Werkzeugunterstützung für eine systematische Klassifizierung von Softwarekomponenten im Softwareentwicklungsprozeß nach WAM

Thorsten Gildhoff

Diplomarbeit

Universität Hamburg
Fachbereich Informatik
Juli 1997

Erstbetreuer:
Prof. Dr. Heinz Züllighoven

Zweitbetreuer:
Prof. Dr. Winfried Lamersdorf

Erklärung

Diese Diplomarbeit wurde am Fachbereich Informatik der Universität Hamburg zur teilweisen Erfüllung der Anforderungen zur Erlangung des Titels Diplom-Informatiker eingereicht. Ich versichere hiermit, diese Arbeit selbständig und unter ausschließlicher Zuhilfenahme der in der Arbeit aufgeführten Hilfsmittel erstellt zu haben.

Baiersbronn, den 14.07.1997

Thorsten Gildhoff

Finkenstraße 22
72270 Baiersbronn

gildhoff@jutho.fds.bawue.de

Betreuung

Prof. Dr. Heinz Züllighoven (Erstbetreuer)
Prof. Dr. Winfried Lamersdorf (Zweitbetreuer)

Prof. Dr. Heinz Züllighoven

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

Prof. Dr. Winfried Lamersdorf

Fachbereich Informatik
Arbeitsgruppe Verteilte Systeme
Universität Hamburg
Vogt-Kölln-Str. 30
D-22527 Hamburg

Für Jule, Tine und Thilo.

Und für meine Eltern, die nach Schwiegertochter und Enkeln nun doch noch einen diplomierten Sohn erhalten.

Inhaltsverzeichnis

1 Vorstellung der Arbeit

Dem Klassifizieren von Softwarekomponenten kommt im Rahmen eines jeden Softwareprojektes hohe Bedeutung zu. Um Softwareprodukte entsprechend dem heutigen Stand von Wissenschaft und Technik zu entwickeln, bedarf es der Bearbeitung einer großen Anzahl unterschiedlichster Dokumente. Nur eine systematische Klassifizierung ermöglicht ihre sichere und effiziente Handhabung. In dieser Arbeit wird am Beispiel der Quelltexte untersucht, ob und wie eine werkzeugunterstützte Anwendung eines Klassifizierungs-Modells die Integration der einzelnen Entwurfsdokumente in den Entwicklungsprozeß verbessern kann. Aufgrund der gefundenen Ergebnisse wird ein Klassifizierungsmodell entwickelt.

1.1 Einleitung

Ein Softwareentwickler steht im Verlauf einer Projektphase häufig vor dem Problem, daß eine Änderung eines Dokumentes, zum Beispiel einer Handhabungsvision, wie in nachfolgender Abb. 1-1 dargestellt, die Änderung zugehöriger Quelltexte nach sich zieht.

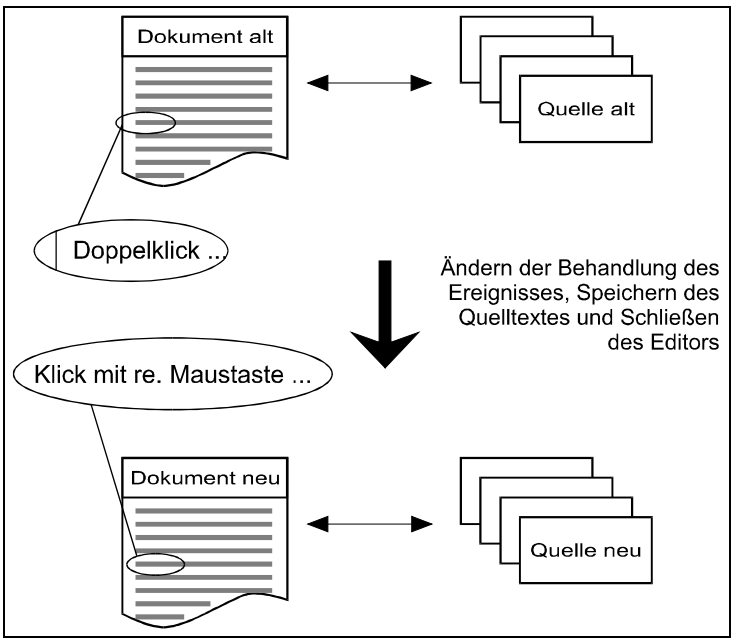


Abb. 1-1: Anpassen der Quelltexte nach Änderung eines Dokumentes

Der Vorteil, der darin liegt feststellen zu können, welches die abhängigen Quelltexte sind, liegt auf der Hand. Folglich wäre es sehr hilfreich, wenn eine Entwicklungsumgebung von Anfang an „wüßte“, welche Abhängigkeiten zwi-

schen den verschiedenen Dokumenten bestehen. Um eine Entwicklungsumgebung mit diesem Wissen auszustatten, bedarf es der Auseinandersetzung in den Bereichen, die im folgenden skizziert werden.

Methoden und Werkzeuge zur Klassifizierung

Hierdurch lassen sich Formalismen gewinnen, auf deren Basis sich Dokumente unter verschiedenen Aspekten gruppieren und so deren Zusammenhänge und Abhängigkeiten darstellen lassen. Dies geschieht in Kapitel 2. Es mündet in der Vorstellung eines Meta-Modells, das durch Instanziierung an konkrete Verwendungszusammenhänge angepaßt werden kann.

Analyse des Prozesses der Softwareentwicklung

Hier ist zu klären, welche Dokumente in welchen Phasen des Prozesses erstellt und geändert werden und ihre Beziehungen untereinander müssen transparent gemacht werden. Dies ist Gegenstand von Kapitel 3, wo eine detaillierte Darstellung des hier zugrunde liegenden Softwareentwicklungsprozesses nach WAM erfolgt. Besonderes Gewicht wird dabei auf die Rolle der verschiedenen Dokumenttypen wie Szenarios, Visionen, etc. gelegt. Ein eigener Abschnitt beschäftigt sich dabei intensiver mit den Quelltexten.

Auf der gefundenen Grundlage widmet sich das 4. Kapitel einer beispielhaften Instanz des Meta-Modells. Die Klassifizierung von Quelltexten im WAM-Prozeß soll die Anwendung des vorgestellten Modells verdeutlichen und dabei gleichzeitig Anregungen für einen sinnvollen Einsatz liefern.

Entwurf und Implementation eines Werkzeuges

Mit den gewonnenen Erkenntnissen ist es in Kapitel 5 dann möglich, ein softwaretechnisches Werkzeug zu entwerfen und zu implementieren, welches den Softwareentwickler in seiner Arbeit sinnvoll unterstützt und welches die gesteckten Ziele im Rahmen einer Entwicklungsumgebung erfüllen könnte.

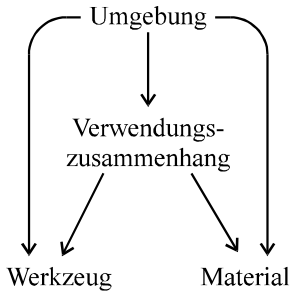
Neben einer abschließenden Zusammenfassung in Kapitel 6 bildet der Anhang mit folgenden Themen das Ende dieser Arbeit: eine Vorstellung des implementierten Rahmenwerkes für EiffelVision, eine kurze Einweisung in die Handhabung des Werkzeuges sowie Hinweise zur Installation und Übersetzung der beigefügten Quelltexte.

1.2 Sprachliche und graphische Konventionen

Den sprachlichen Rahmen dieser Arbeit bildet die Begriffswelt der statisch getypten, objektorientierten Programmiersprachen. Allgemein werden hierbei Bezeichnungen wie Klasse, Objekt, Vererbung und Polymorphie verwendet. Soweit es um konkrete, sprachspezifische Fragen der Implementation geht, werden auch die für Eiffel typischen Begriffe, wie z.B. ‘aufgeschoben’ für ‘abstrakt’ verwendet.

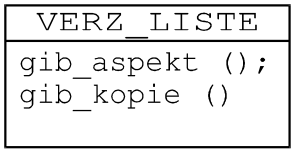
Die Namen von Entwurfskomponenten, wie `MAGAZIN`, *`VERWALTBAR`* oder *`gib_list_rep ()`* werden in Courier-Schrift dargestellt. Bezeichnungen von Mustern, zum Beispiel *Umgebung* und *Ereignismechanismus*, werden im Text kursiv gesetzt.

Die graphische Darstellung der Muster ist [Rie95] entnommen. Die Klassen- und Objektdiagramme folgen im wesentlichen der OMT-Notation nach [Rum95]. Hiervon abweichende Darstellungen werden im folgenden gesondert gekennzeichnet:

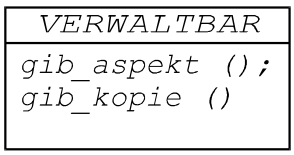


Der Zusammenhang zwischen Mustern wird mit gerichteten Graphen dargestellt. Die Knoten des Graphen bilden die Muster und als Ordnungsrelation dient der Umfang des Kontextes. Ein Pfeil zeigt von einem Muster mit umfassenderem, „größerem“ Kontext, zu einem mit eingeschränkterem, „kleinerem“ Kontext.

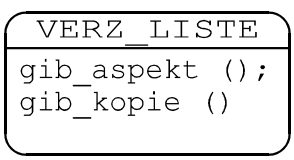
Die Darstellung einer Klasse erfolgt in Form eines Rechtecks. Der Klassenname steht in Großbuchstaben.



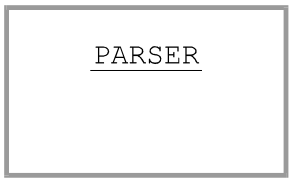
Attribute und Routinen werden, abgetrennt durch eine Linie, unterhalb des Klassennamens aufgeführt.



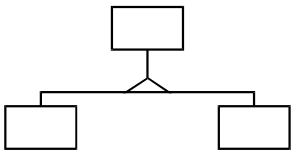
Abstrakte Klassen werden mit kursivem Schriftzug gekennzeichnet.



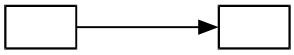
Objekte werden als Rechtecke mit abgerundeten Ecken dargestellt.



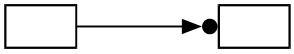
Gruppen von Klassen, die sich als Subsysteme (im Sinne von [Gam92], S. 103) benennen und behandeln lassen, werden als graue Rechtecke dargestellt. Der Name steht in Großbuchstaben und ist unterstrichen (nicht OMT).



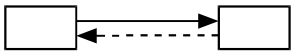
Die Vererbungsbeziehung wird durch ein Dreieck dargestellt, das über gerade Linien Ober- und Unterklassen verbindet. Die Spitze des Dreiecks zeigt auf die Oberklasse.



Die Benutztbeziehung wird durch einen einfachen Pfeil dargestellt.



Wird eine Klasse n-mal benutzt, wird an die Pfeilspitze ein Kreis angehängt.



Die Beobachterbeziehung wird dargestellt durch eine Benutztbeziehung, erweitert durch einen zurücklaufenden gestrichelten Pfeil. Dieser Pfeil kennzeichnet den Weg, auf dem Ereignisse an den Beobachter gemeldet werden (nicht OMT).

2 Klassifizierung von Softwarekomponenten

Jeder Softwareentwickler wendet in seiner täglichen Arbeit Verfahren zur Klassifizierung an. Dabei handelt es sich meist um einfache Gruppierungen, wie zum Beispiel das Einordnen eines Quelltextes in eine vorgegebene Verzeichnisstruktur. Dieses Vorgehen erleichtert zwar die Handhabung, es wird jedoch der großen Menge an Dokumenten innerhalb eines Entwicklungsprozesses nicht gerecht. Das sehr komplexe Beziehungsgeflecht der Dokumente läßt sich auch mit einer noch so ausgeklügelten Verzeichnisstruktur nicht fassen. Die eingehende Beschäftigung mit den formalen Grundlagen der Klassifizierung wird zeigen, daß Modelle zur Verfügung stehen, die auch im Rahmen moderner Softwareentwicklung eine effiziente Handhabung aller anfallenden Dokumente ermöglichen.

Dieses Kapitel bietet einen wertenden Überblick über Klassifizierungsmodelle im allgemeinen und jene von Softwarekomponenten im besonderen. Auf dieser Basis wird ein Meta-Modell zur Klassifizierung vorgestellt, welches die Komplexität eines Entwicklungsprojektes beherrschbarer macht. Darüber hinaus wird mit diesem Modell eine Voraussetzung geschaffen, Softwarekomponenten auch über einzelne Projekte hinaus wiederzuverwenden.

2.1 Bekannte Verfahren zur Klassifizierung

Überall dort, wo eine große Anzahl von Dingen zur Nutzung verwahrt wird, stellt sich das Problem, den Zugriff auf die einzelnen Teile zu gewährleisten. Dabei ist es zunächst einmal unerheblich, ob es sich dabei um Bücher in einer Bibliothek, den Lagerbestand eines Holzgroßhandels oder eben Softwarekomponenten handelt. Die Antwort besteht immer darin, ein Schema zu finden, nach dem sich die Dinge gruppieren und in eine Ordnung bringen lassen. Klassifizieren ist nichts anderes, als ähnliche Dinge zu gruppieren. Ist eine solche Ordnung geschaffen und die Informationen darüber sind zugänglich, so läßt sich der Zugriff auf einzelne Dinge erleichtern, häufig sogar überhaupt erst ermöglichen (ohne eine Klassifizierung der Bücher in einer Bibliothek und einen Bibliothekar würde man kaum jemals ein Buch finden).

Um eine Klassifizierung vornehmen zu können ist es nötig, die Informationen zu generieren, die die einzelnen Dinge beschreiben. Generell kann dabei zwischen zwei Arten der Generierung unterschieden werden:

- Die Informationen werden aus den Dingen selbst erzeugt. Dies ist zum Beispiel der Fall, wenn die CD’s im Geschäft nach den Namen der Interpreten sortiert in den Regalen stehen. Handelt es sich um Softwarekomponenten, so können diese Informationen auch automatisch erzeugt werden. Beispiele finden sich in jeder (brauchbaren) Entwicklungsumgebung, bei der eine objektorientierte Programmiersprache benutzt wird: jeder Browser, der eine Klassenhierarchie am Bildschirm darstellt arbeitet nach diesem Schema.

- Die Informationen werden manuell erzeugt. Durch eine genaue Analyse der Dinge und ihrer zugrunde liegenden Domäne werden Beschreibungen erzeugt und als Code zugeordnet. Dieses Verfahren wird in Bibliotheken angewandt. Bücher können nicht nur über Titel und Autoren gesucht werden, sondern auch nach bestimmten Schlüsselworten, Themenbereichen, etc. Diese Informationen sind den Büchern manuell zugeordnet worden und ergeben sich nicht notwendigerweise aus dem Titel oder dem Inhalt.

Soweit es bei den zu klassifizierenden Dingen um Softwarekomponenten geht, gibt es eine Vielzahl an relevanten Informationen, die sich nicht aus den Komponenten selbst gewinnen lassen. Die Tatsache, daß eine Klasse Bestandteil eines bestimmten Diagrammes ist oder zu einer ganz bestimmten Handhabungsvision gehört, läßt sich nicht aus dem Klassentext ablesen. Auch die Zugehörigkeit zu einem bestimmten Musterexemplar läßt sich praktisch nicht automatisch herleiten.

Es wird deshalb im weiteren nur die zweite Methode näher untersucht werden: die formalisierte Beschreibung wird manuell erzeugt. Wie sich noch zeigen wird heißt das allerdings nicht, daß diese Beschreibung dann in keinem Fall Bestandteil des zu klassifizierenden Elementes ist. Die generierten Informationen können sehr wohl auch dort hinterlegt werden, sie müssen es aber nicht sein.

Bei diesen Klassifizierungsmethoden können wiederum zwei Arten unterschieden werden: Aufzählungen und Facetten-Modell.

2.1.1 Aufzählungen

Als Beispiel für eine aufzählende Klassifizierung kann die Universal Decimal Classification (UDC) [IDF81] dienen. Der Grundgedanke ist dabei folgender: Das gesamte Wissen einer Domäne wird aufgeteilt in zehn Gebiete. Jedes Gebiet wiederum in zehn weitere, usw. Angewandt auf Bücher in einer Bibliothek würde sich dann ein ähnliches Bild wie Abb. 2-1 (siehe Seite 7) ergeben.

Es entsteht so als Code eine Ziffernfolge (deswegen die Bezeichnung *Decimal*), die aus Gründen der besseren Lesbarkeit nach jeweils drei Ziffern durch einen Punkt getrennt wird. So ergibt sich eine lineare Aufzählung, nach der die Bücher geordnet und wiedergefunden werden können.

Ein solches Schema eignet sich sehr gut zur Klassifizierung von großen Mengen sehr unterschiedlicher Dinge. Das System ist jedoch durch die stringente Aufteilung sehr starr, sodaß es schwierig ist, neue Kategorien einzuführen. Als Folge davon lassen sich sehr ähnliche Dinge nur schwer unterscheiden. Aus der Sicht des Softwareentwicklers ist diese Methode deshalb wenig zur Klassifizierung geeignet. Softwarebibliotheken unterliegen einem ständigen Wandel. Aufgrund des häufigen Hinzufügens und Weiterentwickelns von Komponenten wäre ein Klassifikationsschema, das dynamisch erweiterbar ist, wesentlich geeigneter.

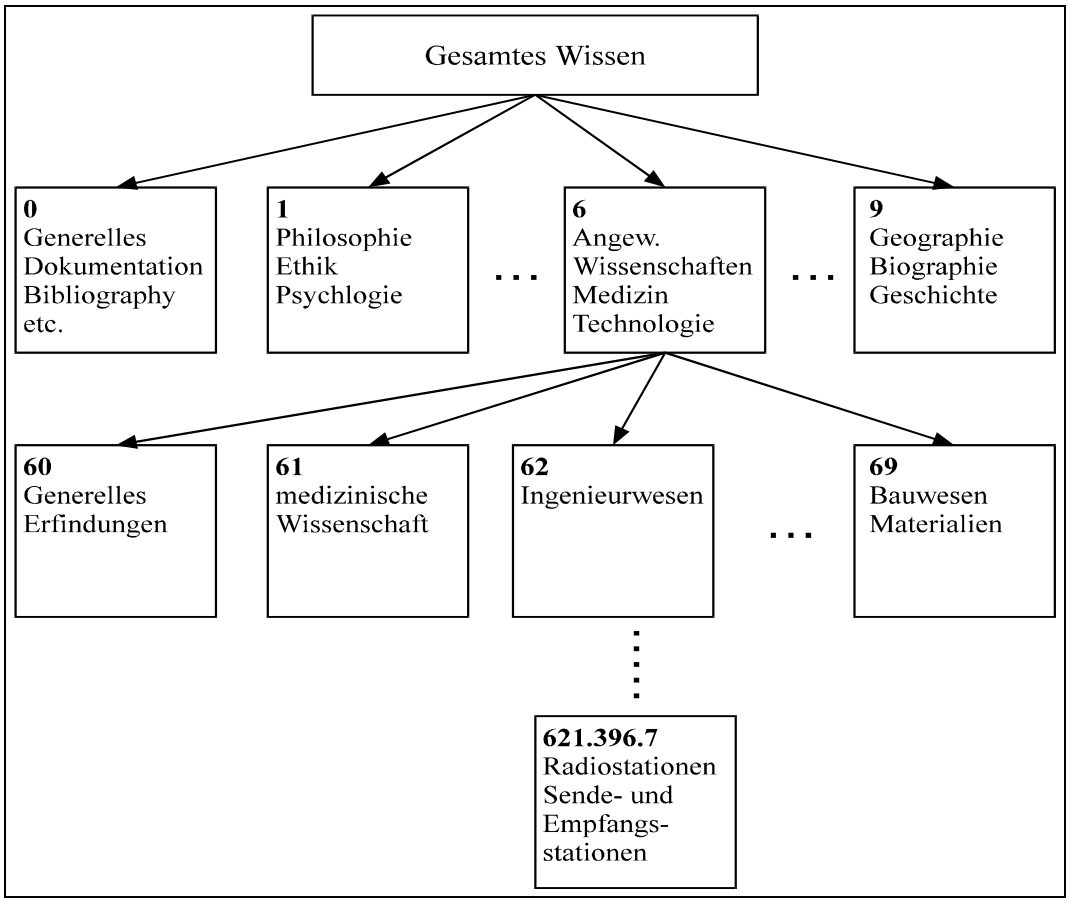


Abb. 2-1: Anwendung des UDC Schemas

2.1.2 Das Facetten-Modell

Ein Klassifikationsschema, das einfach erweiterbar ist, ist das Facetten-Modell. Jeder zu klassifizierenden Komponente wird eine Reihe von Eigenschaften, Facetten genannt, zugewiesen. Zu jeder Facette gibt es eine Vielzahl Werte, auch Terme genannt, die die möglichen Ausprägungen der Facette darstellen. Im Prinzip besteht die Klassifizierung einer Komponente dann aus einem n-Tupel von Eigenschaften, also Termen, wobei n im Bereich von 1 bis zur Anzahl der Facetten liegen kann.

Der wesentliche Unterschied zu den Aufzählungen besteht darin, daß zur Klassifizierung nicht ein bereits vollständiges Konzept zur Charakterisierung jeweils einer Komponente benutzt wird (siehe Abb. 2-2), sondern die Summe der

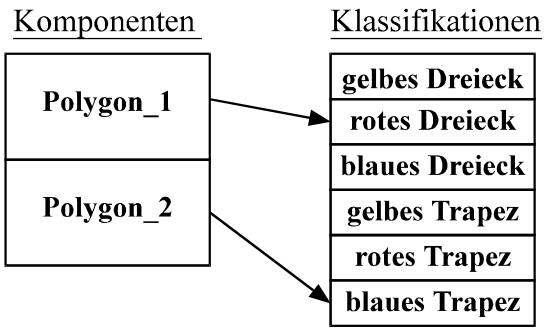


Abb. 2-2: Aufzählung

den Komponenten zugeordneten Terme dieses Konzept erst ausdrückt (dargestellt in Abb. 2-3).

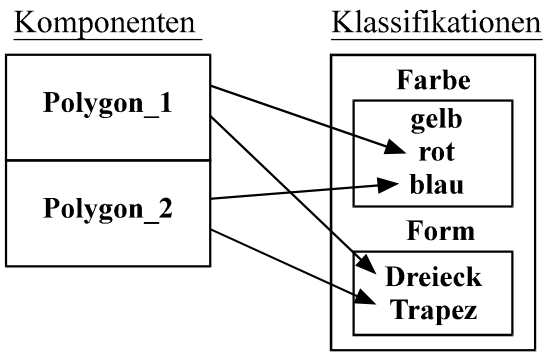


Abb. 2-3: Facetten-Methode

Das erste Klassifikationsschema für Bibliotheken, dem das Facetten-Modell zugrunde lag, war die *Colon Classification*. Sie wurde 1924 von dem Inder S.R.Ranghanathan [Ran57, Ran67] entwickelt. In einem ersten Schritt wird dabei jede Veröffentlichung in eine Hauptkategorie eingeordnet (z.B. Physik, Medizin, Landwirtschaft, etc.), was noch dem Vorgehen bei Aufzählungen entspricht. Im folgenden werden dann jedoch zusätzliche Informationen aus fünf Kategorien – den Facetten – berücksichtigt:

- Personality: Der eigentliche Diskussionsgegenstand.
- Matter: Weitere Themen.
- Energy: Dynamische Aspekte.
- Space: Geographische Lokation.
- Time: Zeitliche Einordnung.

Die Klassifizierungscodes bestehen dann aus dem Code für die Hauptkategorie sowie zusätzlichen Werten aus den fünf Facetten. Zwei Beispiele für solche Codes (entnommen [ST92 S. 41,42]) sind: X:5.1(X6.73)9Z.N5, was für *Commerce of non-dollar areas up to the 1950's* steht und X,61.73'N6, für *Monetary economics in the USA during the 1960's*. Diese einfachen Beispiele verdeutlichen die Wichtigkeit der Facetten bei der Wissensrepräsentation im Gegensatz zur eindimensionalen Darstellung bei reinen Aufzählungen.

Die erste bedeutende Anwendung für den Bereich der Softwareentwicklung stammt von Ruben Prieto-Diaz [PD85, PDF87] aus dem Jahre 1985. Für den Bereich der objektorientierten Anwendung wurde dieser Ansatz dann 1992 von Lars Sivert Sørumgård und Eirik Tryggeseth [ST92] weiterentwickelt. Die Beschäftigung mit Klassifikationsschemata in diesen Arbeiten geschieht dabei unter dem Aspekt der Wiederverwendung von Softwarekomponenten. Dabei wird jeweils von einer großen Menge – häufig sehr ähnlicher – Komponenten ausgegangen.

2.1.3 Facetten-Modell nach Prieto-Diaz

Nach [PDF87] werden Softwarekomponenten beschrieben durch die Funktion, die sie erfüllen, wie sie diese Funktion erfüllen und die zugrundeliegenden Implementationsdetails. Diese lediglich an die Funktionalität einer Komponente gekoppelte Definition führt zur Bildung von sechs Facetten, die in zwei Gruppen gegliedert werden.

Die erste Gruppe – **Functionality** – beschreibt, welche Funktionalität eine Komponente erfüllt:

- **Function:** die primitive Funktion, die ausgeführt wird (insert, delete, move, add, ...).
- **Objects:** die Strukturen, die von der Funktion manipuliert werden (files, arrays, variables, ...).
- **Medium:** das Medium, das zur Ausführung der Funktion benutzt wird (file, array, buffer, aber auch Maus, Bildschirm, Drucker, ...).

In der zweiten Gruppe – **Environment** – wird beschrieben, in welchem Kontext die Komponente eingesetzt wird:

- **System type:** die Aufgabe, die die Komponente – zusammen mit anderen – unabhängig von einer konkreten Anwendung erfüllt (Compiler, Parser, Relationale Datenbank, ...).

