# Entwicklung einer Komponentenbeschreibungssprache zur Unterstützung der Wiederverwendung von Software–Komponenten

Chantal Keller

UNIVERSITÄT - GESAMTHOCHSCHULE SIEGEN

Fachbereich 12 Elektrotechnik und Informatik

# Entwicklung einer Komponentenbeschreibungssprache zur Unterstützung der Wiederverwendung von Software–Komponenten

angefertigt von
Chantal Keller

Betreuer:
Prof. Dr. Hartmut Ring – Fachbereich Mathematik
Prof. Dr. Udo Kelter – Fachbereich Elektrotechnik und Informatik

Tag der Ausgabe: 01.11.1999
Tag der Abgabe: 30.04.2000

# Danksagung

An dieser Stelle möchte ich allen Mitarbeitern der Fachgruppe Praktische Informatik des Fachbereichs Informatik und Elektrotechnik an der Universität Siegen herzlich für ihre moralische und fachliche Unterstützung bei der Erstellung dieser Arbeit danken. Ganz besonders danke ich meinem Betreuer André Berten für seine Hilfe und seine Bereitschaft, Fragen aller Art zu beantworten.

Chantal Keller

**Zusammenfassung**

Diese Arbeit beschäftigt sich mit der Entwicklung eines Beschreibungsmodells für Software–Komponenten sowie eines Prototypen zur Verwaltung dieser Beschreibungen unter dem Datenhaltungssystem H-PCTE. Nach der Analyse verschiedener Komponentenarten werden Datenbankschemata zu ihrer Beschreibung erstellt. Es werden Entwurf und Implementierung des Prototypen, welcher Verwaltungs– und Suchfunktionen auf dem Datenbestand realisiert, unter *Java* geschildert. Eine Evaluierung der Ergebnisse und ein Ausblick auf zukünftige Entwicklungen in diesem Bereich schließen die Arbeit ab.

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Kapitel 1

# Einleitung und Motivation

## 1.1 Wiederverwendung von Software

Heutige Software muß, besonders im kommerziellen Bereich, viele Anforderungen erfüllen: Korrektheit, Effizienz, Erweiterbarkeit, leichte Wartbarkeit und Transparenz bei größtmöglicher Konfigurierbarkeit durch den Anwender, um nur einige zu nennen. Vor allem bei großen, komplexen Software-Systemen sind diese Vorgaben nicht leicht zu erfüllen. *Wiederverwendung* (engl. *software reuse*) bereits erstellter und erprobter Software erleichtert die Entwicklung solcher Systeme. Dabei kommt es allerdings darauf an, in welchem Sinne Wiederverwendung betrieben wird.

Seit jeher ist Wiederverwendung im allgemeinsten Sinne ein integraler Bestandteil jeglicher Software–Entwicklung, denn nur sehr einfache Programme lassen sich von Grund auf, d.h. ohne Nutzung bereits vorhandener Ressourcen, in einem vertretbaren Zeit- und Aufwandsrahmen realisieren.

Im Laufe der Zeit haben sich allerdings Art und Umfang der Wiederverwendung gewandelt. Dadurch veränderte sich ihr Stellenwert im Software–Entwicklungs–Zyklus. Früher blieb es meist dem einzelnen Programmierer überlassen, ob und in welchem Umfang er Programmcode wiederverwenden wollte. Eine systematische Einbindung in die Software–Entwicklung fand nicht statt, und das Kriterium der *Wiederverwendbarkeit* spielte bei neu erstellter Software keine oder kaum eine Rolle.

Mit der Verbreitung modularer bzw. objektorientierter Programmiersprachen und den dazugehörigen Komponentenbibliotheken hat die Bedeutung einer gezielten, im Software–Zyklus festgeschriebenen Wiederverwendung zugenommen. Die Wiederverwendbarkeit als Anforderung an die Software, aber auch an andere Produkte des Software–Zyklus, hat einen viel größeren Stellenwert erhalten. Doch entstehen durch die Einbindung dieser Konzepte und Anforderungen neue Probleme, welche es zu lösen gilt. Erst dann kann sich ein signifikanter Zeit– und Arbeitsgewinn durch Wiederverwendung bemerkbar machen.

### 1.1.1 Arten von Wiederverwendung

**„Cut & Paste“** Dies ist die simpelste Art der Wiederverwendung: Der Programmierer kopiert Ausschnitte aus existierendem eigenen oder fremden Code, um sie nach eventueller Anpassung in einem neuen Programmkontext einzusetzen.

**Funktionsbibliotheken** Sammlungen von nützlichen Funktionen, die z.B. die Bereiche Mathematik, Graphikausgabe und die Erstellung visueller Benutzungselemente umfassen, existieren für alle wichtigen höheren Programmiersprachen und werden ausgiebig eingesetzt[1]. Funktionsbibliotheken sind im Vergleich zu „Cut & Paste“ universeller einsetzbar, da der Benutzer den zugrundeliegenden Code nicht mehr kennen und verstehen muß. Er muß nur die *Schnittstelle*, den Funktionsaufruf mit seinen Parametern, beherrschen.

**Klassenbibliotheken** Mit der Entwicklung objektorientierter Techniken entstand eine weitere Abstraktions- und Anwendungsebene: Klassen sind opake Gebilde, welche nur über ihre definierte Schnittstelle ansprechbar sind, und in ihrem Inneren beliebig komplexes Verhalten verbergen können. Gut konzipierte Klassen verbergen möglichst viel von der Implementierung ihrer Funktionalität vor dem Anwender, und geben ihm gleichzeitig durch eine verständliche, vielseitige Schnittstelle die Möglichkeit, die Klasse ohne großen Aufwand in seine eigene Programmierung zu integrieren.

**Frameworks** Ein Framework wird definiert als

> „an integrated set of reusable and extensible software artifacts for a specific application domain“ [EMT98] (p.124).

Es stellt also einen Rahmen für die Software–Entwicklung, sowie dem Anwendungsbereich angepaßte Komponenten zur Verfügung, die entweder bereits fertig vorliegen (*black-box customization*), oder generischen Charakter besitzen und gemäß der vorliegenden Aufgabenstellung spezialisiert werden (*white-box customization*). Diese Komponenten sind in die Rahmenstruktur eingefügt.

**Nicht–ausführbare Komponenten** Auch andere Komponenten, die im Rahmen der Software–Entwicklung anfallen, können wiederverwendet werden, z.B. Entwurfsmuster, Architekturen, Testfälle, Dokumentation.

Jede weitere Stufe der Wiederverwendung gibt dem Anwender also neue Möglichkeiten einer schnelleren und einfacheren Software–Entwicklung, doch

[1] Z.B. erfordert schon ein einfaches C–Programm, welches eine Bildschirmausgabe erzeugen soll, das Einbinden der Bibliotheks–Header–Datei „stdio.h“.

kann der Einarbeitungs- und Suchaufwand diese Vorteile wieder zunichte machen.

### 1.1.2 Komponentenbasierte Software–Entwicklung

Eine *Komponente* kann definiert werden als

> „an executable asset that may be integrated as–is into an application“ [EMT98] (p.26).

Der Begriff *Asset* wird hier folgendermaßen erklärt:

> „Reusable assets are composed of a collection of related software work products that may be reused from one application to another“ [EMT98] (p.26). „[They] may include such things as requirements, project plans, estimates, architectures, designs, user interfaces, test plans, test cases, data, quality plans, and documentation“ [EMT98] (p.6).

Von Girardi *et al.* wird eine Komponente folgendermaßen beschrieben:

> „Reusable software components include not only generic code components (basic procedures, modules, megamodules), but also systems, subsystems, software specifications (requirements and design specifications), architectures, test cases, etc.“ [RG95] (p. 31).

Im Rahmen dieser Arbeit wird sich der Begriff *Komponente* auf ein eigenständiges Element der Software–Entwicklung beziehen, welches ausführbar sein kann, aber nicht muß. Ganz speziell werden neben ausführbarem Code Entwurfsmuster (engl. *design patterns*) als Komponenten angesehen.

Komponenten sind insofern eigenständig, als sie mehr noch als die Klassen der objektorientierten Entwicklung komplexes Verhalten kapseln, und mittels einer einfachen, oft standardisierten Schnittstelle nach außen vermitteln. Beispiele für komponentenbasierte Technologien sind Micosofts *Active X* und Suns *Java Beans*.

Komponentenbasierte Software–Entwicklung (engl. *component–based software development*) bietet also noch mächtigere Strategien zur schnellen und effizienten Entwicklung von Software. Doch gelten für sie auch wegen der Größe und Komplexität der verwendeten Komponenten in hohem Maße die in Abschnitt 1.1.3 geschilderten Probleme.

### 1.1.3 Probleme der Wiederverwendung

Immer mehr Firmen und Institutionen sind sich heute der Anforderungen bewußt, welche die Integration von Wiederverwendungskonzepten an die

Software–Entwicklung stellt; meist sind Umorganisation von Arbeitsprozessen, Schulungen der Mitarbeiter, sowie das Anlegen eigener Komponentenbibliotheken erforderlich. Die Pflege dieser Bibliotheken, sowie die Verbreitung von Information über ihre Inhalte benötigt ebenfalls Zeit.

Ein Schlüsselproblem ist dabei, wie schon erwähnt, die **Informationsbeschaffung**: Komponentenbibliotheken sind nicht statisch. Vielmehr erweitert sich mit der Entwicklung neuer Komponenten, die die Anforderung der Wiederverwendbarkeit erfüllen und somit in die Bibliothek aufgenommen werden, ihr Umfang erheblich. Große Bibliotheken können viele tausend Komponenten enthalten.

Diese Bibliotheken nach Komponenten zu durchsuchen, welche für die Bewältigung einer bestimmten Aufgabe geeignet sind, ist kein trivialer Akt, sondern erfordert vielmehr vom Programmierer großen Zeit- und Arbeitsaufwand – Aufwand, der besser für die Software–Entwicklung eingesetzt würde. **Schlechte** oder **uneinheitliche Dokumentation** der vorhandenen Komponenten vergrößert diesen Aufwand u.U. noch mehr.

Ein weiteres Problem ist die **Protokollierung von Erfahrung**: Es wäre sinnvoll, Erfahrungen, die der Programmierer mit wiederverwendbaren Komponenten z.B. bzgl. ihrer Nützlichkeit und Zuverlässigkeit gemacht hat, an andere Anwender weiterzugeben. Auf diese Weise könnten Komponenten einfacher evaluiert und der Suchprozess verkürzt werden.

### 1.1.4 Lösungsstrategien

Durch die in Abschnitt 1.1.3 geschilderte Problematik sind bereits die Anforderungen an ein Werkzeug umrissen, welches Abhilfe zu den erwähnten Mißständen schaffen kann. Software–Komponenten sollen:

- leichter zugänglich,
- einheitlich dokumentiert und umfassend einzusehen,
- leichter zu evaluieren sein,
- und ihre Verwendung samt der Ergebnisse soll protokollierbar sein.

## 1.2 Gliederung der Arbeit

Im zweiten Kapitel der Arbeit wird die Aufgabenstellung im Kontext der bereits vorhandenen Ansätze entwickelt werden. Das dritte Kapitel schildert die softwaretechnischen Grundlagen, auf denen diese Arbeit aufbaut. Das vierte Kapitel ist dem Entwurf sowohl der grundlegenden Datenorganisation als auch des gemäß Aufgabenstellung zu entwickelnden Software–Systems (mit Namen *CoLibry*, s. Abschnitt 2.3) gewidmet. Kapitel fünf wiederum befaßt sich mit der Realisierung dieser Entwürfe. Schließlich bietet Kapitel sechs eine Zusammenfassung der Arbeit und einen Ausblick auf weitere Aufgaben beim Ausbau von *CoLibry*.

# Kapitel 2

# Aufgabenstellung

In diesem Kapitel werden zunächst die Anforderungen an diese Arbeit skizziert. Einige ausgewählte Ansätze aus der Literatur werden daraufhin vorgestellt, und gegen die Anforderungen abgegrenzt. Die Angabe der konkreten Aufgabenstellung, welche im Rahmen dieser Arbeit zu realisieren ist, schließt das Kapitel ab.

## 2.1 Aufgaben einer Komponentenbibliothek

In dieser Arbeit wird davon ausgegangen, daß nicht die Komponenten selber, sondern nur ihre Beschreibungen verwaltet werden (s. Abschnitt 4.1). Ausgangspunkt ist also eine Bibliothek von Komponentenbeschreibungen. Für die Beschreibungen ist ein geeignetes Datenmodell zu entwickeln, eine *Komponentenbeschreibungssprache* also, anhand derer die Komponenten gespeichert und auch wieder aufgefunden werden können.

Um den Suchvorgang und die anschließende Evaluierung der gefundenen Komponenten zu optimieren, muß die Beschreibung umfangreich, und Erfahrungen von Benutzern der Komponente müssen ebenfalls gespeichert und zugreifbar sein. Auch statistische Daten wie Anzahl der Benutzungen sollen erfaßt werden.

Für den Zugriff auf die Komponentenbibliothek und die Verwaltung der Einträge ist eine benutzerfreundliche Benutzungsschnittstelle notwendig. Die Schnittstelle sollte die folgenden Funktionen anbieten:

- Anlegen von neuen Beschreibungen; wahlweise per Hand, oder durch Importieren aus einem vorhandenen Datenbestand;
- Modifizieren und Löschen vorhandener Beschreibungen;
- Versionsmanagement;
- Erzeugen und Löschen von Komponentenkategorien, und Zuordnen der Komponenten zu einer oder mehreren Kategorien;

- Anzeige aller vorhandenen Beschreibungen in Listen– und Baumstruktur mit Möglichkeit zum Browsing, d.h. zur Navigation durch die Strukturen;
- Sortier– und Filterfunktionen für diese Anzeige;
- eine angemessene Anzeige einer einzelnen Beschreibung, sowie des Protokolls ihrer Benutzungen;
- eine Anzeige von Statistiken über die Bibliothek als ganzes, sowie zu einer einzelnen Komponente;
- verschiedenartige Suchfunktionen, wie Schlüsselwortsuche, Facettensuche (zu Facetten s. auch Abschnitt 2.2.1), und Expertensuche mittels einer Anfragesprache.

Zweckmäßigerweise gibt es verschiedene Benutzergruppen, die unterschiedliche Zugriffsrechten auf die Komponenten besitzen:

- den normalen Benutzer, der Komponenten suchen, anzeigen, und auf Wunsch einen Eintrag über seine Erfahrungen in das Benutzungslog einer Komponente machen darf;
- den Manager der Komponentenbibliothek, welcher Beschreibungen anlegen, modifizieren, und löschen darf.

## 2.2 Vorhandene Ansätze in der Literatur

Diese Arbeit ist nicht die erste, welche sich mit der Problematik der Unterstützung von komponentenbasierter Software–Entwicklung, und speziell der Suche in Komponentenbibliotheken, beschäftigt. Die in der Literatur beschriebenen Lösungsvorschläge haben meist den Wunsch nach besserem und effizienterem Zugriff auf solche Bibliotheken gemeinsam.

### 2.2.1 Natürlichsprachige Spezifikation von Komponenten

M. R. Girardi and B. Ibrahim beschreiben in ihren Artikeln (s. [RGI93a] und [RGI93b]) den Ansatz einer *natürlichsprachigen Spezifikation* von Software–Komponenten.

Ihre Prämisse ist, daß Benutzer von Komponentenbibliotheken Komponenten einfach und effektiv finden wollen, d.h. der Suchaufwand sollte gering sein, und das Suchergebnis bereits so gut, daß aus der Liste gefundener Komponenten nun ohne großen weiteren Aufwand die gewünschte extrahiert werden kann.

Für diese Zwecke halten sie die bisherigen Suchmethoden auf Bibliotheken (Browsing und Schlüsselwortsuche) für nicht sehr gut geeignet. Die

Autoren setzen daher als Ergänzung dieser Methoden die natürlichsprachige Spezifikation ein, sowohl bei der Suche als auch bereits bei der Kategorisierung der Komponenten zwecks Erfassung für die Bibliothek.

Mittels sogenannter *Rahmen* (engl. *case frames*) wird semantische Information aus natürlichsprachigen Funktionsbeschreibungen gewonnen. Ein Rahmen enthält *thematische Rollen* (auch als *Facetten* bekannt), das sind solche Inhaltseigenschaften von natürlichsprachigem Text wie Agent (die Komponente, welche eine Aktion implementiert), Aktion, Objekt (das durch die Aktion beeinflußte Ziel), Ort (der Aktion) und Zweck (der Aktion).

Automatisches Parsen nach Wörtern, welche thematische Rollen beschreiben (z.B. bezeichnet das Verb mit großer Wahrscheinlichkeit die thematische Rolle „Aktion“), führen unter Zuhilfenahme von Heuristiken zu einem oder mehreren Rahmen, welche plausible Interpretationen des geparsten Textes repräsentieren. Diese Auswertung von Text soll sowohl bei der Beschreibung der Komponenten, als auch bei Suchanfragen eingesetzt werden.

Die beschriebene Methode soll die rein lexikalischen ergänzen, und damit Benutzerfreundlichkeit (der Benutzer muß Beschreibungen und Anfragen nicht formal angeben), und Effizienz (Suchergebnisse passen besser zur Anfrage) erhöhen.

Nachteile der Methode sind die Ungenauigkeit und Vieldeutigkeit von natürlicher Sprache. Ein Parser, welcher zuverlässig semantische Inhalte aus beliebigem Text extrahieren kann, ist sehr schwer zu realisieren.

### 2.2.2 Einsatz formaler Methoden

In J. Jengs und B.H.C. Chengs Artikel [JC92] werden Software–Komponenten durch formale Spezifikationen (in EBNF–Form notiert) dargestellt. Die Betonung liegt dabei auf Funktionen. Der Grad der Generalisierung einer Komponente und ihre Ähnlichkeit zu anderen Komponenten werden berechnet. Durch Subsumptions– und Clusteringalgorithmen, die auf die generellsten Komponenten angewendet werden, wird aus der Sammlung von Komponenten eine Hierarchie gebildet. Eine Suche findet ebenfalls nur anhand formaler Spezifikationen statt.

In [JC93] werden Komponenten mittels sog. *Analogical matching*, also ihrer Ähnlichkeit zueinander gesucht. In [JC95a] kommt das sog. *Specification matching* hinzu; Komponenten werden dabei mittels Prädikatenlogik spezifiziert, und können sowohl durch exakte als auch durch ungenaue Anfragen ermittelt werden. In [JC95b] wird darüber hinaus beschrieben, wie durch Vergleich einer gefundenen Komponente mit der Suchanfrage nötige Modifizierungen an der Komponente ermittelt werden können.

Dieser Ansatz beschränkt sich notwendigerweise auf einige wenige, einfache Komponentensorten wie z.B. Funktionen, da der Aufwand der formalen Spezifikation für umfangreichere Sorten zu groß wäre. Der Ansatz der formalen Spezifikation ist daher nicht sinnvoll für diese Arbeit, in der verschiedenartige, komplexe Komponentensorten beschrieben werden müssen.

### 2.2.3 Virtuelle, verteilte Software–Bibliotheken

Die Autoren betrachten in [BDH96], [BDG95] virtuelle, verteilte Softwarebibliotheken. Virtuell bedeutet, daß eine Bibliothek nur die Katalogeinträge, nicht die Komponenten selber, enthält; verteilt, daß sich die Einträge auf mehrere Bibliotheken an eventuell unterschiedlichen Orten aufteilen.

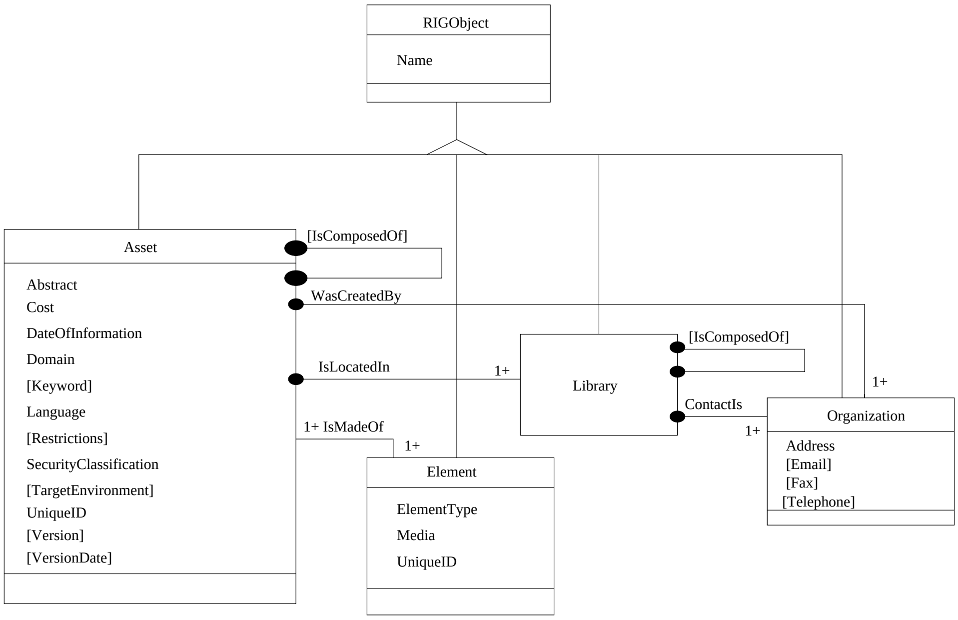


Abbildung 2.1: Basic Interoperability Data Model (aus [BDH96])

Eine einheitliche Beschreibungsschnittstelle für Komponenten, das *Basic Interoperability Data Model (BIDM)* sorgt für die Anbindung der verschiedenen Bibliotheken, die nach innen ihre eigenen Daten– und Suchformate weiterbehalten. Es werden Übertragungen des abstrakten BIDM nach HTML und SGML vorgestellt.

