c`

Das Modul `ps_hpcte.h/c` stellt im wesentlichen allgemeine Funktionen für den Datenzugriff auf PROSET-Werte zur Verfügung. In diesem Modul sind die oben beschriebenen Common-Services, die Zugriffsfunktionen von H-PCTE, gekapselt.

Liste der Funktionsnamen:

1. `h_is_access_right_denied`
2. `h_access_denied_recognize`
3. `h_basis_object_create`
4. `h_default_acl_get`
5. `h_sub_p_file_create`
6. `h_p_file_create`
7. `h_p_store_handle_get`
8. `h_second_component_create`
9. `h_second_component_handle_get`
10. `h_intern_value_create`
11. `h_proset_object_create`
12. `h_deadlock_recognize`
13. `h_deadlock_check`
14. `h_longfield_read`
15. `h_longfield_write`
16. `h_server_init`
17. `h_server_done`
18. `h_object_type_is_equal`
19. `h_object_handle_get_abs`
20. `h_object_handle_get_rel`
21. `h_string_attribute_set`
22. `h_string_attribute_get`
23. `h_object_type_check`
24. `h_set_integer_attribute`
25. `h_get_integer_attribute`
26. `h_set_boolean_attribute`
27. `h_get_boolean_attribute`
28. `h_object_type_get`
29. `h_object_delete`
30. `h_working_schema_set`
31. `h_object_list_links`
32. `h_object_handle_free`
33. `h_error_message_print`
34. `h_object_remove`

Das Modul `ps_install.h/c`

Das Modul `ps_install.h/c` beinhaltet Funktionen zur Installation der PROSET-Verwaltungsobjekte. Zu den Verwaltungsobjekten gehören der P-Store, temporäre Verzeichnisse und initiale Datenobjekte.

Liste der Funktionsnamen:
1. `ins_installation_init`
2. `ins_persistent_root_create`
3. `ins_p_store_create`
4. `ins_tmp_directory_create`
5. `ins_initial_objects_create`

Das Modul `ps_konstant.h/c`

Im Modul `ps_konstant.h/c` werden alle für die Schnittstelle notwendigen Konstanten definiert. Den Konstanten sind bestimmte Wertebereiche zugeordnet, die zu verschiedenen Themengebieten gehören. Weitere Konstanten definieren die Namen von Objekttypen und Attributen. Im folgenden werden die Wertebereiche und ihre Bedeutung erläutert:

1. H-PCTE Fehlerkonstanten (entsprechend dem Modul `ps_error.h/c`)
2. PROSET Fehlerkonstanten (entsprechend dem Modul `ps_error.h/c`)
3. Fehlercodes für DLD (Dynamischer Linker)
4. Fehlercodes für Transaktionen
5. Typkonstanten
6. SDS Objekttypen (Namensdefinition)
7. SDS Attributnamen (Namensdefinition)

Das Modul `ps_lang.h/c`

Das Modul `ps_lang.h/c` beinhaltet alle Funktionen, die von der Sprache PROSET und von den Werkzeugen benötigt werden, um mit der Datenbank H-PCTE zu arbeiten.

Liste der Funktionsnamen:
1. `lg_application_init`
2. `lg_application_finish`
3. `lg_application_abort`
4. `lg_p_file_path_scan`
5. `lg_p_files_scan`

6. `lg_proset_value_read`
7. `lg_proset_value_write`
8. `lg_object_handle_get`
9. `lg_object_handle_free`
10. `lg_proset_value_delete`
11. `lg_creation_mode_get`
12. `lg_proof_deletion`
13. `lg_object_type_read`
14. `lg_is_first_class_proset_value`

Das Modul `ps_p_file.h/c`
Das Modul `ps_p_file.h/c` beinhaltet Funktionen zum Erzeugen und zum Löschen vom P-Files innerhalb des P-Stores.

Liste der Funktionsnamen:
1. `pf_is_p_file_name_correct`
2. `pf_proset_p_file_create`
3. `pf_p_file_create`
4. `pf_p_file_delete`

Das Modul `ps_primitive.h/c`
Das Modul `ps_primitive.h/c` beinhaltet Funktionen für den Zugriff auf primitive PROSET-Datentypen.

Liste der Funktionsnamen:
1. `pr_atom_write`
2. `pr_integer_write`
3. `pr_boolean_write`
4. `pr_real_write`
5. `pr_string_write`
6. `pr_omega_write`
7. `pr_atom_read`
8. `pr_integer_read`
9. `pr_boolean_read`
10. `pr_real_read`
11. `pr_string_read`

Das Modul `ps_security.h/c`

Das Modul `ps_security.h/c` stellt Funktionen zur Unterstützung eines Zugriffsschutzes von PROSET-Werten zur Verfügung [Kap95].

Liste der Funktionsnamen:

1. `sec_application_init`
2. `sec_application_finish`
3. `sec_active_user_handle_get`
4. `sec_active_group_handle_get`
5. `sec_access_right_check`
6. `sec_group_access_rights_set`
7. `sec_tool_subject_access_rights_set`
8. `sec_access_rights_get`
9. `sec_access_rights_set`
10. `sec_user_create`
11. `sec_group_create`
12. `sec_group_handle_get`
13. `sec_administrator_home_set`
14. `sec_user_add_group`
15. `sec_subgroup_add_group`
16. `sec_tool_user_create`
17. `sec_tool_group_create`
18. `sec_user_access_rights_set`

Das Modul `ps_storage.h/c`

Das Modul `ps_storage.h/c` stellt eine Schnittstelle zur Speicherverwaltung zur Verfügung. Da während der Implementierung Probleme mit den Standard-Routinen von ANSI-C auftraten, wurden die benötigten Routinen neu implementiert. Die hier implementierten Routinen bieten gegenüber den von ANSI-C zur Verfügung gestellten ein besseres Handling.

Liste der Funktionsnamen:

1. `st_check_alloc`
2. `st_malloc`
3. `st_calloc`
4. `st_free`
5. `st_cfree`

Das Modul `ps_transact.h/c`

Das Modul `ps_transact.h/c` stellt Funktionen zur Unterstützung eines Transaktionsmanagements zur Verfügung [Kap95].

Liste der Funktionsnamen:

1. `ta_is_transaction_active`
2. `ta_transaction_start`
3. `ta_transaction_commit`

4. `ta_transaction_undo`
5. `ta_transaction_abort`

Das Modul `ps_types.h`
Das Modul `ps_types.h` beinhaltet Typdefinitionen, die im Rahmen der Schnittstelle global gültig sind.

Das Modul `ps_util.h/c`
Das Modul `ps_util.h/c` beinhaltet einige Hilfsfunktionen und -makros. Diese Hilfsfunktionen und -makros stellen keine dokumentspezifische Funktionalität zur Verfügung, sondern gehören zur Gruppe der allgemeinen Funktionen.

Liste der Funktionsnamen:

1. `PRINT_MSG (Makro)`
2. `PRINT_MSG_MORE (Makro)`
3. `u_init_string_copy`
4. `u_string_copy`
5. `u_mem_append`
6. `u_p_file_entry_path_get`
7. `u_p_file_node_path_get`
8. `u_make_link`
9. `u_space`
10. `u_string_dup`
11. `u_dispose_mem`
12. `u_proset_path_2_hpcte_path`
13. `u_split_p_file_path`
14. `ps_att_name`
15. `u_get_proset_type`
16. `u_get_p_file_entry_link`
17. `u_string_to_real`
18. `u_get_link`

Weitere Informationen zu der von den einzelnen Moduln zur Verfügung gestellten Funktionalität finden sich bei [Kap95].

5 Das Meta-Tool

Dieses Kapitel stellt die Konstruktion des Meta-Tools vor. Wie die in Kapitel 3 durchgeführten Betrachtungen ergeben haben, ist ein solches Meta-Tool notwendig. Die Notwendigkeit entsteht aus der Forderung nach konsistenten Werkzeugen innerhalb der Prototyping-Umgebung (siehe Kapitel 3). Zunächst werden die Anforderungen spezifiziert, wobei die allgemeinen Anforderungen, die bereits in den vorangegangenen Kapiteln besprochen wurden, berücksichtigt werden. Im zweiten Abschnitt wird auf der Basis der Anforderungen ein Konzept entworfen. Abschließend wird die Implementation beschrieben, wobei Auszüge aus der Codegenerierung vorgestellt werden.

5.1 Anforderungen

Das Meta-Tool dient zur Erzeugung konsistenter Werkzeuge, die im Rahmen der Prototyping-Umgebung eingesetzt werden können. Dabei entstehen zwei verschiedene Arten von Anforderungen, zum einen die Anforderungen der Prototyping-Umgebung und zum anderen die Anforderungen der Benutzer der Prototyping-Umgebung. Die Anforderungen der Prototyping-Umgebung beziehen sich im wesentlichen auf Integrationsaspekte, während die Anforderungen der Benutzer eher rollenspezifisch sind. Die Ergebnisse der Betrachtungen aus den Abschnitten 1.2 und 3 sollen hier berücksichtigt werden.

5.1.1 Anforderungen der Prototyping-Umgebung an das Meta-Tool

Die Prototyping-Umgebung stellt Anforderungen bzgl. der Integration von Werkzeugen, die in der Umgebung benutzt werden sollen. Da das Meta-Tool Werkzeuge generieren soll, die in die Umgebung eingefügt werden können, müssen schon beim Entwurf des Meta-Tools entsprechende Anforderungen berücksichtigt werden.

Im Rahmen der Datenintegration entsteht die Anforderung, daß ein erzeugtes Werkzeug dokumentunabhängig sein sollte, d.h. es sollte möglich sein, alle in der Prototyping-Umgebung vorkommenden Dokumenttypen mit dem Werkzeug zu bearbeiten. Da für die verschiedenen Dokumenttypen eventuell auch verschiedene Zugriffsmethoden benötigt werden, weitere

Dokumenttypen eingeführt werden und Änderungen bei der Funktionalität notwendig werden können, muß es möglich sein, ein erzeugtes Werkzeug zu einem späteren Zeitpunkt zu modifizieren. Auf diesem Weg kann das erzeugte Werkzeug auch an neue Dokumenttypen angepaßt werden, beziehungsweise seine Funktionalität in Bezug auf bereits vorhandene Dokumenttypen erweitert werden. Zusätzlich zu der dokumentspezifischen Funktionalität die ein Werkzeug anbieten muß, sollte es auch dokumentunabhängige Funktionen unterstützen.

Dokumentunabhängige Funktionen sind Funktionen, deren Ausführung sich nicht nur auf Dokumente eines speziellen Typs bezieht. Darunter fallen zum Beispiel Funktionen, die alle in einem P-File abgelegten Daten auslesen und auflisten, unabhängig davon von welchem Typ diese Daten sind. Dokumentabhängige Funktionen hingegen lassen nur die Bearbeitung eines speziellen Dokumenttyps zu, beispielsweise die Erzeugung eines Objektes vom Typ Integer.

Ein weiterer Aspekt, der im Rahmen der Prototyping-Umgebung wichtig ist, ist ein einheitliches Erscheinungsbild der erzeugten Werkzeuge. Dieser Aspekt fällt in den Bereich der Präsentationsintegration. Dabei sollte auf die Unterstützung verschiedener Benutzungsschnittstellen geachtet werden. Die Werkzeuge die mit dem Meta-Tool erzeugt werden, sollen drei Benutzungsschnittstellen zur Verfügung stellen. Diese werden im Abschnitt 5.2.3 genauer erläutert.

5.1.2 Anforderungen des Werkzeugentwicklers an das Meta-Tool

Für die Werkzeugentwicklung sollte eine einheitliche Basis vorhanden sein. Das bedeutet, daß alle Werkzeuge, die für eine Prototyping-Umgebung konstruiert werden, den gleichen Entwurfs- und Funktionsprinzipien folgen. Diese Anforderung wird durch die Codegenerierung mit dem Meta-Tool unterstützt, da der erzeugte Code bei allen Werkzeugen identisch ist und somit eine Basis für die konsistente Werkzeugentwicklung darstellt. Gleichzeitig muß es möglich sein, die Basis auf der der Code erzeugt wird zu modifizieren, um Anpassungen vornehmen zu können. Das bedeutet, daß die Definition des zu erzeugenden Codes vom Code-Generator getrennt werden muß. Die eigentliche Codeerzeugung basiert also auf zwei Komponenten, zum einen auf dem Codeerzeuger (Meta-Tool) und zum anderen auf der Definition des zu erzeugenden Codes. Diese Definition wird in Form von Vorlagen (*Templates*) formuliert.