- **Functional area:** die Domäne, für die die Komponente erstellt wurde (Buchhaltung, CAD, Datenbankdesign, ...).

- **Setting:** das konkrete Einsatzgebiet der Komponenten (Autowerkstatt, Friseur, Softwareladen, ...).

Bei der Klassifizierung von Softwarekomponenten entstehen dann 6-Tupel wie zum Beispiel <einfügen, Dateien, Datei, RelationaleDB, Programmierung, Softwarewerkstatt>. Die Facetten sind dabei entsprechend ihrer Relevanz geordnet, daß heißt, die größte Bedeutung wird ‘Function’ beigemessen und die geringste ‘Setting’.

Wenn Komponenten zum Zwecke der Wiederverwendung gesucht werden stellt sich die Frage, wie verfahren werden soll, wenn die Suche nach einer Komponente mit einer bestimmten Kombination von Termen nicht erfolgreich ist. Der Suchalgorithmus sollte dann in der Lage sein, möglichst ähnliche Komponenten zur Wiederverwendung aufzufinden. Hierzu müssen die Terme in eine Ordnung gebracht werden, welche die konzeptionelle Nähe der einzelnen Terme zueinander ausdrückt.

Die in [PDF87] vorgeschlagene Antwort besteht darin, die Terme einer jeden Facette in einem gerichteten azyklischen Graphen anzuordnen. Die [PDF87] entlehnte Abb. 2-4 zeigt das Beispiel eines solchen Graphen für die Function-Facette.

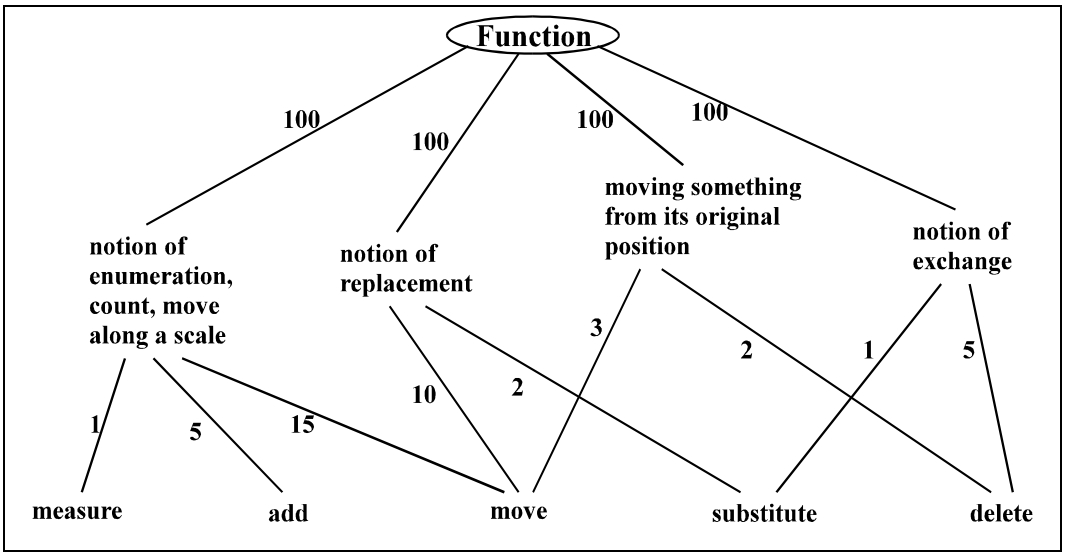


Abb. 2-4: Terme in einem gerichteten azyklischen Graph

Die Kanten kennzeichnen Spezialisierungen, die Knoten bezeichnen jeweils Arten von Funktionen und die Blätter bilden die eigentlichen Terme. Die Kanten sind gewichtet entsprechend der relativen konzeptionellen Nähe, wobei die konzeptionelle Nähe um so größer ist, je kleiner die zugewiesenen Werte sind. Die Nähe eines beliebigen Term-Paares einer Facette kann dann errechnet werden als Summe der Werte entlang des die beiden Terme verbindenden Pfades.

2.1.4 Facetten-Modell nach Sørumgård/Tryggeseth

Den Ausgangspunkt für die Arbeit von Sørumgård und Tryggeseth [ST92] bildet das gerade vorgestellte Facetten-Modell nach Prieto-Diaz, welches sie in folgenden Punkten für den Einsatz im Rahmen des objektorientierten Paradigmas für unzureichend halten:

- Die starke Ausrichtung auf den konkreten Quelltext erscheine problematisch.
- Die rein funktionale Sicht eigne sich nicht zur Klassifizierung von objektorientierten Komponenten.
- Zur Anordnung der Terme werde in den Graphen nur die Spezialisierung und die Generalisierung als Relationen verwendet. Andere wichtige Relationen würden nicht berücksichtigt.
- Nur die Terme an den Blättern der Graphen könnten zur Klassifizierung benutzt werden. Dies erscheine als eine unnötige Einschränkung.

Der erste Schritt für Sørumgård/Tryggeseth besteht darin, eine adäquate Wahl von Facetten für objektorientierte Softwarekomponenten zu treffen. Die Unterteilung der Facetten in zwei Gruppen, Functionality und Environment, wird beibehalten, wobei die Gruppe Environment übernommen wird, da es hierbei unerheblich ist, ob eine Komponente objektorientiert ist oder nicht. Für die Gruppe Functionality werden dann folgende Facetten gewählt:

- **Abstraction:** eine Bezeichnung für die abstrakte Entität eines Objektes als solches (stack, queue, list, collection, ...). Dies sind hauptsächlich abstrakte Datentypen (ADT's); es sollen aber auch domänenspezifische Begriffe benutzt werden können.
- **Operations:** die Operationen, die für das Objekt definiert sind (push, pop, insert, sort, print, ...). Dies ist im wesentlichen die Function-Facette von Prieto-Diaz, nur daß hier jeweils mehrere Terme benutzt werden können.
- **Operates on:** die Objekte, die von der Funktion manipuliert werden (self, files, arrays, variables, ...). Entspricht der Object-Facette von Prieto-Diaz.
- **Dependencies:** beschreibt Abhängigkeiten von Sprache, Betriebssystem, Fenstersystem, etc. (Linux, MS-DOS, Motif, Ingres, ...).

Nach [ST92] besteht nun die Möglichkeit, die Terme einer jeden Facette auf verschiedene Arten über gewichtete Relationen zu verknüpfen. Als wichtigste werden genannt: Spezialisierung, Generalisierung, Synonym, Ist-Teil-von und Enthält. Im Gegensatz zu Prieto-Diaz können alle Terme in dem so entstehenden gerichteten Graphen zur Klassifizierung verwandt werden (Abb. 2-5 zeigt ein einfaches Beispiel).

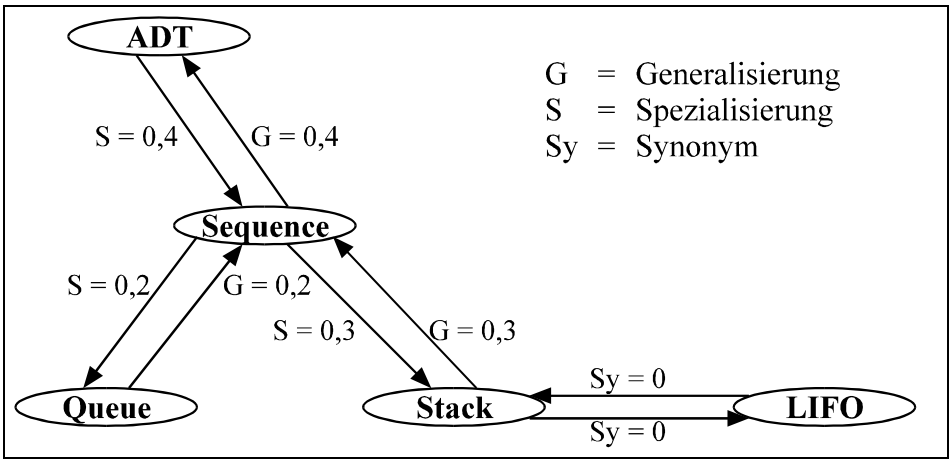


Abb. 2-5: Gewichtete Relationen von Termen

Mit Hilfe dieser gewichteten Relationen wird dann möglich, was in [ST92] „erweiterte Suche“ genannt wird. Ein Such-Werkzeug auf der Basis dieses Klassifikationsschemas ist nicht nur in der Lage, die Frage zu beantworten, ob eine Komponente mit ganz bestimmten Eigenschaften vorhanden ist oder nicht, sondern darüber hinaus können auch ähnliche Komponenten angeboten werden. Bis zu welchem Grad an Ähnlichkeit diese zur Wiederverwendung angeboten werden sollen, kann von dem Benutzer beim Suchen selbst bestimmt werden. Er kann dazu Facetten und Suchterme selber wichten, die Suche entlang bestimmter Relationen fordern oder verbieten und schließlich auch die Tiefe der Suche festlegen.

2.2 Das Meta-Modell zur Klassifizierung

Die hier vorgestellten Klassifizierungsmodelle aus der Literatur kennzeichnen die Entwicklung vom allgemeinen Verfahren bis hin zu einem Facetten-Modell, das auf dem objektorientierten Paradigma basiert. Diese Spezialisierung birgt das Problem, daß die Nutzung nur noch sehr eingeschränkt innerhalb eines klar umrissenen Anwendungsgebietes möglich ist. Innerhalb der Softwaretechnik gibt es jedoch als Anwendungsgebiete viele Sparten, die außerdem einem starken Wandel unterliegen, weil es sich bei der Informatik um eine noch recht junge Wissenschaft handelt.

Anstatt für jedes eng umrissene Anwendungsgebiet eine eigene Klassifizierungsmethode zu entwickeln, erscheint es deshalb angebracht, zunächst einmal von einem konkreten Modell abzusehen und einen allgemeineren Lösungsweg zu suchen. Es zeigt sich dann, daß es möglich ist, auf der Basis der bereits bestehenden Modelle ein Meta-Modell zu finden, welches durch Instanziierung zur Lösung der Klassifizierungsprobleme beitragen kann.

2.2.1 Vom Modell zum Modell eines Modells

Die grundlegende Basis für eine sinnvolle Klassifizierung von Softwarekomponenten bietet ganz allgemein das Facetten-Modell. Solange es sich bei den Komponenten im wesentlichen um ADT's handelt (wie zum Beispiel die Basisbibliotheken der Eiffel Entwicklungsumgebung), ist das Modell von Sørumgård/Tryggeseth gut geeignet. Dies scheinen auch die Erfahrungen zu bestätigen, die in [ST92] angeführt werden.

Sobald jedoch Komponenten klassifiziert werden sollen, die eng mit anderen kooperieren (also zum Beispiel Klassengruppen wie Teams, Subsysteme oder

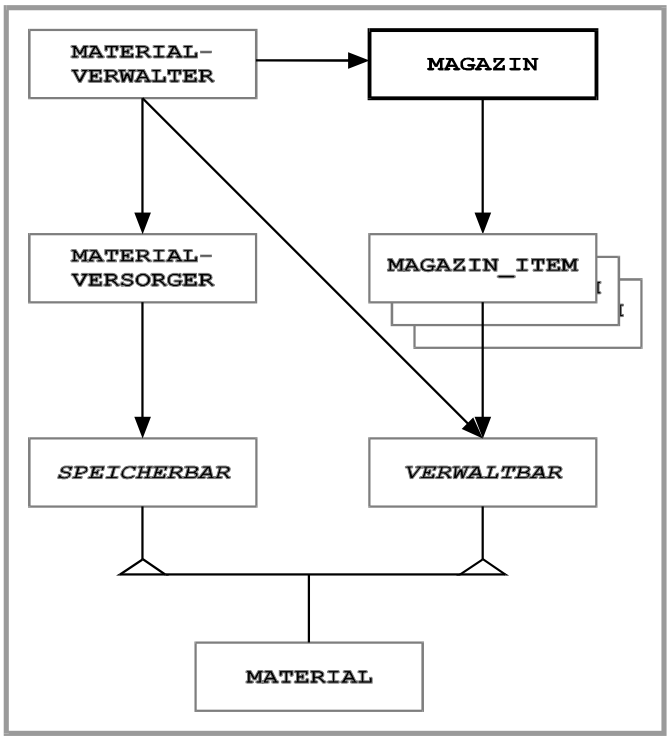


Abb. 2-6: Die Klasse `MAGAZIN` im Rahmen der Materialverwaltung als Verwaltermagazin.

Rahmenwerke), ist dies mit den gewählten Facetten schwierig. In [ST92] wird deshalb bereits die Einführung zweier weiterer Facetten – 'cooperating objects' und 'required operations' – vorgeschlagen.

Noch aus einem weiteren Grund scheint die Wahl der Facetten nicht hinreichend zu sein. Als Beispiel soll hier die Klasse `MAGAZIN` im Rahmen der Materialverwaltung dienen (siehe Abb. 2-6 auf Seite 13). Die Klasse ist implementiert als `LINKED_LIST [MAGAZIN_ITEM]` und sollte deshalb nach [ST92] abstrakt als linked_list mit den Operationen einfügen, löschen, nächstes, etc., klassifiziert werden. Nun läßt sich in der Tat diese Klasse ohne großen Aufwand als beliebige linked_list einsetzen. Ein Softwareentwickler wird aber an anderen Eigenschaften gerade dieser Klasse ein weitaus größeres Interesse haben, nämlich daß sie im Rahmen des Musterexemplares der *Materialverwaltung* die Aufgabe des *Verwaltermagazines* übernimmt und als *Container* beliebige *verwaltbare Materialien* beinhaltet. Es sind dies Eigenschaften, die auf einer ganz anderen Abstraktionsebene liegen, als die reinen Datenstrukturen.

Aus Vorstehendem wird deutlich, daß eine kleine, festgelegte Reihe von Facetten in dem Modell entweder zu unnötigen Einschränkungen führt, oder die Menge der Terme einer Facette viel zu groß und nur noch schlecht handhabbar wird. Auf der einen Seite kann die Entscheidung getroffen werden, die Klasse `MAGAZIN` als Liste oder als Verwaltermagazin zu klassifizieren. Dies wäre eine Einschränkung bei der wichtige Informationen nicht berücksichtigt würden. Andererseits können alle Eigenschaften herangezogen werden. Dann müßten diese jedoch nach [ST92] innerhalb einer Facette (Abstraction) hinterlegt werden. Da zusätzlich noch domänenspezifische Eigenschaften hinzukämen, würde die Anzahl der Terme nicht nur sehr groß werden, auch eine Ordnung der Terme ließe sich kaum mehr sinnvoll schaffen.

Hieraus läßt sich folgender wichtiger Schluß ziehen: Die Wahl der Facetten ist stark abhängig von der Art des Entwicklungsprozesses und der Anwendungsdomäne des Softwareprojektes; die passenden Facetten müssen also von Fall zu Fall gewählt werden. Eine Festlegung auf eine ganz bestimmte Menge innerhalb des Modells ist weder sinnvoll noch praktikabel. Um dieser Erkenntnis Rechnung zu tragen wird das hier vorgestellte Modell in Abgrenzung zu den klassischen Klassifizierungsmodellen als Meta-Modell bezeichnet.

Das vorzustellende Meta-Modell unterwirft den Softwareentwickler keinerlei Beschränkung hinsichtlich Anzahl und Art der Facetten. Es können nicht nur Terme in einem Klassifikationsschema neu aufgenommen werden, sondern auch ganze Facetten. Damit kann das Klassifikationsschema den speziellen Erfordernissen eines Entwicklungsprozesses oder auch einer Anwendungsdomäne angepaßt werden (siehe 2.2.2).

Auf diese Weise wird es möglich, das Facetten-Modell als sehr mächtiges Instrument zum Klassifizieren einzusetzen. Der Softwareentwickler ist nicht mehr von vornherein darauf festgelegt, alle Dokumente in ein vorgegebenes Schema zu pressen. Auch können verschiedene Arbeitsgruppen die Dokumente unter verschiedenen Gesichtspunkten betrachten. Die Anzahl der Facetten wird dabei zweifellos größer. Moderne Softwarewerkzeuge können jedoch auf einfache Weise sicherstellen, daß die Übersicht dabei nicht verlorengeht, indem sie als Filter die Sicht auf eine festzulegende Auswahl von Facetten beschränken.

2.2.2 Klassifikationsschemata als Instanzen des Modells

Da das hier vorzustellende Meta-Modell dem Benutzer die Wahl der Facetten freistellt, ist es um so wichtiger, diese Instanziierung mit großer Sorgfalt zu treffen. Würde in einer Arbeitsgruppe jeder Entwickler nach eigenem Gutdünken Facetten und zugehörige Terme einführen, so würde der Datenbestand sehr schnell ausufern und der Nutzen für den Entwicklungsprozeß würde entsprechend sinken.

Als Leitfaden für das Aufstellen eines Klassifikationsschemas können die folgenden Überlegungen dienen. Beispielhaft erfolgt an dieser Stelle bereits ein Vorgriff auf das Kapitel 4, in dem ausführlich ein Klassifikationsschema für die Entwicklung nach WAM vorgestellt wird. Dabei ist hier jedoch lediglich von Interesse, auf welchem Weg ein Klassifikationsschema erzeugt wird. Die WAM-spezifischen Begriffe, die im folgenden auftauchen (wie etwa *Vision*, *Muster*, *Werkzeug* oder auch *WAM* selbst), werden später detailliert erörtert und sind für das jetzige Verständnis noch nicht von Bedeutung.

Die Klassen eines Softwareprojektes lassen sich entsprechend ihren Aufgaben und Eigenschaften gruppieren. Einige besonders interessante Gruppen sind Klassen, die

- Bestandteil der Basisbibliotheken des Entwicklungssystems sind.
- mit anderen Dokumenten (Visionen, Glossare, Prototypen, Diagrammen) verknüpft sind.
- bestimmte Rollen innerhalb von Musterexemplaren, Subsystemen, etc. wahrnehmen.
- Werkzeuge, Materialien oder Aspekte, beziehungsweise einen Teil davon, darstellen.
- domänenspezifische Zusammenhänge modellieren.

Soweit es sich um Basisklassen handelt, wie etwa `LIST`, `QUEUE`, `SET` o.ä., so kann für das Schema auf die Erfahrungen von Sørumgård/Tryggeseth zurückgegriffen werden, das sich als sehr sinnvoll für ADT’s erwiesen hat.

Für die weiteren Facetten und Terme wird die obige Aufzählung als natürliche Gruppierung gewählt:

Facette	**Terme** (Auswahl)
Dokument	Vision, Glossar, Prototyp, Klassendiagramm, sowie Bezeichnungen aller Dokumente, die in Beziehung zu einer Klassenbeschreibung stehen
Rolle	Beobachter, Interpreter, Rahmenwerk, Fassade, Schnittstelle
WAM	Werkzeug, Automat, Aspekt, Material, Umgebung, IAK, FK
Domäne	Quelle, Index, Facette, Term
…	...

Die ersten drei Facetten dieser Aufstellung charakterisieren damit genau jene Eigenschaften, die von besonderer Wichtigkeit für den WAM-Entwicklungsprozeß sind. Die letzte Facette – Domäne – scheint auf den ersten Blick nicht recht in dieses spezifische Schema zu gehören. Die Berücksichtigung domänenspezifischer Eigenschaften und Zusammenhänge ist jedoch für *jedes* Softwareprojekt von eminenter Bedeutung und im Rahmen von WAM wird dieser Tatsache unter anderem durch die Modellierung fachlicher Werte Rechnung getragen. Eine Klassifizierung danach, wo wichtige fachliche Zusammenhänge modelliert werden, ist in diesem Sinne also durchaus auch WAM-spezifisch.

3 Der Prozeß der Softwareentwicklung nach WAM

Die Klassifizierung von Softwarekomponenten verlangt bei den immer komplexer werdenden Anwendungen ein systematisches und methodisches Arbeiten. Um das oben vorgestellte Meta-Modell zu einem konkreten, anwendbaren Modell für die Softwareentwicklung nach WAM instanziieren zu können, ist deshalb ein umfassendes und detailliertes Verständnis der WAM-Methode nötig.

Der hier notwendige methodische Rahmen läßt sich anhand der Beantwortung folgender Fragen abgrenzen: Welche Vorgehensweise wird gewählt? Was ist das Leitbild, das die weiteren Schritte maßgeblich bestimmt? Welche Metaphern und Muster können hilfreich sein und welche Konstruktionstechnik wird schließlich den Entwurf ausmachen?

Um abschließend den Fokus wieder auf die Klassifizierung der Dokumente zu legen, wird den Entwurfsdokumenten und insbesondere den Quelltexten, jeweils ein eigener Abschnitt am Ende des Kapitels gewidmet.

3.1 Vorgehensweise und Leitbild

Die Vorgehensweise beschreibt das Modell, das den Entwicklungsprozeß in seiner Gesamtheit charakterisiert. Bekannte Beispiele sind das Phasenmodell [Roy70], das Spiralmodell [Boe88] und das evolutionäre Prototyping [Flo84, BKK+92].

Dieser Arbeit liegt das Vorgehen des evolutionären Prototyping zugrunde. Das evolutionäre und partizipative Prototyping beschreibt einen Prozeß, an dem Entwickler und spätere Benutzer gleichermaßen teilhaben. In der Diskussion um Prototypen findet eine gemeinsame Sprachbildung und Rückkopplung statt.

In diesem Sinne ist auch die Studienarbeit „*Ein Prototyp eines Reuse-Tools für Eiffel*“ [Gil94] Bestandteil des Vorgehens, welches zu dem in dieser Arbeit vorgestellten Werkzeug geführt hat.

Als Besonderheit ist bei dem Entwurf eines Programmierwerkzeuges für Softwareentwickler die Tatsache zu sehen, daß dabei Anwender und Entwickler aus demselben Personenkreis stammen. Eine gemeinsame Sprachbildung sowie der sonst so wichtige Übergang vom fachlichen zum technischen Entwurf verlieren dabei an Relevanz, da die Terminologie und die grundlegenden Zusammenhänge als bereits bekannt und angewandt angesehen werden können.

Das Leitbild ist der Arbeitsplatz für qualifizierte menschliche Arbeit [GZ92]. Der Benutzer wird dabei als kompetent und fähig angesehen, seine Aufgaben eigenverantwortlich zu erledigen. Insbesondere wird darauf verzichtet, vordefinierte Arbeitsabläufe einzuführen, die den Anwender in seinen Freiheitsgraden einschränken.

3.2 Metaphern

Entwurfsmetaphern sind in allen Phasen der Systementwicklung von großem Nutzen. Als Analysemittel helfen sie bei der Strukturierung des Anwendungsbereiches und als Entwurfsmittel sind sie dienlich, das zukünftige System fachlich zu entwerfen. Ein guter fachlicher Entwurf läßt sich dann annähernd bruchlos in einen technischen Entwurf überführen.

Die Werkzeug und Material Metapher [BZ90, BZ92, BCS92, GZ92] hat sich als sehr fruchtbarer Ansatz erwiesen, interaktive Softwaresysteme von der Analyse über das Design bis hin zur Implementation zu entwickeln. In den letzten Jahren wurde sie ständig verfeinert [KGZ93, Gry96] und zur Zeit gilt am Arbeitsbereich Softwaretechnik des Fachbereichs Informatik an der Universität Hamburg die Softwareentwicklung nach WAM (für: **W**erkzeug, **A**spekt und **M**aterial oder auch **W**erkzeug, **A**utomat und **M**aterial Metapher) als ‘state of the art’.

Diese Entwurfsmetaphern werden im Verlauf der weiteren Erörterungen von besonderer Bedeutung sein. Im folgenden wird deshalb ein etwas ausführlicherer Einblick in das Repertoire und den Entwicklungsprozeß auf der Basis der WAM-Metaphern gegeben.

Die Grundlage des Entwicklungsprozesses bilden die Metaphern Werkzeug, Aspekt, Material, Automat und Umgebung. Spricht man von der Entwicklung nach WAM, so sind jedoch nicht nur diese Metaphern gemeint. Der Begriff der Softwareentwicklung nach WAM wird synonym verwendet für den hier skizzierten Entwicklungsprozeß als Ganzes, bei dem im besonderen auch die Entwurfsdokumente stark durch die Metaphern geprägt sind.

3.2.1 Werkzeug

Bei der Analyse eines Anwendungsbereiches läßt sich im allgemeinen recht einfach und intuitiv zwischen den vorgefundenen Arbeitsmitteln (den Werkzeugen) und den Arbeitsgegenständen (den Materialien) unterscheiden. Werkzeuge sind dabei die Mittel, die benutzt werden, um die Materialien zu bearbeiten oder zu sondieren.

Aus fachlicher Sicht lassen sich Werkzeuge in Klassen zusammenfassen. Ein Werkzeug kann dabei durchaus verschiedenartige Materialien be- oder verarbeiten. Beim Übergang zum technischen Entwurf werden dann Werkzeugklassen wie z.B. `EDITOR` oder `AUFLISTER` gebildet.

Die Werkzeuge bilden dann die interaktiven Komponenten des Softwaresystems. Sie stellen die aktuellen Materialzustände dar und bieten dem Benutzer die Möglichkeit der Handhabung dieser Materialien. Aufgrund der großen Komplexität solcher Komponenten werden die Werkzeuge häufig durch Komposition aus Unterwerkzeugen konstruiert.

3.2.2 Material

Materialien sind die Arbeitsgegenstände, die von den Werkzeugen bearbeitet werden. Sie lassen sich fachlich zu Klassen gruppieren, die im allgemeinen eine hierarchische Ordnung bilden. Aus technischer Sicht entstehen dann Materialklassen, die ihre Dienstleistungen zur Sichtung und Änderung den Werkzeugen anbieten.