Um sowohl die Suche nach einer speziellen Komponente zu unterstützen, als auch zu gewährleisten, daß der Benutzer bei Anfragen immer die neueste Version einer Komponente erhält, werden zwei Arten von Namen vorgesehen:

Der *Location Independent File Name (LIFN)* bezieht sich immer auf ein Objekt, und die Bindung kann nicht verändert werden. Der *Uniform Resource Name (URN)* dagegen kann im Laufe seines Lebens auf verschiedene Objekte verweisen. Bei Publikation einer Komponente werden nun sowohl

LIFN als auch URN veröffentlicht. URN–Server liefern dann bei Anfrage den zu einem URN assoziierten LIFN zurück, LIFN–Server wiederum zu einem LIFN eine Liste von URLs.

Besonders relevant für diese Arbeit ist das BIDM, das Datenmodell für Komponenten. Eine Komponente (hier *asset* genannt) wird durch Informationen über ihren Inhalt, über eventuelle Elemente, aus denen sie sich zusammensetzt, und über die Organisation, welche die Komponente entwickelt hat, beschrieben (s. Abbildung 2.1). In dieser Arbeit wird ein ähnlicher Ansatz zur Beschreibung verwandt werden.

### 2.2.4 Suche in großen Klassenbibliotheken

Simon Gibbs beschreibt in [G90] ein objekt-orientiertes Datenmodell namens *Xos*, welches zur logischen Organisation und Beschreibung von Klassenbibliotheken benutzt werden kann. Eine SQL–ähnliche Anfragesprache, die boolsche sowie ungenaue (engl. *fuzzy*) Suchprädikate im Bereich von 0 bis 1 zuläßt, ist enthalten.

Am Beispiel einer C++–Klassenbibliothek wird demonstriert, wie man Klassen, Attribute, und Operationen der C++–Welt auf die gleichnamigen Elemente von *Xos* abbilden kann. Klassen können dabei durch Kanten miteinander verbunden werden, die verschiedene Relationen darstellen. Durch die enthaltene Anfragesprache können recht komplexe Abfragen formuliert werden.

Verfeinerungen des Datenmodells, die solche Konzepte wie Versionsmanagement, Projektkontexte, und Authorisierung umfassen, stehen jedoch nach Angaben des Autors noch aus.

Dieser Ansatz ähnelt dem in dieser Arbeit gewählten durch die Vorgaben einer objektorientierten Datenspeicherung der Kompontenten, und der Integration einer mengenorientierten Abfragesprache (s. Abschnitte 3.2 bzw. 3.3). Durch die Ausgereiftheit und Mächtigkeit der in dieser Arbeit zur Verfügung stehenden Werkzeuge ist allerdings schon eine im Vergleich zu *Xos* erweiterte Funktionalität fest verankert.

### 2.2.5 Praktische Wiederverwendung von Software

In ihrem Buch [EMT98] geben M. Ezran *et al.* einen Überblick über praktikable Methoden, Wiederverwendung in den Software–Zyklus zu integrieren. Sie betrachten die Problematik und Lösungsstrategien sowohl aus der Sicht des Programmierers, als auch der des Managers und Geschäftsmanns. Umfangreiche Erfahrungsberichte von verschiedenen Firmen der Software–Branche, in denen die angewendeten Wiederverwendungsmaßnahmen und die daraus resultierenden Ergebnisse gegenübergestellt und diskutiert werden, runden den Überblick ab.

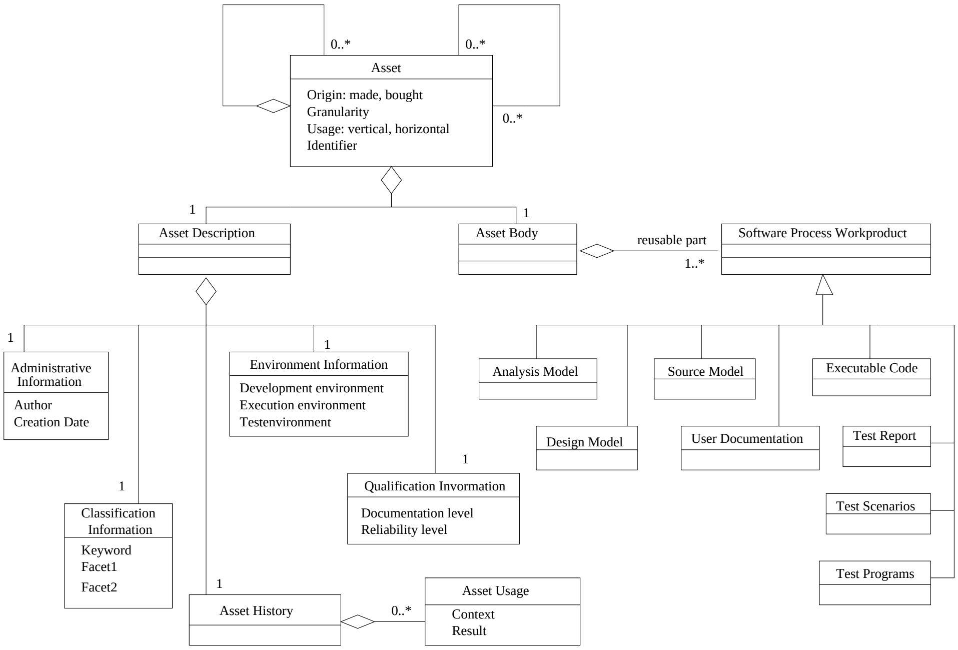


Abbildung 2.2: Das *Asset*–Modell (aus [EMT98])

Diese Arbeit erhielt vor allem durch folgende im Buch besprochenen Bereiche Impulse:

- Beschreibungsmöglichkeiten einzelner Komponenten ([EMT98], Kapitel 2)
- Komponenten–Bibliotheken und ihre Funktionen ([EMT98], Kapitel 3)
- Wiederverwendungs–Evaluierung durch Messen verschiedener Größen (engl. *reuse metrics*), wie z.B. Kostenaufwand zur Erstellung wiederverwendbarer Komponenten, Qualität, und Zugriffshäufigkeit auf die Bibliothek ([EMT98], Kapitel 6).

Vor allem das vorgestellte Datenmodell für Komponenten (s. Abbildung 2.2) wurde als Grundlage für eigene Überlegungen genutzt.

## 2.3 Aufgabenstellung

Die am Anfang dieses Kapitels aufgelisteten Anforderungen sollen, soweit das im Rahmen einer Diplomarbeit möglich ist, verwirklicht werden. Das System erhält den Namen ***CoLibry*** (von **Co***mponent* **Libr***ar***y**). Konkret sollen innerhalb von *CoLibry* folgende Bereiche realisiert werden:

- Entwicklung einer Komponentenbeschreibungssprache, die eine adäquate Erfassung der Informationen über Komponenten ermöglicht. Diese Datenstruktur ist mit den Mitteln des in der Arbeit verwendeten Datenbankmanagementsystems (s. Abschnitt 3.2) zu beschreiben.

- Realisierung einer Benutzungsschnittstelle. Die Anwendung soll so entworfen werden, daß Schnittstellen für alle erwünschten Funktionalitäten (ob realisiert oder nicht) bereitstehen. Von diesen Funktionen (s. Abschnitt 2.1) werden folgende umgesetzt:
  - Anzeige der Komponenten in Listendarstellung;
  - Anzeige der Komponentenbeschreibung in einem eigenen, übersichtlichen Fenster;
  - Anzeigen des Benutzungslogs in einem eigenen Fenster mit Listendarstellung der Einträge;
  - Aus einem oder mehreren Menüs aufrufbare Funktionen zum:
    * Anlegen, Anzeigen, Modifizieren, und Löschen einer Komponente, und der Logeintragungen;
    * Starten einer Schlüsselwort– sowie einer Facettensuche mit übersichtlicher Ausgabe der Ergebnisse.

- Diese Funktionalitäten bedingen die Entwicklung von:
  - einer Schnittstelle zum Datenbankmanagementsystem, und
  - einer automatischen Generierung von visuellen Elementen zur Datenanzeige.

Nicht realisiert werden hingegen folgende Elemente einer Komponentenbibliothek:

- Suchalgorithmen zur Schlüsselwort– und Facettensuche sowie das Parsen und Verarbeiten von Suchanfragen in der Anfragesprache des Datenbankmanagementsystems (s. Abschnitt 3.3);

- eine Baumdarstellung der Komponenten, sowie Möglichkeiten zur Filterung und Umsortierung der Anzeige;

- Statistikfunktionen und ihre Anzeige;

- ein Parser zum automatischen Generieren von Komponentenbeschreibungen aus vorhandener Dokumentation;

- ein Versionsmanagement zur Verwaltung verschiedener Versionen einer Komponente oder einer Komponentensammlung.

# Kapitel 3

# Ausgangspunkt

Diese Arbeit baut auf den in der Fachgruppe Praktische Informatik des Fachbereichs Informatik und Elektrotechnik der Universität Siegen entwickelten Systemen auf. Außerdem orientiert es sich an konkreten Anforderungen für den praktischen Einsatz (s. Abschnitt 3.4). Um verstehen zu können, warum *CoLibry* derart konzipiert und realisiert wurde, werden diese Gegebenheiten in diesem Kapitel vorgestellt.

## 3.1 Java

Die objektorientierte Programmiersprache *Java*, entwickelt von der Firma Sun Microsystems, hat seit ihrer Einführung Anfang der 90er Jahre umfangreiche Veränderungen und Erweiterungen erfahren. Mittlerweile hat sie sich zu einer Art Standard im Bereich der Programmierung von graphischen sowie von Netzanwendungen entwickelt. Dazu beigetragen haben folgende Eigenschaften von *Java*:

- ***Java* als interpretierte Sprache**. *Java* ist keine kompilierte, sondern eine interpretierte Sprache; dies bedeutet, daß *Java*–Programme beim Kompilieren nicht in Maschinencode, sondern in sogenannten *Byte–Code* umgewandelt werden, welcher nicht direkt ausgeführt werden kann. Der Byte–Code wird von einer sogenannten *Java Virtual Machine (JVM)* in für das verwendete Computersystem verständlichen Code interpretiert. Auf diese Weise können *Java*–Programme auf jedem System ausgeführt werden, für das eine JVM erhältlich ist.

  Diese Eigenschaft macht *Java* plattformunabhänging, und damit besonders für Anwendungen in heterogenen Umgebungen wie z.B. dem Internet hochgradig interessant. Doch ergeben sich durch die Interpretation im Vergleich zu kompilierten Sprachen hohe Ausführungszeiten.

- **Multithreading**. Durch den Einsatz sogenannter *Threads*, welche parallele Ausführung von Programmcode erlauben, eignet sich *Java* gut für interaktive, graphische Anwendungen. Wärend der Anwender auf das Ergebnis einer Eingabe wartet, kann er das Programm weiterhin ansprechen, was bei rein sequentieller Verarbeitung nicht möglich wäre.

- **Einfachheit**. *Java* ist eine einfach zu erlernende Sprache. Das ihr zugrundeliegende Prinzip der Objektorientierung ist auch dem unerfahrenen Anwender leicht erschließbar, da die verwendeten Konzepte wie *Klassen*, *Objekte*, und *Vererbung* ihm bereits bekannt und im alltäglichen Leben bei der Strukturierung von Daten behilflich sind.

  Weiterhin versucht *Java* den Programmierer vor solchen Unannehmlichkeiten wie Zeigerverwaltung und den damit verbundenen Fehlerquellen zu bewahren, indem es diese vor dem Anwender „versteckt“. Das mächtige Konzept der Fehlerbehandlung durch die sogenannten *Exceptions* soll ebenfalls Programmierfehler erkennen und Laufzeitfehler vermeiden helfen.

  Umfangreiche Klassensammlungen in Form der Standard–*Packages* von Java erleichtern ebenfalls das Programmieren fortgeschrittener Anwendungen. Z.B. läßt sich eine graphische Benutzerschnittstelle (engl. *Graphical User Interface (GUI)*) mit geringem Aufwand realisieren.

Für eine ausführliche Einführung in *Java* siehe [F96].

## 3.2 H-PCTE und JHPcte

H-PCTE [K92] ist eine partielle, hochperformante, hauptspeicherbasierte Implementierung von PCTE (*Portable Common Tool Environment*), dem ISO– und ECMA–Standard eines Integrationsrahmens für Software–Entwicklungsumgebungen (s. [ECM93]). Dieses Anwendungsgebiet stellt Anforderungen an ein Datenhaltungssystem, welche über die Möglichkeiten konventioneller, relationaler *Datenbankmanagementsysteme (DBMS)* hinausgehen. PCTE als ein strukturell objekt–orientiertes Datenmanagementsystem (bezeichnet als *Objektmanagementsystem (OMS)* oder auch *Repository*) vereinigt Konzepte aus DBMS und persistenten Programmiersprachen. Es bietet transparente Verteilung der Daten und partiellen Zugriff auf dieselben, ein Transaktionskonzept zur Realisierung von *Recovery*– und *Concurrency–Control*–Mechanismen, eine feingranulare Zugriffskontrolle auf die Daten, sowie einen Notifizierungsmechanismus bei Datenmodifikationen.

Die für diese Arbeit relevanten Merkmale von H-PCTE werden im folgenden näher vorgestellt.

### 3.2.1 Das Datenmodell

Das Datenmodell von H-PCTE basiert auf dem *Entity–Relationship–Modell* und kann als *strukturell objekt–orientiert* bezeichnet werden. Eine Objektbank besteht dabei aus eindeutig identifizierbaren *Objekten*, welche durch gerichtete *Links* miteinander verbunden sind. Sowohl Objekte als auch Links können attributiert sein. Ausgehend von der Wurzel der Objektbank, dem *Common–Root–Objekt*, oder dem *Home–Objekt*, welches für jeden Benutzer exisiert, kann durch die Objektbank navigiert werden. Durch sequentielle Angabe der Links, die vom Wurzelobjekt zum gewünschten Objekt führen, ist dieses erreichbar. Durch das Anlegen sogenannter *Handles*, das sind Referenzen auf bestimmte Objekte innerhalb der Objektbank, kann die Länge des anzugebenden Pfades verkürzt werden.

Das OMS ist *selbstreferentiell*, d.h. alle Objekt–, Link–, und Attributtypen, sowie ihre Beziehungen zueinander, sind ebenfalls durch eine Typstruktur innerhalb des OMS dargestellt. Diese Meta-Ebene ist für den Benutzer also genauso einsehbar wie die eigentlichen Daten, aber nur mit speziellen, konsistenzerhaltenden Werkzeugen zu bearbeiten.

Die in der Objektbank enthaltenen Objekte sind in disjunkte Teilmengen aufgeteilt, die auf (eventuell auch räumlich) separaten *Segmenten* (in PCTE: *Volumes*) gespeichert werden. Objekte können zwischen Segmenten verschoben werden, und Objekte in verschiedenen Segmenten können aufeinander verweisen. In der Regel sind diese Vorgänge für den Anwender *transparent*, d.h. es ist für ihn nicht relevant, auf welchen Segmenten sich bestimmte Daten befinden.

#### Attributtypen

Der Attributtyp legt folgende Eigenschaften eines Attributes fest:

- Attributname; dieser ist innerhalb des Schemas (s.u.), und durch Voranstellen des Schemanamens auch global, eindeutig

- Wertebereich; bestimmt die Daten, die in einem Attribut dieses Typs gespeichert werden können

- Initialwert

- Duplikationseigenschaft; bestimmt, ob beim Kopieren eines Attributs dieses Typs auch dessen Wert mitkopiert wird; falls nicht, wird es auf seinen Initialwert gesetzt.

#### Objekttypen

Objekttypen in H-PCTE sind definiert durch:

- eine Menge von Attributen

- eine Menge ausgehender Linktypen
- ein oder mehrere Elterntypen (*multiple inheritance*); der Objekttyp erbt alle Attribute und zulässigen ausgehenden Linktypen der Elterntypen

**Linktypen**

Linktypen in H-PCTE werden durch folgende Merkmale bestimmt:

- eine Menge von Schlüsselattributen; besitzt der Link kein Schlüsselattribut, ist er von der Kardinalität [0,1]
- ein Menge von Nicht–Schlüsselattributen
- eine Menge von zulässigen Zielobjekttypen
- einen Umkehrlinktyp
- Kategorie; Linktypen in H-PCTE gehören zu einer der folgenden Kategorien: *implicit, existance, reference, designation*, oder *composite*. Diese Unterscheidung spiegelt wichtige semantische Unterschiede wider, für die Arbeit sind jedoch nur die Kategorien *existance, reference*, und *composite* von Bedeutung: Jedes Objekt kann nur zusammen mit einem Link mit *Existenzeigenschaft* erzeugt werden, die *Referenzeigenschaft* garantiert, daß das Zielobjekt exisitiert, und die *Kompositionseigenschaft* deutet an, daß die verbundenen Objekte zu einer größeren semantischen Einheit, einem *komplexen Objekt*, gehören
- andere Merkmale, die hier keine Rolle spielen.

### 3.2.2 Zugriffskontrollen

H-PCTE erlaubt durch den Einsatz sogenannter *Access Control Lists (ACL)* feingranulare, dynamische Rechtevergabe auf seinen Daten: Jedem Objekt ist eine ACL zugeordnet. Es können unterschiedliche Benutzer angelegt werden, die wiederum einer oder mehreren Benutzergruppen zugeordnet sind. Für diese Gruppen oder einzelne Benutzer können Einträge in die ACL gemacht werden, welche ihre Rechte für dieses Objekt festlegen. Es gibt in H-PCTE 23 verschiedene Zugriffsarten auf Objekte und Links, welche weit über die Möglichkeiten z.B. des UNIX-Dateisystems hinausgehen (z.B. „Objekt darf gelöscht, aber nicht modifiziert werden").

Auf der Typebene erlaubt das Setzen von *Arbeitsschemata* eine Art Zugriffskontrolle (s. Abschnitt 3.2.4). Schließlich gibt es die Möglichkeit, durch das Setzen und Aufheben feingranularer Sperren den Zugriff auf Objekte zu regulieren (im Zusammenhang mit Transaktionen).

### 3.2.3 Notifizierung

H-PCTE realisiert eine *Client–Server*–Architektur. Der H-PCTE–Server stellt dabei verschiedenen Client–Prozessen seine Dienste zur Verfügung. Initiiert eine der Client–Anwendungen Änderungen in der Objektbank, müssen diese Änderungen auch zu den anderen Clients propagiert werden. Dafür steht ein Notifizierungsmechanismus bereit: Meldet eine Anwendung für die sie interessierenden Daten *Notifizierer* an, erhält sie Mitteilungen über Attributänderungen, das Anlegen und Löschen von Objekten und Links, sowie über die Änderungen von Zugriffskontrollen, und das Anlegen und Aufheben von Sperren. Die Anwendung kann mit diesen Informationen ihre Sicht auf die Objektbank den Änderungen anpassen, und damit konsistent halten.

### 3.2.4 Schemata

Für diese Arbeit ist die Art und Weise, wie in H-PCTE Typstrukturen definiert werden, von besonderer Wichtigkeit, da das zu entwickelnde Modell einer Komponentenbeschreibung darin umgesetzt werden soll.