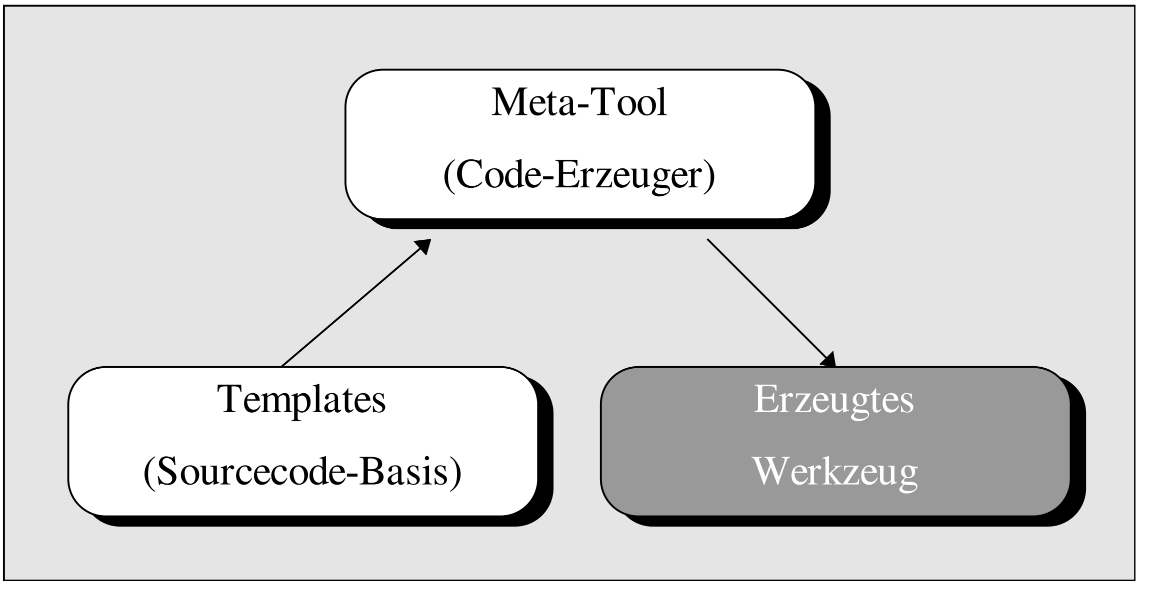

Abbildung 19: Sourcecodegenerierung auf Basis von Templates

Die Templates beinhalten Mustercode für die zu erzeugenden Werkzeuge und können modifiziert werden, ohne das der Code-Generator neu übersetzt werden muß. Eine weitere Anforderung an den zu erzeugenden Code ist die Erweiterbarkeit der erzeugten Werkzeuge zu einem späteren Zeitpunkt. Dieser Aspekt muß durch ein entsprechendes Werkzeug unterstützt werden, das es erlaubt, ein zu einem früheren Zeitpunkt erzeugtes Werkzeug zu modifizieren und seine Funktionalität zu erweitern. Diese Anforderung ist deshalb so wichtig, weil der Werkzeugentwickler zum Zeitpunkt seines ersten Werkzeugentwurfs nicht wissen kann, welche Funktionalität zu einem späteren Zeitpunkt benötigt wird. So könnten zum Beispiel neue Dokumente eingeführt werden, für die entsprechende Zugriffsfunktionen implementiert werden müßten. Bestünde keine Möglichkeit vorhandene Werkzeuge anzupassen oder zu erweitern, müßte für jeden neuen Dokumenttyp ein neues Werkzeug entworfen werden. Im Zusammenhang mit dem Aspekt der Dokumentunabhängigkeit tritt eine weitere Anforderung auf. Da zum Zeitpunkt der Konstruktion nicht klar ist, welche Dokumente später berücksichtigt werden müssen, ist ebensowenig bekannt, welche Rückgabewerte die Funktionen liefern müssen, die durch ein Werkzeug angeboten werden sollen. Aus diesem Grund müssen auch die Rückgabewerte individuell anpaßbar beziehungsweise erweiterbar sein, um eventuelle Konflikte zu vermeiden.

Um eine möglichst große Flexibilität bzgl. des generierten Sourcecodes zu gewährleisten, sollte der Code-Generator auch die Verarbeitung von Makros unterstützen. Makros sind spezielle

Textteile innerhalb eines Templates, die vom Code-Generator erkannt und durch andere Textteile ersetzt werden müssen. Diese Ersetzung von Textteilen kann in einfachen Fällen das Ersetzen von Platzhaltern durch Namen beinhalten, in komplexeren Fällen aber auch das Erzeugen komplexer neuer Strukturen, abhängig vom aktuellen Werkzeugstatus bedeuten. Der aktuelle Werkzeugstatus wird dabei durch die Menge aller dem Werkzeug bekannten Dokument- und Funktionstypen definiert. Da die Aktionen, die nach der Identifikation eines Makros auszuführen sind sehr individuell sein können, sollten die Funktionen in ein eigenes Modul ausgelagert werden. So hat der Benutzer des Code-Generators die Möglichkeit, neue Makros zu definieren und die Aktionen, die im Falle des Vorkommens des Makros auszuführen sind, zu implementieren.

Eine abschließende Anforderung ist, daß der vom Code-Generator erzeugte Code sofort übersetzbar und ausführbar sein sollte. Das bedeutet, daß nachdem mit dem Meta-Tool ein Werkzeugrumpf erzeugt worden ist, der C-Compiler diesen fehlerlos in eine ausführbare Datei übersetzen können muß.

5.2 Aufbau und Erzeugung eines generierten Werkzeugs

Der Aufbau eines zu generierenden Werkzeugs für die Prototyping-Umgebung orientiert sich an den im vorangegangenen Abschnitt beschriebenen Anforderungen. Die Erzeugung splittet sich in zwei wesentliche Teile auf. Der erste Teil betrifft die Erzeugung einer für das Werkzeug geeigneten Verzeichnisstruktur und der zweite Teil betrifft die Erzeugung der Sourcecode-Dateien. Diese verschiedenen Strukturen und Dateien werden im folgenden genauer betrachtet.

5.2.1 Generierte Verzeichnisstruktur für den Sourcecode eines Werkzeugs

Um die Anordnung der zu erzeugenden Dateien übersichtlich zu gestalten, wird als Basis für die eigentliche Codeerzeugung eine UNIX-Verzeichnisstruktur generiert, in der beim weiteren Vorgehen die verschiedenen Teile des Werkzeugs abgelegt werden.

Zunächst wird für ein neues Werkzeug ein neues Basisverzeichnis benötigt, das den Namen des Werkzeugs trägt und in dem alle weiteren Dokumente, die das Werkzeug betreffen, abgelegt werden. Im Basisverzeichnis selber werden die ausführbaren Dateien des Werkzeugs sowie die Übersetzungsvorschriften (in Form eines MAKEFILEs) abgelegt (siehe auch [KR83]). Da neben diesen Dokumenten weitere Dokumente unterschiedlicher Typen vorkommen, werden innerhalb des Basisverzeichnisses weitere Unterverzeichnisse für die verschiedenen

Dokumenttypen benötigt. Bei der Erzeugung werden Verzeichnisse für die Dokumente **Sourcecode**, **Dokumentation** und **Objectcodes** angelegt. Abbildung 20 zeigt beispielhaft die Verzeichnisstruktur für ein Werkzeug mit dem Namen `TESTTOOL`.

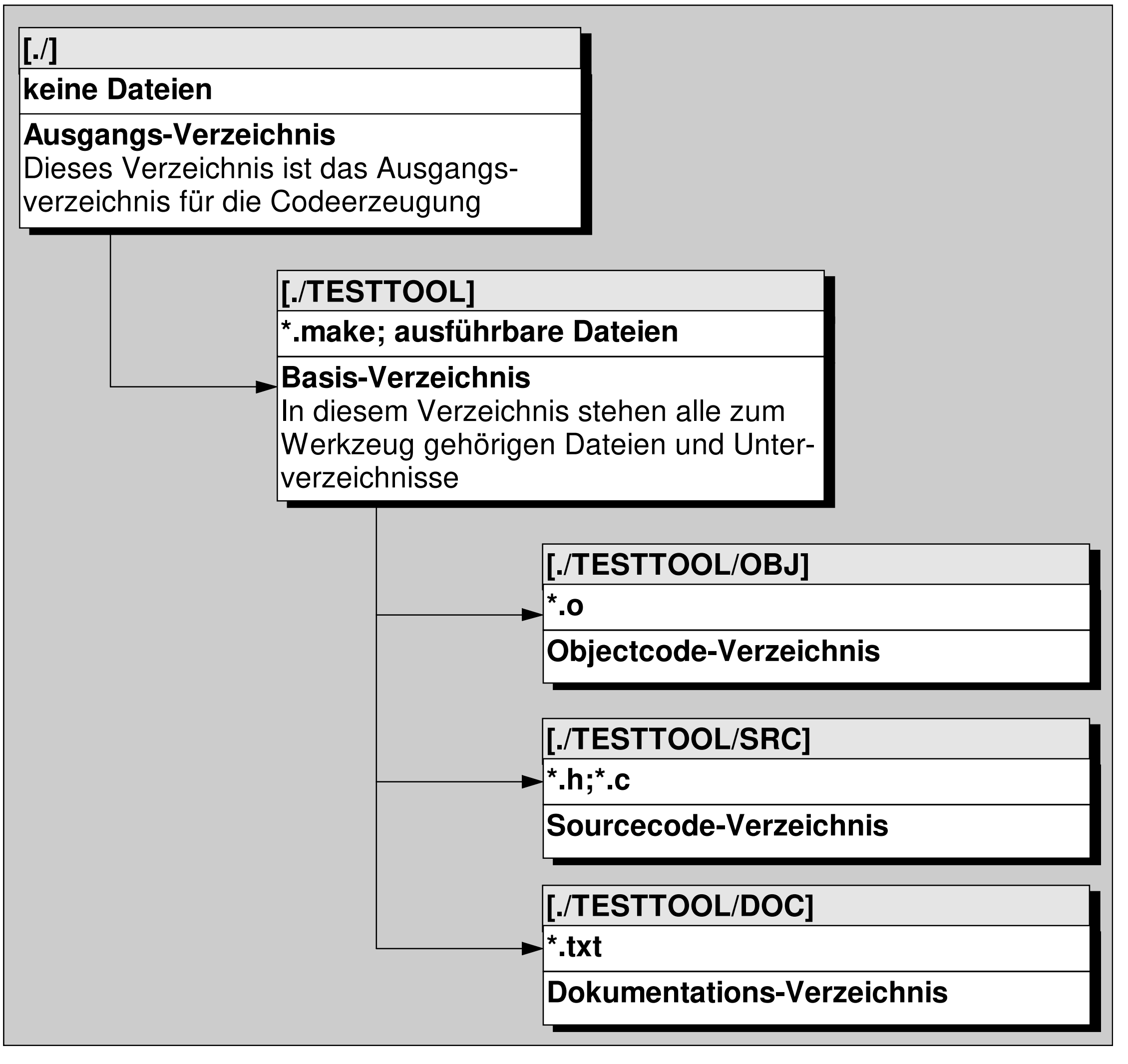

Abbildung 20: Verzeichnisstruktur eines Werkzeugs

5.2.2 Generierte Dateien für ein Werkzeug

Die Dateien die für ein Werkzeug generiert werden müssen, sind zum einen Übersetzungsvorschriften und zum anderen die Sourcecode-Dateien. Die Objectcode-Dateien werden durch den C-Compiler erzeugt. Dokumentations-Dateien werden im Rahmen der Codegenerierung nicht generiert. In einer Ausbaustufe können die Funktionsheader mit einer kurzen Erläuterung als Dokumentation im Verzeichnis `DOC` abgelegt werden. Bei der Erzeugung der Sourcecodes müssen die in den vorangegangenen Abschnitten beschriebenen Anforderungen berücksichtigt werden.

Makefile
Das Makefile beinhaltet die Übersetzungsvorschriften für die Übersetzung des Sourcecodes in ein ablauffähiges Werkzeug. Diese Datei muß bei späteren Modifikationen des Werkzeugs angepaßt werden, damit neu hinzugekommene Moduln, die eine zusätzliche Funktionalität anbieten, korrekt eingebunden werden.

Sourcecode-Dateien
Im Rahmen der Sourcecode-Dateien müssen eine Reihe von Dateien erzeugt werden. Die Dateien werden im Verzeichnis `./<TOOLNAME>/SRC` abgelegt.

Im folgenden werden aufeinander aufbauende Ansätze beschrieben, die sich mit dem zu erzeugenden Sourcecode beschäftigen. Dabei wird im einzelnen diskutiert, welche Dateien zu welchem Zweck erzeugt werden müssen. Es werden die in den vorangegangenen Abschnitten beschriebenen Anforderungen berücksichtigt und schrittweise eingearbeitet.

<u>Ansatz 1</u>
Es muß eine Hauptdatei existieren, die den Sourcecode des Werkzeugs beinhaltet. Diese trägt den Namen des Werkzeugs und den Extender „`.c`“.

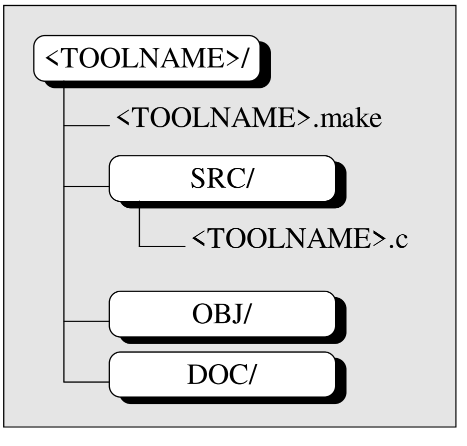

Abbildung 21: Codegenerierung nach Ansatz 1

Ansatz 2

Der Werkzeugentwickler muß nach der Generierung des Sourcecodes die Funktionen mit Funktionalität füllen. Das bedeutet, daß er Veränderungen in der Datei `<TOOLNAME>.c` vornehmen muß. Da im Ansatz 1 nur diese eine Sourcecode-Datei erzeugt wurde, ist in dieser Datei auch die Basisfunktionalität (Menüführung, Kommandozeilenauswertung) abgelegt. Dies kann zu Problemen führen, da der Entwickler auch die Basisfunktionalität verändern kann. Aus Gründen des einheitlichen Designs und damit der Integrationsfähigkeit des Werkzeugs (siehe Abschnitt 1.2.1) sollte dies aber vermieden werden, um die Konsistenz aller generierten Werkzeuge zu gewährleisten. Die Lösung für dieses Problem ist die Trennung der Basisfunktionalität (Basissourcecode) von der vom Entwickler zu implementierenden Funktionalität (Entwicklersourcecode). Die Trennung kann durch eine Splittung der Datei `<TOOLNAME>.c` erreicht werden. Es werden also getrennte Dateien für die Basisfunktionalität und die vom Entwickler zu implementierende Funktionalität erzeugt. Die neue Datei `<TOOLNAME>.c` enthält nur die Basisfunktionalität und eine geeignete Struktur zum Aufruf der ausgelagerten Funktionalität. Die vom Entwickler zu implementierende Funktionalität wird in dokumentspezifische Moduln ausgelagert. Das bedeutet, daß für jeden Dokumenttyp ein eigenes Modul und eine eigene Header-Datei angelegt wird, die die Funktionalität für diesen Dokumenttypen beinhalten. Die Dateien tragen die Namen `<DOKUMENTTYP>.h` und `<DOKUMENTTYP>.c`.