Diese Dienstleistungen sind also dem Benutzer nicht direkt, sondern nur über die Werkzeuge zugänglich. Der Benutzer handhabt Werkzeuge, die dann als Reaktion auf bestimmte Anforderungen die Dienste der Materialien in Anspruch nehmen.

3.2.3 Aspekt

Damit Werkzeuge verschiedene Materialien bearbeiten und Materialien von verschiedenen Werkzeugen bearbeitet werden können, ist es nötig, den Verwendungszusammenhang zwischen beiden auf einer abstrakten Ebene zu definieren. Dies kann dann die Nutzung der Materialien gewährleisten, ohne konkrete Annahmen über ein bestimmtes Material treffen zu müssen.

Fachlich gesehen werden den Materialien jeweils Aspekte zugeordnet, die einzelne Verwendungszusammenhänge festlegen. So können z.B. Briefe und Notizen (als Materialien) entweder unter dem Aspekt ihrer Editierbarkeit, ihrer Druckbarkeit oder auch ihrer Kopierbarkeit betrachtet werden.

Bei der technischen Konstruktion werden dann abstrakte Aspektklassen gebildet, die lediglich die Schnittstellen der Materialien zu den Werkzeugen definieren. Werkzeugklassen benutzen dann Aspektklassen um auf Materialklassen zuzugreifen, die ihrerseits dann Unterklassen der Aspektklassen sind. Die Mehrfachvererbung macht es möglich, daß ein Material über seine Aspekte dann von verschiedenen Werkzeugen bearbeitet werden kann.

Auf diese Weise werden Werkzeuge und Materialien entkoppelt und sobald die Aspekte festgelegt sind, können sie unabhängig voneinander entworfen werden. Dies erhöht nicht zuletzt auch ihren Grad an Wiederverwendbarkeit.

3.2.4 Automat

Im Anwendungsbereich fallen immer wieder Tätigkeiten an, die routinemäßig erledigt werden müssen. Regelmäßig müssen Sicherungskopien von Dokumenten erstellt, Postkörbe geleert oder Druckaufträge in verschiedenen Warteschlangen abgearbeitet werden. Diese Aufgaben brauchen nicht immer wieder mit interaktiven Werkzeugen durchgeführt werden, sondern sie können auch automatisch im Hintergrund ablaufen.

In diesem Sinne sind Automaten den Werkzeugen zwar ähnlich, sie unterscheiden sich von ihnen jedoch in einigen wesentlichen Punkten:

- Automaten erledigen bestimmte Routineaufgaben selbständig und entlasten damit den Anwender, während ein Werkzeug den Anwender bei seinen Tätigkeiten direkt unterstützt.

- Im Hintergrund arbeitende Automaten werden im allgemeinen mit Materialströmen versorgt, die sie verarbeiten. Werkzeuge hingegen reagieren nur direkt auf Anforderungen des Anwenders.

- Aufgrund den ihnen zugewiesenen Aufgaben, haben Automaten nur eine minimale Benutzerschnittstelle, die es erlaubt bestimmte Einstellungen vorzunehmen (in welchen Zeitabständen wird gesichert?).

3.2.5 Umgebung

In einem Anwendungsbereich werden üblicherweise eine Vielzahl von sowohl Werkzeugen als auch Materialien gehandhabt. In der Arbeitswelt haben diese Dinge ihren festen Platz, unterliegen einer gewissen Ordnung und sind bei Bedarf zur Hand. Der Platz kann dabei ein Schreibtisch, eine Werkstatt oder auch ein Archiv sein.

Aus softwaretechnischer Sicht bildet die Umgebung den Rahmen für die fachlich zusammen gehörenden Werkzeuge, Automaten und Materialien. In einem Anwendungssystem stellt ein Umgebungsobjekt den Ort dar, an dem sie ihren festen Platz haben. Insbesondere stellt die Umgebung eine Abgrenzung zu anderen Umgebungen dar. Von außerhalb kann nicht steuernd in eine Umgebung eingegriffen werden, genauso wie Materialien, die innerhalb einer Umgebung bearbeitet werden, nicht gleichzeitig von außerhalb dieser Umgebung bearbeitet werden können.

3.3 Muster

Dem Begriff des Musters im Bereich der Softwaretechnik ist eine ständig wachsende Bedeutung zuzumessen. Viele Arbeiten aus den letzten Jahren [Coa92, Gam92, GHJ+93, Pre94] zeigen dies in aller Deutlichkeit. So gehört der Entwurfsmuster-Katalog [GHJ+95] der „gang-of-four“, wie sich die Autoren selber nennen, bereits heute für viele Entwickler zur Pflichtliteratur, um über ihre Softwaresysteme auf einer gemeinsamen Basis kommunizieren zu können.

3.3.1 Allgemeine Definition

Am Arbeitsbereich Softwaretechnik in Hamburg hat sich Dirk Riehle mit seiner Arbeit „*Muster am Beispiel der Werkzeug und Material Metapher*“ [Rie95] ausführlich mit der Bedeutung der Muster für die Softwaretechnik auseinandergesetzt. Er definiert dort den softwaretechnisch orientierten Musterbegriff, der im folgenden vorgestellt wird. Einen wichtigen Ausgangspunkt bilden für ihn die Arbeiten von Christopher Alexander [AIS77, Ale79], der den Musterbegriff in der

Architektur geprägt hat. Über die Diskussion der Arbeiten von Peter Coad [Coa92], Erich Gamma et al. [Gam92, GHJ+93] und Wolfgang Pree [Pre94] gelangt Riehle zu folgender Definition des Begriffes Muster:

> „Ein Muster ist eine in einem bestimmten Kontext erkennbare Form. Es dient als Vorlage zum Erkennen, Vergleichen und Erzeugen von Musterexemplaren. Ein Muster ist die Essenz aus Erfahrung und Analyse immer wiederkehrender Situationen. Es besitzt eine innere Struktur und Dynamik.“ [Rie95, S.20]

Besonders hervorgehoben wird in der anschließenden Diskussion: „Muster sind immer im Kontext zu sehen, welcher ihre Form und Struktur verständlich macht. Die Form eines Musters läßt sich als Reaktion auf seinen Kontext beschreiben. [...] Erkennen, Vergleichen und Erzeugen von Musterexemplaren“ ist „ein unbestimmter Prozeß. Weder eine Verfahrensvorschrift noch eine allgemeine Methodik sind vorgegeben.“ [Rie95, S. 20].

Um die Bedeutung des Musters für die Softwaretechnik zu verdeutlichen, werden nach der allgemeinen Definition im folgenden drei Ebenen eingeführt, die in Softwareprojekten von unterschiedlicher Relevanz sind: Interpretations- und Gestaltungsmuster, Entwurfsmuster und Programmiermuster.

3.3.2 Interpretations- und Gestaltungsmuster

Den Ausgangspunkt für den Einsatz von Mustern in Softwareprojekten bilden die Interpretations- und Gestaltungsmuster.

> „Ein Interpretations- und Gestaltungsmuster ist ein Muster, welches zur Interpretation und Gestaltung von tatsächlichen oder antizipierten Anwendungssituationen und Softwaresystemen verwendet werden kann.“ [Rie95, S. 21]

Hervorzuheben ist dabei, daß Interpretations- und Gestaltungsmuster der Orientierung aller Beteiligten dienen und helfen, eine gemeinsame Basis für das Verständnis der Anwendungswelt zu erlangen. Auf der anderen Seite sind sie jedoch auch nahe genug am Softwareentwurf, um den Übergang zu einem technischen Entwurf zu erleichtern.

3.3.3 Entwurfsmuster

Entwurfsmuster sind die objektorientierten, softwaretechnischen Umsetzungen der Interpretations- und Gestaltungsmuster.

> „Ein Entwurfsmuster ist ein Muster, das als Vorlage für die Konstruktion eines softwaretechnischen Entwurfs dient. Ein Entwurfsmuster besteht aus dem Zusammenspiel technischer Elemente wie Klassen, Objekte und Operationen. Die Struktur und Dynamik eines Entwurfs-

> musters klärt die beteiligten Komponenten, ihre Zusammenarbeit und die Verteilung der Verantwortlichkeiten.“ [Rie95, S. 22]

Auf dieser Ebene dienen die Entwurfsmuster allein der Kommunikation der Entwickler untereinander. Sie stellen Mikroarchitekturen dar, die flexibel kombiniert werden können, um eine Softwarearchitektur zu entwickeln.

3.3.4 Programmiermuster

Als drittes wird das Programmiermuster definiert:

> „Ein Programmiermuster ist ein Muster, das als Vorlage für die Implementation eines Entwurfs dient. Sie basieren auf Erfahrungswissen der Programmierung und sind in der jeweiligen Programmierkultur weithin bekannt.“ [Rie95, S. 23]

Hier geht es um die Ebene der Programmierung. „Programmiermuster sind jene Muster, die ein Informatiker aufgrund seiner Ausbildung als elementares Wissen der Programmierung mitbringt. [...] Die Verwendung von Programmiermustern ist weitgehend unbewußt, weil sie in langen Jahren eingeübt wurden und ihre Anwendung nicht weiter reflektiert zu werden braucht. Programmiermuster sind somit von der Programmierkultur und -sprache abhängig.“ [Rie95, S. 22,23].

3.4 Konstruktionstechnik

Bei der Entwicklung des Softwaresystems wird eine Vielzahl von Dokumenten erstellt. Eine Konstruktionstechnik beschreibt „Aufbau und Techniken der Erstellung und Weiterführung eines Dokuments“ [Rie95, S. 5]. Beispiele für Dokumente sind Szenarios, Visionen, Glossare, Prototypen und Quelltexte. Einen Überblick über die verschiedenen Dokumenttypen gibt der nächste Abschnitt.

Im Rahmen der hier vorgestellten Arbeit basieren die Dokumente auf dem Paradigma der objektorientierten Konstruktion. Die Quelltexte, sowohl soweit es später um die Klassifizierung geht, wie auch bei der Implementation des Werkzeuges, entstammen der Programmiersprache Eiffel [Mey90, Mey91].

3.5 Entwurfsdokumente

Während des Prototyping-Prozesses wird eine Vielzahl von Dokumenten erstellt und fortgeschrieben. Die folgenden Abschnitte geben einen kurzen Überblick über einige zentrale Dokumenttypen dieser Vorgehensweise. Zu den wichtigsten Dokumenttypen lassen sich detaillierte Ausführungen in [KGZ93, S. 98ff] nachlesen.

3.5.1 Szenarios

Bei der Analyse eines Anwendungsbereiches sind Szenarios von besonderer Relevanz. Sie schildern in prosaischer Form den Ist-Zustand einzelner Arbeitsabläufe. Dabei ist darauf zu achten, daß möglichst deutlich gemacht wird, mit welchen Arbeitsmitteln (=Werkzeugen) welche Arbeitsgegenstände (=Materialien) bearbeitet werden.

Szenarios entstehen aus Gesprächen, die Entwickler mit den Anwendern an deren Arbeitsplätzen führen. Ein wichtiges Kriterium ist hierbei die Verwendung eines gemeinsamen Vokabulars in Bezug auf den Anwendungsbereich. Nur so erlangt der Entwickler den nötigen Einblick in die anfallenden Tätigkeiten und werden dem Anwender seine Aufgaben detailliert ins Bewußtsein gerufen. Dies ermöglicht überhaupt erst das gemeinsame Gespräch über die Aspekte eines zukünftigen computerunterstützten Arbeitens.

3.5.2 Glossare

Die Bedeutung einzelner Begriffe und ihrer Zusammenhänge wird in Glossaren dokumentiert. Auch sie entstehen aus Interviews und Gesprächen mit den Anwendern. Sie werden von den Entwicklern erstellt und in ständigem Abgleich mit den Szenarios fortgeschrieben.

Auch Definitionen von Begriffen wie ‘Fenster’ und ‘Mausklick’ werden in Glossaren festgehalten. Sie sind keine reinen Analyse-Dokumente, sondern werden auch später noch genutzt.

3.5.3 Visionen

Zunächst sind Visionen die Umsetzung der Szenarios in Beschreibungen des zukünftigen Arbeitens mit dem Softwaresystem. Die computerunterstützten Arbeitsabläufe werden so antizipiert und in der Diskussion um die Visionen ergibt sich dann ein Bild der entstehenden Anwendung. Zu einem späteren Zeitpunkt werden dann auch Visionen geschrieben, die Abläufe schildern, welche durch den Einsatz von Software erst möglich werden und somit unabhängig von Szenarios sind.

Sind die Werkzeuge und Materialien mit ihren Umgangsformen ausreichend beschrieben, bilden die Visionen für den Entwickler bereits Spezifikationen der zu entwerfenden technischen Komponenten.

3.5.4 Diagramme

Die verschiedenen Arten von Diagrammen sind unverzichtbare Dokumente sowohl für den Entwurf als auch für die Beschreibung der Architektur eines fertigen Systems. Klassendiagramme zeigen die statische Struktur der Vererbungs- und Benutztbeziehungen, während Objektdiagramme jeweils eine Momentaufnahme der Beziehungen von dynamisch erzeugten Instanzen der Klassen liefern. Anhand

von Interaktionsdiagrammen läßt sich schließlich die Dynamik des ablaufenden Systems selbst visualisieren.

Insbesondere Klassen- und Objektdiagramme eignen sich hervorragend, um sich – im wahrsten Sinne des Wortes – einen Überblick auch über komplexe oder große Komponenten des Systems zu verschaffen. Viele Teile eines Softwaresystems lassen sich auf keine andere Art und Weise anschaulich beschreiben oder kommunizieren.

3.5.5 Quelltexte

Die Quelltexte sind die in der gewählten Programmiersprache abgefaßten Klassenbeschreibungen[1]. In übersetzter, ablauffähiger Form stellen sie das Softwaresystem in seiner Gesamtheit dar.

Auf der Basis der Visionen und Diagramme werden die Quelltexte von den Entwicklern verfaßt. In modernen graphischen Entwicklungsumgebungen steht jedoch auch ein großer Teil von Quellen in Form von Bibliotheken bereits zur Verfügung. Dies betrifft besonders solche Systemkomponenten wie Basis-Datentypen (z.B. Strings, Queues, Listen) oder die Interaktionstypen graphischer Benutzeroberflächen (Knöpfe, Textfelder, Fenster, u.ä.).

3.5.6 Prototypen

Prototypen sind ablauffähige Systeme, die ausgewählte Aspekte der zu entwerfenden Anwendung demonstrieren. Szenarios, Visionen und die anderen Dokumente bilden die Basis eines Prototypen. Er dient dann als Ausgangspunkt für die weitere Diskussion der entstehenden Anwendung, als deren Folge dann erneut Szenarios, Visionen, etc. geschrieben oder überarbeitet werden. Dieser Prototyping-Prozeß führt über Pilotsysteme bis zu voll einsatzfähigen und kompletten Softwaresystemen.

3.6 Quelltexte im Entwicklungsprozeß

Bevor im nächsten Kapitel die Frage erörtert wird, wie Quelltexte sinnvoll klassifiziert werden können, ist näher zu untersuchen, wie die Rolle der Quelltexte im Entwicklungsprozeß aussieht. Hierzu bedarf es der Verdeutlichung, wie mit und an den Klassenbeschreibungen gearbeitet wird und welche Beziehungen zwischen den Quelltexten untereinander und zu anderen Dokumenten bestehen. Die wesentlichen Gesichtspunkte werden nachfolgend aufgeführt.

Bei der Betrachtung der Klassenbeschreibungen muß unterschieden werden zwischen Quellen, die nach WAM erstellt werden und solchen, die in der Entwicklungsumgebung in Form von Bibliotheken bereits vorliegen. Sie wurden im

[1] Gemäß dieser Definition werden die Worte „Quelltext“ (oder auch nur „Quelle“) und „Klassenbeschreibung“ in dieser Arbeit synonym verwendet.

Rahmen eines anderen Softwareentwicklungs-Prozesses erstellt und ihnen liegen deshalb im Zweifel andere Metaphern zugrunde. Dieser andere Prozeß muß zunächst einmal verstanden sein, bevor die Quellen sinnvoll in die Entwicklung nach WAM integriert werden können.

3.6.1 Bearbeitung von Quellen nach WAM

Der wohl wichtigste Aspekt bei der Erstellung bzw. Bearbeitung von Quellen nach WAM ist, daß es sich nicht um eine festgelegte Projektphase mit definiertem Beginn und Ende handelt, bei der alle gewonnenen Erkenntnisse dann lediglich noch kodiert werden. Die Vorgehensweise des Prototypings zeichnet sich ja gerade dadurch aus, daß das einsatzfähige Anwendungssystem evolutionär aus einer Folge von Prototypen entsteht. Diese Prototypen – und damit die zugrunde liegenden Quelltexte – repräsentieren jeweils lediglich einen Ausschnitt, bzw. eine Momentaufnahme des in Entwicklung befindlichen Systems.

Eine wichtige Basis für die Erstellung von Klassenbeschreibungen bilden die Visionen. Aus ihnen lassen sich Materialhierarchien und Werkzeuge ableiten, sodaß die technischen Komponenten – zumindest in grober Form – erstellt werden können. Klassendiagramme leisten dabei eine wertvolle Hilfe durch die anschauliche Form der Darstellung der Beziehungen der Klassen untereinander.

Auch dort, wo es weniger um die Umsetzung fachlicher Begriffe und Konzepte als vielmehr um die Gestaltung der graphischen Benutzerschnittstelle geht, bilden Visionen (aber auch Oberflächenprototypen) einen Ausgangspunkt für die Erstellung von Quelltexten.

Ein weiterer Bereich betrifft die Beschreibung von Klassen, die rein softwaretechnisch motiviert sind. Rein technisch deshalb, weil es auf der fachlichen Seite nichts Äquivalentes gibt. Dies trifft beispielsweise auf Konzepte zur Materialversorgung zu oder auch auf das Parsieren von Texten. Hier spielen Entwurfsmuster eine zentrale Rolle, die Antworten auf viele Fragen der Systemarchitektur liefern können. Dies gilt in gewisser Weise auch für die Entwicklung von Rahmenwerken. Obwohl insbesondere Anwendungsrahmenwerke (vgl. [BGK+95]) durchaus die technische Umsetzung fachlicher Konzepte beinhalten, liegt hierin nicht das primäre Ziel, da ein Rahmenwerk nicht für eine konkrete Anwendung entwickelt wird, sondern längerfristig den Zweck erfüllen soll, den Grad an Wiederverwendung zu erhöhen.

Sobald es ein stabiles Gerüst von Klassen gibt, wird dieses im Rahmen des Prototypingprozesses einerseits Stück für Stück immer weiter verfeinert. Andererseits wachsen bisher voneinander unabhängig entwickelte Teile zusammen. Die Entwickler arbeiten dabei immer an Gruppen von Quelltexten. Ein Werkzeug, dessen Funktionalität erweitert wird, besteht nie aus einer Klasse, sondern immer mindestens aus einer Interaktions- und einer Funktionskomponente. Eine Materialverwaltung besitzt als Schnittstelle zu den Werkzeugen genau eine Klasse. Die einzelnen Teilaufgaben, wie zum Beispiel Materialversorgung oder -erzeugung, werden jedoch von einer Reihe kooperierender Klassen übernommen.

Das unverzichtbare Handwerkszeug bei der Arbeit an den Quelltexten bilden die Editoren und Browser, die die Entwicklungsumgebung zur Verfügung stellt. Es ist im allgemeinen möglich, mehrere Editoren zu öffnen und markierte Textstücke per Drag & Drop zu verschieben. Besonders häufig werden verschiedene Browser benötigt. Sie liefern Informationen zu Ober- und Unterklassen, Routinen und Attributen einer Klasse, aufgeschobenen Routinen und wo diese in der Vererbungshierarchie implementiert sind, etc.

Um zu veranschaulichen, welche Aufgaben solche Werkzeuge übernehmen können, sei hier der *Cross Referencer* der SNiFF+ Entwicklungsumgebung genannt [SNiFF]. Auf der Grundlage der Programmiersprache C++ generiert dieses Werkzeug Informationen zu Referenzen auf oder von Symbolen wie Klassen, Methoden, Variablen, etc. Die Referenzen werden graphisch als Baum dargestellt.

Als Ergebnis ist somit festzuhalten, daß die Arbeit an den Quellen an sich durch eine Vielzahl von Werkzeugen unterstützt wird. Die Informationen werden dabei aus der Syntax und Semantik der verwendeten Programmiersprache gewonnen. Hierbei fällt es jedoch schwer, Strukturen wie Subsysteme, Rahmenwerke, u.ä. zu berücksichtigen, da sich aus den Klassentexten selbst kaum Informationen diesbezüglich generieren lassen. Schließlich lassen sich Aussagen zum Beispiel zu fachlichen Werten oder Zusammenhängen mit anderen Dokumenttypen gar nicht machen.

3.6.2 Bearbeitung anderer Quellen

Im Gegensatz zu den während eines Projektes erstellten Klassen, sind die verwendeten Bibliotheksklassen das Produkt eines anderen Entwicklungsprozesses. Um diese Klassen nutzen zu können, ist zuerst einmal ein genaues Verständnis der Funktionsweise ihrer angebotenen Dienstleistungen nötig. Wie dieses Verständnis erlangt werden kann, wird am Beispiel der Eiffel-Bibliotheken [Mey94] deutlich.

Die Eiffel-Entwicklungsumgebung beinhaltet die Bibliotheken EiffelVision (Klassen für den Bau graphischer Benutzerschnittstellen), EiffelBase (die Basisklassen), EiffelLex (Klassen zur lexikalischen Analyse) und EiffelParse (Klassen zum Parsieren von Texten, die einer festgelegten Syntax genügen). Als schriftliche Dokumentation liegt *ISE Eiffel: The Environment* [ISE95] bei. Dieses Handbuch beschreibt, dem Titel gemäß, die Entwicklungsumgebung selbst. Eine Dokumentation mit Handbuchcharakter zur Nutzung der Bibliotheken ist nicht enthalten. Sie ist jedoch separat zu erwerben [Mey94] und erleichtert die Arbeit in erheblichem Maße. Leider gilt dies nicht für EiffelVision, zu dem es lediglich einen Report gibt [ISE93], der als Handbuch ungeeignet ist.

Um sich ein Bild zu machen, welche Klassen zur Benutzung und Vererbung zur Verfügung stehen, gibt es verschiedene Möglichkeiten zur Analyse der Klassentexte:

- **Ancestors Format:** Zeigt alle Vorfahren (Oberklassen) an, sowie rekursiv deren Vorfahren, etc., bis jeweils herauf zu ANY.

- **Descendants Format:** Zeigt alle Nachfahren (Unterklassen) an, sowie rekursiv deren Nachfahren, etc., bis zu den Blättern des Klassenbaumes.

- **Flat Format:** Zeigt alle Features (Attribute und Routinen) der Klasse, inklusive der geerbten, wobei Redeklarationen und Umbenennungen berücksichtigt werden.

- **Short Format:** Zeigt alle Features der Klasse in einer Kurzform (ohne den Routinenrumpf, aber mit Vor- und Nachbedingungen), ohne dabei die geerbten zu berücksichtigen.

- **Flat-short Format:** Dies ist eine Kombination aus Flat und Short Format. Das Format der einzelnen Features entspricht dem des Short Format, es werden jedoch, wie beim Flat Format, auch geerbte Features angezeigt.

Neben den oben genannten Formaten gibt es noch eine ganze Reihe weiterer (Routines, Attributes, Suppliers, Clients, ...). Die Anzeige erfolgt generell in textueller Form, eine graphische Ausgabe, z.B. der Vererbungsbeziehungen, ist nicht möglich.

Die Nutzung einfacher Strukturen aus den Basis-Bibliotheken geht mit diesen Werkzeugen recht einfach von der Hand. Sehr zeitintensiv und mühsam ist jedoch die Analyse komplexerer Dinge, wie des Parsers und insbesondere der EiffelVision Klassen, für die es überhaupt keine gedruckte Dokumentation gibt. Dennoch ist diese Arbeit unverzichtbar, wenn die Bibliotheken im Rahmen einer Anwendung eingesetzt werden, denn beispielsweise beziehen sich Visionen, wenn es um Knöpfe, Listen und Fenster geht, direkt auf Bibliotheksklassen. Diese Bezüge müssen hergestellt und auch dokumentiert werden.

3.6.3 Gemeinsamkeiten

Unabhängig davon, ob es sich um WAM-Quellen handelt oder um Bibliotheksklassen, die im Rahmen eines Projektes durch Benutzung oder Vererbung Teil des Entwicklungsprozesses werden, gibt es zahllose Bezüge zwischen Quellen und anderen Dokumenten. Diese Informationen sind nicht Bestandteil der Quelltexte, sondern sie verteilen sich auf viele Dokumenttypen.

Ein Bezug in einem Dokument auf einen Quelltext ist dabei immer indirekt (ausgenommen sind hierbei die Prototypen, da sie Quelltexte als integrale Bestandteile enthalten). Steht zum Beispiel in einer Vision: „...nach einem *Klick* auf den *OK-Knopf*...“, so ist zwar einerseits einem Entwickler klar, daß es sich bei *Knopf* um die Klasse `PUSH_B` handelt, anderseits ist aber ein expliziter Verweis von *Knopf* auf den Quelltext `$EIFFEL3/library/vision/oui/widgets/push_b.e` nirgendwo hinterlegt. Diese Information ist – das Wissen, daß Knopf sich in `PUSH_B` wiederfindet vorausgesetzt – zweifellos recht einfach zu beschaffen. Die Werkzeuge der Entwicklungsumgebung (z.B. das Class-Tool) oder des Betriebssystems (der `grep` von UNIX) gewährleisten einen schnellen Zugriff auf den Quelltext `push_b.e`.