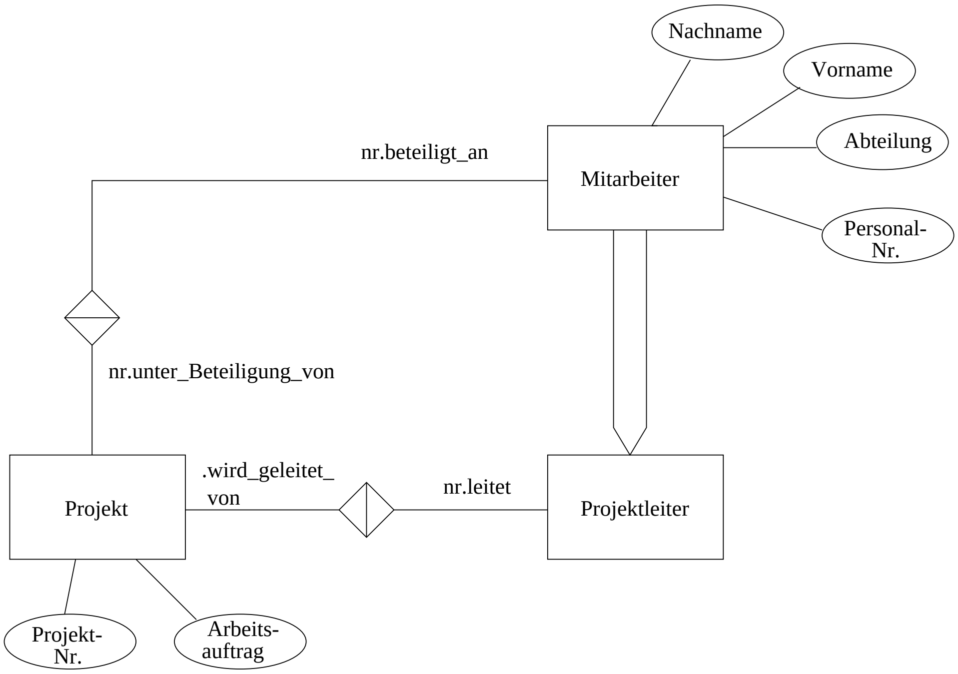


Abbildung 3.1: Graphische Darstellung eines Beispiel–Schemas

In H-PCTE werden alle Typdefinitionen in *Schema Definition Sets (SDS)* verwaltet. Ein *Arbeitsschema* (engl. *working schema*) entspricht einer externen Sicht oder einem externen Schema. Definiert wird es durch eine Folge von SDSen. Jedem Prozeß ist zu einem gegebenen Zeitpunkt genau ein Arbeitsschema zugeordnet, welches seine Sicht auf die Daten bestimmt. Er kennt nur die Typen, welche in mindestens einem der zum Arbeitsschema gehörenden SDSe definiert sind. Wird ein Typ in mehreren SDSen definiert, gilt die letzte Definition als die gültige.

In einem SDS können auch *statische Typrechte* vergeben werden, welche dann für alle Objekte dieses Typs gelten. Wie schon erwähnt, werden auch die Metadaten, welche die Definitionen von Typen enthalten, über SDSe definiert (*Selbst–Referentialität*).

Eine graphische Darstellung eines Beispiel–Schemas ist in Abbildung 3.1 angegeben. Für eine Erläuterung der Notation der graphischen Darstellungen, sowie der SDS–Definitionen selber, sei auf [K99] verwiesen.

### 3.2.5 JHPcte

Im Zuge der Beliebtheit von *Java* und seiner Plattformunabhängigkeit wurde die für ursprünglich in *C* gehaltene Schnittstelle von H-PCTE ein Aufsatz in *Java* entwickelt [M98]. Die *Java*–Schnittstelle heißt JHPcte, und realisiert ebenfalls eine Server–Client–Architektur. Ein *Java*-Client, welcher H-PCTE nutzen will, nutzt den JHPcte–Server. Dieser kommuniziert über eine *Socket–Verbindung* mit dem H-PCTE–Server.

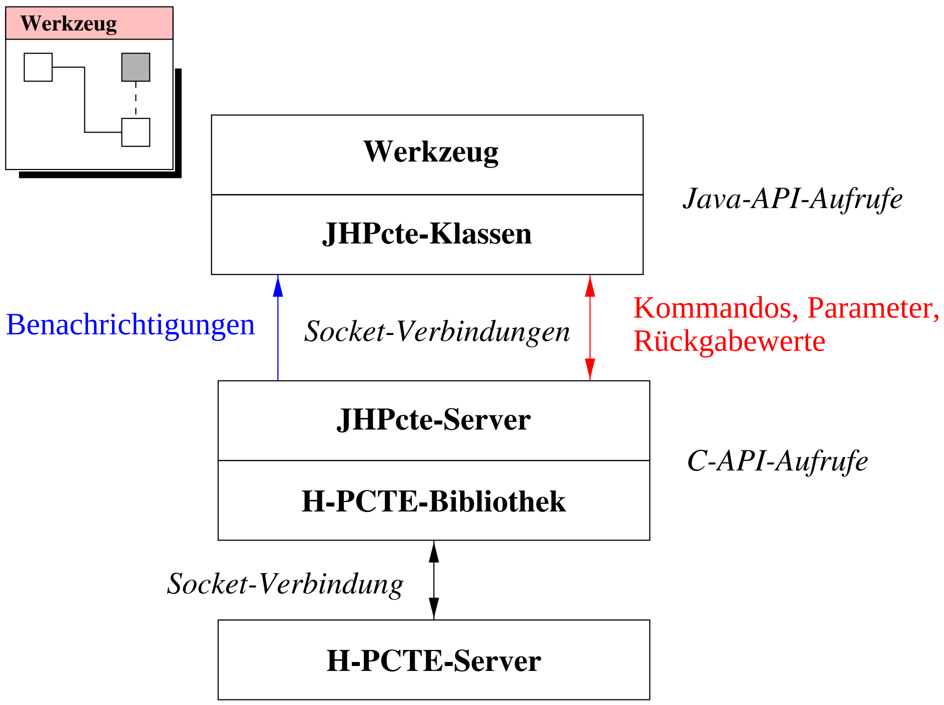


Abbildung 3.2: Prozeß–Architektur von JHPcte (aus [KMP98])

Der JHPcte–Server kann mehrere Clients bedienen, wobei für jeden einzelnen *Java*–Client beim Start ein H-PCTE–Prozeß erzeugt wird. Ein Client kann mehrere Prozesse starten, und für jeden Prozeß existiert im JHPcte–Server ein *Thread*, welcher die in diesem Prozeß erzeugten Kommandos ausführt und Ergebnisse zurücksendet.

Neben der Socket–Verbindung für die Kommunikation exisitiert für jeden Client eine Socket–Verbindung zur Übermittlung von Notifizierungsnachrichten. Für die Überwachung dieses Sockets steht dem Client ein eigener Thread zur Verfügung, so daß der Client unmittelbar auf Änderungen in der Objektbank reagieren kann.

JHPcte bietet ebenfalls Klassen, welche die Einbindung von NTT-Anfragen (s. Abschnitt 3.3) in Applikationen erleichtern.

## 3.3 NTT

Im PCTE–Standard ist kein mengenorientierter Zugriff auf die Objektbank vorgesehen, da dessen Implementierung verschiedene Probleme aufwirft, wie z.B. die Behandlung partieller Unzugreifbarkeit von Daten. Um den Bedarf eines mengenorientierten Zugriffs zu decken, sind einige Anfragesprachen entwickelt worden. Dazu gehören P-OQL (*PCTE Object Query Language*, [He95]) und NTT (*Non–navigating Tool Tongue*, [HK96]).

NTT bietet die üblichen Mengenoperationen namens `union` (Vereinigung), `intersect` (Durchschnitt) und \ (Differenz), mit deren Hilfe Objektmengen verknüpft werden können. Neben diesen Operationen gibt es fünf weitere Operatoren, welche alle bis auf *Extension* eine Objektmenge als Ein– wie auch als Ausgabe haben:

- **Extension**; berechnet alle Objekte, die den angegebenen Typ haben.
- **Relatives**; ermöglicht mengenwertige Navigation über Links
- **Expand**; ermöglicht die Definition *virtueller* Attribute und Links
- **Tuples**; formal die Identität, produziert diese Operation als Nebeneffekt eine tabellarische Darstellung der Objektmenge.

Für eine ausführliche Einführung in NTT sei auf die Literatur verwiesen, z.B. [H94].

## 3.4 JavaPcteLib

Bei JavaPcteLib handelt es sich um eine in der Fachgruppe Praktische Informatik des Fachbereichs Elektrotechnik und Informatik der Universität Siegen erstellte Bibliothek von *Java*–Klassen, welche ständig erweitert wird. Inhaltlich dienen viele Klassen zur Unterstützung der Einbindung von JHPcte in *Java*, indem sie Funktionalitäten der JHPcte–Bibliothek entweder erweitern oder ergänzen.

Logisch und funktionell ist JavaPcteLib in verschiedene *Java Packages* aufgeteilt; z.B. finden sich Klassen zur Realisierung eines graphischen Login–Dialogs für eine auf H-PCTE basierende Applikation in dem Package `de.uniSiegen.informatik.pi.classes.javapctelib.dialogs`. (Der Packagename folgt den Konventionen zur eindeutigen Benennung von Packages durch das Voranstellen des Institutions–/Firmennamens in umgekehrter Reihenfolge.)

Neben Klassen, die eine komfortable Schnittstelle zu H-PCTE bereitstellen, bietet JavaPcteLib z.B. eine Fensterverwaltung an. Mit den Packages `javapctelib.actions` und `javapctelib.help` stehen Erweiterungen von *Java Actions* bzw. eine leicht einzubauende Hilfefunktion zur Verfügung.

JavaPcteLib ist jedoch nicht nur als diese Arbeit unterstützende Bibliothek interessant, sondern auch als Motivation für die Entstehung des Themas dieser Arbeit, und als Arbeitsziel: JavaPcteLib ist selber ein Beispiel für eine schnell wachsende Bibliothek von Software–Komponenten, für die außer der automatisch aus Codekommentaren erstellten *Java*–Dokumentation (*javadoc*) keine Übersicht zur Verfügung steht. Als Anwendungsbeispiel soll also JavaPcteLib mittels *CoLibry* erfaßt werden.

Einige der für *CoLibry* entwickelten Konzepte und Klassen haben ihren Weg nach JavaPcteLib gefunden, und sind damit nicht im Codeumfang von *CoLibry*, sondern in der Bibliothek zu finden, von wo aus sie eingebunden werden (s. Kapitel 4).

# Kapitel 4

# Entwurf von *CoLibry*

## 4.1 Technische Anforderungen

Wie schon in Kapitel 3 angedeutet, erfolgt die Speicherung der Beschreibungsdaten, welche *CoLibry* enthält, mit Hilfe des OMS H-PCTE. Die Daten sind also strukturell objektorientiert zu modellieren, und werden mittels Schemata beschrieben (s. Abschnitt 4.2). Die Benutzungsschnittstelle *CoLibry* wird in *Java* realisiert, der Anschluß an H-PCTE erfolgt über die Schnittstellen, welche JHPcte und JavaPcteLib zur Verfügung stellen, bzw. über eigens im Rahmen dieser Arbeit realisierte Schnittstellen (s. Abschnitt 4.4.2).

Derzeit ist nicht vorgesehen, in *CoLibry* auch die Komponenten selber zu speichern. In einer Beschreibung werden Adressen angegeben, unter denen die Kompontente gefunden werden kann. Damit ist die Verwaltung von Komponentenbeschreibungen nicht auf lokal erfaßte beschränkt, sondern bei Wahl eines adäquaten Adressierungsformats können auch entfernte Komponentensammlungen erfaßt werden. Die Speicherung von Binärcode ist von der technischen Seite her (also für H-PCTE) kein Problem, jedoch zum jetzigen Zeitpunkt noch kein Primärziel.

Wenn in Zukunft von Speicherung von Komponeten o.ä. die Rede ist, sind damit immer implizit die Beschreibungen, und nicht die Komponenten selber, gemeint.

## 4.2 Schema–Entwurf

Aus der vorhandenen Literatur (s. Abschnitt 2.2.5) sowie eigenen Überlegungen entwickelte sich das Datenmodell für *CoLibry*. Ausgangspunkt war der Wunsch, umfangreiche Informationen über eine einzelne Komponente bereitzustellen. Ein solche Menge an Daten muß zur besseren Übersichtlichkeit strukturiert werden. Eine Komponentenbeschreibung deckt also verschiedene logische Bereiche ab, welche sich je nach Komponentensorte unterscheiden können.

Derzeit ist nicht vorgesehen, dynamisch Änderungen bzw. Erweiterungen an Schemata vorzunehmen[1]. Die Schemata sind also so zu spezifizieren, daß sie die Anforderung der Beschreibung unterschiedlicher Komponentensorten von vornherein erfüllen.

### 4.2.1 Identifizierung der verschiedenen Dokumentsorten

Wie bereits in Abschnitt 1.1.2 erwähnt, bezieht sich der Begriff *Komponente* in dieser Arbeit speziell auf ausführbare Elemente, sowie auf Entwurfsmuster (engl. *Design Patterns*). Ausführbare, wiederverwendbare Elemente der Software–Entwicklung können verschiedenen Komplexitäts– und Funktionalitätsstufen angehören (s. Abschnitt 1.1.1). Am einfachsten konzipiert ist die *Funktion* (oder auch *Methode*, *Prozedur*), gefolgt von der *Klasse*, und noch einen Schritt weitergehend dem *Klassenverband*, einer hierarchischen Struktur zusammenhängender und zusammenarbeitender Klassen, die nach außen eine einheitliche Schnittstelle präsentieren. Auch sehr fortgeschrittene Software–Komponenten wie z.B. objektorientierte *Frameworks* können im Grunde als ein solcher Klassenverband angesehen werden.

Weitere Überlegungen führen zu der Erkenntnis, daß ein signifikanter inhaltlicher Unterschied in der Beschreibung einer Funktion und einer Klasse besteht: Eine Funktion ist meistens leicht zu verstehen; unter Angabe ihrer Funktionalität und der Schnittstelle (Ein– und Ausgabeparameter) ist sie bereits hinreichend dokumentiert. Eine Klasse hingegen ist meist komplexer; umfangreichere Dokumentation, Umgebungs– und Bewertungsinformationen, und Angaben zu mit ihr verflochtenen anderen Klassen sind nötig, um sie verstehen und evaluieren zu können. Von der Klasse zum Klassenverband wiederum besteht nur ein quantitativer, kein qualitativer Unterschied: Die höhere Komplexität erfordert noch umfangreichere Informationen in den einzelnen Bereichen, jedoch kommen keine neuen dazu. Entwurfsmuster schließlich unterscheiden sich so sehr von ausführbaren Kompontenten, daß ganz andere Beschreibungskriterien notwendig sind.

Es sind also drei verschiedene Komponentensorten (im folgenden auch *Dokumentsorten* genannt) identifiziert worden:

- **Funktion** (*Function*),
- **Klassenverband** (*Classgroup*), und
- **Entwurfsmuster** (*Pattern*).

[1] Dies würde einerseits eine neuerliche Einkompilierung der Schemata in H-PCTE erfordern, und damit zusammenhängend Kompatibilitäts–Probleme bei der Übernahme der bereits gespeicherten Daten bereiten, als auch eventuelle Anpassungen am Programmcode von *CoLibry* erfordern.

Für jede dieser Dokumentsorten ist ein eigenes Schema zur Speicherung ihrer Beschreibung vorgesehen.

Alle abgespeicherten Komponenten werden einer dieser drei Dokumentsorten zugeordnet. Nach außen kann allerdings eine beliebige Kategorisierung der Kompontenten erfolgen, welche intern auf die Dokumentsorten abgebildet wird. Eine solche sinnvolle Kategorie wie z.B. *Java Bean* kann also angelegt, und ihr können Komponenten zugewiesen werden, diese Kategorie bildet intern jedoch wieder auf die Dokumentsorte *Classgroup* ab. Kategorien sind insofern unabhängig von Beschreibungen, als zu einem gegebenen Zeitpunkt keine der enthaltenen Beschreibungen einer bestimmten Kategorie zugeordnet sein muß. Die Kategorie ist daher kein integraler Bestandteil der Kompontentenbeschreibung, sondert gesondert zu betrachten.

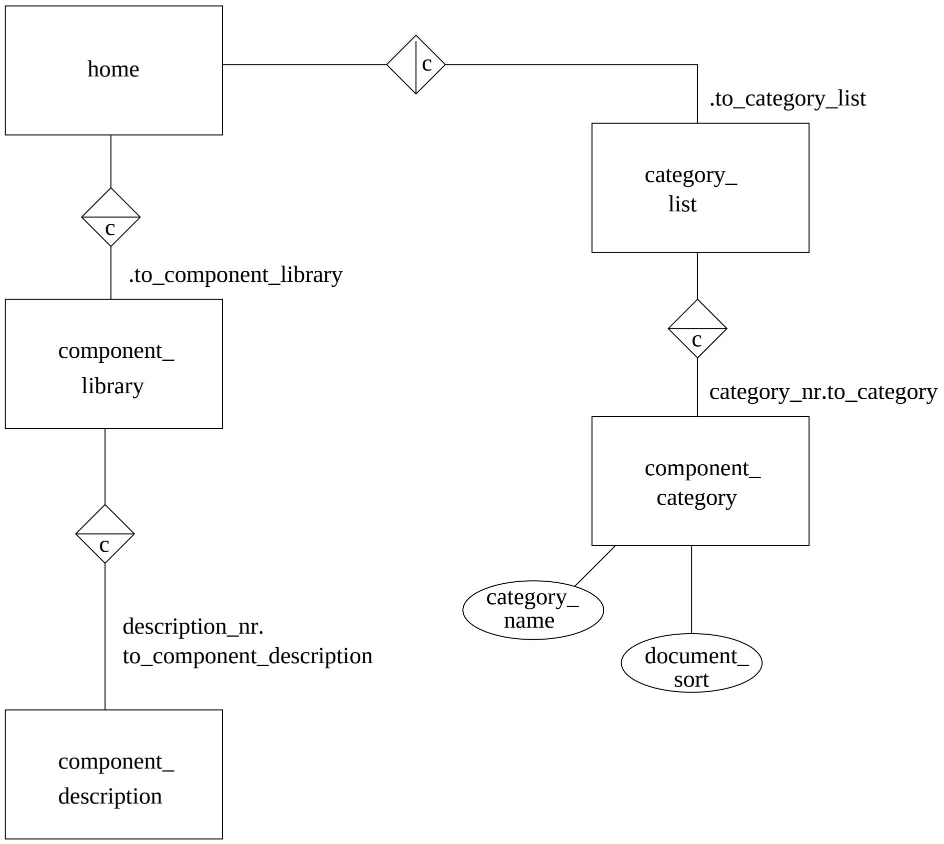


Abbildung 4.1: Graphische Darstellung des Schemas *library*

Damit ist die grundlegende Datenstruktur unter H-PCTE festgelegt. Die graphische Darstellung findet sich in Abbildung 4.1. Ausgehend vom *Home–Objekt* des Benutzers wird die Bibliothek, bestehend aus einer Menge von Komponentenbeschreibungen, erreicht. Die Kategorienliste, welche die Menge der Komponentenkategorien umfaßt, wird ebenfalls an der Wurzel des Objektbaumes gespeichert. Die Links sind vom Typ *composite*, da es sich hier um eine „besteht aus "–Beziehung handelt. Ein Objekt vom Typ `component_category` hat die Attribute Name und Dokumentsorte. Objekte vom Typ `component_description` können einer der drei Dokumentsorten angehören (s. nächster Abschnitt).

Diese Struktur ist im Schema *library* definiert (für die textuelle Fassung siehe Anhang B).

### 4.2.2 Entwurf der Schemata für die Dokumentsorten

#### Das Schema *base*

Es gibt gewisse Teile der Beschreibung, welche allen Komponentensorten gemeinsam sind. Diese gemeinsame Basis, sozusagen das Skelett einer konkreten Beschreibung, ist im Schema *base* modelliert. Eine graphische Darstellung des Schemas bietet Abbildung 4.2. Wie dort zu sehen ist, enthält jede Beschreibung administrative, kategorisierende, und inhaltliche Information, sowie das Benutzungslog mit seinen Einträgen.

Die Aufteilung der Beschreibung (bei konkreten Komponentsorten kommen zum Teil noch weitere Bereiche hinzu, s. den Abschnitt zum Klassenverband) motiviert sich aus dem Wunsch, die umfangreichen Informationen über Komponenten zu ordnen. Ähnliche Ansätze zur Kategorisierung vorhandener Daten finden sich auch in der ausgewerteten Literatur in Abschnitt 2.2.