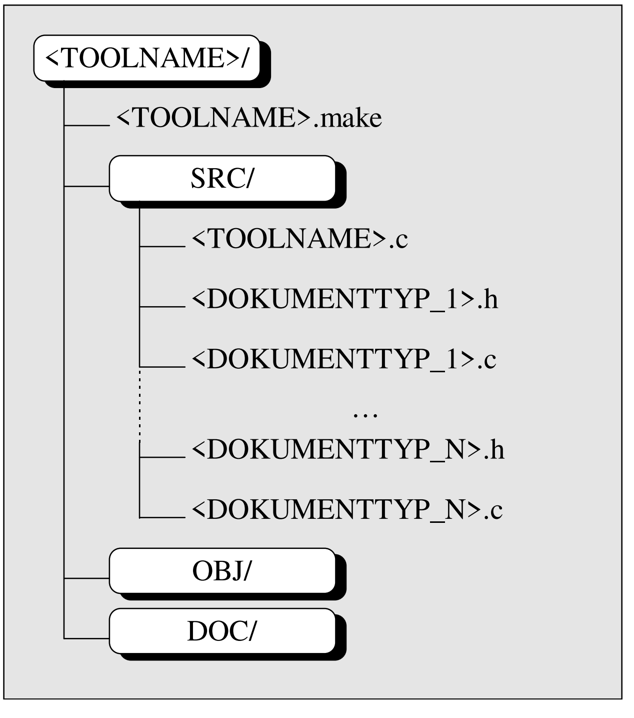

Abbildung 22: Codegenerierung nach Ansatz 2

Ansatz 3

Eine weitere zu berücksichtigende Anforderung ist die Definition von Funktions-Rückgabetypen. Die Rückgabetypen müssen zwei Anforderungen erfüllen. Die erste Anforderung ist, daß die Rückgabetypen für alle erzeugten Funktionen gleich sein müssen, weil der Code-Generator nicht zwischen den Funktionen differenzieren kann. Die zweite Anforderung ist, daß die Möglichkeit bestehen muß, beliebige Rückgabewerte mit Hilfe des Rückgabetypen zurückzugegeben, weil für verschiedene Dokumenttypen verschiedene Ergebnistypen entstehen. Die Lösung dieses Konfliktes, besteht in der Definition eines abstrakten Rückgabetyps, der vom Entwickler nachträglich angepaßt und trotzdem von allen Funktionen weiterbenutzt werden kann. Um eine Anpassung ohne Gefährdung der Konsistenz des Basissourcecodes zu gewährleisten, muß die Definition des Rückgabewertes in eine getrennte Datei ausgelagert werden, die bei der Übersetzung von den anderen Moduln eingebunden wird. Die Datei trägt den Namen `retval.h`. Zusätzlich wird das Modul `standard.h/c` eingeführt. In diesem Modul können allgemeine Routinen, die für das Werkzeug zur Verfügung gestellt werden sollen, eingefügt werden.

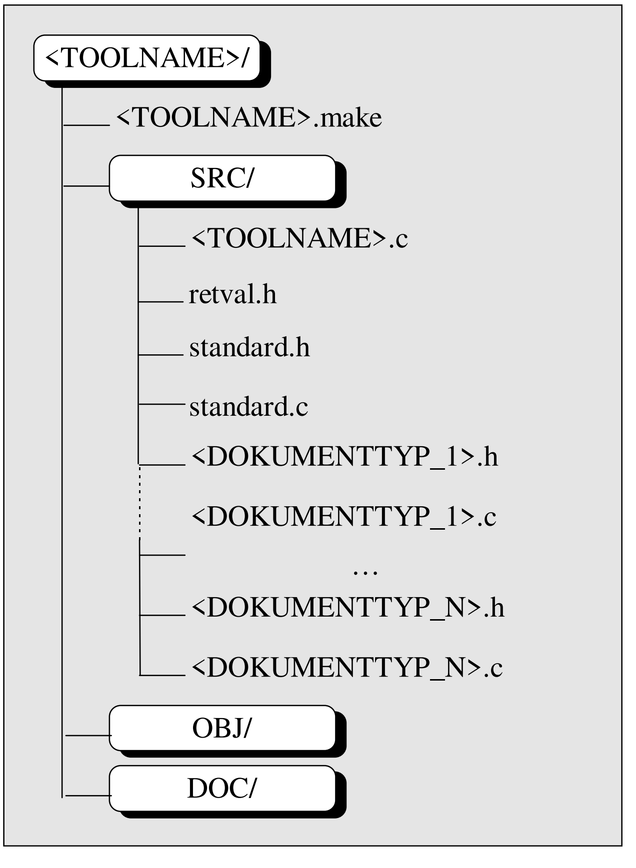

Abbildung 23: Codegenerierung nach Ansatz 3

Ansatz 4

Die letzte zu beachtende Anforderung, ist die Forderung nach der Nutzbarkeit der Funktionalität, durch andere Anwendungen. Um die Möglichkeit der externen Nutzung zu gewährleisten, muß die Funktionalität des Werkzeugs muß in Form eines C-Moduls zur Verfügung stehen, das in andere Anwendungen eingebunden werden kann. Dadurch entsteht eine Schnittstelle, die die gesamte Funktionalität des Werkzeugs zur Verfügung stellt. Dieses Problem ist nur zu lösen, indem eine Hauptfunktion zur Verfügung gestellt wird, die in ein Modul ausgelagert ist. Die Hauptfunktion ermöglicht den Zugriff auf die Funktionalität des Werkzeugs und durch die Auslagerung ist es möglich, das Modul beziehungsweise die Hauptfunktion in andere Anwendungen einzubinden. Diese Forderung kann durch eine Splittung der Datei <TOOLNAME>.c erreicht werden. Die Datei wird in die zwei neuen Dateien <TOOLNAME>.c und <TOOLNAME>mf.h sowie <TOOLNAME>mf.c aufgesplittet.

Das Kürzel mf steht dabei für **M**ain-**F**unction. Die Datei <TOOLNAME>mf.c ist ein Modul und enthält die gesamte Basisfunktionalität, die im vorherigen Ansatz in der Datei <TOOLNAME>.c angesiedelt waren. Diese Funktionalität ist in einer einzigen Funktion gebunden, die den gleichen Aufrufkonventionen wie eine Kommandozeilen-Anwendung folgt. Dadurch wird gewährleistet, daß die gewünschte Funktionalität in andere Anwendungen durch Einbinden dieses Moduls integriert werden kann. Die Datei <TOOLNAME>mf.h ist die zugehörige Header-Datei. Die neue Datei <TOOLNAME>.c enthält lediglich einen einzigen Aufruf der in <TOOLNAME>mf.c befindlichen Hauptfunktion.

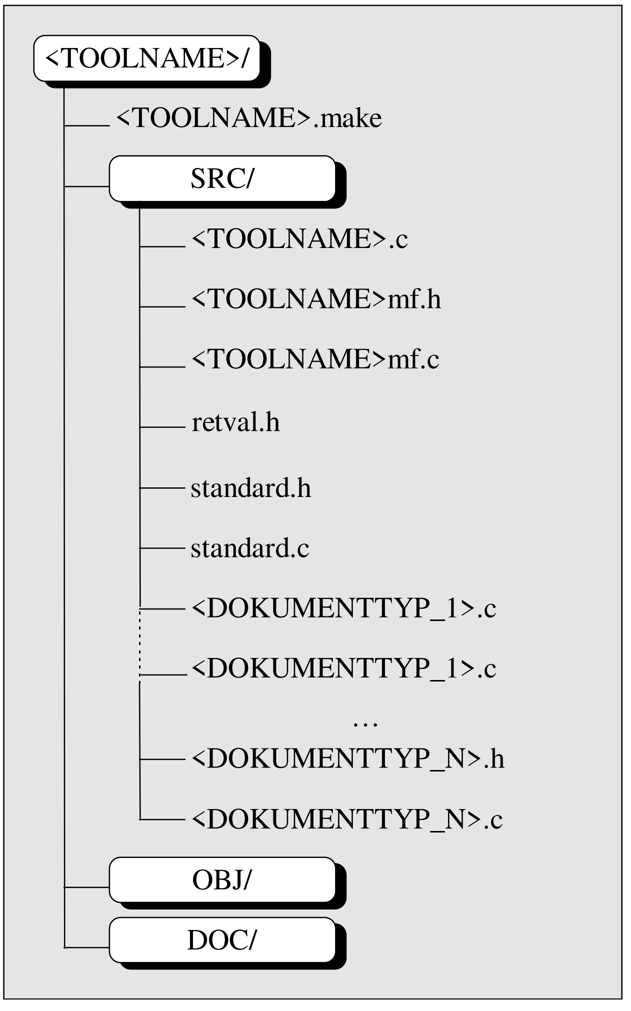

Abbildung 24: Codegenerierung nach Ansatz 4

Die im letzten Ansatz beschriebene Struktur berücksichtigt alle in den vorangegangenen Abschnitten gestellten Anforderungen und wird im folgenden beim Design des Meta-Tool als Basis verwendet.

5.2.3 Schnittstellen des Werkzeugs

Die mit dem Meta-Tool erzeugten Werkzeuge sollen drei verschiedene Schnittstellen zur Verfügung stellen. Diese werden im folgenden detailliert beschrieben.

Die erste Schnittstelle bietet die Möglichkeit eine Funktion eines erzeugten Werkzeugs durch Kommandozeilenaufrufe von einer Kommandoshell aus, auszuführen. Der Kommandozeilen-aufruf

```
TOOLNAME FUNKTIONSNAME DOKUMENTTYP PARAMETER
```

veranlaßt das Werkzeug `TOOLNAME` (z.B. `pdm` oder `dat`) die Funktion `FUNKTIONSNAME` auf dem Dokument `PARAMETER` auszuführen, das vom Typ `DOKUMENTTYP` ist.

Die zweite Schnittstelle bietet die Möglichkeit, mehrere Funktionen nacheinander auszuführen, ohne das Werkzeug für jede Funktion erneut aufrufen zu müssen. Um diese Schnittstelle zu aktivieren, muß das Werkzeug zunächst ohne jegliche Parameter aufgerufen werden. Der Kommandozeilenaufruf

```
TOOLNAME
```

startet das Werkzeug mit dem Namen `TOOLNAME`. Da dem Werkzeug keine Parameter übergeben wurden, wird vom Werkzeug ein Prompt (>) angezeigt. Von diesem Prompt aus, können beliebige Funktionen des Werkzeugs aufgerufen werden. Das Werkzeug führt die Funktion aus und kehrt zum Prompt zurück. Der Aufruf

`> FUNKTIONSNAME DOKUMENTTYP PARAMETER` (Funktionsaufruf)
`>` (Rückkehr zum Prompt)

veranlaßt das Werkzeug die Funktion `FUNKTIONSNAME` auf dem Dokument `PARAMETER` auszuführen, das vom Typ `DOKUMENTTYP` ist. Nach der Rückkehr zum Prompt (>) kann die nächste Funktion ausgeführt werden. Um die Bearbeitung zu beenden, muß der Befehl `exit` eingegeben werden. Daraufhin bricht das Werkzeug ab und kehrt zum Systempromt (zur UNIX-Shell) zurück.

Die dritte Schnittstelle bietet die Möglichkeit, die Funktionalität des Werkzeugs auch von anderen Anwendungen oder Oberflächen aus zu nutzen. Zu diesem Zweck existiert das C-Modul `<TOOLNAME>mf.h/c`, das in andere Anwendungen oder Oberflächen eingebunden werden kann. Das C-Modul stellt eine Funktion

```
TOOLNAME_MainFunc (int argc, char *argv) : tRetVal
```

zur Verfügung, die als Parameter eine Zeichenkette übergeben bekommt. Diese Zeichenkette muß in ihrem Aufbau wie folgt aussehen:

```
FUNKTIONSNAME DOKUMENTTYP PARAMETER.
```

Sie enthält alle notwendigen Angaben zur Ausführung einer bestimmten Funktion des Werkzeugs. Der Aufruf

```
TOOLNAME_MainFunc (int argc, char *argv)
```

führt die Funktion `FUNKTIONSNAME` auf dem Dokument `PARAMETER` aus, das vom Typ `DOKUMENTTYP` ist. Die Funktion gibt als Ergebnis einen Record vom Typ `tRetVal` zurück, der das Ergebnis der Funktion enthält.

5.3 Design

Bei der Konstruktion eines neuen Werkzeugs werden verschiedene Phasen der Werkzeugkonstruktion durchlaufen. Während dieser Phasen kommt das Meta-Tool zum Einsatz und muß dabei im wesentlichen zwei Phasen unterstützen. Die erste Phase ist die Generierung eines neuen Basiswerkzeugs und die zweite Phase ist die Modifikation des Basiswerkzeugs beziehungsweise eines bereits vorhandenen Werkzeugs. Die beiden Phasen werden durch jeweils eigene Teil-Werkzeuge des Meta-Tools unterstützt. Das Meta-Tool 1 für die Generierung eines neuen Basiswerkzeugs trägt den Namen *CT* (**C**reate-**T**ool). Das Meta-Tool 2 für die Modifikation eines bestehenden Werkzeugs trägt den Namen *TM* (**T**ool-**M**odifier). Die folgende Tabelle zeigt den Ablauf einer Werkzeugkonstruktion mit Hilfe des Meta-Tools:

Phase	Operation	auszuführen durch
1	Entwurf eines neuen Werkzeugs	Entwickler
2	Generierung eines Grundgerüsts für ein neues Basiswerkzeug	CT-Werkzeug
3	Ergänzen der Funktionalität des Basiswerkzeugs	TM-Werkzeug
4	Implementation der Funktionalität	Entwickler
5	Kompilieren des Werkzeugs	Entwickler / `make`-Werkzeug

Tabelle 1: Phasen der Werkzeug-Konstruktion

5.3.1 Basis der Meta-Tool-Komponenten (`ct` und `tm`)

Die Codegenerierung basiert auf zwei wesentlichen Komponenten, die sich aus den vorangegangenen Betrachtungen ergeben. Komponente eins sind die beiden Werkzeuge `ct` und `tm`, Komponente zwei sind die Templates, die die Vorlagen für die zu erzeugenden Werkzeuge beinhalten.