Schwierig ist dies jedoch für komplexere Dinge als einen Knopf. Verweist eine Vision zum Beispiel auf ein Werkzeug, etwa einen Editor, so gibt es nicht nur eine Klasse die ihn repräsentiert. Mindestens zwei Klassen, eine Interaktionskomponente (IAK) und eine Funktionskomponente (FK), sind beteiligt. Ist das Werkzeug aus Subwerkzeugen konstruiert, so sind es noch mehr.

Bei allen Fragen, die die Architektur eines Softwareproduktes betreffen und die über Vererbungs- und Benutztbeziehungen hinausgehen, zeigt es sich, daß die üblichen Programmierwerkzeuge hier keine Antworten geben *können*. Das Wissen nämlich, ob eine Klasse Bestandteil eines bestimmten Musterexemplares ist oder ob sie zu einem speziellen Subsystem gehört, kann aus der Klasse selbst nicht hergeleitet werden. Hier sind Informationen über Zusammenhänge nötig, die ein Werkzeug nicht generieren kann, sondern die im Vorfeld dem Werkzeug mitgegeben werden müssen.

4 Klassifizierung im Rahmen von WAM

Nachdem in den beiden vorangegangenen Kapiteln das instanziierbare Klassifizierungsmodell und der WAM-Entwicklungsprozeß eingehend erläutert worden sind, kann nun konkret auf die Klassifizierung im Rahmen von WAM eingegangen werden.

Zunächst wird noch einmal auf das bereits vorgestellte Klassifikationsschema aus 2.2.2 eingegangen, um nun die WAM-Spezifika deutlich zu machen. Es schließt sich dann eine Reihe von Beispielen an, die den Umgang mit diesem Schema vertiefen sollen.

4.1 Die Instanziierung des Meta-Modells

Um ein Schema für die Klassifikation von Quelltexten im Entwicklungsprozeß nach WAM aufstellen zu können, müssen die wesentlichen Eigenschaften herausgearbeitet werden, die eine Klasse in diesem Prozeß kennzeichnen. Dabei ist es zunächst wichtig Oberbegriffe zu finden, die als Facetten dienen können. Die konkreten Eigenschaften können dann in einem weiteren Schritt als Terme den Facetten zugeordnet werden.

Empfehlenswert ist es in jedem Fall, vor dem Einsatz über einen wohldurchdachten Fundus an Termen zu verfügen. Nach den Erfahrungen aus [ST92] ist dieses Vorgehen auf jeden Fall der Strategie vorzuziehen, die Menge der Terme im wesentlichen erst während der Arbeit wachsen zu lassen. Es hat sich gezeigt, daß dann durch Neuklassifizierungen (Terme müssen zum Beispiel gelöscht werden, wenn sich herausstellt, das sie unscharf formuliert waren) ein deutlich größerer Arbeitsaufwand entsteht, als wenn im Vorfeld die Zeit investiert wird, einen großen Teil der Terme bereits festzulegen.

Für den WAM-Prozeß lassen sich folgende Aussagen zu den Eigenschaften von Klassenbeschreibungen, also den Quelltexten, machen:

- Ein wesentliches Merkmal ist ihre Verknüpfung mit anderen **Dokument**en (Visionen, Glossare, Prototypen, Diagrammen). Hierbei sind nicht die Beziehungen zu anderen Quelltext-Dokumenten gemeint; diese werden im folgenden gesondert betrachtet.

- Bedingt durch die objektorientierte Konstruktion nehmen die Quellen bestimmte **Rolle**n innerhalb von Musterexemplaren, Subsystemen, etc. ein.

- Über die gewählten Metaphern stellen die Quellen **W**erkzeuge, **A**spekte oder **M**aterialien, beziehungsweise einen Teil davon, dar. Hierzu gehören auch Begriffe wie IAK, FK, etc.

- Besonderer Wert wird im WAM-Prozeß auf die Modellierung der fachlichen Zusammenhänge gelegt. Durch diese anwendungsorientierte System-

entwicklung treten die spezifischen Begriffe der **Domäne** besonders in den Vordergrund.

Aus diesen Feststellungen ergibt sich dann das Schema, das bereits in Kapitel 2 kurz vorgestellt wurde:

Facette	**Terme** (Auswahl)
Dokument	Vision, Glossar, Prototyp, Klassendiagramm, sowie Bezeichnungen aller Dokumente, die in Beziehung zu einer Klassenbeschreibung stehen,...
Rolle	Beobachter, Interpreter, Rahmenwerk, Fassade, Schnittstelle,...
WAM	Werkzeug, Automat, Aspekt, Material, Umgebung, IAK, FK,...
Domäne	Quelle, Index, Facette, Term,...

Hier finden sich genau jene Oberbegriffe als Facetten wieder, die oben aus den wesentlichen Eigenschaften der Quelltexte abgeleitet wurden. Diese Aufstellung enthält zur Vereinfachung lediglich die für den hier betrachteten Prozeß spezifischen Facetten und Terme. Unbestreitbar ist beispielsweise auch eine Klassifizierung der ADT's aus verfügbaren Bibliotheken für die Entwickler wichtig. Die hierfür notwendigen Facetten sind ausführlich in [ST92] beschrieben. Im Rahmen dieser Arbeit liegt der Schwerpunkt jedoch auf der Betrachtung von WAM, sodaß auf eine vollständige Darstellung aller für die Softwareentwicklung relevanter Facetten bewußt verzichtet wird.

Es steht damit nun ein konkretes Modell und Klassifikationsschema zur Verfügung, welches im folgenden an einigen Beispielen erprobt werden wird.

4.2 Beispiele zur Klassifikation

Die nun folgenden beispielhaften Klassifikationen werden vorgenommen an Klassen, die dem Werkzeug entstammen, welches als Teil dieser Arbeit entworfen und implementiert wurde. Sie sollen zum besseren Verständnis des vorgestellten Klassifikationsschemas beitragen und dienen gleichzeitig als Anregung für die eigene Entwicklungsarbeit.

4.2.1 Rahmenwerk für EiffelVision

Die Abb. 4-1 zeigt einen Teil des Rahmenwerkes für graphische Applikationen

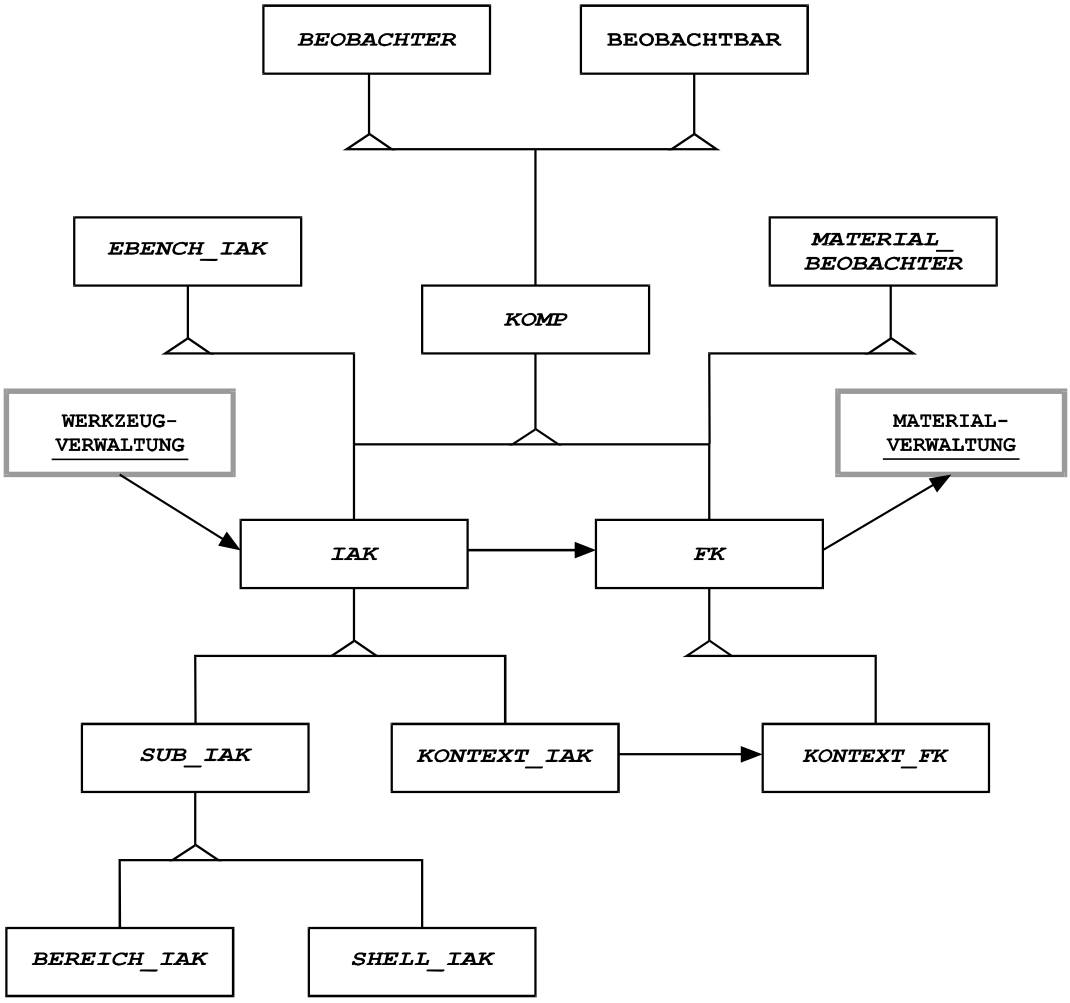


Abb. 4-1: Das Rahmenwerk für EiffelVision

auf der Basis von EiffelVision. Für alle Klassen dieses Rahmenwerkes findet sich ganz offensichtlich folgende triviale Klassifizierung:

alle Klassen des Rahmenwerkes				
Facette	**Dokument**	**Rolle**	**WAM**	**Domäne**
Terme	Klassen-diagramm ‘Rahmen-werk für Eif-felVision’	Rahmenwerk		

Werden nun Stück für Stück immer kleinere Ausschnitte des Rahmenwerkes betrachtet, so wird mit diesem top-down Verfahren schließlich die Klassifizierung einzelner Klassen erreicht.

Zwischenschritte sind beispielsweise:

alle IAK’s				
Facette	**Dokument**	**Rolle**	**WAM**	**Domäne**
Terme	Klassen-diagramm ‘Rahmen-werk für Eif-felVision’	Rahmenwerk	Werkzeug IAK	

oder

BEOBACHTER, BEOBACHTBAR, MATERIAL_BEOBACHTER				
Facette	**Dokument**	**Rolle**	**WAM**	**Domäne**
Terme	Klassen-diagramm ‘Rahmen-werk für Eif-felVision’	Rahmenwerk Beobachter Ereignis-mechanismus		

Am Ende steht dann als einzelne Klasse zum Beispiel:

KONTEXT_IAK				
Facette	**Dokument**	**Rolle**	**WAM**	**Domäne**
Terme	Klassen-diagramm ‘Rahmen-werk für Eif-felVision’	Rahmenwerk Schnittstelle	Werkzeug IAK Kontext-IAK	

Bemerkenswert ist an dieser Stelle, daß beim Klassifizieren Facetten sowohl leer sein können, als auch mehrere Terme enthalten können. Da es sich nicht um ein anwendungsspezifisches Rahmenwerk handelt, muß die Facette Domäne logischerweise leer bleiben. Anderseits sind mehrere Terme dort erforderlich, wo verschiedene Aspekte einer Klasse von Bedeutung sind (eine Klasse in ihrer Eigenschaft als Werkzeugklasse oder spezifischer als IAK, eine Schnittstelle oder ein Teil dieses Rahmenwerkes, etc.).

4.2.2 Materialverwaltung

In 2.2.1 wurde bereits die Klasse `MAGAZIN` im Rahmen der Materialverwaltung erwähnt. Abb. 4-2 zeigt noch einmal den Zusammenhang, in dem die Klasse implementiert wurde.

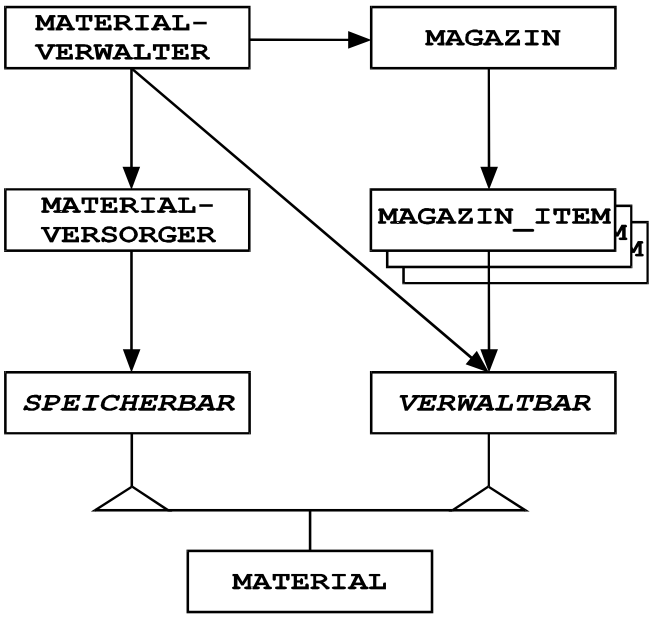


Abb. 4-2: Die Materialverwaltung

Die Materialverwaltung wurde zwar entsprechend den WAM-Metaphern konstruiert, die Klasse `Magazin` ist jedoch für die Klassifizierung genauso wenig WAM-spezifisch, wie sie in irgendeiner Form von der Domäne abhängig wäre. Diese beiden Facetten bleiben deshalb leer. Relevant ist bei ihr die Tatsache, daß sie als Verwaltermagazin im Rahmen der Materialverwaltung fungiert, sodaß folgendermaßen klassifiziert werden kann:

MAGAZIN				
Facette	**Dokument**	**Rolle**	**WAM**	**Domäne**
Terme	Klassen-diagramm 'Materialver-waltung'	Subsystem Materialver-waltung Verwalter-magazin		

4.2.3 Materialien

Für Materialklassen ist die Facette Domäne von besonderer Bedeutung. Die Abb. 4-3 zeigt einen kleinen Ausschnitt aus den Materialien des hier entwickelten Werkzeuges.

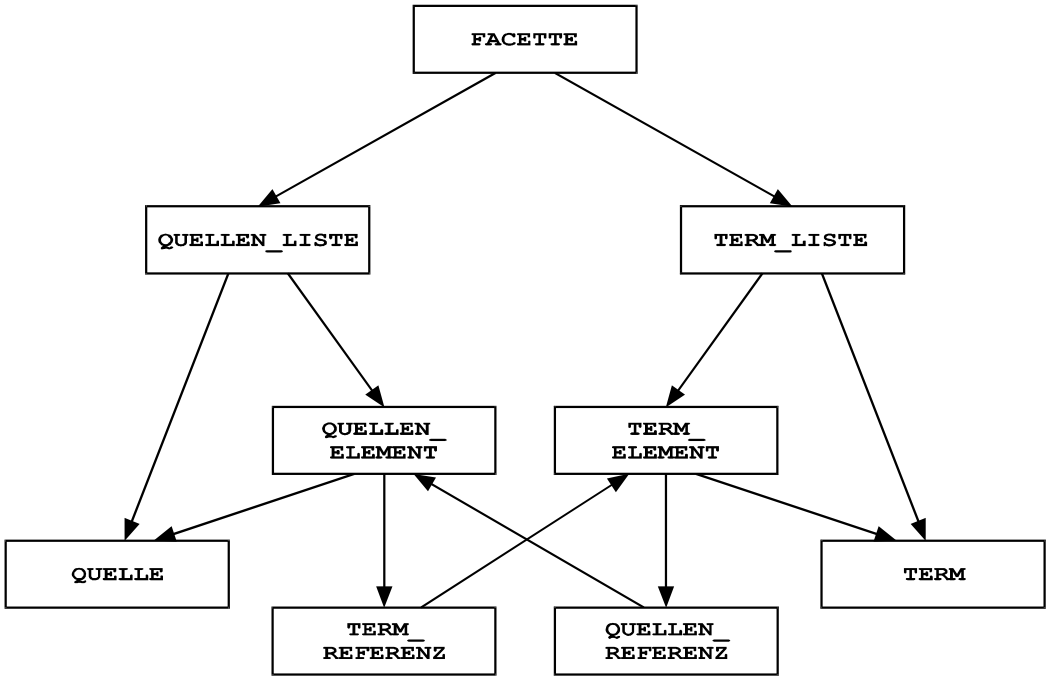


Abb. 4-3: Materialien

Signifikant für die Klassifizierung der Quelltexte der Materialien `QUELLE` und `TERM` ist:

QUELLE				
Facette	**Dokument**	**Rolle**	**WAM**	**Domäne**
Terme	Klassen-diagramm ‘Materialien’		Material	Eiffelquell-text Textdatei

und

TERM				
Facette	**Dokument**	**Rolle**	**WAM**	**Domäne**
Terme	Klassen-diagramm ‘Materialien’		Material	Index Term

4.2.4 Parser für Eiffel-Quelltexte

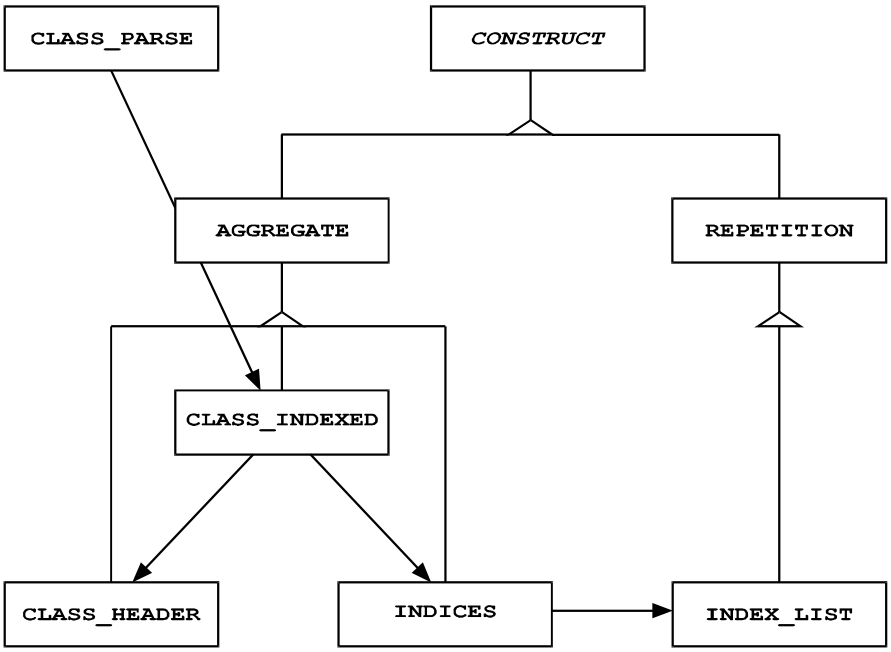


Abb. 4-4: Parser

Als letztes Beispiel sei der verwendete Parser für die Eiffel-Quelltexte genannt. Die Abb. 4-4 dokumentiert einen kleinen Ausschnitt des zugehörigen Klassendiagrammes.

Dabei ergibt sich dann für `CLASS_PARSE` und `CLASS_INDEXED`:

CLASS_PARSE				
Facette	**Dokument**	**Rolle**	**WAM**	**Domäne**
Terme	Klassen-diagramm ‘Parser’	Schnittstelle Parser Interpreter	Automat	Indices Syntaxfehler

CLASS_INDEXED				
Facette	**Dokument**	**Rolle**	**WAM**	**Domäne**
Terme	Klassen-diagramm ‘Parser’	Schnittstelle Parser Interpreter	Automat	Eiffelsyntax Indexing

5 Das Werkzeug

Nachdem das Klassifizierungsmodell ausführlich erläutert ist, wird im folgenden ein Softwarewerkzeug vorgestellt, welches die Entwicklungsarbeit mit Hilfe dieses Modells unterstützen kann. Es wird dargestellt, welche Aufgaben bei der Klassifizierung werkzeugunterstützt durchgeführt werden können und auch wie dies geschehen kann.

Eingehend geschildert wird in diesem Kapitel die Architektur des Werkzeuges. Eine konkrete Beschreibung zur Handhabung und Installation des implementierten Prototypen findet sich im Anhang.

5.1 Anforderungen

Bevor formuliert wird, welche Anforderungen das Werkzeug erfüllen soll, müssen Überlegungen zu den Voraussetzungen für das Softwareprojekt angestellt werden. Hierzu gehören insbesondere die Wahl einer Programmiersprache und einer Entwicklungsumgebung, denn diese Wahl hat starken Einfluß darauf, welche Funktionalität unter welcher Benutzeroberfläche überhaupt sinnvoll entworfen und dann auch entsprechend implementiert werden kann. Der Begriff Funktionalität steht hierbei lediglich dafür, welche Aufgaben mit dem Werkzeug ausgeführt werden können.

5.1.1 Voraussetzungen

Bis vor kurzem hätte die Frage nach einer adäquaten Programmiersprache nur mit C++ beantwortet werden können (vgl. [KGZ93, S. 63ff.]). Klaus Kilberth, Guido Gryczan und Heinz Züllighoven bieten zwar Eiffel als Alternative, schränken aber aufgrund der nicht vollständigen und unausgereiften Entwicklungsumgebung die Benutzbarkeit ein. Diese Einschränkungen haben sich jedoch inzwischen mit Erscheinen der Version 3 der ISE Entwicklungsumgebung [ISE95] erledigt. Zusätzlich steht mit der Arbeit von Jens Ginzel [Gin95] die Basis für ein Rahmenwerk zur Verfügung, um Werkzeuge nach WAM konstruieren zu können.

Folglich steht mit Eiffel also eine echte Alternative zu C++ zu Verfügung. Für die Entscheidung zugunsten von Eiffel sprechen drei Gründe: Zum ersten ist Eiffel eine rein objektorientierte Sprache, die im Gegensatz zu C++ keine prozeduralen Strukturen zuläßt. Zweitens bietet Eiffel in seiner Syntax ein `indexing`-Konstrukt, das sich bei Klassifizierungen nutzen läßt, und drittens sind bisher im Arbeitsbereich Softwaretechnik kaum Projekte in Eiffel realisiert worden, sodaß es einfach reizt, neue und interessante Wege zu beschreiten.

Sowohl der Entwurf und die Implementation als auch das fertige Werkzeug sind unabhängig von einer konkreten Hardwareumgebung, womit die Anwendung auch portierbar ist.

5.1.2 Funktionalität

Die wesentlichen Aufgaben des Werkzeuges lassen sich direkt aus den vorangegangenen Kapiteln ableiten:

- Facetten und zugehörige Terme müssen eingefügt und gelöscht werden können. Dies sollte an zentraler Stelle geschehen, damit nur mit einer vorhanden Kombinationen aus Facette und Term indiziert werden kann.

- Die Indizierung beliebiger Quellen aus frei wählbaren Verzeichnissen soll möglich sein. Es sollten auch Gruppen von Quellen gemeinsam indiziert werden können.

- Quellen müssen über beliebige logische Verknüpfungen von Indices gesucht werden können.

Hinzu kommen noch wichtige Anforderungen an die Architektur, die im Zusammenhang mit einer späteren Weiterentwicklung stehen:

- Der Zugriff auf Materialien muß in einer Art und Weise gekapselt sein, die eine spätere Anbindung an eine Datenbank leicht möglich macht.

- Das Werkzeug sollte es erleichtern, zu einem späteren Zeitpunkt komplett mehrbenutzerfähig eingesetzt werden zu können. (Der implementierte Prototyp geht nur einen Schritt in diese Richtung, indem er allen Benutzern das Suchen erlaubt, aber nur einem Benutzer das Indizieren oder Einfügen/Löschen von Facetten und Termen.)

Als Programmiersprache erlaubt Eiffel mit seiner Syntax noch zusätzliche Funktionen, die zum Beispiel mit C++ so nicht zu realisieren wären. Eiffel kennt ein syntaktisches Konstrukt namens indexing, das am Beginn einer Klassenbeschreibung stehen kann. In [Mey91, S. 49] heißt es dazu:

> „The optional Indexing part has no direct effect on the semantics of the class. It serves to associate information with the class, for use by tools for archiving and retrieving classes based on their properties.“

Und genau das ist die Aufgabe des zu entwickelnden Werkzeuges: Klassenbeschreibungen auf der Basis festgelegter Eigenschaften zu indizieren, zu suchen und wiederzufinden. Damit lassen sich die zusätzlichen Aufgaben wie folgt formulieren:

- Klassen können von den Entwicklern direkt im Quelltext indiziert werden. Das Werkzeug muß dann in der Lage sein, diese Indices zu lesen und in den Datenbestand zu übernehmen.

- Das Werkzeug sollte eine Möglichkeit bieten, die Indices aus dem Datenbestand direkt in die Quelltexte zu schreiben. Damit sind die Informationen zur Klassifizierung auch jederzeit in den Klassenbeschreibungen selbst

präsent und gehen auch dann nicht verloren, wenn die Quellen auf andere Systeme portiert werden.

Die Abb. 5-1 zeigt einen Ausschnitt des Indexing-Teils der Klasse `MAGAZIN`. An der Darstellung wird deutlich, auf welch einfache Art und Weise die Syntax von Eiffel eine Zuordnung zu Facetten und Termen unterstützt.