Der Komponentenname sowie Angaben zu Autor, Firma, und Adresse, unter welcher die Komponente zu beziehen ist, sind administrative Informationen, und finden sich deshalb im Objekt `administrative_info`. Es können mehrere Autoren, Firmen, und Adressen vorhanden sein (oder auch gar keine), weswegen diese Links mit einem Schlüsselattribut versehen sind. Da diese Informationen optionalen Charakter besitzen, werden die Links nicht als vom Typ *composite*, d.h. als Teil des Mutterobjekts `administrative_info`, sondern vom Typ *existence* modelliert.

Die kategorisierende Information, welche eine Einordnung der Komponente ermöglicht, ist dementsprechend im Objekt `categorizing_info` modelliert, welches wiederum zu einer der vorhandenen Komponentenkategorien verlinkt.

Inhaltliche Information, durch das Objekt `contents_info` repräsentiert, besitzt als gemeinsamen Nenner nur Objekte vom Typ `classification`, welche Schlüssel–Wert–Paare enthalten, die Informationen zur Funktionalität der Komponente in Form von *Facetten* (s. Abschnitt 2.2.1) erfassen.

Das Benutzungslog zuletzt besteht aus dem Objekt `usage_log`, an welchem die Anzahl der Benutzungen der Komponente attributiert ist, sowie den einzelnen Eintrags–Objekten `usage_entry`.

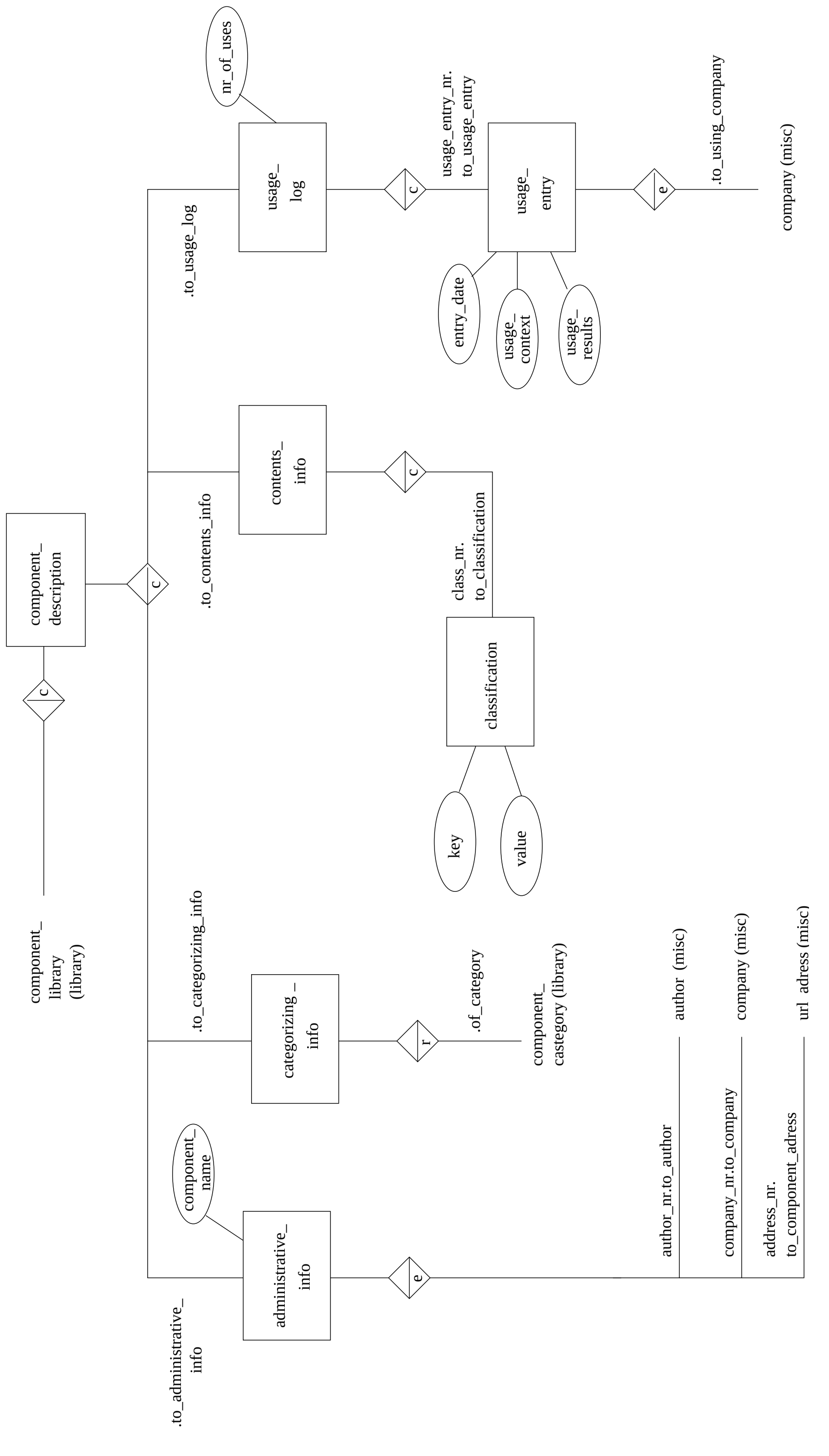


Abbildung 4.2: Graphische Darstellung des Schemas *base*

Diese verzeichnen Eintragsdatum, Kontext, Ergebnisse, sowie einen Link zur benutzenden Firma. Damit können Erfahrungen mit der Komponente differenziert gespeichert und durch Browsing des Logs gezielt wieder abgerufen werden.

### Das Schema *function*

Das Schema zur Komponentsorte Funktion, *function*, ergänzt das Schema *base* um funktionenspezifische Informationen. Abbildung 4.3 zeigt das Schema; der Bereich Benutzungslog ist hier wie in den folgenden Abbildungen weggelassen worden, da er sich nicht von dem des Schemas *base* unterscheidet. Neu ist eine Verlinkung vom Objekt `component_description` zu anderen Objekten desselben Typs. Dieser Link führt zu einer nachfolgenden Version der Komponente, und ist mit einer textuellen Beschreibung der Unterschiede, dem *changes file*, attributiert.

Beim Bereich Administrative Information kommen Angaben zu Version, Versionsdatum und Herkunft der Komponente hinzu, die Kategorisierung erhält die zusätzlichen Attribute Richtung[2], Domäne und Ausführbarkeit.

Die inhaltliche Information wird ergänzt um die Attribute Kurzbeschreibung, Programmiersprache, und Interface. Neu hinzu kommt das Objekt `description_links`, welches Links zu von dieser Komponente benutzten anderen Komponenten sowie eine Strukturbeschreibung dieser Hierarchie enthält. Die Links zu anderen Beschreibungen sind attributiert; die Werte FREE bzw. REQUIRED geben an, ob die benutzte Komponente zwingend erforderlich ist oder nicht. Diese Information wird erst im Bereich des Klassenverbands wirklich wichtig.

### Das Schema *classgroup*

Das Schema *classgroup* (oder auch verkürzt *classgrp*) ist wiederum eine Erweiterung des Schemas *function*. In Abbildung 4.4 des Schemas ist der Bereich Kategorisierung nicht aufgeführt, da er mit dem von *function* übereinstimmt. Im Bereich inhaltliche Information kommen die Angaben zu Dokumentation und Benutzungsbeschreibung hinzu, modelliert durch die Objekte `documentation` und `usage_description`, sowie das Attribut FAQ (engl. *Frequently Asked Questions*, eine Zusammenstellung häufig gestellter Fragen zu einem Thema). Die Dokumentation verlinkt zu Adressen, unter denen sie gefunden werden kann, wird also nicht im *Repository* selber gehalten. Als zwei neue Informationsbereiche, die durch eigene Objekte repräsentiert werden, finden sich Umgebungsinformation und evaluierende Information.

[2]Gibt an, ob die Komponente Funktionalität für höhere Schichten anbietet (*vertikal*), oder für Komponenten der eigenen Schicht (*horizontal*).

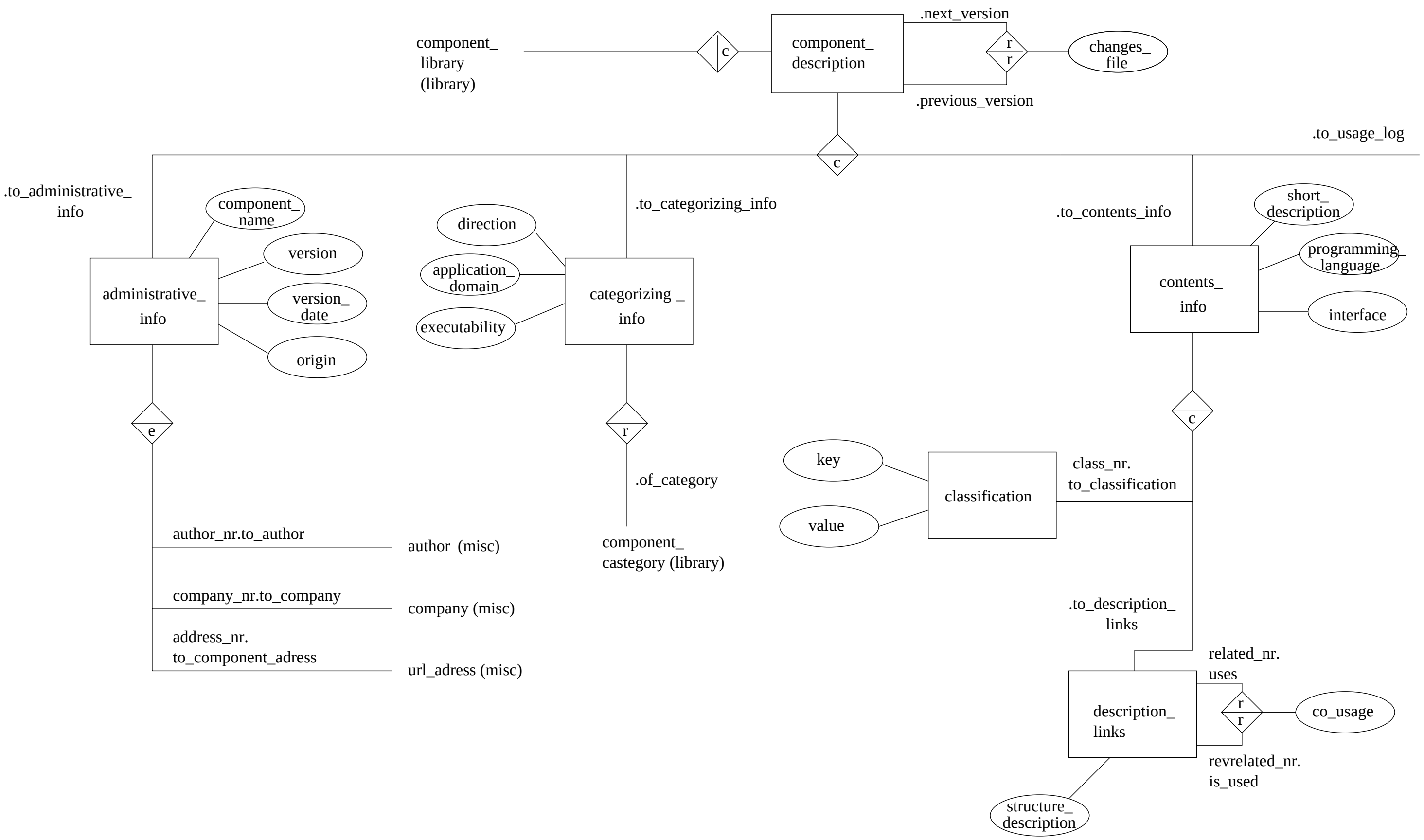


Abbildung 4.3: Graphische Darstellung des Schemas *function*

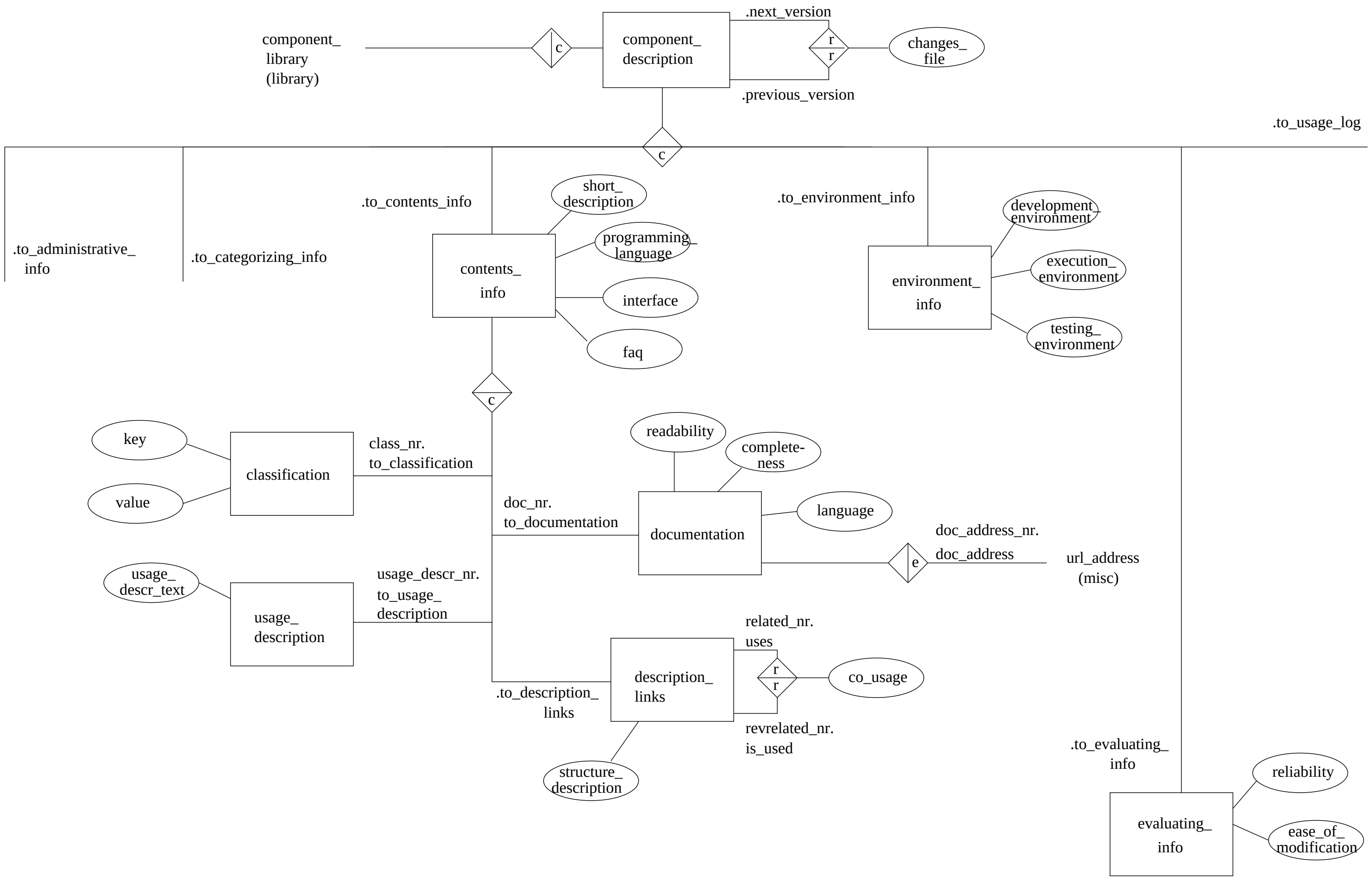


Abbildung 4.4: Graphische Darstellung des Schemas *classgrp*

Erstere enthält Angaben zu Entwicklungs–, Test–, und Einsatzumgebung der Komponente, letztere Bewertungen zur Zuverlässigkeit und Anpaßbarkeit der Komponente an vorgegebene Bedingungen. Diese Bewertungen werden durch den Bibliotheks–Manager explizit eingetragen; weitere und eventuell differierende Bewertungen finden sich im Benutzungslog. Wünschenswert wäre es, wenn der Manager in gewissen Abständen diese Eintragungen im Objekt `evaluating_info` anhand der protokollierten Erfahrungen auf den neuesten Stand bringt. Hier wird wieder deutlich, wie wichtig eine umfangreiche Pflege von Komponentenbibliotheken ist.

### Das Schema *pattern*

*Entwurfsmuster* (engl. *design pattern*) sind persistente Muster in Software–Systemen, die immer wieder im Entwurf auftauchende Probleme lösen. Ein Beispiel ist das Muster *Iterator*, welches eine Möglichkeit bietet, alle Elemente eines zusammengesetzten Objekts sequentiell zu erreichen, ohne daß die zugrundeliegende Datenstruktur bekannt sein muß. Da viele solcher Muster identifiziert worden sind und werden, welche außerdem auf ganz unterschiedliche Bereiche des Entwurfs eingehen, gibt es Bemühungen, Entwurfsmuster einheitlich zu beschreiben und zu klassifizieren. Ein Standardwerk auf diesem Gebiet ist [GHJ95], welches 23 häufig auftretende Muster katalogisiert. Dafür ist von den Autoren eine Klassifizierung entwickelt worden, welche für die Modellierung der Beschreibung eines Musters in dieser Arbeit übernommen wurde. Eine ausführliche Erläuterung dieser Klassifizierung würde den Rahmen der Arbeit sprengen, deshalb sei für Details auf [GHJ95] verwiesen.

Das Schema ist eine direkte Ergänzung des Schemas *base*, und wird in Abbildung 4.5 gezeigt. Im Vergleich zu den anderen Dokumentsorten sind die Attributierungen von `administrative_info`, `categorizing_info` und `contents_info` unterschiedlich. Neu hinzugekommen sind die Objekte `sample`, `known_use`, `structure`, und `participant`.

### Das Schema *misc*

Im Schema *misc* (von engl. *miscellaneous*, „gemischt“) sind alle Objekte definiert, die in anderen Schemata an verschiedenen Stellen gebraucht werden. So taucht z.B. das Objekt `company` im Bereich Administrative Information und Benutzungslog auf (einmal als Hersteller–Firma der Komponente, dann als benutzende Firma). Der Bibliotheksbenutzer, hier als Objekt `library_user` der Firma untergeordnet, erscheint nur im Benutzungslog, in dem benutzende Firma und Ansprechpartner angegeben werden.

Der Autor einer Komponente ist mit `author` ebenfalls modelliert. Alle diese Objekte besitzen gebräuchliche Attribute wie Name und Adresse, sowie u.U. eine URL–Adresse. Das Objekt `url_address` enthält neben der Adresse auch noch das Attribut Zuverlässigkeit.