Die Templates sind Textdateien und werden sowohl von `ct` als auch von `tm` benötigt. Aus diesem Grund müssen sie zentral verwaltet werden, um Inkonsistenzen zu vermeiden. Deshalb werden die Templates in einem getrennten Verzeichnis abgelegt, auf das beide Werkzeuge (`ct` und `tm`) zugreifen können. Das Verzeichnis trägt den Namen `TEM/`.

Die ausführbaren Teile der Werkzeuge `TM` und `CT` sowie ihre Übersetzungsvorschriften werden im Basisverzeichnis des Meta-Tools abgelegt. Die anderen zu den Werkzeugen gehörenden Dokumenten wie Sourcecodes, Objektcodes und Dokumentation werden in getrennten Verzeichnissen abgelegt.

Die Sourcecodes befinden sich im Verzeichnis `SRC/`, die Objektcodes befinden sich im Verzeichnis `OBJ/`, die Dokumentation befindet sich im Verzeichnis `DOC/` und die Templates befinden sich im Verzeichnis `TEM/`.

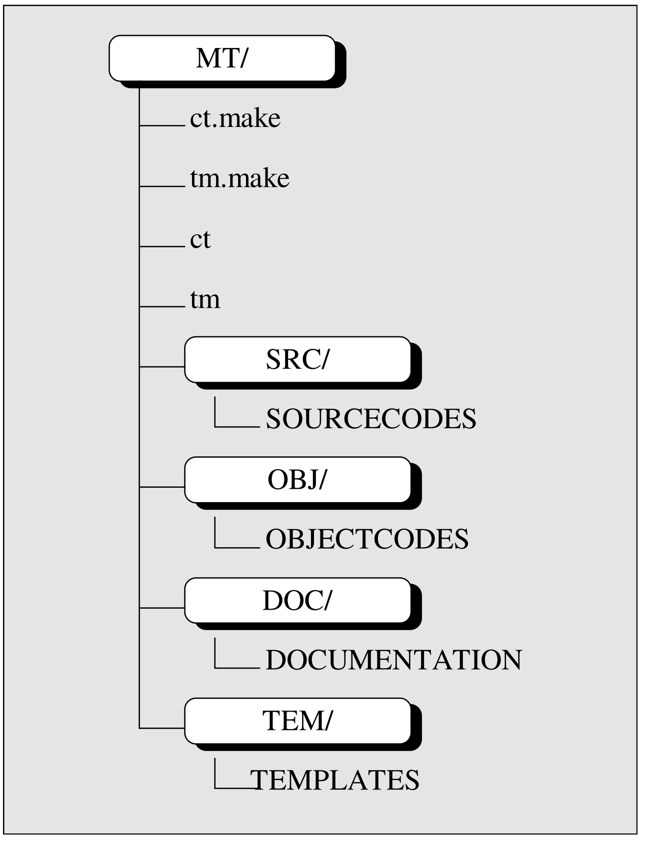

Abbildung 25: Verzeichnisstruktur des (der) Meta-Tools

Die Templates

Templates werden für alle zu generierenden Dateien eines Werkzeugs benötigt. Welche Templates im einzelnen benötigt werden, läßt sich aus Abschnitt 5.2 ableiten:

1. `makefile.tem`

   Template für das zu generierende Makefile `<TOOLNAME>.make`
2. `main.tem`

   Template für die zu generierende Datei `<TOOLNAME>.c`
3. `mf_h.tem`

   Template für die zu generierende Datei `<TOOLNAME>mf.h`
4. `mf_c.tem`

   Template für die zu generierende Datei `<TOOLNAME>mf.c`
5. `module_h.tem`

   Template für die zu generierenden Dateien `<DOKUMENTNAME>.h`

6. `module_c.tem`
   Template für die zu generierenden Dateien `<DOKUMENTNAME>.c`

Wie in den Anforderungen beschrieben, müssen die Meta-Tools die Fähigkeit haben, auf Makros zu reagieren, die in den Templates als Platzhalter verwandt werden. Die Reaktion auf ein Makro ist im Normalfall die Ersetzung des Makros durch Sourcecode. Dies kann entweder durch das einfache Einsetzen eines Namens (Programmname, Funktionsname...) oder durch das Einfügen eines neuen Templates geschehen. Zu diesem Zweck müssen weitere Templates vorhanden sein, die an den entsprechenden Stellen eingefügt werden können. Welche Templates hier im einzelnen von Bedeutung sein können, kann hier noch nicht abgeschätzt werden, da auch der spätere Benutzer der Meta-Tools die Templates erweitern oder verändern kann. Deshalb wird an dieser Stelle nicht auf die Definition weiterer Templates eingegangen. Auf die Struktur der vorhandenen Templates und die zu verwendende Syntax wird im Rahmen der Beschreibung der Implementation näher eingegangen.

Gemeinsam genutzte Funktionen
Die Aufgabe der Meta-Tools besteht im wesentlichen im Einlesen von Templates und von Sourcecode, der Bearbeitung darin befindlicher Makros und der Erzeugung neuer Dateien. Der Ablauf der Codegenerierung sieht dabei wie folgt aus:

Schritt	Aktion
1	Der Benutzer ruft das Meta-Tool mit Kommandozeilenparametern auf.
2	Das Meta-Tool bekommt die Kommandozeilenparameter übergeben und liest die notwendigen Templates oder Sourcecode-Dateien ein.
3	Das Meta-Tool erzeugt aus den Kommandozeilenparametern und dem eingelesenen Template oder Sourcecode-Dateien eine neue Datei.

Tabelle 2: Ablauf der Codeerzeugung

Es existieren also verschiedene Arbeitsbereiche bei der Codegenerierung. Tabelle 3 zeigt die verschiedenen Arbeitsbereiche und die Moduln, die die jeweilige Funktionalität zur Verfügung stellen. Die jeweiligen Funktionen, die für die verschiedenen Arbeitsbereiche benötigt werden, sind in entsprechenden Moduln implementiert.

Funktionalität	zuständiges Modul
Zugriff auf Templates und Sourcecodedateien	`fileedit.h,fileedit.c`
Modifikation von Zeichenketten, um Makros zu bearbeiten	`strutil.h,strutil.c`
Typdefintionen mit allgemeiner Gültigkeit	`types.h`
Bearbeitung von Makros	`macros.h,macros.c`
Zugriffspfade und Templatenamen	`config.h`

Tabelle 3: Moduln und die von ihnen angebotene Funktionalität

Durch die Aufteilung der Funktionalitäten in verschiedene Bereiche entstehen die in der Tabelle genannten Moduln (siehe Teil II dieser Diplomarbeit). Diese Moduln werden von beiden Meta-Tools (`ct` und `tm`) gemeinsam genutzt. Dies hat den Vorteil, daß bei Änderung einzelner Moduln die Konsistenz der beiden Werkzeuge nicht gefährdet ist. Es müssen nach einer Änderung lediglich beide Meta-Werkzeuge neu übersetzt werden.

5.3.2 Das Meta-Tool 1 (`ct`)

`ct` ist für die Generierung eines Basiswerkkzeugs zuständig. Die zu erzeugenden Strukturen und Dateien ergeben sich aus Abschnitt 5.2. Die Aufrufkonvention für `ct` beinhaltet nur einen einzigen zu übergebenden Parameter. Dieser Parameter ist der Name des neu zu erzeugenden Werkzeugs.

Aufruf: `ct /t <TOOLNAME>`

Das Ergebnis der Codegenerierung ist ein übersetzungsfähiges Werkzeug, das mit einem C-Compiler in ein lauffähiges Werkzeug ohne Funktionalität übersetzt werden kann. Der Ablauf der Codegenerierung verläuft dabei wie in der folgenden Tabelle gezeigt. Die zugrundeliegende Verzeichnisstruktur befindet sich im Abschnitt 5.2.1.

	Arbeitsschritt
1	**Erzeugen der Verzeichnisse** 1. <TOOLNAME> 2. <TOOLNAME>/SRC 3. <TOOLNAME>/OBJ 4. <TOOLNAME>/DOC
2	**Erzeugen des Makefiles** 1. Einlesen des Templates makefile.tem – Bearbeitung der Makros des Templates 2. Generieren der Datei <TOOLNAME>/<TOOLNAME>.make
3	**Erzeugen der Sourcecode-Dateien** 1. Einlesen des Templates retval.tem – Bearbeitung der Makros des Templates 2. Generieren der Datei <TOOLNAME>/SRC/retval.h 3. Einlesen des Templates main.tem - Bearbeitung der Makros des Templates 4. Generieren der Datei <TOOLNAME>/SRC/<TOOLNAME>.c 5. Einlesen des Templates mf_h.tem – Bearbeitung der Makros des Templates 6. Generieren der Datei <TOOLNAME>/SRC/<TOOLNAME>mf.h 7. Einlesen des Templates mf_c.tem – Bearbeitung der Makros des Templates 8. Generieren der Datei <TOOLNAME>/SRC/<TOOLNAME>mf.c 9. Einlesen des Templates module_h.tem – Bearbeitung der Makros des Templates 10. Generieren der Datei <TOOLNAME>/SRC/all.h 11. Einlesen des Templates module_c.tem Bearbeitung der Makros des Templates 12. Generieren der Datei <TOOLNAME>/SRC/all.c

Tabelle 4: Ablauf der Codegenerierung durch ct

5.3.3 Das Meta-Tool 2 (tm)

tm ist für die Modifikation bereits generierter Werkzeuge zuständig. Dabei greift tm nicht nur auf die Templates zu, sondern auch auf die bestehenden Sourcecode-Dateien. Die Aufrufkonventionen für tm beinhalten zwei Parameter. Der erste Parameter gibt den Namen des zu modifizierenden Werkzeugs an, der zweite Parameter ist abhängig davon, ob ein neuer Dokumenttyp oder eine neue Funktion eingeführt werden soll.

Aufruf 1: tm /t <TOOLNAME> /d <DOKUMENTTYP>

Aufruf 2: tm /t <TOOLNAME> /f <FUNKTIONSNAME>

Der erste Aufruf entspricht der Einführung eines neuen Dokumenttyps, der zweite Aufruf der Einführung einer neuen Funktion. Das Ergebnis der Codemodifikation ist ein übersetzungsfähiges Werkzeug, das mit einem C-Compiler in ein lauffähiges Werkzeug übersetzt werden kann. Der Ablauf der Codemodifikation verläuft dabei wie in der folgenden Tabelle gezeigt.

	Arbeitsschritt
1	**Modifikation des Makefiles** 1. Einlesen des Templates makefile.tem Bearbeitung der Makros des Templates 2. Generieren der neuen Datei <TOOLNAME>/<TOOLNAME>.make
2	**Erzeugen der Sourcecode-Dateien** 1. Einlesen der Datei <TOOLNAME>/SRC/<TOOLNAME>mf.c Erweiterung der Dokument-/Funktionsdefinition 2. Generieren der Datei <TOOLNAME>/SRC/<TOOLNAME>mf.c 3. Falls ein neuer Dokumenttyp eingebunden werden soll, wird ein neues Modul für diesen Dokumenttyp erzeugt. a) Einlesen des Templates module_h.tem - Bearbeitung der Makros des Templates b) Generieren der Datei <TOOLNAME>/SRC/<DOKUMENTNAME>.h c) Einlesen des Templates module_c.tem - Bearbeitung der Makros des Templates d) Generieren der Datei <TOOLNAME>/SRC/<DOKUMENTNAME>.c e) Ergänzen des neuen Moduls um bereits vorhandene Funktionalität

	Arbeitsschritt
2	4. Falls eine neue Funktion eingeführt werden soll, werden alle bestehenden Moduln um die neue Funktion erweitert. a) Einlesen der vorhandenen Moduln <TOOLNAME>/SRC/<DOKUMENTNAME>.h – Erweitern der Funktionalität um die neue Funktion b) Generieren des neuen Moduls <TOOLNAME>/SRC/<DOKUMENTNAME>.h c) Einlesen der vorhandenen Moduln <TOOLNAME>/SRC/<DOKUMENTNAME>.c – Erweitern der Funktionalität um die neue Funktion d) Generieren des neuen Moduls <TOOLNAME>/SRC/<DOKUMENTNAME>.c Die Schritte 4. a) bis 4. d) werden für alle bereits vorhandenen Moduln ausgeführt.