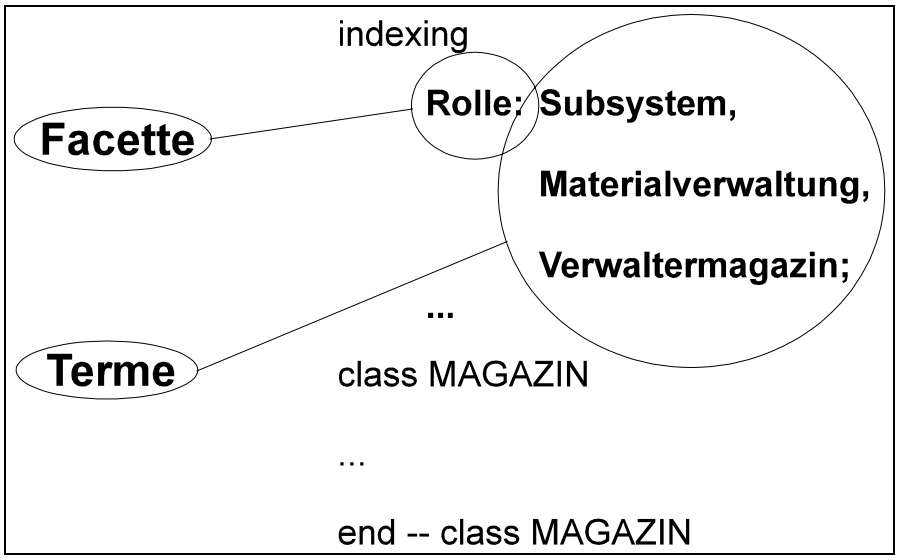


Abb. 5-1: Indexing Teil einer Eiffel-Klasse

Der Anforderungskatalog, der hier formuliert wurde, enthält im wesentlichen nur solche Punkte, die direkt mit den Aufgaben der Klassifizierung zu tun haben. Darüber hinaus sollen auch Anforderungen erfüllt werden, die generell an Softwareprodukte gestellt werden müssen. Eine Aufstellung findet sich zum Beispiel in [PB96, S. 9ff.] als Softwarequalitätsmerkmale.

Diese Merkmale werden, soweit es der (insbesondere zeitliche) Rahmen dieser Diplomarbeit zuläßt, mit berücksichtigt. Es wird jedoch im weiteren nicht mehr explizit darauf eingegangen. Indirekt folgen jedoch noch etliche Verweise auf Qualitätsmerkmale, da ein sorgfältig durchgeführter Entwurf nach den Metaphern von Werkzeug und Material die Wahrung eines hohen Qualitätsstandards impliziert.

5.1.3 Benutzeroberfläche

Die Anforderungen an die Benutzeroberfläche lassen sich grob in zwei Punkten formulieren:

- Die Schnittstelle des Benutzers zur Handhabung des Werkzeuges soll eine graphische sein. Die Interaktion wird dann über die typischen Bedienelemente wie Knöpfe, Textfelder, Rollbalken, etc. erfolgen. Abb. 5-2 zeigt das Beispiel einer graphischen Benutzerschnittstelle des Fenstersystems Motif.

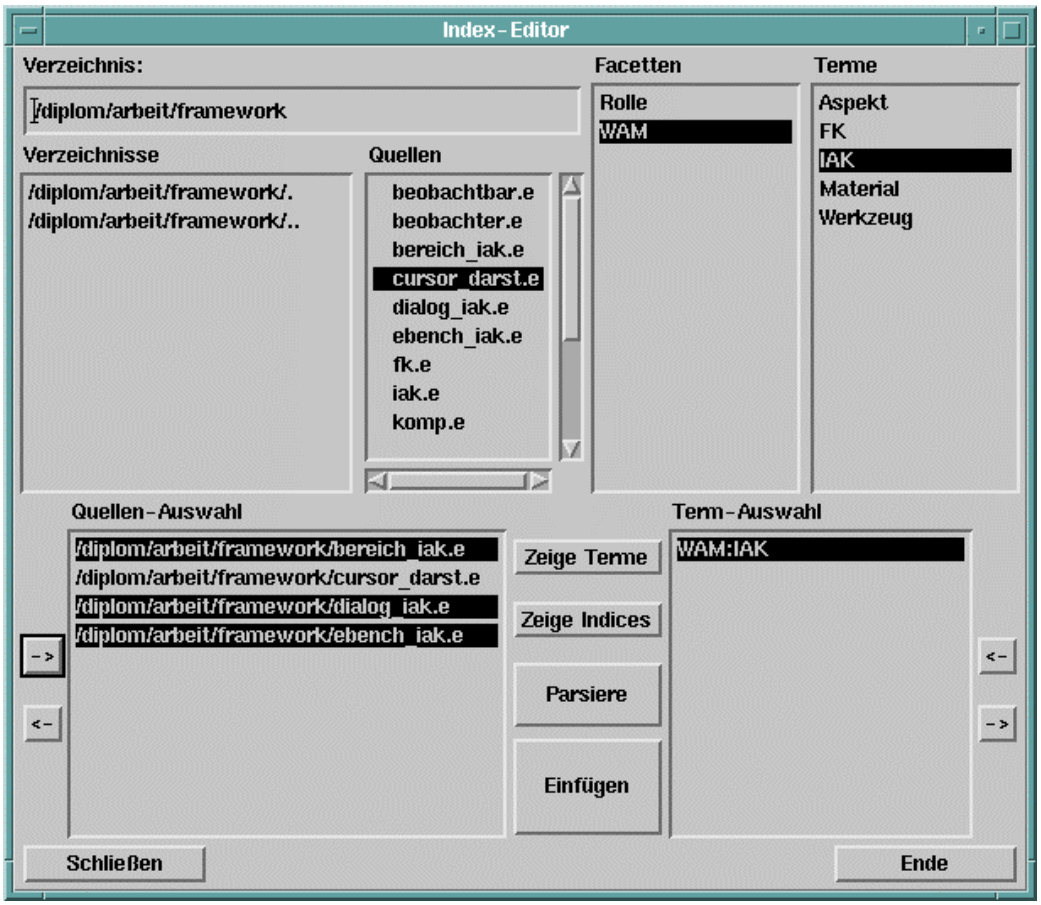


Abb. 5-2: Graphische Benutzerschnittstelle unter Motif

- Das Werkzeug muß einfach an verschiedene Fenstersysteme angepaßt werden können, um die Portierbarkeit zu gewährleisten.

Die Implementation des Werkzeuges wird unter Linux mit dem Fenstersystem Motif erfolgen. Die Portierbarkeit wird durch die Klassenbibliothek EiffelVision gewährleistet, die alle Aufrufe des zugrunde liegenden Fenstersystems kapselt. Bei einem Wechsel zum Beispiel von Motif zu Windows ist deshalb nur in einer einzigen Klasse eine Vererbungsbeziehung zu ändern.

5.2 Architektur

Das Entwurfsdokument beschreibt eine Reihe von Mikroarchitekturen, die im wesentlichen auf den von Riehle in [Rie95] entwickelten Mustern aufbauen. Die ersten beiden Abschnitte dieses Kapitels stellen deshalb diese Muster im Überblick vor. Zu beachten bleibt, daß es bei der Umsetzung der Muster in konkrete Musterexemplare als Systemkomponenten zu Abweichungen von den von Riehle entwickelten Mikroarchitekturen kommt. Dies betrifft insbesondere die Muster *Werkzeugkoordinator* und *Materialcontainer*. Diese Entwurfsentscheidungen werden bei der Diskussion besonders berücksichtigt.

5.2.1 Interpretations- und Gestaltungsmuster

Den Ausgangspunkt für die weitere Entwicklung des Entwurfs bilden die Interpretations- und Gestaltungsmuster für die Werkzeug und Material Metapher. Sie dienen als Basis für die Entwurfsmuster des nächsten Abschnitts.

Abb. 5-3 zeigt die Zusammenhänge zwischen den Interpretations- und Gestaltungsmustern. Die *Umgebung* bildet als erstes Muster mit ihrem Kontext die Grundlage für alle weiteren. Sie bildet den Rahmen, in dem *Werkzeug* und *Material* ihren Platz finden. *Werkzeug* und *Material* sind dabei nahezu identisch mit ihren namensgleichen Metaphern Werkzeug (siehe 3.2.1) und Material (siehe 3.2.2). Im *Verwendungszusammenhang* findet sich die Metapher des Aspekts (siehe 3.2.3) wieder. [Rie95, S. 38ff]

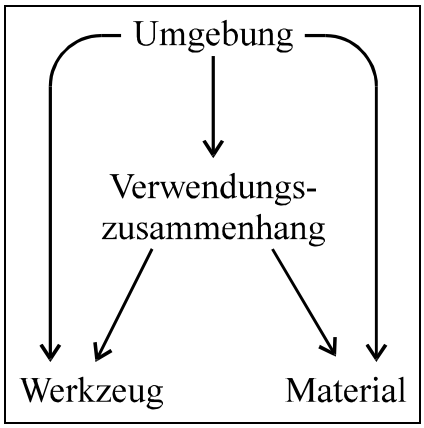


Abb. 5-3: Die Interpretationsmuster für die Werkzeug und Material Metapher

Nach den bisherigen Erörterungen ist es nun möglich, einen Blick auf das entstehende Softwaresystem zu werfen. Die Abb. 5-4 zeigt ein erstes und noch unvollständiges Klassendiagramm des Systems.

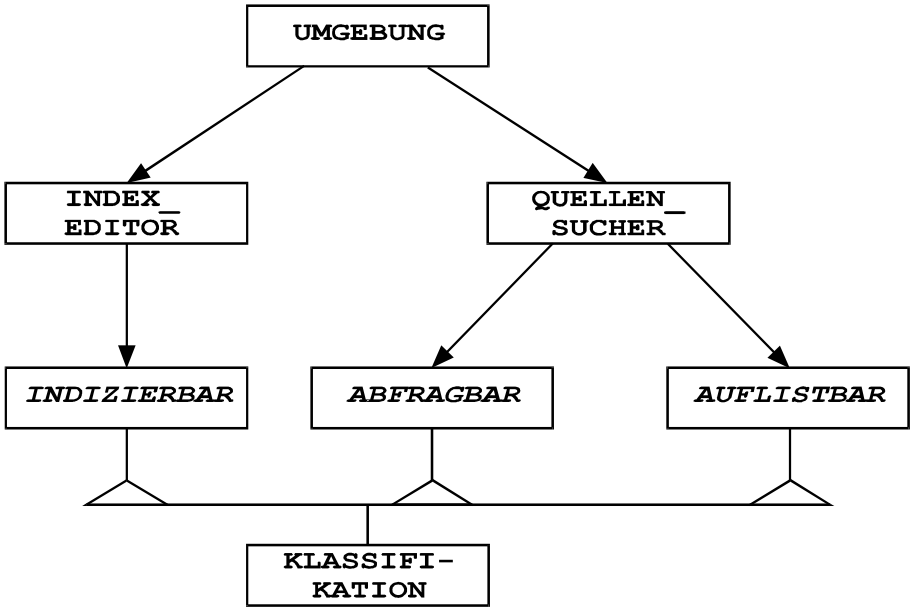


Abb. 5-4: Noch unvollständiges, vorläufiges Klassendiagramm

An oberster Stelle steht die `UMGEBUNG`. Sie benutzt zwei Werkzeuge, `INDEX_EDITOR` und `QUELLEN_SUCHER`. Die Existenz dieser beiden Werkzeuge ergibt sich direkt aus den Anforderungen, wie in 5.1.2 dargelegt. Der `INDEX_EDITOR` ermöglicht die Pflege des Datenbestandes, also das Einfügen und Löschen der Indices von Eiffel-Quelltexten. Mit Hilfe des `QUELLEN_SUCHERs` wird die gezielte Suche nach ganz bestimmten Quellen realisiert.

Der Datenbestand wird repräsentiert durch die Materialklasse `KLASSIFIKATION`. Diese Klasse wird den Umgang mit Facetten und Termen implementieren, sowie den Zugriff auf die Indices der Quellen abbilden.

Zwischen den Werkzeugen und dem Material sind die Aspekte dargestellt, die die Oberklassen für die Materialien bilden und über die die Werkzeuge auf die Materialien zugreifen. Diese (abstrakten) Klassen sind hergeleitet aus den verschiedenen Arbeitstätigkeiten, die den *Verwendungszusammenhang* zwischen Werkzeugen und Material bestimmen.

Der `INDEX_EDITOR` arbeitet auf der `KLASSIFIKATION`, indem er sie als sortierte Liste betrachtet, in der zu bestimmten Quellen Indices eingefügt und ge-

löscht werden können. Deshalb der Aspekt *INDIZIERBAR*[2]. Der Inhalt dieser Liste ist auch nachschlagbar, also AUFLISTBAR. Diesen Aspekt nutzt der QUELLEN_SUCHER, um zum Suchen eine Liste verfügbarer Facetten und Terme anzubieten. Der dritte wichtige Aspekt, ABFRAGBAR, wird benötigt, um Zugriff auf die Relation zwischen Quellen und Indices zu erhalten.

5.2.2 Entwurfsmuster

Die Grundlage für den softwaretechnischen Entwurf bilden die Entwurfsmuster für die Werkzeug und Material Metapher. Sie dienen als Ausgangspunkt für die objektorientierte Konstruktion und sind ein wichtiger Bestandteil der Dokumentation der Architektur des Softwaresystems.

Abb. 5-5 zeigt die verwendeten Muster in ihrem kontextuellem Zusammenhang. Die Interpretations- und Gestaltungsmuster sind hier noch einmal (grau dargestellt) mit aufgeführt, um deutlich zu machen, daß sie die Basis für die Entwurfsmuster bilden und diese den dort gesteckten Rahmen nicht verlassen.

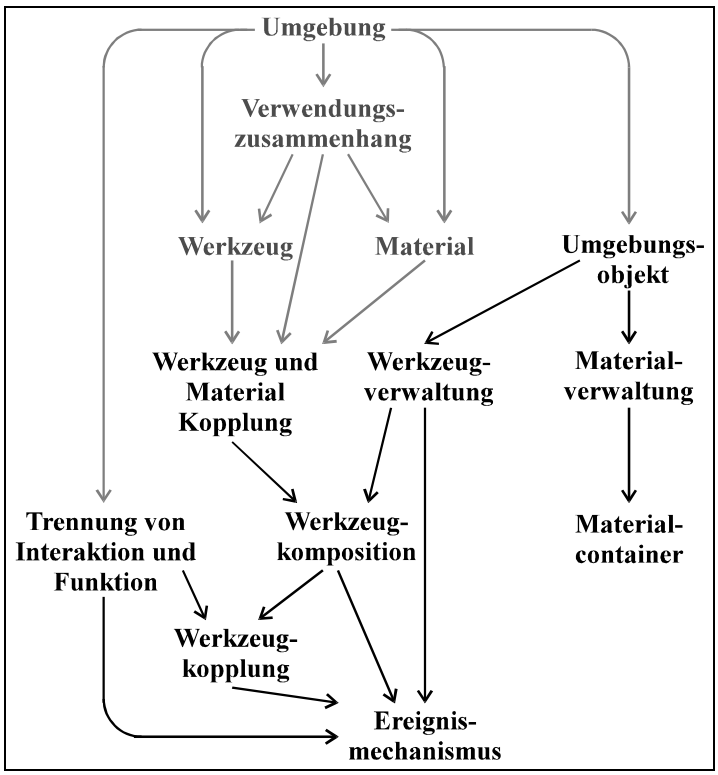


Abb. 5-5: Alle Entwurfsmuster im Überblick

[2] Das Wort Index sowie seine verschiedenen grammatikalischen Formen, wie Indices, indiziert, indizierbar, etc., sind in diesem Text mit zwei verschiedenen Bedeutungen behaftet. Abhängig vom Zusammenhang meint Index entweder den Index-Eintrag im Quelltext einer Eiffelklasse oder - wie hier - die Tatsache, daß Listen aufgrund ihrer Eigenschaft sortiert zu sein, Indices besitzen.

Vergleicht man die Abb. 5-5 mit der ursprünglichen Abbildung (hier wiedergegeben in Abb. 5-6) in [Rie95, S. 32], so fällt auf, daß hier *Werkzeugkoordinator* durch *Werkzeugverwaltung* ersetzt wurde und daß *Materialcontainer* in seinem Kontext nicht mehr auf *Werkzeugverwaltung*, bzw. ursprünglich *Werkzeugkoordinator* basiert.

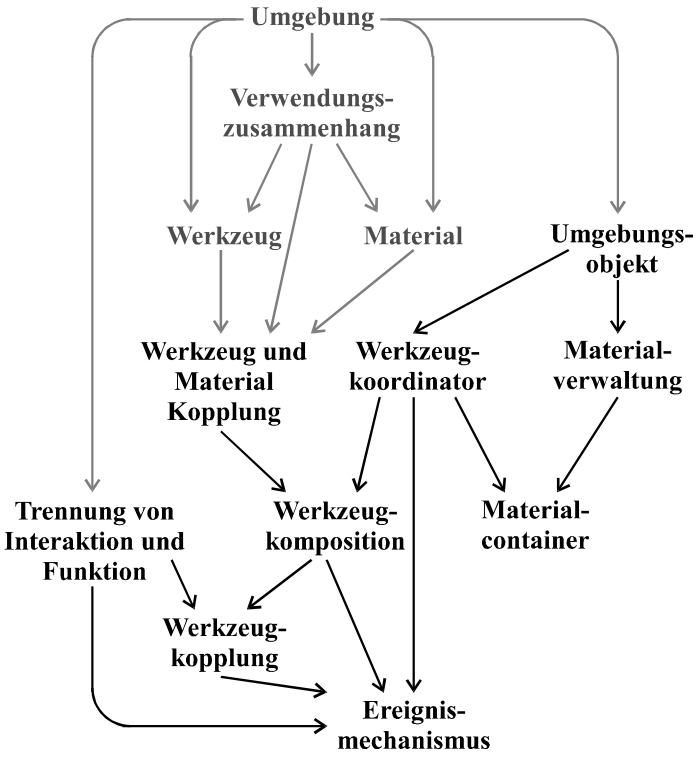


Abb. 5-6: Die Entwurfsmuster nach Riehle

Diese Änderungen werden motiviert durch die Materialien, die das Indexing-Werkzeug bearbeiten soll. Im Gegensatz zu den typischen Büroanwendungen, in denen eine Vielzahl von Materialien (Ordner, Notizen, Briefe, Adressen, Termine, etc.) durch ein dichtes Netz von Abhängigkeiten miteinander verknüpft sind, ist die Anzahl der Materialien des Indexing-Werkzeuges klein und sehr überschaubar. Es gibt lediglich pro Facette ein persistentes Materialobjekt (siehe 5.2.3), sowie eine Reihe transienter Materialien, die von den diversen Auflisterobjekten benutzt werden. Die (ebenfalls transiente) `KLASSIFIKATION` bildet mit den Facettenobjekten auch die einzigen Kombinationen voneinander abhängiger Materialien, die konsistent zu halten sind.

Es gibt also lediglich einen einzigen *Materialcontainer*. Dieser ist das Magazin des Materialverwalters, für den kein eigener Werkzeugkoordinator eingeführt wird. Stattdessen wird der *Ereignismechanismus* auch für die Kommunikation

zwischen Werkzeugen und *Materialverwalter* eingesetzt und über die *Umgebung* die Aufrechterhaltung der Konsistenzbedingungen gewährleistet.

5.2.3 Materialien

Bevor in den nächsten Abschnitten detailliert die Musterexemplare zur Werkzeugkomposition und -integration beschrieben werden, soll zunächst noch einmal etwas genauer auf den Entwurf der Materialklassen eingegangen werden, ohne daß der Entwurf in allen Einzelheiten von der fachlichen Modellierung bis zur Implementation eines Klassenmodells durchgeführt wird. Diese Vorgehensweise erscheint hier als sachgerecht [vgl. KGZ93, S. 14ff.].

Hier wird deshalb lediglich als Resultat das technische Klassenmodell aus Abb. 5-7 vorgestellt und es werden einige wesentliche Überlegungen nachvollzogen, die zu seiner Entstehung geführt haben.

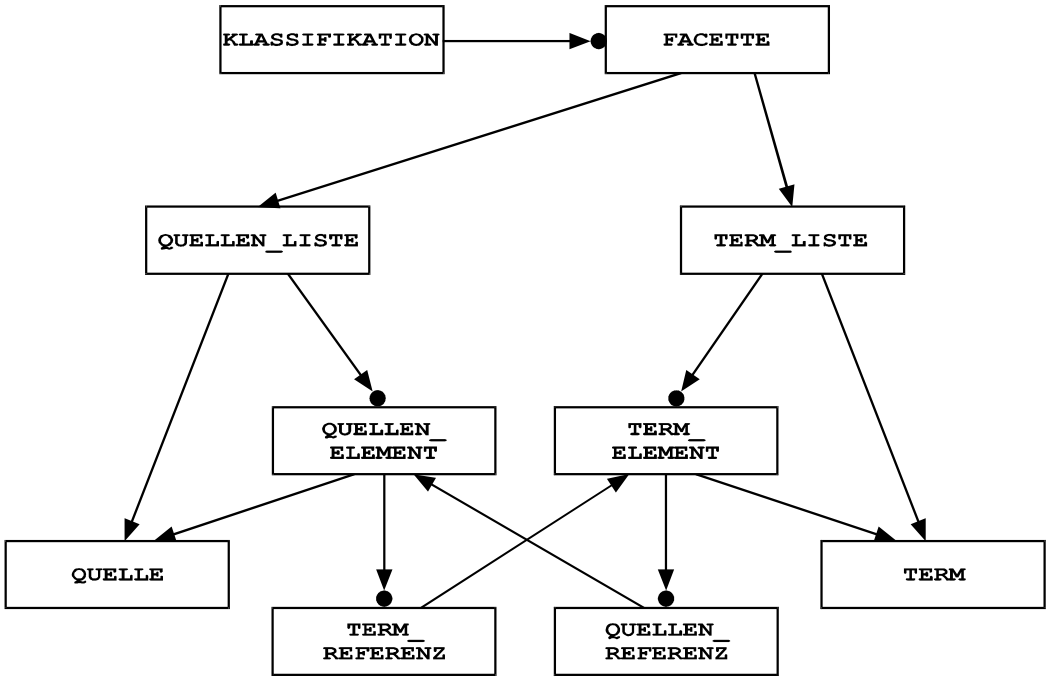


Abb. 5-7: Das Klassenmodell für die Materialien

Als zentrale Arbeitsgegenstände ergeben sich bei der Analyse Facetten, Terme und Quelltexte, die mit den Klassen FACETTE, TERM und QUELLE modelliert werden. Jede Facette enthält eine Reihe von Termen (TERM_LISTE), die jeweils als TERM zusammen mit einer beliebigen Anzahl von Referenzen auf Quellen (QUELLEN_REFERENZ) als TERM_ELEMENT entworfen werden. Auf dieselbe Art finden auch die Quellen ihren Platz innerhalb einer Facette.

Das vollständige Klassendiagramm in Abb. 5-7 verdeutlicht die Symmetrie des Entwurfs. Diese spiegelbildliche Architektur ermöglicht es, Abfragen der Art ‘Welche Quellen besitzen die Indices A und B’ und ‘Welche Indices enthält die Quelle X’ auf dieselbe Weise zu implementieren.

Da das Klassifizieren und Suchen von Quelltexten auf der Basis von n (und nicht 1) Facetten durchgeführt wird, ist es nötig einen Container einzuführen, der die Handhabung dieser Facetten ermöglicht. Dies geschieht mit der Klasse KLASSIFIKATION. Diese Klasse braucht jedoch im Gegensatz zu den Facetten nicht persistent zu sein, da sie lediglich Methoden zur Verfügung stellt, um die Summe der Facetten zu handhaben, jedoch keine ‘eigenen’ Daten enthält.

Die Werkzeuge des Systems werden dann über die entsprechenden Aspekte fast ausschließlich mit dem Material KLASSIFIKATION und nicht mit FACETTE arbeiten.

Nicht im Diagramm dargestellt sind eine Reihe weiterer transienter Materialien. Es handelt sich dabei im wesentlichen um Listen, die von den diversen Editoren und Auflistern benutzt werden, um zum Beispiel Auswahllisten zu verwalten. Dies sind zumeist einfache Klassen, die als Erben von LINKED_LIST oder SORTED_TWO_WAY_LIST (siehe [Mey94, S. 299ff.]) implementiert sind und über die Aspekte INDIZIERBAR und AUFLISTBAR benutzt werden können.

5.2.4 Werkzeug und Material Kopplung

Die Kopplung von Werkzeugen und Materialien erfolgt über Aspekte. Sie wird realisiert durch Aspektklassen. Die Schnittstelle einer Materialklasse, die von verschiedenen Werkzeugen benutzt werden kann, wird dann über Mehrfachvererbung von Aspektklassen zusammengesetzt [Rie95, S. 46ff.].

Für das hier zu entwerfende System stellt sich zum Beispiel die Frage, wie die beiden Werkzeuge INDEX_EDITOR und QUELLEN_SUCHER auf das Material KLASSIFIKATION zugreifen. Während QUELLEN_SUCHER ein lediglich sondierendes - wenn auch sehr komplexes - Werkzeug ist, werden mit dem INDEX_EDITOR Arbeitstätigkeiten wie Einfügen, Löschen und Verändern ausgeführt.

Die beiden Aspektklassen, die den Zusammenhang zwischen INDEX_EDITOR und QUELLEN_SUCHER auf der einen Seite und KLASSIFIKATION auf der anderen darstellen, zeigt Abb. 5-8.

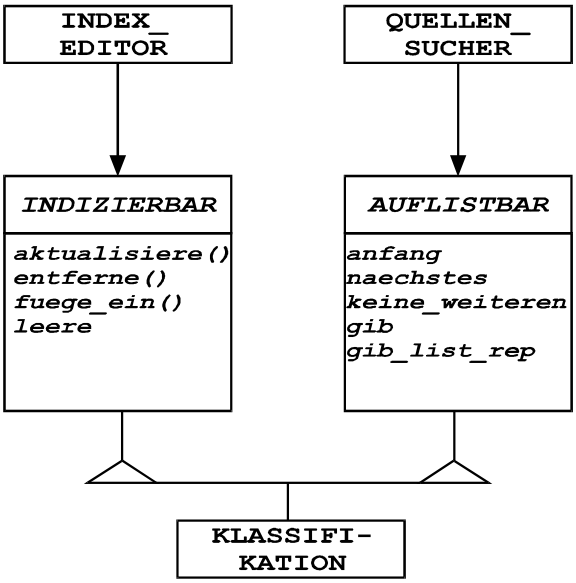


Abb. 5-8: Kopplung der Werkzeugklassen INDEX_EDITOR und QUELLEN_SUCHER mit dem Material KLASSIFIKATION über die Aspekte *INDIZIERBAR* und *AUFLISTBAR*

Die abstrakten Operationen werden in den Aspektklassen als aufgeschobene Routinen (in der Eiffel-Terminologie: deferred features) definiert. Die für die interne Datenrepräsentation ‘passende’ Implementierung wird dann in der Materialklasse vorgenommen.