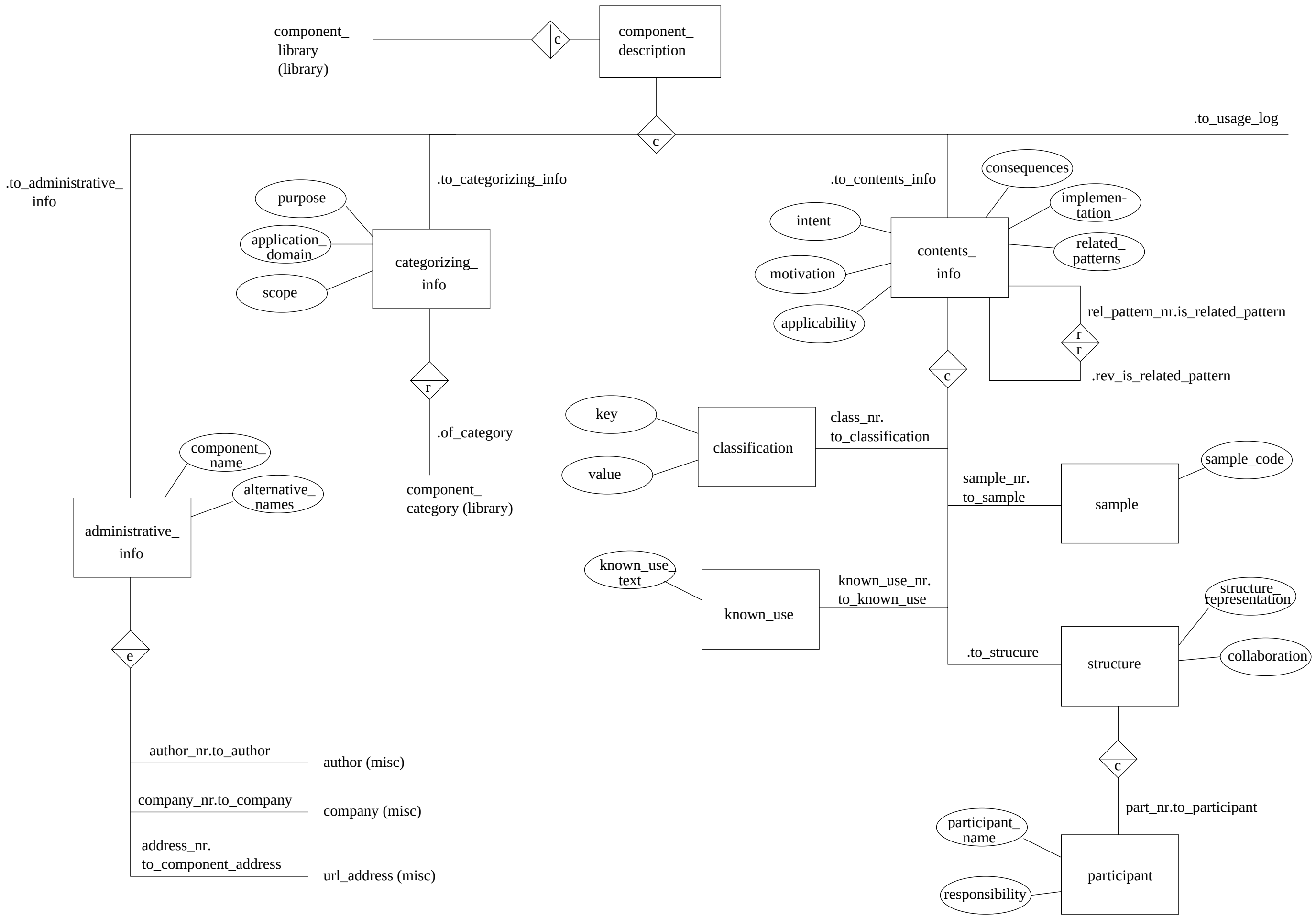


Abbildung 4.5: Graphische Darstellung des Schemas *pattern*

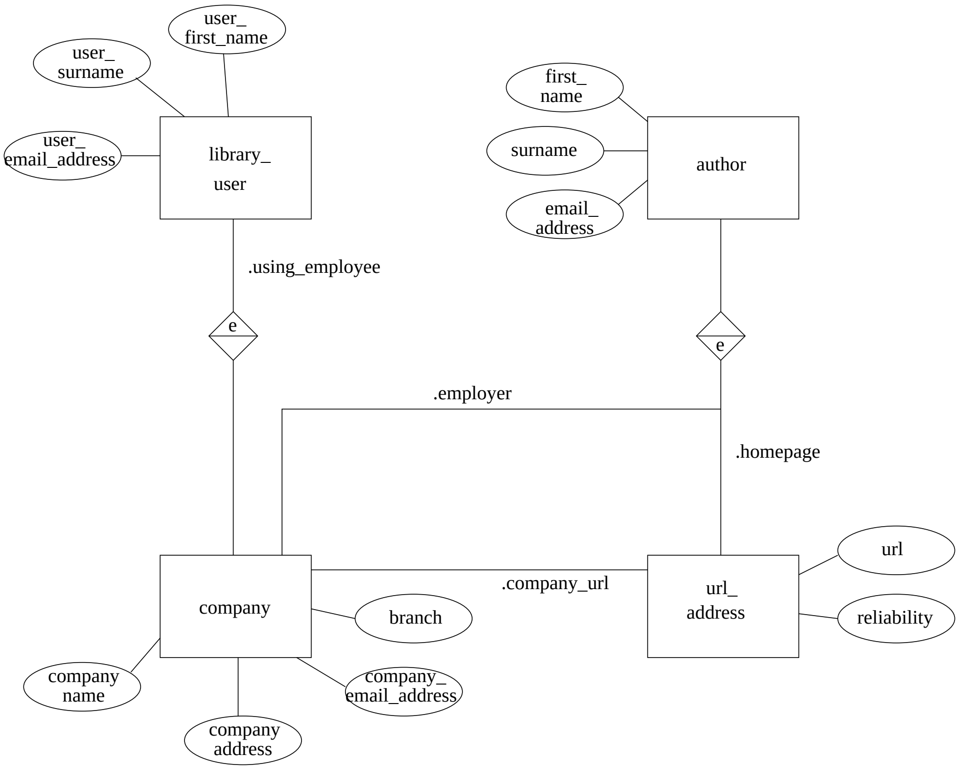


Abbildung 4.6: Graphische Darstellung des Schemas *misc*

## 4.3 Software–Entwurf

Wie schon in der Aufgabenstellung (Abschnitt 2.3) erläutert, kann diese Arbeit nur einen ersten Prototyp einer Komponentenbibliothek mit benutzerfreundlicher Schnittstelle realisieren. Vorgesehen wurde jedoch von vornherein, ein modulares, erweiterbares System zu erstellen, welches bereits Schnittstellen für später hinzuzufügende Funktionalitäten und Bereiche anbietet. Außerdem sollen die erstellten Komponenten möglichst wiederverwendbar gestaltet werden. Mit anderen Worten, *CoLibry* soll selber den an moderne Software gestellten Ansprüchen von Erweiterbarkeit und Wiederverwendbarkeit genügen. Bei der Planung der Software–Architektur sind diese Vorgaben zu berücksichtigen. Natürlich kann eine enge Verzahnung verschiedener Teilsysteme nicht völlig vermieden werden, z.B. zwischen Datenmanagement und darauf aufbauenden Komponenten.

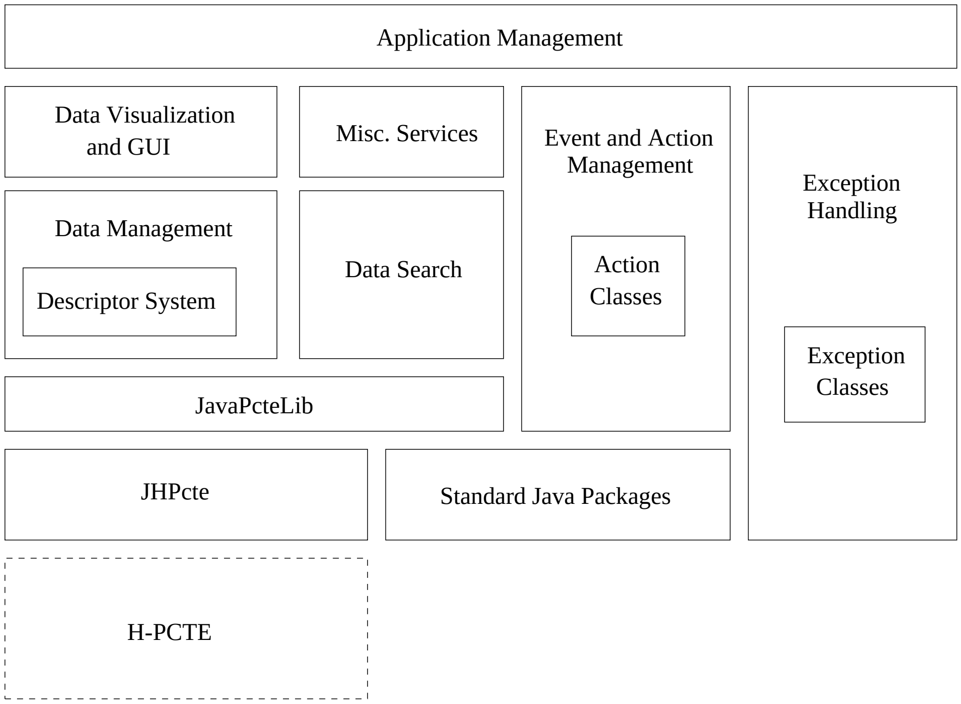


Abbildung 4.7: Die Architektur von *CoLibry*

Die **Grobarchitektur** von *CoLibry* läßt sich, wie in Abbildung 4.7 zu sehen, folgendermaßen beschreiben: Grundlage sind die Standard *Java*–Klassen und H-PCTE, sowie deren Aufsätze JHPcte und JavaPcteLib. Darauf baut die weitere Schicht von *CoLibry* auf, die *Data Management*–Schicht. Der Zugriff auf JHPcte und JavaPcteLib wird damit gekapselt, und eine spezielle Schnittstelle wird bereitgestellt. Ein wichtiger Bestandteil dieser Kapselung ist das *Descriptor System.* Abschnitt 4.4.2 beschreibt diesen Bereich genauer.

Unmittelbare Nutznießer des *Data Management* sind sowohl der darüberliegende Teil *Data Visualization and GUI*, als auch der Bereich *Data Search* (s. Abschnitt 4.4.1). Weitere Funktionen von *CoLibry* sind im Bereich *Miscellaneous Services* zusammengefaßt. Hierunter fallen jene Funktionalitäten, welche im Rahmen der Arbeit nicht realisiert werden können.

*CoLibry* ist eine benutzergesteuertes System, d.h. vom Benutzer werden *Ereignisse* (engl. *Events*) ausgelöst. Diese Ereignisse müssen vom System aufgefangen und verarbeitet werden. Dafür zuständig sind Erweiterungen der aus *Java* bekannten *Actions*, enthalten im Bereich *Event and Action Management.* Siehe Abschnitt 4.4.3 für eine detaillierte Beschreibung.

Die ebenfalls aus *Java* bekannten *Exceptions* werden durchweg zum Abfangen von Fehlern benutzt. Ein in JavaPcteLib enthaltenes Messaging–System erlaubt das Weiterleiten von Exception–Meldungen an die graphische Benutzerschnittstelle.

Die auf den beschriebenen Teilsystemen aufliegende obere Schicht ist das *Application Management.* Dieser für den Benutzer nicht sichtbare Bereich stellt grundlegende Ressourcen zur Verfügung, erledigt die Anmeldung am JHPcte–Server, und startet die graphische Schnittstelle in Form des Hauptfensters.

Erwähnenswert ist noch die Konzeption von *CoLibry* als mehrsprachige Anwendung: *Java* stellt dem Nutzer Möglichkeiten zur *Internationalisierung*[3] zur Verfügung. Länder– bzw. sprachspezifische Ressourcen werden damit nicht mehr im Programmcode, sondern in eigenen Dateien gespeichert, und die gewünschten Ressourcen werden ins Programm eingebunden.

Unter JavaPcteLib steht eine Erweiterung dieses Systems zur Verfügung (im Package `resources`): Ressourcen können direkt in H-PCTE gehalten werden, und mit entsprechenden Befehlen ausgelesen und geschrieben werden. Diese Schnittstelle wird von *CoLibry* genutzt, um sowohl deutsch– als auch englischsprachige Beschriftungen für Textelemente der Anwendung anzubieten. Auch nichttextuelle Eigenschaften der Anwendung, welche leicht konfigurierbar gehalten werden sollen (z.B. Größe von Fenstern), werden in diesem Resourcensystem gespeichert. Die Ressourcen sind in Gruppen aufgeteilt, welche eine interne Sortierung der Elemente widerspiegeln – hier eine Aufteilung in Daten–, GUI–, und Suchbereich.

Eine ebenfalls aus JavaPcteLib eingebundene Funktionalität ist das *Fenstermanagement*, welches eine Verknüpfung von Eltern– und Kindfenstern erlaubt, dafür Abfrage und Manipulationsmöglichkeiten anbietet, sowie ein hierarchisches Löschen aller Kindfenster von einem Ausgangsfenster aus

[3] D.h. Text– und andere Elemente des GUI werden in mehreren Sprachen bzw. länderspezifischen Ausführungen angeboten. Unter Angabe eines sog. *Locale* entscheidet die Anwendung beim Start, welche Darstellung zu realisieren ist.

ermöglicht. Diese Fensterverwaltung war zunächst Bestandteil der Entwicklung von *CoLibry*, wurde jedoch im Laufe der Arbeit für JavaPcteLib adaptiert und erweitert, und schließlich komplett nach dort ausgelagert.

## 4.4 Ausgewählte Teilsysteme

Im folgenden werden die Klassen, welche vorgestellte Entwurfskonzepte realisieren, bereits benannt. Erst im nächsten Kapitel, *Realisierung*, werden konkrete Schnittstellen und Benutzungsbeispiele zu einem Teil dieser Klassen erläutert. Wie allgemein üblich, wird auch *CoLibry* in Form mehrerer *Packages* gestaltet, welche die funktionale Aufteilung widerspiegeln. Der volle Paketname ergibt sich durch Konkatenierung des einzelnen Paketnamens an den Pfad der Anwendung:

`de.uniSiegen.informatik.pi.applications.colibry`.

### 4.4.1 Suchsystem

*CoLibry* bietet Funktionalitäten zur Stichwort– und Facettensuche, sowie zur Expertensuche mittels NTT–Anfragen an. Im Rahmen dieser Arbeit wird ein Suchsystem realisiert, welches das Einbinden verschiedenartiger Suchalgorithmen ermöglicht. Dazu muß die Eingabe der Suchanfrage von der Verarbeitung, und diese wiederum von der Ausgabe der Ergebnisse entkoppelt werden. Ein Suchalgorithmus soll eine Anfrage als Eingabe erhalten, und daraus eine Ausgabe erzeugen, mehr muß er über seine Umgebung nicht wissen. Algorithmen, die das vorgesehene Format einhalten, können dann problemlos gegeneinander ausgetauscht werden.

Der Kontrollfluß einer Suchanfrage wird gesteuert von dem sog. *Search Mediator* (Klasse `search.SearchMediator`). Ein Mediator ist nach Gamma *et al.* ein Entwurfsmuster mit folgender Definition:

> „[The Mediator defines] an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly [...]“ [GHJ95] (p.273)

Der Mediator steuert also die anderen Elemente des Suchsystems an, und vermittelt Informationen zwischen Klassen. Eine Suchanfrage des Benutzers erzeugt damit folgenden internen Verarbeitungsablauf (s. Abbildung 4.8):

Die Anfrage wird mittels Betätigung eines Buttons abgeschickt. Sie kann dabei aus einem oder mehreren Schlüssel–Wert–Paaren bestehen[4] (Klasse `search.SearchRequest`). Diese Paare werden von der zum Button gehörenden *Action* (Klasse `search.SearchAction`) von dem Suchformular (Klasse

[4] Bei der Schlüssselwortsuche sind dies Paare von Attributnamen und –werten, bei der Facettensuche Paare von Facetten und Werten

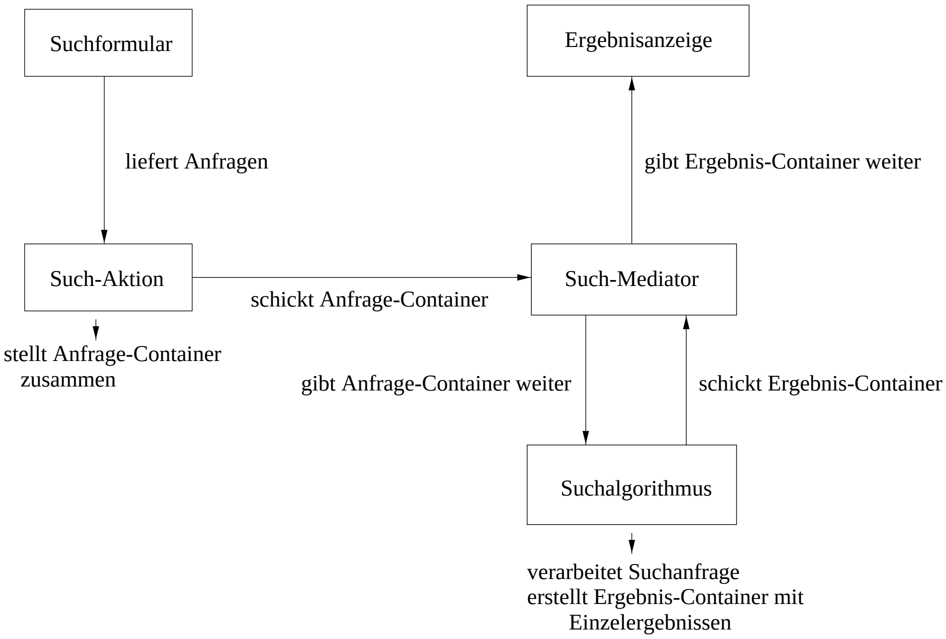


Abbildung 4.8: Kontrollfluß im Teilsystem Suche

`search.SearchRequestPanel`) erfragt, und in einem Suchanfrage–Container gesammelt (Klasse `search.SearchRequestContainer`).

Der Anfrage–Container wird von der Aktion an den Mediator geschickt, und von diesem an den Suchalgorithmus weitergegeben. Der Algorithmus schlüsselt die Anfrage wieder auf, und startet damit eine Suche. Die Ergebnisse der Suche werden vom Algorithmus in einzelne Ergebnisobjekte (Klasse `search.SearchResult`) und dann in den Ergebniscontainer gekapselt (Klasse `search.SearchResultContainer`). Dabei wird in einem einzelnen Ergebnis nicht nur die gefundene Komponente angegeben: Die Schlüsselworte, welche in der Beschreibung gefunden wurden, sowie die Anzahl der bisherigen Benutzungen der Komponente werden ebenfalls zurückgeliefert, um eine Vor–Evaluierung des Suchergebnisses zu ermöglichen, ohne daß der Benutzer sich die Beschreibung selber ansehen muß.

Der Ergebniscontainer wird an den *Search Mediator* zurückgeschickt, der ihn an die Ausgabe weiterleitet (Klasse `search.SearchResultPanel`). Nebenher garantiert der Mediator noch, daß das System keine Anfragen entgegennehmen kann, solange eine Suche läuft, indem es die Suchaktion außer Kraft setzt. Teile der konkreten Schnittstelle werden in Abschnitt 5.3 wiedergegeben.

### 4.4.2 Datenmanagement

Obwohl mit JHPcte in Verbindung mit JavaPcteLib bereits ein leistungsstarker *Java*–Aufsatz für H-PCTE zur Verfügung steht, ist es doch notwendig, eine eigene Schnittstelle anzubieten. Dies resultiert aus dem Wunsch, visuelle Anzeigen von Beschreibungen „on the fly“ zu generieren. Öffnet der Benutzer also eine Beschreibungsanzeige, soll intern ein Anzeigefenster generiert werden, welches die Daten aus dem OMS ausliest und darstellt. Das ist soweit nicht problematisch, doch erfordern die vorhandenen Schnittstellen, daß Informationen zu u.a. Attribut– und Linktypen eines auszulesenden H-PCTE–Objektes bekannt sein müssen. Dies ist offensichtlich nicht der Fall, wenn das Objekt (die Komponentenbeschreibung) erst im Verlauf der Sitzung durch eine Benutzeraktion bekannt wird.

Ebenfalls problematisch ist das Anlegen neuer Beschreibungen, in dessen Verlauf neue H-PCTE–Objekte erzeugt werden. Diese Objekte brauchen Zugriffsinformationen in Form von Einträgen in ihre *Access Control List (ACL)*. Auch diese benötigten Einträge müssen ermittelt werden können.

Das Problem wird durch Bereitstellen eines *Deskriptorensystems* gelöst, welches statische Informationen zu einzelnen Objekten kapselt und dem Generierunssystem zur Verfügung stellt.

#### Das Deskriptorensystem

Ein *Deskriptor* ist ein Element, welches statische Information über einen H-PCTE–Objekttyp speichert, und mittels seiner Schnittstelle nach außen anbietet. Diese Information umfaßt Angaben zu den Links und Attributen des Objekttyps, sowie Angaben zu Einträgen in die ACL des Objekts, welche beim Anlegen eines neuen Objekts dieses Types benötigt werden.

Für ACL–Einträge steht in JHPcte bereits ein Datentyp zur Verfügung (`JHPcte.PcteAclEntry`). Links und Attribute hingegen werden durch neue Klassen modelliert. Die Klasse für Links (`data.Link`) enthält Angaben zu Linkname, Linktyp (damit ist gemeint: mit Schlüsselattribut oder ohne, also einfach oder mehrfach) und Name des Zielobjektes. Die Klasse für Attribute (`data.Attribute`) enthält dementsprechend Attributnamen, Typnamen und Formatangaben zur Anzeige.

Die Klassen `Link` und `Attribute` bieten außerdem Methoden zum dynamischen Auslesen der angegebenen H-PCTE–Objekt–Referenz an, welche eine Auflistung der Zielobjekte einerseits, und der Attributwerte andererseits, zurückliefern.

Zusammengefaßt werden alle Deskriptoren im *Deskriptorensystem*, welches eine Schnittstelle zum Zugriff auf einzelne Deskriptoren anbietet. Identifiziert wird ein Deskriptor über seinen Namen, der durch Zusammenfügen des Namens des entsprechenden H-PCTE–Objektes und der Bezeichnung „Descriptor“ entsteht. `administrative_info` wird somit zum Klassennamen `AdministrativeInfoDescriptor`. Um die verschiedenen Dokumentsorten unterscheiden zu können, wird noch die Sorte gefolgt von einem Unterstrich vorangestellt, so daß ein vollständiger Bezeichner z.B. so aussieht: `Function_AdministrativeInfoDescriptor`.