Tabelle 5: Ablauf der Codegenerierung durch tm

5.4 Implementation

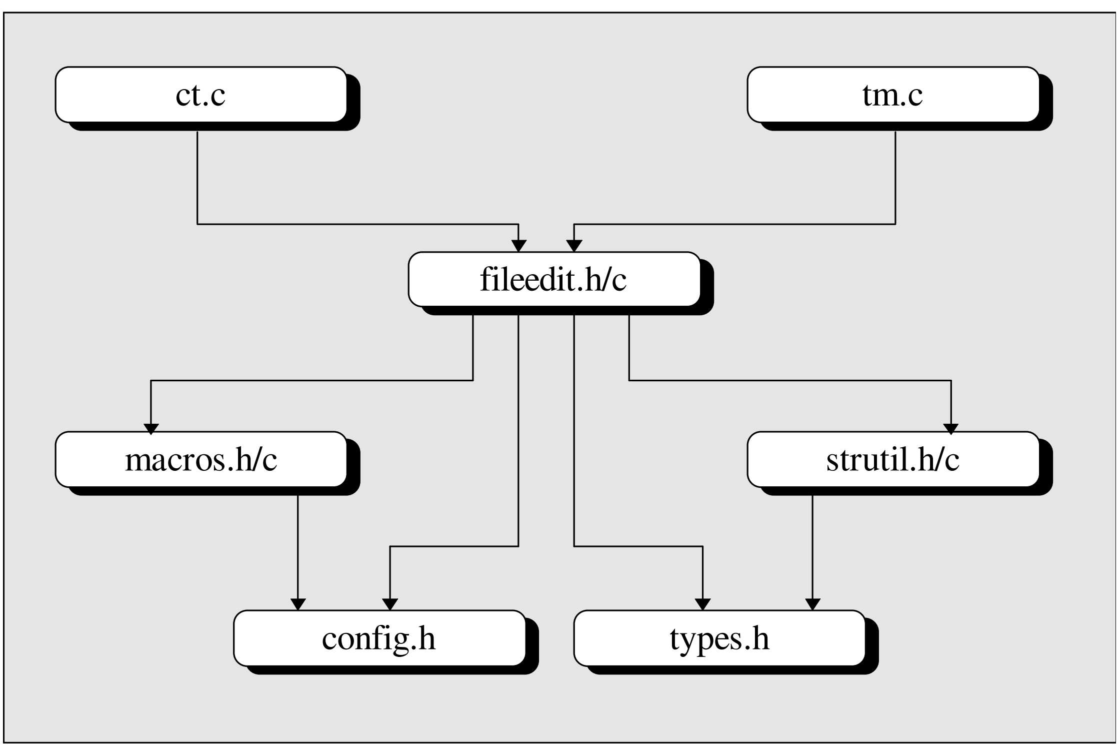

Abbildung 26: Modulhierarchie des Meta-Tools

In diesem Abschnitt wird die Implementation der Meta-Tools beschrieben. Dabei wird zunächst die Modulhierarchie erläutert. Danach werden die einzelnen Moduln beschrieben.

5.4.1 Die Modulhierarchie

Die Abbildung 26 zeigt die Benutzt-Hierachie der Moduln des Meta-Tools. Dabei wurde auf Darstellung der sich durch die Transitivität ergebenden Beziehungen aus Gründen der Übersichtlichkeit verzichtet.

5.4.2 Die Datei `config.h`

```
/* ------------------------------------------- */
/* Directories                                 */
/* ------------------------------------------- */

#define _TOOL_DIR         ""
#define _TOOL_SRC_DIR     "/SRC"
#define _TOOL_OBJ_DIR     "/OBJ"
#define _TOOL_DOC_DIR     "/DOC"
#define _TEMPLATE_DIR     "/TEM"



/* ------------------------------------------- */
/* Templates                                   */
/* ------------------------------------------- */

#define _MAIN_TEM         "/main.tem"
#define _MF_H_TEM         "/mf_h.tem"
#define _MF_C_TEM         "/mf_c.tem"
#define _RETVAL_TEM       "/retval.tem"
#define _MAKEFILE_TEM     "/makefile.tem"
#define _MODULE_C_TEM     "/module_c.tem"
#define _MODULE_H_TEM     "/module_h.tem"
#define _FUNC_C_TEM       "/func_c.tem"
#define _FUNC_H_TEM       "/func_h.tem"
#define _PARAM_TEM        "/param.tem"
```

Abbildung 27: Definitionen in `config.h`

Diese Header-Datei beinhaltet Basisdefinitionen für die Codegenerierung. In ihr werden die Verzeichnisse definiert, die im Rahmen der Codegenerierung angelegt werden müssen. Zusätzlich sind hier die Namen der Templates definiert, die bei der Codegenerierung eingelesen werden. Durch Änderung der Definitionen in der Datei `config.h` verändert sich das Verhalten der Meta-Tools bzgl. der erzeugten Verzeichnisstruktur und der verwendeten Templates. Änderungen sollten nur unter Berücksichtigung der Tatsache durchgeführt werden, daß auch alle bereits generierten Werkzeuge davon betroffen sind.

5.4.3 Die Datei `types.h`

Diese Header-Datei beinhaltet Typdefinitionen, die von allen beteiligten Moduln benötigt werden. Darunter sind abstrakte Datentypen wie `tToolInfo`, der den Status eines erzeugten Werkzeugs erfaßt, als auch Definitionen von global genutzten Konstanten.

```
/*
  *******************************************
  SECTION: LENGTH DEFINITION
  ___________________________________________
*/

#define MAXFILENAMELEN   40
#define MAXFUNCNAMELEN   40


/*
  *******************************************
  SECTION: TYPEDEF
  ___________________________________________
*/

typedef struct {
  char  *ToolName;
  char  *ToolDir;
  char  *SrcDir;
  char  *ObjDir;
  char  *DocDir;
  int    Version;
  int    DocNum;
  int    DocIndex;
  char **Docs;
  int    FuncNum;
  int    FuncIndex;
  char **Funcs;
} tToolInfo;
```

Abbildung 28: Auszug aus der Datei `types.h`

5.4.4 Die Datei `macros.h/c`

Die Dateien `macros.h` und `macros.c` beinhalten Funktionen zur Bearbeitung von Makros, die in den Templates vorkommen. Die Bearbeitung von Makros beruht auf dem Prinzip des Ersetzens eines bestimmten Makros durch anderen Text. Dabei kann ein Makro sowohl durch einen einzelnen Begriff, wie beispielsweise den Namen einer Funktion oder eines Dokuments ersetzt werden als auch durch einen größeren Sourcecode-Teil, wie beispielsweise einer Liste von Funktionen oder Dokumentnamen. Alle Funktionen zur Bearbeitung von Makros haben die gleichen Aufrufkonventionen, da sie durch das Meta-Tool zentral aufgerufen werden. Die Basisdefinition einer Makrofunktion sieht wie folgt aus:

```
char *mf_<MACRONAME> (tFuncParameters Params);
```

Der Typ `tFuncParameters` beinhaltet Informationen über das Werkzeug, das momentan bearbeitet wird sowie die aktuelle Zeile des Templates das gerade bearbeitet wird. Anhand dieser Informationen können die Makrofunktionen die aktuelle Zeile modifizieren und liefern die neue Zeile als Rückgabewert.

```
typedef char* (*pFunction)();

#define NumOfMacros 13

extern char *Macro[];
extern pFunction MacroFunction [];

char *mf_DocName (tFuncParameters Params);

char *mf_FuncName (tFuncParameters Params);

char *mf_Documents (tFuncParameters Params);

char *mf_Functions (tFuncParameters Params);

char *mf_IncludeFiles (tFuncParameters Params);

char *mf_Operations (tFuncParameters Params);

char *mf_MakeModules (tFuncParameters Params);
```

Abbildung 29: Auszug aus der Datei `macros.h`

Die Liste der Makros und Makrofunktionen kann vom Werkzeugentwickler ergänzt und erweitert werden, um den Meta-Tools neue Funktionalität hinzuzufügen. Dies erlaubt eine flexible Codegenerierung, die die individuellen Bedürfnisse der Prototyping-Umgebung angepaßt werden kann. Nachdem die Funktionen zur Makrobearbeitung modifiziert worden sind, muß das Meta-Tool neu übersetzt werden, damit die Änderungen aktiv werden.

Falls ein Werkzeug mit einem Namen generiert wird, der dem Namen eines bereits bestehenden Werkzeugs entspricht, wird das bestehende Werkzeug durch das neue Werkzeug überschrieben.

Basismakros

An dieser Stelle werden die bereits implementierten Basismakros beschrieben. Welche Funktion sie im einzelnen haben, wird im folgenden kurz erläutert.

1. `$TOOLNAME$`

   Dieses Makro wird ersetzt durch den Namen des neu zu erzeugenden Werkzeugs.

2. `$DOCNUM$`

   Dieses Makro wird ersetzt durch die Anzahl der momentan zur Verfügung stehenden Dokumenttypen. Damit sind an dieser Stelle die Dokumenttypen gemeint, die mit dem erzeugten Werkzeug bearbeitet werden können.

3. `$FUNCNUM$`

   Dieses Makro wird ersetzt durch die Anzahl der momentan zur Verfügung stehenden Funktionstypen. Damit sind an dieser Stelle die Funktionstypen gemeint, die mit dem erzeugten Werkzeug ausgeführt werden können.

4. `$DOCNAME$`

   Dieses Makro wird ersetzt durch den einen Dokumentnamen.

5. `$FUNCNAME$`

   Dieses Makro wird ersetzt durch den einen Funktionsnamen.

6. `$DOCUMENTS$`

   Dieses Makro wird ersetzt durch eine Liste aller momentan verfügbaren Dokumenttypen. Damit sind an dieser Stelle die Dokumenttypen gemeint, die mit dem erzeugten Werkzeug bearbeitet werden können.

7. `$FUNCTIONS$`

   Dieses Makro wird ersetzt durch eine Liste aller momentan verfügbaren Funktionstypen. Damit sind an dieser Stelle die Funktionstypen gemeint, die mit dem erzeugten Werkzeug ausgeführt werden können.

8. `$INCLUDEFILES$`

   Bewirkt das Einfügen des Makros der Definitionen aller notwendigen Include-Files. Damit sind hier alle Include-Files (Moduln) gemeint, in denen die Implementation der Dokumentfunktionen steht.

9. `$OPERATIONS$`

   Bewirkt das Einfügen einer Liste der aktuell für das Werkzeug verfügbaren Operationen (Funktionen).

10. `$MAKEMODULES$`

    Bewirkt das Einfügen der Bildungsvorschriften für den Objectcode für die zu erzeugenden Moduln des Werkzeugs.

11. `$MAKEEXE$`

    Bewirkt das Einfügen der Bildungsvorschrift für das zu erzeugende Werkzeug.

12. `$OBJEKTFILES$`

    Bewirkt das Einfügen einer Liste der für das Binden des Werkzeugs erforderlichen Modul-Objektdateien.

13. `$VERSION$`

    Dieses Makro wird ersetzt durch die aktuelle Versionsnummer.

5.4.5 Die Datei `strutil.h/c`

Das Modul `strutil.h/c` bietet Hilfsfunktionen zur Bearbeitung von Zeichenketten an. Die beiden wesentlichen Funktionen werden im folgenden näher erläutert.

```
/*
  ***********************************************
  Name   NEWSTR
  Datum  25.03.1995
  Author J. Heinrich

  STRING-Hilfs-Funktion.
  Die Funktion ersetzt in SearchStr alle
  Vorkommen von Source durch Dest.
  _______________________________________________
*/

char *NewStr (size_t Len);


/*
  ***********************************************
  Name   REPLACESTR
  Datum  21.12.1994
  Author J. Heinrich

  STRING-Hilfs-Funktion.
  Die Funktion ersetzt in SearchStr alle
  Vorkommen von Source durch Dest.
  _______________________________________________
*/

char *ReplaceStr (char  *SearchStr,
                  char  *Source,
                  char  *Dest);
```

Abbildung 30: Auszug aus der Datei `strutil.h`

Die Funktion `char *NewStr (t_size Len);`

Diese Funktion reserviert `Len` Bytes Speicherplatz auf dem Heap für eine neue Zeichenkette und füllt den reservierten Bereich mit NULL-Zeichen (‘\0’). Rückgabewert ist die neue mit Nullen gefüllte Zeichenkette.

Die Funktion
```
char *ReplaceStr (    char *SearchStr,
                      char *Source,
                      char *Dest);
```
Diese Funktion gibt eine neue Zeichenkette zurück, in der alle Vorkommen von `Source` in `SearchStr` durch `Dest` ersetzt worden sind.

5.4.6 Die Datei `fileedit.h/c`

Das Modul `fileedit.h/c` beinhaltet zwei Funktionsgruppen, die für die Generierung der Sourcecode-Dateien notwendig sind. Dies sind zum einen Funktionen, die die Arbeit mit dem bereits oben erwähnten Typ `tToolInfo` betreffen und zum anderen Funktionen, die für den Zugriff auf die Templates sowie die Erzeugung der Verzeichnisstruktur und der Sourcecode-Dateien notwendig sind. Die Funktionen werden im folgenden genauer erläutert:

Die Funktion `void GetToolEnv ()`

liest die Umgebungsvariable mit dem Namen `ToolEnv` ein, in der der Zugriffspfad für die Meta-Tools abgelegt ist.

Die Funktion
```
void GetToolInfo (    char *ToolName,
                      tToolInfo *AToolInfo)
```
liest Informationen aus dem Sourcecode des Werkzeugs `ToolName` in die Variable `AToolInfo` ein. Diese Informationen beinhalten die Anzahl der Dokument-/Funktionstypen, die Namen der Dokument-/Funktionstypen sowie Angaben über die Verzeichnisse in denen die Dateien des Werkzeugs abgelegt sind.

Die Funktion `int MakeDirectories (tToolInfo ToolInfo)`

erzeugt die Verzeichnisstruktur für ein neues Werkzeug. Genaue Informationen zu dem neu zu erzeugenden Werkzeug werden in dem Parameter `ToolInfo` übergeben.

Die Funktion `void GenerateTool (tToolInfo ToolInfo)`

generiert die Sourcecode-Dateien für ein neues Werkzeug. Genaue Informationen zu dem neu zu erzeugenden Werkzeug werden in dem Parameter `ToolInfo` übergeben.

Die Funktion `void GenerateMakefile (tToolInfo ToolInfo)`
generiert das Makefile für ein neues Werkzeug. Genaue Informationen zu dem neu zu erzeugenden Werkzeug werden in dem Parameter `ToolInfo` übergeben.

Die Funktion
```
void EditModules (tToolInfo OldToolInfo,
                  tToolInfo NewToolInfo)
```
prüft die Moduln eines vorhandenen Moduls (`OldToolInfo`) und gleicht die Funktionalität mit den Informationen aus `NewToolInfo` ab. Falls notwendig, werden neue Moduln oder Funktionen erzeugt.

Die Funktion
```
void InitializeToolInfo (pToolInfo  ToolInfo,
                         char      *ToolPath)
```
initialisiert den Übergabeparameter `ToolInfo` mit den Basiswerten für ein neues Werkzeug. Die Basiswerte werden entsprechend dem Parameter `ToolPath` gesetzt, der den Basisnamen des Werkzeugverzeichnisses enthält.

Die Funktion
```
void AddDocument (pToolInfo  ToolInfo,
                  char      *Document)
```
ergänzt die Werkzeuginformation `ToolInfo` um den neuen Dokumenttypen, der im Parameter `Document` übergeben wird.