Vorstehende Abbildung beschreibt damit den Fall, in dem mehrere Werkzeuge über verschiedene Aspekte das gleiche (in dem hier zu entwickelndem System wird es sogar dasselbe sein) Material bearbeiten können. Über genau diese beiden Aspekte werden die Werkzeuge aber auch mit mehreren Auswahllisten arbeiten. Ein Werkzeug kann also über den gleichen Aspekt zwei unterschiedliche Materialien bearbeiten.

Bei der Diskussion von Design-Mustern für die Organisation von Klassenhierarchien unterscheidet Erich Gamma zwischen rein und partiell abstrakten Klassen [Gam92, S. 97]. Rein abstrakte Klassen, also Klassen bei denen alle Routinen abstrakt sind, dienen „dem ausschließlichem Zweck, ein standardisiertes Protokoll zu spezifizieren“. Partiell abstrakte Klassen, die bereits auch effektive Routinen enthalten, stellen hingegen „ein Gerüst für die Implementation abgeleiteter Klassen zur Verfügung“.

Obwohl Aspektklassen dem ausschließlichen Zweck dienen, die Schnittstelle zwischen Werkzeugen und Materialien zu bilden und nicht als Gerüst für die Implementation von Materialien gedacht sind, sind sie deshalb nicht notwendigerweise rein abstrakt. So wird die Aspektklasse *INDIZIERBAR* neben den oben aufgeführten abstrakten Routinen *aktualisiere()*, *entferne()*, *fuege_ein()* und *leere* auch die effektiven Routinen `aktualisiere_alle()`, `entferne_alle()` und `fuege_ein_alle()` enthalten. Sie lassen sich allein auf der Basis der abstrakten Routinen implementieren und schränken die Eigenschaft des Aspektes, Schnittstelle eine Materialklasse zur Benutzung durch Werkzeuge zu sein, in keiner Weise ein.

5.2.5 Werkzeugkomposition

Komplexe Werkzeuge können Teilaufgaben an Subwerkzeuge delegieren. Rekursiv entsteht so eine Baumstruktur, in der die Werkzeugobjekte die ihnen von den Kontextwerkzeugen übertragenen Aufgaben selbst übernehmen oder wiederum von Subwerkzeugen erledigen lassen [Rie95, S. 51ff.]. Die resultierende Baumstruktur entspricht dem Entwurfsmuster *Kompositum* (engl. *Composite*) aus [GHJ+95, S. 163ff.]).

Es entsteht so ein Baum, dessen Knoten die Kontextwerkzeuge bilden und dessen Blätter aus einfachen Werkzeugen bestehen, die zur Erfüllung ihre Aufgaben keine weiteren Subwerkzeuge benötigen. Die zu bearbeitenden Materialien erhalten die Subwerkzeuge dabei von ihren Kontextwerkzeugen oder sie ergeben sich aus der Handhabung durch den Benutzer. Da ein Werkzeug dann selbst keine Annahmen über seinen Kontext macht, wird so die Möglichkeit der Wiederverwendung gewährleistet.

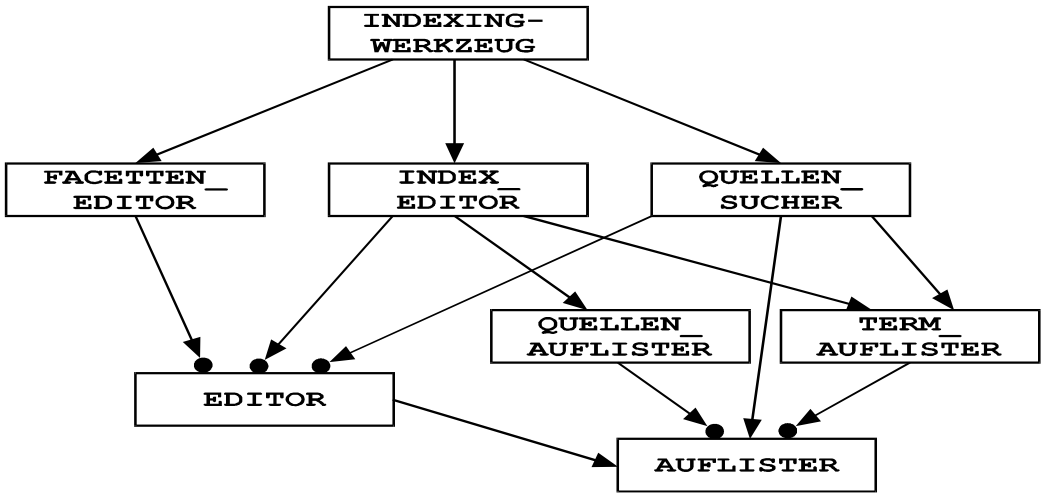


Abb. 5-9: Klassendiagramm für das Indexing-Werkzeug

Für das Indexing-Werkzeug ergibt sich dann ein Klassenmodell, wie es in Abb. 5-9 auf der vorangegangenen Seite dargestellt wird. An oberster Stelle im Werkzeugbaum steht das Indexing-Werkzeug selbst. Es splittet die gestellte Aufgabe in drei Teilaufgaben: Editieren von Facetten, Editieren von Indices und Suchen von Quellen. Die entsprechenden Subwerkzeuge `FACETTEN_EDITOR`, `INDEX_EDITOR` und `QUELLEN_SUCHER` bedienen sich ihrerseits weiterer Subwerkzeuge. Die Ebene der Blätter wird gebildet durch den `AUFLISTER`. Die Abb. 5-10 zeigt den entsprechenden Objektbaum. Hier wird deutlich, wie das Werkzeug `AUFLISTER` in unterschiedlichen Zusammenhängen benutzt werden kann.

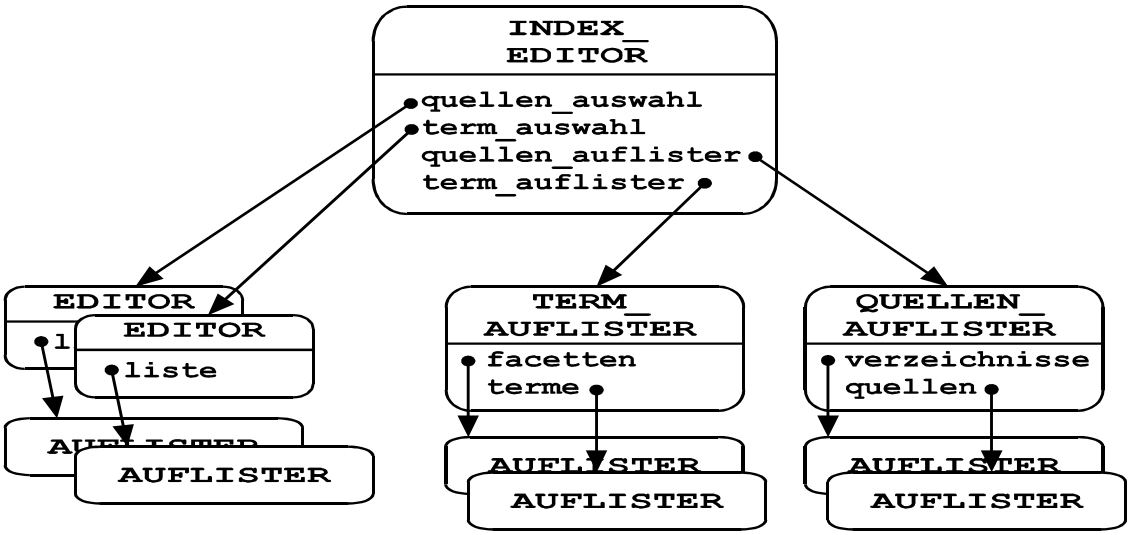


Abb. 5-10: Objektdiagramm für das Indexing-Werkzeug

5.2.6 Trennung von Interaktion und Funktion

Ein Werkzeug ist aufgebaut aus einer Funktionskomponente (FK) und mindestens einer Interaktionskomponente (IAK). Die IAK übernimmt dabei die Aufgabe, das Werkzeug und das Material an der Benutzeroberfläche darzustellen und auf Aktionen des Benutzers zu reagieren. Die FK realisiert die an der Oberfläche angebotene Funktionalität unabhängig von einer konkreten IAK (vgl. [Rie95, S. 53ff.]).

Ein Werkzeug kann also begriffen werden als Summe zweier Teilwerkzeuge. Eines für Interaktion und Präsentation und das andere zur Realisierung von Funktionalität. Bei Darstellung und Reaktion auf Benutzeraktionen muß sichergestellt werden, daß das Werkzeug sich reaktiv und möglichst nicht modal verhält. Die Funktionalität muß sich abstützen auf die durch den Verwendungszusammenhang definierten Aspekte.

Die IAK benutzt zur Darstellung Interaktionstypen (IAT's), die im allgemeinen in den Bibliotheken der Entwicklungsumgebung bereits zur Verfügung stehen (Listen, Knöpfe, Fenster, etc.). Sie reagiert auf Benutzeraktionen, indem sie die Aktionen in Aufrufe von Routinen der FK umsetzt. Die IAK steht also zu ihrer FK in einer Benutztbeziehung. Die FK wiederum benutzt die entsprechenden Aspektklassen, um die Materialien zu bearbeiten. Sie muß ferner Routinen zur Sondierung der Materialien anbieten, damit die IAK in die Lage versetzt wird, auch den Materialzustand darzustellen.

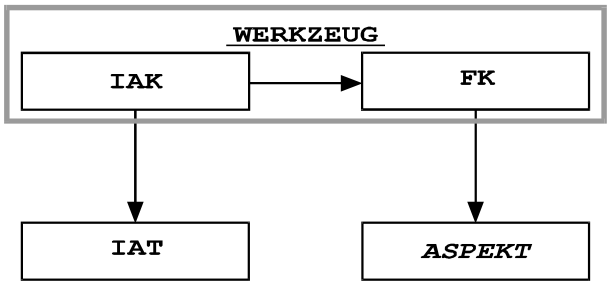


Abb. 5-11: Klassenstruktur eines Werkzeuges

Abb. 5-11 zeigt den allgemeinen Aufbau eines Werkzeuges. Alle Werkzeuge des zu entwerfenden Systems werden nach diesem Schema konstruiert. Die zu benutzenden IAT's entstammen der graphischen Benutzeroberfläche Motif. Die Eiffel-Entwicklungsumgebung kapselt die IAT's in eigenen Klassen wie `PUSH_B`, `TOGGLE_B`, `SCROLLED_T` und `LABEL`. Diese Klassen werden dann von den IAK's benutzt. Dabei stellt die Entwicklungsumgebung auch einen *Ereignismechanismus* zur Verfügung, der die Benutzeraktionen in Aufrufe von Routinen umleitet.

5.2.7 Werkzeugkopplung

Ist ein Werkzeug aus Subwerkzeugen aufgebaut, wird es zum Kontextwerkzeug. Die IAK wird dann zur Kontext-IAK, die die IAK's der Subwerkzeuge benutzt und die FK zur Kontext-FK, welche die FK's der Subwerkzeuge benutzt (vgl. [Rie95, S. 55ff.]).

Nach dem Muster der *Werkzeugkomposition* entsteht durch die Konstruktion eine Baumstruktur, deren Knoten und Blätter die einzelnen Werkzeuge bilden. Diese Vorgehensweise wird auch hier angewandt. Für die IAK's und die FK's wird eine identische Baumstruktur aufgebaut, so als würden sie allein das Werkzeug repräsentieren. Der vorgegebene Anwendungsrahmen [Gin95] legt dabei fest, daß diese Strukturen zur Systeminitialisierungszeit erzeugt werden (genaueres hierzu im Muster *Werkzeugverwaltung*).

Die Abb. 5-12 zeigt den Werkzeugbaum des zu entwerfenden Systems. Im Vorgriff auf den Ereignismechanismus sei hier bereits erwähnt, daß zusätzlich zu den abgebildeten Beziehungen noch Beobachter eine Rolle spielen werden, wobei jede IAK ihre FK und Sub-IAK beobachtet.

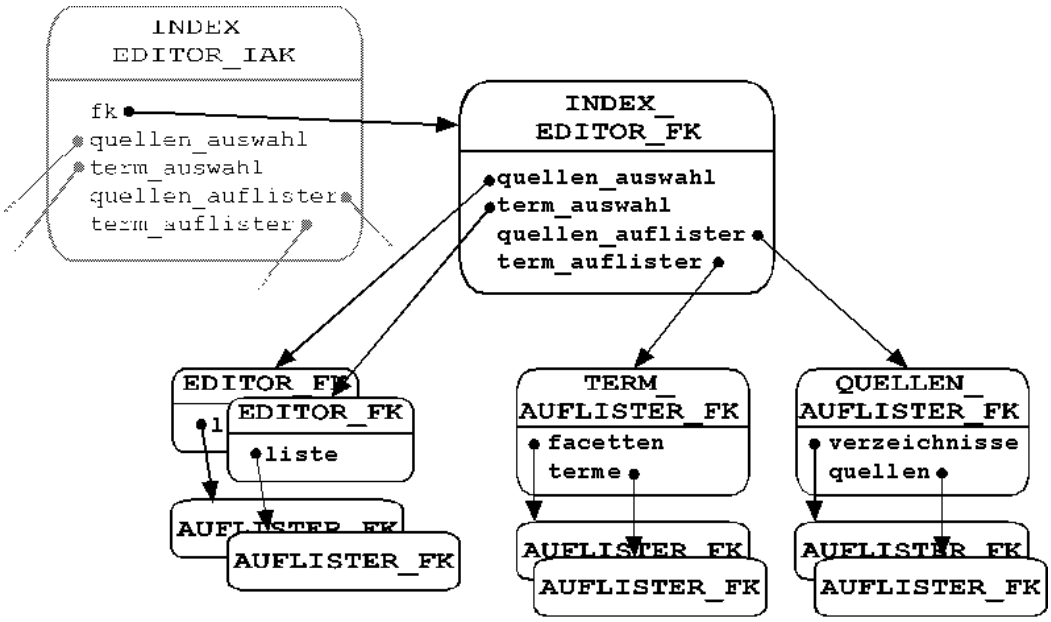


Abb. 5-12: Objektstruktur des Werkzeugbaumes

Das Benutzen der Sub-IAK's durch die IAK's, und somit die Erstellung einer kompletten Baumstruktur, ist keine unbedingte Notwendigkeit. Eine IAK muß mit ihrer Sub-IAK nur dann direkt verbunden sein, wenn Änderungen in der Präsentation der Sub-IAK Auswirkungen auf die Darstellungen der IAK haben. Es wäre mit Hilfe des *Ereignismechanismus* sogar möglich, diese Informationen ohne direkte Verbindung über die Sub-FK und die FK an die IAK zu leiten. Bestehen jedoch solche Abhängigkeiten, ist es in jedem Fall günstiger, die Informationen direkt von Sub-IAK zu IAK fließen zu lassen.

Im Gegensatz zu der hier vorgestellten Lösung, die in engem Zusammenhang mit dem Anwendungsrahmen und der sehr überschaubaren Größe des Softwaresystems steht, wird in [Rie95] ein allgemeines Vorgehen geschildert. Werkzeuge werden dort dynamisch während der Laufzeit des Systems bei Bedarf erzeugt und gelöscht. Dabei steht die FK als Stellvertreter für das Werkzeugs. Eine Kontext-FK trifft dann die Entscheidung über Auf- und Abbau von Sub-Werkzeugen. Die IAK wird nach dem Erstellen der FK erzeugt und beim Löschen vor der FK wieder entfernt. Die zur Wahrung der Konsistenz nötigen Verwaltungsinformationen werden dabei von einem Anwendungsrahmen bereitgestellt.

5.2.8 Umgebungsobjekt

Das *Umgebungsobjekt* ist als Muster für den objektorientierten Entwurf eine direkte Konsequenz aus dem Interpretationsmuster *Umgebung*. Unter Benutzung der *Werkzeugverwaltung* stellt es die Werkzeuge zur Bearbeitung der Materialien zur Verfügung. Die *Materialverwaltung* dient sowohl dazu, den Werkzeugen eine allgemeine Schnittstelle zu den Materialien anzubieten, als auch die Anbindung an externe Dienste, wie Datenbanken oder auch lediglich das Dateisystem, zu gewährleisten [Rie95, S. 62ff.].

Es wird ein Umgebungsobjekt konstruiert, welches diese Aufgaben übernimmt, indem es die Teilaufgaben an weitere Objekte delegiert. Diese Objekte werden beschrieben durch die *Werkzeug-* und die *Materialverwaltung*.

Die Werkzeugverwaltung stellt dem Benutzer die Werkzeuge der Arbeitsumgebung zur Verfügung. Die Materialverwaltung bietet den Werkzeugen eine Schnittstelle zur Bearbeitung von Materialien sowie über die Materialversorger eine Anbindung an das Dateisystem. Die Materialversorger werden vom Umgebungsobjekt erzeugt und der Materialverwaltung bei ihrer Erstellung übergeben. Diesen Zusammenhang zeigt Abb. 5-13 in Form eines Klassendiagrammes.

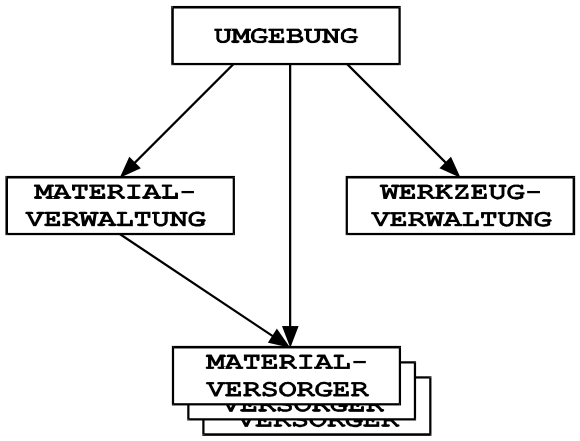


Abb. 5-13: Aufgabenverteilung des Umgebungsobjektes

In [Rie95] wird noch eine weitergehende Aufgabe des Umgebungsobjektes beschrieben: die Realisierung eines Schreibtisches, die an ein Schreibtischobjekt delegiert wird. Der Schreibtisch stellt Werkzeuge und Materialien in Form von Ikonen dar. Das Umgebungsobjekt erfährt dann vom Schreibtischobjekt, welches Objekt auf ein anderes angewendet werden soll und versucht, dies umzusetzen. Ist dies möglich, so wird ein zugehöriges Werkzeug erzeugt und diesem das ausgewählte Material zur Bearbeitung übergeben.

Eine solche Aufgabe wird von dem hier entworfenen *Umgebungsobjekt* aus mehreren Gründen nicht wahrgenommen. Ein Schwerpunkt dieser Arbeit besteht

darin, überhaupt eine Möglichkeit zu schaffen, Werkzeuge nach WAM mit Hilfe der Eiffel-Entwicklungsumgebung zu konstruieren. Der zum jetzigen Zeitpunkt verfügbare Anwendungsrahmen versetzt einen Entwickler jedoch noch nicht in die Lage, die von Riehle beschriebenen Entwurfsmuster in letzter Konsequenz softwaretechnisch umzusetzen. Dies ist einer der Punkte, an dem in Zukunft noch weitere Arbeit zu leisten sein wird.

Des weiteren stellt sich die Frage, ob es bei dem hier vorgestellten Projekt überhaupt nötig ist, ein Schreibtischobjekt einzuführen. Es wird hier keine komplette Arbeitsumgebung abgebildet, sondern lediglich ein kleiner Teil einer solchen. Das Indexing-Werkzeug kann sicherlich in eine Entwicklungsumgebung integriert werden. Diese würde dann den Schreibtisch darstellen, auf dem auch das Indexing-Werkzeug seinen Platz fände.

5.2.9 Werkzeugverwaltung

Die Werkzeugverwaltung erhält vom Umgebungsobjekt die Referenz auf ein Kontext-Werkzeug und initiiert die rekursive Erstellung des FK- und IAK-Baumes. Anschließend übernimmt sie die Aufgabe, jeder IAK die zu benutzende FK zuzuordnen und für die Einrichtung der zugehörigen Beobachterbeziehung zu sorgen.

Wird ein Werkzeug nach den Mustern *Trennung von Interaktion und Funktion* und *Werkzeugkopplung* aufgebaut, so entsteht eine Objektstruktur aus zwei Bäumen (dem IAK- und dem FK-Baum), deren einzelne Komponenten paarweise miteinander verknüpft sind (vgl. Abb. 5-12).

Zum Erzeugen dieser Struktur stehen zwei Wege zur Verfügung. Die eine Alternative besteht darin, zuerst ein einzelnes Werkzeug, bestehend aus IAK und FK zu erzeugen. Die beiden Komponenten werden miteinander verknüpft und dann werden, rekursiv über die IAK absteigend[3], die weiteren IAK/FK-Paare erzeugt. Bei der anderen Alternative werden nacheinander jeweils rekursiv der IAK- und dann der FK-Baum erzeugt. Bei beiden Alternativen bleibt am Ende eine Art von Objektbeziehung ausgespart, die dann nachträglich eingerichtet werden muß. Bei der ersten Methode fehlen die Verknüpfungen zwischen FK und Sub-FK's, bei der zweiten müssen noch jeweils die IAK/FK-Paare miteinander verbunden werden.

Daß die *Werkzeugverwaltung* sich hier der zweiten Alternative bedient, basiert im wesentlichen auf Implementationsentscheidungen, die im Zuge der Erweiterung des Anwendungsrahmens getroffen wurden. Dort werden IAK's und FK's abstrakt als Komponenten eines Werkzeuges implementiert, die unter anderem eine wichtige Eigenschaft gemeinsam haben: rekursiv erzeugbar zu sein. Es erscheint dann als elegante Lösung, dies bei der Erstellung des Werkzeugbaumes auch auszunutzen, anstatt nachträglich noch die FK's verknüpfen zu müssen.

---

[3] Eine Rekursion über die FK's ist hier nicht möglich, da einer FK seine zugehörige IAK nicht bekannt ist.

Das *Umgebungsobjekt* enthält als initiales Werkzeug eine Kontext-IAK und eine Kontext-FK. Sie bilden die Wurzeln für die Objektstrukturen der Werkzeug-Teilbäume. Dieses Werkzeug ist gleichzeitig Ausgangspunkt jeglicher Interaktion eines Benutzers mit dem Softwaresystem.

Die Aufgabe der Erzeugung des Werkzeugbaumes delegiert das *Umgebungsobjekt* an die *Werkzeugverwaltung*. Diese erzeugt die Komponenten des initialen Werkzeuges und setzt damit die rekursive Erstellung eines IAK- und eines FK-Baumes in Gang. Anschließend verknüpft sie paarweise die IAK's mit den FK's. Das Umgebungsobjekt erhält dann vom Werkzeugverwalter die Referenzen auf die initiale IAK und FK und kann sie seiner Kontext-IAK und -FK zuweisen.

Abb. 5-14: Initiales Werkzeug des Systems

Die Abb. 5-14 zeigt die Benutzungsoberfläche dieses Kontext-Werkzeuges. Es dient nicht zur Bearbeitung von Materialien, sondern seine einzige Aufgabe besteht darin, über Knöpfe die einzelnen Werkzeuge zur Verfügung zu stellen. Damit ist es gewissermaßen auch der Ersatz für das *Schreibtischobjekt*, das Riehle beschreibt. Die drei Subwerkzeuge – also die eigentlichen Werkzeuge des Systems – werden über die entsprechenden Knöpfe ausgewählt.

5.2.10 Materialverwaltung

Die Materialverwaltung bildet ein Subsystem, welches eine Schnittstelle zur Verfügung stellt, über die Werkzeuge auf Materialien zugreifen können. Aus der Sicht der Werkzeuge können dabei alle Materialien gleichrangig behandelt werden, unabhängig davon, welcher Quelle sie entstammen, ob sie überhaupt persistente oder lediglich transiente Materialien sind [Rie95, S. 64ff.].

Die hier verwendete Materialverwaltung basiert auf der Arbeit von Martina Wulf und Dirk Weske [WW94], die sich intensiv mit der Thematik der Materialverwaltung im Rahmen der Werkzeug und Material Metapher auseinandergesetzt haben.

Im allgemeinen werden Materialien nicht einmalig an einem Arbeitsplatz bearbeitet, sondern es wird mehrfach und von verschiedenen Stellen aus auf sie zugegriffen. Materialien werden dann nicht an einem einzelnen Arbeitsplatz gespeichert, sondern etwa in Datenbanken, deren Dienste von allen Systemen genutzt werden.

Das *Umgebungsobjekt* eines Softwaresystems muß demnach eine Schnittstelle zur Verfügung stellen, die den Zugriff auf solche externen Dienste gestattet. Diese Schnittstelle sollte so einheitlich wie möglich gestaltet werden, sodaß den Werkzeugen verborgen bleibt, ob die Materialien letztendlich z.B. einer objektorientierten oder einer relationalen Datenbank entnommen wurden. Günstig ist hierbei eine Lösung, die auch transiente Materialien, die nur innerhalb einer Umgebung bearbeitet werden, mit einschließt. Die Werkzeuge können dann eine Schnittstelle für alle Materialien benutzen.