### Zugriffskontrolle

Durch den Einsatz von ACLs wird ein Teil der Zugriffskontrolle innerhalb der Anwendung realisiert. Der Start–Prozeß von *CoLibry* erhält eine Initial–ACL, welche damit für alle neuangelegten Objekte gültig ist. Jedem Datenbank–Objekt wird außerdem eine eigene ACL mit einem oder mehreren Einträgen durch das Deskriptorsystem zugewiesen, welche die Initial–ACL überschreibt. Attribute des Objekts werden mittels der in JavaPcteLib enthaltenen Klasse `pcte.PcteDefaultAttribute` gekapselt; diese Klasse enthält auch eine Sicht (oder *View*) auf das Datum. Die ACL bestimmt nun, ob die Sicht, also das graphische Anzeigeelement, modifizierbar oder nicht, je nachdem, welche Rechte der Benutzer auf das Objekt besitzt.

Die Zuordnung des Benutzers von *CoLibry* zu einer der beiden Benutzergruppen entscheidet, welche Aktionen überhaupt von ihm während des Programmablaufes ausgelöst werden können (durch Buttons, Menüeinträge, etc. ). Dafür werden alle relevanten Aktionen entsprechend verwaltet (s. Abschnitt 4.4.3).

### Generierung von Anzeigeelementen

Die graphische Anzeige einer Beschreibung, sowie des Benutzungslogs, soll automatisch generiert werden. Das Deskriptorensystem stellt dafür alle benötigten Informationen zur Verfügung. Die Klasse, welche für die Generierung zuständig ist, heißt `data.PanelGenerator`. Der Panel–Generator erhält als Parameter eine Objektreferenz, und stellt daraus eine graphische Anzeige in Form eines *Java–Panels* zusammen.

Attribute werden ausgelesen, und als beschriftete Textelemente angezeigt. Einfache Links werden weiterverfolgt, und die Attribute der Zielobjekte gleichartig verarbeitet. Bei schlüsselattributierten Linktypen, d.h. solchen, zu denen es zu einem beliebigen Zeitpunkt keine, eine, oder auch mehrere Instanzen geben kann, werden gesondert behandelt: Es wird eine Liste der vorhandenen Links angelegt und angezeigt, und durch Rekursion werden wieder eigene Panels erzeugt. Dies ermöglicht die differenzierte Anzeige von Beschriftungen in einem geteilten Fenster, wie sie in Kapitel 5 näher beschrieben wird.

Insgesamt enthält der PanelGenerator damit „hart“ einprogrammierte Verhaltensregeln zur Generierung. Das bedeutet, daß kein beliebiges Panel erzeugt werden kann, und durch die Abhängigkeit von der unterliegenden

Datenstruktur[5] reagiert der Generator entsprechend empfindlich auf Änderungen derselben. Dies läßt sich jedoch nicht vermeiden, da eine ganz bestimmte Form der Anzeige gewünscht ist.

### 4.4.3 Ereignis– und Aktionsmanagement

JavaPcteLib bietet ein im Vergleich zu Standard *Java* stark erweitertes Aktionsmanagement an. Durch Einsatz des sog. *ActionManagers* der Bibliothek können *Actions* komfortabel verwaltet, und insbesondere verschiedenen Gruppen zugewiesen werden. Aktionen können außerdem einzeln oder gruppengebunden aktiviert und deaktiviert werden. Dies wird bei der Zugriffskontrolle eingesetzt.

Ebenfalls erweitert ist die Funktionalität der eingesetzten *Actions*. Standard *Java Actions* kapseln Programmaktionen, welche durch eine Benutzereingabe angestoßen werden, sowie Angaben zu Beschriftungen und Icons der zugeordneten graphischen Elemente. Durch Hinzufügen der *Action* zu Pulldown– und Popup–Menüs sowie der Toolbar werden diese graphischen Elemente automatisch erzeugt. Deren Anwahl durch den Benutzer löst eine Nachricht an die *Action* aus, welche dann die Ausführung übernimmt.

Die Erweiterung in JavaPcteLib, `RefinedAbstractAction`, verfügt über eine sauberere Schnittstelle in Bezug auf Eigenschaften der Graphikelemente: Beschriftung, Tooltip, aktives und inaktives Icon, *Mnemonics* und Acceleratoren (d.h. Tastaturkürzel) können explizit zugewiesen werden. Außerdem ermöglicht sie die Erzeugung von *Buttons* (vom Typ `ActionButton`), etwas, was unter Standard *Java* nicht möglich ist. Entsprechende Erweiterungen zur standardisierten Anzeige von Mnemonics etc. in Menüs und Toolbar sind ebenfalls vorhanden, sowie spezialisierte *Action*–Klassen wie die `AbstractToggleAction`, mit der ein Umschaltelement erzeugt wird.

Damit können alle aktionsauslösenden Elemente von *CoLibry* mit Hilfe von *Actions* realisiert werden. Beim Öffnen eines neuen Fensters werden die zugehörigen *Actions* angelegt bzw. aus dem *ActionManager* geholt, und damit Buttons, Menüeinträge, etc. erzeugt.

[5] Der Generator verläßt sich z.B. darauf, daß von einem Zielobjekt eines mehrwertigen Links nicht wieder mehrwertige Links ausgehen, die Rekursiontiefe beträgt also maximal zwei.

# Kapitel 5

# Realisierung

## 5.1 Allgemeines zur Implementierung

Dieses Kapitel erläutert die Realisierung ausgewählter Bereiche des Entwurfs. Insbesondere wird dabei auf jene Teilsysteme näher eingegangen, welche schon im vorigen Kapitel ausführlicher beschrieben wurden.

Zwei eigenständige Programme wurden neben *CoLibry* erstellt, welche vor seinem Einsatz gestartet werden müssen: *colibryfiller* setzt die vorhandenen *Java*–Properties–Dateien zur Internationalisierung in das interne Ressourcensystem, welches JavaPcteLib unter H-PCTE realisiert, um. *initusers* legt die Benutzergruppen MANAGER und STAFF, sowie die den Gruppen zugehörigen Nutzer, unter H-PCTE an.

Ansonsten sind alle vorgestellten Funktionen innerhalb von *CoLibry* realisiert. Der vorhandene Code gliedert sich in die folgenden *Packages*:

- `colibry`: Enthält die Basisklassen des Systems, unter anderem `ColibryApplication`, die eigene Erweiterung der JavaPcteLib–*Application*–Klasse, `ColibryException`, oder `ColibryAction`.
- `colibry.actions`: Hier sind die *Actions* zum Öffnen und Schließen von Anzeigen, zum Löschen, Editieren, usw. gesammelt.
- `colibry.data`: Enthält grundlegende Teile des Deskriptorensystems (`DescriptorInterface`, `DescriptorSystem`, `Attribute` und `Link`), sowie den `PanelGenerator`.
- `colibry.data.descriptors`: Hier sind die eigentlichen Deskriptoren abgelegt, zusammen mit einer Basisklasse `Descriptor`.
- `colibry.gui`: Die graphischen Elemente von *CoLibry* wie Fensteranzeigen (z.B. `MainFrame`) sind hier eingeordnet.
- `colibry.interfaces`: Enthält benötigte *Interfaces*.

- `colibry.search`: Enthält die Klassen des Suchsystems (s. Abschnitt 4.4.1 für eine Auflistung).

Generell wurden Klassen immer nach folgender Regel an *Packages* zugewiesen: Ist die Klasse einem bestimmten Bereich eindeutig zuzuordnen, dann gehört sie in das entsprechende *Package* (z.B. `SearchAction` ins *Package* `search`). Kann sie keinem bestimmten Bereich zugordnet werden, wird sie nach der angebotenen Funktionalität eingeordnet, wie z.B. `QuitAction` dem *Package* `actions`.

## 5.2 Graphische Umsetzung

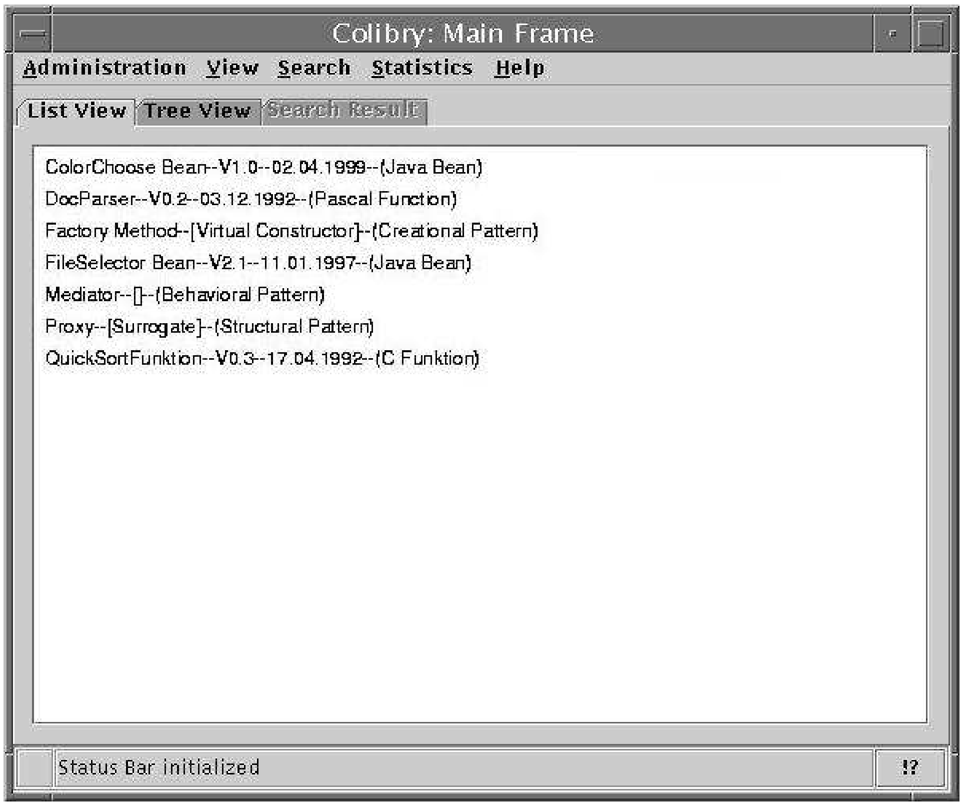


Abbildung 5.1: Hauptfenster von *CoLibry*

*CoLibry* bietet beim Start nach dem Einloggen ein **Hauptfenster**, siehe Abbildung 5.1. Hier finden sich alle angebotenen Funktionalitäten in Form der entsprechenden Menüs und ihrer Einträge:

- **Administration** (**Administration**):

  - Neue Beschreibung anlegen (Create New Description),
  - Beschreibung editieren (Edit Description),
  - Beschreibung löschen (Delete Description),
  - Neue Kategorie anlegen (Create New Category),
  - Kategorie löschen (Delete Category), und
  - Beenden (Quit).

- **Anzeigen** (**View**):
  - Beschreibung anschauen (Show Description),
  - Benutzungslog anschauen (Show Usage Log),

- **Suche** (**Search**):
  - Stichwort– /Facettensuche (Keyword/Facet Search),
  - Expertensuche (Expert Search).

- **Statisitik** (**Statistics**):
  - Statistik anschauen (Show Statistics).

- **Hilfe** (**Help**):
  - Inhalt (Contents),
  - Info (About),
  - Version (Release Notes).

Der Benutzer kann den Anzeigebereich zwischen drei *Tabs* umschalten: Listenansicht der in der Bibliothek enthaltenen Komponenten, Baumansicht (nicht realisiert), und Suchergebnisanzeige. Mittels eines Popup–Menüs, welches bei Rechtsklick nach Auswahl einer Komponente erscheint, kann der Benutzer ebenfalls Anzeige und Editieren von Beschreibung und Benutzungslog starten.

Im unteren Bereich des Hauptfensters befindet sich die Statuszeile, welche mit Hilfe von JavaPcteLib realisiert ist. Hier werden (Fehler–)Meldungen angezeigt, sowie Auskunft über den Zustand der Anwendung gegeben. Eine entsprechende Statuszeile erhalten auch die Anzeigefenster für Beschreibung und Benutzungslog.

Das **Beschreibungsfenster** (Abbildung 5.2) wird durch Anwahl einer Komponente und Starten der Aktionen Anschauen bzw. Editieren geöfnet. Anschauen ermöglicht nur ein Ansehen, kein Editieren der einzelnen Attribute, durch Anwahl des Buttons *Editieren* kann jedoch auch von hier aus eine Anzeige mit Editiermöglichkeit erhalten werden (sofern der Benutzer das Recht zum Modifizieren besitzt).


Abbildung 5.2: Anzeigefenster für eine Beschreibung

Die logischen Bereiche einer Komponentenbeschreibung werden in einzelnen Tabs angezeigt. Wie in Abschnitt 4.4.2 erläutert, werden mehrwertige Links durch Listen angezeigt. Neben diesen Listen finden sich Buttons, mit denen ein weiterer Listeneintrag erzeugt, oder ein vorhandener gelöscht werden kann. Durch Auswahl eines Listeneintrags öffnet sich im rechten Bereich des Anzeigefensters eine Detailanzeige des Inhalts des Zielobjektes.

Änderungen im Inhalt können mit dem Button *Ausführen* (*Apply*) an das OMS abgeschickt, bzw. mit *Zurücksetzen* (*Reset*) rückgängig gemacht werden. Auch von hier aus kann mit *Benutzungslog anschauen* (*Show Usage Log*) die Loganzeige gestartet werden. *Abbrechen* (*Cancel*) schließt hier, wie auch an anderen Stellen der Anwendung, das aktuelle Fenster.

Ähnlich gestaltet sich die Anzeige für das **Benutzungslog** einer Komponente (Abbildung 5.3): Links befindet sich eine Liste der Einträge, und dazu rechts eine Darstellung des Eintragsinhaltes. Buttons zum Anlegen eines neuen Eintrags, bzw. zum Löschen vorhandenen Einträge zur Verkürzung des Logs durch den Manager, fehlen noch. Ein Eintrag kann übrigens nur von dem Benutzer modifiziert werden, der ihn angelegt hat, und nur durch den Manager der Bibliothek gelöscht werden. Dies wird durch entsprechende ACL-Einträge an dem Logeintrags–Objekt gewährleistet.

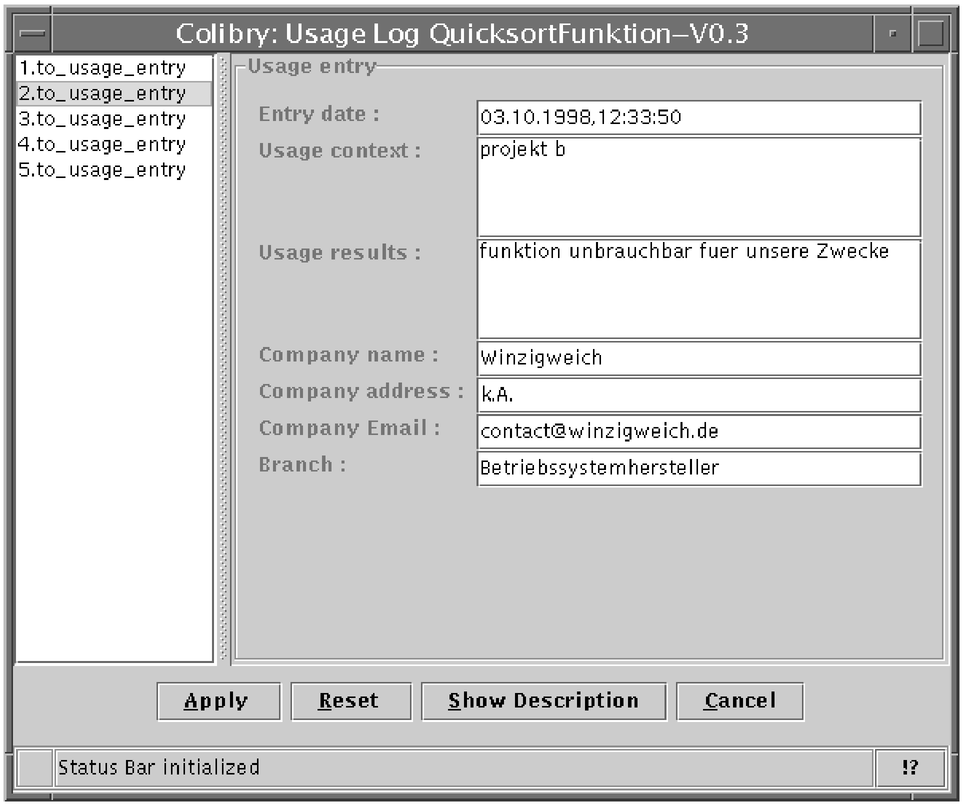


Abbildung 5.3: Anzeigefenster für das Benutzungslog einer Komponente

Als weiteres graphisches Element, welches im Rahmen dieser Arbeit realisert worden ist, findet sich das **Suchfenster** mit den Eingabeformularen für Schlüsselwort– und Facettensuche. Als Abbildung siehe dazu 5.4. Eine Suche kann mittels des Buttons *Suchen* (*Search*) gestartet werden.

Als Suchwerte für eine Schlüsselwortsuche können Komponentenname, Version, Funktionalität, Domäne, Komponentenkategorie, Programmiersprache, Ausführungsumgebung, Autor, und Hersteller–Firma eingegeben werden. Zwei Entwurfsmuster–spezifische Eingabefelder für Zweck und Arbeitsbereich (s. [GHJ95] für eine Spezifizierung) sind ebenfalls vorhanden.

Bei der Facettensuche können der *Agent* der gesuchten Funktionalität, sowie *Aktion–Objekt*–Paare angegeben werden. Dies sind nur ein paar von vielen möglichen Facetten. Für eine ausführliche Erläuterung zum Thema Facetten siehe [RGI93b].

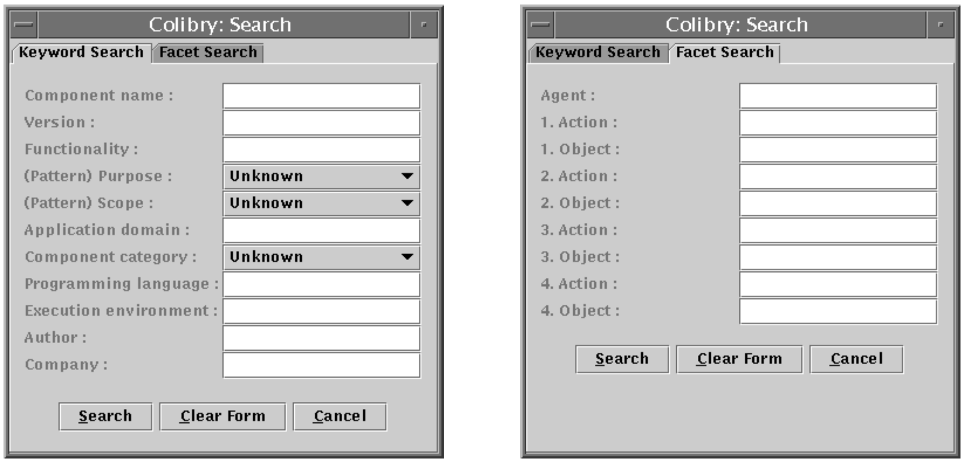


Abbildung 5.4: Suchfenster für die Schlüsselwort– und Facettensuche

## 5.3 Schnittstellen und Benutzung

Hier werden einige der Konzepte, auf welche bereits in Abschnitt 4.4 eingegangen wurde, näher betrachtet, und zwar im Hinblick auf ihren Einsatz und die Gestaltung der Schnittstelle. Es wird hier nicht versucht, umfangreichen Quellcode wiederzugeben, vielmehr, ein Gefühl für die Konzeption von *CoLibry* zu vermitteln.

### 5.3.1 Einsatz des Ressourcensystems

Die Klasse `ColibryApplication` übernimmt die Initialisierung des Ressourcensystems (welches in Abschnitt 4.4 vorgestellt wurde) und liefert dieses mit der Methode `getResourceSystem()` zurück. Eine Klasse, welche auf Ressourcen zurückgreift, muß also von der Applikation das Ressourcensystem erhalten, und daraus die gewünschte Ressourcengruppe ermitteln (bestehend aus Bezeichnung und Versionsnummer). Auch das *Locale*[1] muß gesetzt werden; dieses wird sinnvollerweise wieder von der Applikation angeboten. Eine typische Initialisierung und Anwendung zur Erzeugung eines Labels (aus `KeywordSearchRequestPanel`) sieht folgendermaßen aus:

```
resourceSystem = this.getApplication().getResourceSystem();

dataResources = resourceSystem.getResourceGroup("Data", "1");
dataResources.setLocale(this.getApplication().getLocale());

JLabel nameLabel =
   new JLabel(dataResources.getResource("component_name")
              + " : ");
```

[1]Das Locale gibt Land, Sprache und eventuelle Sprachvarianten an.