Die Funktion
```
void AddFunction (pToolInfo  ToolInfo,
                  char      *Function)
```
ergänzt die Werkzeuginformation `ToolInfo` um den neuen Funktionstypen der im Parameter `Function` übergeben wird.

5.4.7 Die Datei `ct.c`

Diese Datei beinhaltet den Sourcecode des Meta-Tools `ct`. Dieser besteht im wesentlichen aus einer Routine zur Bearbeitung der Kommandozeilenparameter und aus den Aufrufen der entsprechenden Generierungsfunktionen aus dem Modul `fileedit.h/c`. Die folgende Abbildung zeigt einen Auszug aus der Datei `ct.c`, der die wesentlichen Funktionsaufrufe beinhaltet. Das Meta-Tool `ct` erzeugt anhand des Kommandozeilenparameters `<TOOLNAME>` ein neues Werkzeug mit entsprechendem Namen.

```
GetToolEnv ();

/* ----------------------- */
/* Verzeichnisse anlegen   */
/* ----------------------- */

MakeDirectories (NewToolInfo);

/* ----------------------- */
/* Werkzeug generieren     */
/* ----------------------- */

GenerateTool (NewToolInfo);

/* ----------------------- */
/* Makefile generieren     */
/* ----------------------- */

GenerateMakefile (NewToolInfo);

/* ----------------------- */
/* Module generieren       */
/* ----------------------- */

EditModules (OldToolInfo,NewToolInfo);
```

Abbildung 31: Auszug aus der Datei `ct.c`

5.4.8 Die Datei `tm.c`

Diese Datei beinhaltet den Sourcecode des Meta-Tools `tm`. Dieser besteht im wesentlichen aus einer Routine zur Bearbeitung der Kommandozeilenparameter und aus den Aufrufen der entsprechenden Generierungsfunktionen aus dem Modul `fileedit.h/c`. Die folgende Abbildung zeigt einen Auszug aus der Datei `tm.c`, der die wesentlichen Funktionsaufrufe beinhaltet. Das Werkzeug `tm` erzeugt anhand der übergebenen Kommandozeilenparameter eine neue Version eines bereits vorhandenen Werkzeugs.

```
/* Neuen Dokumenttyp einfuegen */

if ((ToolName != NULL) &&
    (DocName  != NULL))
 {
  AddDocument (&NewToolInfo,DocName);
  GenerateTool (NewToolInfo);
  GenerateMakefile (NewToolInfo);
  EditModules (OldToolInfo,NewToolInfo);
 }

/* Neuen Funktionstyp einfuegen */

else if ((ToolName != NULL) &&
         (FuncName != NULL))
 {
  AddFunction (&NewToolInfo,FuncName);
  GenerateTool (NewToolInfo);
  GenerateMakefile (NewToolInfo);
  EditModules (OldToolInfo,NewToolInfo);
 }
GetToolEnv ();

MakeDirectories (NewToolInfo);
GenerateTool (NewToolInfo);
GenerateMakefile (NewToolInfo);
EditModules (OldToolInfo,NewToolInfo);
```

Abbildung 32: Auszug aus der Datei `tm.c`

6 Die konkreten Werkzeuge

In diesem Abschnitt wird die Implementation der beiden konkret zu realisierenden Werkzeuge beschrieben.

6.1 Datenverwaltungswerkzeug (`pdm`)

Die Konstruktion des Datenverwaltungswerkzeugs wird in den folgenden drei Abschnitten erläutert. Dabei werden zuerst die Anforderungen beschrieben, danach folgt das Design und zum Schluß die Implementation des Werkzeugs. Das Datenverwaltungswerkzeug soll den Namen `pdm` (PROSET-Data-Manager) tragen.

6.1.1 Anforderungen

Die Anforderungen, die an das Datenverwaltungswerkzeug gestellt werden, beziehen sich auf verschiedene Bereiche. Die Bereiche die im folgenden betrachtet werden, sind *Funktionalität*, *Sicherheitsaspekte* und *Benutzungsschnittstellen*.

Funktionalität

Der erste Teil der Anforderungen ergibt sich aus der gewünschten Funktionalität, die das Werkzeug `pdm` anbieten soll. Die Hauptaufgabe von `pdm` ist es, die Verwaltung von PROSET-Werten innerhalb des P-Store zu ermöglichen. Der P-Store ist ein Datenbereich innerhalb der Datenbank H-PCTE, in dem nur persistente PROSET-Daten abgelegt werden. Der P-Store ist zu vergleichen mit einem Dateisystem, in dem Dateien abgelegt werden. In einem Dateisystem gibt es zum einen Verzeichnisse und zum anderen Dateien. Für die Verwaltung von PROSET-Werten werden entsprechende Strukturen benötigt. Die PROSET-Werte werden in Datenobjekten abgelegt. Zur Nachbildung von Verzeichnissen werden P-Files genutzt, in denen beliebig viele Datenobjekte abgelegt werden können. und die verschiedenen PROSET-Datentypen benötigt (siehe Abschnitt 1.4). Da die P-Files eine ähnliche Aufgabe wie Verzeichnisse in einem Dateisystem haben, werden auch entsprechende Operationen benötigt. Das Werkzeug `pdm` muß es ermöglichen, P-Files im P-Store anzulegen und sie zu löschen. Um Unterstrukturen anlegen zu können, muß es ebenso möglich sein innerhalb von P-Files sogenannte Sub-P-Files anzulegen. Um in einer vorhandenen Struktur eine beliebige Ebene erreichen zu können, werden Operationen zur Navigation benötigt. Sie müssen es ermöglichen, von einem P-File in einen anderen P-File zu wechseln, beziehungsweise durch Strukturen von geschachtelten P-Files zu navigieren.

In den P-Files werden PROSET-Werte abgelegt. Um mit diesen Werten arbeiten zu können, müssen Zugriffsfunktionen angeboten werden. Diese Zugriffsfunktionen müssen es dem Benutzer erlauben, neue Werte anzulegen und bestehende Werte zu löschen. Zusätzlich werden Funktionen zum Lesen und zum Schreiben der verschiedenen Werte benötigt. Diese Funktionen werden für alle PROSET-Datentypen gebraucht. Der Benutzer benötigt im Rahmen der Navigation eine Kontrollmöglichkeit, um seinen aktuellen Standort in der P-File-Struktur bestimmen zu können. Das Werkzeug sollte anhand einer Anzeige eine solche Kontrollmöglichkeit anbieten (z.B. Anzeige des aktuellen Navigations-Pfades). Um den Inhalt eines P-Files sichtbar zu machen, muß eine dem UNIX-Shell-Kommando `ls` ähnliche Funktion zur Verfügung gestellt werden. Diese muß alle innerhalb des P-Files abgelegten PROSET-Werte und Sub-P-Files anzeigen. Diese Anzeige sollte gleichzeitig Auskunft über den Typ der angezeigten Daten geben.

Die benötigte Funktionalität, läßt sich in zwei Bereiche gliedern. Der erste Bereich enthält dokumentspezifische Funktionen, der zweite Bereich enthält dokumentunabhängige Funktionen. Die folgende Tabelle faßt die benötigten dokumentspezifischen Funktionen zusammen.

Datentyp/Operation	Lesen	Schreiben	Erzeugen	Löschen
`Integer`	X	X	X	X
`Real`	X	X	X	X
`String`	X	X	X	X
`Boolean`	X	X	X	X
`Set`	X		X	X
`Tupel`	X		X	X

Tabelle 6: Dokumentspezifische Operationen

Sicherheitsaspekte

Im Rahmen der Sicherheitsaspekte spielt u.a. neben der Systemsicherheit die Datensicherheit eine wesentliche Rolle. Die Datensicherheit in einem Mehrbenutzersystem kann nur mit entsprechenden Mechanismen gewährleistet werden. Das Werkzeug sollte das von der ProSet-H-PCTE-Schnittstelle unterstützte Transaktionsmanagement nutzen. Dies ist insbesondere deshalb wichtig, weil sich lesende und schreibende Zugriffe auf Daten mit den Zugriffen anderer Benutzer zeitlich überschneiden können. In einem solchen Fall könnten Inkonsistenzen auftreten, die durch ein Transaktionsmanagement vermieden werden. Desweiteren kann es passieren, daß Zugriffe auf Daten, die intern aus mehreren einzelnen Operationen bestehen, nicht vollständig durchgeführt werden. In einem solchen Fall muß das Werkzeug die unvollständig ausgeführte Operation rückgängig machen und den ursprünglichen Datenzustand wiederherstellen.

Benutzungsschnittstellen

Das Werkzeug soll verschiedene Benutzungsschnittstellen unterstützen. Die drei benötigten Benutzungsschnittstellen werden im folgenden beschrieben.

<u>Einzel-Funktions-Benutzungsschnittstelle</u>

Es soll möglich sein, eine einzelne Funktion des Werkzeugs durch einen Kommandozeilenaufruf mit den notwendigen Parametern auszuführen. Dabei wird dem Werkzeug der Name der auszuführenden Funktion sowie der Name und der Typ des ProSet-Wertes, auf das die Funktion anzuwenden ist, mitgeteilt.

<u>Multi-Funktions-Benutzungsschnittstelle</u>

Eine zweite Benutzungsschnittstelle muß es dem Benutzer ermöglichen, mehrere Funktionen hintereinander auszuführen, ohne das Werkzeug dabei jedesmal erneut aufrufen zu müssen. Das bedeutet, daß eine interaktive Benutzungsschnittstelle zur Verfügung gestellt werden muß. Das Werkzeug muß in diesem Fall ohne Kommandozeilenparameter aufgerufen werden. Nach dem Aufruf muß das Werkzeug eine Verbindung zur Datenbank H-PCTE herstellen und eine interaktive textorientierte Schnittstelle zur Verfügung stellen, so daß die Funktionen aufgerufen werden können. Erst nach der Eingabe eines definierten Kommandos wird die Verbindung zur Datenbank unterbrochen und die Ausführung des Werkzeugs beendet.

<u>Externe-Benutzungsschnittstelle</u>

Eine dritte Schnittstelle muß es ermöglichen, die Funktionalität des Werkzeugs auch für andere Anwendungen verfügbar zu machen. Die Funktionen des Werkzeugs müssen zu diesem Zweck in einem C-Modul zur Verfügung gestellt werden. Dieses C-Modul kann in andere Anwendungen eingebunden werden, so daß diese die Funktionen nutzen können.

Weitere Informationen zu dem Bereich Anforderungen bzgl. der Benutzungsschnittstelle befinden sich in Abschnitt 5.2.3.

6.1.2 Design

Das Datenverwaltungswerkzeug `pdm` soll so aufgebaut sein, daß die Funktionalität für verschiedene Dokumenttypen in verschiedenen C-Moduln implementiert wird. Dabei wird der Ansatz verfolgt, für jeden Dokumenttyp ein eigenes C-Modul anzulegen. Entsprechend den PROSET-Datentypen müssen C-Moduln für die Datentypen `Integer`, `Real`, `String`, `Boolean`, `Set` und `Tuple` angelegt werden. Die dokumentunabhängigen Funktionen werden ebenso in einem eigenen C-Modul abgelegt. Dieses C-Modul bekommt den Namen eines Dummy-Dokumenttypen zugewiesen. Dieser trägt den Namen `all`. Die Sourcecodes, Objectcodes und Dokumentationsdateien des Werkzeugs werden in eigenen Verzeichnissen abgelegt. Die dafür notwendige Verzeichnisstruktur entspricht der in Abschnitt 5.2.1 beschriebenen.

Die zu realisierenden Benutzungsschnittstellen entsprechen den folgenden Aufrufkonventionen:

Einzel-Funktions-Benutzungsschnittstelle:

```
TOOLNAME FUNKTIONSNAME DOKUMENTTYP PARAMETER
```

Bei einem Einzel-Funktions-Aufruf wird das Werkzeug (`TOOLNAME`) mit den folgenden Parametern aufgerufen. Der erste Parameter (`FUNKTIONSNAME`) ist der Name der Funktion, die ausgeführt werden soll. Der zweite Parameter (`DOKUMENTTYP`) gibt den Typ des Dokuments an, auf den die durch Parameter eins definierte Funktion angewendet werden soll. Der dritte Parameter ist ein Zusatzparameter, mit dem der auszuführenden Funktion eine weitere Information übergeben wird, beispielsweise ein Dokumentname.

Multi-Funktions-Benutzungsschnittstelle:

`TOOLNAME` **(Start des Werkzeugs)**

`> FUNKTIONSNAME DOKUMENTTYP PARAMETER` **(Funktionsaufruf 1)**

…

```
> FUNKTIONSNAME DOKUMENTTYP PARAMETER
> exit
```

(Funktionsaufruf n)
(Abbruch des Werkzeugs)

Ähnlich wie bei der vorangegangenen Schnittstelle, ist der Aufruf einer Funktion mit Hilfe der Multi-Funktions-Schnittstelle strukturiert. Der Werkzeugname wird nicht benötigt, da das Werkzeug bereits gestartet ist und auf eine Eingabe wartet. Die eigentlichen Parameter entsprechen denen der Einzel-Funktions-Schnittstelle. Der erste Parameter gibt den Namen der auszuführenden Funktion an, der zweite Parameter identifiziert den Dokumenttyp auf den die Funktion anzuwenden ist und der dritte Parameter enthält eine Zusatzinformation (Dokumentname).

Externe-Benutzungsschnittstelle:

```
TOOLNAME_MainFunc (int argc, char *argv)
```

Der Aufruf der Funktion `TOOLNAME_MainFunc` beinhaltet zwei Paramter. Der erste Parameter (`argc`) sagt aus wieviele Parameter in (`argv`) übergeben werden. Die Parameter in `argv` entsprechen den in den vorangegangenen Abschnitten beschriebenen. Der erste Parameter gibt den Namen der auszuführenden Funktion an, der zweite Parameter beschreibt den Dokumenttyp auf den die Funktion angewendet werden soll und der dritte Parameter liefert eine Zusatzinformation, wie beispielsweise den Dokumentnamen.