Zum Zeitpunkt der Systeminitialisierung erstellt das Umgebungsobjekt ein Subsystem, die *Materialverwaltung*, und übergibt ihr für jeden externen Dienst einen Materialversorger. Das konkrete Wissen über die Außenwelt verbleibt somit beim Umgebungsobjekt und damit vor der Materialverwaltung verborgen. Als einheitliche Schnittstelle dieses Subsystems dient den Werkzeugen der `MATERIAL_VERWALTER`.

Die Abb. 5-15 zeigt das Klassendiagramm der Materialverwaltung. Eine besondere Bedeutung kommt dabei dem Aspekt *`VERWALTBAR`* zu. Er muß Teil der

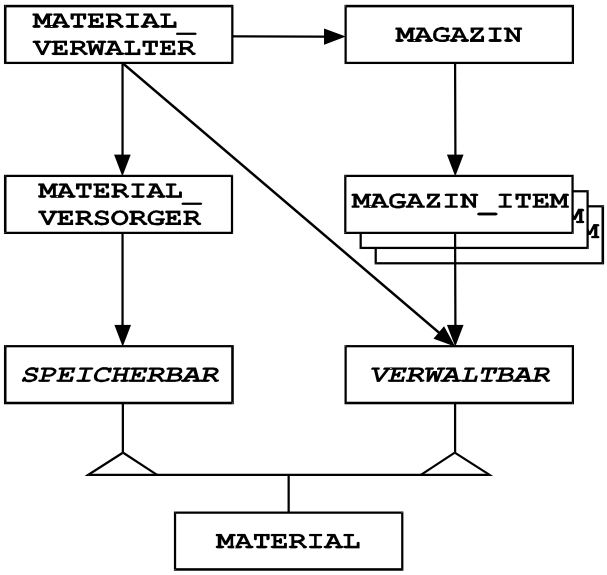


Abb. 5-15: Die Materialverwaltung des Systems

Schnittstelle eines jeden Materials sein, damit die Materialverwaltung in der Lage ist, ein Material überhaupt zu verwalten. Er ist aber weiterhin auch die Schnittstelle für den Transport von Materialien zwischen Werkzeugen und Materialverwaltung.

Der Aspekt *VERWALTBAR* beschreibt damit eine Funktionalität, der ein Material in allen Verwendungszusammenhängen genügen muß. Dies wird softwaretechnisch durch die Beziehungen ausgedrückt, die in dem Klassendiagramm in Abb.

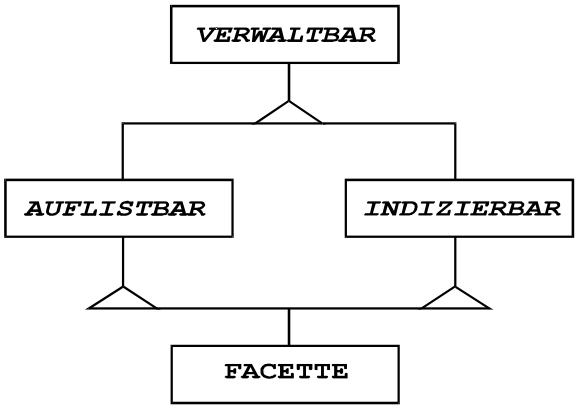


Abb. 5-16: Das verwaltbare Material *VERZ_LISTE* mit seinen Aspekten

5-16 dargestellt sind. Jeder Aspekt eines Materials wird als Unterklasse der Aspektklasse *VERWALTBAR* beschrieben[4]. Damit wird sichergestellt, das jedes Material in jedem Verwendungszusammenhang auch von der Materialverwaltung ‘erkannt’ wird.

5.2.11 Materialcontainer

Bei ihrer Erzeugung initialisiert die *Materialverwaltung* einen *Materialcontainer* – das Verwalter-`MAGAZIN` – mit einer Vielzahl von `MAGAZIN_ITEMS`. Jedes Item enthält die Spezifikation eines Material-Objektes. In ihrer Gesamtheit liefern sie eine vollständige Beschreibung aller existierenden Materialien.

Diese Mikroarchitektur basiert auf den Materialcontainern, die im Rahmen des Entwurfs einer Materialverwaltung in [WW94] eingeführt werden. Dort werden die Container auch genutzt, um Materialsammlungen als Ergebnisse von Anfragen an den Material-Verwalter herauszugeben. Antworten auf Anfragen, die innerhalb des Indexing-Werkzeuges entstehen, können jedoch immer nur genau

[4] Dieser Aussage scheint die Darstellung in Abb. 5-15 zu widersprechen, welche die Aspekte *SPEICHERBAR* und *VERWALTBAR* als unabhängig voneinander zeigt. Im Diagramm wird jedoch lediglich aus Gründen der Übersichtlichkeit die Tatsache ‘unterschlagen’, daß auch *SPEICHERBAR* von *VERWALTBAR* erbt.

ein Exemplar eines Materials enthalten. Der *Materialcontainer* wird deshalb nur als Verwalter-Magazin implementiert, und Anfragen liefern immer dieses Exemplar direkt.

Im allgemeinen wird sich zu einem beliebigen Zeitpunkt nur ein Teil aller verfügbaren Materialien in Bearbeitung befinden. Diese sind der Materialverwaltung natürlich bekannt. Sie muß aber auch Kenntnis von der Existenz von Materialien haben, die gerade nicht aktiviert sind, da sonst die Anfrage nach einem solchen Materialexemplar nur mit dem Erzeugen und Einfügen eines neuen Exemplars beantwortet werden könnte. Materialien müssen also auf eine Art und Weise beschrieben und aufbewahrt werden, die es dem Materialverwalter ermöglicht, sinnvoll auf alle Anfragen reagieren zu können.

Materialien werden eindeutig identifiziert anhand von Spezifikationen. Jede `SPEZIFIKATION` enthält Name und Typ eines ganz bestimmten Materialexemplares. Die *Materialverwaltung* benutzt dann einen *Materialcontainer*, das `MAGAZIN`, welches eine Liste von `MAGAZIN_ITEM`s enthält (vergl. Abb. 5-15). Eine der Komponenten jedes Items ist eine `SPEZIFIKATION`.

Der `MATERIAL_VERWALTER` initialisiert dann bei seiner Erzeugung das `MAGAZIN` mit einem `MAGAZIN_ITEM` für jedes verfügbare Materialexemplar. Die Spezifikationen dieser Exemplare erhält der Verwalter dabei von den ihm bekannten `MATERIAL_VERSORGERN`.

Zur Beantwortung von Werkzeug-Anfragen kann über die Items des Magazins iteriert werden. Zu aktivierende Materialien werden in ‘ihr’ zugehöriges Item eingebettet, und können den Werkzeugen von dort aus verfügbar gemacht werden.

Obwohl gleichen Namens hat der hier beschriebene Entwurf eines Materialcontainers wenig gemein mit dem in [Rie95, S. 66ff.] beschriebenen *Materialcontainer*. Dort wird der Materialcontainer eingeführt, um die Konsistenz voneinander abhängiger Materialien zu gewährleisten. Dies geschieht durch die weitere Einführung von Konsistenzbedingungsojekten und *Werkzeugkoordinatoren*. Da in dem zu entwerfenden Softwaresystem nur eine kleine Anzahl an Materialien benutzt wird und deren Abhängigkeiten voneinander nur von geringer Bedeutung sind, wurde auf die Umsetzung dieses Konzeptes verzichtet.

5.2.12 Ereignismechanismus

Eine IAK ist vom Zustand ihrer FK und Sub-IAK’s abhängig. Zusätzlich zu der Benutztbeziehung wird deshalb noch die Eigenschaft des Beobachters eingeführt. Die IAK beobachtet dabei ihre FK und Sub-IAK’s und wird von ihnen über Zustandsänderungen informiert. Das gleiche gilt für das Umgebungsobjekt als Beobachter seines Kontext-Werkzeuges und für die FK’s, die den Materialverwalter beobachten. Das Muster *Ereignismechanismus* wird in [Rie95, S. 57ff.] beschrieben. Das Konzept des Beobachters im Rahmen der Materialverwaltung ist angelehnt an [WW94].

In der Darstellung des Werkzeugzustandes und der Materialien ist eine IAK abhängig vom Zustand ihrer FK und ihrer Sub-IAK's. Nach den Mustern der *Werkzeugkopplung* und der *Trennung von Interaktion und Funktion* bleibt die IAK jedoch anonym für ihre FK und Sub-IAK's. Es besteht somit keine Möglichkeit, Zustandsänderungen direkt an die IAK zu leiten.

Auch der Umgebung sind zwar die Werkzeuge bekannt, aber die simple Tatsache, daß ein Benutzer den Ende-Knopf angeklickt hat, um die Ausführung des Systems zu beenden, bleibt vor ihr verborgen. Erst ein Mechanismus, der dieses Ereignis zurückliefern kann, könnte das Dilemma beheben.

Ein ähnliches Problem besteht bei der Benutzung des Materialverwalters durch eine FK. Ihr ist durch das Umgebungsobjekt der Verwalter bekannt, aber Änderungen im Materialbestand, die durch andere FK's herbeigeführt werden, können sie nicht erreichen.

Zur Lösung der Problematik wird das Konzept des Beobachters in den Entwurf eingeführt. Als *BEOBACHTER* beobachtet ein Objekt A ein anderes Objekt B, das BEOBACHTBAR ist. Dazu läßt A sich bei B unter seiner Objektreferenz registrieren. Dies versetzt B in die Lage, Ereignisse zu Zustandsänderungen an A zu melden.

Das Klassendiagramm in Abb. 5-17 zeigt die Stellen, an denen Beobachter-Beziehungen realisiert werden. Der Anwendungsrahmen, der in [Gin95] beschrieben wird, stellt dabei bereits ein einfaches Konzept des Beobachters zur Verfügung. Dort können sich beliebige Objekte bei einem beobachtbaren Objekt registrieren lassen, die dann im Falle einer Zustandsänderung alle benachrichtigt wer-

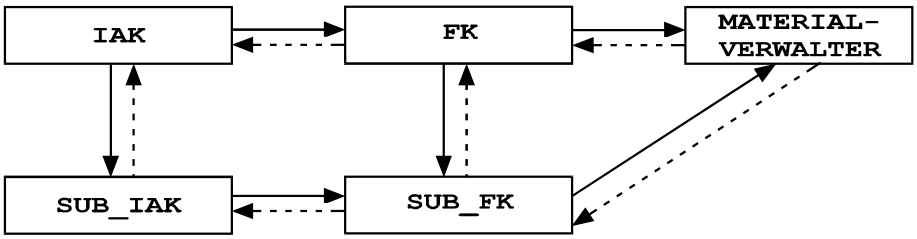


Abb. 5-17: Die Beobachter-Beziehungen des Systems

den. Dabei werden unabhängig von der Art der Änderung immer alle Beobachter von dieser Änderung informiert. Da eine FK immer nur zwei Beobachter (ihre IAK und ihre Kontext-FK) und eine IAK immer nur einen (ihre Kontext-IAK) hat, ist dieses Vorgehen für die Kommunikation der Werkzeugkomponenten untereinander (inklusive des Umgebungsobjektes) akzeptabel.

Problematisch ist dies jedoch für die FK's und den Materialverwalter. Die Beobachter-Beziehung ist in diesem Fall eine 1-n-Beziehung mit einem beobachtbarem Objekt und n FK's als Beobachtern. Der Anwendungsrahmen wird deshalb

erweitert um *MATERIAL_BEOBACHTER* und MATERIAL_BEOBACHTBARe Objekte. Ein *MATERIAL_BEOBACHTER* trägt sich dabei nicht nur unter seiner Objektreferenz ein, sondern spezifiziert auch das beobachtete Materialexemplar und die Art der Zustandsänderung, von der er benachrichtigt werden möchte. Wird also zum Beispiel ein bestimmtes Materialexemplar verändert, so werden nur diejenigen FK's davon benachrichtigt, die genau diese Änderung dieses einen Materialexemplares beobachten.

Diese beiden Ereignismechanismen werden über Vererbung realisiert und ähneln einander stark. Wie in Abb. 5-18 dargestellt, gilt für die Werkzeugkomponenten: jede beobachtende Klasse erbt von *BEOBACHTBAR* und jede beobachtbare von BEOBACHTBAR. Analog gilt gemäß Abb. 5-19, daß der Materialverwalter von MATERIAL_BEOBACHTBAR und alle FK's von *MATERIAL_BEOBACHTER* erben.

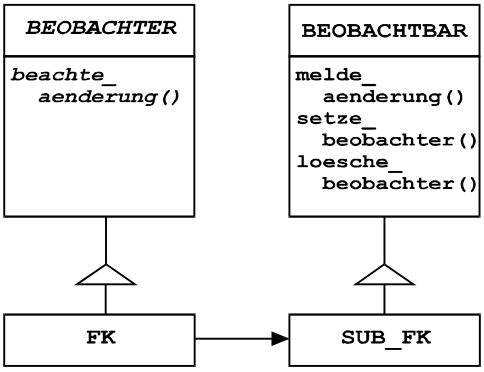


Abb. 5-18: Ereignismechanismus für die Werkzeugkomponenten

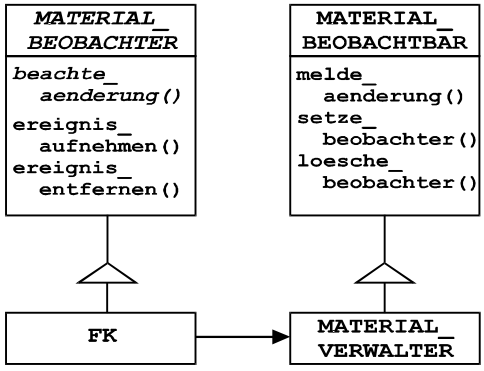


Abb. 5-19: Ereignismechanismus für den Materialverwalter

Die Nutzung dieses Ereignismechanismus ist nicht ganz unproblematisch, da es passieren kann, daß ein gemeldetes Ereignis zu Änderungen führt, die wiederum zu Ereignissen führen, die Änderungen zur Folge haben ... . [Rie95, S. 59] formuliert deshalb zwei Richtlinien, die unbedingt beim Entwurf von Beobachtern und Beobachteten eingehalten werden sollten:

„**Richtlinie zum Entwurf von Beobachtern:**

Entwerfe einen Beobachter immer so, daß er im Falle eines Ereignisses nie verändernd auf den Urheber reagiert, sondern immer nur sondierend.

Richtlinie zum Entwurf von beobachtbaren Objekten (Seiteneffektfreiheit):

> Entwerfe die Klassenschnittstelle und implementiere die Klasse so, daß klar zwischen verändernden und sondierenden Operationen unterschieden werden kann.“

Aufgrund der oben beschriebenen Problematik, und weil die Komplexität des Softwaresystems durch den Einsatz von Beobachtern zunimmt, wird empfohlen [Rie95], Beobachter nur im Zusammenhang mit Werkzeugkomponenten einzusetzen. Dennoch wird hier der auch in [WW94] eingesetzte Ereignismechanismus benutzt, da das Softwaresystem wegen seiner leicht überschaubaren Größe nicht merklich davon beeinträchtigt wird. Im besonderen hilft auch die strikte Trennung von Werkzeug- und Materialbeobachtern, eine übermäßige Komplexität zu vermeiden.

6 Zusammenfassung und Ausblick

Auf der Grundlage der Arbeiten von Prieto-Diaz und insbesondere von Sørumgård und Tryggeseth läßt sich ein Meta-Modell entwickeln, welches durch Instanziierung in ein Facetten-Modell ein universelles Werkzeug zur Klassifizierung von Softwarekomponenten liefert. Die Erstellung eines solchen Klassifikationsschemas für den Entwicklungsprozeß nach WAM zeigt, daß sich die Quelltexte, die im Rahmen des Prozesses bearbeitet und verwaltet werden müssen, sinnvoll klassifizieren lassen.

Der Entwurf und die Implementation des Prototypen demonstrieren, daß die Klassifizierung und Suche der Komponenten werkzeugunterstützt erfolgen kann. Es wird damit möglich, die Integration der Quelltexte in den Entwicklungsprozeß wesentlich zu verbessern.

Diese Arbeit beschäftigt sich schwerpunktmäßig mit der Klassifikation von Klassenbeschreibungen. Die anderen zentralen Dokumenttypen des WAM-Prozesses spielen nur eine untergeordnete Rolle. Der nächste logische Schritt wäre daher die Erweiterung des Klassifikationsschemas, um auch Visionen, Szenarios, Glossare, etc. mit aufzunehmen. Eine weitere wichtige Aufgabe besteht darin, die gewichteten Relationen, wie sie von Sørumgård und Tryggeseth eingeführt worden sind, in das Schema zu integrieren und als Teil des Werkzeuges zu implementieren.

Um wirklich effektiv mit dem Werkzeug arbeiten zu können, und damit wichtige Erfahrungen für die weitere Entwicklung gewonnen werden können, sollte eine Anbindung sowohl an eine Datenbank als auch an eine bestehende Entwicklungsumgebung ins Auge gefaßt werden.

Am Beginn der Erstellung dieser Diplomarbeit erschien fast unmöglich, das zugehörige Softwarewerkzeug mittels Eiffel zu konstruieren *und* auch zu implementieren. Insbesondere die Entwicklungsumgebung kam – nicht zuletzt aufgrund mangelnder Dokumentation – für eine Implementation nicht in Frage. Die Weiterentwicklung machte jedoch so rasche Fortschritte, daß der – schließlich erfolgreiche – Versuch dann doch unternommen wurde.

Die Erfahrungen, die im Verlauf dieser Arbeit mit Eiffel gemacht wurden, geben berechtigten Anlaß zu der Hoffnung, daß Eiffel, auch über den Einsatz als Entwurfssprache hinaus, zur Implementation weitere Verbreitung findet.

Anhang

Dieser Anhang liefert in zwei Abschnitten weitere Informationen zu dem Klassifizierungs-Werkzeug, insbesondere seine Handhabung und Installation, sowie zu einigen weiterführenden Aspekten im Rahmen von Klassifikation und Wiederverwendbarkeit, die in dieser Arbeit nur sehr am Rande erwähnt werden.

I. Handhabung des Werkzeuges

Teil dieser Diplomarbeit ist die Implementation eines Klassifizierungs-Werkzeuges. Der Quellcode des Werkzeuges liegt dieser Arbeit bei. Im folgenden wird die Installation beschrieben und ein kurzer, handbuchartiger Einblick in die Handhabung gegeben.

I.1 Installation

Voraussetzungen für die Installation sind: ISE Eiffel in der Version 3.3.7 oder höher und X-Windows mit Motif als Fenstersystem.

Die Installation erfolgt in folgenden Schritten:

1.) Kopieren des komprimierten tar-files von der Diskette in ein Zielverzeichnis.

2.) Entpacken des tar-files.

3.) Bevor die Quellen compiliert werden können, sind noch Änderungen an der Systemdatei nötig. Es stehen zwei Dateien zur Auswahl, die an die Systemumgebung angepaßt werden können: `Ace.ace` und `Ace.finalize`. Die beiden Dateien unterscheiden sich lediglich in der Art des zu erzeugenden Codes: `Ace.finalize` bietet ein Optimum von Laufzeitverhalten und Code bei einem Maximum an Compilezeit, während `Ace.ace` auf eine möglichst geringe Compilezeit ausgerichtet ist. Im folgenden ist der Inhalt von `Ace.finalize` wiedergegeben:

```
system xeindex

root

   umgebung (root_cluster): "make"

default

   fail_on_rescue (yes);
   array_optimization (yes);
   assertion (no);
   dead_code_removal (yes);
```

```
    -- precompiled ("$EIFFEL3/precomp/spec/$PLATFORM/mvision");

cluster

    root_cluster:       "/diplom/arbeit";

    werkzeuge:              "/diplom/arbeit/werkzeuge";
    werkzeug_verw:      "/diplom/arbeit/werkzeuge/verwaltung";
    parser:                 "/diplom/arbeit/werkzeuge/parser";
    materialien:   "/diplom/arbeit/materialien";
    material_verw:      "/diplom/arbeit/materialien/verwaltung";

    eiffel:          "/diplom/arbeit/eiffel";
    framework:              "/diplom/arbeit/framework";

-- STANDARD STRUCTURES:

    kernel:          "$EIFFEL3/library/base/kernel";
    support:          "$EIFFEL3/library/base/support";
    access:          "$EIFFEL3/library/base/structures/access";
    cursors:          "$EIFFEL3/library/base/structures/cursors";
    cursor_tree:       "$EIFFEL3/library/base/structures/cursor_tree";
    dispenser:        "$EIFFEL3/library/base/structures/dispenser";
    iteration:       "$EIFFEL3/library/base/structures/iteration";
    list:           "$EIFFEL3/library/base/structures/list";
    obsolete:         "$EIFFEL3/library/base/structures/obsolete";
    set:            "$EIFFEL3/library/base/structures/set";
    sort:           "$EIFFEL3/library/base/structures/sort";
    storage:          "$EIFFEL3/library/base/structures/storage";
    table:           "$EIFFEL3/library/base/structures/table";
    traversing:       "$EIFFEL3/library/base/structures/traversing";
    tree:           "$EIFFEL3/library/base/structures/tree";

-- EiffelVision

     graph_resources:   "$EIFFEL3/library/vision/implement/motif/Resources";
     graph_widgets:     "$EIFFEL3/library/vision/implement/motif/widgets";
     graph_toolkit:     "$EIFFEL3/library/vision/implement/toolkit";
     graph_X:          "$EIFFEL3/library/vision/implement/X";
     graph_commands:    "$EIFFEL3/library/vision/oui/commands";
     graph_kernel:      "$EIFFEL3/library/vision/oui/kernel";
     graph_oui_widgets: "$EIFFEL3/library/vision/oui/widgets";
     graph_figures:     "$EIFFEL3/library/vision/figures";
     graph_tools: "$EIFFEL3/library/vision/tools";

-- EiffelLex+Parse

     lex:          "$EIFFEL3/library/lex";
     parse:                "$EIFFEL3/library/parse";

external

    object:         "$(EIFFEL3)/library/lex/spec/$(PLATFORM)/lib/lex.a";
                    "$(EIFFEL3)/library/vision/spec/$(PLATFORM)/lib/Xt.a";
                    "$(EIFFEL3)/library/vision/spec/$(PLATFORM)/lib/motif_Clib.a";
                    "/usr/X11R6/lib/libMrm.a";
                    "/usr/X11R6/lib/libXm.a";
                    "/usr/X11R6/lib/libXpm.a";
                    "/usr/X11R6/lib/libXmu.a";
                    "/usr/X11R6/lib/libXt.a";
```

```
        "/usr/X11R6/lib/libXext.a";
        "/usr/X11R6/lib/libX11.a";
        "/usr/X11R6/lib/libSM.a";
        "/usr/X11R6/lib/libICE.a"

end
```

Anzupassen sind hier die Verzeichniseinträge, die mit /diplom/arbeit beginnen. Stattdessen ist jeweils der Name der Zielverzeichnisses, in dem sich die entsprechenden Quellen befinden, einzugeben. Des weiteren sind die externals am Ende den Systemgegebenheiten anzupassen.

4.) Nun kann das System compiliert werden. Dies kann sowohl innerhalb der Entwicklungsumgebung als auch in einer shell geschehen (Details zu Kommandozeilenparametern für die Umwandlung können in [ISE95] nachgelesen werden). Das Compilat erhält den Name `xeindex` und findet sich im Verzeichnis *`Installationsverzeichnis`*`/EIFGEN/F_code`. Diese ausführbare Datei kann in ein beliebiges Verzeichnis kopiert werden.

5.) Abschließend sind noch zwei Umgebungsvariablen zu setzen, `$XEI_INST` und `$XEI_DATEN`. `$XEI_INST` bezeichnet jenes Verzeichnis, in dem sich `xeindex` befindet. `$XEI_DATEN` gibt das Datenverzeichnis an. Es kann später beim Aufruf des Programms über den Kommandozeilenparameter -dv auch ein anderes bestimmt werden.

Nach erfolgreicher Installation kann das System über den Befehl `xeindex` aufgerufen werden.

I.2 Handbuch

Der Aufruf des Klassifizierungs-Werkzeuges erfolgt über den Befehl `xeindex`. Nach dem Start erscheint ein Fenster ähnlich dem aus Abb. I-1. Über die Knöpfe kann zu den Subwerkzeugen verzweigt oder die Arbeit mit dem Werkzeug beendet werden.

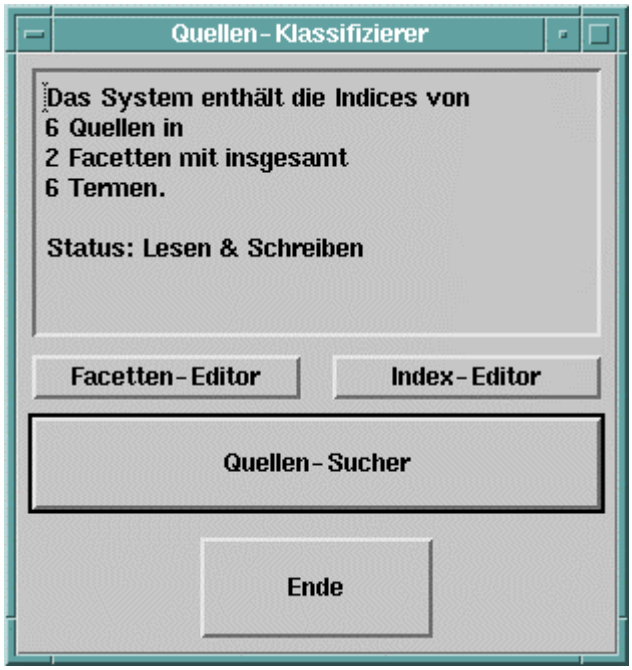


Abb. I-1: Startfenster für den ersten Benutzer

Im oberen Teil des Fensters liefert dieses Kontext-Werkzeug ständig die aktuellen Informationen über die Anzahl der klassifizierten Quellen, Facetten und Terme.