Die Beschriftung des Labels wird, wie alle visuell dargestellten Textelemente von *CoLibry*, in der entsprechenden Ressourcengruppe abgelegt. Mittels der Methode `getResource(`*`Bezeichner`*`)` wird der Eintrag ausgelesen, und kann, da er im Stringformat vorliegt, direkt an den Konstruktor des Labels übergeben werden.

Dieser Einsatz des Ressourcensystems erzeugt natürlich längeren und vielleicht auch nicht ganz so übersichtlichen Code, als wenn die Texte direkt im Programmcode eingefügt wären. Doch die Möglichkeit, mit nur einem Befehl (dem Austausch des *Locale* nämlich) die Sprache der Anwendung zu wechseln, rechtfertigt diesen erhöhten Schreibaufwand in jedem Fall.

### 5.3.2 Einsatz des Aktionsmanagements

Wie in Abschnitt 4.4.3 bereits erklärt, werden die Steuerungselemente von *CoLibry* durch *Actions* erzeugt. Für die Erzeugung eines Menüs mit Einträgen muß zunächst ein *Java*–Objekt vom Typ `MenuBar` angelegt werden, dem dann Objekte vom Typ `ActionMenu` zugewiesen werden. Im Konstruktor eines Menüs kann dabei das Erscheinungsbild festgelegt werden. Hier wird ein Menü ohne Icons, aber mit Mnemonics, Accelerator, und Tooltip-Text festgelegt (aus `gui.MainFrame`, leicht modifiziert):

```
// add menubar to frame
menubar = new JMenuBar();
frame.setJMenuBar(menubar);

// add menu to menubar
adminMenu =
   new ActionMenu(guiResources.getResource("admin_menu_text"),
                  true, false, true, true, true);
adminMenu.setMnemonic(guiResources.getChar(
                         "admin_menu_mnemonic"));
menubar.add(adminMenu);
```

Um nun einen Menüeinträg mittels einer *Action* erzeugen zu können, muß diese erst aus dem *ActionManager* geholt werden. Außerdem wird für jedes Fenster eine *Action*–Gruppe angelegt, in welcher die *Actions* abgelegt werden:

```
// get ActionManager
ActionManager am = application.getActionManager();

// create action group
String frameID = this.getName();
if(!am.existsGroup(frameID))
   am.createGroup(frameID);

// process new description action
```

```
NewDescriptionAction newDescriptionAction;
if (am.existsAction("newDescriptionAction" + frameID)) {
   newDescriptionAction =
      (NewDescriptionAction)am.getActionObject(
                                    "newDescriptionAction" +
                                    frameID);
}
else {
   newDescriptionAction =
      new NewDescriptionAction(
           application,
           process,
           this,
           guiResources.getResource(
                        "new_description_action_text"),
           guiResources.getChar(
                        "new_description_action_mnemonic"),
           guiResources.getResource(
                        "new_description_action_accelerator"),
           guiResources.getResource(
                        "new_description_action_tooltip"),
           "newDescriptionAction" + frameID);
   am.putActionObject("newDescriptionAction" + frameID,
                      newDescriptionAction, frameID);
}
adminMenu.add(newDescriptionAction);
```

Die *Action* wird durch einen Stringbezeichner und die Konkatenation der Fenster–ID identifiziert. Ist sie dem *ActionManager* schon bekannt, wird eine Referenz auf sie geholt (`getActionObject(`*`Actionbezeichner`*`)`); wenn nicht, wird die *Action* neu angelegt, und im Manager gespeichert: `putActionObject(`*`Actionbezeichner, Gruppenbezeichner`*`)`.

Um einen entsprechenden Button zu erzeugen, schließt sich einfach folgender Code an:

```
ActionButton newDescriptionButton =
  new ActionButton((RefinedAbstractAction)newDescriptionAction,
                   true,
                   true,
                   true);
buttonPanel.add(newDescriptionButton);
```

Der Konstruktor des Buttons erhält eine Referenz des *Action*–Objektes, sowie boolsche Angaben zu seiner visuellen Gestaltung (hier: mit Text, Icon, und Tooltip).

Der obige Codeausschnitt zur Erzeugung einer *Action* sieht zwar recht umfangreich aus, jedoch ist zu bedenken, daß eine *Action* nur einmal angelegt werden muß. Bei ihrer weiterer Benutzung kann sie aus dem *Ac-*

*tionManager* erhalten werden. Insgesamt ermöglicht der Einsatz von *Actions* eine sehr übersichtliche Programmstruktur, und schnelles Anlegen der Steuerungselemente, die andernfalls explizit erzeugt und mit eigenem *Eventhandling* versehen werden müßten.

### 5.3.3 Einsatz des Suchsystems

Wie in Abschnitt 4.4.1 erwähnt, werden Suchanfragen sowie die Ergebnisse einer Suche in jeweils einem Container gesammelt, und an den Mediator geschickt, welcher für die ordnungsgemäße Weiterleitung zuständig ist. Die Klasse `SearchRequest`, die eine einzelne Anfrage, bestehend aus Attribut–Wert–Paar, enthält, besitzt folgende Schnittstelle:

- Konstruktor `SearchRequest(String key, String value)`;
- `String getKey()`;
- `String getValue()`.

Die Klasse `SearchResult` kapselt eine Referenz auf ein Ergebnis–Objekt, sowie Angaben zu der Anzahl der Treffer in der Beschreibung, und eine Auflistung der Treffer:

- Konstruktor `SearchResult(PcteObjectReference reference, String hits, int hitcount)`;
- `PcteObjectReference getReference()`;
- `String getKeywordHits()`;
- `int getNrOfHits()`.

Die Containerklassen verfügen über einander ähnliche Schnittstellen. Zunächst der Container für Suchanfragen:

- Konstruktor `SearchRequestContainer`, legt einen leeren Container an;
- `void addRequest(SearchRequest request)`;
- `boolean hasNextRequest()`;
- `SearchRequest getNextRequest()`.

Die Schnittstelle des Containers für Suchergebnisse gestaltet sich entsprechend:

- Konstruktor `SearchResultContainer`, legt einen leeren Container an;

- `void addResult(SearchResult result);`
- `boolean hasNextResult();`
- `SearchResult getNextResult();`
- `int NrOfResults()`.

Die Methode zur Ermittlung der Anzahl der Ergebnisse dient dazu, im Suchergebnis–Panel, wie in Suchmaschinen üblich, die Anzahl der Treffer am Anfang angeben zu können.

### 5.3.4 Einsatz des Deskriptorensystems

Der grundlegende Aufbau des Deskriptorensystems wurde schon in Abschnitt 4.4.2 erläutert. Zur Erinnerung: H-PCTE–Objekte werden durch Deskriptoren beschrieben, welche wiederum im Deskriptorensystem verwaltet sind. Ein einzelner Deskriptor wird durch seinen Klassennamen definiert, welcher aus Konkatentation des Objektnamens (welchem noch die Dokumentsorte vorangestellt wird) und der Bezeichnung „Descriptor“ entsteht. Das System besitzt folgende Schnittstelle:

- Konstruktor `DescriptorSystem()`, erzeugt ein neues Deskriptorsystem;
- `void initializeSystem(PcteProcess process);`
- `DescriptorInterface getDescriptor(String classname);`
- `boolean hasDescriptor(String classname)`.

Das Deskriptorsystem wird innerhalb der Klasse `ColibryApplication` initialisiert:

```
if (this.descriptorSystem == null) {
   try {
      // create and initialize
      this.descriptorSystem = new DescriptorSystem();
      this.descriptorSystem.initializeSystem(process);
   }
   catch (PcteException ex) {
      mainframe.addMessage(ex.toString());
   }
}
```

Die Initialisierung ist in einen *try–catch*–Block gekapselt, um eventuell auftretende H-PCTE *Exceptions* aufzufangen, deren Meldungen an die Statuszeile des Hauptfensters weitergeleitet werden.

Die Abarbeitung innerhalb von `initializeSystem()` gestaltet sich folgendermaßen (hier nur das Anlegen eines einzelnen Deskriptors):

```
Base_AdministrativeInfoDescriptor base_administrativeInfo =
   new Base_AdministrativeInfoDescriptor(process);
descriptors.put("Base_AdministrativeInfoDescriptor",
                 base_administrativeInfo);
```

Gespeichert werden die Deskriptoren in einem Hashtable mit Namen `descriptors`.

Das Interface `DescriptorInterface` legt fest, wie sich die Schnittstelle eines Deskriptors gestaltet. Zur Erinnerung: Der Deskriptor speichert statische Angaben über Attribute. Links und ACL–Einträge des Objektes.

- `int nrOfAclEntries()`;
- `PcteAclEntry getAclEntryAt(int i)`;
- `int nrOfAttributes()`;
- `Attribute getAttributeAt(int i)`;
- `int nrOfLinks()`;
- `Link getLinkAt(int i)`.

Die Basisklasse aller Deskriptoren, `Descriptor`, legt Standard–ACL–Einträge für die Gruppen MANAGER und STAFF an, wobei dem Manager alle Rechte gewährt werden, dem normalen Benutzer jedoch nur Navigation über und Auflisten von Links, sowie das Lesen von Objekt– und Linkattributen gestattet ist. Deskriptoren von Objekten mit abweichenden Rechten legen eigene ACL-Einträge an. Es folgt der Konstruktor des Basis–Deskriptors:

```
public Descriptor(PcteProcess process)
  throws PcteException {

   // initialize vectors
   attributes = new Vector();
   links = new Vector();
   acl_entries = new Vector();

   // create standard acl entries and store them
   int group_id = this.getGroupId(ColibryConfig.STAFF);
   PcteAclEntry acl_entry =
      new PcteAclEntry(group_id, staff_rights);
   this.acl_entries.addElement(acl_entry);

   group_id = this.getGroupId(ColibryConfig.MANAGER);
   acl_entry =
      new PcteAclEntry(group_id, manager_rights);
   this.acl_entries.addElement(acl_entry);
}
```

Das Anlegen eines ACL–Eintrags erfordert die Angabe eines Gruppenidentifizierers, welcher in der Klasse `ColibryConfig` gespeichert wird. (Diese ist für das Bereitstellen konfigurationeller Elemente wie z.B. Pfadnamen und Sprache der Anwendung zuständig.)

Der Konstruktor eines Deskriptors für das Objekt `administrative_info` der Dokumentsorte *base* sieht nun folgendermaßen aus:

```
public Base_AdministrativeInfoDescriptor(PcteProcess process)
   throws PcteException {

   super(process);

   // create attributes and store them
   Attribute attr =
      new Attribute("component_name",
                    PcteAttributeValue.PCTE_STRING_ATTRIBUTE,
                    Attribute.VIEW_TEXTFIELD);
   this.attributes.addElement(attr);

   // create links and store them
   Link link = new Link("to_author",
                        Link.IS_MULTIPLE_LINK,
                        "author");
   this.links.addElement(link);
   link = new Link("to_company",
                   Link.IS_MULTIPLE_LINK,
                   "company");
   this.links.addElement(link);
   link = new Link("to_component_address",
                   Link.IS_MULTIPLE_LINK,
                   "url_address");
   this.links.addElement(link);
}
```

In diesem Konstruktor werden Objekte des Typs `Attribute` und `Link` angelegt. Die Schnittstelle von `Attribute` ist:

- Konstruktor `Attribute(String name, int type, int format)`;
- `int getAttributeFormat()`;
- `String getAttributeName()`;
- `int getAttributeType()`;
- `DefaultPcteAttribute getAttribute(PcteProcess process, PcteObjectReference actual_objref)`.

Das Format eines Attributes bestimmt seine Darstellung und wird vom Panel–Generator deswegen abgefragt. Die letzte Methode ist im Gegensatz zu den anderen dynamisch, und liefert unter Angabe des Prozesses und einer Objektreferenz den Attributwert des entsprechenden Attributes dieses Objekts.

In der Klasse `Link` beschreibt der *Linktyp* nicht eine der H-PCTE–Kategorien, sondern, ob der Link ein– oder mehrwertig ist. Diese Angabe wird wiederum vom Panel–Generator zur Erzeugung einer korrekten Anzeige benötigt:

- Konstruktor `Link(String name, int type, String targetName);`
- `String getLinkName();`
- `getLinkType();`
- `String getTargetName();`
- `PcteObjectReference[] getTargets(PcteObjectReference objref).`

Mit `getTargets()` wird eine Liste der Zielobjekte der existierenden Links dieses Typs, ausgehend vom angegebenen Objekt, erzeugt.

### 5.3.5 Einsatz des Panel–Generators

Die Hauptmethode des Panel–Generators sieht folgendermaßen aus:

```
getPanel(ColibryApplication application,
         PcteProcess process,
         PcteObjectReference objref,
         String documentSort,
         ListSelectionListener listListener,
         boolean showTitle,
         boolean show_only)
```

Der Generator erhält eine Objektreferenz, sowie die Dokumentsorte mitgeteilt. Da die aufrufende Klasse, welche das erzeugte Panel zur Weiterverarbeitung erhält, nichts über dessen Inhalt weiß, muß sie einen `ListSelectionListener` implementieren, und per Parameter übergeben. Der Listener wird an den erzeugten Listen angemeldet, so daß die aufrufende Klasse (normalerweise `DescriptionFrame` oder `UsageLogFrame`) auf Selektions–Ereignisse der Listen reagieren kann. Zwei boolsche Parameter schließlich geben an, ob das Panel über eine Umrandung mit Titel (enthält Name des Objekttyps) verfügen soll, sowie, ob die Elemente des Panel modifizierbar sein sollen oder nicht.

Der Panel–Generator erzeugt zunächst Anzeigeelemente für die Attribute des angegebenen Objektes. Dann betrachtet er dessen Links; handelt es sich um einwertige, d.h. nicht mit Schlüsselattribut versehene Links, navigiert er zum Zielobjekt, und erfragt dessen Attribute, welche dann ebenfalls ins Panel integriert werden. Bei einem mehrwertigen Link erzeugt er eine Liste der Zielobjekte und zeigt diese an.

Ein Selektierungs–Ereignis, ausgelöst in der aufrufenden Klasse, hat ebenfalls einen Aufruf des Panel–Generators zur Folge, welcher als Objektreferenz das ausgewählte Zielobjekt erhält. Das erzeugt Panel wird von dem Aufrufer an einer passenden Stelle angezeigt, normalerweise auf der rechten Seite des aufgeteilten Anzeigefensters.

Der folgende Quellcode–Ausschnitt zeigt das Anlegen eines *Tabs* innerhalb des Beschreibungsfensters (Klasse `DescriptionFrame`). Die Klasse erhält eine Referenz auf das Objekt `component_description`. Es ermittelt über den Deskriptor alle davon ausgehenden Linktypen, prüft, ob tatsächlich Links zu einem Typ existieren, und legt für jedes der Zielobjekte ein *Tab* an (siehe dazu auch die Schema–Abbildungen in Abschnitt 4.2.2). Dieses wird mit dem von `PanelGenerator` erzeugten Panel gefüllt.

```
for (int i = 0; i < descriptor.nrOfLinks(); i++) {
   link = descriptor.getLinkAt(i);
   [...]
   PcteObjectReference[] link_targets =
      link.getTargets(objref);
   if (link_targets != null) {
      // only one target can exist; get it
      target = link_targets[0];
      tabPanel =
         PanelGenerator.getPanel(
            ((ColibryApplication)this.getApplication()),
            process,
            target,
            documentSort,
            (ListSelectionListener)this,
            false,
            show_only);
      [...]
   }
}
```

Mit Hilfe des Deskriptorensystems können also H-PCTE–Objekte schnell und relativ unkompliziert ausgewertet werden. Das ermöglicht dem Panel–Generator das automatische Erzeugen von Panels, welche sonst mühsam von Hand erzeugt werden müßten (und zwar für jede mögliche Darstellung von Datenbankinhalten eines).

# Kapitel 6

# Zusammenfassung und Ausblick

### Zusammenfassung

Ausgangspunkt dieser Arbeit war der Wunsch, Software–Komponenten, die Bausteine einer erfolgreichen Wiederverwendung, einheitlich zu beschreiben und zu verwalten. Zu diesem Zweck wurde zunächst eine *Komponentenbeschreibungssprache* entwickelt, sowie eine Umsetzung der gewonnenen Strukturierung und Inhalte in das Datenbankschema von H-PCTE, die *Schema Definition Sets*. Darauf aufbauend wurde ein Prototyp einer benutzerfreundlichen Schnittstelle zur Verwaltung sowie zum Durchsuchen einer solchen Sammlung von Beschreibungen entwickelt. Der Prototyp bietet Basisfunktionalitäten, insbesondere eine eigene Schnittstelle zum Datenbankmanagementsystem, um „on the fly“ Anzeigen für Beschreibungen generieren zu können. Die realisierte Schnittstelle bietet ebenfalls eine rein aktionsbasierte Ereignisbehandlung über eine Erweiterung der Standard *Java Actions*, sowie eine zweisprachige Anzeige durch *Internationalisierung.*

### Evaluierung der Arbeit

*CoLibry* ist eine recht komplexe Anwendung, welche von Grund auf konzipiert und realisiert wurde, sowie als Probeanwendung für das Ausprobieren neuentwickelter Konzepte von JavaPcteLib wie z.B. dem Aktionsmanagement und Fenstermanagement benutzt wurde. Dadurch war die Entwicklung streckenweise eng verzahnt mit der von JavaPcteLib, und beide Arbeiten wirkten aufeinander befruchtend.

Im gegenwärtigen Stadium verfügt *CoLibry* über 7 *Packages* mit insgesamt 71 Klassen. Die Anwendung ist bereits beschränkt einsetzbar: Komponentenbeschreibungen können angelegt, modifiziert, und wieder gelöscht werden, desgleichen Kategorien und Einträge ins Benutzungslog. Jedoch wird durch das Fehlen eines automatischen Parsers die Erfassung von Komponenten erschwert, da die Eingabe von Hand recht mühselig ist.

### Ausblick

Priorität bei der weiteren Entwicklung von *CoLibry* werden zunächst die Erstellung eines Parsers zur automatischen Generierung von Beschreibungen aus vorhandener *javadoc*–Dokumentation, sowie die Vervollständigung der Funktionen von *CoLibry* im Bereich Suche, Statistik, und Anzeige haben.

Ebenfalls anvisiert ist das Parsen von Dokumentation im XML–Format, um damit eine der momentan wichtigsten Dokumentbeschreibungssprachen zu unterstützen. Zu prüfen ist auch, ob das Bereitstellen einer äußeren Schnittstelle der Beschreibungen in einer HTML– oder SGML–Umsetzung des *Basic Interoperability Data Models (BIDM)*, wie es von der *Reuse Library Interoperability Group (RIG)* propagiert wird, sinnvoll wäre (siehe dazu [BDH96] und Abschnitt 2.2.3).

In langer Sicht sollen nicht nur die Beschreibungen von Komponenten, sondern auch die Komponenten selber gespeichert und verwaltet werden. Dafür werden Änderungen und Ergänzungen der Schemata und des Programmcodes notwendig werden. Überhaupt kann das momentan vorliegende Produkt nicht als ein Endprodukt, vielmehr als ein frühes Zwischenstadium betrachtet werden, welches seine Praxistauglichkeit im Einsatz noch beweisen muß. Durch seine Konzeption im Hinblick auf Erweiterbarkeit und Wiederverwendbarkeit sollte *CoLibry* neu hinzukommenden Anforderungen allerdings gewachsen sein.

# Literaturverzeichnis

[BDH96] Shirley Browne, Jack Dongarra, Kay Hohn and Tim Niesen. *Software Repository Interoperability.* Technical report UT-CS-96-329, July 1996.

[BDG95] Shirley Browne, Jack Dongarra, Stan Green, Eric Grosse, Keith Moore, Theresa Pepin, Tom Rowan and Reed Wade. *Location–Independent Naming for Virtual Distributed Software Repositories.* 1995.

[ECM93] European Computers Manufacturers Association. *Portable Common Tool Environment – Abstract Specification (Standard ECMA–149).* Geneva, June 1993.

[EMT98] Michel Ezran, Maurizio Movisio and Colin Tully. *Practical Software Reuse: the essential guide.* 1998.

[F96] David Flanagan. *Java in a Nutshell. A Desktop Quick Reference for Java Programmers.* O'Reilly & Associates, Inc., 1996.

[GHJ95] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. *Design Patterns. Elements of Reusable Object–Oriented Software.* Addison–Wesley, 1995.