6.1.3 Implementation

Die Implementation wird mit Hilfe des in Abschnitt 5 beschriebenen Meta-Tools durchgeführt. Im ersten Schritt wird mit dem Meta-Tool `ct` ein Basiswerkzeug mit dem Namen `pdm` erzeugt. Der entsprechende Aufruf lautet:

```
ct /t pdm
```

Durch den Aufruf werden die benötigten Verzeichnisse (siehe Abschnitt 5.2.1) sowie die für ein Basiswerkzeug notwendigen Dateien (siehe Abschnitt 5.2.2) erzeugt. Nach der Erzeugung des Basiswerkzeugs werden die für die verschiedenen Datentypen benötigten C-Moduln und Funktionen mit Hilfe des Meta-Tools `tm` erzeugt.

Die notwendigen Aufrufe werden im folgenden beschrieben:

1. Erzeugung eines C-Moduls für den Datentyp integer
   `tm /t pdm /d integer`
2. Erzeugung eines C-Moduls für den Datentyp `real`
   `tm /t pdm /d real`
3. Erzeugung eines C-Moduls für den Datentyp `string`
   `tm /t pdm /d string`
4. Erzeugung eines C-Moduls für den Datentyp `boolean`
   `tm /t pdm /d boolean`
5. Erzeugung eines C-Moduls für den Datentyp `set`
   `tm /t pdm /d set`
6. Erzeugung eines C-Moduls für den Datentyp `tuple`
   `tm /t pdm /d tuple`

Im zweiten Schritt werden die benötigten Funktionsrümpfe mit Hilfe des Meta-Tools `tm` in die C-Moduln eingefügt. Die notwendigen Aufrufe werden im folgenden beschrieben:

1. Einfügen der Funktion `read`
   `tm /t pdm /f read`
2. Einfügen der Funktion `write`
   `tm /t pdm /f write`
3. Einfügen der Funktion `create`
   `tm /t pdm /f create`
4. Einfügen der Funktion `read`
   `tm /t pdm /f delete`

Nachdem das Basiswerkzeug um die benötigten Moduln und Funktionsrümpfe ergänzt worden ist, werden die Zugriffsfunktionen der einzelnen Datentypen von Hand implementiert. Dabei werden im wesentlichen die von der ProSet-H-PCTE-Schnittstelle (siehe Kapitel 4) angebotenen Funktionen genutzt.

6.2 Datenanalysewerkzeug (`dat`)

Die Konstruktion des Datenanalysewerkzeugs wird ebenso wie die des Datenverwaltungswerkzeugs in drei Abschnitten erläutert. Dabei werden wiederum zuerst die Anforderungen beschrieben, danach erfolgt die Beschreibung des Designs und zum Schluß die Beschreibung der Implementation des Werkzeugs. Das Datenanalysewerkzeug soll den Namen `dat` (Data-Analysis-Tool) tragen.

6.2.1 Anforderungen

Die Anforderungen, die an das Datenanalysewerkzeug gestellt werden, sind wie beim Datenverwaltungswerkzeug in drei verschiedene Bereiche eingeteilt. Die Bereiche sind *Funktionalität*, *Sicherheitsaspekte* und *Benutzungsschnittstellen*.

Funktionalität

Die Hauptaufgabe von `dat` ist es, die Analyse von PROSET-Werten durch einfache Funktionen zu unterstützen. Einfache Funktionen bedeutet hier im wesentlichen Suchfunktionen. Mit den Suchfunktionen soll dem Benutzer die Möglichkeit gegeben werden, nach bestimmten Attributen von PROSET-Werten zu suchen. Diese Attribute können entweder der Name des Datums oder sein Typ sein. Die Suche nach Namen und Typen von PROSET-Werten soll unterstützt werden. Es ist sinnvoll auch eine Kombination von Namen- und Typattributen zuzulassen. Die Suchfunktionen müssen, ausgehend vom aktuellen Navigationspfad, alle Unterstrukturen gemäß den vom Benutzer gemachten Angaben untersuchen. Falls die Suche erfolgreich ist, müssen alle Daten auf die das Suchmuster zutrifft, angezeigt werden.

In einer weiteren Ausbaustufe kann die Möglichkeit angeboten werden, selektive Sichten auf den Datenbestand zu realisieren. Dazu wären Funktionen notwendig, die in die Kategorie Filter fallen. Diese Filter haben Auswirkungen auf alle nach der Definition des Filters ausgeführten Funktionen. Mit einem Filter kann die Sicht beispielsweise auf einen bestimmten Datentypen eingeschränkt werden. Dadurch würden sich alle folgenden Operationen nur noch auf diesen Datentypen beziehen.

Sicherheitsaspekte

Ebenso wie beim Datenverwaltungswerkzeug müssen alle kritischen Operationen Sicherheitsaspekte berücksichtigen. Aus diesem Grund sollte auch dieses Werkzeug das von der PROSET-H-PCTE-Schnittstelle unterstützte Transaktionsmanagement nutzen.

Benutzungsschnittstellen

Das Werkzeug soll verschiedene Benutzungsschnittstellen unterstützen. Die drei benötigten Benutzungsschnittstellen entsprechen denen des Datenverwaltungswerkzeugs und sollen hier nicht mehr erläutert werden.

Weitere Informationen zu dem Bereich Anforderungen bzgl. der Benutzungsschnittstelle befinden sich in Abschnitt 5.2.3..

6.2.2 Design

Ebenso wie das Datenverwaltungswerkzeug `dat`, soll das Datenanalysewerkzeug `dat` so aufgebaut sein, daß die Funktionalität für verschiedene Dokumenttypen in verschiedenen C-Moduln implementiert wird. Dabei wird wiederum der Ansatz verfolgt, für jeden Dokumenttyp ein eigenes C-Modul anzulegen. Entsprechend den PROSET-Datentypen müssen C-Moduln für die Datentypen `Integer`, `Real`, `String`, `Boolean`, `Set` und `Tuple` angelegt werden. Die dokumentunabhängigen Funktionen werden, wie bereits oben beschrieben, in einem eigenen C-Modul abgelegt. Die dafür notwendige Verzeichnisstruktur entspricht der in Abschnitt 5.2.1 beschriebenen. Die benötigten Funktionen werden in der folgenden Tabelle beschrieben.

Operation	Beschreibung
`searchname`	Sucht nach dem Namen eines PROSET-Wertes. Als Parameter wird nur der gesuchte Name übergeben. Diese Funktion ist typunabhängig.
`searchtype`	Sucht nach dem Typ eines PROSET-Wertes. Als Parameter wird nur der gesuchte Typ übergeben. Diese Funktion ist typspezifisch.

Tabelle 7: Beschreibung der Funktionen des Werkzeugs `dat`

Die zu realisierenden Benutzungsschnittstellen sollen wiederum den folgenden Aufrufkonventionen entsprechen:

Einzel-Funktions-Benutzungsschnittstelle:
`TOOLNAME FUNKTIONSNAME DOKUMENTTYP PARAMETER`

Bei einem Einzel-Funktions-Aufruf wird das Werkzeug (`TOOLNAME`) mit den folgenden Parametern aufgerufen. Der erste Parameter (`FUNKTIONSNAME`) ist der Name der Funktion, die ausgeführt werden soll. Der zweite Parameter (`DOKUMENTTYP`) gibt den Typ des Dokuments an, auf den die durch Parameter eins definierte Funktion angewendet werden soll.

Der dritte Parameter ist ein Zusatzparameter, mit dem der auszuführenden Funktion eine weitere Information übergeben wird, beispielsweise ein Dokumentname.

Multi-Funktions-Benutzungsschnittstelle:

`TOOLNAME`	**(Start des Werkzeugs)**
`> FUNKTIONSNAME DOKUMENTTYP PARAMETER`	**(Funktionsaufruf 1)**
…	
`> FUNKTIONSNAME DOKUMENTTYP PARAMETER`	**(Funktionsaufruf n)**
`> exit`	**(Abbruch des Werkzeugs)**

Ähnlich wie bei der vorangegangenen Schnittstelle, ist der Aufruf einer Funktion mit Hilfe der Multi-Funktions-Schnittstelle strukturiert. Der Werkzeugname wird nicht benötigt, da das Werkzeug bereits gestartet ist und auf eine Eingabe wartet. Die eigentlichen Parameter entsprechen denen der Einzel-Funktions-Schnittstelle. Der erste Parameter gibt den Namen der auszuführenden Funktion an, der zweite Parameter identifiziert den Dokumenttyp auf den die Funktion anzuwenden ist und der dritte Parameter enthält eine Zusatzinformation (Dokumentname).

Externe-Benutzungsschnittstelle:

```
TOOLNAME_MainFunc (int argc, char *argv) : tRetVal
```

Der Aufruf der Funktion `TOOLNAME_MainFunc` beinhaltet zwei Paramter. Der erste Parameter (`argc`) sagt aus wieviele Parameter in (`argv`) übergeben werden. Die Parameter in `argv` entsprechen den in den vorangegangenen Abschnitten beschriebenen. Der erste Parameter gibt den Namen der auszuführenden Funktion an, der zweite Parameter beschreibt den Dokumenttyp auf den die Funktion angewendet werden soll und der dritte Parameter liefert eine Zusatzinformation, wie beispielsweise den Dokumentnamen.

6.2.3 Implementation

Die Implementation wird auch beim Datenanalysetool mit Hilfe des Meta-Tools durchgeführt (siehe Kapitel 5). Im ersten Schritt wird mit dem Meta-Tool `ct` ein Basiswerkzeug mit dem Namen `dat` erzeugt. Der entsprechende Aufruf lautet:

```
ct /t dat
```

Durch den Aufruf werden die benötigten Verzeichnisse (siehe Abschnitt 5.2.1) sowie die für ein Basiswerkzeug notwendigen Dateien (siehe Abschnitt 5.2.2) erzeugt. Nach der Erzeugung des Basiswerkzeugs werden die für die verschiedenen Datentypen benötigten C-Moduln und Funktionen mit Hilfe des Meta-Tools `tm` erzeugt.

Die notwendigen Aufrufe werden im folgenden beschrieben:

1. Erzeugung eines C-Moduls für den Datentyp `integer`
   `tm /t dat /d integer`
2. Erzeugung eines C-Moduls für den Datentyp `real`
   `tm /t dat /d real`
3. Erzeugung eines C-Moduls für den Datentyp `string`
   `tm /t dat /d string`
4. Erzeugung eines C-Moduls für den Datentyp `boolean`
   `tm /t dat /d boolean`
5. Erzeugung eines C-Moduls für den Datentyp `set`
   `tm /t dat /d set`
6. Erzeugung eines C-Moduls für den Datentyp `tuple`
   `tm /t dat /d tuple`

Im zweiten Schritt werden die benötigten Funktionen mit Hilfe des Meta-Tools `tm` in die C-Moduln eingefügt. Die notwendigen Aufrufe werden im folgenden beschrieben:

1. Einfügen der Funktion `searchname`
   `tm /t dat /f searchname`
2. Einfügen der Funktion `searchtype`
   `tm /t dat /f searchtype`

Die Implementation der eigentlichen Funktionalität wird wiederum von Hand durchgeführt. Dabei werden auch hier im wesentlichen die von der ProSet-H-PCTE-Schnittstelle (siehe Kapitel 4) angebotenen Funktionen genutzt.

7 Zusammenfassung

In dieser Diplomarbeit wurde eine Schnittstelle zwischen PROSET und H-PCTE sowie verschiedene Werkzeuge zur Unterstützung einer Prototyping-Umgebung konstruiert.

Die aus PROSET und H-PCTE bestehende Umgebung unterstützt Prototyping als Vorgehensweise zur Softwareentwicklung. In diesem Rahmen wurde eine Schnittstelle konstruiert, die die Kommunikation zwischen der Programmiersprache und der Datenbank unterstützt.

Bei der Werkzeugentwicklung wurde zunächst eine Anforderungsanalyse durchgeführt. Diese ergab die Notwendigkeit eines konsistenten Konzeptes für alle innerhalb der Prototyping-Umgebung zu entwickelnden Werkzeuge.

Ein solches Konzept wurde in Form eines Meta-Tools realisiert. Das Meta-Tool übernimmt die Aufgabe eines Werkzeuggenerators. Dadurch wird eine einfache und flexible Werkzeugkonstruktion für die Prototyping-Umgebung gewährleistet. Um die Flexibilität des Meta-Tools zu unterstützen, werden zur Codegenerierung Templates verwendet. Zusätzlich können Makros verarbeitet werden, die eine individuelle Anpassung zu erzeugenden Sourcecodes erlauben.

Ein weiterer Aspekt der Generierung ist die Berücksichtigung der Modifizierbarkeit von bestehenden Werkzeugen, die zu einem früheren Zeitpunkt mit dem Werkzeuggenerator erzeugt worden sind. Ein zweites Meta-Tool erlaubt es dem Werkzeugentwickler, die von ihm generierten Werkzeuge zu einem späteren Zeitpunkt in ihrer Funktionalität zu erweitern. Die konkret zu entwickelnden Werkzeuge wurden mit dem beschriebenen Werkzeuggenerator erzeugt, wodurch die Funktionsfähigkeit desselben verifiziert werden konnte.

8 Ausblick

Im Bereich der Werkzeugkonstruktion sind noch viele Bereiche zur Vervollständigung der Prototyping-Umgebung abzudecken. Mögliche Werkzeuge aus diesem Bereich wurden bereits in Abschnitt 3 diskutiert.