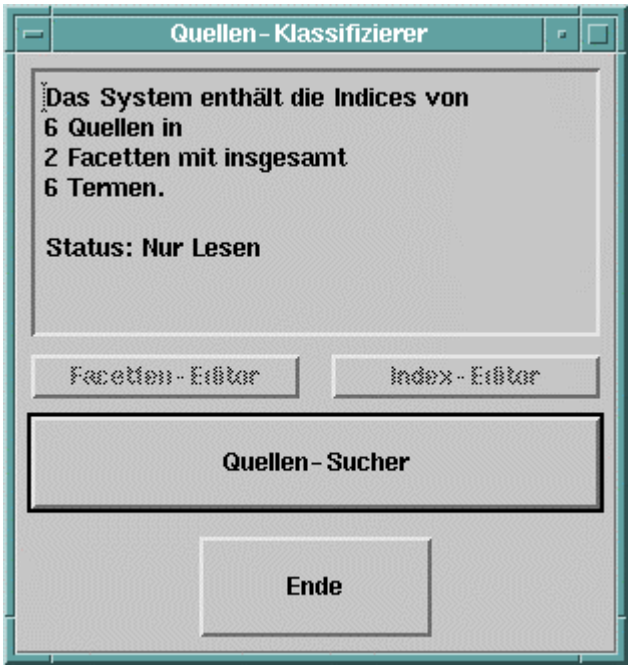


Abb. I-2: Startfenster für folgende Benutzer

Der Status dient zur deutlichen Unterscheidung zwischen den Rechten, die mehrere Benutzer desselben Datenbestandes haben. Nur der erste Benutzer hat Schreib-Rechte. Beendet dieser die Arbeit mit dem Werkzeug, so gehen die Schreib-Rechte auf den nächsten Nutzer über, der das Werkzeug aufruft.

Alle Benutzer, die lediglich lesend auf den Datenbestand zugreifen können, erhalten nach dem Start ein Kontext-Werkzeug wie in Abb. I-2.

Facetten-Editor

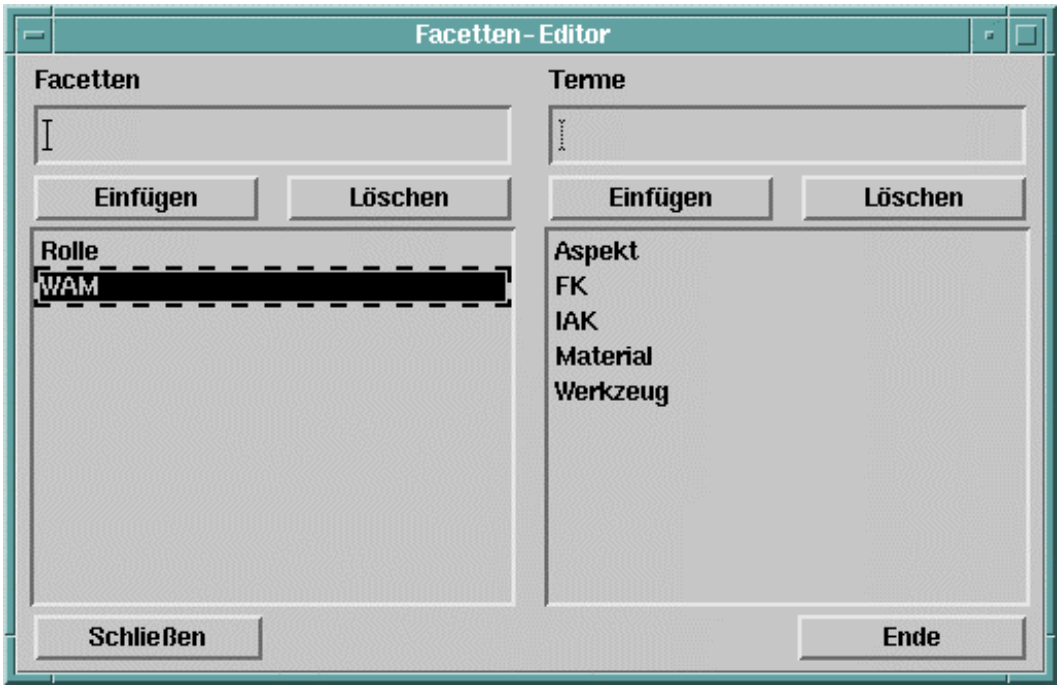


Abb. I-3: Der Facetten-Editor

Der Facetten-Editor erlaubt die Festlegung des Klassifikationsschemas, dient also zur Instanziierung des Modells (siehe Abb. I-3). Auf der linken Seite können die Facetten eingefügt und gelöscht werden, auf der rechten die zugehörigen Terme.

Wie auch bei den folgenden Werkzeugen sind die Knöpfe `Schließen` und `Ende` enthalten. `Schließen` beendet lediglich die Arbeit mit dem Subwerkzeug, während `Ende` das Klassifizierungs-Werkzeug insgesamt abschließt.

Index-Editor

Dieses Sub-Werkzeug (siehe Abb. I-4) dient der Klassifizierung von Eiffel-Quelltexten nach dem im Facetten-Editor festgelegten Schema.

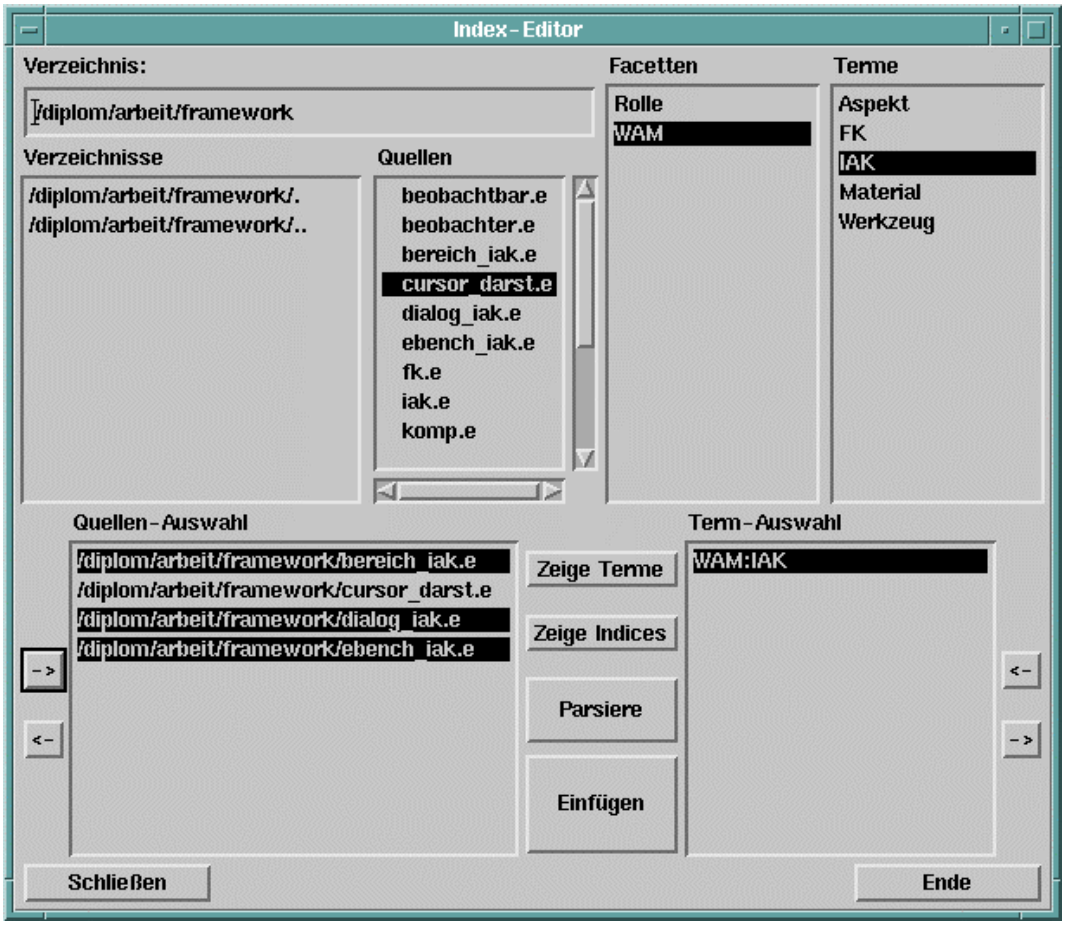


Abb. I-4: Der Index-Editor

Im oberen linken Teil des Fensters werden die zu klassifizierenden Quelltexte ausgewählt. Rechts davon werden die entsprechenden Indices, bestehend aus einer Facette und einem oder mehreren Termen, gewählt. Diese Auswahl wird über die Knöpfe `->` und `<-` in die korrespondierenden Auswahlfenster im unteren Teil übernommen.

Der Knopf `Einfügen` fügt für alle ausgewählten Quellen, die ausgewählten Facetten/Term-Paare ein. `Parsiere` fügt die im Indexing-Teil der Quellen angegebenen Indices der Datenbank hinzu. `Zeige Indices` parsiert lediglich die Quellen und zeigt die gefundenen Einträge an. `Zeige Terme` zeigt die im Datenbestand hinterlegten Indices der ausgewählten Quellen an und ermöglicht anschließend ein gezieltes Löschen der gefundenen Einträge.

Quellen-Sucher

Der Quellen-Sucher (Abb. I-5) ermöglicht das Auffinden von Quellen anhand der im Datenbestand hinterlegten Indices.

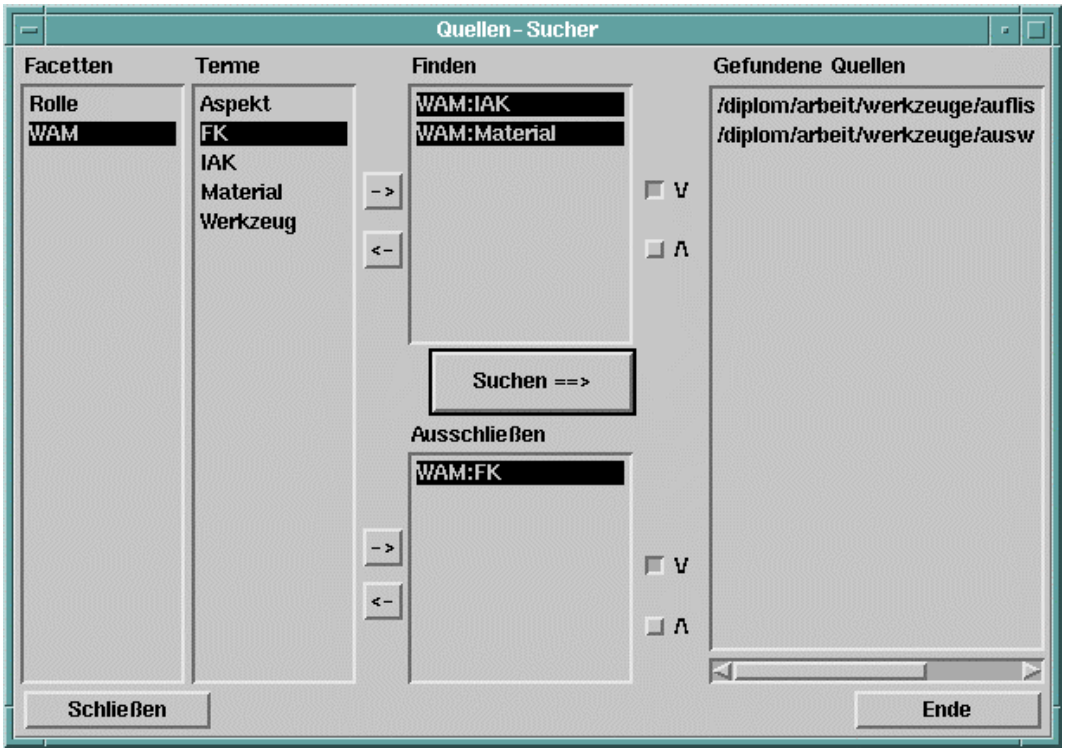


Abb. I-5: Der Quellen-Sucher

Links sind die Facetten und Terme des Klassifikationsschemas aufgelistet. Die hier vorgenommenen Auswahlen können über `->` in die Auswahllisten `Finden` und `Ausschließen` übernommen werden. Um eine Suche durchzuführen, werden in diesen Listen dann Indices ausgewählt und jeweils mit `\/` oder `/\` als Und- oder Oder-Verknüpfungen bestimmt. Die Suche wird über `Suche ==>` gestartet.

Über die Knöpfe `<-` können markierte Einträge aus den Auswahllisten wieder entfernt werden.

Kommandozeilen-Parameter

Beim Aufruf des Klassifizierungs-Werkzeuges können wahlweise drei verschiedene Parameter übergeben werden:

- -dv *Datenverzeichnis*: Gibt das Verzeichnis des zu benutzenden Datenbestandes an. Hiermit kann für die aktuelle Sitzung ein von `$XEI_DATEN` abweichendes Datenverzeichnis bestimmt werden. Ohne Angabe dieses Parameters wird das Datenverzeichnis aus `$XEI_DATEN` genutzt. Ist es nicht vorhanden, so wird es angelegt.

- - ea *Analyzer-Verzeichnis*: Beim erstmaligen Aufruf des Werkzeuges wird ein Eiffel-analyzer erzeugt, der zum Parsieren der Quelltexte benötigt wird. Ohne diesen Parameter wird er im Datenverzeichnis gesucht. Wird er nicht gefunden, so wird er neu erzeugt.

- -nm *Name*: Ein Name für die aktuelle Instanz des Werkzeuges. Dies ist nur sinnvoll, wenn `xeindex` mehrfach vom selben Benutzer aufgerufen wird. Der angegebene Name wird dann zur besseren Unterscheidung der Werkzeug-Fenster im Titel mit angezeigt.

II. Weiterführende Aspekte

Das Klassifizieren von Softwarekomponenten ist immer nur eine Teilaufgabe im Rahmen eines Prozesses. Es wird nicht nur um des Klassifizierens willen durchgeführt, sondern dient als Voraussetzung zur Erfüllung weiterer Aufgaben. In dieser Arbeit zum Beispiel primär zur Verbesserung der Integration der Komponenten in den Entwicklungsprozeß.

Es gibt eine ganze Reihe weiterer Aspekte, die thematisch an die hier geführte Diskussion anknüpfen und zur Zeit Gegenstand intensiver Forschungsbemühungen sind. Es soll deshalb hier nicht darauf verzichtet werden, eine kurze Zusammenstellung weiterführender Aspekte, sowie eine Reihe zugehöriger Literaturverweise[5] vorzustellen.

II.1 Wiederverwendbarkeit

Das Facetten-Modell ist ursprünglich für die Softwaretechnik weiterentwickelt worden, um den Zugriff auf große Mengen wiederverwendbarer Softwarekomponenten zu erleichtern. Die Komponenten, die dabei zur Wiederverwendung in Betracht gezogen werden, sind Klassen oder Quelltexte. Es stellt sich jedoch die Frage, ob nicht auch andere Arten von Wiederverwendbarkeit untersucht werden sollten. Für den WAM-Prozeß heißt dies, zu untersuchen, inwieweit andere Dokumente (Szenarios, Visionen, ...) zu klassifizieren und dann für spätere Projekte wieder zu verwenden sind.

Wichtig ist es in jedem Fall, Bewertungsmaßstäbe zu finden, die eine Aussage darüber gestatten, unter welchen Voraussetzungen Komponenten wiederverwendbar sind. Dazu gehört nicht zuletzt auch eine betriebswirtschaftliche Kosten/Nutzen Analyse. Es sollte in jedem Fall sichergestellt werden können, daß die Änderung einer vorhandenen Komponente nicht aufwendiger ist, als eine komplett neue zu erzeugen.

[5] An dieser Stelle sei ein Hinweis zu ‘Literatur’verweisen angebracht: Aufgrund ihrer Aktualität sind einige Verweise nicht auf gedruckt publizierte Werke zu machen. Solche Artikel haben – zumindest nach Wissen des Autors – den Weg in die Printmedien noch nicht gefunden (und werden ihn vielleicht auch nie finden). Anstelle klassischer Literaturverweise stehen dann Verweise auf bestimmte Seiten im WWW oder ftp-server. Es bleibt also lediglich zu hoffen, daß der Leser die entsprechenden Links noch vorfindet oder ansonsten über eine Suchmaschine erfolgreich ist.

Eine brauchbare Zusammenstellung von Modellen und Verfahren zur Wiederverwendbarkeit bieten William Frakes und Carol Terry in [FT96].

II.2 Domänen Analyse

Für Softwareprojekte, die sich im Rahmen einer speziellen Anwendungsdomäne bewegen, fallen ad hoc zwei Punkte auf, die einer genaueren Untersuchung unterzogen werden sollten. Zum einen könnte es sinnvoll sein, eine Reihe domänenspezifischer Facetten einzuführen, die präzise den Bereich des Anwendungsgebietes abdecken. Zum andern wäre zu überlegen, inwieweit Glossare als Sammlungen von Termen für diese Facetten dienen könnten.

Die Domänen Analyse ist in jedem Fall ein wichtiger Aufgabenbereich, wenn es um Fragen der Wiederverwendbarkeit geht. Eine ganze Reihe Artikel zu diesem Thema finden sich zum Beispiel in den Proceedings des letzten Workshops on Institutionalizing Software Reuse, der Ende August 1995 in den USA stattfand [WISR7].

II.3 Metriken

Wenn es darum geht zu beurteilen oder gar zu messen, inwieweit Klassen wiederverwendet werden können, sind Metriken im Moment ein vieldiskutiertes Thema. So wird teilweise im Usenet sehr kontrovers darüber debattiert, ob zum Beispiel

- die Länge des Sourcecodes,
- die Anzahl der features,
- die Anzahl der benutzten und benutzenden Klassen,
- die Tiefe der Vererbungshierarchie

eine Aussage darüber ermöglichen, wie hoch der Grad an Wiederverwendbarkeit ist.

Eine große Anzahl von Verweisen zu Metriken im Bereich der objektorientierten Softwareentwicklung findet sich in den cetus-links http://www.rhein-neckar.de/~cetus/software.html.

Abbildungsverzeichnis

Literaturverzeichnis

AIS77 Christopher Alexander, S. Ishikawa, M. Silverstein, M. Jacobsen, I. Fisksdahl-King, S. *Angel: A Pattern Language*. Oxford University Press, 1977.

Ale79 Christopher Alexander: *The Timeless Way of Building*. New York: Oxford University Press, 1979.

BCS92 Reinhard Budde, Marie-Luise Christ-Neumann, Karl-Heinz Sylla: „Tools and Materials: An Analysis and Design Metaphor.“ TOOL-7, *Technology of Object-Oriented Languages and Systems, Europe '92*. Edited by G. Heeg, B. Magnusson and B. Meyer. Prentice-Hall, 1992, S. 135-146.

BGK+95 Dirk Bäumer, Guido Gryczan, Rolf Knoll, Wolfgang Strunk, Heinz Züllighoven: „Objektorientierte Entwicklung anwendungsspezifischer Rahmenwerke“. *OBJEKTspektrum* Nr. 6, Nov./Dez. 1995, München: Verlag SIGS Conferences GmbH, S. 48-59.

BKK+92 Reinhard Budde, Karlheinz Kautz, Karin Kuhlenkamp, Heinz Züllighoven: *Prototyping.* Springer, 1992.

Boe88 Barry W. Boehm: „A Spiral Model of Software Development and Enhancement.“ *IEEE Computer* 21, May 1988, S. 61-72.

BZ90 Reinhard Budde, Heinz Züllighoven: *Software-Werkzeuge in einer Programmierwerkstatt*. München, Wien: R. Oldenbourg Verlag, 1990.

BZ92 Reinhard Budde, Heinz Züllighoven: „Software Tools in a Programming Workshop.“ *Software Development and Reality Construction*. Edited by Christiane Floyd, Heinz Züllighoven, Reinhard Budde and Reinhard Keil-Slawik. Berlin, Heidelberg: Springer-Verlag, 1992, S. 252-268.

Coa92 Peter Coad: „Object-Oriented Patterns.“ *Communications of the ACM* 35, Sept. 1992, S. 152-159.

Flo84 Christiane Floyd: „A Systematic Look At Prototyping“. *Approaches to Prototyping*. Edited by Reinhard Budde, Karin Kuhlenkamp, Lars Mathiassen and Heinz Züllighoven. Berlin, Heidelberg: Springer-Verlag, 1984, S. 1-18.

FT96 William Frakes, Carol Terry: „Software Reuse: Metrics and Models.“ *ACM Computing Surveys*, Vol. 28, No. 2, Juni 1996.

Gam92 Erich Gamma*: Objektorientierte Software-Entwicklung am Beispiel von ET++*. Berlin, Heidelberg: Springer-Verlag, 1992.

GHJ+93 Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: „Design Patterns: Abstraction and Reuse of Object-Oriented Design.“ ECOOP ‘93, LNCS 707, *Conference Proceedings*. Berlin, Heidelberg: Springer-Verlag, 1993, S. 406-431.

GHJ+95 Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides*: Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Massachusetts: Addison-Wesley, 1995.

Gil94 Thorsten Gildhoff: *Ein Prototyp eines Reuse-Tools für Eiffel*. Studienarbeit, Hamburg: Universität Hamburg, Fachbereich Informatik, Arbeitsbereich Softwaretechnik, 1994.

Gin95 Jens Ginzel: *Analyse des User-Interface-Toolkits EiffelVision im Hinblick auf seine Verwendbarkeit zur Werkzeugkonstruktion nach der Werkzeug- und Material-Metapher*. Studienarbeit, Hamburg: Universität Hamburg, Fachbereich Informatik, Arbeitsbereich Softwaretechnik, 1995.

Gry96 Guido Gryczan*: Prozeßmuster zur Unterstützung kooperativer Tätigkeit*. Wiesbaden: Deutscher Universitäts-Verlag, 1996

GZ92 Guido Gryczan, Heinz Züllighoven: „Objektorientierte Systementwicklung: Leitbild und Entwicklungsdokumente.“ *Informatik-Spektrum* 15, Oktober 1992, S. 264-272.

IFD81 International Federation for Documentation. *Principles of the universal decimal classification*. 1981.

ISE93 Interactive Software Engineering Inc.: *EiffelVision: An Introduction*. ISE Technical Report TR-EI-40/IE. Goleta, California, 1993.

ISE95 Interactive Software Engineering Inc.: *ISE Eiffel. The Environment*. ISE Technical Report TR-EI-39/IE. Goleta, California, 1995.

KGZ93 Klaus Kilberth, Guido Gryczan, Heinz Züllighoven*: Objektorientierte Anwendungsentwicklung*. Vieweg, 1993.

Mey90 Bertrand Meyer: *Objektorientierte Softwareentwicklung*. Hanser, 1990.

Mey91 Bertrand Meyer: *Eiffel: The Language.* Prentice Hall, 1991.

Mey94 Bertrand Meyer: *Reusable software: The Base object-oriented component libraries*. Prentice Hall, 1994.

PB96 Gustav Pomberger, Günther *Blaschek: Software Engineering: Prototyping und objektorientierte Software-Entwicklung*. München, Wien: Hanser Verlag, 1996.

PD85 Ruben Prieto-Diaz: *A Software Classification Scheme*. PhD thesis, Univesity of California, Irvine, 1985.

PDF87 Ruben Prieto-Diaz, Peter Freeman: „Classifying Software for Reusability.“ *IEEE Software*, S. 6-16, Januar 1987.

Pre94 Wolfgang Pree: „Meta Patterns - A Means for Capturing the Essentials of Reusable Object-Oriented Design.“ ECOOP ‘94, LNCS 821, *Object Oriented Programming*. Edited by Mario Tokoro and Remo Pareschi. Berlin, Heidelberg: Springer-Verlag, 1994, S. 150-160.

Ran57 S. R. Ranghanathan: *Prolegomena to Library Classification*. The Garden City Press Ltd., Letchworth, Hertfordshire, 1957.

Ran67 S. R. Ranghanathan: *Prolegomena to Library Classification*. Asia Publishing House, Bombay, India, 1967.

Rie95 Dirk Riehle: *Muster am Beispiel der Werkzeug und Material Metapher*. Schriftenreihe Softwaretechnik, Hamburg: Universität Hamburg, Fachbereich Informatik, Arbeitsbereich Softwaretechnik, 1995.

Roy70 W. W. Royce: „Managing the Development of Large Software Systems.“ *Proceedings of IEEE WESCON*. IEEE, S. 1-9, 1970. Reprinted in*: IEEE Tutorial on Software Engineering Project Managment*. Edited by R. H. Thayer. IEEE, 1988.

Rum95 James Rumbaugh: „OMT: The Object Model.“ *Journal of Object-Oriented Programming*, Januar 1995, S. 21-27.

SNiFF SNiFF+ Reference, S. 183ff.

ST92 Lars Sivert Sørumgård, Eirik Tryggeseth: *Classification, Search and Retrieval of Reusable Software Components*. Master degree thesis, Norwegian Institute of Technology, Trondheim, 1992.

WISR7 *Seventh Annual Workshop on Institutionalizing Software Reuse*. Proceedings. St. Charles, Illinois, August 28-30, 1995. ftp://gandalf.umcs.maine.edu/pub/WISR/wisr7/proceedings

WW94 Martina Wulf, Dirk Weske: *Konzepte zur Materialversorgung verteilter Werkzeugumgebungen am Beispiel der Anbindung einer objektorientierten Datenbank*. Studienarbeit, Hamburg: Universität Hamburg, Fachbereich Informatik, Arbeitsbereich Softwaretechnik, 1994.