[G90] Simon Gibbs. *Querying large class collections.* Object management, Centre Universitaire d'Informatique, University of Geneva, July 1990.

[H94] O. Haase. *NTT – Eine mengenorientierte, algebraische Abfragesprache für PCTE.* Universität–GH Siegen, Internes Memorandum 94/3, Oktober 1994.

[He95] A. Henrich. *P-OQL: an OQL–Oriented Query Language for PCTE.* In Proceedings of the Seventh Conference on Software Engineering Environments, pp. 48–60, Netherlands, 1995. IEEE Computer Society Press.

[HK96] O. Haase and U. Kelter. *NTT – A Query Algebra for PCTE.* Technical Report 96/2, Universität–GH Siegen, D-57068 Siegen, 1996.

[JC92] Jun-Jang Jeng and Betty H. C. Cheng. *Using Automated Reasoning Techniques to Determine Software Reuse.* Int'l J. of Soft. Eng. and Knowledge Eng., Vol. 2, No. 4, pp. 523-546, Dec. 1992.

[JC93] Jun-Jang Jeng and Betty H. C. Cheng. *Using Analogy and Formal Methods for Software Reuse.* Proc. of IEEE 5th Int'l Conference on Tools with AI, pp. 113-116, November 1993.

[JC95a] Jun-Jang Jeng and Betty H. C. Cheng. *Specification Matching for Software Reuse: A Foundation.* Proc. of ACM Symposium on Software Reuse, Seattle, Washington, pp. 97-105, 1995.

[JC95b] Jun-Jang Jeng and Betty H. C. Cheng. *Reusing Analogous Components.* Technical report, MSU-CPS-94-28, April 1994 (Revised June 1995).

[K92] Udo Kelter. *H-PCTE – A high performance object management system for system development environments.* Proc. COMPSAC Illinois, September 23–25, pp. 45–50. IEEE Press, 1992.

[K99] Udo Kelter. *Einführung in H-PCTE.* Universität Siegen, 1999. Siehe auch:
http://pi.informatik.uni-siegen.de/kelter/lehre/99w/hpcte/skripten.html

[KMP98] U. Kelter, M. Monecke, und D. Platz. *Realisierung von verteilten Editoren in Java auf Basis eines aktiven Repositories.* In Proc. JIT'98 – Java–Informations–Tage 1998, Frankfurt, Nov. 1998. Springer.

[M98] Marc Monecke. *JHPcte – Eine Java–Schnittstelle für H-PCTE.* Universität Siegen, 1998.

[RG95] Maria del Rosario Girardi. *Classification and Retrieval of Software through their Description in Natural Language.* Ph. D. Thesis No. 2782, Computer Science Department, University of Geneva, 1995.

[RGI93a] Maria del Rosario Girardi and Bertrand Ibrahim. *An approach to improve the effectiveness of software retrieval.* Technical report. Also in Proceedings of the 3rd Irvine Software Symposium, Ed. Debra J. Richardson and Richard N. Taylor, University of California, Irvine, California, April 30, 1993, pp 89-100.

[RGI93b] Maria del Rosario Girardi and Bertrand Ibrahim. *A software reuse system based on natural language specifications.* Technical report. Also in Proceedings of the 5th International Conference on Computing and Information, Ed. Osman Abou-Rabia, Carl K. Chang and Waldemar W. Koczkodaj, IEEE Computer Society Press, Sudbury, Ontario, Canada, May 27-29, 1993, pp 507-511.

# Anhang A

# Glossar

Es folgt eine Auflistung und Erläuterung häufig im Text auftauchender Abkürzungen:

**ACL** *Access Control List.* In PCTE wird jedem Prozeß und jedem Datenbankobjekt eine ACL zugewiesen. Die Einträge darin, die sog. *ACL Entries*, legen jeweils Rechte für eine bestimmte Benutzergruppe fest.

***CoLibry*** **Co***mponent* **Lib***rar***y**. Die in dieser Arbeit entwickelte Schnittstelle zur Verwaltung von Komponentenbeschreibungen.

**DBMS** *Data Base Management System* oder auch *Datenbankmanagementsystem.* Software, über welche Anwendungen auf eine (konventionelle) *Datenbank* zugreifen können.

**GUI** *Graphical User Interface.* Die graphische Benutzerschnittstelle einer Anwendung.

**H-PCTE** Partielle, hauptspeicherbasierte Umsetzung der PCTE–Spezifikation.

**NTT** *Non–navigating Tool Tongue.* Eine mengenorientierte Abfragesprache unter H-PCTE.

**OMS** *Object Management System* oder *Objektmanagementsystem.* Wie DBMS, bezieht sich im Gegensatz dazu aber auf objektorientierte Datenbanken.

**PCTE** *Portable Common Tool Environment.* ISO– und ECMA–Standard eines Integrationsrahmens für Software–Entwicklungsumgebungen.

# Anhang B

# Die *Schema Definition Sets* von *CoLibry*

```
---------------------------------------------------------
--- SDS library
---------------------------------------------------------
--- Autor: Chantal Keller
--- Stand: 29.09.99
---------------------------------------------------------
--- Contents:
---   Attribute types:
---     description_nr, category_nr
---   Object types:
---     home, component_library, component_description,
---     category_list, category
---
---------------------------------------------------------

sds library:

---------------------------------------------------------
--- IMPORTS ---------------------------------------------
---------------------------------------------------------

import objecttype       hpcte-home;
import objecttype       system-object;
import attribute        system-system_key;

---------------------------------------------------------
--- ATTRIBUTE TYPES -------------------------------------
---------------------------------------------------------

attribute description_nr        : natural;
attribute category_nr           : natural;

---------------------------------------------------------
--- OBJECT TYPES ----------------------------------------
---------------------------------------------------------
```

```
--- HOME ----------------------------------------------

extend objecttype home
with
  component
        to_component_library:
                composition link to component_library;
        to_category_list:
                composition link to category_list;
end home;

--- COMPONENT_LIBRARY -----------------------------------

objecttype component_library : child type of object
with
  component
        to_component_description:
                composition link (description_nr) to
                        component_description;
end component_library;

--- COMPONENT_DESCRIPTION -------------------------------

objecttype component_description : child type of object
with
end component_description;

--- CATEGORY_LIST ----------------------------------------

objecttype category_list: child type of object
with
  component
        to_category:
                composition link (category_nr) to
                        component_category;
end category_list;

--- COMPONENT_CATEGORY -----------------------------------

objecttype component_category: child type of object
with
  attribute
        category_name           : string;
        document_sort           : enumeration (
                                          FUNCTION,
                                          CLASSGROUP,
                                          PATTERN);
        -- saves of which internal document sort this
        -- category is
end component_category;

end library;
```

```
--------------------------------------------------------
--- SDS misc
--------------------------------------------------------
--- Autor: Chantal Keller
--- Stand: 19.04.00
--------------------------------------------------------
--- Contents:
---   Attribute types:
---
---   Object types:
---     company, author, library_user,
---     url_address
--------------------------------------------------------

sds misc:

--------------------------------------------------------
--- IMPORTS --------------------------------------------
--------------------------------------------------------

import objecttype       hpcte-home;
import objecttype       security-user;
import objecttype       system-object;
import attribute        system-system_key;

--------------------------------------------------------
--- ATTRIBUTE TYPES ------------------------------------
--------------------------------------------------------

--------------------------------------------------------
--- OBJECT TYPES ---------------------------------------
--------------------------------------------------------

--- COMPANY --------------------------------------------

objecttype company : child type of object
with
  attribute
        company_name            : string;
        company_address         : string;
        branch                  : string;
        company_email_address   : string;
  link
        company_url:
                existence link to url_address;

  --- used only for usage log entry
  link
        .using_employee:
                existence link to library_user;
end company;

--- AUTHOR --------------------------------------------
```

```
objecttype author : child type of object
with
  attribute
        first_name              : string;
        surname                 : string;
        email_address           : string;
  link
        homepage:
                existence link to url_address;
        employer:
                existence link to company;
end author;

--- LIBRARY_USER ---------------------------------------

objecttype library_user : child type of user
with
  attribute
        user_first_name                 : string;
        user_surname                    : string;
        user_email_address              : string;
end library_user;

--- URL_ADDRESS ----------------------------------------

objecttype url_address : child type of object
with
  attribute
        url                     : string;
        reliability             : natural;
end url_address;

end misc;
```

```
-------------------------------------------------------
--- SDS base
-------------------------------------------------------
--- Autor: Chantal Keller
--- Stand: 17.09.99
-------------------------------------------------------
--- Contents:
---     those specifications that are common to all
---     categories of component descriptions
---
---   Attribute types:
---     company_nr, author_nr, address_nr, category_nr,
---     class_nr, usage_entry_nr
---   Object types:
---     component_description, administrative_info,
---     categorizing_info, contents_info, classification,
---     usage_log, usage_entry
---
-------------------------------------------------------

sds base:

-------------------------------------------------------
--- IMPORTS -------------------------------------------
-------------------------------------------------------

import objecttype       hpcte-home;

import objecttype       system-object;
import attribute        system-system_key;

--- IMPORTS FROM SDS MISC -----------------------------

import objecttype       misc-user;
import objecttype       misc-author;
import objecttype       misc-url_address;
import objecttype       misc-company;

--- IMPORTS FROM SDS LIBRARY --------------------------

import objecttype       library-component_description;
import objecttype       library-component_category;

-------------------------------------------------------
--- ATTRIBUTE TYPES -----------------------------------
-------------------------------------------------------

attribute company_nr            : natural;
attribute author_nr             : natural;
attribute address_nr            : natural;
attribute category_nr           : natural;
attribute class_nr              : natural;
attribute usage_entry_nr        : natural;
```

```
-------------------------------------------------------
--- OBJECT TYPES --------------------------------------
-------------------------------------------------------

--- COMPONENT_DESCRIPTION -----------------------------

extend objecttype component_description
with
  component
        to_administrative_info:
                composition link to administrative_info;
        to_categorizing_info:
                composition link to categorizing_info;
        to_contents_info:
                composition link to contents_info;
        to_usage_log:
                composition link to usage_log;

end component_description;

-- ADMINISTRATIVE_INFO --------------------------------

objecttype administrative_info : child type of object
with
  attribute
        component_name          : string;
  link
        to_company:
                existence link (company_nr) to company;
        to_author:
                existence link (author_nr) to author;
        to_component_address:
                existence link (address_nr) to url_address;

end administrative_info;

--- CATEGORIZING_INFO ---------------------------------

objecttype categorizing_info : child type of object
with
  link
        of_category:
                reference link to component_category;
end categorizing_info;


--- CONTENTS_INFO -------------------------------------

objecttype contents_info: child type of object
with
  component
        to_classification:
                composition link (class_nr) to
                        classification;
```

```
end contents_info;

--- CLASSIFICATION -----------------------------------

objecttype classification: child type of object
with
  attribute
        key                     : string;
        value                   : string;
end classification;

--- USAGE_LOG ----------------------------------------

objecttype usage_log: child type of object
with
  attribute
        nr_of_uses              : natural;
  component
        to_usage_entry:
                composition link (usage_entry_nr) to
                usage_entry;
end usage_log;

--- USAGE_ENTRY --------------------------------------

objecttype usage_entry: child type of object
with
  attribute
        entry_date              : time;
        usage_context           : string;
        usage_results           : string;
  link
        to_using_company:
                existence link to company;
end usage_entry;

end base;
```

```
----------------------------------------------------
--- SDS function
----------------------------------------------------
--- Autor: Chantal Keller
--- Stand: 19.04.00
----------------------------------------------------
--- Contents:
---     extends SDS base with function-relevant parts
---
---   Attribute types:
---      related_nr, revrelated_nr
---   Object types:
---     description_links
---
----------------------------------------------------

sds function:

----------------------------------------------------
--- IMPORTS ----------------------------------------
----------------------------------------------------

import objecttype       hpcte-home;

import objecttype       system-object;
import attribute        system-system_key;

--- IMPORTS FROM SDS MISC --------------------------

import objecttype       misc-user;
import objecttype       misc-author;
import objecttype       misc-url_address;
import objecttype       misc-company;

--- IMPORTS FROM SDS LIBRARY -----------------------

import objecttype       library-component_description;
import objecttype       library-component_category;

---IMPORT FROM SDS BASE ----------------------------

import objecttype       base-administrative_info;
import objecttype       base-categorizing_info;
import objecttype       base-contents_info;
import objecttype       base-classification;
import objecttype       base-usage_log;
import objecttype       base-usage_entry;

----------------------------------------------------
--- ATTRIBUTE TYPES --------------------------------
----------------------------------------------------

attribute related_nr           : natural;
attribute revrelated_nr        : natural;
```

```
----------------------------------------------------
--- OBJECT TYPES -----------------------------------
----------------------------------------------------

--- COMPONENT_DESCRIPTION --------------------------

extend objecttype component_description
with
  link
        next_version:
                reference link to component_description
                reverse previous_version
                with
                  attribute
                        changes_log     : string;
        end next_version;

        previous_version:
                reference link to component_description
                reverse next_version;

end component_description;

-- ADMINISTRATIVE_INFO -----------------------------

extend objecttype administrative_info
with
  attribute
        version                 : string;
        version_date            : time;
        origin                  : enumeration (
                                        UNKNOWN,
                                        MADE,
                                        BOUGHT);
end administrative_info;

--- CATEGORIZING_INFO ------------------------------

extend objecttype categorizing_info
with
  attribute
        direction               : enumeration (
                                        UNKNOWN,
                                        HORIZONTAL,
                                        VERTICAL);
        application_domain      : string;
        executability           : enumeration (
                                        UNKNOWN,
                                        YES,
                                        NO);
end categorizing_info;

--- CONTENTS_INFO ----------------------------------
```

```
extend objecttype contents_info
with
  attribute
        short_description       : string;
        programming_language    : string;
        interface               : string;
  component
        to_description_links:
                composition link to description_links;
end contents_info;

--- DESCRIPTION_LINKS -------------------------------

objecttype description_links: child type of object
with
  attribute
        structure_description           : string;
  link
        uses:
                reference link (related_nr) to
                            description_links
                reverse is_used
                with
                  attribute
                        co_usage        : enumeration (
                                                UNKNOWN,
                                                REQUIRED,
                                                FREE);
        end uses;

        is_used:
                reference link (revrelated_nr) to
                             description_links
                reverse uses;
end description_links;

end function;
```

```
-----------------------------------------------------
--- SDS classgrp
-----------------------------------------------------
--- Autor: Chantal Keller
--- Stand: 12.09.99
-----------------------------------------------------
--- Contents:
---     extends SDS function with classgroup-relevant
---     parts
---
---   Attribute types:
---     doc_nr, usage_descr_nr, doc_address_nr
---   Object types:
---     documentation, usage_description,
---     environment_info,  evaluating_info
---
-----------------------------------------------------

sds classgrp:

-----------------------------------------------------
--- IMPORTS -----------------------------------------
-----------------------------------------------------

import objecttype       hpcte-home;
import objecttype       system-object;
import attribute        system-system_key;

--- IMPORTS FROM SDS MISC ---------------------------

import objecttype       misc-user;
import objecttype       misc-author;
import objecttype       misc-url_address;
import objecttype       misc-company;

--- IMPORTS FROM SDS LIBRARY ------------------------

import objecttype       library-component_category;

--- IMPORTS FROM SDS FUNCTION -----------------------

import objecttype       function-component_description;
import objecttype       function-administrative_info;
import objecttype       function-categorizing_info;
import objecttype       function-contents_info;
import objecttype       function-classification;
import objecttype       function-description_links;
import objecttype       function-structure_description;
import objecttype       function-usage_log;
import objecttype       function-usage_entry;

-----------------------------------------------------
--- ATTRIBUTE TYPES ---------------------------------
-----------------------------------------------------
```

```
attribute doc_nr                : natural;
attribute usage_descr_nr        : natural;
attribute doc_address_nr        : natural;

-----------------------------------------------------
--- OBJECT TYPES ------------------------------------
-----------------------------------------------------

--- COMPONENT_DESCRIPTION ---------------------------

extend objecttype component_description
with
  component
        to_environment_info:
                composition link to environment_info;
        to_evaluating_info:
                composition link to evaluating_info;
end component_description;

--- CONTENTS_INFO -----------------------------------

extend objecttype contents_info
with
  attribute
        faq                     : string;
  component
        to_documentation:
                composition link (doc_nr) to
                        documentation;
        to_usage_description:
                composition link (usage_descr_nr) to
                        usage_description;
end contents_info;

--- DOCUMENTATION -----------------------------------

objecttype documentation: child type of object
with
  attribute
        language                : string;
        readability             : string;
        completeness            : string;
  link
        doc_address:
                existence link (doc_address_nr) to
                       url_address;
end documentation;

--- USAGE_DESCRIPTION -------------------------------

objecttype usage_description: child type of object
with
  attribute
```

```
        usage_descr_text        : string;
end usage_description;

--- ENVIRONMENT_INFO -------------------------------

objecttype environment_info: child type of object
with
  attribute
        development_environment : string;
        execution_environment   : string;
        testing_environment     : string;
end environment_info;

--- EVALUATING_INFO --------------------------------

objecttype evaluating_info: child type of object
with
  attribute
        reliability             : string;
        ease_of_modification    : string;
end evaluating_info;

end classgrp;
```

```
-----------------------------------------------------
--- SDS pattern
-----------------------------------------------------
--- Autor: Chantal Keller
--- Stand: 14.04.00
-----------------------------------------------------
--- Contents:
---     extends SDS base with pattern-relevant parts
---
---   Attribute types:
---     part_nr, sample_nr, known_use_nr, rel_pattern_nr
---   Object types:
---     structure, participant, sample, known_use
---
-----------------------------------------------------

sds pattern:

-----------------------------------------------------
--- IMPORTS -----------------------------------------
-----------------------------------------------------

import objecttype       hpcte-home;
import objecttype       system-object;
import attribute        system-system_key;

--- IMPORTS FROM SDS MISC ---------------------------

import objecttype       misc-user;
import objecttype       misc-author;
import objecttype       misc-company;
import objecttype       misc-url_address;

--- IMPORTS FROM SDS LIBRARY ------------------------

import objecttype       library-component_category;

--- IMPORTS FROM SDS BASE ---------------------------

import objecttype       base-administrative_info;
import objecttype       base-categorizing_info;
import objecttype       base-contents_info;
import objecttype       base-classification;
import objecttype       base-usage_log;
import objecttype       base-usage_entry;

-----------------------------------------------------
--- ATTRIBUTE TYPES ---------------------------------
-----------------------------------------------------

attribute part_nr               : natural;
attribute sample_nr             : natural;
attribute known_use_nr          : natural;
attribute rel_pattern_nr        : natural;
```

```
----------------------------------------------------
--- OBJECT TYPES -----------------------------------
----------------------------------------------------

--- ADMINISTRATIVE_INFO ----------------------------
extend objecttype administrative_info
with
  attribute
        alternative_names       : string;
end administrative_info;

--- CATEGORIZING_INFO ------------------------------

extend objecttype categorizing_info
with
  attribute
        application_domain      : string;
        purpose                 : enumeration (
                                    UNKNOWN,
                                    CREATIONAL,
                                    STRUCTURAL,
                                    BEHAVIORAL);
        scope                   : enumeration (
                                    UNKNOWN,
                                    CLASS,
                                    OBJECT);
end categorizing_info;

--- CONTENTS_INFO ----------------------------------

extend objecttype contents_info
with
  attribute
        intent                  : string;
        motivation              : string;
        applicability           : string;
        consequences            : string;
        implementation          : string;
        related_patterns        : string;
  link
        is_related_pattern:
                reference link (rel_pattern_nr) to
                        contents_info
                reverse rev_is_related_pattern;
        rev_is_related_pattern:
                reference link to contents_info
                reverse is_related_pattern;
  component
        to_structure:
                composition link to structure;
        to_sample:
                composition link (sample_nr) to
                        sample;
```

```
        to_known_use:
                composition link (known_use_nr) to
                        known_use;
end contents_info;

--- STRUCTURE -------------------------------------

objecttype structure: child type of object
with
  attribute
        structure_representation        : string;
        collaboration                   : string;
  component
        to_participant:
                composition link (part_nr) to
                        participant;
end structure;

--- PARTICIPANT -----------------------------------

objecttype participant: child type of object
with
  attribute
        partipant_name          : string;
        responsibility          : string;
end participant;

--- SAMPLE ----------------------------------------

objecttype sample: child type of object
with
  attribute
        sample_code             : string;
end sample;

--- KNOWN_USE -------------------------------------

objecttype known_use: child type of object
with
  attribute
        known_use_text          : string;
end known_use;

end pattern;
```

**Erklärung**

Hiermit versichere ich, daß ich diese Arbeit selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Siegen, im April 2000