- Ein Browser
  zur Betrachtung der Daten/Dokumente und Realisierung verschiedener Views.
- Werkzeuge zur Realisierung von Schutzkonzepten
  um Schutzattributzuweisung an bestimmte Dokumente oder Dokumentbereiche vornehmen zu können.
- Systeminstallations-/Pflegewerkzeuge
  zur Installation des Systems und der späteren Anpassung der Konfiguration an die aktuellen Bedürfnisse.
- Ein Archivierungswerkzeug
  zur Datensicherung und selektiven Wiederherstellung beschädigter Daten und Dokumente.
- Ein Reportgenerator
  zur Erstellung von Listen und Auswertungen, die die Analyse des Datenbestandes unterstützen.
- Ein Reparaturwerkzeug
  zur Reparatur von Defekten innerhalb des Systems, die sich durch Wiederherstellung von Daten nicht beheben lassen.
- Optimierungs-/Monitoring-Werkzeug
  zur Kontrolle des Systems und zur Auswertung der Ressourcennutzung sowie der Modifikation des Ressourcenzugriffs zwecks Optimierung.
- Benutzerverwaltungs-Werkzeug
  zur Registration und Löschung von Benutzern sowie der Zuweisung von Zugriffsrechten auf Ressourcen.

Einen weiteren wichtigen Ansatzpunkt bietet das Datenanlysewerkzeug. Um die Analysefähigkeit voll auszubauen, kann eine Abfragesprache in das Werkzeug integriert werden. Eine mögliche Alternative ist P-OQL [Hen94]. P-OQL ist eine Abfragesprache, die speziell für die Datenbank PCTE entwickelt wurde und damit einen Großteil der benötigten Funktionalität beherrscht, der für die Prototyping-Umgebung notwendig ist.

Die Benutzungsschnittstellen können erweitert werden, indem eine graphische Benutzungsoberfläche implementiert wird. Die Funktionälität der Werkzeuge kann über die Schnittstelle für externe Anwendungen genutzt werden. Die graphische Oberfläche könnte auf diese Weise, d.h. durch einbinden mehrerer Werkzeugmodule, die Funktionen verschiedener Werkzeug integrieren.

9 Literatur

[Boe76] B. W. Boehm. Software Engineering. IEEE Transactions on Computers. Vol. 25, No. 12, Dec (1976).

[Boe88] B. W. Boehm. A Spiral Model of Software Development and Enhancement. IEEE Computer. Vol. 21, No. 5, May (1988).

[BP92] W. Bischofberger, G. Pomberger. Prototypingoriented Software Developement: Concepts and Tools. Springer-Verlag (1992).

[DF89] E.-E. Doberkat, D. Fox. Software Prototyping mit SETL. B.G. Teubner, Stuttgart, (1989).

[DFG92] E.-E. Doberkat, W, Franke, U. Gutenbeil, W. Hasselbring, U. Lammers, C. Pahl. PROSET - Prototyping with Sets: Language Definition. Universität-GH-Essen. Informatik-Bericht 02/92, (1992).

[DFG92] E.-E. Doberkat, W. Franke, U. Gutenbeil, W. Hasselbring, U. Lammers, C. Pahl. A First Implementation of PROSET. Universität Essen (1992).

[DFK93] E.-E. Doberkat, W. Franke, U. Kelter, W. Seelbach. Verwaltung persistenter Daten in einer Prototyping-Umgebung, in: Requirements Engineering ’93: Prototyping. B. G. Teubner Stuttgart (1993).

[ECM93] European Computer Manufacturers Association. Standard ECMA-149. Portable Common Tool Environment - Abstract Specification. Juni (1993).

[ECM93] European Computer Manufacturers Association. Standard ECMA-158. Portable Common Tool Environment - C Programming Language Binding. Juni (1993).

[Gel85] D. Gelernter. Generative Communication in Linda. ACM Transactions on Programming Languages and Systems, 7(1):80-112, (1985).

[Hal85] R. H. Halstead. Multilisp: A language for current symbolic computation. ACM Transactions on Programming Languages and Systems, 7(4):501-538, (1985).

[Hen94] A. Henrich. P-OQL: an OQL-oriented Query Language for PCTE. Universität Siegen, Praktische Informatik, Fachbereich Elektrotechnik und Informatik.

[Heu92] A. Heuer. Objektorientierte Datenbanken: Konzepte, Modelle, Systeme. Addison-Wessley (1992).

[HS91] A. Heuer, M. H. Scholl. Principles of object-oriented query language. In Proc. GI-Fachtagung „Datenbanksysteme für Büro, Technik und Wissenschaft. Springer, Informatik Fachbericht 270: 178-197. Kaiserslautern, Germany (1991).

[HS93] E. Horn, W. Schubert. Objektorientierte Software-Konstruktion: Grundlagen, Modelle, Methoden, Beispiele. Carl Hanser Verlag München (1993).

[Iso87] R. Ison. Software Engineering Environments: An Experimantal Ada Programming Support Environment in the HP CASEdge Integration Framework. Springer (1987)

[Kap95] C. Kappert. Integration von Persistenzkonzepten in eine Prototypingsprache und Realisierung mit Hilfe eines Nicht-Standard-Datenbanksystems. Diplomarbeit an der Universität Dortmund (1995).

[Kel91] U. Kelter. Einführung in H-PCTE. Praktische Informatik V, Fachbereich Mathematik und Informatik, FernUniversität Hagen (1991).

[Kel92] U. Kelter. H-PCTE - A High-Performance Object Management System for System Developement Environments. In Proc. 16th Annual International Computer Software and Applications Conference (COMPSAC '92). Chicago, Illinois, USA. September (1992).

[Kel93] U. Kelter. Integrationsrahmen für Software-Entwicklungsumgebungen. Informatik-Spektrum (1993) 16: 281-285.

[Kel93] U. Kelter. An Information Retrieval Common Service Based on H-PCTE. In Proc. 6th Software Engineering Environments (SEE '93). Reading, UK. Juli (1993).

[KR83] B. Kernigan, D. M. Ritchie. Programmieren in C. Carl Hanser Verlag München Wien (1983).

[Nag93] M. Nagl. Software-Entwicklungsumgebungen: Einordnung und zukünftige Entwicklungslinien. In Informatik-Spektrum (1993).

[Ost86] L. Osterweil. Software Process Interpretation and Software Environments. University of Colorado at Boulder. Technical Report CU-CS-324-86. (1986).

[Ost87] L. Osterweil. Software Processes are Software too. Proceedings of the 9th International Conference on Software Engineering. (1987).

10 Abbildungsverzeichnis

11 Tabellenverzeichnis

Anhang A
Generieren eines Werkzeugs mit dem Meta-Tool

In diesem Abschnitt soll die Generierung eines Werkzeugs mit Hilfe des in Kapitel 5 beschriebenen Meta-Tools erläutert werden. Es wird das Vorgehen, in den Schritten Planung, Codegenerierung mit dem Meta-Tool und Implementation, beschrieben. Parallel zur abstrakten Beschreibung des Vorgehens, wird die praktische Ausführung anhand eines Beispiels demonstriert. Alle Textteile, die sich auf das Beispiel beziehen, sind *kursiv* gedruckt.

1. Planung

In dieser Phase müssen vier Vorgaben in bezug auf das zu konstruierende Werkzeug definiert werden.

1. Der Name für das neue Werkzeug muß festgelegt werden.
2. Die Dokumenttypen, mit denen das Werkzeug arbeiteten soll, müssen festgelegt werden.
3. Die Funktionen, die das Werkzeug für die Dokumenttypen zur Verfügung stellen soll, müssen festgelegt werden.
4. Die Funktionalität der Funktionen des Werkzeugs muß formuliert werden.

In unserem Beispiel soll ein Werkzeug entwickelt werden, das es ermöglicht neue Dokumente vom Typ `INTEGER` und `STRING`, zu erzeugen und zu löschen. Die Vorgaben werden wie folgt festgelegt:

1. *Der Name für das neue Werkzeug lautet `INTSTR`.*
2. *Die Dokumenttypen, mit denen das Werkzeug arbeiten soll, sind:*
   `INTEGER, STRING`.
3. *Die Funktionen, die das Werkzeug zur Verfügung stellen soll, sind:*
   `CREATE`, `DELETE`.

2. Codegenerierung mit dem Meta-Tool

Nachdem der Name und die zu implementierende Funktionalität festgelegt worden sind, sollen im zweiten Schritt automatisch die notwendigen Verzeichnisse und Source-Code-Dateien erzeugt werden. Zuerst wird mit dem Meta-Tool `ct` ein Basiswerkzeug generiert. Dabei legt das `ct` die notwendige Verzeichnisstruktur an.

Das Meta-Tool erzeugt weiterhin die Source-Code-Dateien mit den Funktions-Rümpfen, wie bereits in Abschnitt 5.2.2 beschrieben. Der Aufruf des Werkzeugs sieht wie folgt aus:

```
ct /t <TOOLNAME>
```

In unserem Beispiel muß das Meta-Tool also `ct` wie folgt aufgerufen werden:

```
ct /t INTSTR
```

Daraufhin wird durch ct die folgende Verzeichnisstruktur generiert:

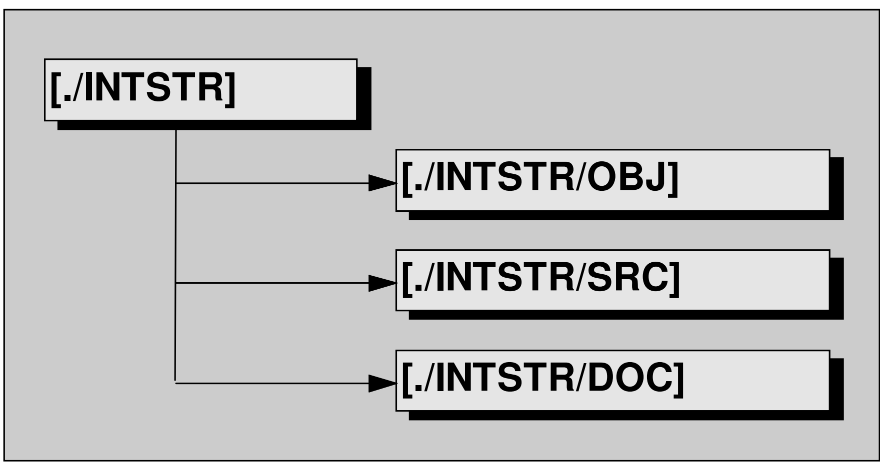

Abbildung 33: Für das Werkzeug `INTSTR` generierte Verzeichnisse

In den in der Abbildung 33 zu sehenden Verzeichnissen, werden die folgenden Source-Code-Dateien erzeugt:

```
INTSTR/TOOL1.make          INTSTR/SRC/standard.h
INTSTR/SRC/INTSTR.c        INTSTR/SRC/standard.c
```

INTSTR/SRC/INTSTRmf.h *INTSTR/SRC/all.h*
INTSTR/SRC/INTSTRmf.c *INTSTR/SRC/all.c*
INTSTR/SRC/retval.h

Nachdem das Basiswerkzeug `INTSTR` generiert worden ist, soll der generierte Source-Code automatisch um die in der Planung formulierten Dokumenttypen und Funktions-Rümpfe ergänzt werden. Dazu muß mit dem Meta-Tool tm für jeden neuen Dokumenttyp ein C-Modul erzeugt werden. Dieses dokumentspezifische C-Modul soll später die Funktionen, mit denen dieser Dokumenttyp bearbeitet werden soll, enthalten. Der Aufruf von `tm` zur Modulgenerierung sieht wie folgt aus:

```
tm /t <TOOLNAME> /d <DOKUMENTTYP>
```

In unserem Beispiel müssen C-Moduln für die Dokumenttypen INTEGER und STRING erzeugt werden. Die notwendigen Aufrufe lauten:

```
tm /t INTSTR /d INTEGER
tm /t INTSTR /d STRING
```

Durch diese Aufrufe werden in den bestehenden Verzeichnissen vier zusätzliche Source-Code-Dateien generiert:

INTSTR/SRC/INTEGER.h *INTSTR/SRC/STRING.h*
INTSTR/SRC/INTEGER.c *INTSTR/SRC/STRING.c*

Nachdem alle Moduln erzeugt worden sind, müssen die Funktionsrümpfe in die Moduln eingefügt werden. Zu diesem Zweck wird wiederum das Meta-Tool `tm` genutzt. Jede Funktion muß einzeln mit dem Meta-Tool tm generiert werden. Durch einen Aufruf vom `tm` wird die gewünschte Funktion in alle vorhandenen Dokumenttyp-Moduln eingetragen. Ein solcher Aufruf sieht wie folgt aus:

```
tm /t <TOOLNAME> /f <FUNCTIONNAME>
```

Für unser Beispiel müssen wir zwei Funktions-Rümpfe generieren. Die Funktion CREATE und die Funktion DELETE. Die nowendigen Aufrufe sehen wie folgt aus:

```
tm /t INTSTR /f CREATE
tm /t INTSTR /f DELETE
```

Durch diese Aufrufe des Meta-Tools `tm` *werden die Funktionen CREATEINTEGER und DELETEINTEGER in den bestehenden C-Moduln INTSTR/SRC/INTEGER.H und INTSTR/SRC/INTEGER.C und die Funktionen CREATESTRING und DELETESTRING in den bestehenden C-Moduln INTSTR/SRC/STRING.H und INTSTR/SRC/STRING.C generiert.*

Wenn alle C-Moduln und Funktionen erzeugt worden sind, ist es die Aufgabe des Werkzeugentwicklers, die erzeugten Funktionsrümpfe in den C-Moduln mit der gewünschten Funktionalität zu füllen. Nachdem dies geschehen ist, kann der nun vollständige Source-Code mit Hilfe der Datei <TOOLNAME>.make und dem Make-Utility von UNIX übersetzt werden. Danach steht ein funktionsfähiges Werkzeug zur Verfügung. Der Aufruf des Make-Utilities sieht wie folgt aus:

```
make -f <TOOLNAME>.make
```

In unserem Beispiel müssen die Funktionen CREATEINTEGER, DELETEINTEGER, CREATESTRING und DELETEINTEGER implementiert werden. Danach kann das Werkzeug übersetzt werden.Der Aufruf des Make-Utilities sieht wie folgt aus:

```
make -f INTSTR.make
```

Nach der Übersetzung der Werkzeuge, können diese mit Hilfe der oben beschriebenen Benutzungsschnittstellen genutzt werden.